Towards Efficiency and Portability: Programming with the BSP Model

Mark Goudreau[1] *Kevin Lang*[2] *Satish Rao*[2,4]
Torsten Suel[2,4] *Thanasis Tsantilas*[3]

Abstract

The Bulk-Synchronous Parallel (BSP) model was proposed by Valiant as a model for general-purpose parallel computation. The objective of the model is to allow the design of parallel programs that can be executed efficiently on a variety of architectures. While many theoretical arguments in support of the BSP model have been presented, the degree to which the model can be efficiently utilized on existing parallel machines remains unclear.

To explore this question, we implemented a small library of BSP functions, called the *Green BSP* library, on several parallel platforms. We also created a number of parallel applications based on this library. Here, we report on the performance of six of these applications on three different parallel platforms. Our preliminary results suggest that the BSP model can be used to develop efficient and portable programs for a range of machines and applications.

1 Introduction

A fundamental obstacle to the widespread use of parallel machines for general-purpose computing is the lack of a widely accepted standard model of parallel computation. Unlike the world of sequential computing, where the widely accepted von-Neumann model facilitates the development of portable software, parallel programs developed on one machine often require major modifications before they can be efficiently employed on other parallel machines.

The *Bulk-Synchronous Parallel* or *BSP* model [34] was proposed by Valiant as a "bridging model" that provides a standard interface between the domains of parallel architectures and algorithms. In the BSP model, a parallel machine consists of a set of processors, each with its own private 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 global synchronization of all processors.

The communication time of an algorithm in the BSP model is given by a simple cost function. The two basic parameters that model a parallel machine are (1) the *gap* g, which reflects network bandwidth on a per-processor basis, and (2) 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 global synchronization.

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

$$w_i + gh_i + L,$$

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 superstep. The execution time of the entire program is defined as

$$W + gH + LS, \tag{1}$$

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.

Efficient programming of a BSP machine is based on a simple objective. To minimize the execution time as given by Equation (1), the programmer must attempt to (1) minimize the work depth of the program, (2) minimize the maximum number of packets sent or received by any processor in each superstep, and (3) 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 g and L parameters of the underlying machine.

Valiant [34, 35, 33] argues that, at least in theory, this approach is sufficient for portability and efficiency, by showing that many other programming styles can be automatically

[1] Department of Computer Science, University of Central Florida, Orlando, FL 32816–2362. Email: `goudreau@cs.ucf.edu`.
[2] NEC Research Institute, 4 Independence Way, Princeton, NJ 08540. Email: {`kevin,satish,torsten`}`@research.nj.nec.com`.
[3] Department of Computer Science, Columbia University, New York, NY 10027. Email: `thanasis@cs.columbia.edu`.
[4] Part of this work was done while visiting UC Berkeley. Present address: Computer Science Division, University of California at Berkeley, Berkeley, CA 94720.

and efficiently transformed into a BSP style. Furthermore, Gerbessiotis and Valiant [13] point out that a direct implementation on the BSP model will often lead to even better performance.

We briefly discuss two aspects of the BSP model. One is that 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).

Another observation is 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 has 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 achieve better bandwidth when routing large batches of 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.

Finally, we also feel that it is fundamentally easier to reason about the correctness and performance of BSP programs, as opposed to aggressively asynchronous message-passing programs.

1.1 Content of this Paper

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.

In particular, we designed several parallel applications that use the Green BSP library [15], a small library of BSP message-passing functions that we have implemented on a number of parallel platforms. The applications are:

- an N-body simulation using the Barnes-Hut algorithm,
- an ocean eddy simulation program adapted from the SPLASH application suite [31],
- a minimum spanning tree algorithm,
- a shortest paths algorithm,
- a multiple shortest paths algorithm, and
- a dense matrix multiplication algorithm.

