Programming Research Group

AN EXPERIMENTAL STUDY OF SORTING ALGORITHMS ON THE BSP MODEL: THEORY, PRACTICE AND EXPERIENCE

Alexandros V. Gerbessiotis and Constantinos J. Siniolakis

PRG-TR-06-97

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

An Experimental Study of Sorting Algorithms on the BSP Model: Theory, Practice and Experience*

Alexandros V. Gerbessiotis and Constantinos J. Siniolakis

Computing Laboratory,
Oxford University,
Wolfson Building,
Oxford, OX1 3QD,
UK.

February 1997

Abstract

The Bulk-Synchronous Parallel (BSP) model of computation has been proposed by L.G. Valiant as a unifying and bridging model for the design, analysis and implementation of parallel algorithms that are both scalable and portable among diverse parallel machines. In this work we implement, using the Oxford BSP Toolset, *BSPlib*, a multiplicity of BSP deterministic and randomized sorting algorithms on a variety of parallel systems, including the SGI Power Challenge and the CRAY T3D. Our experimental results illustrate the efficiency, portability and reusability of our algorithms across such diverse hardware platforms, and reinforce our claim that the BSP model can be profitably employed as a programming platform or, indeed, as a kind of programming paradigm.

*The work of the first author was supported in part by EPSRC(UK) under grant GR/K16999 and the second author was supported in part by a Bodossaki Foundation Graduate Scholarship.

1 Introduction

The main reason for the moderate success of parallel computing has been the lack of a bridging and unifying model for parallel computation the way the von Neumann model has been for sequential computing [23]. The introduction of the Bulk-Synchronous Parallel (BSP) model of computation by L.G. Valiant [31, 32, 33] comes to address these limitations of parallel computing and presents itself as a possible bridge between parallel software and hardware.

A considerable amount of work has been devoted in the study of the BSP model, and the analysis and design of BSP algorithms. In this work, we present implementations, using the Oxford BSP Toolset, *BSPlib* [18], of deterministic and randomized BSP sorting algorithms – as well as algorithms designed for other models of computation (e.g., bitonic-sort) or dedicated to specific input data types (e.g., radix-sort) – on a variety of hardware platforms. We report on the performance and relative merits of these algorithms and the experience gained from studying the predicted theoretical performance of these algorithms as opposed to their actual running time.

1.1 The Model

The *Bulk-Synchronous Parallel* (BSP) model of computation deals explicitly with the notion of communication and synchronization among computational threads and has been proposed in [31, 32, 33] as a unified framework for the design, analysis and programming of general purpose parallel computing systems. It offers the prospect of achieving both scalable parallel performance and architecture independent parallel software and provides a framework which permits the performance of parallel and distributed systems to be analyzed and predicted in a consistent and concise way. The term *bulk-synchronous* reflects the underlying position of the model between the two extremes, (i) entirely synchronous systems and (ii) fully asynchronous systems. Processing units in the BSP model advance jointly through the program, with the required remote communication occurring between supersteps. The BSP computer as described in [31, 32, 33] consists of the following three components.

(1) A *collection of p processor/memory components* numbered $0, \dots, p-1$.

(2) A *communication network* that delivers messages point to point among the processors.

(3) A *mechanism* for efficient *barrier synchronization* of all or a subset of the processors.

The performance of any BSP computer can be characterized by the following three parameters: (i) p – the number of processing units, (ii) g – the ratio of the total throughput of the whole system in terms of basic computational operations, to the throughput of the communication network in terms of words of information delivered; more formally, g is related to the time required to realize h-relations in continuous message traffic in gh time units [23, 32], and (c) L – the minimum elapsed time, measured in basic computational operations, between successive synchronization operations that allows remote memory accesses to become effective through the network.

Computation on the BSP model proceeds in a succession of *supersteps*. During a superstep each processing unit is given a task to perform using data already available there locally before the start of the superstep. The task may include (i) computation steps on values held locally at the start of the superstep, (ii) message transmissions and (iii) message receipts. A superstep can complete at any time after the first L time units. The time complexity of a superstep S in a BSP algorithm is determined as follows. Each superstep is charged $\max\{L, x + gh\}$ basic time steps, where x is the maximum number of basic computational operations executed by any processor during S, and h is the maximum number of messages transmitted or received by any processor. Note that, if h_t is the maximum number of messages transmitted by any processor during S, and h_r is the maximum number of messages received by any processor during S, then $h = \max\{h_t, h_r\}$. The actual values of L and g for any BSP machine are likely to depend on p.

