A Quantitative Comparison of Parallel Computation Models

BEN H. H. JUURLINK
Delft University of Technology
and
HARRY A. G. WIJSHOFF
Leiden University

In recent years, a large number of parallel computation models have been proposed to replace the PRAM as the parallel computation model presented to the algorithm designer. Although mostly the theoretical justifications for these models are sound, and many algorithmic results were obtained through these models, little experimentation has been conducted to validate the effectiveness of these models for developing cost-effective algorithms and applications on existing hardware platforms. In this article a first attempt is made to perform a detailed experimental account on the preciseness of these models. To achieve this, three models (BSP, E-BSP, and BPRAM) were selected and validated on five parallel platforms (Cray T3E, Thinking Machines CM-5, Intel Paragon, MasPar MP-1, and Parsytec GCel). The work described in this article consists of three parts. First, the predictive capabilities of the models are investigated. Unlike previous experimental work, which mostly demonstrated a close match between the measured and predicted execution times, this article shows that there are several situations in which the models do not precisely predict the actual runtime behavior of an algorithm implementation. Second, a comparison between the models is provided in order to determine the model that induces the most efficient algorithms. Lastly, the performance achieved by the model-derived algorithms is compared with the performance attained by machine-specific algorithms in order to examine the effectiveness of deriving fast algorithms through the formalisms of the models.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessor); C.4 [**Computer Systems Organization**]: Performance of Systems—*modeling techniques*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms: Experimentation, Performance

This research was supported in part by DFG-SFB 376 “Massive Parallelität” and EU ESPRIT Long Term Research Project 20244 (ALCOM-IT). Ben H. H. Juurlink was affiliated with Leiden University and, subsequently, with Paderborn University when this work was performed. Preliminary versions of this article appeared in SPAA’96 and IPPS/SPDP’98.
Authors’ addresses: B. H. H. Juurlink, Laboratory of Computer Architecture and Digital Techniques, Department of Electrical Engineering, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands; H. A. G. Wijshoff, High Performance Computing Division, Department of Computer Science, Leiden University, P.O. Box 9512, 2300 RA Leiden, The Netherlands.

Additional Key Words and Phrases: Parallel computation models, performance evaluation

1. INTRODUCTION

Parallel algorithmic research has long been burdened by the lack of a standard model of parallel computation. Such a model should on the one hand be simple enough to disclose fundamental parallel algorithmic techniques. On the other hand, it should accurately reflect the cost of executing parallel programs on existing and foreseeable parallel architectures [Skillicorn 1991; Heywood and Ranka 1992]. In addition, the model should not be tied to a particular architecture, so that algorithms developed for it will be portable across platforms. We note that there are two aspects to portability. First, not all parallel platforms support the same programming language or message-passing interface. This problem seems to be resolved by widely available message-passing libraries like PVM [Geist et al. 1994] and MPI [The MPI Forum 1993]. However, a more fundamental problem is that many parallel programs are *architecture dependent*, meaning that they are customized for a particular architecture in order to achieve the highest performance. The result of this practice is that parallel programs cannot be ported across platforms without incurring a large performance penalty.

The PRAM [Fortune and Wyllie 1978] is the prevailing model in the theoretical community. Its shared-memory abstraction and the assumption that the processors operate fully synchronously make it a relatively easy model to use. The PRAM also provides a very simple cost model. Each step, regardless of whether it is a local computation step or an access to the shared memory (communication step), is assumed to take unit time. On many existing parallel architectures, however, communication is much more expensive than local computations, and synchronization is also very expensive. A large body of parallel algorithms has also been developed for theoretical network models, in which it assumed that it takes unit time to send a packet between adjacent processors. This does not seem to be a promising approach either, because it leads to nonportable programs.

For these reasons, several alternative models have been proposed that try to capture communication and synchronization cost without sacrificing too much of the PRAM's simplicity and generality. Although mostly the theoretical justifications for these models are sound, and many algorithmic results were obtained through these models, little experimentation has been conducted in order to examine the effectiveness of deriving fast algorithms through the formalisms of the models. This article tries to fill that gap by experimentally validating three of the proposed parallel computation models on five parallel platforms. Our investigation concentrates on three questions. First, do the models accurately predict the execution time of an algorithm implementation? Obviously, the ability to accurately predict the runtime behavior of a parallel program is a crucial property that a parallel computation model must possess, since it enables the algorithm developer to pick the fastest algorithm from a set of alternatives without having to implement them all. Second, how do the models compare with each other? In other words, we would like to determine the model that induces the fastest algorithms. Third, how do the model-derived algorithms compare with algorithms customized for the target architecture? It seems that the price to be paid for portability is performance, but what is the performance loss that can be expected? Indeed, there is little prospect that parallel software developers will adopt a new model if it does not induce algorithms that are nearly as efficient as programs customized for the target architecture.

To answer these questions we selected three problems (matrix multiplication, sorting, and all pairs shortest path), designed algorithms for them based on the precepts of the models, and implemented these algorithms on five parallel platforms: a Cray T3E, an Intel Paragon, a Thinking Machines CM-5, a MasPar MP-1, and a Parsytec GCel. The problems were picked for two reasons. First, they are a key component of many parallel applications. Second, they are relatively easy and well understood, so that a precise analysis under the various models is feasible. The hardware platforms were chosen because they represent a wide range of different realizations of parallel distributed-memory architectures. Because of the fact that the scalability properties of the parallel computation models are used throughout the article, we did not include any physically shared memory platforms into this study.

1.1 Related Work

The paper introducing the LogP model [Culler et al. 1993] presented execution times for an FFT algorithm implemented on the CM-5. The measured times showed a close match with the predicted execution times, provided the processors were synchronized periodically using the hardware barrier available on the CM-5. Several LogP sorting algorithms were analyzed analytically and empirically in Culler et al. [1994a].

The BSP model was analyzed experimentally in Goudreau et al. [1996]. We have the following remarks about the experimental data presented there. First, the authors noted that the BSP cost model should not be expected to predict precise running times, since this is only possible for fairly simple algorithms such as sorting and broadcast. We remark that the algorithms we experimented with (purposely) belong to the class of algorithms for which a precise analysis seems feasible. Second, as was acknowledged in Goudreau et al. [1996], the results were collected on platforms with a small number of processors (at most 16). As will be shown in this article, several phenomena that might affect the accuracy of the BSP cost model do not become visible until at least a moderate number of processors are employed. Finally, as was also mentioned in Goudreau et al. [1996], for most applications considered there the communication overhead was only a small component of the total execution time. We are interested in cases where the communication overhead accounts for a significant part of the overall running time, since it is the only cost captured in detail by most parallel computation models.

Skillicorn et al. [1997] measured the BSP parameters belonging to several platforms. It is unclear to us whether these parameters were obtained by measuring the time required for sending large messages or for short messages. Because the BSP model does not penalize fine-grain communication, we used a packet size equal to the word size of the machine.

Shumaker and Goudreau [1997] described an implementation of a BSP library on the MasPar MP-2. Their results our difficult to compare to ours, because they normalized the BSP parameters to the time it takes to multiply two 32-bit floating-point numbers.

1.2 Organization

This article is organized as follows. In Section 2 the parallel computation models considered in this article are described. After that, in Section 3, the experimental platforms are described as well as the experiments conducted to determine the model parameters belonging to each platform. The implemented algorithms are described and analyzed in Section 4. Thereupon, in Section 5, the predictive capabilities of the models are investigated. A comparison between the models is provided in Section 6, and in Section 7 the performance achieved by the model-derived algorithms is compared with the performance attained by machine-specific algorithms in order to validate the efficiency of the model-derived algorithms. Conclusions are given in Section 8.

2. MODEL DESCRIPTIONS

In this section, the parallel computation models considered in this article are briefly reviewed.

2.1 BSP

The Bulk-Synchronous Parallel (BSP) model proposed by Valiant [1990] is a model that has received considerable attention over the last few years. It consists of the following attributes: (1) a set of p processors with local memories, (2) a communication medium (router) that can deliver messages between any pair of processors, and (3) a mechanism to synchronize the processors in a barrier style. BSP computations are organized in a series of phases called *supersteps*. In a superstep a processor can perform local operations and send messages to other processors. Supersteps are separated by a barrier synchronization, after which it is assured that all messages have reached their destinations. This programming model is different from the one provided by most message-passing libraries like PVM [Geist et al. 1994] or MPI [The MPI Forum 1993]. These libraries are based on pairwise sends and receives, whereas in the BSP model explicit receives are unnecessary. Instead, a barrier synchronization signifies the end of all communication operations.

The cost of a BSP computation depends on the following parameters that characterize the communication capabilities of the architecture: the combined latency/synchronization cost L and the computational-to-communication bandwidth ratio g, which 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 one time unit. Central to the BSP model is the concept of an *h-relation*: a communication pattern in which each processor sends and receives at most h messages. The parameters g and L are such that an arbitrary h-relation followed by a barrier synchronization can be performed in $g \cdot h + L$ time. The cost of a superstep is therefore given by $w + g \cdot h + L$, where w is the maximum amount of local work performed by any processor during the superstep, and h is the maximum number of messages sent or received by any processor. McColl [1993] also includes a speed parameter s which is defined as the number of operations performed per time unit. We decided not to include this parameter because it depends very much on the application domain. We also do not normalize g and L with resect to processor speed, as is done traditionally, but use actual times (in μsec.) instead.

2.2 BPRAM

In the BSP model it is assumed that all messages have a fixed short size (essentially the word size of the machine). However, it is well known that on many parallel architectures there is a large startup cost associated with transmitting a message. The Message-Passing Block PRAM (BPRAM, for short) [Aggarwal et al. 1989] is a model in which block transfers are rewarded. Briefly, a BPRAM consists of p processors, each provided with a local memory of unbounded size, that communicate with each other by exchanging messages. A processor can send and receive only one message in one communication step (i.e., every communication pattern should correspond to a (partial) permutation), and a message of length m is transferred in time $m + \ell$, where ℓ is the startup cost of a message transmission. Furthermore, the model is synchronous, meaning that every processor waits for the longest block transfer to complete before it proceeds to the next step. Because generally computation and communication are not equally expensive, we model the time to send an m-byte message by the formula $\sigma \cdot m + \ell$, where σ is the transfer time per byte.

2.3 E-BSP

The BSP as well as the BPRAM assume that the time needed for communication is independent of the network load. The BSP model conservatively assumes that all h-relations are *full* h-relations in which *all* processors send and receive exactly h messages. Likewise, in the BPRAM it is assumed that sending one m-byte message between two processors takes the same amount of time as a full block permutation in which all processors send and receive an m-byte message. The E-BSP model [Juurlink and Wijshoff 1996c] extends the basic BSP model to deal with unbalanced communication patterns, i.e., communication patterns in which the processors send or receive different amounts of data.

Like BSP, the E-BSP model is strongly motivated by various routing results. Furthermore, the cost function supplied by E-BSP generally is a nonlinear function that strongly depends on the network topology. We therefore employ the following simplified version of E-BSP in this article. Let an (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*). Further, let an h-item scatter operation be a communication pattern in which one processor scatters h data items evenly among the processors so that each processor receives h / p items (arguably a very unbalanced h-relation), and let g' be such that an arbitrary h-item scatter operation takes $g' \cdot h + L$ time. In the E-BSP model, every communication pattern is treated as an (V, h)-relation with cost $\max\{g \cdot V/p, g' \cdot h\} + L$, where g and L are identical to BSP's parameters. Thus, the parameter g' essentially captures node-to-network bandwidth, whereas g captures intranetwork bandwidth. Note that this cost model essentially differentiates between communication patterns that are insensitive to the bisection bandwidth and those that are not.

2.4 Other Models

Of course, there are many other models which are not considered in this article. In this section, we briefly mention a few of them.

A model which is similar to BSP is the LogP model [Culler et al. 1993]. There are three differences between the two models. First, LogP is completely asynchronous whereas BSP can be classified as "semisynchronous." Second, LogP has an extra overhead parameter that represents the time a processor is engaged in transmitting a message. Third, it is assumed that the communication network has a finite capacity such that only a limited number of messages can be in transit to or from any processor at any time. LogP is not considered in this article for the following reasons. The algorithms described in Section 4 do not attempt to overlap computation and communication. The overhead parameter can therefore be ignored. Furthermore, the BSP library implemented on the Paragon and the Gcel sends messages in a staggered order, whereas the implementation of BSPlib on the T3E uses a Latin square [Skillicorn et al. 1997]. For the algorithms we experimented with, this technique was sufficient to ignore the capacity constraint of LogP. It will be shown, however, that in one case the capacity constraint helps to explain the differences between predicted and observed running times.

The models considered here also do not allow topological locality (network proximity) to be exploited. Such models include the H-PRAM [Heywood and Ranka 1992], the Y-PRAM [de la Torre and Kruskal 1991], and the original E-BSP model. These type of models are not considered for the following reasons. First, on the MasPar we exclusively used the global router for communication which is an indirect network that does not allow network proximity to be exploited. Second, the processor numbering on the CM-5 was proprietary. Furthermore, the configurations of the T3E and the Paragon were too small to notice a significant difference between the time required for routing a random communication pattern and a more local routing distribution. On the GCel, however, the model-derived matrix multiplication algorithms will be compared with an algorithm that requires only near-neighbor communication in order to investigate the price of giving up network proximity.

3. EXPERIMENTAL PLATFORMS

In this section, the experimental platforms are described, and the values of the model parameters are determined. In an earlier paper, a limited study for a T800 platform was conducted [Juurlink and Wijshoff 1993].

3.1 Intel Paragon

The Intel Paragon [Groscup 1992] is a MIMD computer with a mesh topology. Each node is a shared-memory multiprocessor with two (sometimes three) 50MHz i860XP microprocessors, connected by a 400MB/sec., cache-coherent memory bus. One processor, called the *message processor*, is dedicated to communication, so that the *compute processor* is released from message-passing operations. All experiments were conducted on an 8×8 configuration.

In order to implement the algorithms in a BSP- and BPRAM-like fashion, we implemented two small communication libraries on top of the native NX message-passing library. The implemented BSP library is similar to the recently released BSPlib library [Hill et al. 1997]. Basically, it provides functions for Direct Remote Memory Access (DRMA) (`bsp_store`), Bulk-Synchronous Message Passing (BSMP) (`bsp_send`), and barrier synchronization (`bsp_sync`). In previous experiments on the Paragon [Juurlink 1998], we determined that the startup cost of a message transmission is roughly 146 μsec. whereas the transfer time per byte is about 11.5 nsec. For this reason, the implementation of the BSP library postpones all communication until the end of the superstep (by writing the messages in an output buffer associated with the destination processor), and combines all packets destined for the same processor into a single message. It has been shown in Skillicorn et al. [1997] that this technique reduces the importance of sending large messages.

Figure 1 plots the times required for routing full h-relations as well as the times needed for performing h-item scatter operations on the Paragon. The minimum and maximum measured times are also shown using vertical error bars. Because the BSP cost model does not penalize fine-grain

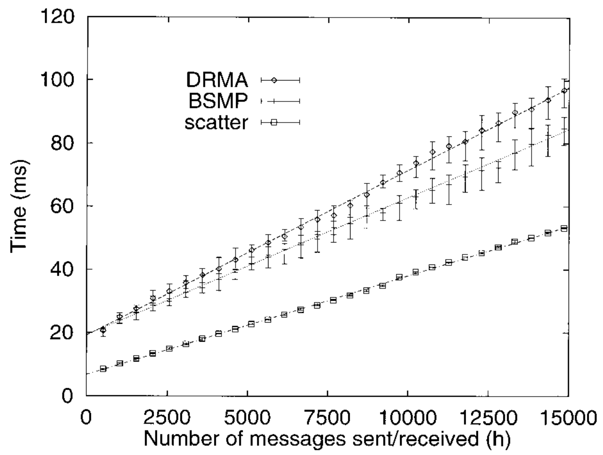

Fig. 1. Time required for routing full h-relations and for scatter operations on the Paragon.

communication, we used a packet size equal to the word size of the machine (four bytes). The curve labeled "DRMA" shows the time needed when the DRMA function `bsp_store` is used, whereas the curve labeled "BSMP" plots the time when `bsp_send` is used. The scatter operation was implemented using `bsp_store`. By fitting straight lines to the measured data points, we determined that the BSP parameters belonging to the Paragon are given by $g = 5.42$ μsec. and $L = 1.95 \times 10^4$ μsec., and that E-BSP's additional parameter is given by $g' = 3.13$ μsec. Two important remarks concerning the experimental data need to be made. First, the latency/synchronization cost L is relatively high. This is because the startup cost of a message transmission is relatively high on this platform, and the cost of $p - 1$ startups are folded into L. Second, the bandwidth parameter g is not limited by the network capacity. Instead, the bulk of the cost of sending a packet is due to copying the packet into the output buffer and out of the input buffer. This is also the main reason why an h-item scatter operation is about a factor of 1.7 cheaper than a full h-relation, since in this pattern every processor receives only $\lceil h / p \rceil$ instead of h packets.