In all of our applications, we used only the BSP cost model in both the design and optimization stages of the program development. That is, we made all of our design and optimization choices based purely on the BSP cost function as described by Equation (1). As stated earlier, a BSP programmer may use knowledge of a machine's g and L parameters in order to write more efficient code. Our approach, however, merely assumed that communication is somewhat more expensive than local computation and that global synchronization is considerably more expensive than communication. This approach appears reasonable for a wide range of current machines. In discussing our applications, we touch upon some of our programming decisions and their relationship to the BSP cost model.

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. That is, we provide some evidence that the BSP model is useful for designing efficient and portable parallel programs.

Another question that we investigate is the accuracy of the BSP cost function in comparison to the actual running times. Following [6], 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 was 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 many of our application these overhead costs were a small component of the overall execution time.

Even for those applications for which the communication and synchronization costs were significant, our results suggest the cost function is quite useful for predicting performance trends. For example, consider the performance of the ocean simulation with input size 130 in Figure 1.1. The cost model accurately predicts that little will be gained by using 4 PCs rather than 2, and that performance will severely degrade when using 8 PCs. Similarly, the cost function accurately predicts that the performance of the NEC Cenju on this application will not improve much by using more than 4 processors on this input size.

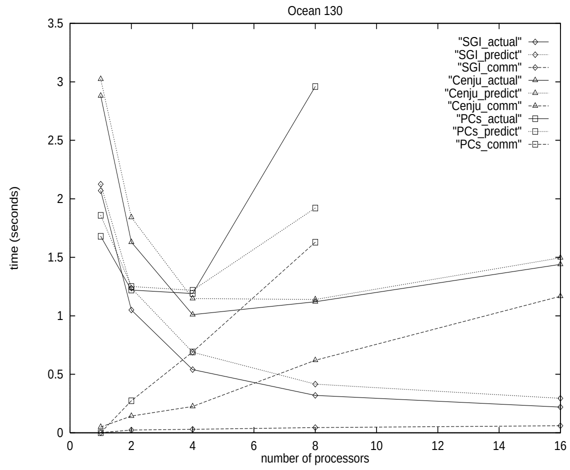

Figure 1.1: Actual and predicted times and predicted communication times (including synchronization) for Ocean (size 130)

1.2 Caveats

Before proceeding, we mention some caveats that 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. For most of our applications this is the case, even though the sequential code may not be completely optimized. For matrix multiplication, however, many highly optimized sequential codes exist, and thus our speed-ups for this application should be interpreted cautiously.

- Several of our results exhibit superlinear speed-up. As has been repeatedly observed, the total computational work for a parallel program may actually be less than that of a sequential version of the program. There are a variety of possible explanations for this effect (ranging from caching to suggestions that the sequential program is flawed.) As stated above, we provide data about the total work for each application and problem size. We discuss this issue in more detail in Section 3.

- Part of our objective is to examine the predictive capability of the BSP cost model. 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, and the work depths are measured as best as possible on our platforms.

- 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). The extent to which our results are applicable to larger machines is an open question. Promising initial results have been obtained for experiments on machines with 64 and more processors, but are not included here.

1.3 Related Work

Since the introduction of the BSP model, a number of papers have considered the design and analysis of algorithms under the BSP model; see, for example, [4, 6, 13, 25, 33].

Several groups of researchers are currently exploring the use of the BSP model on existing parallel machines. The Oxford BSP library, developed by Miller [27] while at Oxford University, allows a processor to directly access the memory of another processor. This makes the library very efficient to implement on shared-memory machines. Moreover, it is well suited for many static computations that arise in scientific computing. In contrast, the Green BSP library is based on message passing, which requires the programmer to prepare and read messages. On the other hand, the Green BSP library is better suited for the dynamic applications that we have experimented with.

Also at Oxford University, W. McColl's group is working on the development of several BSP programming languages and industrial applications [18, 20, 26].

A group at Harvard University lead by T. Cheatham and L. Valiant is studying higher-level programming languages and compilation techniques for the BSP model [9, 8]. R. Bisseling at the University of Utrecht is studying the use of the BSP model in the implementation of scientific computations [5, 6]. A recent implementation of a plasma simulation using the Oxford BSP library is described in [28].