1.2 Specifying the Performance of BSP Algorithms

We specify the performance of a BSP algorithm $\mathcal{A}$ in three parts. First, we specify a sequential algorithm $\mathcal{A}^*$ with which we are comparing it and make explicit how we charge for the basic operations of both $\mathcal{A}$ and $\mathcal{A}^*$. Second, we specify two ratios π and μ. The former, π, is the ratio between the computation time $C_{\mathcal{A}}$, of the BSP algorithm, over the time $C_{\mathcal{A}^*}$ of the comparing sequential algorithm divided by p, i.e., $\pi = pC_{\mathcal{A}}/C_{\mathcal{A}^*}$. The latter, μ, is the ratio between the parallel time $M_{\mathcal{A}}$ required by the communication supersteps of the BSP algorithm and the computation time of $\mathcal{A}^*$ divided by p, i.e., $\mu = pM_{\mathcal{A}}/C_{\mathcal{A}^*}$. When we specify μ it is necessary to be explicit about the amount of information that can be conveyed in a single message. Finally, we specify conditions on n, p, L and g that are sufficient for the algorithm to make sense and the claimed bounds on π and μ to hold. We also claim as corollaries some sufficient conditions for the most interesting optimality criteria, such as c-optimality, i.e., $\pi = c + o(1)$ and $\mu = o(1)$. All asymptotic bounds refer to the problem size as $n \to \infty$. The other parameters may also tend to ∞ if they are expressed in terms of n, though it is not assumed so.

2 Related Work

A considerable amount of work has been devoted in the study of the BSP model [22, 23, 24, 31, 32, 33] and the analysis [3, 4, 6, 7, 15] and design [11, 8, 9, 10, 12, 22, 23, 24] of BSP algorithms.

Previously known results on BSP sorting include deterministic [1, 7, 12, 15, 29] and randomized [6, 8, 14] algorithms. In [1], it is shown that for all values of n and p such that $p = n^{1-\epsilon}$, for any constant $0 < \epsilon < 1$, a version of column-sort provides an upper bound of $\lceil \epsilon^{-3.42} \rceil O(n \lg n/p)$ and $O(gn \lg n/p)$ for computation and communication respectively. In [7], an improved upper bound of $(1 + \lfloor \epsilon^{-1}(1-\epsilon) \rfloor ((1-\epsilon)/2) + o(1))(T_{seq}/p)$ and $O(gn/p)$ is shown for computation and communication respectively, where T_{seq} is the time required by sequential sorting. In [12], it is shown that for all values of n and p such that $p = n/\lg^{2+a} n$, for any constant $a > 0$, a version of the algorithm of [7] provides an upper bound of $(1 + 2/a)(T_{seq}/p)$ and $O(gn \lg n/(p \lg\lg n))$ for computation and communication respectively. The work of [15, 29] resolves the problem of BSP deterministic sorting by establishing $\Theta(n \lg n/p)$ and $\Theta(n \lg n/(p \lg (n/p)))$ bounds on communication and computation respectively; unfortunately, the constant factors involved in the analysis of [15, 29] are large, and therefore, the underlying algorithms seem to be of little practical use.

In [6], a randomized algorithm is presented that provides with an upper bound of $(1 + o(1))(T_{seq}/p)$ and $O(gn \lg n/(p \lg (n/p)))$ (with high-probability) for computation and communication respectively, for all values of n and p such that $p = O(n/\lg^{1+\alpha} n)$, for any constant $\alpha > 0$. Finally, in [12, 14], it is shown that performance comparable to that of [6] can be achieved (by employing a new data structure) for values of p much closer to n, i.e., $p = \omega(n \lg\lg n/\lg n)$.

In this work we implement a variety of BSP deterministic [12] and randomized [6] sorting algorithms – as well as algorithms designed for other models of computation (e.g., bitonic-sort [2, 13]) or dedicated to specific input data types (e.g., radix-sort [26, 13]) – on a variety of hardware platforms using the Oxford BSP Toolset.

3 The Algorithms

The deterministic algorithm that we implement is a non-iterative variant of the sorting algorithm of [7, 12] which has been shown to be one-optimal for a wide range of the BSP parameters. The process to manage the deterministic sorting operation is outlined in figure 1, by function SORT_DET_BSP(X), where in that figure, X denotes the input key set, n the size of X, p the number of processors, and s is a user defined parameter (oversampling factor) that inversely relates to the maximum possible imbalance of the sets formed in step (13) of the algorithm.

We note that this algorithm, as described, cannot handle duplicate keys. A solution to the duplicate key problem is attained by making the input keys distinct, i.e., by padding each key with information describing its position (processor and array index) [6]. Alternatively, as suggested in [12], maintaining such information for the sample keys and subsequently, for the pivot keys, allows the algorithm to maintain its one optimality even at the presence of duplicate keys. The latter approach is adopted in our implementations.

The randomized algorithm implementation is based on the non-iterative version of the one-optimal algorithm of [6]. The process to manage the randomized sorting operation is outlined in figure 2 by function SORT_RAN_BSP(X), where in that figure, X denotes the input key set, n the size of X, p the number of processors, and s is a user defined parameter (oversampling factor) that inversely relates to the maximum possible imbalance of the sets formed in step (13) of the algorithm.

