A Parallel Sorting Algorithm With an Experimental Study (*Preliminary Draft*)

David R. Helman David A. Bader* Joseph JáJá†

Institute for Advanced Computer Studies, and
Department of Electrical Engineering,
University of Maryland, College Park, MD 20742
{helman, dbader, joseph}@umiacs.umd.edu

December 29, 1995

Abstract

Previous schemes for sorting on general-purpose parallel machines have had to choose between poor load balancing and irregular communication or multiple rounds of all-to-all personalized communication. In this paper, we introduce a novel variation on sample sort which uses only two rounds of regular all-to-all personalized communication in a scheme that yields very good load balancing with virtually no overhead. This algorithm was implemented in Split-C and run on a variety of platforms, including the Thinking Machines CM-5, the IBM SP-2, and the Cray Research T3D. We ran our code using widely different benchmarks to examine the dependence of our algorithm on the input distribution. Our experimental results are consistent with the theoretical analysis and illustrate the efficiency and scalability of our algorithm across different platforms. In fact, it seems to outperform all similar algorithms known to the authors on these platforms, and its performance is invariant over the set of input distributions unlike previous efficient algorithms. Our results also compare favorably with those reported for the simpler ranking problem posed by the NAS Integer Sorting (IS) Benchmark.

Keywords: Parallel Algorithms, Generalized Sorting, Integer Sorting, Sample Sort, Parallel Performance.

*The support by NASA Graduate Student Researcher Fellowship No. NGT-50951 is gratefully acknowledged.
†Supported in part by NSF grant No. CCR-9103135 and NSF HPCC/GCAG grant No. BIR-9318183.

1 Introduction

Sorting is arguably the most studied problem in computer science, both because of its intrinsic theoretical importance and its use in so many applications. Its significant requirements for interprocessor communication bandwidth and the irregular communication patterns that are typically generated have earned its inclusion in several parallel benchmarks such as NAS [7] and SPLASH [35]. Moreover, its practical importance has motivated the publication of a number of empirical studies seeking to identify the most efficient sorting routines. Yet, parallel sorting strategies have still generally fallen into one of two groups, each with its respective disadvantages. The first group, using the classification of Li and Sevcik [24], is the single-step algorithms, so named because data is moved once between processors. Examples of this include sample sort [20, 10], parallel sorting by regular sampling [32, 25], and parallel sorting by overpartitioning [24]. The price paid by these single-step algorithms is an irregular communication scheme and difficulty with load balancing. The other group of sorting algorithms is the multi-step algorithms, which include bitonic sort [9], column sort [23], rotate sort [26], hyperquicksort [29], flashsort [30], B-flashsort [19], smoothsort [28], and Tridgell and Brent's sort [33]. Generally speaking, these algorithms accept multiple rounds of communication in return for better load balancing and, in some cases, regular communication.

In this paper, we present a novel variation on the sample sort algorithm which addresses the limitations of previous implementations. We exchange the single step of irregular communication for two steps of regular communication. In return, we reduce the problem of poor load balancing because we are able to sustain a very high oversampling ratio at virtually no cost. Second, we obtain predictable, regular communication requirements which are essentially invariant with respect to the input distribution. The importance of utilizing regular communication has become more important with the advent of message passing standards, such as MPI [27], which seek to guarantee the availability of very efficient (often machine specific) implementations of certain basic collective communication routines.

Our algorithm was implemented in a high-level language and run on a variety of platforms, including the Thinking Machines CM-5, the IBM SP-2, and the Cray Research T3D. We ran our code using a variety of benchmarks that we identified to examine the dependence of our algorithm on the input distribution. Our experimental results are consistent with the theoretical analysis and illustrate the scalability and efficiency of our algorithm across different platforms. In fact, it seems to outperform all similar algorithms known to the authors on these platforms, and its performance is indifferent to the set of input distributions unlike previous efficient algorithms.

The high-level language used in our studies is Split-C [14], an extension of C for distributed memory machines. The algorithm makes use of MPI-like communication primitives but does not make any assumptions as to how these primitives are actually implemented. The basic data transport is a **read** or **write** operation. The remote read and write typically have both blocking and non-blocking versions. Also, when reading or writing more than a single element, bulk data transports are provided with corresponding **bulk_read** and **bulk_write** primitives. Our collective communication primitives, described in detail in [6], are similar to those of the MPI [27], the IBM POWERparallel [8], and the Cray MPP systems [13] and, for example, include the following: **transpose**, **bcast**, **gather**, and **scatter**. Brief descriptions of these are as follows. The **transpose** primitive is an all-to-all personalized communication in which each processor has to send a unique block of data to every processor, and all the blocks are of the same size. The **bcast** primitive is used to copy a block of data from a single source to all the other processors. The primitives **gather** and **scatter** are companion primitives. **Scatter** divides a single array residing on a processor into equal-sized blocks, each of which is distributed to a unique processor, and **gather** coalesces these blocks back into a single array at a particular processor. See [3, 6, 4, 5] for algorithmic details, performance analyses, and empirical results for these communication primitives.

The organization of this paper is as follows. **Section 2** presents our computation model for analyzing parallel algorithms. **Section 3** describes in detail our improved sample sort algorithm. Finally, **Section 4** describes our data sets and the experimental performance of our sorting algorithm.

2 The Parallel Computation Model

We use a simple model to analyze the performance of our parallel algorithms. Each of our hardware platforms can be viewed as a collection of powerful processors connected by a communication network that can be modeled as a complete graph on which communication is subject to the restrictions imposed by the latency and the bandwidth properties of the network. We view a parallel algorithm as a sequence of local computations interleaved with communication steps, and we allow computation and communication to overlap. We account for communication costs as follows.

Assuming no congestion, the transfer of a block consisting of m contiguous words between two processors takes $O(\tau + \sigma m)$ time, where τ is an upper bound on the latency of the network and σ is the time per word at which a processor can inject or receive data from the network. The cost of each of the collective communication primitives will be modeled by $O(\tau + \sigma \max(m, p))$, where m is the maximum amount of data transmitted or received by a processor. Such a cost (which is an overestimate) can be justified by using our earlier work [22, 21, 6, 5]. Using this cost model, we can evaluate the communication time $T_{comm}(n, p)$ of an algorithm as a function of the input size n, the number of processors p , and the parameters τ and σ. The coefficient of τ gives the total number of times collective communication primitives are used, and the coefficient of σ gives the maximum total amount of data exchanged between a processor and the remaining processors.

This communication model is close to a number of similar models (e.g. [16, 34, 1]) that have recently appeared in the literature and seems to be well-suited for designing parallel algorithms on current high performance platforms.

We define the computation time T_{comp} as the maximum time it takes a processor to perform all the local computation steps. In general, the overall performance $T_{comp}+T_{comm}$ involves a tradeoff between T_{comp} and T_{comm}. Our aim is to develop parallel algorithms that achieve $T_{comp} = O\left(\frac{T_{seq}}{p}\right)$ such that T_{comm} is minimum, where T_{seq} is the complexity of the best sequential algorithm. Such optimization has worked very well for the problems we have looked at, but other optimization criteria are possible. The important point to notice is that, in addition to scalability, our optimization criterion requires that the parallel algorithm be an efficient sequential algorithm (i.e., the total number of operations of the parallel algorithm is of the same order as T_{seq}).