A number of other models for general-purpose parallel computing have been proposed in recent years; see [24] for an overview. An important example for a model based on asynchronous message passing is the LogP model [11], which models 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[36] and the Split-C language [10], where it has been applied to the analysis of several algorithms.

Other related models are the Postal Model [2], the Atomic Model [22], and several models for end-point contention (e.g., see [1]) 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 somewhat different approach to portable parallel programming is based on standardized message-passing libraries such as PVM [12] and MPI [16]. 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. In fact, 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.

Finally, our choice of the application programs and presentation of the results is influenced by the SPLASH application suite for shared-memory machines [30]. Also, our BSP code for the ocean simulation was obtained by modifying the corresponding SPLASH program.

The remainder of the paper is organized as follows. Section 2 describes the versions of the Green BSP library used in our experiments and their performance. Section 3 describes the application programs and their performance. Finally, Sections 4 and 5 offer some concluding remarks and directions for future research.

2 BSP System: Implementation and Results

Our experiments use the Green BSP library [15], a small library of functions that implement the BSP model. The philosophy behind the library is to provide basic BSP functionality with a minimal number of functions. Thus, Green BSP offers only one type of communication and one type of synchronization operation.

This minimalist approach serves two purposes. First, it greatly simplifies the implementation of the library. A Green BSP library can be implemented on almost any parallel platform. Second, it focuses attention on the fundamental aspects of the BSP model. Part of our objective is to demonstrate that efficiency can be achieved even with such simple functionality. A description of the functions in the Green BSP library is given in Appendix A.

It should be noted that it is also possible to write applications in the BSP programming style using existing portable libraries such as PVM [12] and MPI [16]. However, these libraries provide far greater functionality than is required for the BSP model, and were not designed with the goal of supporting efficient BSP computation.

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,
- an MPI version, used on an NEC Cenju consisting of sixteen MIPS R4400 processors connected by a multistage network, with a peak bandwidth of 20 Mbytes/sec available for each processor, and
- a TCP verison, used on a system of eight 166-MHz Pentium PCs connected by a 100-Mbit Ethernet switch.

A short description of each of these library versions can be found in Appendix B.

Figure 2.1 shows the values of L and g achieved by the different versions of our library. The value for L corresponds to the time for a superstep in which each processor sends a single packet. The bandwidth parameter g is the time per 16-byte packet for a sufficiently large superstep with a total-exchange communication pattern.

3 Applications: Implementation and Results

For each of our applications, we ran experiments on 4 or 5 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. The complete data for all experiments is given in Appendix C, where we also explain how the numbers were obtained. A brief overview of the performance results is shown in Figures 3.1 and 3.2.

Figure 3.1 shows speed-up results for large input sizes, for each application and system. The speed-up 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 speed-up.

In analyzing the performance of our algorithms we noticed that the total work (i.e., local computation) performed by the 16-processor programs on the SGI were typically less than the total work performed by the single-processor programs. (A possible explanation is that the parallel codes were in fact better sequential programs than the single processor codes on these applications.) For this reason, we also include in Figure 3.1 the ratios of total work and parallel time for the 16 processor SGI. (These are the values in parentheses in the speed-up column for the SGI.)

In Figure 3.2, 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 messages 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 2.1.

nprocs	SGI bandwidth cost (microseconds)	SGI latency cost (microseconds)	Cenju bandwidth cost (microseconds)	Cenju latency cost (microseconds)	PC bandwidth cost (microseconds)	PC latency cost (microseconds)
1	.77	3	2.2	130	.92	2
2	.82	16	2.2	260	3.3	540
4	.88	29	2.2	470	4.8	1556
8	.97	52	2.5	1470	8.6	3715
9	1.0	57	2.7	1680	-	-
16	.95	105	3.6	2880	-	-

Figure 2.1: BSP system parameters

We also include the total work on 16 processors for the SGI, where the total work is defined as the sum of the local computation done by all the processors. This specifically does not include idle times caused by load imbalance, or any communication time.

The work depth 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. In some of our applications, this introduces systematic errors that produce high predicted running times. That is, occasionally the work depth will be more than the actual parallel runtime. 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.