The implementation of the randomized algorithm of figure 2 proved in practice to be 15% – 20% slower than that of the corresponding deterministic algorithm. This fact is due to two major factors: (i) step (9) (a binary search routine) proves to be inefficient

begin SORT_DET_BSP (X)
1. **let** $s = \lg\lg n$;
2. **for** each processor $p^{\langle k\rangle}$, $k \in \{0,\ldots,p-1\}$, in parallel
3. **do** SORT_SEQ$(X^{\langle k\rangle})$;
4. **select** locally a sample $Y^{\langle k\rangle} = \langle y_1,\ldots,y_{sp-1}\rangle$ of $sp-1$ evenly spaced splitters that partition $X^{\langle k\rangle}$ into sp evenly sized segments ;
5. **let** $\bar{Y} =$ BITONIC_SORT(Y) ;
6. **select** a sample $S = \langle s_1,\ldots,s_{p-1}\rangle$ of $p-1$ evenly spaced pivots that partition $\bar{Y}$ into p evenly sized segments ;
7. **communicate** S to all processors ;
8. **for** each processor $p^{\langle k\rangle}$, $k \in \{0,\ldots,p-1\}$, in parallel
9. **do partition** $X^{\langle k\rangle}$ into p subsequences $X_0^{\langle k\rangle},\ldots,X_{p-1}^{\langle k\rangle}$ as induced by the $p-1$ pivots of S ;
10. **for** all $i \in \{0,\ldots,p-1\}$,
11. **do communicate** subsequence $X_i^{\langle k\rangle}$ to $Z_k^{\langle i\rangle}$;
12. **let** $\bar{X}^{\langle k\rangle} =$ MERGE_SEQ$(\bigcup_i Z_i^{\langle k\rangle})$;
13. **return** $\bar{X}^{\langle k\rangle}$;

end SORT_DET_BSP

Figure 1: Procedure SORT_DET_BSP.

in practice, and (ii) the communication pattern of step (11) is fine-grained as compared to that of the sequential methods. To overcome the latter limitation of the randomized algorithm we employed radix-sort techniques to group keys together before communicating them in step (11). This approach attains a performance to within a $5\% - 10\%$ of the corresponding deterministic algorithm. In the final implementation of the algorithm we also addressed limitation (i); to this end we employed ideas from [7, 12]. In particular, we sorted the input keys before communicating them in step (11), and we replaced the sequential sorting algorithm in step (12) by a multi-way merging algorithm. This approach alleviates the need for binary search in step (9) and allows for coarse-grained communication in step (11), and thus, attains performance comparable to that of the corresponding deterministic algorithm. For handling duplicate keys we employed a method similar to the one adopted in the deterministic algorithm implementation.

We also implement a variant of Batcher's bitonic-sort algorithm [2]. The variant implemented is the one described in [12], where comparators are replaced by (n/p)-way mergers [20]. For details refer to [12].

An implementation on the BSP model of a radix-sort algorithm of [26], as described in [10], is presented. The process to manage the radix-sort operation is outlined in figure 3, by function SORT_RDX_BSP$(X,\ R)$, where in that figure, X denotes the input key set, $R = [0,\ldots,r-1]$ the range of the keys to be sorted, n the size of X, and p the number of processors.

begin SORT_RAN_BSP (X)
1. **let** $s = 2\lg^2 n$;
2. **select** uniformly at random a sample $Y = \langle y_1, \ldots, y_{sp-1} \rangle$ of $sp-1$ splitters ;
3. **communicate** Y to processor $p^{\langle 0 \rangle}$;
4 if $p^{\langle 0 \rangle}$ **then**
5. **do let** $\bar{Y} = $ SORT_SEQ(Y) ;
6. **select** locally a sample $S = \langle s_1, \ldots, s_{p-1} \rangle$ of $p-1$ evenly spaced pivots that partition $\bar{Y}$ into p evenly sized segments ;
7. BROADCAST(S) ;
8. **for** each processor $p^{\langle k \rangle}$, $k \in \{0, \ldots, p-1\}$, in parallel
9. **do partition** $X^{\langle k \rangle}$ into p subsets $X_0^{\langle k \rangle}, \ldots, X_{p-1}^{\langle k \rangle}$ as induced by the $p-1$ pivots of S ;
10. **for** all $i \in \{0, \ldots, p-1\}$,
11. **do communicate** subset $X_i^{\langle k \rangle}$ to $Z_k^{\langle i \rangle}$;
12. **let** $\bar{X}^{\langle k \rangle} = $ SORT_SEQ$(\bigcup_i Z_i^{\langle k \rangle})$;
13. **return** $\bar{X}^{\langle k \rangle}$;

end SORT_RAN_BSP

Figure 2: Procedure SORT_RAN_BSP.

4 Performance Evaluation

In all of our algorithms we employed the BSP model in both the design and implementation stages of the algorithm development. That is, we based our design and implementation choices purely on the BSP cost model, i.e., the algorithms take into explicit consideration the BSP parameters p, g and L.

Although the BSP model, as previously described, may be regarded as an architectural model, it can be readily employed as a programming platform or, indeed, as a kind of programming paradigm. The underlying concept in the BSP model is the notion of the superstep and the abstraction that non-local communication associated with a superstep takes place between supersteps as a global operation. Thus, a BSP program may be viewed as a succession of supersteps, with the required non-local communication occurring at the end of each superstep. Viewed this way, the BSP model can be realized as a library of functions with architecture independent semantics for process creation, remote data access and bulk synchronization. The Oxford BSP Toolset, *BSPlib*, implements such a paradigm [18]. The library functions are callable from standard imperative languages such as, C and Fortran. The implementations presented in this work were programmed in ANSI C. Only recompilation is required to run the same code on different hardware platform. Such platforms have been a Silicon Graphics Power Challenge shared-memory system and a CRAY T3D distributed-memory machine.

