Portable and Efficient Parallel Computing Using the BSP Model*

MARK W. GOUDREAU† KEVIN LANG‡ SATISH B. RAO§

TORSTEN SUEL¶ THANASIS TSANTILAS‖

4 June 1998

Abstract

The Bulk-Synchronous Parallel (BSP) model was proposed by Valiant as a standard interface between parallel software and hardware. In theory, the BSP model has been shown to allow the asymptotically optimal execution of architecture-independent software on a variety of architectures. Our goal in this work is to experimentally examine the practical use of the BSP model on current parallel architectures. We describe the design and implementation of the Green BSP Library, a small library of functions that implement the BSP model, and of several applications that were written for this library. We then discuss the performance of the library and application programs on several parallel architectures. Our results are positive, in that we demonstrate efficiency and portability over a range of parallel architectures, and show that the BSP cost model is useful for predicting performance trends and estimating execution times.

Index Terms: BSP, minimum spanning tree problem, models of parallel computation, N-body problem, parallel computing, parallel graph algorithms, shortest path problem.

1 Introduction

A fundamental goal of parallel computing is the development of portable and efficient parallel programs. Valiant has argued that his Bulk-Synchronous Parallel (BSP) model achieves both portability and efficiency for a large class of problems [57]. The cost of portability is that BSP code may require a larger input size than machine-specific code in order to achieve

*A preliminary version of this work appeared in the *8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

†Department of Computer Science, University of Central Florida, Orlando, FL 32816-2362.

‡NEC Research Institute, 4 Independence Way, Princeton, NJ 08540.

§NEC Research Institute, 4 Independence Way, Princeton, NJ 08540.

¶Bell Laboratories, 700 Mountain Avenue, Murray Hill, NJ 07974. Most of this work was done while at the NEC Research Institute.

‖Department of Computer Science, Columbia University, New York, NY 10027.

the desired level of efficiency. Our objective is to determine if the BSP model is a practical model for current parallel systems. Specifically, we wish to discover if portability using the BSP model can be demonstrated while achieving efficiency for realistic input sizes.

The BSP model, being an interface between software and hardware, incorporates aspects of both domains. The BSP model is discussed in detail in Section 3. Here we mention several of the most significant arguments in support of the model:

- For the architect the BSP model describes a parallel computer with three attributes: A collection of components each performing stand-alone processing and memory functions; a router that delivers messages point-to-point between any two components; and a mechanism for synchronizing all components. The BSP model presents clearly-defined design goals while allowing for a wide range of implementations.

- For the programmer the BSP model dictates a disciplined but fairly general and user-friendly programming style. (In this paper, we examine only BSP programs written in *direct mode* [24]. Such programs are written directly for a BSP computer, take into account the number of processors, and have one process per processor.)

- For the algorithm designer the BSP model provides a simple cost function for analyzing the complexity of algorithms. This allows BSP to serve as a framework for developing a theory of efficient algorithm design.

- The BSP model can efficiently simulate several other models of parallel computation [8, 24, 28, 27, 58], including the PRAM model. (Such BSP programs are said to be written in *automatic mode* [24].)

Despite these arguments in support of the BSP model, the short-term applicability of the BSP model is yet uncertain. In particular, most current parallel computers are not designed to support the fundamental routing problem of the BSP model: the h-relation. An h-relation is a routing problem such that h is the maximum number of (fixed-sized) packets sent or received by any processor. While the efficient routing of h-relations has been the subject of numerous theoretical studies, most system designers focus instead on optimizing communication at the single-message level [31]. In addition, the BSP model provides a highly abstract model of the underlying computer. In some situations the BSP cost function may be overly simplistic and lead the programmer astray. For example, the cost function assumes that all h-relations require the same amount of time to route, which may be unrealistic for some systems.

We attempt to evaluate the use of the BSP model for the design of efficient and portable parallel programs. In particular, we are interested in exploring the range of algorithms and applications that can be efficiently implemented in the BSP model. While there seems to be general agreement that some problems can be efficiently solved in this model, it has also been argued that there may be other problems that require asynchronous message passing or even shared memory for an efficient implementation on current machines. Thus, we believe that in order to argue for BSP as a basis of general-purpose parallel computing, it is necessary to show that the model is not restricted to certain classes of well-behaved problems, but can indeed efficiently implement most parallel applications of interest. By exploring this issue, we also wish to give a basis for a comparison with asynchronous models such as LogP and certain shared-memory models.

We designed several parallel applications that use the Green BSP Library [30], a small library of BSP message-passing functions we have implemented on a number of parallel platforms. Inspired by the SPLASH suite of shared-memory applications [56], we focus on a variety of realistic applications. The applications are:

- an N-body simulation using the Barnes-Hut algorithm (N-body),
- an ocean eddy simulation program adapted from the SPLASH application suite (Ocean) [56],
- a minimum spanning tree algorithm (MST),
- a shortest paths algorithm (SP),
- a multiple shortest paths algorithm (MSP), and
- a dense matrix multiplication algorithm (Matmult).

In all of our applications, we used only the BSP cost function in both the design and optimization stages of the program development. Our approach assumed that communication is somewhat more expensive than local computation, and that barrier synchronization is considerably more expensive than communication. This approach appears reasonable for a wide range of current machines. In discussing our applications, we will touch upon some of our programming decisions and their relationship to the BSP cost function.

We describe implementations of the Green BSP Library on three different machines: a shared-memory machine, a distributed-memory machine, and a network of PCs. We then characterize the performance of these machines in terms of the BSP cost model, and evaluate the performance of our applications on these machines. Our results are encouraging, in that our BSP applications obtain significant speed-ups on all three systems, including nearly perfect speed-up in several instances.

Another question that we investigate is the accuracy of the BSP cost function in predicting execution times. Following [9], we provide data for our applications that can be used to predict the execution times on each machine under the BSP cost model. Our results demonstrate that the model is able to predict execution times fairly accurately, although we emphasize that we used the BSP cost function only to model communication and synchronization costs, and for some of our application these costs turned out to be a small component of the overall execution time. An example is shown in Figure 1. For this particular application, the communication and synchronization overheads in our implementation were negligible. (In fact, we credit the simplicity of the BSP cost model for guiding us to such efficient solutions).

However, even for those applications for which the communication and synchronization costs are significant, our results suggest the cost function is quite reliable in predicting performance trends. For example, consider the performance of the Ocean simulation with input size 130 in Figure 3. The cost model accurately predicts that little will be gained by using four PCs rather than two, and that performance will severely degrade when using eight PCs. Similarly, the cost function accurately predicts that the performance of the NEC Cenju

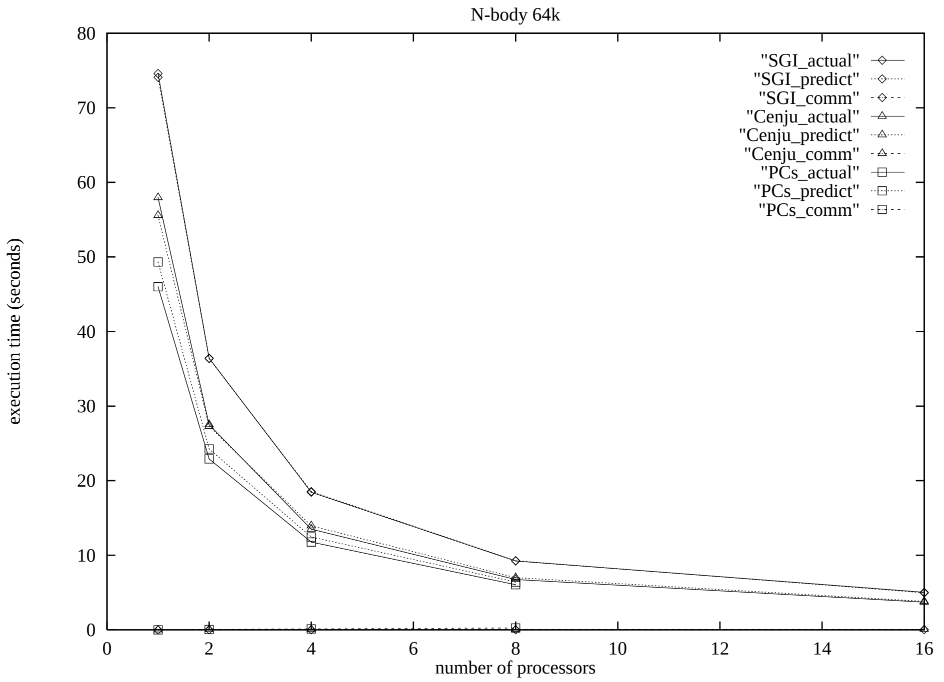

Figure 1: Actual and predicted times and predicted communication times (including synchronization) for N-body (size 64k). Experiments such as these are useful for demonstrating efficiency (our primary objective), but are less useful for determining the utility of the BSP cost model.

on this application will not improve much by using more than four processors on this input size. Note that the accuracy of the cost function depends of course on the choices made in the implementation of our BSP library. Thus, inaccuracies in the prediction may also be due to shortcomings of the library implementation, rather than the BSP cost function itself.

The rest of the paper is arranged as follows. Cautionary statements concerning the evaluation of our results are in Section 2. Section 3 describes the BSP model. Some related work is described in Section 4. The Green BSP Library is presented in Section 5. Implementations of the Green BSP Library on several parallel platforms are described in Section 6. Our applications are considered in Section 7. Finally, Sections 9 and 10 contain some concluding remarks and directions for future research.

2 Caveats

Before proceeding, we mention some caveats the reader should keep in mind when evaluating our data.

- We report our speed-up numbers in terms of the ratio of the parallel runtime and the runtime of the *same program* on a single processor. Viewing this definition of speed-up as a performance gain assumes that the single processor code is a reasonable sequential program. We believe that for most of our applications this is the case. For matrix multiplication, however, many highly optimized sequential codes exist, and thus our speed-ups should be interpreted cautiously. Some performance gains are also possible for the Barnes-Hut implementation. Writing high-performance sequential codes for these applications on modern workstations can be a challenging and time-consuming task, and the best performance is often achieved by optimizing for a particular machine configuration.

- Several of our results exhibit superlinear speed-up. We define the total work to be the sum of the run times spent on doing computational work across all processors; in the case of superlinear speed-up, the total work on p processors is less than the total work on one. Superlinear effects may occur if the problem size is such that the data fits in the main memory (or cache) of p processors while it cannot fit in the main memory (or cache) of one processor. It is also possible that a program written for p processes but running on a single processor may order operations such that it exploits more data and instruction locality than a program written for one process. Though we limit our problem sizes so that they fit into the main memory of one processor, superlinearity due to caching may still occur. We discuss this issue in more detail in Section 7.

- Part of our objective is to examine the predictive capability of the BSP cost function. We consider BSP to model only communication and synchronization. I/O and local computation are not modeled. As a result, none of our experiments include I/O.

- One would like to compare results using the BSP model with results obtained by using other models, or by programming directly for a particular machine. While we compare our Ocean and N-body applications with shared-memory implementations, we warn the reader that detailed comparisons will not be found in this work. We hope that our applications can be used as a basis for future research along these lines.

- The machines used for this paper all exhibit only a moderate level of parallelism (up to 16 processors). Promising initial results have been obtained for experiments on machines with significantly more processors. In particular, Green BSP Libraries have been implemented on a 54-processor BBN Butterfly GP1000 [32] and a 8,192-processor Maspar [52]. We have also used the MPI version of our library to run larger N-body simulations on the NAS IBM SP2 and a large Cray T3E at NERSC.