3.1 Ocean Simulation

We converted an ocean eddy simulation program from the Stanford Parallel Library for Shared Memory Applications (SPLASH) [31] to our BSP system. The program computes ocean eddy currents using a multigrid technique on an underlying grid; see [29] for details. The conversion to BSP was fairly straightforward, due to the fact that the SPLASH code for this application was basically already in a BSP style.

3.1.1 Discussion

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 C.1 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.

We note that our estimates for the computational 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 computational work of the parallel program goes down dramatically for the PC-LAN, while it does not for the SGI system. Thus, any observed speed-up for the PC-LAN may have as much to do with this effect as with parallelism.

3.2 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 [3], which uses an irregular oct-tree structure, called BH tree, to hierarchically group bodies into clusters according to their distribution in three-dimensional space. Our parallel implementation is similar to those of Warren and Salmon [37] and Liu and Bhatt [23]. 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 [37], we only do so if the load imbalance reaches

	SGI (16 procs)		Cenju (16 procs)		PCs+LAN (8 procs)	
	time	spdp	time	spdp	time	spdp
Ocean (size = 514)	2.23	17.0(15.88)$^{\#}$	4.0	13*	6.46	7.2
N-Body (size = 64K)	5.04	14.8(13.9)	3.72	15.6	6.06	7.6
MST (size = 40K)	0.4	15.8 (9.8)	0.56	10.1	0.65	4.2
SP (size = 40K)	0.26	9.7 (7.23)	0.48	5.3	0.59	2.6
MSP (size = 40K)	4.71	9.4 (8.4)	3.68	12*	4.88	7.1
MM (size = 576)	2.42	11.4	2.31	13	na	na

Figure 3.1: Speed-up summaries for large problem size

app	size	SGI pred	SGI time	SGI W	H	S	Total Work 16 procs	Total Work 1 proc
ocean	514	2.48	2.23	2.38	69946	312	35.43	38.43
nbody	64k	4.97	5.04	4.95	24661	6	70.06	74.08
mst	40k	0.34	0.4	0.32	9562	62	3.92	6.3
sp	40k	0.28	0.26	0.26	2820	101	1.88	2.54
msp	40k	3.64	4.71	3.58	39874	138	39.57	44.36
matmult	576	2.09	2.42	1.97	124416	7	31.21	27.53

Figure 3.2: Algorithmic and model summaries for large problem size on 16 processor SGI system.

a certain threshold, as suggested in [23].

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 "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 by the sequential algorithm. A detailed description of our implementation can be found in [32].

3.2.1 Discussion

As input for our experiments we used the Plummer model generated by the SPLASH code [31]. The timing and speed-up results in Figures 3.1 and C.4 show that for large enough input sizes, the N-body code achieves nearly perfect parallel speed-up on the SGI and NEC Cenju. Our implementation needs slightly larger input sizes than the SPLASH code to achieve the same speed-up. However, even the largest input size in Figure C.4 is not overly large, given that simulations are currently performed with hundreds of thousands and even millions of bodies [37].

*This is an estimate on the speed-up as we were unable to run the largest problem size on a single processsor.

$^{\#}$Value in parentheses is the speed-up relative to the total work performed by all 16 processors. This speed-up is smaller than the speed-up relative to the single-processor version, thus indicating that the parallel program is in fact performing less work than the single-processor version.

The running time of the single-processor version of our implementation is slightly faster than that of the SPLASH code. As in the SPLASH code, we did not attempt to fully optimize the computation of the interactions, which take around 97% of the 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 decrease the resulting speed-up.

Our N-body code performs only six supersteps per iteration. This makes the program efficient even on fairly small problem sizes and high-latency platforms. The application is irregular and dynamic, due to the 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. The bandwidth requirements are fairly modest, as we were careful in minimizing the amount of data sent during the transmission of the "essential trees".

3.3 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*[1] 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 [21]. 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 [14] 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.