4.1 BSP Algorithms

The following BSP sorting algorithms have been implemented on top of *BSPlib*.

begin Sort_Rdx_BSP (X, R)
for each processor $p^{\langle k \rangle}$, $k \in \{0, \ldots, p-1\}$, in parallel
  do for $i = 1$ **to** $i = r$
    do let $X_{i-1}^{\langle k \rangle} = \{x — x = (i-1)$ and $x \in X^{\langle k \rangle}\}$;
      let $C^{\langle k \rangle}[i-1] = |X_{i-1}^{\langle k \rangle}|$;
for each processor $p^{\langle k \rangle}$, $k \in \{0, \ldots, p-1\}$, in parallel
  do for $i = 1$ **to** $i = r$
    do communicate $C^{\langle k \rangle}[i-1]$ to $\bar{C}^{\langle ((i-1)*p+k)/r \rangle}[((i-1)*p+k) \bmod r]$;
Parallel_Prefix_Sum($\bar{C}^{\langle k \rangle}$) ;
for each processor $p^{\langle k \rangle}$, $k \in \{0, \ldots, p-1\}$, in parallel
  do for $i = 1$ **to** $i = r$
    do communicate $\bar{C}^{\langle k \rangle}[i-1]$ to $C^{\langle (k*r+(i-1)) \bmod p \rangle}[(k*r+(i-1))/p]$;
for each processor $p^{\langle k \rangle}$, $k \in \{0, \ldots, p-1\}$, in parallel
  do communicate $X_{i-1}^{\langle k \rangle}$ to its final destination ;
end Sort_Rdx_BSP

Figure 3: Procedure Sort_Rdx_BSP.

(1) Three versions of a variant of the one-optimal deterministic BSP sorting algorithm of [12], called [DSG] (generic), [DSI] (integer), and [DSD] (double). The first version, [DSG], uses, as the underlying sequential routine, the standard C library quick-sort [19, 20] function `qsort` ([QS]) and our own merge-sort [20] function `msort` ([MS]), and can be employed to sort *any* data type. The second version, [DSI], uses our own radix-sort [20] function `xsort_i` ([XS]), and can be employed to sort *int*s. Finally, the third version, [DSD], uses own own merge-sort function `msort_d` ([MS]), that can be employed to sort *double*s (floating-point numbers).

(2) Three versions of a variant of the one-optimal randomized BSP sorting algorithm of [6], called [RSG] (generic), [RSI] (integer), and [RSD] (double). The way these three versions operate is in one to one correspondence to [DSG], [DSI], and [DSD].

(3) Three versions of a variant of bitonic sort [2, 13], called [BSG] (generic), [BSI] (integer), and [BSD] (double). The way these three versions operate is in one to one correspondence to [DSG], [DSI], and [DSD].

(4) A version of a variant of the parallel radix-sort algorithm of [26], called [XSI] (integer). The way this version operates is in one to one correspondence to [DSI].

Remark. All algorithms are stable; moreover, algorithms [DS{G,D,I}] and [RS{G,D,I}] handle duplicate keys, as mentioned earlier, by employing the method proposed in [12].

The reasoning behind the choice of the standard C library function `qsort` is that `qsort` is a widely available generic sorting routine, and thus, facilitates the comparison of parallel sorting algorithms that use it as the underlying sequential sorting algorithm.

The term *generic* implies that `qsort` can be used to sort *any* data type. To this end, a comparison function `compare` is used to facilitate the comparison of two elements e_1 and e_2. In particular, the `compare` function returns 1, 0 or -1 depending on whether e_1 is greater, equal or smaller than e_2.

We note, that the BSP implementations of our algorithms are *portable* and *reusable* enough to allow any sequential sorting algorithm (and merging algorithm where appropriate) to be used as the underlying sequential method.

4.2 Sorting Benchmarks

We have tested our implementations on a variety of input sets. We tried to conform to previously published data sets [16, 17] by including them in our experimental study. Our six sorting benchmarks are defined as follows, where `INT_MAX` defines the maximum integer value accommodated in a 32-bit signed arithmetic data type (e.g., $2^{31} - 1$). For simplicity, we also assume that floating-point numbers are upper bounded by `INT_MAX`.

(1) Uniform [U], the input is uniformly and at random distributed, and is generated by calling a pseudo random number generator, the C standard library function `lrand48`, which returns a long (integer) in the range $[0, \ldots, 2^{31} - 1]$. Floating-point numbers are defined similarly.

(2) Gaussian [G], the input follows a Gaussian distribution, and is approximated by adding the results of four calls to `lrand48` and dividing the sum by four.

(3) Bucket Sorted [B], the input (per processor) is split into p buckets, each of size n/p^2, so that the i-th such bucket, $0 \leq i \leq p-1$, consists of numbers uniformly and at random distributed in the range $[i\,\mathtt{INT\_MAX}/p, \ldots, (i+1)\,\mathtt{INT\_MAX}/p-1]$.