The implemented BPRAM library provides only one communication primitive that sends a message to a specified destination processor and receives a message from an arbitrary processor. In keeping with the BPRAM semantics, all processors must call this function simultaneously in order to prevent deadlock, and the communication pattern must correspond to a (partial) permutation. Furthermore, a global synchronization is executed after every communication step for two reasons. First, the communication primitive does not use tags to distinguish messages. Therefore, in order to be sure that the program is semantically correct, a global synchronization is needed. Second, the BPRAM cost model assumes that the time needed for a communication step depends linearly on the size of the largest message.

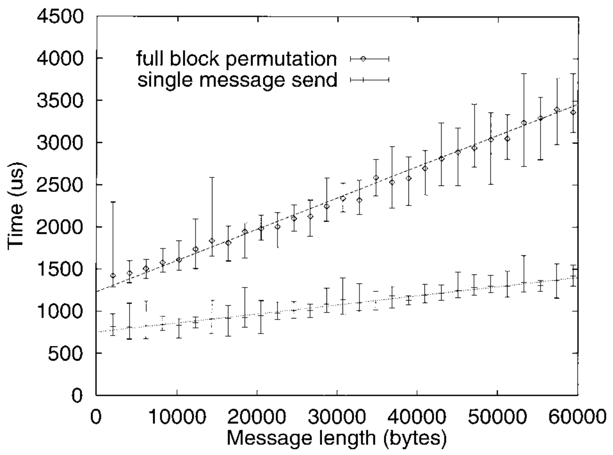

Fig. 2. Time needed for full block permutations and for sending one message on the Paragon.

Figure 2 plots the time needed for performing full block permutations on the Paragon as a function of the message length. It also shows the time needed for a communication pattern that does not saturate the network: a "single message send" in which one processor sends an m-byte message to another processor. The measurements indicate that the BPRAM parameters belonging to the Paragon are given by $\sigma \approx 3.72 \times 10^{-2}$ μsec. and $\ell \approx 1.23 \times 10^{3}$ μsec. Furthermore, a single message send is a factor of 3.44 cheaper than a full block permutation. We note that this is only partially due to bisection bandwidth limitations, since when the two processors send an m-byte message to each other simultaneously, the communication time approximately doubles. Thus, it appears that the links connecting the processors to the routing network are only half-duplex. The reader is referred to Juurlink [1998] for a more complete explanation of the experimental data on the Paragon. The BSP, E-BSP, and BPRAM parameters belonging to the Paragon are summarized in Table I.

3.2 Parsytec GCel

The second experimental platform is a 64-processor Parsytec GCel [Langhammer 1992], an 8×8 mesh-connected MIMD computer. Every processor is a T805 transputer, running at 30MHz, with 4MB of RAM. The BSP and BPRAM library described in the previous section were also implemented on the GCel on top of the native message-passing library Parix.

Figure 3 plots the time required for routing randomly generated full h-relations on the GCel. It can be seen that the time needed for routing h-relations grows linearly with h up to a certain point. After that, it stays approximately constant, and finally it grows linearly with h again, but with a smaller slope than before. This behavior is due to the Parix message-passing functions `SendNode/RecvNode` on top of which the BSP library was implemented. For messages up to 1KB, Parix uses a store-and-forward

Table I. Summary of All Platforms and Parameters. Summary of the BSP (using DRMA), E-BSP (also using DRMA), and BPRAM parameters belonging to each platform. All parameters are given in μsec. For the GCel, only the g and g' parameters obtained for routing large h-relations and scatter operations are shown. The ratio $g/(w \cdot \sigma)$ (on the MasPar under the MP-BSP model the ratio $(g + L)/(w \cdot \sigma)$) is an indication of the gain that can be obtained by grouping data into large messages, where w is the word size of the machine (8 on the T3E and 4 on all other platforms).

		(MP-)BSP		E-BSP	BPRAM		
Platform	p	g	L	g'	σ	ℓ	$g/(w \cdot \sigma)$
Paragon	64	5.42	1.95×10^4	3.13	3.72×10^{-2}	1.23×10^3	36.4
GCel	64	118.5	2.1×10^5	47.8	8.5	1.6×10^4	3.5
T3E	8	1.10	36.9	0.69	4.94×10^{-3}	11.7	27.8
CM-5	64	9.1	45	6.3	2.75×10^{-1}	75	8.3
MasPar	1024	32.2	1.4×10^3	—	1.07×10^2	6.3×10^2	5.7

mechanism for routing messages through the network, whereas for larger messages, a temporary virtual link is established. In order to avoid that this effect disturbs the accuracy of the predictions, we approximate the time needed for routing h-relations on the GCel by the piecewise linear function shown in Figure 3. For example, when `bsp_store` is used, the following function is used to model the time needed for routing h-relations:

$$T(h) = \begin{cases} 172.7 \cdot h + 2.6 \times 10^5 \ \mu\text{sec.} & \text{for } h \leq 4500 \\ 1.04 \times 10^6 & \text{for } 4500 < h < 7000 \\ 118.5 \cdot h + 2.1 \times 10^5 \ \mu\text{sec.} & \text{for } h \geq 7000. \end{cases}$$

The value of g' on this architecture is given by $g' = 87.9$ μsec. for $h \leq 5300$ and $g' = 47.8$ μsec. otherwise.

Three other important conclusions can be drawn from Figure 3. First, the cost of routing h-relations using `bsp_send` is about two-thirds the cost of performing h-relations using `bsp_store`. This is to be expected, since 12 bytes of data (target address, packet length, and packet data) are sent for every 4-byte data word when `bsp_store` is used, whereas 8 bytes of data (packet length and packet data) are sent when `bsp_send` is used. Second, the packet combining technique has reduced the value of g by a factor of about 13.8 compared to a previous implementation of the BSP library that sent messages as soon as they were generated [Juurlink and Wijshoff 1996a]. The value of L, however, has increased by a factor of about 5.7. This is due to the fact that L now includes the startup cost of $p - 1$ message transmissions. Third, a scatter operation is about a factor of 2.5 cheaper than a full h-relation, thereby showing the effects of unbalanced communication.

The times needed for routing full block permutations on the GCel are shown in Figure 4. By fitting a straight line to the measured data points, we determined that the BPRAM parameters belonging to the GCel are given by $\sigma \approx 8.5$ μsec. and $\ell \approx 1.6 \times 10^4$ μsec. Again, it can be observed

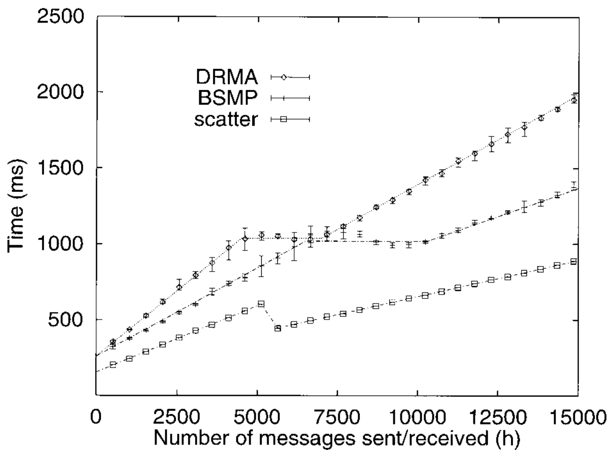

Fig. 3. Time required for routing full h-relations and for performing scatter operations on the GCel.

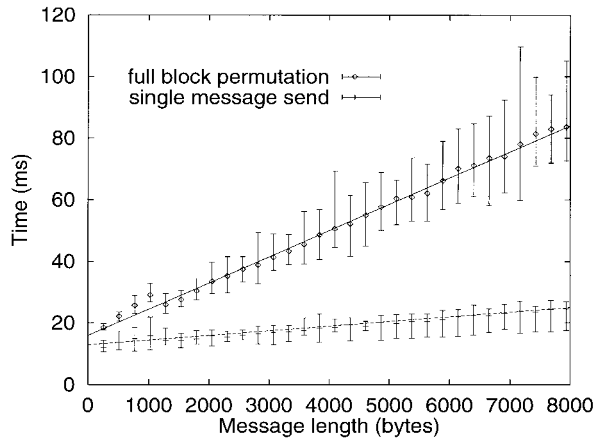

Fig. 4. Time required for routing full block permutations and for sending one message on the GCel.

that for short messages the execution time tends to increase more rapidly than for long messages. Figure 4 also shows the time needed for a single message send. Asymptotically, this communication pattern is a factor of about 5.7 cheaper than a full block permutation.

3.3 Cray T3E

The Cray Research T3E [Oberlin et al. 1996] is a MIMD parallel computer with a 3D torus interconnect network. Each node consist of a DEC Alpha 21164 RISC microprocessor, multiple banks of DRAM, and a network router. All experiments were conducted on an eight-processor configuration. The BSP programs for the T3E were implemented using the BSPlib library [Hill et al. 1997], and the BPRAM library described in Section 3 was

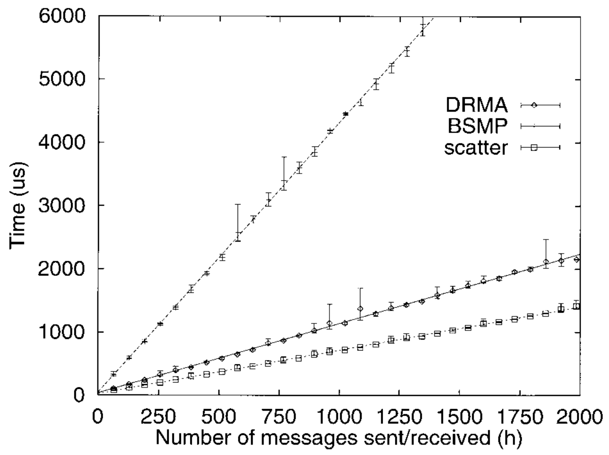

Fig. 5. Time required for routing full h-relations and for performing scatter operations on the T3E.

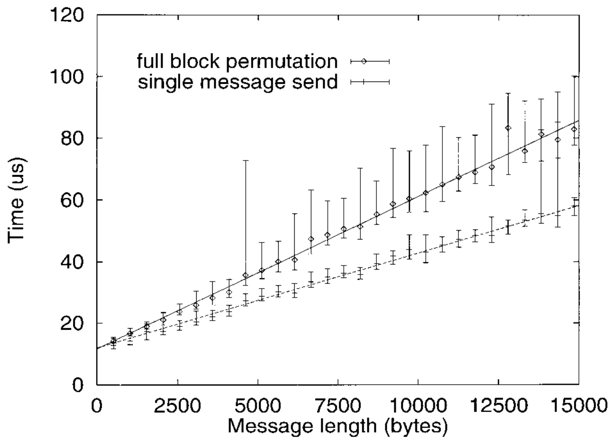

Fig. 6. Time needed for routing full block permutations and for sending one message on the T3E.

also implemented on the T3E. Both of these libraries are implemented on top of the native, explicit shared-memory (SHMEM) communication functions shmem_put and shmem_get.

Figure 5 shows the fine-grain communication performance of the T3E. As usual, the times needed for routing full h-relations using the BSPlib DRMA function bsp_hpput and the BSMP function bsp_send are depicted, as well as the times needed for performing scatter operations. It is conspicuous that on this platform bsp_send is almost a factor of 3.9 slower than bsp_hpput. We believe that this is due to the fact that packets that are sent using bsp_send are buffered on the source and destination, whereas no buffering is needed for bsp_hpput, since it can be implemented directly on top of shmem_put. It can also be seen that scatter operations are about a

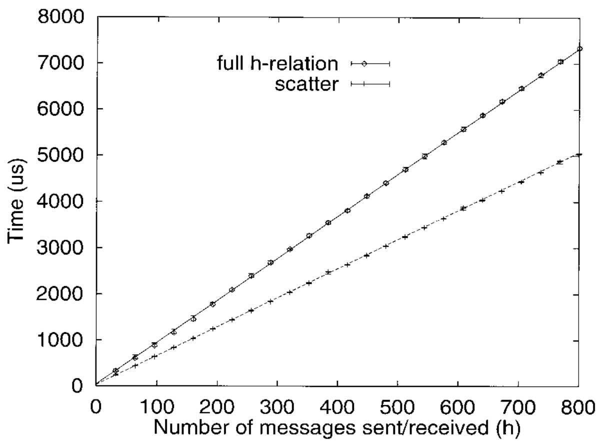

Fig. 7. Time required for routing full h-relations and for performing scatter operations on the CM-5.

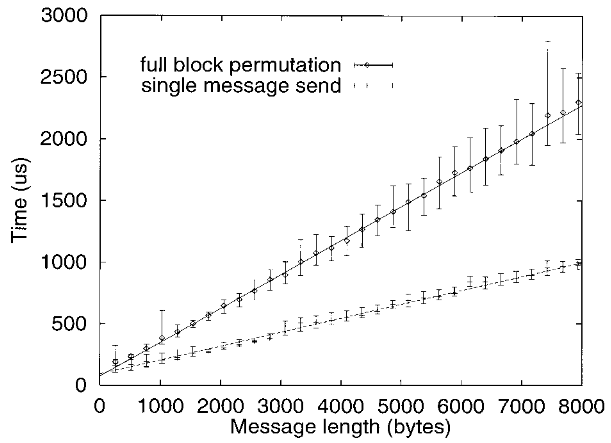

Fig. 8. Time required for routing full block permutations and for sending one message on the CM-5.

factor of 1.6 cheaper than full h-relations, but this should again be attributed to send and receive overheads and not to congestion in the network.

Figure 6 shows the bulk message-passing performance of the T3E. From the measured data points, we determined that the BPRAM parameters belonging to the T3E are given by $\sigma \approx 4.94 \times 10^{-3}$ μsec. and $\ell \approx 11.7$ μsec. Furthermore, a single message send is about a factor of 1.6 cheaper than a full block permutation.

3.4 CM-5

The Thinking Machines CM-5 [Leiserson et al. 1992] is a MIMD parallel computer with a fat-tree interconnection network. Each node contains a

32MHz Sparc Cypress processor, 32MB of local memory, a 64KB direct-mapped cache, and a network interface chip. In addition to the data network, a broadcast/scan/prefix control network is present which can be used for fast barrier synchronization. Our programs for the CM-5 are written in Split-C [Krishnamurthy et al. 1993]; a parallel extension of the C programming language that follows an SPMD programming model and provides a shared global address space.

Figure 7 plots the times needed for routing full h-relations (using the Split-C *store* operation) on the CM-5, and it verifies that the routing time is well approximated by a straight line with slope $g \approx 9.1$ μsec. and offset $L \approx 45$ μsec. E-BSP's additional parameter is given by $g' = 6.3$ μsec. Thus, h-item scatter operations are slightly cheaper (about 30%) than full h-relations, but this should again be attributed to send and receive overheads.

The BPRAM parameters belonging to the CM-5 are given by $\sigma \approx 0.275$ μsec. and $\ell \approx 75$ μsec. (see Figure 8). On this platform, a single message send takes up to a factor of 2.4 less time than a full block permutation.