3 A New Sample Sort Algorithm

Consider the problem of sorting n elements equally distributed amongst p processors, where we assume without loss of generality that p divides n evenly. The idea behind sample sort is to find a set of $p-1$ *splitters* to partition the n input elements into p groups indexed from 0 up to $p-1$ such that every element in the i^{th} group is less than or equal to each of the elements in the $(i+1)^{th}$ group, for $0 \leq i \leq p-2$. Then the task of sorting each of the p groups can be turned over to the correspondingly indexed processor, after which the n elements will be arranged in sorted order. The efficiency of this algorithm obviously depends on how well we divide the input, and this in turn depends on how well we choose the *splitters*. One way to choose the *splitters* is by randomly sampling the input elements at each processor - hence the name **sample sort**.

Previous versions of sample sort [20, 10, 17, 15] have randomly chosen s *samples* from the $\frac{n}{p}$ elements at each processor, routed them to a single processor, sorted them at that processor, and then selected every s^{th} element as a *splitter*. Each processor P_i then performs a binary search on these *splitters* for each of its input values and then uses the results to route the values to the appropriate destination, after which local sorting is done to complete the sorting process. The first difficulty with this approach is the work involved in gathering and sorting the *splitters*. A larger value of s results in better load balancing, but it also increases the overhead. The other difficulty is that no matter how the routing is scheduled, there exist inputs that give rise to large variations in the number of elements destined for different processors, and this in turn results in an inefficient use of the communication bandwidth. Moreover, such an irregular communication scheme cannot take advantage of the regular communication primitives proposed under the MPI standard [27].

In our solution, we incur no overhead in obtaining $\frac{n}{p^2}$ *samples* from each processor and in sorting these *samples* to identify the *splitters*. Because of this very high oversampling, we are able to replace the irregular routing with exactly two calls to our **transpose** primitive.

The pseudo code for our algorithm is as follows:

- **Step (1):** Each processor P_i $(0 \leq i \leq p-1)$ randomly assigns each of its $\frac{n}{p}$ elements to one of p buckets. With high probability, no bucket will receive more than $c_1\frac{n}{p^2}$ elements, where c_1 is a constant to be defined later.
- **Step (2):** Each processor P_i routes the contents of bucket j to processor P_j, for $(0 \leq i, j \leq p-1)$. Since with high probability no bucket will receive more than $c_1\frac{n}{p^2}$ elements, this is equivalent to performing a **transpose** operation with block size $c_1\frac{n}{p^2}$.
- **Step (3):** Each processor P_i sorts the $(\alpha_1\frac{n}{p} \leq c_1\frac{n}{p})$ values received in **Step (2)** using an appropriate sequential sorting algorithm. For integers we use the radix sort algorithm, whereas for floating point numbers we use the merge sort algorithm.
- **Step (4):** From its sorted list of $(\beta\frac{n}{p} \leq c_1\frac{n}{p})$ elements, processor P_0 selects each $(j\beta\frac{n}{p^2})^{th}$ element as a *splitter*, for $(1 \leq j \leq p-1)$. By default, the first and last *splitters* are respectively the smallest and largest values allowed by the data type used.
- **Step (5):** Processor P_0 **broadcasts** the $p-1$ intermediate *splitters* to the other $p-1$ processors.
- **Step (6):** Each processor P_i finds the positions of the *splitters* in its local array of sorted elements by performing a binary search for each of these *splitters*.
- **Step (7):** Each processor P_i routes the subsequence falling between *splitter* j and *splitter* $j+1$ to processor P_j, for $(0 \leq i, j \leq p-1)$. Since with high probability no sequence will contain more than $c_2\frac{n}{p^2}$ elements, where c_2 is a constant to be defined later, this is equivalent to performing a **transpose** operation with block size $c_2\frac{n}{p^2}$.
- **Step (8):** Each processor P_i merges the p sorted subsequences received in **Step (7)** to produce the i^{th} column of the sorted array. Note that, with high probability, no processor has received more than $\alpha_2\frac{n}{p}$ elements, where α_2 is a constant to be defined later.

We can establish the complexity of this algorithm with high probability - that is with probability $\geq (1-n^{-\epsilon})$ for some positive constant ϵ. But before doing this, we need to establish the results of the following four lemmas.

Lemma 1: At the completion of **Step (1)**, the number of elements in each bucket is at most $c_1\frac{n}{p^2}$ with high probability, for any $c_1 \geq 2$ and $p^2 \leq \frac{n}{3\ln n}$.

Proof: The probability that exactly $c_1\frac{n}{p^2}$ elements are placed in a particular bucket in **Step (1)** is given by the binomial distribution

$$b(s; r, q) = \binom{r}{s} q^s (1-q)^{r-s}, \tag{1}$$

where $s = c_1 \frac{n}{p^2}$, $r = \frac{n}{p}$, and $q = \frac{1}{p}$. Using the following Chernoff bound [12] for estimating the tail of a binomial distribution

$$\sum_{s \geq (1+\epsilon)rq} b(s; r, q) \leq e^{-\frac{\epsilon^2 rq}{3}}, \tag{2}$$

the probability that a particular bucket will contain at least $c_1 \frac{n}{p^2}$ elements can be bounded by $e^{-(c_1-1)^2 \frac{n}{3p^2}}$. Hence, the probability that any of the p^2 buckets contains at least $c_1 \frac{n}{p^2}$ elements can be bounded by $p^2 e^{-(c_1-1)^2 \frac{n}{3p^2}}$, and Lemma 1 follows.

Lemma 2: At the completion of **Step (2)**, the total number of elements received by processor P_0, which comprise the set of *samples* from which the *splitters* are chosen, is at most $\beta \frac{n}{p}$ with high probability, for any $\beta > 1$ and $p^2 \leq \frac{n}{3 \ln n}$.

Proof: The probability that processor P_0 receives exactly $\beta \frac{n}{p}$ elements is given by the binomial distribution $b(\beta \frac{n}{p}; n, \frac{1}{p})$. Using the Chernoff bound for estimating the tail of a binomial distribution, the probability that processor P_0 receives at least $\beta \frac{n}{p}$ elements can be bounded by $e^{-(\beta-1)^2 \frac{n}{3p}}$ and Lemma 2 follows.

Lemma 3: At the completion of **Step (7)**, the number of elements received by each processor is at most $\alpha_2 \frac{n}{p}$ with high probability, for any $\alpha_2 \geq 1.33$ and $p^2 \leq \frac{n}{3 \ln n}$.