3.3.1 Discussion

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 4 on the very-high latency 8 processor PC-LAN, and a factor of 10 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. For the largest problem sizes the ratio is less than an eighth on our worst BSP machine. This suggests that we could perform MST computations on more highly connected graphs without much degradation in performance.

Finally, as discussed in the introduction, the good speed-up results for the minimum spanning tree application on large input sizes shown in Figure 3.1 should be qualified, since the total work (3.9 seconds) for 16 processors is significantly less than the work (6.3) for a single processor. The parenthesized number next to the speed-up, obtained by dividing the total work by the time required on 16 processors, is perhaps a more reasonable measure of the performance of the algorithm.

Thus, the best we can claim is about 70% of ideal speed-up (despite the speed-ups 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.

3.4 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 works 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.

3.4.1 Discussion

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 speed-up of 10 for a two-second long computation is not an embarassment, 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 speed-ups may be considered generous.

[1] This concept was originally defined for the DRAM model [21].

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.

3.5 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.

3.5.1 Discussion

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 speed-up 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 speed-up 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.

3.6 Matrix Multiplication

This program multiplies two dense $n \times n$ matrices A and B using Cannon's algorithm (e.g., see [19]). 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 $\sqrt{p}$).

3.6.1 Discussion

The matrix multiplication program 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 speed-up than the SGI. Comparing the results with the predicted times, we observe that our 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.

4 Conclusions

We have described the implementation and performance of several parallel applications that use a simple message-passing library based on the BSP model. We believe that our results are encouraging, and that they suggest that the BSP model can efficiently execute a range of applications on many current machines.

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.

5 Future Research

Additional work is certainly 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, and we plan to extend our study to several larger machines.

We are also currently working on the implementation of some additional application programs, including the adaptive Fast Multipole Method [7] and a hierarchical algorithm for the radiosity problem in computer graphics [17].

Finally, much algorithmic and experimental work is still needed in the implementation of optimized BSP libraries on different parallel machines.

Acknowledgements

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. We also thank the anonymous reviewers for their helpful comments.

References

[1] M. Adler, J. Byers, and R. M. Karp. Scheduling parallel communication: The h-relation problem. In *Proc. 20th Symp. on Math. Foundations of Computer Science*, Aug. 1995.

[2] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 13–22, June 1992.

[3] J. Barnes and P. Hut. A hierarchical $O(NlogN)$ force-calculation algorithm. *Nature*, 324(4):446–449, 1986.

[4] 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 *Proc. 3rd Annual European Symp. on Algorithms*, Sep. 1995.

[5] R. H. Bisseling. Sparse matrix computations on bulk synchronous parallel computers. In *Proc. Int. Conf. on Industrial and Applied Math.*, July 1995.

[6] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Proc. 13th IFIP World Computer Congress*, volume 1, pages 509–514. Elsevier, 1994.

[7] J. Carrier, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm for particle simulations. *SIAM J. on Scientific and Statistical Computing*, 9(4):669–686, July 1988.

[8] T. Cheatham, A. Fahmy, and D. Stefanescu. General purpose optimization technology. Tech. report, Center for Research in Computing Technology, Harvard University, Dec. 1994.

[9] T. Cheatham, A. Fahmy, D. C. Stefanescu, and L. G. Valiant. Bulk synchronous parallel computing – a paradigm for transportable software. In *Proc. 28th Hawaii Int. Conf. on System Science*. IEEE, Jan. 1995.

[10] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proc. Supercomputing 1993*, Nov. 1993.

[11] D. Culler, R. M. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM Symp. on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

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

[13] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *J. of Parallel and Distributed Computing*, 22(2):251–267, Aug. 1994.

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

[15] M. Goudreau, K. Lang, S. Rao, and T. Tsantilas. The Green BSP library. Tech. Report CR-TR-95-11, University of Central Florida, 1995.

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

[17] P. Hanrahan, D. Saltzman, and L. Aupperle. A Rapid Hierarchical Radiosity Algorithm. *Computer Graphics*, 25(4):197–206, July 1991.