3 The BSP Model

It is well-known that scalable parallel performance can be achieved for many problems by using machine-dependent software. Similarly, portability can be achieved if scalable performance is sacrificed. The simultaneous achievement of portability and efficiency, however, is still a challenging problem in practice, despite the fact that no real theoretical impediments exist that would preclude this. In a noteworthy paper, Valiant argued that what is missing is a *bridging model* that serves as an interface between "the diverse and chaotic world of software [and the] diverse and chaotic world of hardware" [57]. Valiant proposed the Bulk-Synchronous Parallel (BSP) model as a candidate for this role and gave theoretical arguments in its support.

3.1 The Model

In the BSP model, a parallel machine consists of a set of processors, each with its own local memory, and an interconnection network that can route packets of some fixed size between processors. The computation is divided into *supersteps*. In each superstep, a processor can perform operations on local data, send packets, and receive packets. A packet sent in one superstep is delivered to the destination processor at the beginning of the next superstep. Consecutive supersteps are separated by a barrier synchronization of all processors.

The communication time of an algorithm in the BSP model is given by a simple cost function. The three basic parameters that model a parallel machine are: (i) the number of processors p, (ii) the *gap* g, which reflects network bandwidth on a per-processor basis, and (iii) the *latency* L, which is the minimum duration of a superstep, and which reflects the latency to send a packet through the network as well as the overhead to perform a barrier synchronization.

Consider a BSP program consisting of S supersteps. Then the execution time for superstep i is given as:

$$w_i + gh_i + L \tag{1}$$

where w_i is the largest amount of work (local computation) performed, and h_i the largest number of packets sent or received, by any processor during the ith superstep. The execution time of the entire program is:

$$W + gH + LS \tag{2}$$

where $W = \sum_{i=0}^{S-1} w_i$ and $H = \sum_{i=0}^{S-1} h_i$. We call w_i and W the *work depths* of the superstep and the program, respectively.

Thus, efficient programming of a BSP machine is based on several simple principles. To minimize the execution time, the programmer must (i) minimize the work depth of the program, (ii) minimize the maximum number of packets sent or received by any processor in each superstep, and (iii) minimize the total number of supersteps in the program. In practice, these objectives can conflict, and trade-offs must be made. The correct trade-offs can be selected by taking into account the particular g and L parameters of the underlying machine.

3.2 Discussion of the Model

By examining the BSP cost function, it is clear that efficient parallelism can be achieved under certain conditions. Assume a parallel program performs the same amount of computation as the corresponding sequential program, and that computation is balanced among the p processors. In particular, if the rate of growth of the work depth (W) exceeds that of both communication (H) and synchronization (S), close to optimal efficiency can be achieved by making the input size sufficiently large. This has led to the design of so-called *one-optimal* BSP algorithms—algorithms that are within a factor of $1 + o(1)$ of optimal time [6, 23, 24]. Many important applications meet the conditions that guarantee efficiency. In practice, this approach for efficient BSP computation is related to Gustafson's view of parallel speedup [34]—to achieve efficiency, use large problem sizes. An example of this phenomenon can be seen by observing Figures 2 to 5. As the problem size increases for

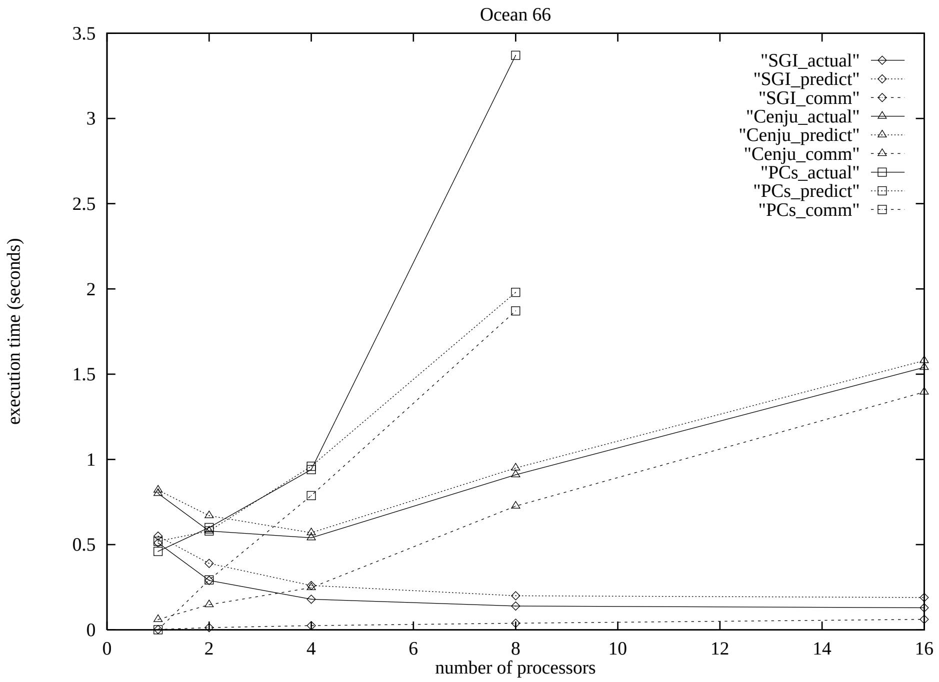

Figure 2: Ocean 66—Actual times, predicted times, and predicted communication times (including synchronization).

the Ocean application, the relative expense of communication and synchronization decreases and the attainable speedup improves.

More generally, it appears that the efficient execution of many abstract programming models, including BSP, depends on the existence of a sufficient degree of parallel slackness, often in excess of that required by machine-dependent solutions. (We define parallel slackness informally as the ratio of the degree of parallelism in the problem to the number of processors.) On the other hand, it can be argued that for many problems increasing the input size to the point of efficiency will eventually become unrealistic as the number of processors increases, due to the resulting increase in the overall execution time (e.g., see [55]). Underlying this argument, however, is an assumption that as we increase the number of processors, the power of each individual processor stays the same. It is important to realize that as the speed and memory size of today's processors continue to increase rapidly, we will be able to run larger and larger problem sizes, which in turn should allow us to efficiently use more and more processors[1]. Thus, we believe that in today's parallel machines—which are mostly based on commodity processors—increases in sequential processor speed have become an ally, rather than adversary, of efficient parallelism. In particular, we expect that portable parallel programming will become feasible on larger numbers of processors.

In support of the BSP model, it has been shown that many other programming styles can be automatically and efficiently transformed into a BSP style. In particular, Valiant [57, 58] and Gerbessiotis and Valiant [24] have shown that the BSP model can efficiently simulate the EREW PRAM. This result was subsequently extended to the more powerful QRQW PRAM

[1]This assumes that the values of L and g, and thus the ratio of communication to computation speed, remain about the same.

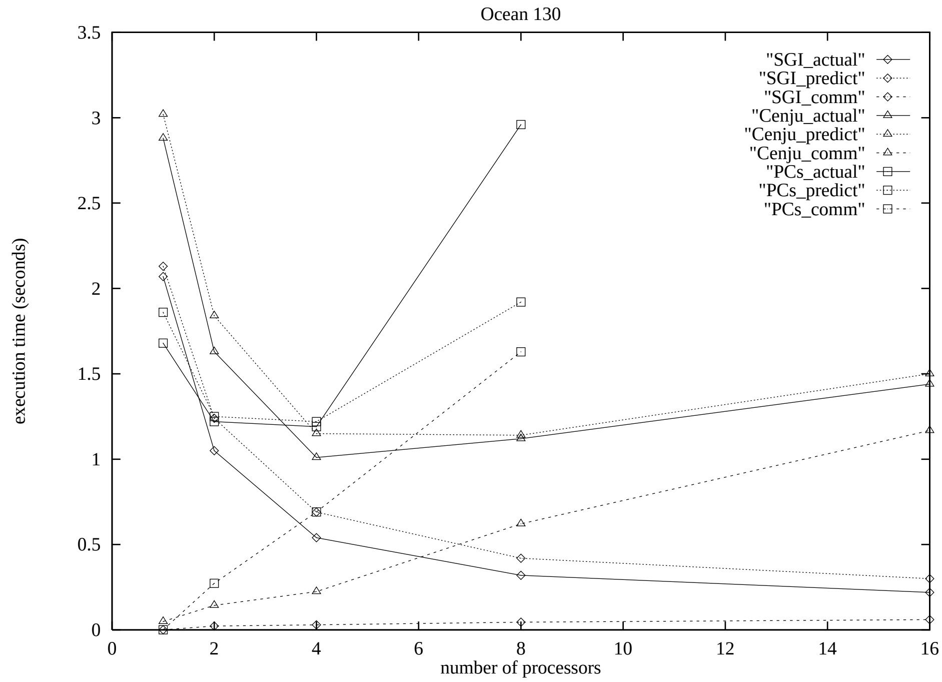

Figure 3: Ocean 130—Actual times, predicted times, and predicted communication times (including synchronization).

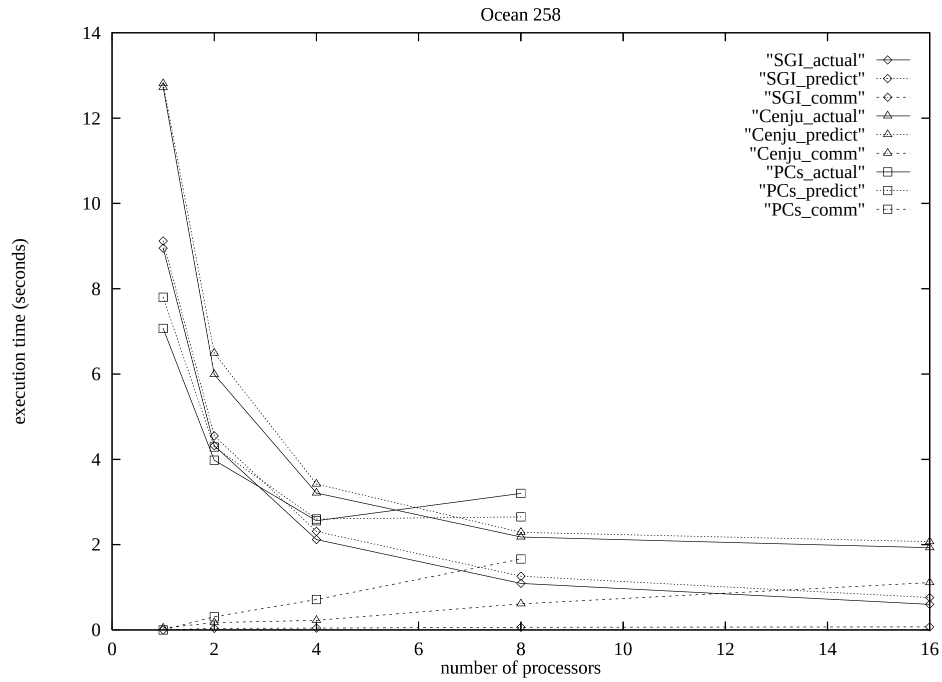

Figure 4: Ocean 258—Actual times, predicted times, and predicted communication times (including synchronization).