Proof: Establishing a bound on the number of elements received by any processor in **Step (7)** is equivalent to establishing a bound on the number of elements which fall between any two consecutive *splitters* in the sorted order. But as Blelloch et al. [10] observed, the number of elements which fall between any two consecutive *splitters* in the sorted order can only be greater than $\alpha_2 \frac{n}{p}$ if in the sorted order there are less than $\frac{n}{p^2}$ *samples* drawn from the $\alpha_2 \frac{n}{p}$ elements which follow the first *splitter*. Since every element has an equal and independent probability of being a *sample*, the probability that exactly $\frac{n}{p^2}$ *samples* will be found amongst the next $\alpha_2 \frac{n}{p}$ elements is given by the binomial distribution $b(\frac{n}{p^2}; \alpha_2 \frac{n}{p}, \frac{1}{p})$. Using the following "Chernoff" type bound [18] for estimating the head of a binomial distribution

$$\sum_{s \leq \epsilon rq} b(s; r, q) \leq e^{-(1-\epsilon)^2 \frac{rq}{2}}, \tag{3}$$

where $s = \frac{n}{p^2}$, $r = \alpha_2 \frac{n}{p}$, and $q = \frac{1}{p}$, the probability that $\frac{n}{p^2}$ or less *samples* will be found amongst the next $\alpha_2 \frac{n}{p}$ elements following any of the p *splitters* can be bounded by $pe^{-(1-\frac{1}{\alpha_2})^2 \frac{\alpha_2 n}{2p^2}}$ and Lemma 3 follows.

Lemma 4: The number of elements exchanged by any two processors in **Step (7)** is at most $c_2 \frac{n}{p^2}$ with high probability, for any $c_2 \geq 2.48$ and $p^2 \leq \frac{n}{3 \ln n}$.

Proof: Since with high probability no processor can receive more than $\alpha_2 \frac{n}{p}$ elements in **Step (7)**, and since the randomization in **Step (1)** means that each of these elements can originate with equal probability from any of the p processors, the probability that exactly $c_2\frac{n}{p^2}$ elements are exchanged by any two particular processors is given by the binomial distribution $b(c_2\frac{n}{p^2};\alpha_2\frac{n}{p},\frac{1}{p})$. Using the Chernoff bound for estimating the tail of the binomial distribution, the probability that any of the p processors exchange at least $c_2\frac{n}{p^2}$ elements can be bounded by $p^2e^{-(\frac{c_2}{\alpha_2}-1)^2\frac{\alpha_2 n}{3p^2}}$ and Lemma 4 follows.

With these bounds on the values of c_1, α_2, and c_2, the analysis of our sample sort algorithm is as follows. **Steps (1)**, **(3)**, **(4)**, **(6)**, and **(8)** involve no communication and are dominated by the cost of the sequential sorting in **Step (3)** and the merging in **Step (8)**. Sorting integers using radix sort requires $O(\frac{n}{p})$ time, whereas sorting floating point numbers using merge sort requires $O(\frac{n}{p}\log\frac{n}{p})$ time. **Step (8)** requires $O(\frac{n}{p}\log p)$ time if we merge the sorted subsequences in a binary tree fashion. **Steps (2)**, **(5)**, and **(7)** call the communication primitives **transpose**, **bcast**, and **transpose**, respectively. The analysis of these primitives in [6] shows that with high probability these three steps require $T_{comm}(n,p) \le (\tau + 2\frac{n}{p^2}(p-1)\sigma)$, $T_{comm}(n,p) \le (\tau + (p-1)\sigma)$, and $T_{comm}(n,p) \le (\tau + 2.48\frac{n}{p^2}(p-1)\sigma)$, respectively. Hence, with high probability, the overall complexity of our sample sort algorithm is given (for floating point numbers) by

$$\begin{aligned} T(n,p) &= T_{comp}(n,p) + T_{comm}(n,p) \\ &= O\left(\frac{n}{p}\log n + \tau + \frac{n}{p}\sigma\right) \end{aligned} \tag{4}$$

for $p^2 < \frac{n}{3\ln n}$.

Clearly, our algorithm is asymptotically optimal with very small coefficients. But a theoretical comparison of our running time with previous sorting algorithms is difficult, since there is no consensus on how to model the cost of the irregular communication used by the most efficient algorithms. Hence, it is very important to perform an empirical evaluation of an algorithm using a wide variety of benchmarks, as we will do next.

4 Performance Evaluation

Sample sort was implemented using SPLIT-C [14] and run on a variety of machines and processors, including the Thinking Machines CM-5, the IBM SP-2-WN and SP-2-TN2, and the Cray Research T3D. For every platform, we tested our code on six different benchmarks, each of which had both a 32-bit *integer* version (64-bit on the Cray T3D) and a 64-bit double precision floating point number (*double*) version.

4.1 Sorting Benchmarks

Our six sorting benchmarks are defined as follows, in which MAX is $(2^{31}-1)$ for *integers* and approximately 1.8×10^{308} for *doubles*:

1. **Uniform [U]**, a uniformly distributed random input, obtained by calling the C library random number generator *random*(). This function, which returns integers in the range 0 to $(2^{31}-1)$, is initialized by each processor P_i with the value $(23+1001i)$. For the *double* data type, we "normalize" these values by first assigning the integer returned by *random()* a randomly chosen sign bit and then scaling the result by $\frac{\mathrm{MAX}}{(2^{31}-1)}$.

2. **Gaussian [G]**, a Gaussian distributed random input, approximated by adding four calls to *random*() and then dividing the result by four. For the *double* type, we first normalize the values returned by *random*() in the manner described for **[U]**.

3. **Zero [Z]**, a zero entropy input, created by setting every value to a constant such as zero.

4. **Bucket Sorted [B]**, an input that is sorted into p buckets, obtained by setting the first $\frac{n}{p^2}$ elements at each processor to be random numbers between 0 and $\left(\frac{\mathrm{MAX}}{p}-1\right)$, the second $\frac{n}{p^2}$ elements at each processor to be random numbers between $\frac{\mathrm{MAX}}{p}$ and $\left(2\frac{\mathrm{MAX}}{p}-1\right)$, and so forth.

5. g-**Group** [g-**G**], an input created by first dividing the processors into groups of consecutive processors of size g, where g can be any integer which partitions p evenly. If we index these groups in consecutive order, then for group j we set the first $\frac{n}{pg}$ elements to be random numbers between $\left((jg+\frac{p}{2}) \bmod p\right)\frac{\mathrm{MAX}}{p}$ and $\left(\left((jg+\frac{p}{2}+1) \bmod p\right)\frac{\mathrm{MAX}}{p}-1\right)$, the second $\frac{n}{pg}$ elements at each processor to be random numbers between $\left((jg+\frac{p}{2}+1) \bmod p\right)\frac{\mathrm{MAX}}{p}$ and $\left(\left((jg+\frac{p}{2}+2) \bmod p\right)\frac{\mathrm{MAX}}{p}-1\right)$, and so forth.

6. **Staggered [S]**, created as follows: if the processor index i is $<\frac{p}{2}$, then we set all $\frac{n}{p}$ elements at that processor to be random numbers between $(2i+1)\frac{\mathrm{MAX}}{p}$ and $\left((2i+2)\frac{\mathrm{MAX}}{p}-1\right)$, and so forth. Otherwise, we set all $\frac{n}{p}$ elements to be random numbers between $(i-\frac{p}{2})\frac{\mathrm{MAX}}{p}$ and $\left((i-\frac{p}{2}+1)\frac{\mathrm{MAX}}{p}-1\right)$, and so forth.