(4) g-Group [g-G], the processors are first divided into p/g groups each consisting of g processors, and within group j each processors splits its input into g buckets so that the i-th such bucket consists of numbers uniformly and at random distributed in the range $[((jg + p/2 + i) \bmod p)\,\mathtt{INT\_MAX}/p, \ldots, ((jg + p/2 + i + 1) \bmod p)\,\mathtt{INT\_MAX}/p - 1]$.

(5) Staggered [S], the input of processor $i < p/2$ is uniformly and at random distributed in the range $[(2i+1)\,\mathtt{INT\_MAX}/p, \ldots, (2i+2)\,\mathtt{INT\_MAX}/p-1]$, and of processor $i \geq p/2$ in the range $[(i-p/2)\,\mathtt{INT\_MAX}/p, \ldots, (i-p/2+1)\,\mathtt{INT\_MAX}/p-1]$.

(6) Deterministic Duplicates [DD], the $n/2^i$ input keys of the first $p/2^i$ consecutive processors are all set to $\lg{(n/p^{i-1})}$, and so forth. In processor $p-1$, $n/(p2^i)$ keys, starting from the lower indexed and proceeding to higher indexed, are set to $\lg n/(p2^{i-1})$ and so forth.

For a discussion of the practicality and suitability of this set of sorting benchmarks refer to [16, 17].

Sorting *doubles* with [DSD] – `msort_d`												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
1	2.19	2.19	2.19	2.19	2.19	1.24	-	-	-	-	-	-
8	0.29	0.29	0.26	0.26	0.27	0.16	1.26	1.26	1.17	1.17	1.20	0.72
32	0.08	0.08	0.07	0.07	0.07	0.05	0.32	0.32	0.29	0.28	0.30	0.19

Table 1: Execution time on a CRAY T3D for algorithm [DSD].

4.3 Experimental Result

In this section, we report on the performance and relative merits of the algorithms of section 3 and the experience gained from studying the predicted theoretical performance of these algorithms as opposed to their actual running time. The tables below summarize, due to space limitations, only a small fraction of the results we obtained.

In the tables to follow the following remarks apply.

- CPU times are given in seconds on a shared memory Silicon Graphics Power Challenge system and a distributed-memory CRAY T3D.
- The implementations are written in ANSI C - by the authors - and linked with the Oxford BSP Toolset, *BSPlib* version *0.81alpha*. The code is compiled using the native `cc` compiler with options `-O3` and `-s -O1` on the Silicon Graphics Power Challenge and the CRAY T3D systems respectively.
- The real-time clock function `bsp_time` [18] (wall-clock time) has been employed for timings.

It is noted that no particular effort was taken to optimize the code (other than using the optimization switches of the compiler) as our purpose has been to write comparatively readable and reusable code.

Remark. Each table reports the performance of our implementation on a single processor. The reason we do this is to allow comparisons between similar but not identical hardware. Thus, for example, if the uniprocessor running time of say, [RSG], for a particular problem size on some platform, is substantially different from that reported for `qsort` ([QS]) on a similar architecture elsewhere, this would indicate that the two system configurations are different (different RAM sizes, cache sizes, CPU clocks etc.). Then, comparisons of running times would be meaningful if such configuration differences were taken into consideration.

It is evident from tables 1 and 2 that our algorithms exhibit satisfactory performance (speedup ratios close to p). Moreover, the algorithms are oblivious to different data-distributions, as expected. Distribution [DD] is particularly well suited to our algorithms, as the duplicate key-handling technique we use along with the fact that the input in each processor is almost sorted result in high performance.

The bitonic-sort algorithm, as demonstrated in the timing results of table 3, performs relatively well for small processor numbers in accordance to the theoretical bound. In

Sorting *doubles* with [RSD] – `msort_d`												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
1	2.21	2.22	2.22	2.22	2.22	1.24	-	-	-	-	-	-
8	0.28	0.28	0.25	0.26	0.25	0.16	1.25	1.23	1.13	1.15	1.10	1.09
32	0.08	0.08	0.06	0.07	0.07	0.08	0.31	0.31	0.24	0.31	0.27	0.32

Table 2: Execution time on a CRAY T3D for algorithm [RSD].

Sorting *doubles* with [BSD] – `msort_d`												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
1	2.19	2.19	2.19	2.19	2.19	1.24	-	-	-	-	-	-
8	0.39	0.39	0.37	0.38	0.39	0.29	1.68	1.68	1.60	1.65	1.67	1.23
32	0.18	0.18	0.17	0.18	0.18	0.16	0.75	0.75	0.72	0.74	0.75	0.65

Table 3: Execution time on a CRAY T3D for algorithm [BSD].

particular, the algorithm is competitive to [DSD] and [RSD] up to four processors and for small problem instances. For this reason, bitonic-sort has been employed in algorithms [DSD] and [RSD] as the underlying sample-sort method.

For tables 4, 5 and 6 the same observations apply as for tables 1, 2 and 3.

The practical performance of algorithm [XSI] does not match its theoretical bound. The algorithm is much slower than the other integer sorting implementations. This is due to the fine grained communication pattern of step (13) of procedure SORT_RDX_BSP. In future versions of [XSI] we intend to reschedule the communication scheme as to result in coarse-grained communication. The algorithm is then expected to compete with more sophisticated BSP algorithms. Particularly, [XSI] delivers high performance for input of finite and small range. This is evidenced from the performance of the algorithm for the [DD] distribution.