3.5 MasPar MP-1

The MasPar MP-1 [Blank 1990; Nickolls 1990] is a massively parallel SIMD architecture. The system used for the experimentation consists of 1024 Processor Elements (PEs), each of which is a four-bit, 80 nsec. load/store arithmetic processor with 64KB of memory. A MasPar has two communication networks: the Xnet and the global router. The Xnet is a two-dimensional toroid, in which each PE is connected directly to eight neighbors. With Xnet communication, all PEs must be communicating in the same direction and at the same distance. The global router is a multistage interconnection network with a predefined greedy routing scheme. We have worked exclusively with router communication, because it is faster than the Xnet for irregular communication patterns [Shumaker and Goudreau 1997]. The programs for the MasPar were written in the MasPar Programming Language (MPL): a data-parallel extension of C.

BSP does not seem to be a good model for the MasPar architecture. Most importantly, the BSP model assumes that messages can be pipelined, but the PEs of the MasPar can have at most one outstanding message. The BSP model and the MasPar architecture are not completely incompatible, however. For example, it is possible to buffer the messages and to invoke a routing algorithm at the end of a superstep, as was done in Shumaker and Goudreau [1997]. However, this requires substantial memory space, which is a severe limitation given that every PE has only 64KB of memory. We therefore define the MP-BSP model: a small variation of BSP that reflects this architecture more accurately.

The MP-BSP model is a synchronous model in which the processors communicate by writing into the local memory of some other processor. Each step is either a computation step or a communication step:

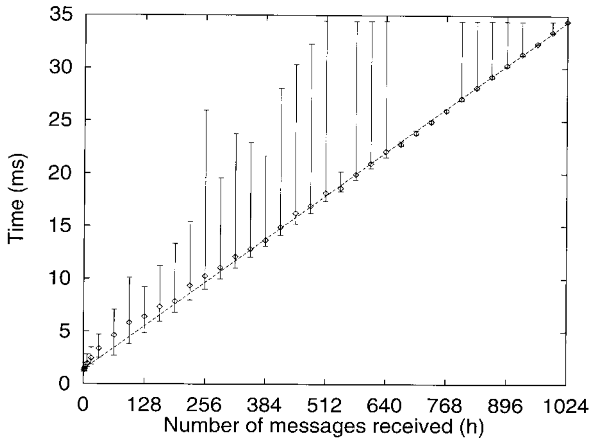

Fig. 9. Time required for routing (1, h)-relations on the MasPar.

(1) In a *computation step*, each processor P_i performs the same operation on data present in its local memory.

(2) In a *communication step*, each processor P_i writes one data item into the local memory of some other processor.

Let h_i be the number of processors accessing the local memory of processor P_i during a communication step. The cost of this step will be modeled by the formula $L + g \cdot \max_i h_i$. Thus, every communication step corresponds to a (1, h)-relation, in which each processor sends at most one message and receives at most h messages.

In order to determine the MP-BSP parameters belonging to the MasPar, the following experiment was conducted. A set of $\lceil P/h \rceil$ destinations is picked at random. Thereupon, we measured the time taken by a communication step in which $\lfloor P/h \rfloor$ PEs receive h messages, while the remaining destination (if any) receives $P - h \cdot \lfloor P/h \rfloor < h$ messages. The results of this experiment are shown in Figure 9.

Ideally, the data points would form a straight line with slope g and offset L, but the observed behavior is not completely linear. Thus, by charging $g \cdot h + L$ time for routing (1, h)-relations, an error will be introduced. The large variation in the measurements is due to the limitation that there is only one router channel available for each cluster of 16 PEs. If more than one PE in a cluster needs to receive a message, then the messages are serialized. This is clearly shown in Figure 10, which depicts the distribution of the times needed for routing (1,128)-relations over 1000 experiments. Thus, in this experiment there are $P/h = 8$ PEs, each receiving $h = 128$ messages. Let u denote the maximum number of destinations that fall in the same cluster. When $u = 1$ (the destinations are spread evenly among the clusters), a (1,128)-relation takes between 4.8 and 6.2

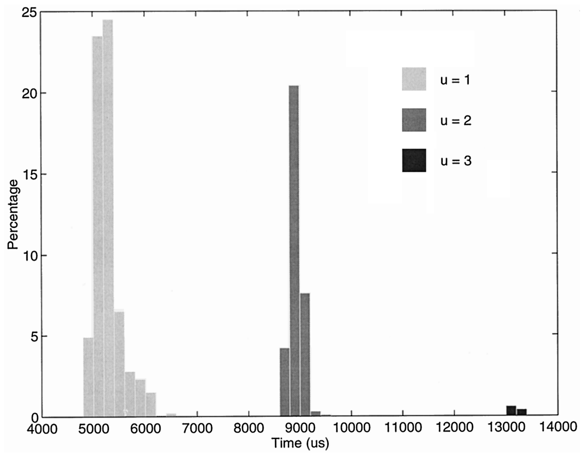

Fig. 10. Distribution of the measured running times for $h = 128$.

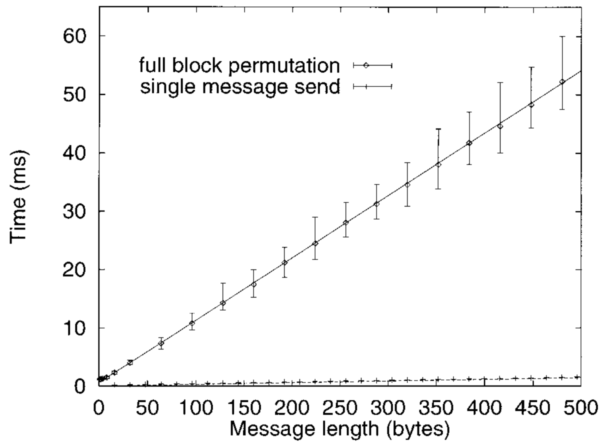

Fig. 11. Time required for performing full block permutations and for sending one message on the MasPar.

msec. This occurred in the majority (about 66%) of the experiments conducted. In 33% of the experiments, it took 8.6–9.2 msec. to perform a (1,128)-relation, and in all these cases u was equal to 2. In less than 1% of the experiments, the maximum number of destinations that fell in the same cluster was $u = 3$, and in all these cases the running time was approximately 13.1 msec.

Figure 11 plots the time required for routing full block permutations on the MasPar as well as the time needed for a single message send. From the measurements we determined that the BPRAM parameters belonging to the MasPar are given by $\sigma \approx 107$ μsec. and $\ell \approx 6.3 \times 10^2$ μsec. Furthermore, asymptotically a single message send is a factor of about 37.3 cheaper than a full block permutation. The MP-BSP and BPRAM parameters belonging to the MasPar are summarized in Table I.

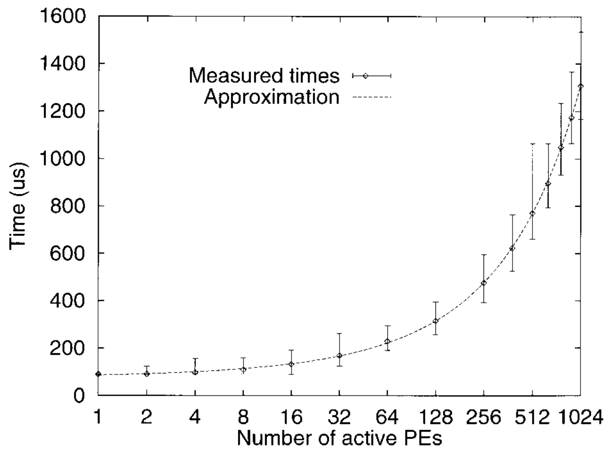

Fig. 12. Time needed for routing partial permutations as a function of the number of active PEs.

We also used a small variation of E-BSP for the MasPar, which will be denoted by MP-E-BSP. In this model, the time needed for routing partial permutations will be modeled as a function $T_{\mathrm{unb}}(p')$ of the number of PEs p' which are active during the communication step. An approximation for this function was determined by conducting the following experiment. A set of p' sending processors $P_{s_1}, P_{s_2}, \ldots, P_{s_{p'}}$ is picked at random, as well as a set of p' receiving processors $P_{r_1}, P_{r_2}, \ldots, P_{r_{p'}}$. We then measured the time required for performing a communication step in which processor P_{s_i} sends a message to processor $P_{r_i}(1 \leq i \leq p')$. The results of these experiments are shown in Figure 12, where a logarithmic scale is used for the x-axis. It can be seen that the communication time depends heavily on the number of active PEs. For example, with 32 active PEs, a partial permutation takes only about 13% of the time required for routing full permutations. By performing a second order polynomial fit, we found that

$$T_{\mathrm{unb}}(p') = 0.84 \cdot p' + 11.8 \cdot \sqrt{p'} + 73.3\ \mu\mathrm{sec}. \tag{1}$$

yields a good approximation. This approximation is also shown in Figure 12.

4. ALGORITHM DESCRIPTIONS

In order to investigate the predictive capabilities of the models, the following problems were selected: matrix multiplication, sorting, and all pairs shortest path. We picked these problems for various reasons. First, they are a key component of many important parallel processing applications. For example, matrix multiplication is present in almost all dense linear algebra computations, and sorting is used in a variety of applications such as query processing and polygon rendering. Furthermore, the problems are relatively easy and well understood, so that a precise analysis under the various models is feasible.

In this section, the implemented algorithms are described and analyzed. Only the communication cost is determined precisely, because this is the only cost captured in detail by the parallel computation models considered. The local computation time is always determined empirically. The BSP and the MP-BSP cost of an algorithm will be written as $W + g \cdot H + L \cdot S$, where W is the (measured) local computation time, $H = \Sigma_{i=1}^{S} h_i$, h_i is the maximum number of messages sent or received by any processor during superstep i, and S is the number of supersteps. Likewise, the BPRAM complexity of an algorithm will be written as $W + \sigma \cdot w \cdot M + \ell \cdot R$, where $M = \Sigma_{i=1}^{R} m_i$, m_i is the maximum length of a message sent or received by any processor during the ith communication step, and R is the number of communication steps. The analysis under the E-BSP model is somewhat more complex and will be dealt with differently.

4.1 Matrix Multiplication

We implemented the following matrix multiplication algorithm, which follows a strategy similar to the one described in Aggarwal et al. [1990], and Hong and Kung [1981] and which was adapted also for the BSP model in McColl [1995]. Lower-bound proofs [Aggarwal et al. 1990; Hong and Kung 1981] show that this algorithm is optimal under the BSP model (for any algorithm that only uses the semiring operations $(+\ ,\times)$). The algorithm uses $p = q^3$ processors. It is convenient to think of the processors as being arranged in a $q \times q \times q$ cube, i.e., let the processors be designated by $P_{i,j,k}$ for $0 \le i, j, k < q$. The input matrices A and B as well as the output matrix C are partitioned into q^2 square submatrices A_{ij}, B_{ij}, and $C_{ij}(0 \le i, j < q)$ of size $n/q \times n/q$ each. It will be assumed that, initially, processor $P_{i,j,0}$ holds the first n/q^2 rows of A_{ij} and B_{ij}, processor $P_{i,j,1}$ contains the second n/q^2 rows of A_{ij} and B_{ij}, and so on.

The BSP algorithm consists of four supersteps. In the first superstep, each processor $P_{i,j,k}$ receives the elements belonging to A_{ij} and B_{jk}. The BSP cost of this superstep is $2 \cdot g \cdot n^2/q^2 + L$. In the second superstep, every processor $P_{i,j,k}$ computes $C_{ijk} = A_{ij} \cdot B_{jk}$. This takes $O(n^3/p)$ local computation time. Each processor is responsible for computing an $n/q^2 \times n/q$ submatrix of the output matrix C. In the third superstep, each processor $P_{i,j,k}$ sends every element of C_{ijk} to the processor responsible for computing the corresponding element in C. The communication cost of this superstep is given by $g \cdot n^2/q^2 + L$. Finally, every processor adds q submatrices of size $n/q^2 \times n/q$, which takes $O(n^2/q^2)$ local computation time. The total BSP cost of this algorithm is given in Table II. An extra synchronization step may be required after the second superstep if communication is unbuffered. In this case, the synchronization cost is given by $3 \cdot L$.

Table II. (MP-)BSP and BPRAM Cost of Matrix Multiplication

BSP	BPRAM	MP-BSP
$W = \Theta(n^3/p)$	$W = \Theta(n^3/p)$	$W = \Theta(n^3/p)$
$H = 3 \cdot n^2/p^{2/3}$	$M = 3 \cdot n^2/p^{2/3}$	$H = 3 \cdot n^2/p^{2/3}$
S = 2 or 3	$R = 3 \cdot p^{1/3}$	$S = 3 \cdot n^2/p^{2/3}$

Under the MP-BSP model, care must be taken to avoid concurrent writes to the same memory module. To achieve this, the communication is staggered so that processor $P_{i,j,k}$ first sends data to processor $P_{i,j,(k+1) \bmod q}$, then to $P_{i,j,(k+2) \bmod q}$, and so on. Table II shows the total MP-BSP cost of matrix multiplication.

The algorithm can be easily restructured to use blocks of size n^2/p for data transfer. Note that the ability to use blocks of this size depends on the initial distribution of the matrices. If the initial distribution is different, an extra communication phase bringing the data in the desired layout is required. In the BSP model this is not an issue; the precise initial distribution is irrelevant as long as the data are distributed evenly among the processors. The total BPRAM cost of matrix multiplication is also given in Table II.

4.1.1 *Optimizing Local Computation.* Local computations remain unspecified in the models. It is important, however, to optimize the local matrix multiply carefully in order to obtain competitive results. On the MasPar and the GCel, an optimized blocked innerproduct algorithm is used that keeps parts of the output matrix C in registers. On the CM-5, the local matrix multiply is designed to pay careful attention to the local cache size and has a kernel written in assembly.[1] On the Paragon and the T3E, we used the BLAS 3 subroutine `sgemm()`.

4.2 Bitonic Sort

The sorting problem is defined as follows. Given n keys, initially distributed evenly among the processors, rearrange them so that every key in processor P_i is less than or equal to every key in processor P_{i+1}, for $0 \leq i < p - 1$. In addition, the list of keys within each processor needs to be sorted. We implemented two parallel sorting algorithms. One is based on Batcher's bitonic sort [Batcher 1968]. The other is a variation of sample sort as described, for example, in Blelloch et al. [1991] and Gerbessiotis and Valiant [1992]. Several papers on implementations of parallel sorting algorithms have reported that bitonic sort is the fastest algorithm when the number of keys per processor is small, whereas sample sort is the most efficient algorithm when the number of keys per processor is large.

Briefly, the implemented variation of bitonic sort works as follows. First, each processor sorts the set of n/p keys it contains locally. After that, the

[1]The local matrix multiply is written by Culler et al.

Table III. (MP-)BSP and BPRAM Cost of Bitonic Sort

BSP	$W = \Theta(T_{\text{local-sort}}(n/p) + 0.5 \cdot \log p \cdot (\log p + 1) \cdot n/p)$ $H = 0.5 \cdot \log p \cdot (\log p + 1) \cdot n/p$ $S = 0.5 \cdot \log p \cdot (\log p + 1)$
BPRAM	$W = \Theta(T_{\text{local-sort}}(n/p) + 0.5 \cdot \log p \cdot (\log p + 1) \cdot n/p)$ $M = 0.5 \cdot \log p \cdot (\log p + 1) \cdot n/p$ $R = 0.5 \cdot \log p \cdot (\log p + 1)$
MP-BSP	$W = \Theta(T_{\text{local-sort}}(n/p) + 0.5 \cdot \log p \cdot (\log p + 1) \cdot n/p)$ $H = 0.5 \cdot \log p \cdot (\log p + 1) \cdot n/p$ $S = 0.5 \cdot \log p \cdot (\log p + 1) \cdot n/p$

algorithm executes $\log p$ merge stages, where stage $s (1 \le s \le \log p)$ consists of s merge steps. In step $j (0 \le j < s)$ of stage s, each processor P_i sends the list of n/p keys in its possession to the processor whose ID is obtained by complementing the jth bit of i. Thereafter, every processor executes a linear-time sequential merging procedure and keeps either the lower half of the merged list or the upper half. Bitonic sort can directly be adapted to utilize block transfers. The BSP, BPRAM, and MP-BSP complexities of bitonic sort are summarized in Table III.