We selected these six benchmarks for a variety of reasons. Previous researchers have used the **Uniform**, **Gaussian**, and **Zero** benchmarks, and so we too included them for purposes of comparison. But benchmarks should be designed to illicit the worst case behavior from an algorithm, and in this sense the **Uniform** benchmark is not appropriate. For example, for $n \gg p$, one would expect that the optimal choice of the *splitters* in the **Uniform** benchmark would be those which partition the range of possible values into equal intervals. Thus, algorithms which try to guess the *splitters* might perform misleadingly well on such an input. In this respect, the **Gaussian** benchmark is more telling. But we also wanted to find benchmarks which would evaluate the cost of irregular communication. Thus, we wanted to include benchmarks for which an algorithm which uses a single phase of routing would find contention difficult or even impossible to avoid. A naive approach to rearranging the data would perform poorly on the **Bucket Sorted** benchmark. Here, every processor would try to route data to the same processor at the same time, resulting in poor utilization of communication bandwidth. This problem might be avoided by an algorithm in which at each processor the elements are first grouped by destination and then routed according to the specifications of a sequence of p destination permutations. Perhaps the most straightforward way to do this is by iterating over the possible communication strides. But such a strategy would perform poorly with the g-**Group** benchmark, for a suitably chosen value of g. In this case, using stride iteration, those processors which belong to a particular group all route data to the same subset of g destination processors. This subset of destinations is selected so that, when the g processors route to this subset, they choose the processors in exactly the same order, producing contention and possibly stalling. Alternatively, one can synchronize the processors after each permutation, but this in turn will reduce the communication bandwidth by a factor of $\frac{p}{g}$. In the worst case scenario, each processor needs to send data to a single processor a unique stride away. This is the case of the **Staggered** benchmark, and the result is a reduction of the communication bandwidth by a factor of p. Of course, one can correctly object that both the **g-Group** benchmark and the **Staggered** benchmark have been tailored to thwart a routing scheme which iterates over the possible strides, and that another sequences of permutations might be found which performs better. This is possible, but at the same time we are unaware of any single phase deterministic algorithm which could avoid an equivalent challenge.

4.2 Experimental Results

For each experiment, the input is evenly distributed amongst the processors. The output consists of the elements in non-descending order arranged amongst the processors so that the elements at each processor are in sorted order and no element at processor P_i is greater than any element at processor P_j, for all $i < j$.

Two variations were allowed in our experiments. First, radix sort was used to sequentially sort *integers*, whereas merge sort was used to sort double precision floating point numbers (*doubles*). Second, different implementations of the communication primitives were allowed for each machine. Wherever possible, we tried to use the vendor supplied implementations. In fact, IBM does provide all of our communication primitives as part of its machine specific Collective Communication Library (CCL) [8]. As one might expect, they were faster than the high level SPLIT-C implementation.

The graphs in **Figures 1** and **2** display the performance of our sample sort as a function of input distribution for a variety of input sizes. In each case, the performance is essentially independent of the input distribution. These figures present results obtained on a 64 node Cray T3D; results obtained from other machines validate this claim as well. Because of this independence, the remainder of this section will only discuss the performance of our sample sort on the single benchmark **[U]**.

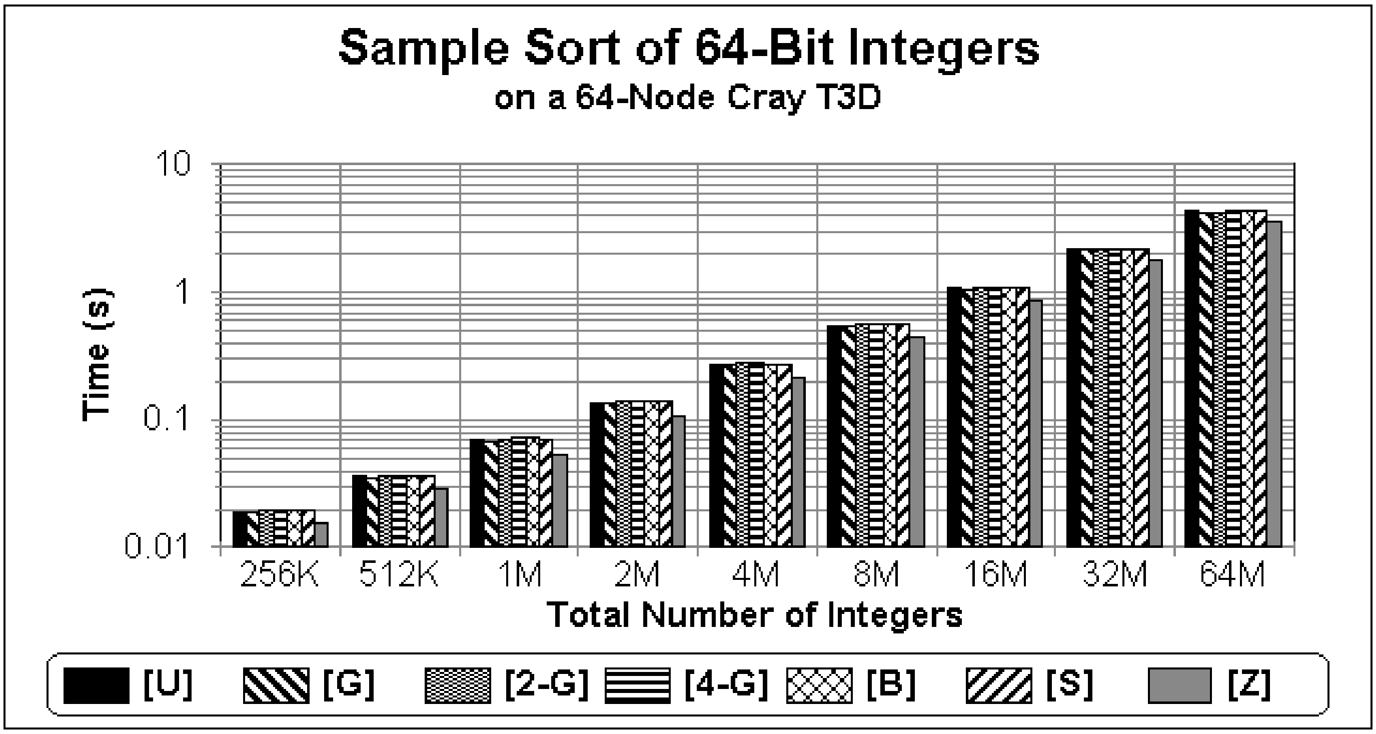


Figure 1: Performance is independent of input distribution for *integers*.