[18] S. Knee. Program development and performance prediction on BSP machines using Opal. Technical Report PRG-TR-18-94, Oxford University Computing Laboratory, Aug. 1994.

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

[20] D. Lecomber. An object-oriented programming model for BSP computations. Tech. Report, Oxford University Computing Laboratory, 1994.

[21] C. Leiserson and B. M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmics*, 3:53–77, 1988.

[22] P. Liu, W. Aiello, and S. Bhatt. An atomic model for message-passing. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 154–163, June 1993.

[23] P. Liu and S. Bhatt. Experiences with parallel N-body simulation. *Proc. 6th ACM Symp. on Parallel Algorithms and Architectures*, pages 122–131, June, 1994.

[24] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: A survey and synthesis. In *Proc. 28th Hawaii Int. Conf. on System Sciences (HICSS)*, pages 61–70. IEEE, Jan. 1995.

[25] W. McColl. General purpose parallel computing. In A M Gibbons and P Spirakis, editors, *Lectures on Parallel Computation*, pages 337–391, Cambridge University Press, 1993.

[26] W. F. McColl. BSP programming. In G.E. Blelloch, K.M. Chandy, and S. Jagannathan, editors, *Proc. DIMACS Workshop on Specification of Parallel Algorithms*, pages 21–35. American Math. Soc. May 1994.

[27] Richard Miller. A library for bulk-synchronous parallel programming. In *Proc. of the British Computer Society Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing*, Dec. 1993.

[28] M. Nibhanupudi, C. Norton, and B. Szymanski. Plasma simulation on networks of workstations using the bulk synchronous parallel model. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, Nov. 1995.

[29] J. P. Singh. Data locality and memory system performance in the parallel simulation of ocean eddy currents. In *Proc. 2nd Int. Symp. on High Performance Computing*, Oct. 1991.

[30] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *Computer*, 27(7):45–55, July 1994.

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

[32] T. Suel. Programming Parallel N-Body Simulations with the Bulk-Synchronous Parallel Model. In preparation, April 1996.

[33] L. Valiant. General purpose parallel architecture. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 943–971, North Holland, 1990.

[34] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[35] L. G. Valiant. Why BSP computers? In *Proc. 7th Int. Parallel Processing Symp.*, pages 2–5. IEEE Press, April 1993.

[36] T. von Eicken, D. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. 19th Int. Symp. on Computer Architecture*, May 1992.

[37] M. S. Warren and J. K. Salmon. Astrophysical N-body simulations using hierarchical tree data structures. In *Proc. Supercomputing 1992*, pages 570–576, 1992.

A Description of the Green BSP Library

The basic communication and synchronisation functions in the Green BSP library consists of the following three functions:

- `void bspSynch(void)` provides barrier synchronization. When a process calls this function, it is stopped until all other processes have called it. After a process returns from a `bspSynch()` call, all packets that were sent to it in the previous superstep can be assumed to be available.
- `void bspSendPkt(int, bspPkt *)` sends a packet to another process. The first argument is the ID of the destination, and the second argument is a pointer to the packet. All results in this paper were obtained with a fixed packet size of 16 bytes[2]. The data in the packet can be in any format, and it is up to the programmer to provide sufficient labeling information.
- `bspPkt *bspGetPkt(void)` returns a pointer to a packet that was sent to the process in the previous superstep. Packets may be returned by `bspGetPkt` in any arbitrary order. The function returns `NULL` if there are no further packets to get.

In addition, the library contains several auxiliary functions (e.g., for determining the process ID or the number of unreceived packets). While the Green BSP library provides a message-passing environment, it can also be very easily and efficiently implemented on shared-memory systems. In particular, this allows us to compare our results to those of programs written specifically for shared memory.

B Library Implementations

Following we give a brief description of each of the three library implementations used in this paper.

B.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.[3] 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 alorithm, which is tuned for the case of low contention. There is one case that probably would generate 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 processors to indicate that they are ready to proceed to the next superstep. Processor 0 then spins on variables 1 through $p-1$, while processors 1 through $p-1$ spin on variable 0.