There are many variations of bitonic sort. For $n \ge p^2$, a variation that requires only two all-to-all communication patterns in every merge stage is given in Culler et al. [1994]. We did not implement that variation because the constraint on the input size is problematic for the MasPar.

4.2.1 *Local Sort.* To sort the keys locally, we used an eight-bit radix sort. Let b denote the number of bits in a key (32-bit integers were used), and let 2^r be the radix of the sort. Radix sort requires time $T_{\text{local-sort}}(m) = \Theta((b/r) \cdot (2^r + m))$. Radix sort was implemented because it is faster than comparison-based sorting algorithms such as quicksort and heapsort.

4.3 Sample Sort

The implemented variation of sample sort proceeds in three phases. In Phase 1, $p - 1$ keys (*splitters*) are selected that partition the input into p subsets (*buckets*) of roughly equal size. In Phase 2, each key is routed to its appropriate bucket, where the ith bucket is stored in processor P_i. Finally, in Phase 3, the keys are sorted locally within each bucket.

The splitters are chosen as follows. First, every processor randomly picks a set of s sample keys from the keys in its possession, where s is called the oversampling ratio. This takes $O(s)$ local computation time. After that, the samples are concentrated in processor P_0. The BSP cost of this step is $g \cdot (p - 1) \cdot s + L$. At this point, processor P_0 sorts the samples and chooses the samples with ranks $s, 2 \cdot s, \dots, (p - 1) \cdot s$ as the $p - 1$ splitters. This step requires $O(T_{\text{local-sort}}(s \cdot p))$ local work. Finally, the splitters are broadcast from processor P_0 to all other processors

Table IV. BSP, MP-BSP, and BPRAM Cost of the Described Variations of Sample Sort

BSP		$W = \Theta(T_{\text{local-sort}}(s \cdot p) + T_{\text{local-sort}}(n/p) + T_{\text{local-sort}}(b_{\max}))$ $H = (p - 1) \cdot s + 2 \cdot p - 3 + b_{\max}$ $S = 4$
BPRAM	SSDR	$W = \Theta(T_{\text{local-sort}}(s \cdot p) + T_{\text{local-sort}}(n/p) + T_{\text{local-sort}}(b_{\max}))$ $M = (p - 1) \cdot (s + \log p + m_{\text{avg}})$, where $m_{\text{avg}} \geq n/p$ $R = 2 \cdot (p - 1 + \log p)$
BPRAM	SSBR	$W = \Theta(T_{\text{local-sort}}(s \cdot p) + T_{\text{local-sort}}(n/p) + T_{\text{local-sort}}(b_{\max}))$ $M = (p - 1) \cdot (s + \log p) + \log p \cdot m_{\text{avg}}$, where $m_{\text{avg}} \geq n/2$ $R = 4 \cdot \log p$
MP-BSP		$W = \Theta(T_{\text{local-sort}}(b^2_{\max}) + s \cdot \log^2 p + b^1_{\max} \cdot \log p)$ $H = \Sigma_{i=1}^{n/p} h_i^1 + \Sigma_{i=1}^{b^1_{\max}} h_i^2 + s \cdot \log p \cdot (\log p + 1) + 2 \cdot \sqrt{p} - 2 + 0.5 \cdot \log p$ $S = n/p + b^1_{\max} + s \cdot \log p \cdot (\log p + 1) + 2 \cdot \sqrt{p} - 2 + 0.5 \cdot \log p$
BPRAM	MasPar	$W = \Theta(T_{\text{local-sort}}(b^1_{\max}) + T_{\text{local-sort}}(b^2_{\max}) + s \cdot \log^2 p)$ $M = (\sqrt{p} - 1) \cdot m^1_{\text{avg}} + (\sqrt{p} - 1) \cdot m^2_{\text{avg}} + s \cdot \log p \cdot (\log p + 1)$ $+ 2 \cdot \sqrt{p} - 2 + 0.5 \cdot \log p$ $R = 4 \cdot \sqrt{p} - 4 + \log p \cdot (\log p + 1) + 1.5 \cdot \log p$

in two supersteps. In the first superstep, processor P_0 distributes the splitters so that every processor except P_{p-1} gets exactly one of them. In the second superstep, each processor sends the splitter it contains to all other processors. The BSP complexity of this step can be seen to be $g \cdot (2 \cdot p - 3) + 2 \cdot L$.

In Phase 2, every key is labeled with its appropriate bucket and routed to its destination processor. This is done as follows. First, every processor sorts its local list of input keys. This takes $O(T_{\text{local-sort}}(n/p))$ local computation time. Since the keys and splitters are now sorted, the bucket to which each key belongs can be determined in $O(n/p + s)$ time. (In other variations of sample sort [Blelloch et al. 1991; Gerbessiotis and Valiant 1992], the buckets are determined by performing a binary search over the array of splitters. We found that the described scheme is more efficient.) In the last step of Phase 2, each key is routed to its appropriate bucket. The BSP cost of this step is given by $g \cdot b_{\max} + L$, where $b_{\max}$ is the maximum number of keys in any bucket. The value of $b_{\max}$ was determined empirically.

In the third and final phase, the buckets are sorted locally. This takes local computation time $O(T_{\text{local-sort}}(b_{\max}))$. The total BSP complexity of sample sort is shown in Table IV.

4.3.1 *BRRAM Variation.* Unlike matrix multiplication and bitonic sort, sample sort cannot directly be adapted to utilize block transfers. The following substeps need to be implemented differently: (1) concentrating the samples in processor P_0, (2) broadcasting the splitters from processor P_0 to all other processors, and (3) routing the keys to their buckets.

The first substep is implemented in a binary tree fashion. The BPRAM cost of this is given by $\sigma \cdot w \cdot s \cdot (p - 1) + \ell \cdot \log p$. For the second substep there are several options. We can either use a scatter-broadcast approach as in the BSP algorithm, or we can simply broadcast the splitters using a binary-tree-like algorithm. The first approach takes $2 \cdot (\sigma \cdot w \cdot (p - 1) + \ell \cdot \log p)$ time under the BPRAM cost model, whereas the second approach takes $\log p \cdot (\sigma \cdot w \cdot (p - 1) + \ell)$ time. We found that because of its smaller startup cost, the second algorithm is to be preferred.

The most interesting part is the third substep in which each key is routed to its appropriate bucket. Again, there are several possibilities to implement this step. The simplest way is to send all keys destined for the same processor in a single large message. However, since every BPRAM processor may send/receive at most one message in a communication step, the communication must be staggered so that during communication step $j (1 \leq j \leq p - 1)$ processor P_i sends the keys destined for processor $P_{(i+j) \bmod p}$. Let m_i denote the maximum number of keys sent by any processor in communication step i, and let $m_{\text{avg}} = \sum_{i=1}^{p-1} m_i/(p - 1)$. The value of m_{avg} was determined empirically. The BPRAM cost of the routing substep is given by $(p - 1) \cdot (\sigma \cdot w \cdot m_{\text{avg}} + 2 \cdot \ell)$. Note that every communication step incurs two startups, because the sender needs to inform the recipient each time of the length of the incoming message. We call this variation of the BPRAM sort “sample sort with direct routing” (SSDR). Its total BPRAM complexity is given in Table IV.

The main problem with algorithm SSDR is that the routing substep incurs a large number of startups. It can be reduced by employing a butterfly-like algorithm. Then, the BPRAM cost of the routing substep is given by $\log p \cdot (\sigma \cdot w \cdot m_{\text{avg}} + 2 \cdot \ell)$, where $m_{\text{avg}} = \sum_{i=1}^{\log p} m_i/\log p$ and where m_i is the maximum length of any message sent during communication step $i (1 \leq i \leq \log p)$. This variation is called “sample sort with butterfly routing” (SSBR). Its total BPRAM cost is also summarized in Table IV. It is important to note that this variation increases the communication volume as well as the number of global synchronizations. It therefore goes against the incentives provided by the BSP model.

Both SSDR and SSBR can be problematic if the keys are not uniformly distributed, because the time needed for a communication step depends on the length of the longest message transferred. In this article, this issue is ignored because a random input is assumed.

4.3.2 *MasPar Variations.* On the MasPar, because of the small per-processor memory size, a different variation of sample sort was implemented. The MP-BSP version proceeds in three phases.

In the first step of the first phase, every processor picks s sample keys from its local set of keys. After that, the samples are sorted using bitonic sort. Thereupon, every processor P_i, where $i \equiv 0 (\bmod \sqrt{p})$ and $i \neq 0$, sends the smallest sample key in its possession to all other processors.

These $\sqrt{p} - 1$ splitters divide the processors into $\sqrt{p}$ groups consisting of $\sqrt{p}$ processors each. Now, every processor labels each key in its possession with the group it should be routed to. This is done by performing a binary search over the array of splitters. In the last step of the first phase, each key is sent to a processor in the appropriate group. The cost of this step will be modeled by the formula $\Sigma_{i=1}^{n/p}(g \cdot h_i^1 + L)$, where h_i^1 is the maximum number of keys received by any processor during the ith communication step. The values of the h_i^1's were determined empirically.

The second phase proceeds similarly. After that, each key is located in its appropriate destination processor. The cost of the routing step in the second phase is given by $\Sigma_{i=1}^{b_{\max}^1}(g \cdot h_i^2 + L)$, where $b_{\max}^1 \geq n/p$ is the maximum number of keys in any processor after Phase 1. Finally, a local sort is performed which takes $T_{\text{local-sort}}(b_{\max}^2)$ time, where $b_{\max}^2$ is the maximum number of keys in any processor after Phase 2.

The BPRAM variation proceeds almost identically, except that the keys are sorted locally prior to each routing step, so that the keys destined for the same processor can be sent at once. Refer to Table IV for the total MP-BSP and BPRAM complexities of the described variations of sample sort.

4.4 All Pairs Shortest Path

An important graph-theoretic problem is all pairs shortest path (APSP), which is defined as follows. Given a directed graph $G = (V, E)$, where $V = \{v_0, v_1, \ldots, v_{n-1}\}$ is a set of n vertices and $E \subseteq V \times V$ is a set of edges. With each edge (v_i, v_j), a length l_{ij} is associated. The task is to compute, for each pair of vertices v_i and v_j, the length of the shortest path from v_i to v_j. There are several algorithms for solving this problem. We implemented parallel variations of Floyd's algorithm [Aho et al. 1983], which is widely recognized as the fastest algorithm if the input graph is dense.

Floyd's algorithm can briefly be described as follows. An $n \times n$ matrix $D = (d_{ij})_{0 \leq i, j < n}$ is used to store the currently shortest path between any pair of nodes v_i and v_j. Initially, $d_{ii} = 0$, $d_{ij} = l_{ij}$ if there is an edge with length l_{ij} between v_i and v_j, and $d_{ij} = \infty$ otherwise. The algorithm then executes n iterations, and in iteration $k (0 \leq k < n)$ every element d_{ij} is set to $\min\{d_{ij}, d_{ik} + d_{kj}\}$.

There are several ways to parallelize this algorithm. Let the processors be designated by $P_{i,j}$ for $0 \leq i, j < \sqrt{p}$. In the simplest variation, which will be denoted by APSP 1, the matrix D is partitioned into p square submatrices $D_{ij} (0 \leq i, j < \sqrt{p})$ of size $n/\sqrt{p} \times n/\sqrt{p}$ each, and processor $P_{i,j}$ is assigned the task of updating D_{ij} in each iteration k. Then, in every iteration k, each processor $P_{i,j}$ that contains a segment of $D[*, k]$ needs to send it to the processors $P_{i,*}$, and each processor $P_{i,j}$ that contains a

Table V. (MP-)BSP and BPRAM Cost of All Pairs Shortest Path

BSP		BPRAM	
APSP 1	APSP 2	APSP 1	APSP 2
$W = \Theta(n^3/p)$ $H = 4 \cdot n^2/\sqrt{p}$ $S = 2 \cdot n$	$W = \Theta(n^3/p)$ $H = 6 \cdot n^2/\sqrt{p}$ $S = 6 \cdot \sqrt{p}$	$W = \Theta(n^3/p)$ $M = \log p \cdot n^2/\sqrt{p}$ $R = n \cdot \log p$	$W = \Theta(n^3/p)$ $M = 2 \cdot \log p \cdot n^2/\sqrt{p}$ $R = 2 \cdot \sqrt{p} \cdot \log p$
MP-BSP	$W = \Theta(n^3/p)$ $H = 4 \cdot n^2/\sqrt{p} + 2 \cdot n \cdot \max\{0, \log(p/n)\}$ $S = 4 \cdot n^2/\sqrt{p} + 2 \cdot n \cdot \max\{0, \log(p/n)\}$		

segment of $D[k, *]$ needs to send it to the processors $P_{*,j}$. Both steps correspond to an $(n/\sqrt{p})$-item broadcast operation for which an optimal BSP algorithm was presented in Juurlink and Wijshoff [1996b]. When $n \geq p$, the communication cost incurred in every iteration is given by $4 \cdot g \cdot n/\sqrt{p} + 2 \cdot L$ under the BSP cost model. In the BPRAM version, the broadcasts are implemented in a binary tree fashion, resulting in communication cost $\log p \cdot (\sigma \cdot w \cdot n/\sqrt{p} + \ell)$ in every iteration. On the MasPar under the MP-BSP model, it may happen that $n < p$. In this case, an extra phase in which each element is broadcast to p/n processors is required. This is implemented in a binary tree fashion, resulting in communication cost $(g + L) \cdot (4 \cdot n/\sqrt{p} + 2 \cdot \max\{0, \log\ (p/n)\})$ in every iteration.

If L and/or ℓ are large, it may be advantageous to use a different parallel variation of Floyd's algorithm derived from the external-memory algorithm given in Ullman and Yannakakis [1991]. This variation will be denoted by APSP 2. Briefly, the algorithm consists of $\sqrt{p}$ iteration, and in iteration $k(0 \leq k < \sqrt{p})$ the transitive closure of the diagonal submatrix D_{kk} is computed and broadcast to all processors. In addition, processor $P_{i,j}$ receives the submatrices D_{ik} and D_{kj}. For the broadcasts, we again used the scatter-broadcast approach of Juurlink and Wijshoff [1996b] in the BSP version, and the binary-tree-like algorithm in the BPRAM version. The BSP version requires $6 \cdot (g \cdot n^2/p + L)$ communication time in every iteration, and the BPRAM version needs $2 \cdot \log p \cdot (\sigma \cdot w \cdot n^2/p + \ell)$ communication time per iteration. It needs to be mentioned that although APSP 1 and APSP 2 require the same amount of local computation time asymptotically $(O(n^3/p))$, the constant hidden in the big-Oh notation is larger in APSP 2. The total running times of all variations are shown in Table V.

4.4.1 *E-BSP Analysis.* So far, we have ignored unbalanced communication. The communication patterns that arise in matrix multiplication and bitonic sort are perfectly balanced, and so the E-BSP complexities of these algorithms are equal to their BSP complexities. In sample sort, unbalanced communication occurs at three points: (1) when the samples are concentrated in processor P_0 (which corresponds to an $(s \cdot (p - 1), s \cdot (p - 1))$-relation), (2) when the splitters are distributed across the processors ($(p - 2, p - 2)$-relation), and (3) when each key is routed to its appropriate bucket ($(n, b_{\max})$-relation). However, because the communication overhead incurred in sample sort is dominated by the send phase for all but very small input sizes and because $b_{\max}$ is expected to be close to n/p, unbalanced communication should have almost no effect on the accuracy of the BSP model in sample sort. In this section, the E-BSP cost of the APSP algorithm is determined.