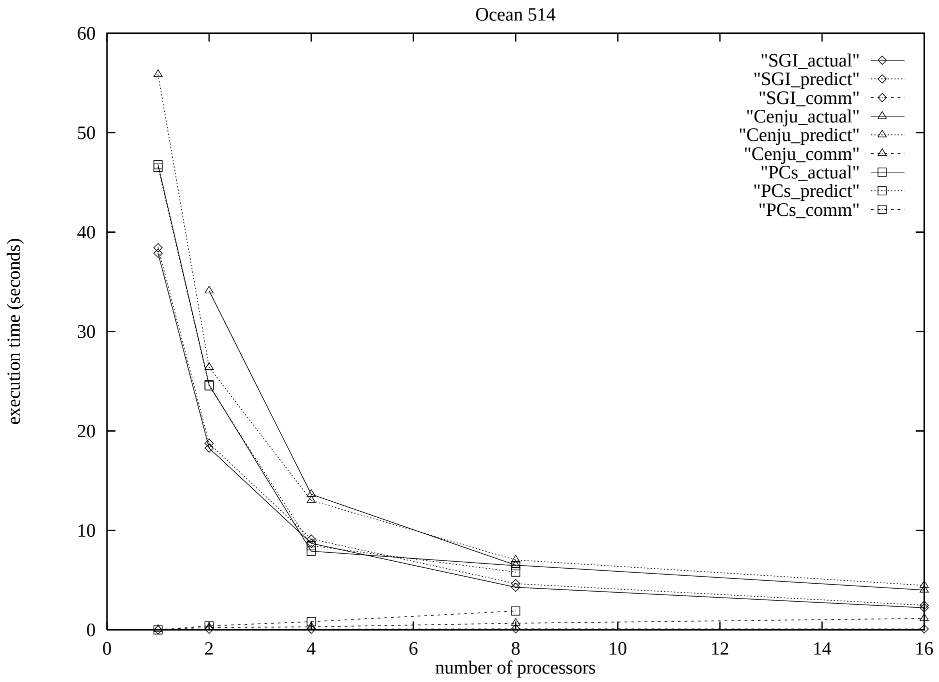

Figure 5: Ocean 514—Actual times, predicted times, and predicted communication times (including synchronization).

model by Gibbons, Matias, and Ramachandran [27]. A simulation of asynchronous message-passing programs in BSP is given in [8]. Finally, Gibbons, Matias, and Ramachandran have shown an emulation of the recently proposed QSM (Queued Shared Memory) model [28]. Of course, as Gerbessiotis and Valiant [24] point out, a direct implementation on the BSP model will often lead to even better performance.

We briefly discuss a few more aspects of the BSP model. First, the BSP model views the interconnection network as a batch-routing network that can efficiently route arbitrary balanced communication patterns. The model ignores the particular network topology of the underlying machine. Hence, the model only considers two levels of locality: local (inside a processor) or remote (outside a processor).

Second, we note that the BSP model requires complete cooperation among all processors to route even a single message. While this may seem an unnatural restriction, we argue that it is appropriate. As stated above, Valiant and others have made numerous theoretical arguments that parallel programming need not be optimized at the single-message level. Moreover, in the context of interconnection networks, one can often achieve better bandwidth when routing large batches of messages rather than individual messages.

In contrast, asynchronous models seem to encourage the programmer to design and optimize their code with respect to the arrival of single messages. Thus, it is contingent upon the architect to attempt to minimize single-message latencies. The requirement of synchrony in the BSP model also contributes to its overall simplicity. As a result, we feel that it is fundamentally easier to reason about the correctness and performance of BSP programs, as opposed to aggressively asynchronous message-passing programs.

Finally, as the BSP model emphasizes the efficient routing of large h-relations, it appears to be particularly suitable for emerging "ultra-high-latency" environments such as meta-computers consisting of several clusters of PCs connected by wide-areas networks, or for parallel and sequential computations that process large data in secondary memory. In fact, one of the codes described in this paper (the Barnes-Hut N-body code) was recently adapted and optimized by colleagues to run on the Albatross wide-area cluster, where it has achieved high performance even for very high latencies [51]. For computing in secondary memory, several simulation results have been established that show that BSP algorithms that tolerate high-latencies can also be used for efficient computing in secondary memory [21, 53]. In both of these cases, certain adjustments and optimizations will be needed to make the model really practical. Nonetheless, we believe that many of the principles and properties of the basic BSP model will also prove valuable in these new environments.

4 Related Work

This section gives a brief overview of related work. We describe other work on the BSP model, including some proposed extensions to the model. We also give a critical discussion of some alternative candidates for a unifying model for parallel computing, and refer to other libraries and programming languages for portable parallel computing. Due to the immense amount of work in these areas, we have to restrict ourselves to work that is very closely related to our own, or that directly influenced our approach and design decisions.

4.1 BSP Algorithms, Libraries, and Languages

Since the introduction of the BSP model, a number of papers have considered the design and analysis of BSP algorithms—see, for example, [7, 9, 23, 24, 48, 58].

Several groups of researchers are currently studying the use of the BSP model on existing parallel machines. The Oxford BSP Library, developed by Miller [49] while at Oxford University, is based on shared-memory operations similar to those found in the Cray SHMEM library. This makes the library very simple and efficient to implement on shared-memory machines. Oxford BSP is restricted, however, by the fact that only statically allocated memory can be accessed by other processors. Nonetheless, this approach is powerful enough for many static computations that arise in scientific computing. In contrast, the Green BSP Library described in this paper is based on message passing, which requires the programmer to prepare and read messages. Thus, we believe that the Green BSP Library is better suited for the irregular and dynamic applications that we have experimented with.

Also at Oxford University, McColl's group is working on the development of several BSP programming languages, tools, and industrial applications [35, 39, 41, 47].

A group at Harvard University lead by Cheatham and Valiant is studying higher-level programming languages and compilation techniques for the BSP model [16, 17]. Bisseling at the University of Utrecht is studying the use of the BSP model in the implementation of scientific computations [9, 10]. A recent implementation of a plasma simulation using the Oxford BSP Library is described in [50].

Finally, BSP researchers have proposed a standard BSP library [36], which will incorporate much of the previous experimental work in this area, including the work reported in this paper.

4.2 Extensions to the BSP Model

Gerbessiotis and Valiant [24] examined several extensions of the BSP model, including special mechanisms for parallel-prefix computation (PPF-BSP), broadcasting (b-BSP), and concurrent reads and writes (c-BSP). Their overall conclusion is that these extensions provide only modest improvements in efficiency, though in some cases they achieve one-optimality for smaller input sizes or larger latency values.

The BSP* model of Bäumker, Dittrich, and Meyer auf der Heide [7, 5, 6] introduces an additional parameter B that specifies the minimum length of a message, thus rewarding the programmer for sending large messages. This means that the cost of an h-relation depends not only on the value of h, but also on the structure of the h-relation. (In particular, the BSP* model will assign a a higher cost to small h-relations in which a processor exchanges packets with many other processors, and a lower cost to small h-relations where all h packets are sent to the same destination.)

The dx-BSP ("deluxe BSP") model proposed in [12] attempts to model the performance of high-bandwidth shared-memory machines such as the Cray C90 in which the memory banks are significantly slower than the processors.

Finally, the Extended BSP (E-BSP) model [37] provides a more accurate cost function for unbalanced communication in networks where the primary bottleneck is not at the processor-network interface. A comparison of the BSP and E-BSP models on several machines and applications is given in [38]. However, the programs in this study are written in PVM, MPL, and Split-C, which were not designed with efficient BSP computation in mind. Our approach, on the other hand, is based on the belief that efficient and portable BSP computation requires a careful implementation of the basic BSP functions, although it is an interesting question to what degree the BSP cost function can also be used to predict the performance of bulk-synchronous-style programs executed on other platforms.

4.3 Other Cost Models

A number of other models for general-purpose parallel computing have been proposed in recent years—see [46] for an overview. In the following, we only mention those models that are most closely related to BSP.

LogP model [20] is based on asynchronous message passing. It measures the performance of point-to-point messages with three parameters representing software overhead, network latency, and communication bandwidth. The LogP model has been used as a performance model for active messages [59] and the Split-C language [18], and it has been applied to the analysis of several application programs [19, 45]. A theoretical comparison of the BSP and LogP models can be found in [8]. This study concludes that the two models are substantially equivalent in terms of asymptotic analysis. While the LogP model may be valuable for modeling the behavior of current asynchronous message-passing layers and low-level communication routines (such as broadcasting or prefix computations), it seems that applying the model to more complex parallel programs is often quite difficult. We thus believe that the BSP model is preferable for designing and analyzing parallel application programs, due to its extreme simplicity. For example, the BSP model frees the programmer from concerns such as scheduling messages to avoid endpoint contention, or choosing the right message size to avoid large software overheads.

Other closely related models are the Postal Model [3], the Atomic Model [43], and several models for end-point contention (e.g., see [2]) inspired by the prospect of optical communication in parallel machines. Like BSP and LogP, these models do not refer to the topology of the underlying machine, but assume that the interconnection network behaves essentially like a completely connected network, with the only contention arising at the processor-network interface.

A shared-memory model that is closely related to BSP is the Queued Shared Memory (QSM) model recently proposed by Gibbons and Matias [28]. This model can be seen as a generalization of the QRQW PRAM model [27, 26] that incorporates the g parameter of the BSP and LogP models in order to model shared-memory platforms with limited communication bandwidth.

4.4 Standardized Message-Passing Libraries

A somewhat different approach to portable parallel programming is based on standardized message-passing libraries such as PVM [22] and MPI [33]. While these libraries provide a common set of functions on a variety of parallel machines, they do not offer any cost function (in the strict sense) that could guide the programmer in the design of efficiently portable code. It seems that the very idea of these libraries is to offer a fairly rich set of functions, including various collective operations, each of which can be optimized with respect to the underlying architecture. This rules out any simple cost model based on just a few parameters, whereas the BSP and LogP models assume a very small set of basic functions and—at least in theory—require any other operations to be implemented on top of these functions.

4.5 High-level Programming Models and Languages

One of the most popular approaches to parallel programming has been the use of the shared-memory model. While it can be argued that shared memory is a natural and user-friendly approach to parallel programming, it also seems to have serious limitations in terms of scalability. The basic impediment is that most shared-memory machines assume a view of the hardware where all memory accesses have the same cost. As stated before, if there is sufficient communication bandwidth, then the BSP model can efficiently execute programs written for various shared-memory models. However, the BSP model also allows efficient solutions to many problems in the case of large values of g and L, by using the locality inherent in these problems.

In addition, hardware support for *sequential consistency*, the most straightforward shared-memory model, becomes more inefficient as the number of processors increases. To overcome this problem, a number of other models for shared-memory consistency have been proposed [25]. However, these weaker and usually more complicated consistency models can hinder both programmability and portability.

High-Performance Fortran (HPF) is a parallel extension of Fortran that has received considerable attention over the last few years [1]. HPF allows a fairly easy implementation of scientific computations with regular data and communication structures. The main disadvantage of HPF is that it provides limited support for implementing adaptive algorithms.

The NESL language of Blelloch [13, 14] is an example of a data-parallel language that attempts to overcome these shortcomings of HPF, while retaining a large degree of efficiency. NESL also provides a simple cost model that is based on the total work and depth of a computation. The NESL model is not processor based, and thus there is no notion of processor locality. As a result, efficient computation with NESL is currently restricted to parallel machines with high communication bandwidth.

5 The Green BSP Library

The Green BSP Library is a set of functions for BSP programming. In this section, we give a brief description of the library design and the semantics of the functions.

The Green BSP Library is designed to be as simple and as portable as possible without sacrificing basic BSP functionality. Information is transferred through the use of fixed-length packets. The size of the packets is set at compile time. Program text is in a SPMD (Single-Program, Multiple-Data) format. The library makes no particular effort to provide a user-friendly programming environment, nor does it make concessions to improve efficiency on specific parallel platforms. Despite these self-imposed restrictions, we claim the the Green BSP Library provides a relatively easy-to-understand parallel programming environment and that it will run with reasonable efficiency on almost any type of parallel system.

The library consists of the seven functions shown in Table 1. For a BSP computer, the basic tasks of the interconnection network are to provide point-to-point packet delivery and barrier synchronization. In the Green BSP Library the functions that implement these tasks are called fundamental functions. In Table 1 the first three functions are the fundamental functions. The last four functions are less central to the BSP philosophy and are called supplemental functions.

Supplemental functions have been included in the library sparingly. As stated earlier, it is not our purpose to develop a user-friendly environment—presumably, such an environment would have several other supplemental functions that could be useful in certain contexts. For a supplemental function to be included in the Green BSP Library it had to be extremely useful, efficient, and simple to implement.