It is evident from table 8 that our generic sorting algorithms [DSG] and [RSG] exhibit high performance (speedup ratios very close to p, and in some cases, exceeding p due to caching effects). Moreover, the algorithms are oblivious to different data-distributions, as expected.

Remark. The time reported in table 9 are inflated because of the additional profiling overhead required to collect the timing results.

Sorting *ints* with [DSI] – `xsort_i`												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
1	1.31	1.27	1.31	1.31	1.31	0.40	-	-	-	-	-	-
8	0.22	0.22	0.22	0.20	0.19	0.09	0.86	0.84	0.86	0.76	0.74	0.36
32	0.07	0.07	0.07	0.06	0.06	0.05	0.25	0.25	0.25	0.21	0.20	0.12

Table 4: Execution time on a CRAY T3D for algorithm [DSI].

Sorting *ints* with [RSI] – xsort_i												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
1	1.30	1.27	1.30	1.30	1.30	0.41	-	-	-	-	-	-
8	0.23	0.23	0.23	0.21	0.20	0.10	0.88	0.85	0.86	0.77	0.70	0.56
32	0.08	0.08	0.07	0.07	0.06	0.08	0.27	0.27	0.24	0.27	0.20	0.23

Table 5: Execution time on a CRAY T3D for algorithm [RSI].

Sorting *ints* with [BSI] – xsort_i												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
1	1.35	1.32	1.35	1.35	1.35	0.48	-	-	-	-	-	-
8	0.31	0.31	0.31	0.30	0.29	0.20	1.23	1.22	1.24	1.17	1.16	0.77
32	0.17	0.17	0.17	0.16	0.16	0.14	0.65	0.65	0.65	0.62	0.61	0.53

Table 6: Execution time on a CRAY T3D for algorithm [BSI].

As indicated in table 9, approximately 75% of the execution time of algorithms [DSI] and [RSI] is consumed on local sequential sorting. The remaining time is mainly allotted to local merging and routing operations. It is clear that our algorithms are communication efficient, i.e., no more than 5%-25% of the total time. We note that the discrepancy in the running time of sample sorting for these two algorithms is attributed to the fact that the randomized algorithm selects a larger sample (for the given configuration); it is however expected that the randomized algorithm will outperform the deterministic one for larger processor configurations.

The results obtained on the SGI Power Challenge machine are summarized in figures 10– 17. The comments applying to the CRAY T3D system can be extended to the SGI Power Challenge system. The results demonstrate that our sorting algorithms exhibit high performance (speedup ratios very close to p, and in some cases, exceeding p due to caching effects). Moreover, the algorithms are oblivious to different data-distributions, as expected.

4.4 Comparison with Previous Results

As noted in [16], despite the enormous theoretical interest in parallel sorting, the number of empirical studies is relatively small. All but three of these empirical studies used machines which are no longer available. The three exceptions are the recent work of

Sorting *ints* with [XSI] – xsort_i												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
8	1.16	1.06	1.15	1.20	1.10	0.28	2.30	2.20	2.30	2.32	2.17	0.49
32	0.16	0.15	0.16	0.16	0.15	0.05	1.21	1.16	1.21	1.20	1.10	0.31

Table 7: Execution time on a CRAY T3D for algorithm [XSI].

Sorting *ints* with [DSG], [RSG] and [BSG] – `qsort` and `msort`												
	[U]						[S]					
	[DSG]		[RSG]		[BSG]		[DSG]		[RSG]		[BSG]	
Procs	[q]	[M]	[q]	[M]	[q]	[M]	[q]	[M]	[q]	[M]	[q]	[M]
1	22.5	13.5	22.4	14.8	22.5	13.5	22.5	13.5	21.7	14.8	22.5	13.5
8	2.87	1.72	2.64	1.83	2.99	1.98	2.65	1.54	2.47	1.58	2.94	1.98
32	0.70	0.42	0.69	0.44	0.94	0.70	0.62	0.37	0.62	0.38	0.92	0.69

Table 8: Execution time on a CRAY T3D for algorithms [DSG], [RSG] and [BSG].

Sorting 2048K *ints* with [DSI] and [RSI] – `xsort_i`				
	[DSI]		[RSI]	
Stage	8	32	8	32
Initialization	0.000237	0.000686	0.000234	0.000539
Local Sorting	0.649214	0.168760	0.649633	0.168937
Sample Selection	0.000053	0.000184	0.001502	0.001503
Sample Sorting	0.001696	0.006198	0.005704	0.014347
Bucket location	0.000885	0.002628	0.000890	0.002438
Routing	0.044435	0.017483	0.045379	0.016557
Local Merging	0.168548	0.060152	0.159580	0.056584
Conclusion	0.000140	0.000233	0.009026	0.003122
Total	0.865208	0.256324	0.871948	0.264027

Table 9: Cost analysis for [DSS] and [RSS] on a CRAY T3D.

[16, 17, 30]. Figures 4, 5 and 6 compare the performance of these studies with that of our deterministic and randomized sample sort algorithms. We note that in [16, 17] the code is written in SPLIT-C and makes use of high-performance MPI-like primitives. On the other hand, in [30] the code is written in ANSI C and makes use of the Oxford BSP Toolset.