The E-BSP cost of APSP 1 is determined as follows. In the first superstep in the broadcast procedure a total of $2 \cdot n$ messages are being routed, and the processor assigned to the diagonal block sends the maximum number of messages, namely $2 \cdot n/\sqrt{p}$. The second superstep corresponds to a full h-relation with $h = 2 \cdot n/\sqrt{p}$. The communication cost incurred in every iteration is therefore $\max\{2 \cdot g \cdot n/p, 2 \cdot g' \cdot n/\sqrt{p}\} + 2 \cdot g \cdot n/\sqrt{p} + 2 \cdot L$. Similarly, the communication cost incurred in every iteration of APSP 2 can be seen to be $2 \cdot \max\{g \cdot n^2/p^{1.5}, g' \cdot n^2/p\} + \max\{g \cdot n^2/p^2, g' \cdot n^2/p\} + 3 \cdot g \cdot n^2/p + 6 \cdot L$.

The MP-E-BSP cost of APSP 1 is determined as follows. If $n \geq p$, then the first phase of the broadcast procedure consists of $n/\sqrt{p}$ communication steps, and in each step only $\sqrt{p}$ processors are active. The MP-E-BSP cost of this phase is therefore given by $(n/\sqrt{p}) \cdot T_{\text{unb}}(\sqrt{p})$. In the second phase, all processors are active, so that the communication cost is $(n/\sqrt{p}) \cdot T_{\text{unb}}(p)$. The total communication overhead is therefore

$$2 \cdot n \cdot \left(\frac{n}{\sqrt{p}} \cdot T_{\text{unb}}(\sqrt{p}) + \frac{n}{\sqrt{p}} \cdot T_{\text{unb}}(p)\right). \tag{2}$$

If $n < p$, the extra phase consists of $\log(p/n)$ communication steps, and in communication step i ($0 \leq i < \log(p/n)$), $2^i \cdot n$ PEs are active. In this case, the total communication cost is

$$2 \cdot n \cdot \left(\frac{n}{\sqrt{p}} \cdot T_{\text{unb}}(\sqrt{p}) + \frac{n}{\sqrt{p}} \cdot T_{\text{unb}}(p) + \sum_{i=0}^{\log(p/n)-1} T_{\text{unb}}(2^i \cdot n)\right). \tag{3}$$

5. COMPARING MEASURED AND PREDICTED PERFORMANCE

In this section, we investigate the predictive capabilities of the models by comparing the measured execution times for the set of synthetic benchmarks described in the previous section with the running times predicted by the models.

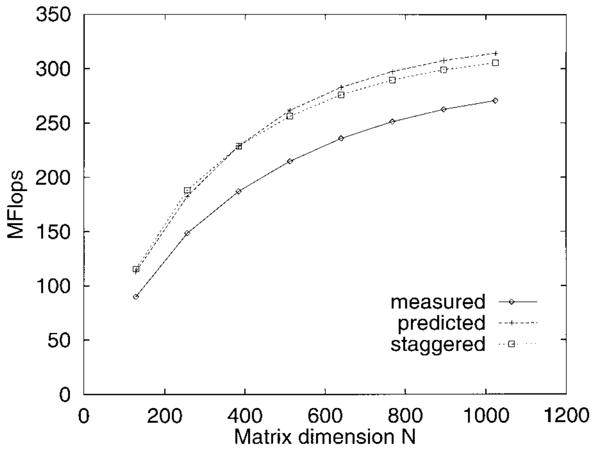

Fig. 13. Measured and predicted performance of the BSP matrix multiply on the CM-5.

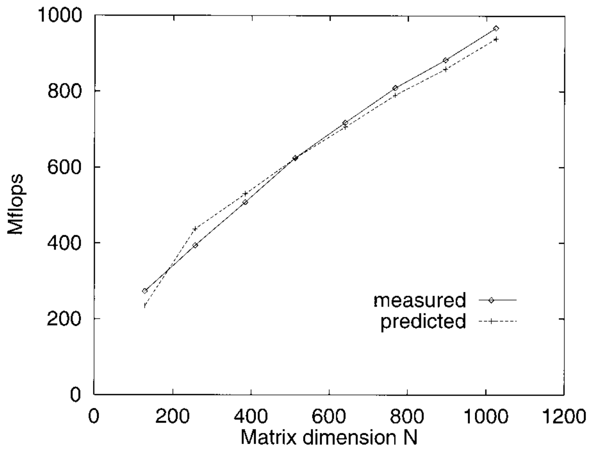

Fig. 14. Measured and predicted performance of the BSP matrix multiply on the T3E.

5.1 BSP

Figure 13 shows the measured and predicted performance of the BSP (short message) version of matrix multiplication on the CM-5. It can be seen that the BSP model does not accurately predict the actual performance. For example, for a matrix size of 128×128, the performance predicted by the BSP model is 113.0 Mflops, but the measured performance is 89.9 Mflops. The relative error is about 26%, which is too large to ignore given the regularity of the computation. The defect is caused by the fact that each processor $P_{i,j,k}$ first sends data to processor $P_{i,j,0}$, then to $P_{i,j,1}$, and so on. This causes stalls at the processor-network interface. We call the phenomenon *endpoint-contention*. In this case, it can be easily avoided by staggering the communication so that processor $P_{i,j,k}$ first sends data to processor $P_{i,j,(k+1) \bmod q}$, then to $P_{i,j,(k+2) \bmod q}$, and so on, just as was done explicitly in the MP-BSP and BPRAM variations. The curve labeled "staggered" in

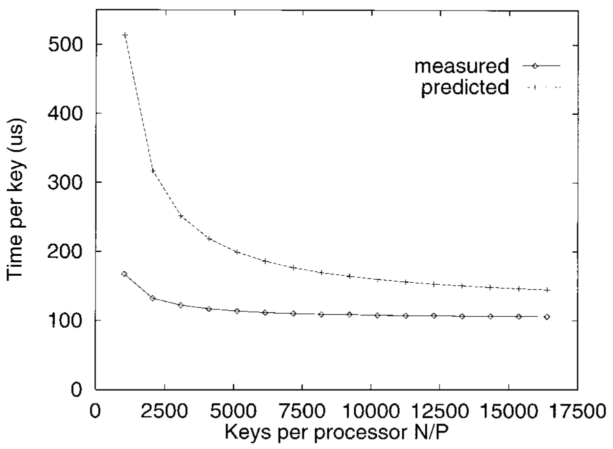

Fig. 15. Measured and predicted times per key for the BSP bitonic sort on the Paragon.

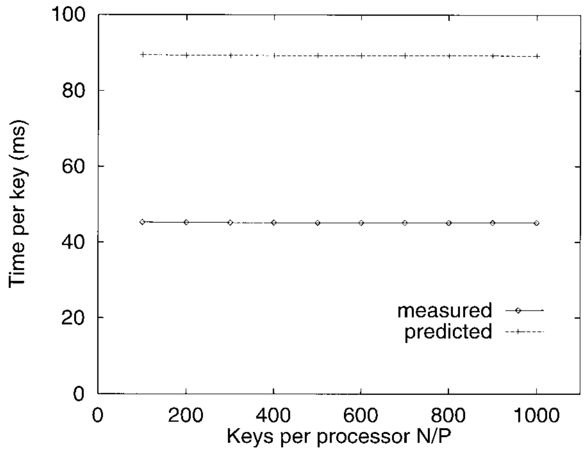

Fig. 16. Measured and predicted times per key for the MP-BSP bitonic sort on the MasPar.

Figure 13 shows the performance measured for this implementation. It can be seen that it indeed closely matches the performance predicted by the BSP model.

The implementation of BSPlib on the T3E defers all communication until the end of the superstep and sends messages in an order specified by a randomly generated Latin square, where row i is used as the schedule for processor P_i [Skillicorn et al. 1997]. Indeed, on this platform endpoint-contention does not occur (cf. Figure 14). The implementations of the BSP library on the GCel and the Paragon send messages in a staggered order. We found that for the set of algorithms we experimented with, both techniques were sufficient to ignore endpoint-contention. However, a major drawback of this approach is that it requires substantial memory space for buffering.

Figure 15 shows the measured and predicted times per key for the BSP

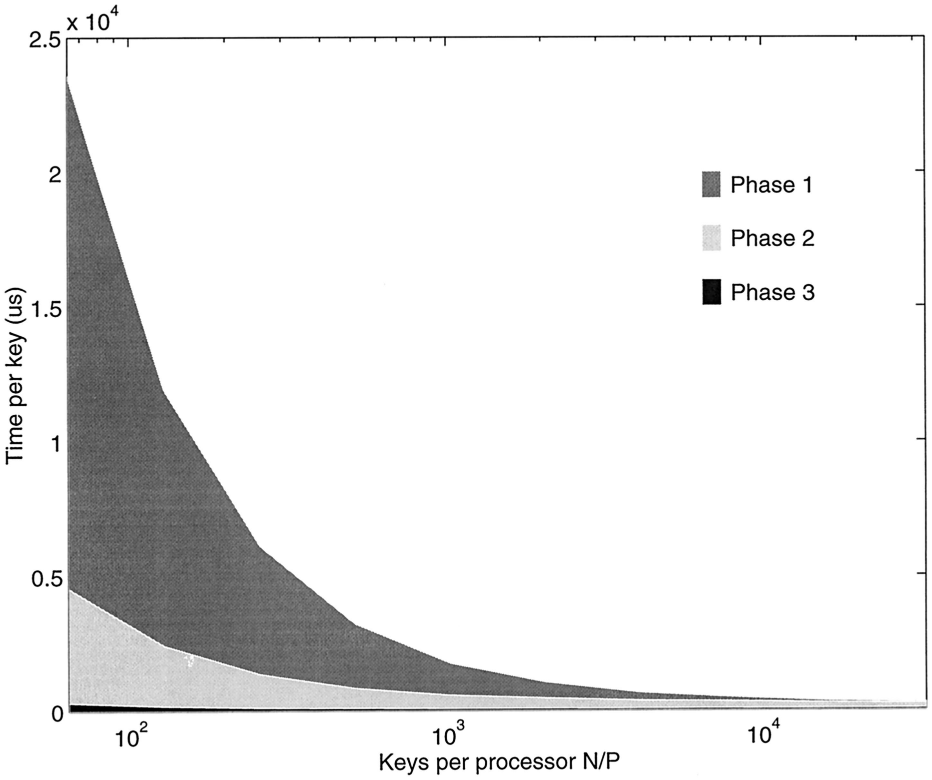

Fig. 17. Predicted times per key for the BSP version of sample sort on the GCel.

version of bitonic sort on the Paragon. The total running time is obtained by multiplying the time per key by the number of keys per processor (n/p). It can be seen that there are significant errors in the predictions, especially for small input sizes. For example, when $n/p = 1024$, the BSP model predicts that bitonic sort requires about 513 μsec. per key, but the measured time per key is only one-third of that, about 167 μsec. The error is caused by the fact that the communication patterns that arise in bitonic sort correspond to permutation-routing patterns (one-to-one routing), whereas the BSP cost model conservatively assumes that every communication superstep involves the routing of an h-relation (all-to-all routing). The implementation of the BSP library routes an arbitrary communication pattern using $d + s + \log p$ startups, where d (respectively s) is the maximum number of messages sent (respectively, received) by any processor. The extra $\log p$ startups are used for exchanging information about buffer sizes, number of packets in the buffer, etc. In bitonic sort $d = s = 1$, but in an arbitrary h-relation $d = s = p - 1$. So, when the startup times dominate (i.e., when n/p is small), this causes the BSP model to overestimate the actual execution times.

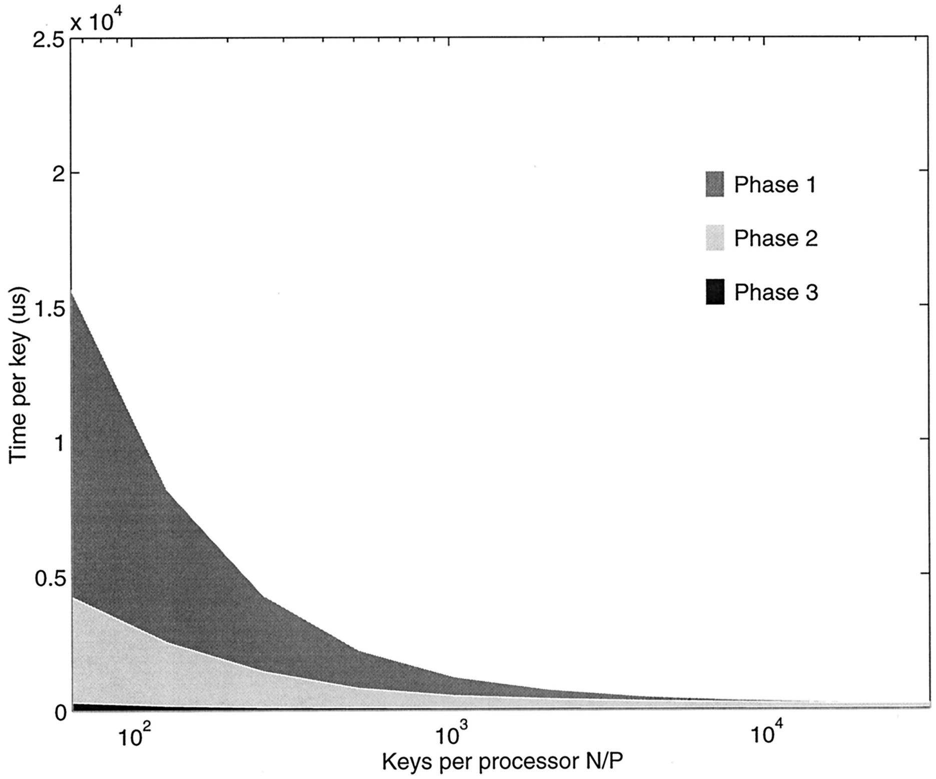

Fig. 18. Measured times per key for the BSP version of sample sort on the GCel.

There are also large errors in the MP-BSP predictions for bitonic sort on the MasPar. Figure 16 shows that the MP-BSP model continually overestimates the measured times per key by almost a factor of 2.0. The reason for this is that bitonic sort makes use of a communication pattern that is rather cheap on the MasPar global router. Experiments showed that permutations in which every processor P_i communicates with a processor whose ID is obtained by complementing a bit in i require approximately 590 μsec., which is less than 46% of the time needed for routing an average random permutation. Other frequently occurring communication patterns, such as regular shifts in which each processor P_i sends a message to processor $P_{(i+k) \bmod p}$, also take up to a factor of 2.2 less time than normally predicted. This effect could be nullified by randomly assigning processes to processors, but this is not a good idea for two reasons. First, it requires a large table in every PE's memory. Second, it would increase the communication time, and we do not believe that trading efficiency for accuracy is a good trade-off.

Figures 17 and 18 break down the predicted and measured running times of the BSP sample sort on the GCel into the time taken by each of the three phases. It can be seen that the BSP model overestimates the time taken by Phase 1 by almost a factor of two. For example, with 128 keys per processor, the BSP model predicts that Phase 1 requires about 9.5 msec. per key, but the measured time per key is approximately 5.6 msec. The errors should be attributed to unbalanced communication, since when the samples are concentrated in processor P_0, only one processor receives $h = s \cdot (p - 1)$ messages while all other processors send only s messages and receive none. The time taken by the routing phase is also not exactly predicted. For example, with 32K keys per processors, it needs 88 μsec. per key instead of the predicted 117 μsec. This error is also due to unbalanced communication, since not all processors send or receive as many as $b_{\max}$ keys.