We begin by discussing the fundamental functions. To sending of packets, we use:

```
void bspSendPkt(int pid, const bspPkt *pktPtr);
```

The address of the destination process is `pid`, and the pointer to the packet to be sent is `pktPtr`.

To receive a pointer to a packet that was sent in the previous superstep, the following function is provided:

```
bspPkt *bspGetPkt(void);
```

If there are no more packets to be accessed, this function will return the symbolic constant `NULL`. The returned pointer is only guaranteed to be valid during the current superstep.

Barrier synchronization requires the use of the following function:

Function	*Semantics*
`void bspSendPkt()`	send packet to another process
`bspPkt *bspGetPkt()`	receive packet sent in the previous superstep
`void bspSynch()`	perform barrier synchronization
`int bspGetPid()`	return pid
`int bspGetNumProcs()`	return number of processes
`int bspGetNumPkts()`	return number of packets sent to the process in the previous superstep that have not yet been accessed
`int bspGetNumStep()`	return number of current superstep

Table 1: The seven functions that constitute the Green BSP Library. The first three functions are classified as *fundamental* functions while the last four functions are classified as *supplemental* functions.

```
void bspSynch(void);
```

After a process returns from a `bspSynch()` call, it can begin to access the packets that were sent to it in the previous superstep by utilizing `bspGetPkt()`.

The supplemental functions can be used to obtain the process id:

```
int bspGetPid(void);
```

the number of processes:

```
int bspGetNumProcs(void);
```

the number of packets awaiting access in the incoming-packet buffer:

```
int bspGetNumPkts(void);
```

and the number of the current superstep (the initial superstep being superstep 0):

```
int bspGetNumStep(void);
```

To demonstrate the programming style supported by the Green BSP Library, a short toy program is presented in Figure 6. This program demonstrates the basic BSP functionalities of packet sending and receiving as well as barrier synchronization.

Note that the program is written at a very low level. The packets are stuffed and unstuffed using `memcpy()`, thus the programmer must know that the size of an integer is (in this example) four bytes long. The program further assumes that all processes use the same representation for integers. It is possible to allow for greater flexibility by creating system-dependent utility functions on top of the Green BSP Library that ensure a common representation for all relevant data structures.

Although it is not an issue in this example, the programmer must remember that packets can arrive in arbitrary order. Thus, packets must have sufficient labeling information for proper utilization by the destination process. Since there is no implicit labeling information that arrives with the packet, programmers must explicitly decide the packet format. Of course, even in one superstep there may be several distinct packet formats in use.

6 Library Implementations

The Green BSP library has been implemented on a number of platforms. The results in this paper are based on the following library versions and parallel machines:

- a shared-memory version, used on an SGI Challenge with sixteen MIPS R4400 processors (SGI),
- an MPI version, used on an NEC Cenju consisting of sixteen MIPS R4400 processors connected by a multi-stage network, with a peak bandwidth of 20 Mbytes/s available for each processor (Cenju), and
- a TCP version, used on a system of eight 166-MHz Pentium PCs running Linux and connected by a 100-Mbps Ethernet switch (PC-LAN).

Following we give a brief description of each of the three library implementations used in this paper, and then analyze the library performance in terms of the BSP parameters L and g.

6.1 The Shared-Memory Version

In the shared-memory implementation, each process has two large input buffers in shared memory, which are used in alternating supersteps.[2] Because the input buffers have many writers, they are protected by locks. However, when a process acquires a lock it allocates enough space for 1000 packets, so the locking cost is small per packet. Also, because the locks are used infrequently, we were able to use Lamport's software locking algorithm, which is tuned for the case of low contention. There is one case that probably would generate substantial lock contention: supersteps with small all-to-all communication patterns. To eliminate this case we begin each superstep by pre-allocating p memory blocks (one for each writer) at the start of each input buffer. With this scheme, the locks are only used when there is actually enough communication to pay for them.

Note that, unlike the MPI and TCP implementations, which synchronize implicitly via their all-to-all communication patterns, the shared-memory version requires explicit synchronization at superstep boundaries. We accomplish this using p variables in shared memory that are incremented by the processes to indicate that they are ready to proceed to the next superstep. Process 0 then spins on variables 1 through $p-1$, while processes 1 through $p-1$ spin on variable 0.

[2]The processes themselves run in separate address spaces.

```
void program(void)
{
  int pid, numProcs, A, B, C;
  bspPkt pkt, *pktPtr;

  pktPtr = &pkt;

  pid = bspGetPid();                      /* Get process ID */
  numProcs = bspGetNumProcs();            /* Get number of processes */

  if (pid == 0) {A = 3; B = 12;}          /* Initialize A and B */
  if (pid == 1) {A = 1; B = 18;}
  if (pid == 2) {A = 5; B = 7;}

  memcpy((void *)pktPtr, (void *)&A, 4); /* Store A in packet */
  bspSendPkt((pid+1)%numProcs, pktPtr);  /* Send packet to neighbor */

  bspSynch();                             /* Superstep synch */

  pktPtr = bspGetPkt();                   /* Receive packet */
  memcpy((void *)&C, (void *)pktPtr, 4); /* Put packet value in C */

  C = C + B;                              /* Calculate final C */

  fprintf(stdout, "Process %d, C = %d\n", pid, C);

  bspSynch();                             /* Superstep synch */
}
```

Figure 6: A sample program using the Green BSP Library. The program is designed for three processes. Each process starts with initial values for integers A and B. Each process sends the value of A to its neighbor process at pid + 1, where the addition is modulo the number of processes. Finally, each process calculates and prints its own value for C, equal to its original value for B plus its neighbor's value for A.

6.2 The MPI Version

In the MPI version, each process has a distinct input and output buffer for each of the other processes. There is no overlap of computation and communication—during a superstep, messages are simply read from and written to the appropriate buffers. When a process reaches a superstep boundary, it posts an `Irecv` for each input buffer and an `Isend` for each output buffer, and then waits until all $2p$ incoming and outgoing transmissions are completed, before starting the next superstep.

6.3 The TCP Version

As in the MPI version, each process uses a distinct input and output buffer to communicate with each of the other processes, and communication only occurs at superstep boundaries. The blocking TCP protocol we employ requires receivers to actively empty the pipe whenever another process sends a large amount of data, so deadlock could occur if we are not careful in scheduling the communication. In our setup, the processes pair off and talk according to a pre-computed $p-1$ stage total-exchange pattern. Note that while this rigid scheduling method works well for random h-relations, it is not efficient for certain worst-case communication patterns. We ran this version on a system of eight PCs connected by a 100-Mbps Ethernet switch that allows the $p/2$ conversations in each communication stage to occur in parallel. As it turned out, the maximum bandwidth that we were able to obtain between two processors was about 5 MB/s, and thus significantly below the 12.5 MB/s restriction of the Fast Ethernet connections. (We conjecture that this is due to some bottleneck in the operating system.)

6.4 Library Performance

Figure 2 shows the values of L and g achieved by the different versions of our library. Synthetic benchmarks were used to measure these values. The value for L corresponds to the time for a superstep in which each process sends a single packet; this incorporates both the message latency and the barrier synchronization overhead. The bandwidth parameter g is the time to route a large, balanced h-relation divided by h. These benchmark h-relations were randomly generated.

Looking at the entries in Figure 2, we can see that L grows linearly with the number of processors, due to our implementation choices. Of course, some changes in the implementations would be necessary to achieve acceptable values of L on significantly more processors. However, for the fairly small machines used in this study, it seems difficult to get significant improvements in this area.

7 Applications

For each of our applications, we ran experiments on four or five different input sizes and numbers of processors. In this section, we give a brief description of each application, and summarize the results of our experiments. A brief overview of the performance results is shown in Tables 3, 4, and 5.

p	SGI g (μs)	SGI L (μs)	Cenju g (μs)	Cenju L (μs)	PC-LAN g (μs)	PC-LAN L (μs)
1	0.77	3	2.20	130	0.92	2
2	0.82	16	2.20	260	3.30	540
4	0.88	29	2.20	470	4.80	1556
8	0.97	52	2.50	1470	8.60	3715
9	1.00	57	2.70	1680	—	—
16	0.95	105	3.60	2880	—	—

Table 2: BSP system parameters.

app	size	SGI (p=16) time (s)	SGI (p=16) spdp	Cenju (p=16) time (s)	Cenju (p=16) spdp	PC-LAN (p=8) time (s)	PC-LAN (p=8) spdp
Ocean	514	2.23	17.0	4.0	13*	6.46	7.2
N-body	64k	5.04	14.8	3.72	15.6	6.06	7.6
MST	40k	0.40	15.8	0.56	10.1	0.65	4.2
SP	40k	0.26	9.7	0.48	5.3	0.59	2.6
MSP	40k	4.71	9.4	3.68	12*	4.88	7.1
Matmult	576	2.42	11.4	2.31	13	—	—

Table 3: Speedup summaries for large problem sizes; * indicates an estimate on the speedup as we were unable to run the largest problem size on a single processor.

app	size	time (s)	seq. work p=16 (s)	seq. work p=1 (s)	spdp w.r.t. p=16
Ocean	514	2.23	35.43	38.43	15.9
N-body	64k	5.04	70.06	74.08	13.9
MST	40k	0.40	3.92	6.3	9.8
SP	40k	0.26	1.88	2.54	7.2
MSP	40k	4.71	39.57	44.36	8.4
Matmult	576	2.42	31.21	27.53	12.9

Table 4: For all applications except Matmult, the measured total sequential work of the 16-processor SGI program was less than that of the 1-processor SGI program.

app	size	SGI time (s)	SGI pred (s)	SGI W (s)	H	S
Ocean	514	2.23	2.48	2.38	69946	312
N-body	64k	5.04	4.97	4.95	24661	6
MST	40k	0.40	0.34	0.32	9562	62
SP	40k	0.26	0.28	0.26	2820	101
MSP	40k	4.71	3.64	3.58	39874	138
Matmult	576	2.42	2.09	1.97	124416	7

Table 5: Algorithmic and model summaries for large problem sizes on 16-processor SGI system.

Table 3 shows speedup results for large input sizes, for each application and system. To obtain meaningful values for speedup, we limit the problem sizes so no swapping to disk is necessary. The speedup results are usually stated as the ratio of single-processor time and parallel time. In two cases, we were unable to run the relevant problem size on a single processor; here we give estimates of the speedup.

In analyzing the performance of our algorithms we noticed that the total sequential work (i.e., local computation) performed by the 16-processor programs on the SGI was typically less than the total work performed by the single-processor programs. For this reason, we also include the relative speed-ups with respect to the total sequential work on 16 processors in Table 4.

In Table 5, we provide some data about the abstract BSP performance of our applications. We also provide the algorithmic parameters, including the work depth (as measured on the SGI), the sum over all supersteps of the maximum number of packets sent or received by any processor, and the number of supersteps. We also include the actual running times and predicted running times using the BSP model, where the values for L and g are taken from Table 2.

The work depth W and the total work of the parallel programs were computed by simulating the parallel computation on a single processor using an IPC shared-memory implementation of our library. Initially, we had considered using a single constant factor to translate the measured work depths of the SGI to estimated work depths for the Cenju. A different constant factor would be used to estimate the work depth of the PC-LAN. Unfortunately, this approach proved to be insufficient—the estimated work depths were often far from their actual values. In short, we were unable to use a constant factor to parameterize the relative speed of local computation on a platform. To provide better estimates for work depths, therefore, we used a different constant factor for each (application, input size) pair. For example, if Ocean 130 on one processor took time A on the SGI, time B on the Cenju, and time C on the PC-LAN, the factor B/A was used to translate SGI work depths to Cenju work depths and the factor C/A was used to translate SGI work depths to PC-LAN work depths.