[2] We are currently changing our system to allow the programmer to send packets of any arbitrary length. While this improves the readability and simplicity of some of our code, we do not expect any significant changes in performance on our current applications.

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

B.2 The MPI Version

In the MPI version, each process uses a distinct input and output buffer to communicate with 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.

B.3 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 that 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 processors pair off and talk according to a precomputed $p-1$ stage total-exchange pattern. Note that while this rigid scheduling method works well for most (i.e., 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-Mbit Ethernet switch that allows the $p/2$ conversations in each communication stage to occur in parallel. The maximum bandwidth that we observed between a pair of processors was 5 Mbytes/sec.

C Performance Data for the Applications

We provide performance numbers for our applications on 4 or 5 different input sizes and numbers of processors for our three systems. We also provide algorithmic data for each application. In particular, the tables on the next two pages contain the following data:

- Predicted execution times based on the BSP cost function.
- Runtimes and speedups on all platforms.
- W, the measured work depth on the SGI, and estimated work depths for the Cenju and PC-LAN.
- H, the sum of the h-relation sizes.
- S, the number of supersteps.
- The total measured sequential work for the SGI.

The work depths for the SGI are measured, while the work depths for the Cenju and the PC-LAN are estimated.

app	size	NP	SGI pred	SGI time	SGI spdp	Cenju pred	Cenju time	Cenju spdp	PC pred	PC time	PC spdp	SGI W	H	S	SGI TWk
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

Figure C.1: Data for Ocean Application

app	size	NP	SGI pred	SGI time	SGI spdp	Cenju pred	Cenju time	Cenju spdp	PC pred	PC time	PC spdp	SGI W	H	S	SGI TWk
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

Figure C.2: Data for MST Application

app	size	NP	SGI pred	SGI time	SGI spdp	Cenju pred	Cenju time	Cenju spdp	PC pred	PC time	PC spdp	SGI W	H	S	SGI TWk
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

Figure C.3: Data for Matrix Multiplication Application

app	size	NP	SGI pred	SGI time	SGI spdp	Cenju pred	Cenju time	Cenju spdp	PC pred	PC time	PC spdp	SGI W	H	S	SGI TWk
nbody	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
nbody	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
nbody	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
nbody	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
nbody	1k	16	0.05	0.05	9.2	0.06	0.07	4.6	-	-	-	0.05	2530	6	0.68
nbody	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
nbody	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
nbody	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
nbody	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
nbody	4k	16	0.3	0.25	11.6	0.26	0.22	9.5	-	-	-	0.29	6249	6	3.34
nbody	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
nbody	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
nbody	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
nbody	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
nbody	16k	16	1.13	1.12	13.8	0.9	0.86	13.5	-	-	-	1.12	12100	6	15.37
nbody	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
nbody	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
nbody	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
nbody	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
nbody	64k	16	4.97	5.04	14.8	3.82	3.72	15.6	-	-	-	4.95	24661	6	70.06
nbody	256k	1	344	345.59	1.0	258	-	-	229	212	1.0	344.43	0	4	344.43
nbody	256k	2	168	167.92	2.1	126	-	-	112	111	1.9	168.07	37493	6	333.3
nbody	256k	4	83	83.71	4.1	62	62.92	-	56	57	3.7	83.0	63321	6	322.04
nbody	256k	8	42	41.86	8.3	31	31.66	-	28	26.4	8.0	41.7	59251	6	318.65
nbody	256k	16	22	22.16	15.6	17	16.37	-	-	-	-	22.1	53422	6	316.38

Figure C.4: Data for N-body Application

app	size	NP	SGI pred	SGI time	SGI spdp	Cenju pred	Cenju time	Cenju spdp	PC pred	PC time	PC spdp	SGI W	H	S	SGI TWk
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

Figure C.5: Data for Shortest Path Application

app	size	NP	SGI pred	SGI time	SGI spdp	Cenju pred	Cenju time	Cenju spdp	PC pred	PC time	PC spdp	SGI W	H	S	SGI TWk
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

Figure C.6: Data for Multiple Shortest Paths Application