5.2 BPRAM

Figure 19 depicts the measured and predicted performance of the BPRAM matrix multiplication program on the GCel. It appears that there are only minor errors in the predictions. However, when we measured the time taken by each of the three communication phases separately ((1) the broadcast of the A blocks, (2) the broadcast of the B blocks, and (3) the routing of the partial products), we found considerable differences between them. For example, when $n = 256$, Phase 1 took 76.5 msec., Phase 2 took 265.1 msec., and Phase 3 took 134.4 msec. The expected time for each phase is 205.6 msec. This can be explained by looking at the way the "logical" processors are mapped to physical processors. We simply assigned processor $P_{i,j,k}$ to the processor with PARIX ID $i \cdot p^{2/3} + j \cdot p^{1/3} + k$. Given that the processors are indexed row major under PARIX, the processors that communicate with each other in the first communication phase are close to each other. Furthermore, this mapping causes contention during the second communication phase, which explains why it needs more time than the other communication phases. The inaccuracy can be overcome by randomly assigning jobs to processors. Figure 19 also shows the performance attained by an implementation that employs a random mapping (the curve labeled "measured randomized"), and it can be verified that there is a close match with the predicted performance. A similar effect occurred on the Paragon. From now on, all results given for the GCel and the Paragon employ a random mapping of jobs to processors. The implementation of BSPlib on the T3E also uses a random mapping.

Figure 20 compares the estimated and measured times per key for the BPRAM version of bitonic sort on the MasPar. As was the case for the MP-BSP version of this algorithm, there are significant errors in the predictions; they are always off by a factor of approximately 1.5. Again, this is caused by the fact that the communication pattern that arises in bitonic sort is especially cheap on the MasPar global router. To incorporate this feature, the model needs to provide multiple bandwidth parameters σ that depend on the communication pattern being routed. Obviously, this cannot be done without sacrificing simplicity.

Figure 21 shows the measured and predicted communication times per element for the BPRAM version of APSP 2 on the Paragon. Because in this

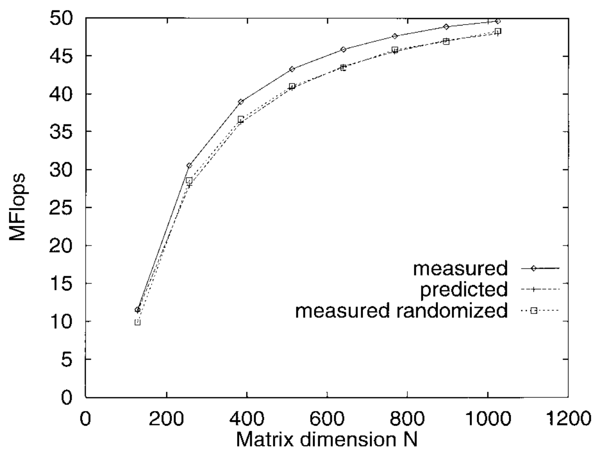

Fig. 19. Measured and predicted performance of the BPRAM matrix multiply on the GCel.

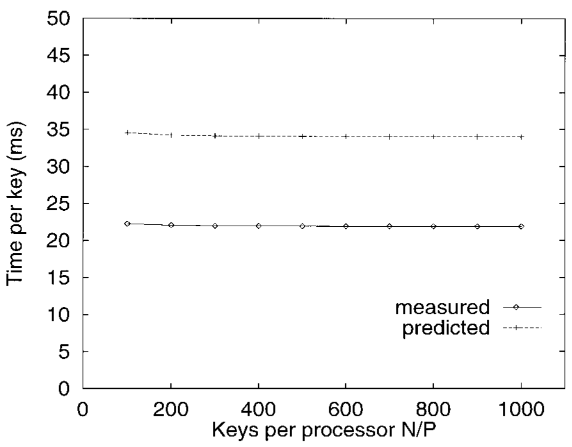

Fig. 20. Measured and predicted times per key for the BPRAM bitonic sort on the MasPar.

algorithm the communication overhead is only a small fraction of the total execution time, and because we consider the BPRAM to model only communication cost, the local computation time is not included. It can be seen that the BPRAM model does not accurately predict the actual communication times per element. For example, when $n = 256$, the measured and predicted times per element are about 106 μsec. and 158 μsec., respectively. For $n = 1024$, the measured and predicted times per element are given by 13.8 μsec. and 26.2 μsec., respectively, which corresponds to an error of almost 90%. When n is small, the error is mainly caused by the fact that in APSP 2 a processor never simultaneously acts as a sender and as a recipient. This leads to a smaller startup cost than in a full block permutation, in which each processor sends and receives simultaneously. On the other hand, when n is large, the errors are caused by unbalanced communication, because when the diagonal block is broadcast only a few proces-

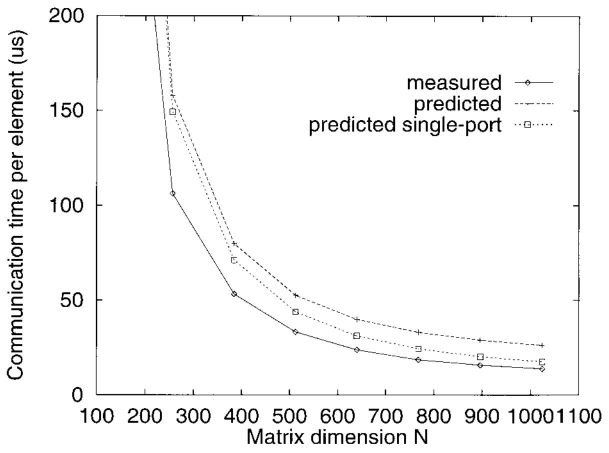

Fig. 21. Measured and predicted communication times per element for the BPRAM version of APSP 2 on the Paragon.

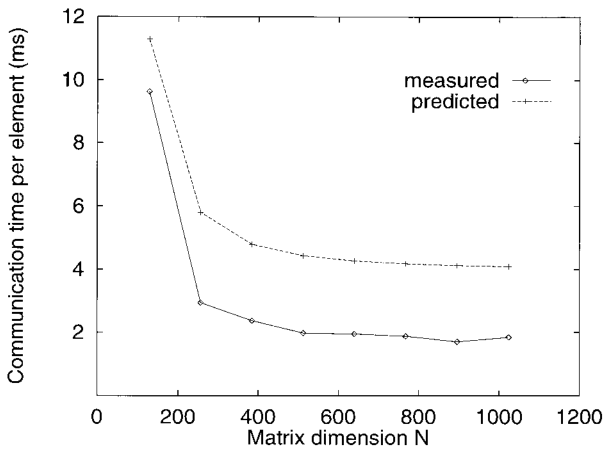

Fig. 22. Measured and predicted communication times per element for the BPRAM version of APSP 2 on the GCel.

sors are active during the top levels of the broadcast tree. However, the errors should only be partially attributed to bisection bandwidth limitations, because it appears that the links connecting the processors to the network are only half-duplex. To illustrate this, Figure 21 also shows the predictions of a single-port variation of the BPRAM that charges $\sigma \cdot \max_i\{m_s^i + m_r^i\} + \ell$ time for a communication step, where m_s^i is the length of the message sent by processor P_i and m_r^i is the length of the message received by processor P_i. These predictions are more accurate than the predictions of the original BPRAM model, but still do not coincide with the measured communication times.

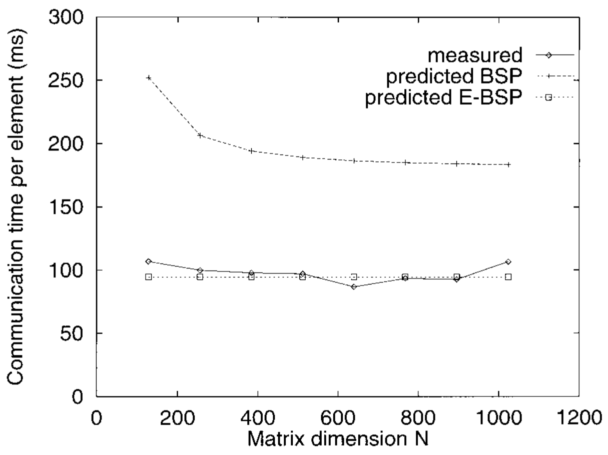

Fig. 23. Measured and predicted communication times per element for the MP-BSP version of APSP 1 on the MasPar.

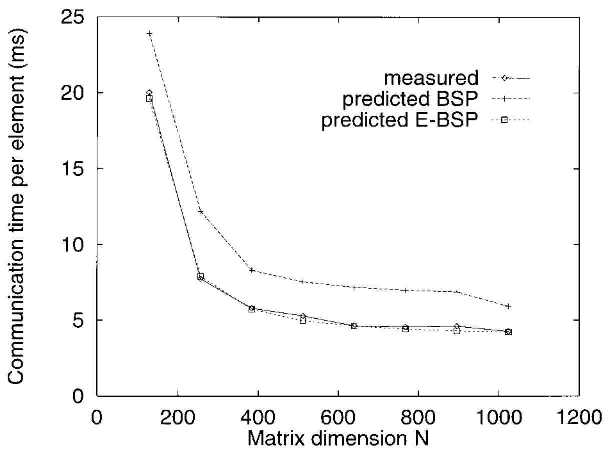

Fig. 24. Measured and predicted communication times per element for the BSP version of APSP 2 on the GCel.

5.3 E-BSP

In this section, we take the APSP problem as a case study to investigate the effect of unbalanced communication on the accuracy of the BSP model, and we show that the E-BSP model can be used to explain the difference between the measured and predicted execution times. Note that in APSP 1 as well as in APSP 2 unbalanced communication can cause errors of at most a factor of 2 (assuming everything else is modeled precisely), since the second communication superstep in every iteration corresponds to a full h-relation. (This does not hold for the MP-BSP variation of APSP 1 when $n < p$). Furthermore, on all platforms except the MasPar the computation time quickly dominates the communication overhead. Because of this, and because communication and synchronization costs are the only runtime

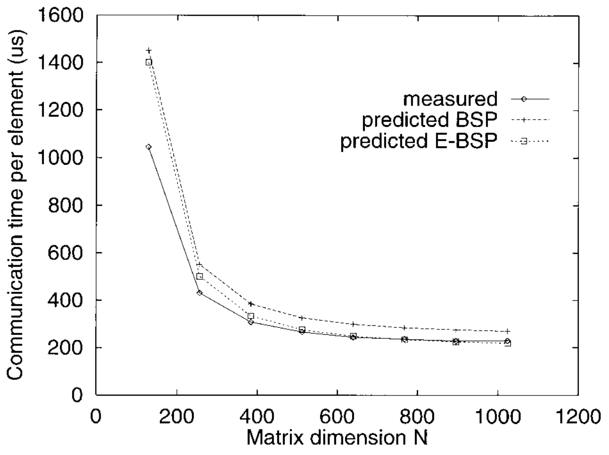

Fig. 25. Measured and predicted communication times per element for the BSP version of APSP 2 on the Paragon.

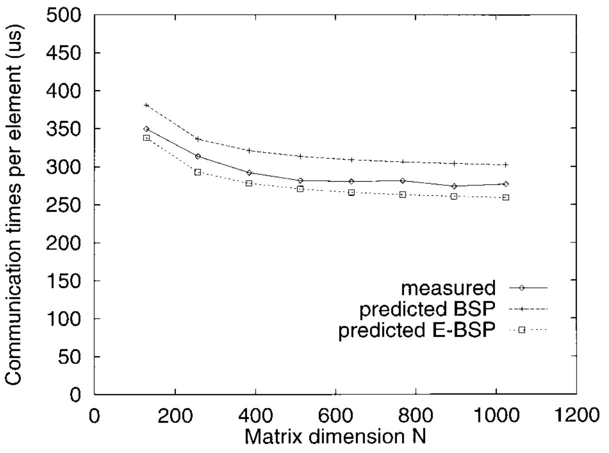

Fig. 26. Measured and predicted communication times per element for the BSP version of APSP 1 on the CM-5.

components modeled in detail by BSP, only the communication time incurred in the algorithm will be considered in this section.

Figure 23 shows the measured and predicted communication times for the MP-BSP variation of APSP 1. In order to place the data on a common scale, the total communication time is divided by the number of elements per processor (n^2/p). It can be seen that the estimated communication times per element deviate significantly from the measured ones. The largest error of 128% occurs at $n = 128$. The times predicted by MP-E-BSP (using Eq. (1)) are also depicted. These predictions portray the actual runtime behavior much more precisely; all errors are less than 12%. A similar effect occurred on the GCel (Figure 24). The BSP model continually overestimates the measured communication times per element, with a

Table VI. Summary of the Errors in the Predictions. Remarks: (1) For the APSP problem only the error in the predictions for the communication time is given. (2) Sample sort was not implemented on the CM-5, and APSP was not implemented on the T3E. (3) The error of 12% for the BPRAM matrix multiply on the GCel occurred when the nonrandomized version of the BPRAM library was used. Shown in parentheses is the maximum error observed when the version that randomly assigns jobs to processors was used. (4) The error of 26% for the BSP matrix multiply on the CM-5 is the maximum error measured for the naive (nonstaggered) version. Shown in parentheses is the maximum error observed for the staggered version. (5) The error of 90% for the BPRAM version of APSP 2 on the Paragon is the maximum error when the full-duplex BPRAM cost model was used. Shown in parentheses is the maximum error measured for the single-port variation of the BPRAM cost model.

		Platform				
Algorithm	Model	Paragon	GCel	T3E	CM-5	MasPar
matrix	BSP	41%	19%	16%	26% (3.0%)	11%
multiplication	BPRAM	12%	12% (4.6%)	1.7%	6.8%	1.6%
bitonic	BSP	207%	75%	33%	6.8%	98%
sort	BPRAM	6.8%	7.5%	1.6%	11%	55%
sample	BSP	16%	51%	21%	N/A	79%
sort	BPRAM	15%	13%	5.7%	N/A	9.5%
all pairs	BSP	39%	58%	N/A	11%	128%
shortest	E-BSP	34%	7.0%	N/A	7.1%	12%
path	BPRAM	90% (40%)	142%	N/A	14%	202%

largest error of 58% at $n = 256$. On the other hand, the predictions produced by the E-BSP model almost coincide with the measured data points.

Figures 25 and 26 show the measured and predicted communication times per element for APSP 2 on the Paragon and APSP 1 on the CM-5, respectively. Although BSP always overestimates the actual communication times, we emphasize that the errors are not due to congestion in the network, but to the fact that a small overhead is payed for every inbound and every outbound message. The BSP cost model could be easily modified to account for this. Furthermore, on the Paragon the largest error occurs at $n = 128$. For this problem size, the error is mainly due to the fact that in APSP 2 a processor never simultaneously acts as a sender and a receiver, which leads to a smaller startup cost and hence a smaller value of L. Since E-BSP also does not take this into account, its prediction is not much better than the one produced by BSP. On the CM-5, we observe a different phenomenon. Whereas BSP continually overestimates the measured communication times per element, E-BSP always underestimates the actual communication times.

5.4 Summary

Table VI shows the maximum errors measured for each algorithm-model-platform combination, where the error is defined as follows. Let T_{meas} be the measured execution time, and let T_{pred} be the predicted execution time. The error in the prediction is given by $|T_{\text{meas}} - T_{\text{pred}}|/\min\{T_{\text{meas}}, T_{\text{pred}}\}$. We used this definition so that the error is—say—100% when the model overestimates the actual execution time by a factor of 2, as well as when it underestimates the actual execution time by a factor of 2.

It can be seen that in most cases the BPRAM is more accurate than the BSP model. This is due to two reasons. First, and most importantly, the communication overhead incurred in the BPRAM algorithm is always less than the communication overhead incurred in the corresponding BSP algorithm. Second, the BPRAM model is more restrictive than the BSP model, since it requires that any communication pattern is broken down into a sequence of block permutations. The BPRAM, therefore, implicitly captures endpoint-contention. The BSP model is more flexible, but because of that, its predictions are less accurate. E-BSP is also more accurate than BSP in those cases that the communication is not perfectly balanced, especially on low-bandwidth systems such as the MasPar and the GCel. We also note that some of the errors in the BSP predictions on the Paragon and the T3E should be attributed to the inability to separate the computation from the communication cost. For example, sometimes address calculations were necessary in the communication loop that were not performed in the loop we used to perform h-relations. On the other platforms, communication is much more expensive than local computations, so that any overhead for address calculations can be neglected.