We note that figure 4 compares different data-distributions and therefore should be interpreted with caution. As far as figure 6 is concerned the underlying sequential sorting algorithms are author-made versions of quick-sort [19].

References

[1] M. Adler, J. W. Byers, and R. M. Karp. Parallel sorting with limited bandwidth. In *Proceedings of the 7-th ACM Symposium on Parallel Algorithms and Architectures,*

Sorting *doubles* with [DSD] – `msort_d`												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
1	3.26	3.26	3.25	3.28	3.25	2.02	15.1	15.1	15.0	15.1	15.2	9.60
4	0.68	0.67	0.63	0.61	0.62	0.34	3.91	3.95	3.81	3.72	3.90	2.54

Table 10: Execution time on an SGI Power Challenge for algorithm [DSD].

Sorting *doubles* with [RSD] – `msort_d`												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
1	3.26	3.24	3.26	3.26	3.27	2.04	15.0	14.9	15.0	14.9	15.0	9.57
4	0.70	0.69	0.67	0.66	0.63	0.33	3.87	3.82	3.76	3.79	3.65	2.30

Table 11: Execution time on an SGI Power Challenge for algorithm [RSD].

Sorting *doubles* with [BSD] – `msort_d`												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
1	3.19	3.19	3.18	3.19	3.18	1.88	14.8	14.9	14.8	14.8	14.8	9.21
4	0.81	0.82	0.78	0.80	0.81	0.51	4.67	4.68	4.59	4.60	4.68	3.40

Table 12: Execution time on an SGI Power Challenge for algorithm [BSD].

pp. 129-136, 1995.

[2] K. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference,* pp. 307-314, 1968.

[3] A. Baumker, 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 Annual European Symposium on Algorithms*, 1995.

[4] R. H. Bisseling and W. F. McColl. Scientific computing on Bulk-Synchronous Parallel architectures. Preprint 836, Department of Mathematics, University of Utrecht, December 1993.

[5] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10-th Annual ACM Symposium on Theory of Computing,* pp. 114-118, 1978.

[6] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel algorithms. *Journal of Parallel and Distributed Computing,* 22:251-267, 1994.

[7] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, Padova, Italy, June 1996.

Sorting *ints* with [DSI] – `xsort_i`												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
1	0.69	0.70	0.71	0.69	0.69	0.35	4.17	4.16	4.15	4.16	4.20	2.47
4	0.23	0.23	0.23	0.20	0.20	0.12	0.81	0.85	0.86	0.78	0.74	0.50

Table 13: Execution time on an SGI Power Challenge for algorithm [DSI].

Sorting *int*s with [RSI] – `xsort_i`												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
1	0.69	0.68	0.71	0.69	0.69	0.36	4.17	4.15	4.15	4.15	4.23	2.49
4	0.23	0.21	0.25	0.23	0.21	0.13	0.81	0.83	0.88	0.81	0.75	0.51

Table 14: Execution time on an SGI Power Challenge for algorithm [RSI].

Sorting *int*s with [BSI] – `xsort_i`												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
1	0.67	0.68	0.67	0.67	0.67	0.33	4.18	4.17	4.16	4.16	4.21	2.50
4	0.28	0.28	0.31	0.30	0.28	0.27	1.40	1.41	1.35	1.30	1.36	1.15

Table 15: Execution time on an SGI Power Challenge for algorithm [BSI].

[8] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications. Technical Report PRG-TR-13-96, Computing Laboratory, Oxford University, June 1996.

[9] A. V. Gerbessiotis and C. J. Siniolakis. Concurrent heaps on the BSP model. Technical Report PRG-TR-14-96, Computing Laboratory, Oxford University, June 1996.

[10] A. V. Gerbessiotis and C. J. Siniolakis. Selection on the Bulk-Synchronous Parallel model with applications to priority queues. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, California, USA, August 1996.

[11] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications in computational geometry. In *Proceedings of EURO-PAR'96*, Lyon, France, Lecture Notes in Computer Science, Springer-Verlag, August 1996.

[12] A. V. Gerbessiotis and C. J. Siniolakis. Efficient Deterministic Sorting on the BSP Model. Technical Report PRG-TR-19-96, Oxford University Computing Laboratory, October 1996.

[13] A. V. Gerbessiotis and C. J. Siniolakis. Primitive Operations on the BSP Model. Technical Report PRG-TR-23-96, Computing Laboratory, Oxford University, October 1996.

Sorting *int*s with [XSI] – `xsort_i`												
	512K						2048K					
Procs	[U]	[G]	[B]	[2-G]	[S]	[DD]	[U]	[G]	[B]	[2-G]	[S]	[DD]
1	2.06	2.01	2.10	2.06	2.07	0.81	8.48	8.07	8.48	8.49	8.49	3.35
4	2.65	2.52	2.72	2.66	2.58	0.28	4.03	3.76	3.82	3.82	3.74	0.99

Table 16: Execution time on an SGI Power Challenge for algorithm [XSI].