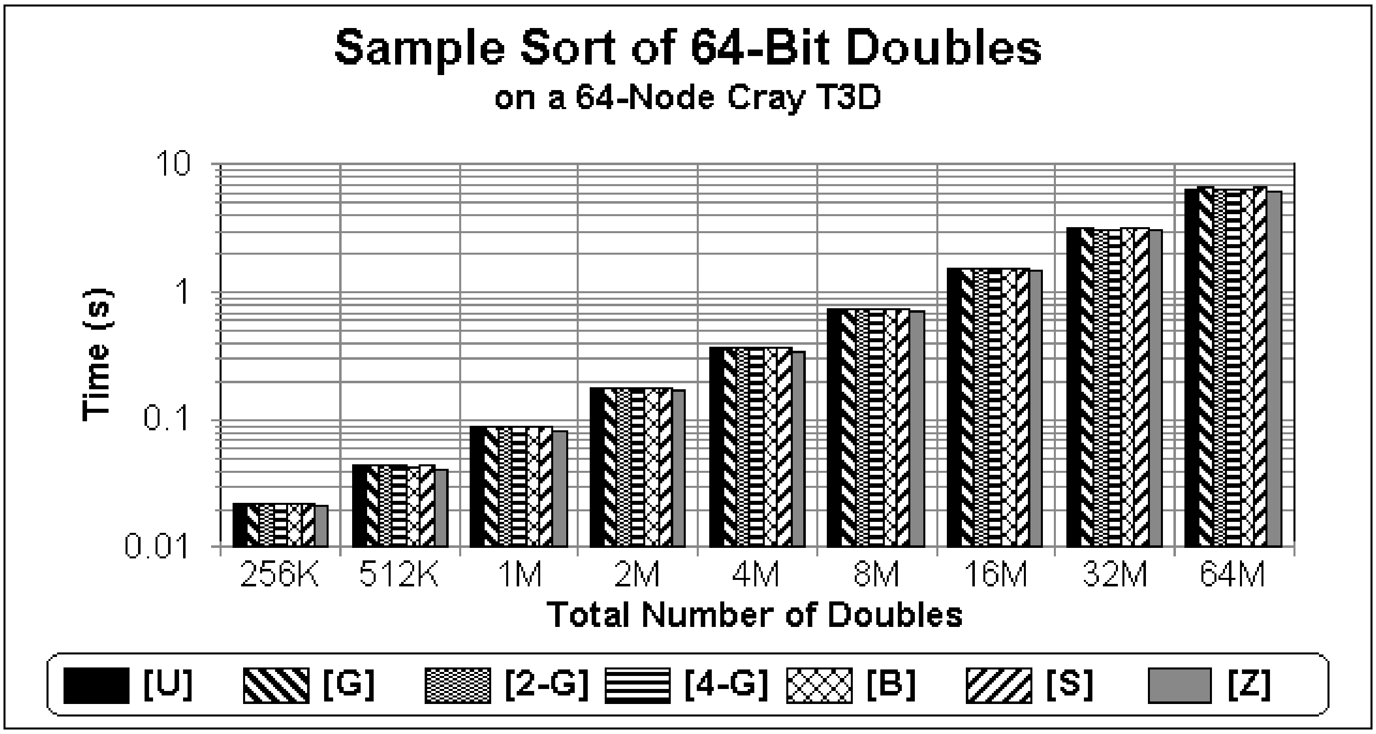


Figure 2: Performance is independent of input distribution for *doubles*.

Sample Sorting of 4M Integers						
	Number of Processors					
Machine	4	8	16	32	64	128
CRAY T3D	-	1.66	0.894	0.486	0.272	0.149
IBM SP2-WN	2.04	1.03	-	-	-	-
IBM SP2-TN2	2.97	1.34	0.755	-	-	-
TMC CM-5	-	-	3.61	1.67	0.761	0.444

Table I: Total execution time (in seconds) for sorting 4M *integers* on a variety of machines and processors. A hyphen indicates that that particular platform was unavailable to us.

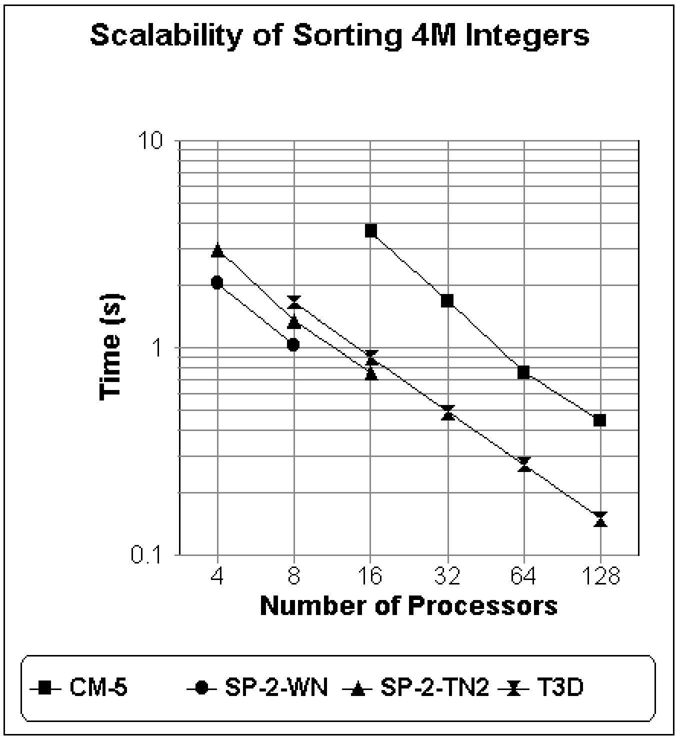
The results in **Tables I** and **II** together with their graphs in **Figure 3** examine the scalability of our sample sort as a function of machine size. Results are shown for the CM-5, the SP-2-WN, the SP2-TN2, and the T3D. Bearing in mind that these graphs are log-log plots, they show that for a given input size n the execution time scales almost inversely with the number of processors p. While this is certainly the expectation of our analytical model for *doubles*, it might at first appear to exceed our prediction of an $O(\frac{n}{p}\log p)$ computational complexity for *integers*. However, the appearance of an inverse relationship is still quite reasonable when we note that this $O(\frac{n}{p}\log p)$ complexity is entirely due to the merging in **Step (8)**, and in practice, as we show later with **Figure 6**, **Step (8)** only accounts for about 25% of the observed execution time. Note that the complexity of **Step 8** could be reduced to $O(\frac{n}{p})$ for *integers* using radix sort, but the resulting execution time would be slower.

Figures 4 and **5** examine the scalability of our sample sort as a function of problem size, for differing numbers of processors. They show that for a fixed number of processors there is an almost linear dependence between the execution time and the total number of elements n. While this is certainly the expectation of our analytic model for *integers*, it might at first appear to exceed our prediction of a $O(\frac{n}{p}\log n)$ computational complexity for floating point values. However, this appearance of a linear relationship is still quite reasonable when we consider that for the range of values shown $\log n$ differs by only a factor of 1.2.

Next, the graphs in **Figures 6** and **7** examine the relative costs of the eight steps in our sample sort on a 64 node T3D. Notice that the sequential sorting and merging performed in **Steps (3)** and

Sample Sorting of 4M Doubles						
	Number of Processors					
Machine	4	8	16	32	64	128
CRAY T3D	-	2.61	1.32	0.683	0.361	0.191
IBM SP2-WN	6.96	3.67	-	-	-	-
IBM SP2-TN2	8.78	4.43	2.28	-	-	-
TMC CM-5	-	-	6.51	3.31	1.85	0.915

Table II: Total execution time (in seconds) for sorting 4M *doubles* on a variety of machines and processors. A hyphen indicates that that particular platform was unavailable to us.