Concluding, we find that the models did not accurately predict the actual execution times in the following situations:

—Certain frequently occurring communication patterns require less time than normally predicted. This happened in both variations of bitonic sort on the MasPar and, to a lesser extent, in the BPRAM matrix multiply on the GCel.

—BSP does not capture endpoint-contention. This affected the accuracy of its predictions for matrix multiplication on the CM-5. The implementations of the BSP library on the GCel and the Paragon send the messages in a staggered order, whereas BSPlib on the T3E uses a Latin square to schedule the communication. For the algorithms we experimented with, this technique was sufficient to ignore endpoint-contention.

—Especially on the Paragon there is a high startup cost associated with every message transmission. Because of this, the BSP model significantly overestimated the time needed for performing communication patterns in which each processor does not send/receive messages to/from all other processors. This happened in bitonic sort and, to a lesser extent, in matrix multiplication and APSP. It also affected the accuracy of the E-BSP model.

—Both the BSP and the BPRAM model ignore unbalanced communication. This introduced significant errors in the communication times predicted

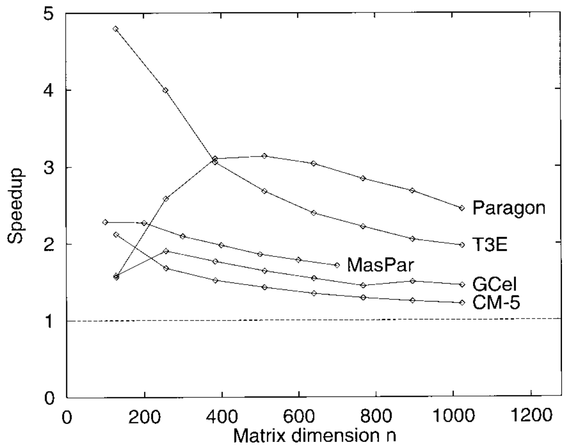

Fig. 27. Speedup of the BPRAM matrix multiply over the BSP matrix multiply.

for the APSP algorithms, especially on low-bandwidth systems like the MasPar and the GCel. On the CM-5 and the Paragon, the communication time is dominated by send and receive overheads when small messages are being routed. Unbalanced communication has therefore almost no effect on the accuracy of the BSP model on these platforms. Furthermore, a single-port variation of the BPRAM model produced much better predictions for APSP 2 on the Paragon, provided that the startup cost does not dominate.

6. COMPARISON OF THE MODELS

In this section, we compare the BSP and BPRAM models in order to determine the model that induces the fastest algorithms. The E-BSP model is not considered in this section, because the differences between the time needed for routing full h-relations and partial ones was too small to give an example of a problem for which the E-BSP model yields a faster algorithm than BSP. An example of such a problem is given in Juurlink and Wijshoff [1996a].

Figure 27 shows the speedup achieved by the BPRAM matrix multiplication algorithm over the BSP version of the matrix multiplication algorithm. It can be seen that the BPRAM algorithm always outperforms the BSP algorithm. On all platforms, except on the Paragon, the performance improvement decreases as the matrix dimension n increases. This is because the communication overhead becomes less significant as the matrix dimension n increases, since the amount of local work grows as $\Theta(n^3/p)$, whereas the communication time behaves as $\Theta(n^2/p^{2/3})$. Evidently, if the matrix dimension is increased even further, the BSP matrix multiply will eventually achieve performance comparable to the BPRAM matrix multiply. This is unsatisfactory, however, because for matrix multiplication it is easy to find an algorithm in which the computation time increases much more rapidly than the communication time. In other words,

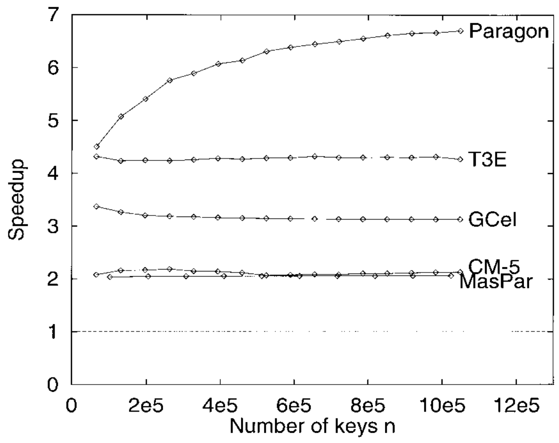

Fig. 28. Speedup of BPRAM bitonic sort over BSP bitonic sort.

the BSP algorithm is not very scalable. On the Paragon the speedup increases until $n = 512$, and after that it decreases. On this platform, the startup times dominate the communication overhead when n is small. The advantage of grouping data into long messages therefore increases as the amount of communication increases, until the local computation time starts to dominate the overall running time.

The speedup of the BPRAM version of bitonic sort over the BSP variation of the same algorithm is given in Figure 28. In this benchmark, the amount of local work does not increase more rapidly than the communication overhead as the problem size increases. On the T3E, GCel, CM-5, and MasPar, the speedups stay approximately constant, at about 4.3, 3.2, 2.1, and 2.1, respectively. The precise speedup depends very much on the ratio of processor speed to communication bandwidth. On the GCel, for example, the local sorting and merging steps contribute very little to the total running time, which explains why on this platform the improvement is about equal to the maximum improvement that one can expect: $g/(w \cdot \sigma) = 3.5$. On the T3E, on the other hand, the amount of local work in the BPRAM variation of bitonic sort is about 11 times larger than the communication time. Again, on the Paragon the speedup increases with increasing problem size. This is again due to the fact that the startup times dominate the communication overhead when n is small, whereas the data transfer time dominates when n is large.

Figure 29 plots the speedups of the BPRAM version of sample sort over the BSP variation of the same algorithm, where we have taken the fastest BPRAM algorithm (SSDR or SSBR). On the Paragon, SSBR was more efficient than SSDR for all measured data points due to its smaller number of startups. On the GCel, SSBR was the fastest sample sort variation up to $n/p = 8$K, and after that SSDR was the fastest variation. On the T3E, SSDR always outperformed SSBR.

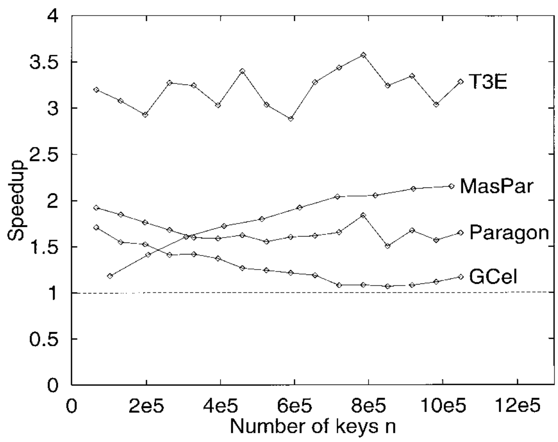

Fig. 29. Speedup of the BPRAM sample sort over the BSP sample sort.

For this algorithm, the largest speedup was measured on the T3E. However, the main reason for this is that the BSPlib message-passing primitive `bsp_send` which was used for routing the keys to their buckets performs rather poorly compared to the DRMA operation `bsp_hpput` (cf. Figure 5). On this platform, it would have been better to use `bsp_hpput`, but this requires a different program that computes the position of each key within its destination bucket. Of course, having to develop different program variations depending on the performance of the communication primitives cannot be called truly portable. On the Paragon, the BPRAM version of sample sort indeed outperforms the BSP variation, but not as clearly as in matrix multiplication and bitonic sort. This is because for these relatively small input sizes the communication overhead is dominated by the startup cost, and although SSBR reduces the number of startups, it needs extra time for rearranging the data so that the keys destined for the same processor are stored consecutively. Here we observe that on the MasPar the speedup increases as the problem size increases. This is due to the fact that the BPRAM algorithm is unable to send large messages when the number of keys per processor is small.

Thus, in most cases, sending a few long messages yields a significant improvement over sending many small ones, even if the BSP library packs small messages together. However, on many platforms a satisfactory performance can be obtained if fixed-size short messages are used, but they should be larger than one computational word. In order to validate this claim, we also programmed a BSP variation of sample sort in which 32-byte messages are used during the send phase. On the Paragon, the T3E, and the GCel, this required hardly any algorithmic changes, since the keys are already sorted prior to the send phase. On the MasPar, it required an intermediate local sorting step in both routing phases.

Figure 30 shows the speedup of BPRAM sample sort over the BSP version of sample sort that uses 32-byte messages. Now, the speedup is

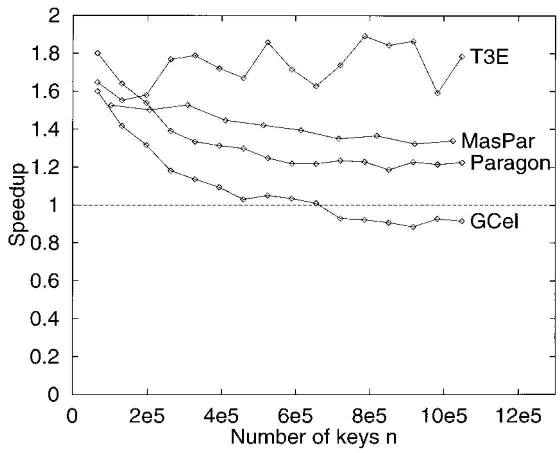

Fig. 30. Speedup of BPRAM sample sort over BSP sample sort with 32-byte messages.

never larger than 2. The speedup of about 1.85 on the T3E should again be attributed to the fact that `bsp_send` is relatively expensive compared to `bsp_hpput`. On the Paragon, the BSP version eventually incurs a performance penalty of about 20% compared to the BPRAM algorithm, which can be called acceptable. On the GCel, because the BSP algorithm performs fewer synchronizations, it even outperforms the BPRAM algorithm for $n/p > 10\text{K}$. On the MasPar, this variation of the BSP sample sort actually performs worse than the previous one for $n/p \leq 200$, because the intermediate local sorting steps require extra time. In the end, using 32-byte messages does save time overall.

7. EFFICIENCY VALIDATION

The ability to accurately predict the performance of parallel programs is a crucial property that a parallel computation model must possess, since it enables the programmer to pick the fastest algorithm from a set of possible alternatives without having to implement them. However, to be useful in practice, a model must also induce algorithms that are (nearly) as efficient as implementations customized for the architecture. In this section, we validate the efficiency of the model-derived algorithms by comparing them with machine-specific solutions. When available, we used the software present on each platform or software that is publically available. In some cases, we have written our own programs based on algorithms available in the literature.

7.1 Matrix Multiplication

In order to validate the efficiency of the model-derived matrix multiplication algorithms, we compared them with matrix multiplication routines present in mathematical libraries, where available. On the MasPar we used the `matmul` intrinsic. On the CM-5 we used the `gen_matrix_mult` routine

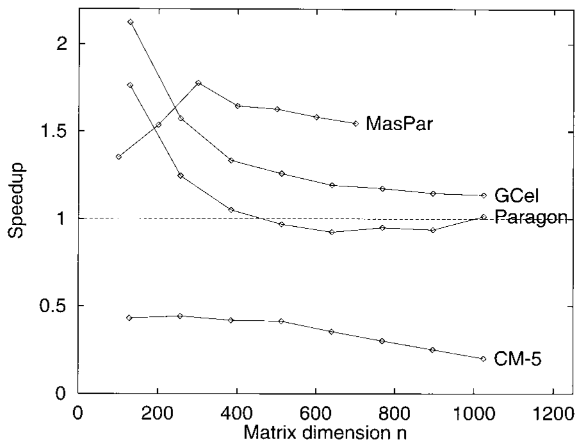

Fig. 31. Speedup of the machine-specific matrix multiplication programs over the fastest model-derived matrix multiplication algorithms.

present in the Connection Machine Scientific Software Library, and on the Paragon we used the psgemm routine from Parallel BLAS (PBLAS). On the GCel and the T3E, no parallel mathematical library was available. On the GCel, we therefore implemented Cannon's algorithm (e.g., see Kumar et al. [1994]), a systolic algorithm which is very well suited for mesh architectures like the GCel. It is important to note that the communication cost in the model-derived algorithms grows as $\Theta(n^2/p^{2/3})$, whereas the communication cost in Cannon's algorithm grows as $\Theta(n^2/p^{1/2})$. The BSP as well as the BPRAM therefore predict that Cannon's algorithm is inferior. We are unaware of a matrix multiplication algorithm tailored toward the T3E architecture.

Figure 31 shows the speedup of the machine-specific matrix multiplication programs over the fastest model-derived algorithms. On all platforms, this turned out to be the BPRAM matrix multiply. Since the communication overhead in matrix multiplication becomes less significant as the matrix size n increases, one would expect the speedup to approach 1 for large data size (provided that the local matrix multiplications perform equally well). Below, we discuss the results for each platform in detail.

7.1.1 *MasPar.* On this platform, due to the small per-processor memory size and due to the fact that the parallel algorithm requires $\Theta(n^2 \cdot p^{1/3})$ space, the largest matrix size that could be tested was $n = 700$. It can be seen that the BPRAM matrix multiply incurs a performance penalty between 35% and 78% compared to the matmul intrinsic. Since we do not know how matmul is implemented, we cannot be certain about the reason for the performance penalty. However, it is reasonable to assume that it uses the Xnet for communication, which has a bandwidth that is 16 times higher than that of the global router. Also notice that on the MasPar the performance penalty is largest for large matrix size. This should be attributed to the fact that on this platform the computation-to-communication ratio is smallest, since it is the platform with the largest number of processors.

7.1.2 *GCel*. On this platform, Cannon's algorithm outperforms the BPRAM algorithm for all measured matrix sizes, even though the BPRAM predicts otherwise. The reason is, of course, that Cannon's algorithm exploits network proximity (it requires only near-neighbor communication), whereas in the BPRAM model the network topology is hidden. Nevertheless, we believe that the performance penalty can be called acceptable, since for all $n \geq 512$, the BPRAM matrix multiply incurs a performance penalty of at most 25% compared to Cannon's algorithm. Obviously, the ability to exploit network proximity will become more important on larger scale platforms as the number of processors increases relative to the problem size.

7.1.3 *Paragon*. The speedup curve for the Paragon is similar to that for the GCel, but on this high-bandwidth platform, the communication overhead very quickly becomes insignificant. For matrix sizes between $n = 512$ and $n = 896$, the BPRAM matrix multiply even slightly (by at most 7%) outperforms the PBLAS subroutine `sgemm`. Thus, the larger the bandwidth, the easier it is to write portable algorithms that do not incur a large performance penalty compared to machine-specific code.

7.1.4 *CM-5*. Surprisingly, on the CM-5, the BPRAM matrix multiply turned out to be significantly faster than the vendor-supplied routine `gen_matrix_mult`. It needs to be mentioned, however, that the implementations do not use the CM-5's vector units, because they were not available under Split-C. For example, if compiled for the vector-units model, `gen_matrix_mult` achieves 1016 MFlops at $n = 512$, whereas the peak performance achieved by the scalar BPRAM program is 372 MFlops. Nevertheless, the BPRAM program compares fairly well with other matrix multiplication codes on the CM-5. For example, out of the LogP work a matrix multiplication program was developed [Krishnamurthy et al. 1993], which achieves a maximum performance of 413 MFlops. Compared to this program, the BPRAM algorithm incurs a performance penalty of about 10%.

7.2 Sorting

For matrix multiplication it is relatively easy to find an algorithm in which the communication overhead grows at a much smaller rate than the computation time. For integer sorting, however, the amount of computation per processor as well as the communication grows proportionally to the number of keys per processor. In this section we validate the efficiency of the model-derived sorting algorithm on the MasPar, the GCel, and the Paragon. Because sample sort was not implemented on the CM-5, and because we know of no comparison material for the T3E, these two platforms are not considered in this section.

