

AN EXPERIMENTAL EVALUATION OF BSP SORTING ALGORITHMS

TIFFANI L. WILLIAMS*
School of Computer Science
University of Central Florida
Orlando, FL 32816-2362
williams@cs.ucf.edu
USA

MARK W. GOUDREAU†
C&C Research Laboratories
NEC USA, Inc.
4 Independence Way
Princeton, NJ 08540
goudreau@cctl.nj.nec.com
USA

ABSTRACT

Though parallel sorting algorithms have been investigated for many different machines and models, experimental demonstration of portable and efficient parallel sorting is still a challenge. This paper investigates parallel sorting based on the Bulk-Synchronous Parallel (BSP) model. Proponents of the model argue that the structured programming approach is easy to reason about, and that the resulting code is portable at negligible cost for interesting (large) problem sizes. We empirically study the performance of four sorting algorithms (randomized sample sort, deterministic sample sort, bitonic sort, and radix sort) on two different parallel platforms (an SGI Challenge and an Intel Paragon). Although the results indicate relatively low speedups for small problem sizes, the results also demonstrate that efficient use of larger machines can be attained by increasing the problem size. Moreover, these results suggest that, from a practical standpoint, there is little need for more complex approaches to parallel sorting.

KEYWORDS: BSP, LogP, parallel sorting algorithms

1 INTRODUCTION

Sorting is one of the most common applications performed by computers. Sequential sorting algorithms have been developed under the Random-Access Machine (RAM) model, an abstraction of the von Neu-

mann model which has guided uniprocessor hardware design for decades. Parallel sorting algorithms have been investigated for many different machines and models, but unlike sequential computing, parallel computing has no widely accepted model for program development. As a result, efficient parallel programs are often machine-specific. One bridging model that attempts to provide a realistic basis for both software and hardware is the Bulk-Synchronous Parallel (BSP) model developed by Valiant [1]. We explore the parallel sorting problem under this model. In particular, we developed implementations of sorting algorithms—randomized sample sort, deterministic sample sort, bitonic sort, and radix sort—and evaluated their performance on two parallel platforms.

A BSP computer consists of a set of processor-memory pairs, a communication network, and a mechanism for (efficient) barrier synchronization of all the processors. The performance of a BSP computer is characterized by p (the number of processors), L (the latency between successive barrier synchronization operations), and g (the time interval between consecutive message transmissions at a processor). g and L are determined experimentally for each parallel computer. A computation consists of a sequence of parallel *supersteps*. During a superstep, each processor is allocated a task consisting of some combination of local computation steps, message transmissions, and message arrivals from other processors. A message sent in one superstep is guaranteed to be available to the destination processor at the beginning of the next superstep. Each superstep is followed by a global barrier synchronization of all processors.

The time complexity of a superstep in a BSP program is:

$$w + gh + L,$$

where w is the maximum number of basic computa-

*Supported by the McKnight Doctoral Fellowship from the Florida Education Fund.

†Supported partially by a grant from the NEC Research Institute. Most of this work was done while the author was at the University of Central Florida.

tional operations executed by any processor in the local computation phase, and h is the maximum number of messages sent or received by any processor. The total execution time for the program is the sum of all the superstep times.

The claim that both efficiency and portability can be achieved by using the BSP model is supported by both theoretical [1] and experimental [2] results. However, other general-purpose models, such as LogP [3], make similar claims. LogP models the performance of point-to-point messages with three parameters representing software overhead, network latency, and communication bandwidth. Under LogP, the programmer is not constrained by a superstep programming style. Although proponents of LogP argue that it offers a more flexible style of programming, Goudreau and Rao [4] argue that the advantages are largely illusory, since both approaches lead to very similar high-level parallel algorithms. In fact, most of the BSP sorting algorithms discussed here are, from a high-level perspective, virtually identical to Dusseau *et al.*'s LogP implementations [5]. The main difference between the two models is that under LogP the scheduling of communication at the single-message level is the responsibility of the application programmer, while under BSP the underlying system performs that task. We argue that the cost of allowing the underlying system to handle communication scheduling is negligible, thus the higher-level BSP approach is preferable.

Similar parallel sorting studies are described by Blelloch *et al.* for a Connection Machine CM-2 [6], Hightower, Prins, and Reif for a MasPar MP-1 [7], and by Helman, Bader, and JáJá on a Connection Machine CM-5, an IBM SP-2, and a Cray Research T3D using various input distributions [8]. Of particular relevance is the work of Dusseau *et al.* [5], which described several sorting approaches on a Connection Machine CM-5 using the LogP cost model [9]. Dusseau *et al.*'s work is very similar in philosophy to this work in that it advocates the use of a bridging model for the design of portable and efficient code.

Experimental results for sorting based directly on the BSP model can be found in the work of Gerbessiotis and Siniolakis for an SGI Power Challenge [10], Jurulink and Wijshoff for a MasPar MP-1 and GCel [11], Shumaker and Goudreau for a MasPar MP-2 [12], and Hill *et al.* for a Cray T3E [13].