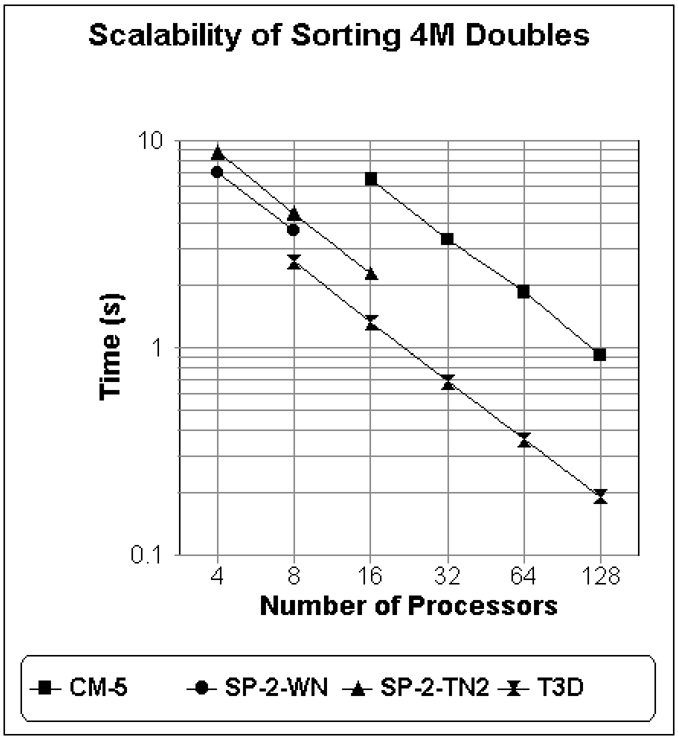


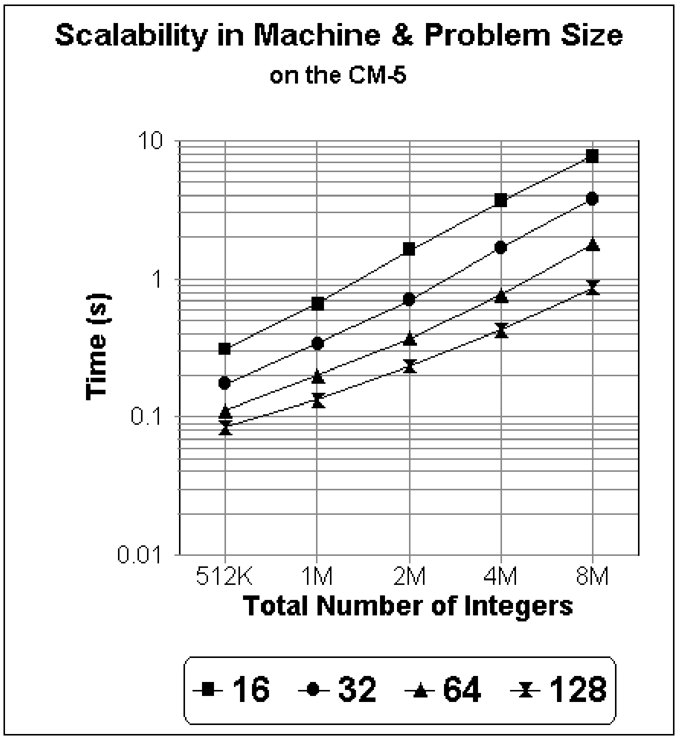
Figure 3: Scalability of sorting *integers* and *doubles* with respect to machine size.

(8) consume nearly 80% of the execution time, whereas the two **transpose** operations in **Steps (2)** and **(7)** together consume only about 20% of the execution time (and less for *doubles*). Similar results were obtained for all of our benchmarks, showing that our algorithm is extremely efficient in its communication performance.

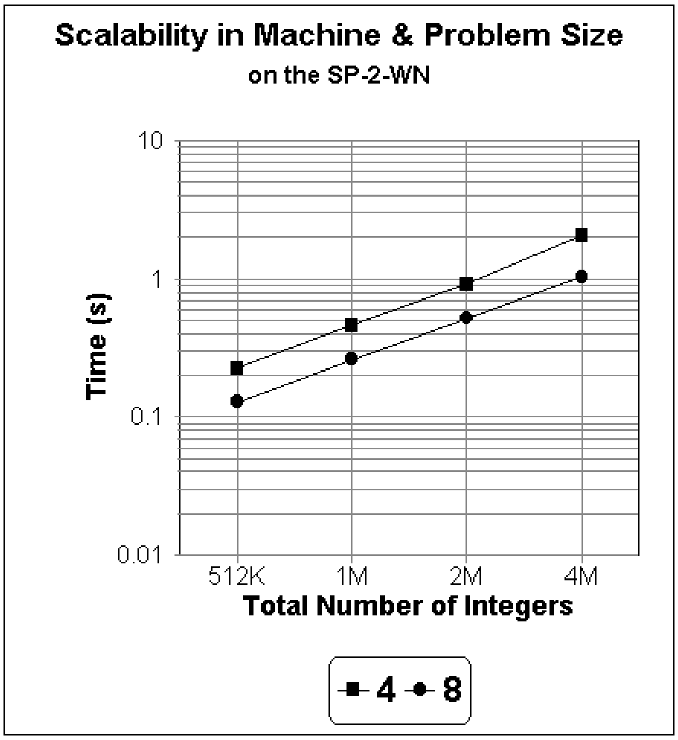
Finally, **Table III** shows the experimentally derived expected value (E) and sample standard deviation (STD) of the coefficients c_1, α_1, c_2, and α_2 used to describe the complexity of our algorithm in **Section 3**. For each input size, the values were obtained by analyzing data collected while sorting the **[G]**, **[B]**, **[2-G]**, **[4-G]**, and **[S]** benchmarks. Each of these benchmarks was generated and sorted

keys/proc	**E**(c_1)	**STD**(c_1)	**E**(α_1)	**STD**(α_1)	**E**(c_2)	**STD**(c_2)	**E**(α_2)	**STD**(α_2)
4K	2.02	0.091	1.08	0.017	2.64	0.94	1.55	0.18
8K	1.70	0.066	1.06	0.012	1.98	0.40	1.37	0.11
16K	1.48	0.040	1.04	0.007	1.66	0.23	1.25	0.07
32K	1.33	0.031	1.03	0.005	1.43	0.13	1.18	0.05
64K	1.23	0.025	1.02	0.003	1.29	0.08	1.12	0.03
128K	1.16	0.012	1.01	0.002	1.20	0.05	1.09	0.02
256K	1.11	0.011	1.01	0.002	1.14	0.04	1.06	0.02
512K	1.08	0.008	1.01	0.001	1.10	0.02	1.05	0.01
1M	1.06	0.004	1.00	0.001	1.07	0.02	1.03	0.01