In some of our applications, our approach introduced systematic errors that produced high predicted running times. That is, the work depth is in some cases more than the actual parallel execution time. We point out the applications where we believe these errors to occur in the discussion below.

In the following, we give a brief discussion of the applications. For each application, we describe its implementation, and discuss the resulting performance in terms of highlights, lowlights, algorithmic performance in the BSP cost model, and possible implications. We also discuss some additional experiments and analyses whose data was not included in the main part of this paper.

7.1 Ocean Simulation

We converted an ocean eddy simulation program from the Stanford Parallel Library for Shared Memory Applications (SPLASH) [56] to our BSP system. The program computes ocean eddy currents using a multigrid technique on an underlying grid; see [54] for details. The conversion to BSP was fairly straightforward, due to the fact that the SPLASH code for this application was already in a BSP style; the grid is partitioned among processors, the processors compute on their own portions of the grid, and the processors communicate exclusively at global synchronization points.

We remark that initial versions of the SPLASH codes were not in a BSP style. They were in a style more consistent with a shared memory approach for parallel computing; the entire grid was allocated in one data structure, and all processors computed and modified the common data structure. This method was found to be ineffective. The fact that the effective version is in a BSP style suggests that the BSP model is more appropriate for this application.

7.1.1 Discussion

Table 6 contains our results for the Ocean application. Not shown in the table is the fact the performance of the BSP Ocean code on the SGI matches that of the direct shared-memory SPLASH implementation for problem size 258. This may be seen as somewhat surprising given that we are using message passing on a shared-memory architecture. We believe this speaks well of our library implementation in particular and of the prospect of efficient BSP library implementations in general.

On the NEC Cenju, the Ocean code performs relatively poorly with 16 processors, except for the largest problem size, where it performs much better (perhaps nearly ideal; we only give a plausible lower bound in the table, as the problem was too large for a single processor). We suspect that this is due to the fairly large latency of the BSP implementation on the NEC Cenju, given that the BSP algorithmic data in Table 6 shows that the number of supersteps is quite large.

A surprising aspect of the Ocean program is that the number of supersteps actually decreases with increasing problem size. Thus, as the problem size increases, the latency overheads will become less significant at an even faster rate than one would normally expect in parallel computing. It can be hoped that the high-latency systems quickly “catch up” as the problem size grows. Our data shows that this occurs for both high-latency systems (8 processor PC-LAN and 16 processor NEC Cenju) at a problem size of 514.

app	size	p	SGI pred (s)	SGI time (s)	SGI spdp	Cenju pred (s)	Cenju time (s)	Cenju spdp	PC-LAN pred (s)	PC-LAN time (s)	PC-LAN spdp	SGI W (s)	H	S	SGI TWk (s)
Ocean	66	1	0.55	0.51	1.0	0.82	0.8	1.0	0.52	0.46	1.0	0.54	114	468	0.54
Ocean	66	2	0.39	0.29	1.8	0.67	0.58	1.4	0.58	0.6	0.8	0.38	12192	468	0.73
Ocean	66	4	0.26	0.18	2.8	0.57	0.54	1.5	0.96	0.94	0.5	0.23	12530	468	0.86
Ocean	66	8	0.2	0.14	3.6	0.95	0.91	0.9	1.98	3.37	0.1	0.16	15400	468	1.11
Ocean	66	16	0.19	0.13	3.9	1.58	1.54	0.5	—	—	—	0.13	13360	468	1.78
Ocean	130	1	2.13	2.07	1.0	3.02	2.88	1.0	1.86	1.68	1.0	2.12	91	379	2.12
Ocean	130	2	1.24	1.05	2.0	1.84	1.63	1.8	1.25	1.22	1.4	1.21	20762	379	2.36
Ocean	130	4	0.69	0.54	3.8	1.15	1.01	2.9	1.22	1.19	1.4	0.66	21034	379	2.46
Ocean	130	8	0.42	0.32	6.5	1.14	1.12	2.6	1.92	2.96	0.6	0.37	25700	379	2.68
Ocean	130	16	0.3	0.22	9.4	1.5	1.44	2.0	—	—	—	0.24	21316	379	3.28
Ocean	258	1	9.12	8.95	1.0	12.81	12.72	1.0	7.8	7.07	1.0	9.12	81	339	9.12
Ocean	258	2	4.55	4.32	2.1	6.49	5.99	2.1	4.29	3.98	1.8	4.51	38170	339	8.95
Ocean	258	4	2.31	2.12	4.2	3.42	3.21	4.0	2.6	2.56	2.8	2.27	38412	339	8.77
Ocean	258	8	1.26	1.09	8.2	2.29	2.18	5.8	2.65	3.2	2.2	1.2	46818	339	8.88
Ocean	258	16	0.76	0.6	14.9	2.07	1.93	6.6	—	—	—	0.68	37994	339	9.74
Ocean	514	1	38.43	37.87	1.0	53.85	—	—	46.51	46.77	1.0	38.43	72	312	38.43
Ocean	514	2	18.76	18.28	2.1	26.41	34.08	—	24.53	24.64	1.9	18.7	71688	312	37.24
Ocean	514	4	9.14	8.71	4.3	13.01	13.64	—	8.47	7.92	5.9	9.07	71912	312	35.62
Ocean	514	8	4.65	4.29	8.8	7.04	6.51	—	5.82	6.46	7.2	4.55	87226	312	34.83
Ocean	514	16	2.48	2.23	17.0	4.48	4.0	—	—	—	—	2.38	69946	312	35.43

Table 6: Data for Ocean application.

We note that our estimates for the total work of the Ocean program are systematically too high. In particular, the estimates obtained through the IPC single-processor simulation are actually higher than the actual running time of the code. Thus, our predicted times for the Ocean program are too high. We also ran additional experiments on the PC-LAN for this application that suggested that the total work of the parallel program goes down dramatically for the PC-LAN, while it does not for the SGI system. Thus, any observed speedup for the PC-LAN may have as much to do with this effect as with parallelism.

7.2 N-Body Simulation Using Barnes-Hut

The N-body problem is the problem of simulating the movement of a set of N bodies under the influence of a gravitational, electrostatic, or other type of force. The problem has numerous applications in astrophysics, molecular dynamics, fluid dynamics, and even computer graphics.

The N-body code in this study is based on the Barnes-Hut algorithm [4], which uses an irregular oct-tree structure, called BH tree, to hierarchically group bodies into clusters according to their distribution in three-dimensional space. The basic structure of our implementation is similar to those of Warren and Salmon [60] and Liu and Bhatt [44]. In particular, we use the ORB partitioning scheme to partition the bodies among the processors. Instead of repartitioning the bodies after each iteration as in [60], we only do so if the load imbalance reaches a certain threshold, as suggested in [44].

The positions of the bodies are updated in discrete time steps. In each step, the BH tree is first constructed locally inside each processor. Then appropriate subtrees, called "locally essential trees," are exchanged between every pair of processors, such that afterwards every processor has a local BH tree that contains all the data needed to compute the forces on its bodies, and whose structure is consistent with that of the global BH tree constructed in the sequential algorithm. More details about the implementation can be found in [11], which

app	size	p	SGI pred (s)	SGI time (s)	SGI spdp	Cenju pred (s)	Cenju time (s)	Cenju spdp	PC-LAN pred (s)	PC-LAN time (s)	PC-LAN spdp	SGI W (s)	H	S	SGI TWk (s)
N-body	1k	1	0.46	0.46	1.0	0.35	0.32	1.0	0.31	0.31	1.0	0.46	0	4	0.46
N-body	1k	2	0.24	0.24	1.9	0.18	0.18	1.8	0.17	0.17	1.8	0.24	824	6	0.48
N-body	1k	4	0.13	0.13	3.5	0.1	0.1	3.2	0.1	0.1	3.1	0.13	1798	6	0.5
N-body	1k	8	0.08	0.08	5.8	0.07	0.07	4.6	0.09	0.08	3.9	0.08	2360	6	0.56
N-body	1k	16	0.05	0.05	9.2	0.06	0.07	4.6	—	—	—	0.05	2530	6	0.68
N-body	4k	1	2.9	2.89	1.0	2.17	2.1	1.0	1.93	1.91	1.0	2.9	0	4	2.9
N-body	4k	2	1.45	1.43	2.0	1.09	1.02	2.1	0.98	0.97	2.0	1.45	2067	6	2.89
N-body	4k	4	0.75	0.75	3.9	0.57	0.54	3.9	0.53	0.54	3.5	0.75	4353	6	2.91
N-body	4k	8	0.41	0.4	7.2	0.32	0.3	7.0	0.34	0.32	6.0	0.4	5506	6	3.02
N-body	4k	16	0.3	0.25	11.6	0.26	0.22	9.5	—	—	—	0.29	6249	6	3.34
N-body	16k	1	15.38	15.42	1.0	11.54	11.64	1.0	10.25	9.86	1.0	15.38	0	4	15.38
N-body	16k	2	7.64	7.65	2.0	5.74	5.56	2.1	5.11	4.89	2.0	7.64	5700	6	15.22
N-body	16k	4	3.86	3.86	4.0	2.91	2.89	4.0	2.63	2.59	3.8	3.85	10692	6	14.79
N-body	16k	8	1.96	1.96	7.9	1.5	1.44	8.1	1.43	1.38	7.1	1.95	12235	6	14.95
N-body	16k	16	1.13	1.12	13.8	0.9	0.86	13.5	—	—	—	1.12	12100	6	15.37
N-body	64k	1	74.08	74.59	1.0	55.56	57.96	1.0	49.33	46.01	1.0	74.08	0	4	74.08
N-body	64k	2	36.37	36.42	2.0	27.3	27.52	2.1	24.26	22.92	2.0	36.35	15046	6	72.52
N-body	64k	4	18.54	18.45	4.0	13.95	13.5	4.3	12.46	11.78	3.9	18.52	25443	6	71.25
N-body	64k	8	9.27	9.23	8.1	7.01	6.75	8.6	6.41	6.06	7.6	9.25	26003	6	70.58
N-body	64k	16	4.97	5.04	14.8	3.82	3.72	15.6	—	—	—	4.95	24661	6	70.06
N-body	256k	1	344	345.59	1.0	258	—	—	229	212	1.0	344.43	0	4	344.43
N-body	256k	2	168	167.92	2.1	126	—	—	112	111	1.9	168.07	37493	6	333.3
N-body	256k	4	83	83.71	4.1	62	62.92	—	56	57	3.7	83.0	63321	6	322.04
N-body	256k	8	42	41.86	8.3	31	31.66	—	28	26.4	8.0	41.7	59251	6	318.65
N-body	256k	16	22	22.16	15.6	17	16.37	—	—	—	—	22.1	53422	6	316.38

Table 7: Data for N-body application.

also describes a parallel implementation of adaptive multipole methods in Green BSP based on a similar replication schemes.

Our implementation of the Barnes-Hut algorithm was strongly guided by the BSP model. In particular, our choice of the data structures and replication scheme was directly influenced by the emphasis the BSP model places on the efficient routing of large h-relations. Another helpful feature was the separation of latency and bandwidth in the BSP cost model. After fixing the basic replication scheme, and thus the number of supersteps, we were able to focus on minimizing the total amount of information transmitted during replication, without having to consider issues such as packet size or send overhead[3].

7.2.1 Discussion

Results for the N-body application are in Table 7. As input for our experiments we used the Plummer model generated by the SPLASH code [56]. The timing and speedup results in Tables 3 and 7 show that for large enough input sizes, the N-body code achieves nearly perfect parallel speedup on all three machines. Our implementation needs slightly larger input sizes than the SPLASH code to achieve the same speedup. However, even the largest input size in Table 7 is not overly large, given that simulations are currently performed with hundreds of thousands and even millions of bodies [60].