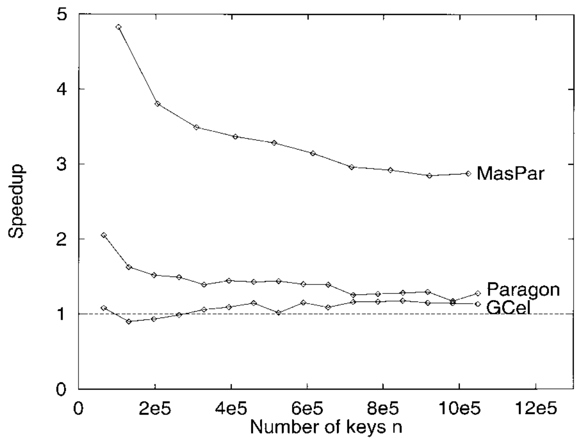

Fig. 32. Speedup of the machine-specific sorting algorithms over the model-derived sorting algorithms.

On the MasPar, we compare the fastest model-derived sorting algorithm with B-Flashsort [Hightower et al. 1992], which is a variation of sample sort that employs splitter-directed routing in order to reduce the memory requirements of sample sort. The source code of B-Flashsort is publically available. Sorting on a 1024-processor GCel was considered in Diekmann et al. [1994], where it was shown that a variation of bitonic sort is the fastest sorting algorithm for the GCel when n/p is small, whereas sample sort is the fastest algorithm when n/p is large. The bitonic sort variation employs a clever embedding of the hypercube into the mesh, that maps hypercube links that are used more frequently (links in the lower numbered dimensions) to short paths in the mesh. The sample sort variation described in Diekmann et al. [1994] is very similar to SSDR, except that no global synchronization is performed after every communication step during the routing phase. Because the Paragon also has a 2D mesh topology, we also implemented these two algorithms on this platform.

Figure 32 shows the speedup of the machine-specific sorting algorithms over the fastest model-derived algorithms. Below, the results for each platform are discussed in detail.

7.2.1 *Paragon.* It can be seen from Figure 32 that the fastest machine-specific sorting algorithm outperforms the fastest model-derived algorithm for all measured problem sizes. The speedup ranges from 2.05 (for $n = 64$K) to 1.17 (for $n = 960$K). However, these speedups do not reflect the reductions in communication costs, which are much larger. For example, when $n = 1$M, the fastest BPRAM sort (SSBR) requires 5.1 μsec. per key, whereas the fastest machine-specific sort (sample sort) needs 2.1 μsec. per key, corresponding to an improvement by a factor of about 2.4. However, because the local sorting steps require about 7.0 μsec. per key, this does not result in a similar overall improvement. Thus, if the processor speed would increase, the speedup of the machine-specific sort over the model-derived sorting algorithm would also increase.

7.2.2 *GCel.* On this platform, the communication overhead incurred in the variation of bitonic sort that employs a hypercube embedding is almost a factor of 3.6 smaller than the communication overhead incurred in the BPRAM bitonic sort. However, sample sort turned out to be more efficient than bitonic sort for all measured problem sizes. The reason is that on this moderate-scale platform, the number of startups incurred in bitonic sort is not much smaller than the number of startups incurred in sample sort. Figure 32 therefore shows the speedup of the machine-specific sample sort (without global synchronizations between successive communication steps) over the BPRAM sample sort (with global synchronizations). It can be seen that the BPRAM algorithm does not incur a large performance penalty. The reason is that the cost of a global synchronization is not prohibitive on the GCel. For $2048 \leq n/p \leq 4096$, it is even slightly faster than the machine-specific sample sort, but this should be attributed to differences in the bucket expansion.

7.2.3 *MasPar.* On this platform, B-Flashsort is substantially faster than the BPRAM sample sort. The speedup varies from 4.8 (for $n = 100\text{K}$) to 2.9 (for $n = 1000\text{K}$). The main reason for this is that B-Flashsort uses the Xnet for communication, whereas the BPRAM sample sort uses the global router. Some of the difference, however, should be attributed to coding differences. For example, in B-Flashsort the local arrays are padded with ∞'s to make them all the same size, so that so-called singular loops can be used during the local radix sort instead of the more expensive plural loops. Nevertheless, the fact that for matrix multiplication as well as for sorting the largest performance penalty is incurred on the MasPar shows that the models are less suited for developing cost-effective algorithms on massively parallel platforms when the computation-to-communication ratio is small.

8. CONCLUSIONS

In this article for the first time an attempt was made to provide a rigorous account of the usefulness of parallel computation models for designing parallel algorithms for existing parallel computer platforms. The investigation concentrated on two issues: how accurately do parallel computer models predict actual performance and how efficient are the implementations of the parallel algorithms derived from these models. For the latter question we compared actual performance between implementations derived through different models against each other, as well as implementations obtained through these models against optimized implementations on specific architectures.

Concerning the predictive capabilities of the models, this article has shown that there are several situations in which the models do not accurately predict the actual runtime behavior of an algorithm implementation. In fact, on all of the five hardware platforms we have been observing performance deviations of at least 25%. In some cases even performance deviations of the order of 100% to 200% were observed (see Table VI). Although this might not be very surprising it definitely makes these models unsuitable for actually predicting performance on existing hardware platforms.

Of the three models we investigated, the BPRAM model more accurately predicted the performance of three of the four algorithms on all five hardware platforms (except for bitonic sort on the CM-5, cf. Table VI). From this we can conclude that concentrating all the communication into supersteps (BSP) does not trade off at all with the explicit message blocking of communication (BPRAM). Therefore, as long as we cannot rely solely on runtime support to combine messages into larger blocks, the BPRAM model should be considered better in predicting actual performance behavior. Actually, one might wonder whether the randomized routing assumption, which is an integral part of the BSP model, does not contradict the need for blocking messages.

In the light of the fact that none of the models were able to accurately predict performance, it comes to no surprise that the implementations derived from these models also did not measure up against optimized implementations on the hardware platforms. The question to be raised is whether these models could be used to incrementally improve the performance of implementations. For this to be valid the following assertion must hold: if algorithm A is faster than algorithm B for a specific model $\mathcal{M}$, then the implementation of algorithm A is going to be executed faster than the implementation of algorithm B on a particular architecture. Although this latter assertion has not been explicitly addressed by this article, we can conclude from the observation that the SSBR implementation in fact performed better on the Intel Paragon although its BSP costs are worse than the SSDR implementation, and from the fact that Cannon's algorithms outperforms the BPRAM algorithm on the GCel, while its costs according to the BPRAM model should be higher, that this assertion also does not hold.

The above-summarized conclusions might seem to be very negative. One should take into account, though, that the proposed computational models definitely serve a purpose within the complexity of algorithms community, in which actual performance numbers are not as important as providing a better insight into the nature of parallel computing. As such their merit should probably be better measured in the architectural details they are able to avoid while still providing reasonable prediction of actual performance behavior. Also, the von Neumann model, which was always assumed to be very accurate by the theory community and which led to very important theoretical results, in fact never was very accurate. Cache performance, especially in the case of multilevel cache organizations, has never been modeled accurately by the von Neumann model, and still is very much an issue for research.

ACKNOWLEDGMENTS

Computational support was provided by IC^{3}A in Amsterdam, Leiden University, NCSA at Urbana-Champaign, HPAC Delft, and KFA Jülich. The authors wish to thank Arno van Duin, Arjan Bik, and the anonymous referees for their comments.

REFERENCES

AGGARWAL, A., CHANDRA, A. K., AND SNIR, M. 1989. On communication latency in PRAM computations. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures* (SPAA '89, Santa Fe, NM, June 18–21), F. T. Leighton, Ed. ACM Press, New York, NY, 11–21.

AGGARWAL, A., CHANDRA, A. K., AND SNIR, M. 1990. Communication complexity of PRAMs. *Theor. Comput. Sci. 71*, 1 (Mar.), 3–28.

AHO, A., HOPCROFT, J., AND ULLMAN, J. 1983. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA.

BATCHER, K. 1968. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*. AFIPS Press, Arlington, VA, 307–314.

BLANK, T. 1990. The MasPar MP-1 architecture. In *Proceedings of IEEE CompCon Spring*. IEEE Press, Piscataway, NJ, 20–24.

BLELLOCH, G. E., LEISERSON, C. E., MAGGS, B. M., PLAXTON, C. G., SMITH, S. J., AND ZAGHA, M. 1991. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures* (SPAA '91, Hilton Head, SC, July 21–24). ACM Press, New York, NY, 3–16.

CULLER, D., DUSSEAU, A., MARTIN, R., AND SCHAUSER, K. 1994. Fast parallel sorting under LogP: From theory to practice. In *Portability and Performance for Parallel Processing*, Hey, T. and Ferrante, J., Eds. John Wiley & Sons, Inc., New York, NY.

CULLER, D., KARP, R., PATTERSON, D., SAHAY, A., SCHAUSER, K. E., SANTOS, E., SUBRAMONIAN, R., AND VON EICKEN, T. 1993. LogP: Towards a realistic model of parallel computation. *SIGPLAN Not. 28*, 7 (July), 1–12.

DE LA TORRE, P. AND KRUSHAL, C. P. 1991. Towards a single model of efficient computation in real parallel machines. In *Proceedings of the Conference on Parallel Architectures and Languages Europe: Vol. 1, Parallel Architectures and Algorithms* (PARLE '91, Eindhoven, The Netherlands, June 10–13), E. H. L. Aarts, J. van Leeuwen, and M. Rem, Eds. Lecture Notes in Computer Science, vol. 505. Springer-Verlag, New York, NY, 7–24.

DIEKMANN, R., GEHRIG, J., LÜLING, R., MONIEN, B., NUBEL, M., AND WANKA, R. 1994. Sorting large data sets on a massively parallel system. In *Proceedings of the Syposium on Parallel and Distributed Processing*.

FORTUNE, S. AND WYLLIE, J. 1978. Parallelism in random access machines. In *Proceedings of the 10th Symposium on Theory of Computing*. ACM Press, New York, NY, 114–118.

GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R., AND SUNDERAM, V. 1993. PVM 3 user's guide and reference manual. Tech. Rep. TM-12187. Oak Ridge National Laboratory, Oak Ridge, TN.

GERBESSIOTIS, A. AND VALIANT, L. 1992. Direct bulk-synchronous parallel algorithms. In *Proceedings of the 3rd Scandinavian Workshop on Algorithm Theory*, O. Nurmi, Ed. Lecture Notes in Computer Science, vol. 621. Springer-Verlag, Berlin, Germany, 1–18.

GOUDREAU, M., LANG, K., RAO, S., SUEL, T., AND TSANTILAS, T. 1996. Towards efficiency and portability: Programming with the BSP model. In *Proceedings of the 8th Symposium on Parallel Algorithms and Architectures*. ACM Press, New York, NY, 1–12.

GROSCUP, W. 1992. The Intel Paragon XP/S supercomputer. In *Proceedings of the 5th ECMWF Workshop on the Use of Parallel Processors in Meteorology*.

HEYWOOD, T. AND RANKA, S. 1992. A practical hierarchical model of parallel computation I: The model. *J. Parallel Distrib. Comput. 16*, 212–232.

HIGHTOWER, W. L., PRINS, J. F., AND REIF, J. H. 1992. Implementations of randomized sorting on large parallel machines. In *Proceedings of the 4th Annual ACM Symposium on Parallel*

Algorithms and Architectures (SPAA '92, San Diego, CA, June 29–July 1). ACM Press, New York, NY, 158–167.

HILL, J., MCCOLL, W., STEFANESCU, D., GOUDREAU, M., LANG, K., RAO, S., SUEL, T., TSANTILAS, T., AND BISSELING, R. 1997. The BSPlib—The BSP programming library.

HONG, J. AND KUNG, H. 1981. I/O complexity: The red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing* (STOC 81). ACM, New York, NY, 326–333.

JUURLINK, B. H. H. 1998. Experimental validation of parallel computations models on the Intel Paragon. In *Proceedings of the International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing* (IPPS/SPDP '98).

JUURLINK, B. H. H. AND WIJSHOFF, H. A. G. 1993. Experiences with a model for parallel computation. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing* (PODC '93, Ithaca, NY, August 15–18). ACM Press, New York, NY, 87–96.

JUURLINK, B. AND WIJSHOFF, H. 1996a. A quantitative comparison of parallel computation models. In *Proceedings of the 8th Symposium on Parallel Algorithms and Architectures*. ACM Press, New York, NY, 13–24. Full version available as TR-96-01, Leiden University, The Netherlands.

JUURLINK, B. AND WIJSHOFF, H. 1996b. Communication primitives for BSP computers. *Inf. Process. Lett. 58*, 6 (June), 303–310.

JUURLINK, B. AND WIJSHOFF, H. 1996c. The E-BSP model: Incorporating unbalanced communication and general locality into the BSP model. In *Proceedings of Eur-Par '96* (Euro-Par '96). Lecture Notes in Computer Science, vol. 1124. Springer-Verlag, Berlin, Germany, 339–347.

KRISHNAMURTHY, A., CULLER, D. E., DUSSEAU, A., GOLDSTEIN, S. C., LUMETTA, S., VON EICKEN, T., AND YELICK, K. 1993. Parallel programming in Split-C. In *Proceedings of Supercomputing* (Supercomputing '93, Portland, OR, Nov. 15–19). IEEE Computer Society Press, Los Alamitos, CA, 262–273.

KUMAR, V., GRAMA, A., GUPTA, A., AND KARYPIS, G. 1994. *Introduction to Parallel Programming*. Benjamin-Cummings Publ. Co., Inc., Redwood City, CA.

LANGHAMMER, F. 1992. Second generation and teraflops parallel computers. In *Parallel Computing and Transputer Applications*, Valero, M., Onate, E., Jane, M., Larriba, J., and Suarez, B., Eds. IOS Press, Amsterdam, The Netherlands, 62–79.

LEISERSON, C. E., ABUHAMDEH, Z. S., DOUGLAS, D. C., FEYNMAN, C. R., GANMUKHI, M. N., HILL, J. V., HILLIS, D., KUSZMAUL, B. C., ST. PIERRE, M. A., WELLS, D. S., WONG, M. C., YANG, S.-W., AND ZAK, R. 1992. The network architecture of the Connection Machine CM-5 (extended abstract). In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures* (SPAA '92, San Diego, CA, June 29–July 1). ACM Press, New York, NY, 272–285.

MCCOLL, W. F. 1993. General purpose parallel computing. In *Lectures on Parallel Computation*, Gibbons, A. and Spirakis, P., Eds. Cambridge International Series on Parallel Computation. Cambridge University Press, New York, NY, 337–391.

MCCOLL, W. 1995. Scalable computing. In *Computer Science Today: Recent Trends and Developments*. Springer Lecture Notes in Computer Science, vol. 1000. Springer-Verlag, Berlin, Germany.

THE MPI FORUM. 1993. MPI: A message passing interface. In *Proceedings of Supercomputing* (Supercomputing '93, Portland, OR, Nov. 15–19). IEEE Computer Society Press, Los Alamitos, CA, 878–883.

NICKOLLS, J. 1990. The design of the MasPar MP-1: A cost-effective massively parallel computer. In *Proceedings of IEEE CompCon Spring*. IEEE Press, Piscataway, NJ, 25–28.

OBERLIN, S., KESSLER, R., SCOTT, S., AND THORSON, G. 1996. Cray T3E architecture overview. Cray Supercomputers, Chippewa Falls, MN.

SHUMAKER, G. AND GOUDREAU, M. 1997. Bulk-synchronous parallel computing on the Maspar. In *Proceedings of the World Multiconference on Systemics, Cybernetics and Informatics*. 475–481.

SKILLICORN, D. 1991. Models for practical parallel computation. *Int. J. Parallel Program. 20*, 2, 133–158.

SKILLICORN, D., HILL, J., AND MCCOLL, W. 1997. Questions and answers about BSP. *J. Sci. Program. 6*, 3, 249–274.

ULLMAN, J. AND YANNAKAKIS, M. 1991. The input/output complexity of transitive closure. *Ann. Math. Art. Intell. 3*, 331–360.

VALIANT, L. G. 1990. A bridging model for parallel computation. *Commun. ACM 33*, 8 (Aug.), 103–111.

Received: April 1996; revised: May 1997; accepted: May 1998