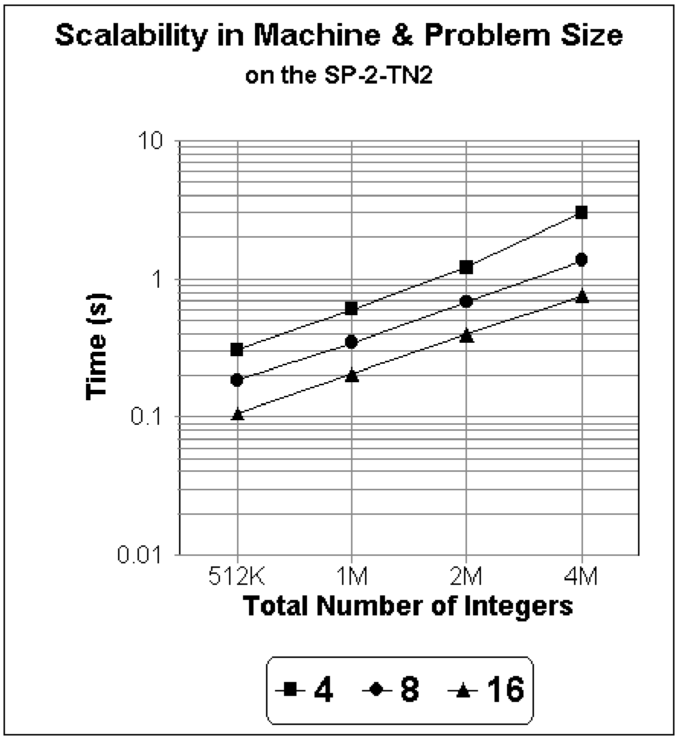
Table III: Statistical evaluation of the experimentally observed values of the algorithm coefficients on a 64 node T3D.