Throughout the rest of the paper, n designates the total number of keys to be sorted and p represents the number of processors. Section 2 gives a brief overview of the sorting algorithms, a description of our experimental platforms, and describes the experimental results. Concluding remarks are given in Section 3.

2 EXPERIMENTS

The sorting algorithms used in these experiments were selected because of the communication challenges they present to a parallel machine. Detailed descriptions of the algorithms are precluded due to the space limitations of this paper. However, in this section we give high-level descriptions of the algorithms, to give the reader some intuition.

Randomized sample sort and deterministic sample sort are based on the selection of a set of $p - 1$ "splitters" from the set of input keys. The sorted splitters, s_1, \dots, s_{p-1} , logically divide the input keys into p buckets, where bucket 0 contains all keys with values less than s_1 , bucket 1 contains all keys with values greater than or equal to s_1 but less than s_2 , etc. All processors gain knowledge of the splitters, and after a large communication stage, processor i will locally sort all the keys in bucket i . Efficient operation of these algorithms depend on the selection of splitters; in particular, we seek splitters that will divide the input keys into approximately equal-sized buckets. These sample sorts require irregular and unbalanced communication.

Bitonic sort was developed by Batcher [14]. The approach used here is one for which the input size is far larger than the number of processors available; a straightforward mapping of comparison node operations to processors is used. Bitonic sort consists of regular and balanced communication.

Radix sort relies on the binary representation of the unordered list of keys. The keys are sorted over a number of passes based on the number of bits in the keys and the chosen radix. Each pass can require a large amount of communication, as the keys are moved to new processors. The resulting communication patterns are irregular but balanced.

The code was written using the BSPlib library [15] and experiments were run on two platforms:

- An SGI Challenge—a shared-memory platform—with 16 MIPS R4400 processors running IRIX System V.4. A shared memory implementation of the BSP library developed by Kevin Lang was used.
- An Intel Paragon—a message-passing machine—with 32 i860 XP processors running Paragon OSF/1 Release 1.0.4. The BSPlib implementation used was developed by Travis Terry at the University of Central Florida.

The code was compiled with the cc compiler using the -O2 optimization flag. The run times do not include I/O, and timings started when the input data

		SGI Challenge			Intel Paragon		
	n	p	t (sec)	S_p	p	t (sec)	S_p
RD	10^5	4,7	0.10	1.10	6	0.26	1.27
	10^6	10	0.53	4.30	18, 20	0.95	4.59
	10^7	10	5.97	4.45	32	5.80	*
DT	10^5	6	0.08	1.38	4	0.23	1.43
	10^6	10	0.68	3.38	20	1.27	3.43
	10^7	14	6.80	3.90	32	9.67	*
BT	2^{17}	2	0.21	0.81	16	0.36	1.31
	2^{20}	8	2.00	1.16	32	1.40	3.29
	2^{23}	1	24.22	0.92	32	11.43	*
RX	10^5	6	0.35	0.31	8	0.80	0.41
	10^6	9	2.49	0.92	16, 18	3.44	1.27
	10^7	12	23.60	1.13	32	16.65	*

Table 1: p , execution time, and speedup of the sorting applications on problems of size n . * An undefined value resulting from the problem size being too large for one processor.

was evenly distributed among the processors. The input data consisted of uniformly distributed integers. For sequential sorting, we used an 11-bit radix sort, which was the fastest sort that we could find.

For each sorting algorithm, a number of different problem sizes were tested. The data for each problem size represents the result of one test case. We characterize the performance of our sorting algorithms by measuring the speedup, S_p , relative to an 11-bit sequential radix sort.

Table 1 summarizes the experimental results for an SGI Challenge and an Intel Paragon¹. Execution time reflects the best execution time achieved on a parallel machine, and p represents the number of processors used to attain the best execution time. Two numbers in the p column represent a tie.

The data indicates a general trend that for these applications, greater efficiency can be achieved by increasing the problem size. This is true not only for these sorting algorithms, but for a wide range of important applications. Intuitively, this will occur whenever the computation can be equally balanced among the processors, and communication and synchronization requirements grow more slowly than the computation requirements.

Dusseau *et al.* do not use speedups for their experimental results on the CM-5. Instead, they show their experimental results in terms of $\mu\text{s}/(\text{key}/\text{proc})$ which allows them to run larger problem sizes for multiple processors. As long as the key/proc factor is small enough that it fits into a single processor’s main memory, their performance comparisons are reasonably valid. Since randomized sample sort performed the best, additional experiments were run to evaluate

¹For bitonic sort, the input size and the number of processors both had to be powers of two.