The running time of the single-processor version of our implementation on the SGI is slightly faster than that of the SPLASH code. In the code used in the above experiments, we did not fully optimize the computation of the interactions, which take around 97% of the

[3] Apart from the very small cost of the calls to the BSP library routines.

total sequential running time for a problem size of 16k on the SGI. Of course, doing this might increase the relative weight of the parallel overhead, and thus slightly decrease the resulting speedup.

Our N-body code performs only six supersteps per iteration, and its bandwidth requirements are fairly modest as we were careful to minimize the amount of data sent during the transmission of the "locally essential trees." This makes the program efficient even for fairly small problem sizes, while for larger input sizes the program achieves high performance even on platforms with very high latency. In fact, a slightly modified version of our code was recently shown to achieve high performance on a meta-computer consisting of several clusters of PCs located in different cities and connected by a wide-area network [51]. The application is irregular and dynamic, due to the uneven and changing positions of the bodies. However, the load distribution can be predicted fairly accurately from that of the previous iteration, as the system evolves only slowly.

7.3 Minimum Spanning Tree

The minimum spanning tree of a weighted graph G is the tree of minimum weight that contains all the nodes of G. In our parallel implementation, we assume that the input graph is initially partitioned among the processors. Each processor contains a data structure representing the portion of the graph for which it is responsible, and also a copy of each node in the graph that is connected to a node in its portion. The nodes for which a processor is responsible are called *home nodes* and the other nodes are called *border nodes*.

The algorithm is *conservative*[4] for the BSP model in that the number of messages communicated by any processor is at most the number of its border nodes. The program starts out with a completely local phase that computes the local components of the minimum spanning tree. The program then enters a parallel phase that uses a simplification of a conservative DRAM algorithm developed by Leiserson and Maggs [42]. Once the number of components becomes small, the program switches to a mixed parallel/sequential phase that first uses all the processors to find subforests of the remaining components using edges that are guaranteed to be in the minimum spanning tree, and then uses a single processor to assemble the forests into components. See [29] for more details.

The input graphs are generated as follows. Nodes are assigned uniformly at random to points on the unit square. Now construct a graph $G(r)$ on the nodes by adding an edge between all nodes within distance r. The graph G is $G(\delta)$ where δ is the minimum value such that $G(\delta)$ is a single connected component. The weight assigned to edge (u, v) is the distance between the points corresponding to u and v.

For this class of input graphs, the running time of the single-processor version of our parallel MST code is within 5% of a sequential implementation of Kruskal's algorithm on 10k-node graphs.

We reiterate that this algorithm was designed with the the BSP model very much in mind; First the algorithm is designed to fully utilize available sequential work which corresponds to bulking the computation. Second, the conservative methodology used in the parallel portion of the algorithm is demanded by the BSP design goal of having small and evenly balanced

[4] This concept was originally defined for the DRAM model [42].

			SGI			Cenju			PC-LAN			SGI			SGI
app	size	p	pred (s)	time (s)	spdp	pred (s)	time (s)	spdp	pred (s)	time (s)	spdp	W (s)	H	S	TWk (s)
MST	2.5k	1	0.1	0.1	1.0	0.1	0.1	1.0	0.1	0.08	1.0	0.1	3	12	0.1
MST	2.5k	2	0.08	0.07	1.4	0.09	0.09	1.1	0.1	0.08	1.0	0.08	666	30	0.15
MST	2.5k	4	0.06	0.05	2.0	0.07	0.09	1.1	0.12	0.09	0.9	0.05	1276	36	0.18
MST	2.5k	8	0.05	0.05	2.0	0.12	0.14	0.7	0.23	0.22	0.4	0.04	2224	46	0.26
MST	2.5k	16	0.07	0.18	0.6	0.24	0.25	0.4	—	—	—	0.06	3014	60	0.41
MST	10k	1	0.8	0.81	1.0	0.8	1.03	1.0	0.6	0.61	1.0	0.8	3	12	0.8
MST	10k	2	0.44	0.4	2.0	0.45	0.53	1.9	0.35	0.34	1.8	0.44	1377	30	0.85
MST	10k	4	0.23	0.2	4.0	0.25	0.27	3.8	0.24	0.22	2.8	0.22	3288	36	0.79
MST	10k	8	0.15	0.15	5.4	0.22	0.22	4.7	0.31	0.28	2.2	0.14	5302	42	0.92
MST	10k	16	0.13	0.19	4.3	0.3	0.3	3.4	—	—	—	0.11	5866	56	1.17
MST	40k	1	6.3	6.34	1.0	6.3	5.63	1.0	2.71	2.71	1.0	6.3	3	12	6.3
MST	40k	2	3.86	3.87	1.6	3.88	3.13	1.8	1.69	1.6	1.7	3.86	3163	36	7.46
MST	40k	4	1.2	1.1	5.8	1.22	1.38	4.1	0.61	0.92	2.9	1.19	6287	42	4.24
MST	40k	8	0.6	0.56	11.3	0.69	0.83	6.8	0.53	0.65	4.2	0.59	10335	52	3.91
MST	40k	16	0.34	0.4	15.8	0.53	0.56	10.1	—	—	—	0.32	9562	62	3.92

Table 8: Data for MST application.

communication patterns. Both aspects are crucial to good performance.

7.3.1 Discussion

Results for the MST experiment are shown in Table 8. This application is a fast computation (less than a second for the parallel code on the largest problem size). Thus, even a modest number of communication steps can figure significantly into the running time of the algorithm on high-latency systems. As a result of this, we once again obtain significantly better results for the low-latency SGI than the high-latency systems. Still we achieved a factor of four on the very-high latency 8-processor PC-LAN, and a factor of ten on the high-latency 16-processor Cenju.

Looking at the algorithmic data, we observe that the number of supersteps required for this computation grows quite slowly with the problem size. Furthermore, the total volume of communication is quite small relative to the computation costs for even the smallest problem size. That is, even for our worst machine the ratio between the total bandwidth cost and running time for the smallest problem size is less than a third, while for the largest problem sizes the ratio is less than an eighth. This suggests that we could perform MST computations on more highly connected graphs without much degradation in performance.

Finally, as discussed earlier, the good speedup results for the minimum spanning tree application on large input sizes shown in Table 3 should be qualified, since the total work for sixteen processors (3.9 seconds) is significantly less than the total work for a single processor (6.3 seconds). For the SGI, the 9.8 speedup shown in Table 4 is perhaps more reasonable than the 15.8 speedup shown in Table 3.

Thus, the best we can claim is about 70% of ideal speedup (despite the speedups reported in the table for the SGI). We argue that this is still quite good since our initial graph partitioning is only load-balanced to within about 10%, and the nature of the computation is quite dynamic.

7.4 Shortest Paths

A single source shortest paths computation on a weighted graph labels each node u with a distance label that corresponds to the length of the shortest path from u to the source. In our implementation, we assume that the input graph is initially partitioned in the same way as in the minimum spanning tree application. The class of graphs in our experiments is also the same.

We first implemented a naive parallel version of Dijkstra's algorithm, where each processor contains a priority queue of nodes whose distance labels have recently changed. Each processor proceeds by removing nodes from the priority queue and updating the neighbors as in Dijkstra's algorithm, until the priority queue is empty. Then each processor sends, for each home node whose distance label has changed, a message to any processor that contains that node as a border node, and ends its superstep. This process repeats until no node is entered into the priority queue during a superstep.

On noticing that this approach worked poorly, we redesigned the algorithm. We allowed a processor to communicate and end its superstep whenever it had worked on its local piece of the graph for some period of time called the work factor, rather than having it continue until it had absolutely no work left. This may lead to both better load balancing and quicker convergence. In any case, it leads to better performance.

The appropriate way to use this algorithm is to adjust the work factor according to the architecture (i.e., the work factor should grow with L). In our data, we chose one work factor to optimize performance across our platforms. That is, our numbers are for the exact same program and input on all of the architectures.

The BSP model was used heavily in both the initial design and the eventual algorithm. The key issues are the tradeoffs between communication frequency and load balancing. That is, an infinite work factor essentially minimizes communication frequency at some cost in load balance and perhaps also in the total work performed by the algorithm due to the lack of global information. A smaller work factor increases the communication frequency while improving the load balance and the total computation cost. The bulk computation/communication style of the BSP model is perfect for reasoning at this level.

7.4.1 Discussion

Table 9 contains the data from the SP experiments. For this application, the performance was limited by load-balancing issues for the low-latency systems and by synchronization costs for the high-latency systems.

For the single source shortest path problem, no efficient parallel algorithms are currently known; this was the reason for choosing a naive parallelization of the sequential algorithm. While our best speedup of 10 for a two-second long computation is not an embarrassment, one can question the scalability of this approach for shortest path computations in general. Also, since the sequential work again decreases with increasing numbers of processors, the reported speedups may be considered generous.

Still, we felt that this was an interesting first step towards the application of performing several shortest path computations on the same graph. Indeed, this algorithm does serve as the fine-grained inner loop of our next application.

			SGI			Cenju			PC-LAN			SGI			SGI
app	size	p	pred (s)	time (s)	spdp	pred (s)	time (s)	spdp	pred (s)	time (s)	spdp	W (s)	H	S	TWk (s)
SP	2.5k	1	0.06	0.07	1.0	0.06	0.07	1.0	0.04	0.05	1.0	0.06	4	8	0.06
SP	2.5k	2	0.05	0.04	1.8	0.07	0.05	1.4	0.06	0.06	0.8	0.05	244	50	0.09
SP	2.5k	4	0.04	0.03	2.3	0.07	0.05	1.4	0.12	0.12	0.4	0.04	399	59	0.09
SP	2.5k	8	0.04	0.03	2.3	0.16	0.15	0.5	0.34	0.63	0.1	0.03	883	83	0.13
SP	2.5k	16	0.05	0.1	0.7	0.33	0.31	0.2	—	—	—	0.04	1382	101	0.19
SP	10k	1	0.52	0.53	1.0	0.52	0.56	1.0	0.35	0.35	1.0	0.52	4	8	0.52
SP	10k	2	0.31	0.26	2.0	0.32	0.29	1.9	0.23	0.22	1.6	0.31	457	50	0.52
SP	10k	4	0.14	0.12	4.4	0.16	0.14	4.0	0.17	0.16	2.2	0.14	806	47	0.46
SP	10k	8	0.12	0.1	5.3	0.23	0.21	2.7	0.36	0.52	0.7	0.12	1407	74	0.51
SP	10k	16	0.09	0.12	4.4	0.32	0.3	1.9	—	—	—	0.08	1954	83	0.64
SP	40k	1	2.54	2.52	1.0	2.54	2.56	1.0	1.69	1.51	1.0	2.54	4	8	2.54
SP	40k	2	1.53	1.46	1.7	1.54	1.49	1.7	1.05	0.91	1.7	1.53	1308	56	2.53
SP	40k	4	0.82	0.75	3.4	0.85	0.81	3.2	0.66	0.7	2.2	0.81	1774	68	2.25
SP	40k	8	0.49	0.41	6.1	0.61	0.54	4.7	0.66	0.59	2.6	0.48	2198	86	2.01
SP	40k	16	0.28	0.26	9.7	0.56	0.48	5.3	—	—	—	0.26	2820	101	1.88

Table 9: Data for Shortest Path application.

7.5 Multiple Shortest Paths

In many situations, it is useful to perform a number of shortest path computations simultaneously. Examples are the all-pairs shortest paths problem (or a subset of all-pairs), the global routing phase in VLSI layout, and some graph-partitioning heuristics. Thus, we modified the code in the previous application to allow the computation of many shortest path trees simultaneously.

Here, one can use the same underlying (read-only) graph and keep data structures for each computation for the read-write data required in Dijkstra's algorithm. We note that the graph itself required $\Omega(|E| + |V|)$ storage, while the read-write data is $O(|V|)$, or more specifically, three integers and one double per node.