Sorting *ints* with [DSG], [RSG] and [BSG] – `qsort` and `msort`												
	[U]						[S]					
	[DSG]		[RSG]		[BSG]		[DSG]		[RSG]		[BSG]	
Procs	[q]	[M]	[q]	[M]	[q]	[M]	[q]	[M]	[q]	[M]	[q]	[M]
1	14.7	26.1	14.5	26.1	14.7	24.8	14.7	26.0	14.5	26.1	14.6	24.9
4	3.94	6.03	3.87	6.01	4.90	7.22	3.78	5.80	3.46	5.54	4.79	7.23

Table 17: Execution time on an SGI Power Challenge for [DSG], [RSG] and [BSG].

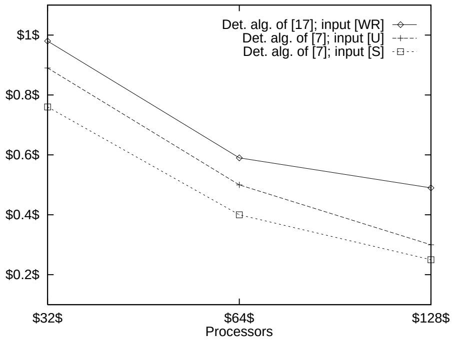

Figure 4: Comparative performance of this work and [16].

[14] A. V. Gerbessiotis and C. J. Siniolakis. A Randomized Sorting Algorithm on the BSP model. To appear in IPPS'97.

[15] M. T. Goodrich. Communication-Efficient Parallel Sorting. *Proceedings 26th ACM STOC*, 1996.

[16] D. R. Helman, J. JaJa, and D. A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. Technical Report UMIACS-TR-96-54/CS-TR-3670, The University of Maryland Institute for Advanced Computer Studies, August 1996.

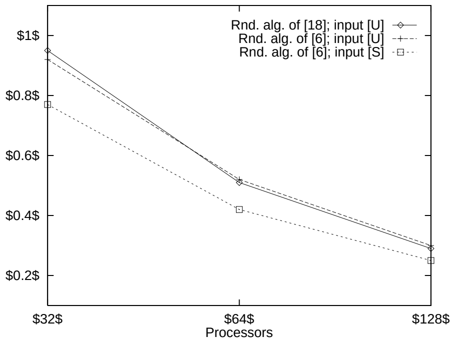

Figure 5: Comparative performance of this work and [17].

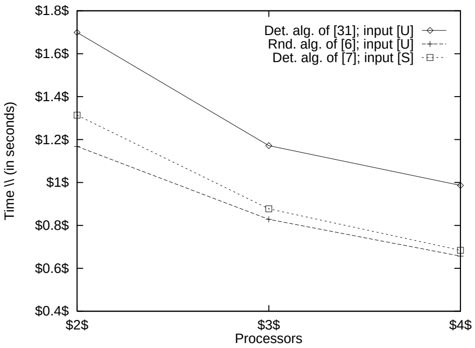

Figure 6: Comparative performance of this work and [30].

[17] D. R. Helman, D. A. Bader, and J. JaJa. A randomized parallel sorting algorithm with an experimental study. Technical Report UMIACS-TR-96-53/CS-TR-3669, The University of Maryland Institute for Advanced Computer Studies, August 1996.

[18] J. M. D. Hill and D. Skillicorn. Lessons learned from implementing BSP. To appear in High Performance Computing and Networking (HPCN'97), Lecture Notes in Computer Science, Springer-Verlag, April 1997.

[19] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5:10-15, 1962.

[20] D. E. Knuth. *The Art of Computer Programming. Volume III: Sorting and Searching.* Addison-Wesley, Reading, 1973.

[21] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes.* Morgan Kaufmann, California, 1991.

[22] W. F. McColl. General purpose parallel computing. In *Lectures on parallel computation,* (A. Gibbons and P. Spirakis, eds.), Cambridge University Press, 1993.

[23] W. F. McColl. Scalable parallel computing: A grand unified theory and its practical development. In *Proceedings of IFIP World Congress*, 1:539-546, Hamburg, August 1994.

[24] W. F. McColl. An architecture independent programming model for scalable parallel computing. *Portability and Performance for Parallel Processors,* (J. Ferrante and A. J. G. Hey, eds.), John Wiley and Sons, 1994.

[25] 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*, December 1993.

[26] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing,* 18:594-607, 1989.

[27] H. J. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM,* 34:60-76, January 1987.

[28] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal on Computing,* 14(2):396-409, 1985.

[29] C. J. Siniolakis. On the Complexity of BSP Sorting. Technical Report PRG-TR-09-96, Computing Laboratory, Oxford University, May 1996.

[30] K. R. Sujithan and J. M. D. Hill. Collection types for database programming in the BSP model. In *5-th EuroMicro Workshop on Parallel and Distributed Processing*, 1997.

[31] L. G. Valiant. Bulk-synchronous parallel computers. In *Parallel Processing and Artificial Intelligence*, (M. Reeve and S. E. Zenith, eds.), Wiley, 1989.

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

[33] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science,* (J. van Leeuwen, ed.), North Holland, 1990.

[34] L. G. Valiant. A combining mechanism for parallel computers. In *Parallel Architectures and Their Efficient Use*, pages 1-10, LNCS 678, Springer-Verlag, 1993.