SGI Challenge				Intel Paragon			
p	t (sec)	$\mu\text{s}/(\text{k}/\text{p})$	$S_{p'}$	p	t (sec)	$\mu\text{s}/(\text{k}/\text{p})$	$S_{p'}$
1	7.02	2.34	1.00	1	3.94	4.38	1.00
2	11.87	3.96	1.18	4	9.06	10.07	1.74
3	12.46	4.15	1.69	6	9.51	10.57	2.46
4	14.09	4.70	1.99	8	10.24	11.38	3.08
5	14.86	4.95	2.36	10	14.91	16.57	2.64
6	14.92	4.97	2.82	12	48.91	54.34	0.97
7	15.30	5.10	3.21	14	11.58	12.87	4.76
8	15.99	5.33	3.51	16	11.69	12.99	5.39
9	17.54	5.85	3.60	18	12.34	13.71	5.75
10	17.81	5.94	3.94	20	12.64	14.04	6.23
11	25.71	8.57	3.00	22	13.23	14.70	6.55
12	25.16	8.39	3.35	24	13.88	15.42	6.81
13	28.66	9.55	3.19	26	14.28	15.87	7.17
14	38.10	12.70	2.58	28	15.00	16.67	7.35
15	41.07	13.69	2.56	30	15.11	16.79	7.82
16	50.56	16.85	2.22	32	18.28	20.31	6.90

Table 2: Execution time, $\mu\text{s}/(\text{key}/\text{proc})$, and speedup of randomized sample sort.

its performance in terms of $\mu\text{s}/(\text{key}/\text{proc})$ for larger problem sizes.

The performance of randomized sample sort is measured by a speedup value $S_{p'} = (p \times t_1)/t_p$ where t_1 is the execution time on 1 processor and t_p is the execution time on p processors. Table 2 shows the speedup for randomized sample sort when $n/p = 3 \times 10^6$ on an SGI Challenge and $n/p = 9 \times 10^5$ on an Intel Paragon, respectively. These n/p values represent the largest problem sizes that fit into each processor’s main memory. For randomized sample sort, the best speedups obtained on the SGI Challenge and the Intel Paragon are 3.94 and 7.82, respectively.

3 CONCLUSIONS

We have shown the performance of four sorting algorithms. LogP proponents argue that the application programmer should not be constrained by the super-step programming style of BSP. However, the BSP sorting implementations used here are virtually identical to Dusseau *et al.*’s LogP implementations. Additionally, the asynchronous single-message passing of LogP is particularly ineffective for analyzing communication that cannot be predicted at compile time. In particular, Dusseau *et al.* ignore analyzing the communication for their radix and randomized sample sort implementations.

The experimental results show that even in the best parallel algorithms implemented here, the speedups obtained are moderate. We point out two reasons for this. First, the speedups are relative to the “best” sequential algorithm we could find, an 11-bit radix sort. This radix sort is considerably faster than other common sorts, such as the qsort function that is part of the standard C library. Second, to allow for

fair comparisons, the problem sizes were selected such that all the data could fit into the main memory, so that swapping would not be an issue. The overall data clearly demonstrates the trend that for larger problem sizes, all the parallel sorting algorithms implemented achieve greater efficiency. With larger memory sizes and larger problem sizes, greater speedup could have been demonstrated.

In conclusion, these experimental results demonstrate how increased efficiency under BSP can often be achieved by increasing the problem size. This is also the case for many other important applications. Thus, the cost of portable parallel computing is that larger problem sizes are needed to achieve the desired level of efficiency.

ACKNOWLEDGEMENTS

The authors thank Kevin Lang and Travis Terry for their BSPlib implementations, Satish Rao for enlightening discussions, the NEC Research Institute for the use of the SGI Challenge, and the anonymous reviewers for their helpful comments.

REFERENCES

- [1] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
- [2] M. W. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards efficiency and portability: Programming with the BSP model. In *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, June 1996.
- [3] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauer, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.
- [4] M. W. Goudreau and S. B. Rao. Single message vs. batch communication. In M. Heath, A. Ranade, and R. Schreiber, editors, *Algorithms for Parallel Processing*, volume 105 of *IMA Volumes in Mathematics and Applications*, pages 61–74. Springer-Verlag, 1998.
- [5] A. C. Dusseau, D. E. Culler, K. E. Schauer, and R. P. Martin. Fast parallel sorting under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):791–805, August 1996.
- [6] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167, March/April 1998.
- [7] W. L. Hightower, J. F. Prins, and J. H. Reif. Implementations of randomized sorting on large parallel machines. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–167, June 1992.
- [8] D. R. Helman, D. A. Bader, and J. JáJá. Parallel algorithms for personalized communication and sorting with an experimental study. In *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 211–222, June 1996.
- [9] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauer, 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*, pages 1–12, May 1993.
- [10] 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*, pages 223–232, June 1996.
- [11] 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*, pages 13–24, June 1996.
- [12] G. Shumaker and M. W. Goudreau. Bulk-synchronous parallel computing on the Maspar. In *World Multiconference on Systemics, Cybernetics and Informatics*, volume 1, pages 475–481, July 1997. Invited paper.
- [13] J. M. Hill, S. A. Jarvis, C. Siniolakis, and V. P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool: A case study in optimising SQL. Technical Report PRG-TR-17-97, Oxford University Computing Laboratory, 1997.
- [14] K. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.
- [15] 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.