The bulk computation/communication style of the BSP approach leads to this implementation. For example, with our approach, each shortest path computation runs once on each process in each superstep. This can be contrasted to an asynchronous message passing algorithm that could switch between shortest path computations on the granularity of single message arrivals. While some advantage may be gained in the asynchronous approach, the complexity and overhead makes it seem unlikely.

7.5.1 Discussion

Results for MSP are in Table 10. In our experiments, we performed 25 shortest path computations simultaneously. We used the same work factor as in the shortest path experiments. The total sequential work decreased only slightly with increasing numbers of processors. Thus, our speedup numbers are mostly due to parallelism rather than computational advantages.

Our results for this experiment are particularly impressive for the PC-LAN considering the high latency of this system. We obtain a speedup of 7.1 on our 8-processor setup. Moreover, its raw performance is essentially the same as the 16 processor SGI system, while its cost is a fraction of the cost of the SGI system. This bodes well for the prospect of distributed data applications on networks of workstations.

			SGI			Cenju			PC-LAN			SGI			SGI
app	size	p	pred (s)	time (s)	spdp	pred (s)	time (s)	spdp	pred (s)	time (s)	spdp	W (s)	H	S	TWk (s)
MSP	2.5k	1	1.18	1.2	1.0	1.18	1.25	1.0	0.79	0.88	1.0	1.18	28	9	1.18
MSP	2.5k	2	0.81	0.74	1.6	0.83	0.74	1.7	0.58	0.62	1.4	0.8	4833	51	1.51
MSP	2.5k	4	0.52	0.46	2.6	0.57	0.48	2.6	0.49	0.47	1.9	0.52	7569	72	1.66
MSP	2.5k	8	0.43	0.45	2.7	0.57	0.48	2.6	0.68	0.67	1.3	0.42	9856	87	2.25
MSP	2.5k	16	0.33	0.47	2.6	0.64	0.58	2.2	—	—	—	0.31	10030	102	2.84
MSP	10k	1	8.93	8.9	1.0	8.93	9.95	1.0	5.94	7.02	1.0	8.93	28	9	8.93
MSP	10k	2	4.89	4.85	1.8	4.92	4.99	2.0	3.31	3.22	2.2	4.88	10265	57	8.52
MSP	10k	4	2.7	2.63	3.4	2.77	2.54	3.9	2.02	1.93	3.6	2.68	23467	78	8.72
MSP	10k	8	1.73	1.72	5.2	1.91	1.69	5.9	1.76	1.7	4.1	1.69	28938	102	9.48
MSP	10k	16	1.14	1.36	6.5	1.54	1.27	7.8	—	—	—	1.1	26717	120	11.29
MSP	40k	1	44.36	44.34	1.0	44.36	—	—	29.54	34.6	1.0	44.36	28	9	44.36
MSP	40k	2	24.21	24.43	1.8	24.27	—	—	16.25	17.9	1.9	24.18	34879	60	45.28
MSP	40k	4	12.41	12.2	3.6	12.49	13.14	—	8.53	10.3	3.4	12.37	35056	78	42.04
MSP	40k	8	6.83	7.05	6.3	7.04	6.89	—	5.24	4.88	7.1	6.79	38849	105	37.96
MSP	40k	16	3.64	4.71	9.4	4.12	3.68	—	—	—	—	3.58	39874	138	39.57

Table 10: Data for Multiple Shortest Paths application.

7.6 Matrix Multiplication

This program multiplies two dense $n \times n$ matrices A and B using Cannon's Algorithm (e.g., see [40]). The input matrices are assumed to be initially partitioned into blocks of size $n/\sqrt{p} \times n/\sqrt{p}$, such that processor i holds the block with index $(x, x + y \bmod \sqrt{p})$ of A, and the block with index $(x + y \bmod \sqrt{p}, y)$ of B, where $x = \lfloor i/\sqrt{p} \rfloor$ and $y = i \bmod \sqrt{p}$.

The algorithm then proceeds in $\sqrt{p}$ iterations. In each iteration, each processor first multiplies its two local blocks using a sequential blocked matrix multiplication algorithm, and adds the result to the local part of the result matrix C. It then sends the A block to the next processor on its right, and the B block to the next processor below it (modulo $\sqrt{p}$).

Cannon's Algorithm was chosen for its simplicity and its space and communication efficiency. The latency contribution due to the $\sqrt{p}$ supersteps is negligible in most situations. In fact, the example of matrix multiplication shows that for reasons of space-efficiency it may sometimes be desirable to increase the number of supersteps in order to decrease the amount of main memory needed to buffer the packets at the sending and receiving ends. In Cannon's Algorithm, each processor sends and receives in each superstep an amount of data proportional to the size of the input. For large input instances, it may become necessary to split each such large superstep into several smaller ones in order to avoid using a large amount of buffer space. (In the experiments reported here, the input instances are small enough to not require this modification.)

7.6.1 Discussion

Table 11 contains the data for the matrix multiplication application, which is the most trivial of our applications, and the most regular one in terms of the communication pattern. The number of supersteps is small (proportional to $\sqrt{p}$), and the communication cost is mainly determined by the size of the h-relations. Of course, as the input size increases, this cost is itself dominated by the local computation cost.

Note that this is the only application where the NEC Cenju achieves significantly better speedup than the SGI. Comparing the results with the predicted times, we observe that our

			SGI			Cenju			PC-LAN			SGI			SGI
app	size	p	pred (s)	time (s)	spdp	pred (s)	time (s)	spdp	pred (s)	time (s)	spdp	W (s)	H	S	TWk (s)
Matmult	144	1	0.43	0.42	1.0	0.43	0.47	1.0	0.29	0.3	1.0	0.43	0	1	0.43
Matmult	144	4	0.15	0.15	2.8	0.16	0.16	2.9	0.15	0.18	1.7	0.14	10368	3	0.54
Matmult	144	9	0.09	0.12	3.5	0.11	0.09	5.2	—	—	—	0.08	9216	5	0.64
Matmult	144	16	0.06	0.11	3.8	0.1	0.07	6.7	—	—	—	0.05	7776	7	0.7
Matmult	288	1	3.4	3.37	1.0	3.4	3.71	1.0	2.26	2.32	1.0	3.4	0	1	3.4
Matmult	288	4	0.99	1.01	3.3	1.05	1.11	3.3	0.84	1.1	2.1	0.95	41472	3	3.79
Matmult	288	9	0.5	0.59	5.7	0.57	0.55	6.7	—	—	—	0.46	36864	5	4.13
Matmult	288	16	0.32	0.42	8.0	0.42	0.36	10.3	—	—	—	0.29	31104	7	4.49
Matmult	432	1	11.53	11.49	1.0	11.53	12.55	1.0	7.68	7.83	1.0	11.53	0	1	11.53
Matmult	432	4	3.17	3.18	3.6	3.3	3.49	3.6	2.51	3.34	2.3	3.09	93312	3	12.33
Matmult	432	9	1.54	1.65	7.0	1.69	1.7	7.4	—	—	—	1.46	82944	5	13.03
Matmult	432	16	0.93	1.14	10.1	1.13	1.04	12.1	—	—	—	0.86	69984	7	13.66
Matmult	576	1	27.53	27.51	1.0	27.53	29.94	1.0	18.33	18.71	1.0	27.53	0	1	27.53
Matmult	576	4	7.29	7.33	3.8	7.52	8.09	3.7	5.56	7.52	2.5	7.15	165888	3	28.52
Matmult	576	9	3.47	3.69	7.5	3.72	3.84	7.8	—	—	—	3.32	147456	5	29.78
Matmult	576	16	2.09	2.42	11.4	2.43	2.31	13.0	—	—	—	1.97	124416	7	31.21

Table 11: Data for matrix multiplication application.

predictions for the SGI are too optimistic. We suspect that this may be due to the fact that the SGI is not a true BSP machine, as the only private memory in the SGI are the caches.

8 Interpretation of Experimental Results

In this section, we attempt to develop a context in which to understand the experimental results. In particular, we address the issues of performance relative to other programming approaches, and the accuracy of the BSP cost model.

8.1 Performance Relative to Other Approaches

We first consider the performance of the BSP code relative to other programming models. Would code based on other approaches—such as machine-specific code, MPI, LogP, etc.—have a substantial performance advantage? An examination of the experimental data indicates that for this set of BSP applications, larger problem sizes achieve greater efficiency. Indeed, the performance trends suggest that if sufficient main memory were available, an arbitrary level of efficiency could be achieved. Thus, the speed advantage obtained from *any* other approach, even a machine-specific one, will be negligible for large problem sizes.

The reason these BSP programs achieve high efficiency is simple: each application is designed such that the computation is balanced among the processors, and such that the communication to computation ratio decreases as problem size increases. It is important to note that not all algorithms are so well behaved. For example, if one assumes a cost model for which communication is inexpensive, as in the PRAM model, it is quite reasonable to design algorithms for which the communication to computation ratio is either constant or growing as problem size increases.

Though detailed head-to-head comparisons with alternative parallel programming approaches are beyond the scope of this paper, such comparisons would certainly be of interest. We encourage other researchers to compare their experimental results to those shown here.

8.2 Accuracy of the BSP Cost Model

We next consider the predictive accuracy of the BSP cost model. To keep the discussion in context, there are several points that should be considered. First, it can be argued that the purpose of the model is primarily prescriptive, as opposed to descriptive. (That is, from the point of view of the system designer, the model is primarily intended to guide the design. The extent to which the model provides accurate descriptions of current parallel systems is less significant.) Second, it should be noted that prediction accuracy is not just a property of the model itself, but also of its implementation. Third, there seems to be limited value in a parallel model that is very accurate if we have no way of predicting sequential performance. With these caveats in mind, we begin our discussion of the suitability of the BSP cost model for current systems.

The BSP model assigns a communication cost at the level of a batch of messages. In contrast, many other models (such as LogP) assign a cost at the single-message level. By modeling a finer-grained communication, such models may appear to provide a more accurate description of the underlying system. However, experimental results indicate that this is not the case [31]. Asynchronous single-message models assume overly precise and predictable timing behavior for low-level operations, and are particularly ineffective when communication cannot be predicted at run-time.

Our experimental results suggest that the BSP model provides reasonable accuracy, though it should be noted that many of these applications are large enough that they become computation-bound, and our use of BSP models only communication cost. It also appears that the model is sufficiently accurate to reveal overall performance trends, such as indicating when the use of additional processors will have a negative effect on performance. Whether the prediction accuracy demonstrated in these experiments is sufficient depends on the needs of the user and the application. In general, greater predictability could be obtained by having a model with more parameters, at the cost of making the model more complex and the algorithmic tradeoffs less obvious. The related work described in Section 4.2 addresses several proposed extensions to the BSP model.

An examination of the experimental results indicates there was one case where the cost model turned out to be too simplistic. The system using PCs and the Ethernet switch exhibited bad behavior on some of our programs when 8 processors were employed. It appears that in this configuration there were certain programs whose times were dominated by latency costs that triggered bad behavior. This happened on small instances of SP and Ocean. It did not happen with MST, the other program whose small-instance time was dominated by latency on the PC cluster, so perhaps that program did not trigger the switch's bad behavior. In this case, one could argue that the BSP cost model is providing an overly abstract description of a complex real-world system. However, of the systems that we tested, only the case of the 8-processor Ethernet switch resulted in a severe discrepancy between predicted and actual run times.

9 Conclusions

We have described the implementation and performance of several parallel applications that use a simple message-passing library based on the BSP model. Our results may be viewed as a partial validation of the practicality of the BSP model, since both efficiency and portability were demonstrated for a range of applications on diverse platforms.

Concerning the accuracy of the BSP cost model, we believe that the cost model should not be expected to accurately predict the precise running times on various input sizes and machines. Such a "curve fitting" approach seems more realistic on fairly simple subroutines (i.e., broadcast or sorting) than on more complex application programs. Also, note that the degree to which computation and communication can be overlapped depends on the particular architecture and application. (While we have defined the cost function as the sum of communication and computation costs, it is also sometimes defined as the maximum of the two.)

However, we found the cost model to be very reliable in modeling the overall behavior of an application, including the prediction of "breakpoints" at which the performance changes fundamentally due to the effects of latency, bandwidth, or local computation. We believe that this should make the BSP model a good evaluation tool for parallel architectures and algorithms. In general, we feel that the cost model was accurate enough to guide us towards an efficient solution.

10 Future Research

Additional work is needed in order to arrive at a more complete assessment of the strengths and limitations of the BSP approach. In particular, all the experiments in this paper were performed on parallel machines with a fairly small number of processors. Implementations on machines with larger numbers of processors exist [32, 52], but further work is needed, particularly on more recent massively parallel machines.

Investigating the range of applications that can be efficiently handled in a BSP style is another important issue. We are currently working on the implementation of some additional application programs, including several variants of the adaptive Fast Multipole Method [15].

Additional work is also needed to develop optimized and scalable BSP library implementations. The BSP models postulates the efficient resolution of arbitrary balanced communication patterns, which presents a challenging problem to every library designer. This type of communication makes it possible to consider a whole range of algorithmic ideas in order to avoid contention in the routing phase. (In contrast, many asynchronous models require the application programmer to schedule the communication, as there is very little a library designer can do to avoid contention.) Particularly in the case of large message-startup costs, this situation leads to a number of interesting algorithmic problems.

Finally, to promote utilization and further study of the model, standardization seems necessary. We believe that the BSP Standard Library proposed in [36] is an important first step in this direction.

Acknowledgments

We thank Andrew Goldberg and Marios Papaefthymiou for their help in the implementations of the minimum spanning tree and shortest paths applications. We acknowledge helpful discussions with Kai Li and Jim Philbin on the BSP library, and with J. P. Singh on N-body simulations. Thanks also to the reviewers of this paper, whose suggestions led to substantial improvements.

References

[1] "High Performance Fortran language specification," November 1994. High Performance Fortran Forum, Version 1.1.

[2] M. Adler, J. W. Byers, and R. M. Karp, "Scheduling parallel communication: The h-relation problem," in *Proceedings of the 20th Symposium on Mathematical Foundations of Computer Science*, pp. 1–20, August 1995.

[3] A. Bar-Noy and S. Kipnis, "Designing broadcasting algorithms in the postal model for message-passing systems," in *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 13–22, 1992.

[4] J. Barnes and P. Hut, "A hierarchical $O(N \log N)$ force-calculation algorithm," *Nature*, no. 324, pp. 446–449, 1986.

[5] A. Bäumker and W. Dittrich, "Fully dynamic search trees for an extension of the BSP model," in *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 233–242, June 1996.

[6] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide, "Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model," Tech. Rep. tr-rsfb-96-008, University of Paderborn, 1996.

[7] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide, "Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model," in *Proceedings of the 3rd Annual European Symposium on Algorithms*, pp. 17–30, September 1995.

[8] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis, "BSP vs LogP," in *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 25–32, June 1996.

[9] R. H. Bisseling and W. F. McColl, "Scientific computing on bulk synchronous parallel architectures," in *Proceedings of the 13th IFIP World Computer Congress* (B. Pehrson and I. Simon, eds.), vol. 1, pp. 509–514, Elsevier, 1994.

[10] R. H. Bisseling, "Sparse matrix computations on bulk synchronous parallel computers," in *Proceedings of the International Conference on Industrial and Applied Mathematics*, (Hamburg), July 1995.

[11] D. Blackston and T. Suel, "Highly portable and efficient implementations of parallel adaptive N-body methods," in *Proceedings of SC'97: High Performance Networking and Computing*, November 1997.

[12] G. Blelloch, P. Gibbons, Y. Matias, and M. Zagha, "Accounting for memory bank contention and delay in high-bandwidth multiprocessors," in *Seventh ACM Symposium on Parallel Algorithms and Architectures*, pp. 84–94, June 1995.

[13] G. E. Blelloch, "NESL: A nested data-parallel language (version 2.6)," Tech. Rep. CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, 1993.

[14] G. E. Blelloch, "Programming parallel algorithms," *Communications of the ACM*, vol. 39, no. 3, pp. 85–97, March 1996.

[15] J. Carrier, L. Greengard, and V. Rokhlin, "A fast adaptive multipole algorithm for particle simulations," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 4, pp. 669–686, July 1988.

[16] T. Cheatham, A. Fahmy, and D. Stefanescu, "General purpose optimization technology," tech. rep., Center for Research in Computing Technology, Harvard University, December 1994.

[17] T. Cheatham, A. Fahmy, D. C. Stefanescu, and L. G. Valiant, "Bulk synchronous parallel computing—a paradigm for transportable software," in *Proceedings of the 28th Hawaii International Conference on System Science*, vol. II, IEEE Computer Society Press, January 1995.

[18] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel programming in Split-C," in *Supercomputing '93*, November 1993.

[19] D. Culler, A. Dusseau, R. Martin, and K. E. Schauser, "Fast parallel sorting under LogP: From theory to practice," in *Proceedings of the Workshop for Portability and Performance for Parallel Processing*, July 1993.

[20] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a realistic model of parallel computation," in *Fourth ACM Symposium on Principles and Practice of Parallel Programming*, pp. 1–12, May 1993.

[21] F. Dehne, W. Dittrich, and D. Hutchinson, "Efficient external memory algorithms by simulating coarse-grained parallel algorithms," in *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 106–115, June 1997.

[22] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sundera, *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing.* Cambridge, MA: MIT Press, 1994.

[23] A. V. Gerbessiotis and C. J. Siniolakis, "Deterministic sorting and randomized mean finding on the BSP model," in *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 223–232, June 1996.

[24] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *Journal of Parallel and Distributed Computing*, vol. 22, no. 2, pp. 251–267, August 1994.

[25] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill, "Programming for different memory consistency models," *Journal of Parallel and Distributed Systems*, vol. 15, no. 4, pp. 399–407, August 1992.

[26] P. Gibbons, Y. Matias, and V. Ramachandran, "Efficient low-contention parallel algorithms," in *Sixth ACM Symposium on Parallel Algorithms and Architectures*, pp. 236–247, June 1994.

[27] P. Gibbons, Y. Matias, and V. Ramachandran, "The QRQW PRAM: Accounting for contention in parallel algorithms," in *Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 638–648, January 1994.

[28] P. B. Gibbons, Y. Matias, and V. Ramachandran, "Can a shared-memory model serve as a bridging model for parallel computation?," in *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 72–83, June 1997.

[29] A. V. Goldberg, K. Lang, and S. B. Rao, "Computing minimum spanning tree with the Green BSP library," In preparation, April 1996.

[30] M. W. Goudreau, K. Lang, S. B. Rao, and T. Tsantilas, "The Green BSP Library," Tech. Rep. CS-TR-95-11, Department of Computer Science, University of Central Florida, Orlando, Florida, June 1995.

[31] M. W. Goudreau and S. B. Rao, "Single message vs. batch communication," in *Algorithms for Parallel Processing* (M. Heath, A. Ranade, and R. Schreiber, eds.), vol. 105 of *IMA Volumes in Mathematics and Applications*, pp. 61–74, Springer-Verlag, 1998.

[32] M. W. Goudreau and E. D. Root, "A bulk-synchronous parallel library implementation for the BBN Butterfly GP1000," in *8th IEEE Symposium on Parallel and Distributed Processing*, pp. 288–297, October 1996.

[33] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[34] J. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[35] J. M. D. Hill, P. I. Crumpton, and D. A. Burgess, "Theory, practice, and a tool for BSP performance prediction," in *EuroPar'96*, no. 1124 in LNCS, pp. 697–705, Springer-Verlag, 1996.

[36] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling, “BSPlib: The BSP programming library,” *Parallel Computing*, 1998. To be published.

[37] B. H. H. Juurlink and H. A. G. Wijshoff, “The E-BSP model: Incorporating unbalanced communication and general locality into the BSP model,” Tech. Rep. 95-44, Leiden University, 1995.

[38] B. H. H. Juurlink and H. A. G. Wijshoff, “A quantitative comparison of parallel computation models,” in *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 13–24, June 1996.

[39] S. Knee, “Program development and performance prediction on BSP machines using Opal,” Tech. Rep. PRG-TR-18-94, Oxford University Computing Laboratory, August 1994.

[40] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

[41] D. Lecomber, “An object-oriented programming model for BSP computations,” tech. rep., Oxford University Computing Laboratory, 1994.

[42] C. Leiserson and B. M. Maggs, “Communication-efficient parallel algorithms for distributed random-access machines,” *Algorithmics*, vol. 3, pp. 53–77, 1988.

[43] P. Liu, W. Aiello, and S. Bhatt, “An atomic model for message-passing,” in *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 154–163, June 1993.

[44] P. Liu and S. N. Bhatt, “Experiences with parallel N-body simulations,” in *6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 122–131, 1994.

[45] S. Lumetta, A. Krishnamurthy, and D. E. Culler, “Towards modeling the performance of a fast connected components algorithms on parallel machines,” in *Supercomputing '95*, November 1995.

[46] B. M. Maggs, L. R. Matheson, and R. E. Tarjan, “Models of parallel computation: A survey and synthesis,” in *Proceedings of the 28th Hawaii International Conference on System Sciences*, vol. 2, pp. 61–70, IEEE Press, January 1995.

[47] W. F. McColl, “BSP programming,” in *Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms* (G. Blelloch, K. Chandy, and S. Jagannathan, eds.), pp. 21–35, American Mathematical Society, May 1994.

[48] W. F. McColl, “General purpose parallel computing,” in *Lectures in Parallel Computation, Proceedings 1991 ALCOM Spring School on Parallel Computation* (A. M. Gibbons and P. Spirakis, eds.), pp. 337–391, Cambridge University Press, 1993.

[49] R. Miller, "A library for bulk-synchronous parallel programming," in *Proceedings of the British Computer Society Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing*, pp. 100–108, December 1993.

[50] M. V. Nibhanupudi, C. D. Norton, and B. K. Szymanski, "Plasma simulation on networks of workstations using the bulk-synchronous parallel model," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, November 1995.

[51] A. Plaat, H. Bal, and R. Hofman, "Bandwidth and latency sensitivity of parallel applications in a wide-area system," unpublished manuscript, March 1998.

[52] G. W. Shumaker, "A bulk-synchronous parallel implementation on the Maspar," Master's thesis, University of Central Florida, Orlando, Florida, 1996.

[53] J. F. Sibeyn and M. Kaufmann, "BSP-like external-memory computation," in *Proceedings of the Italian Conference on Algorithms and Complexity (CIAC), LNCS 1203*, pp. 229–240, Springer-Verlag, 1997.

[54] J. P. Singh and J. L. Hennessy, "Data locality and memory system performance in the parallel simulation of ocean eddy currents," in *Proceedings of the 2nd International Symposium on High Performance Computing*, October 1991.

[55] J. P. Singh, J. L. Hennessy, and A. Gupta, "Scaling parallel programs for multiprocessors: Methodology and examples," *Computer*, vol. 26, no. 7, pp. 42–50, July 1993.

[56] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory," Tech. Rep. CSL-TR-92-526, Stanford University, Palo Alto, CA, June 1992.

[57] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[58] L. G. Valiant, "General purpose parallel architectures," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A: Algorithms and Complexity, ch. 18, pp. 943–971, Cambridge, MA: MIT Press, 1990.

[59] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 256–266, May 1992.

[60] M. Warren and J. Salmon, "Astrophysical N-body simulations using hierarchical tree data structures," in *Supercomputing '92*, pp. 570–576, 1992.