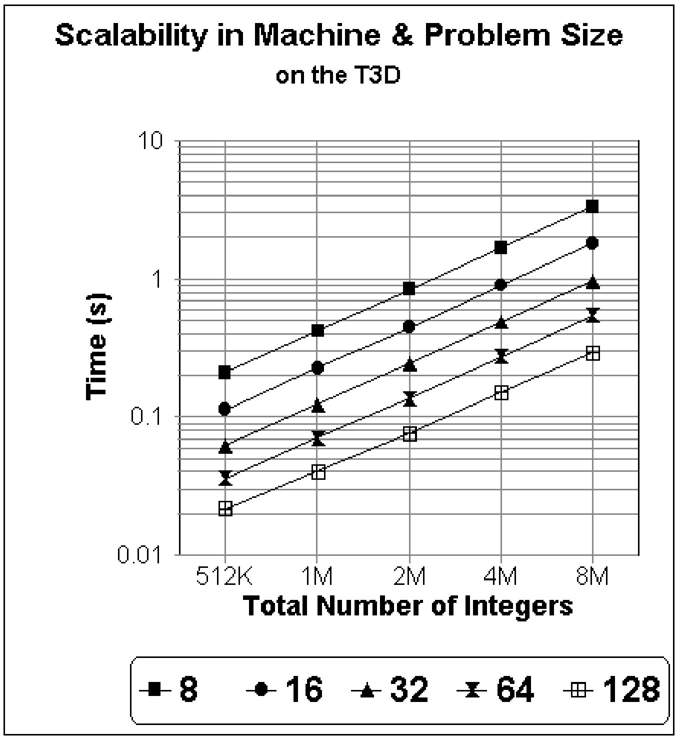
TMC CM-5



IBM SP-2-WN

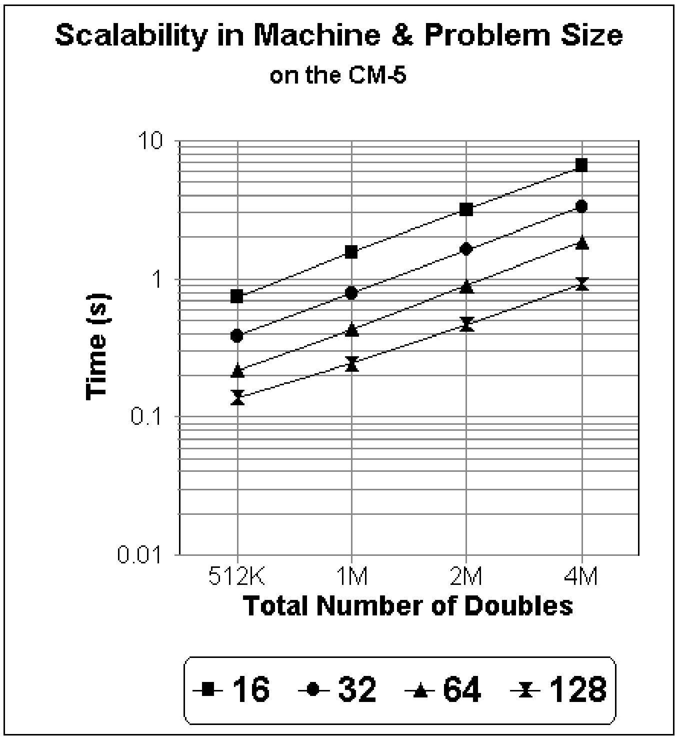


IBM SP-2-TN2

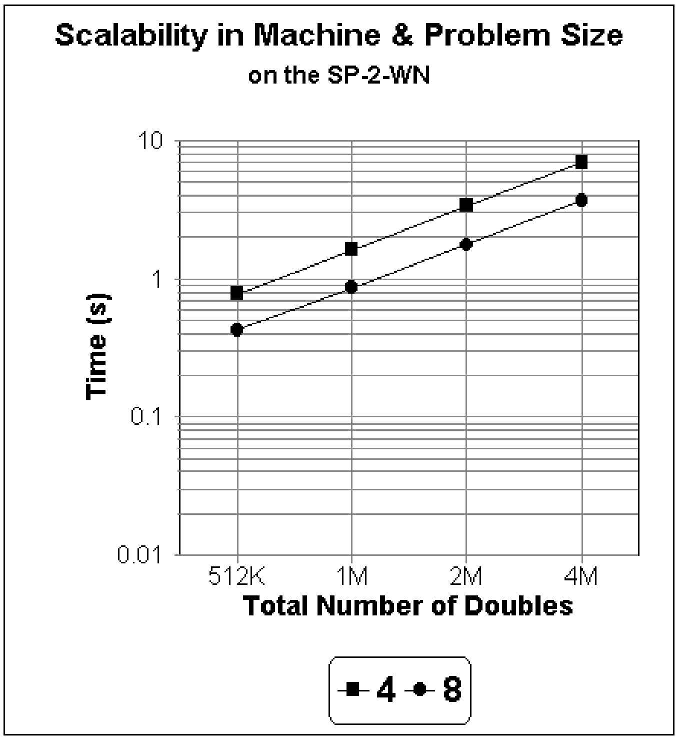


Cray T3D

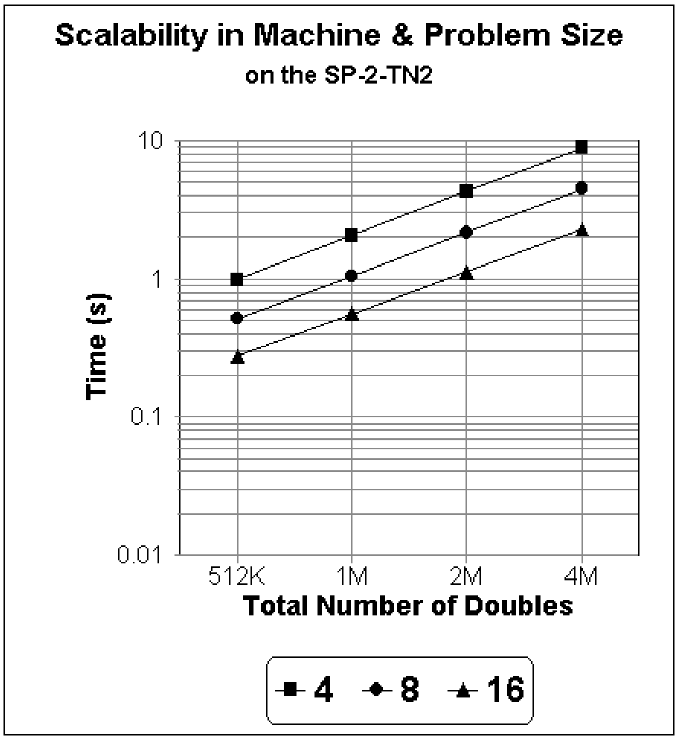
Figure 4: Scalability of sorting *integers* with respect to problem size, for differing numbers of processors.



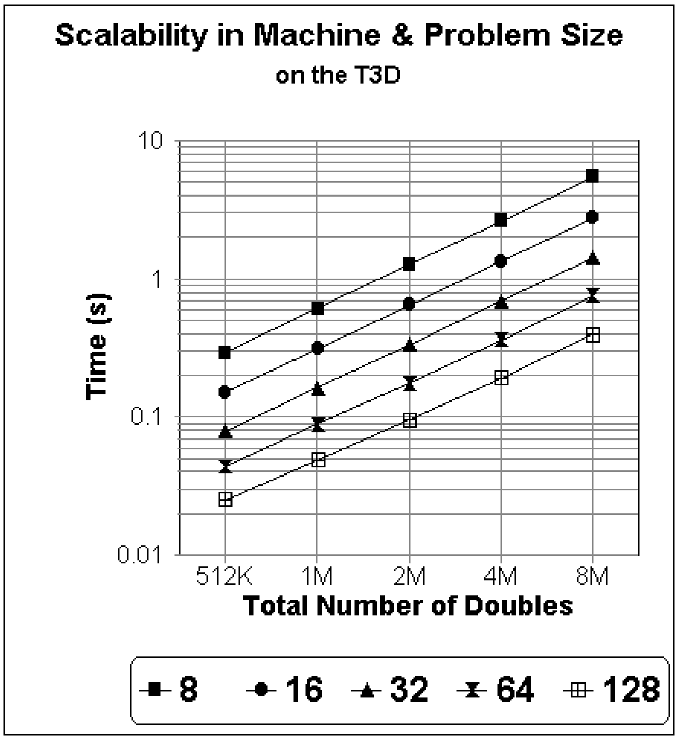
TMC CM-5



IBM SP-2-WN



IBM SP-2-TN2



Cray T3D

Figure 5: Scalability of sorting *doubles* with respect to problem size, for differing numbers of processors.

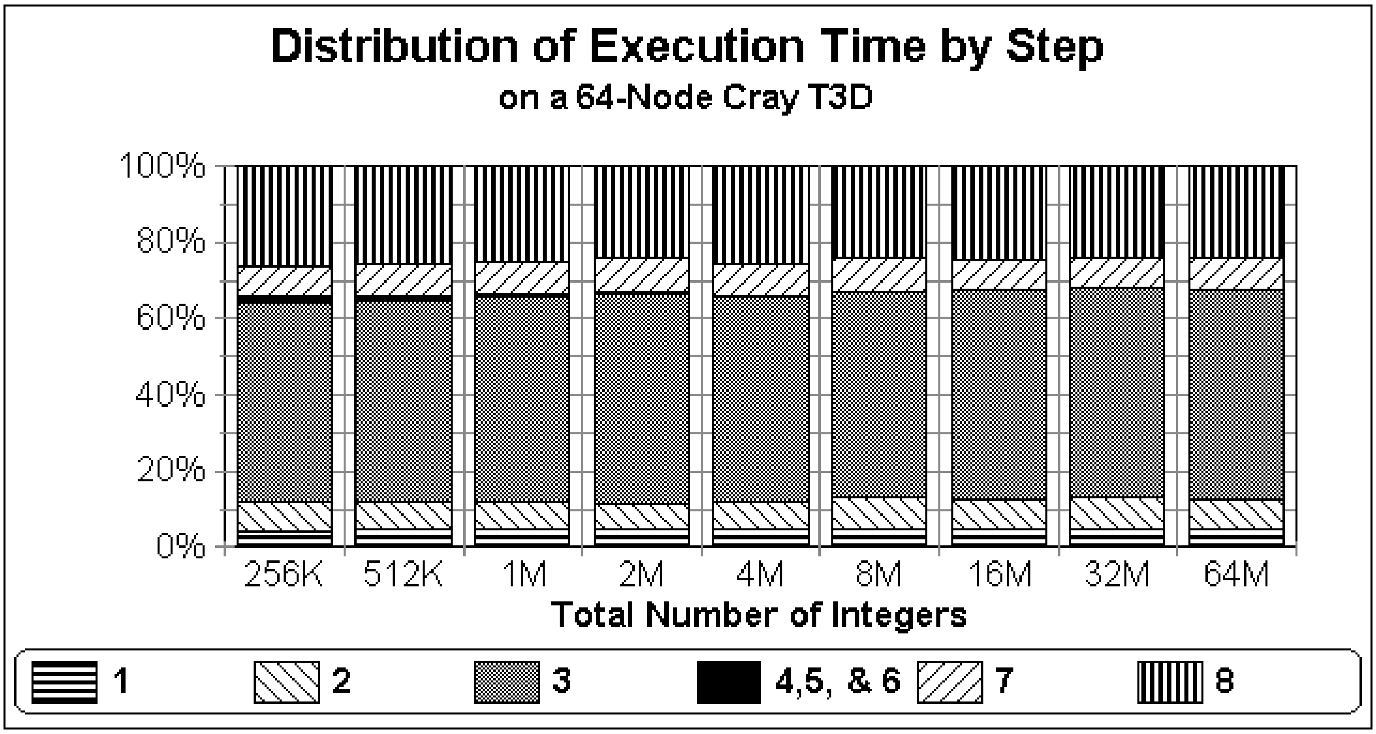


Figure 6: Distribution of execution time amongst the eight steps of sample sort for *integers*. Times are obtained on a 64 node T3D.

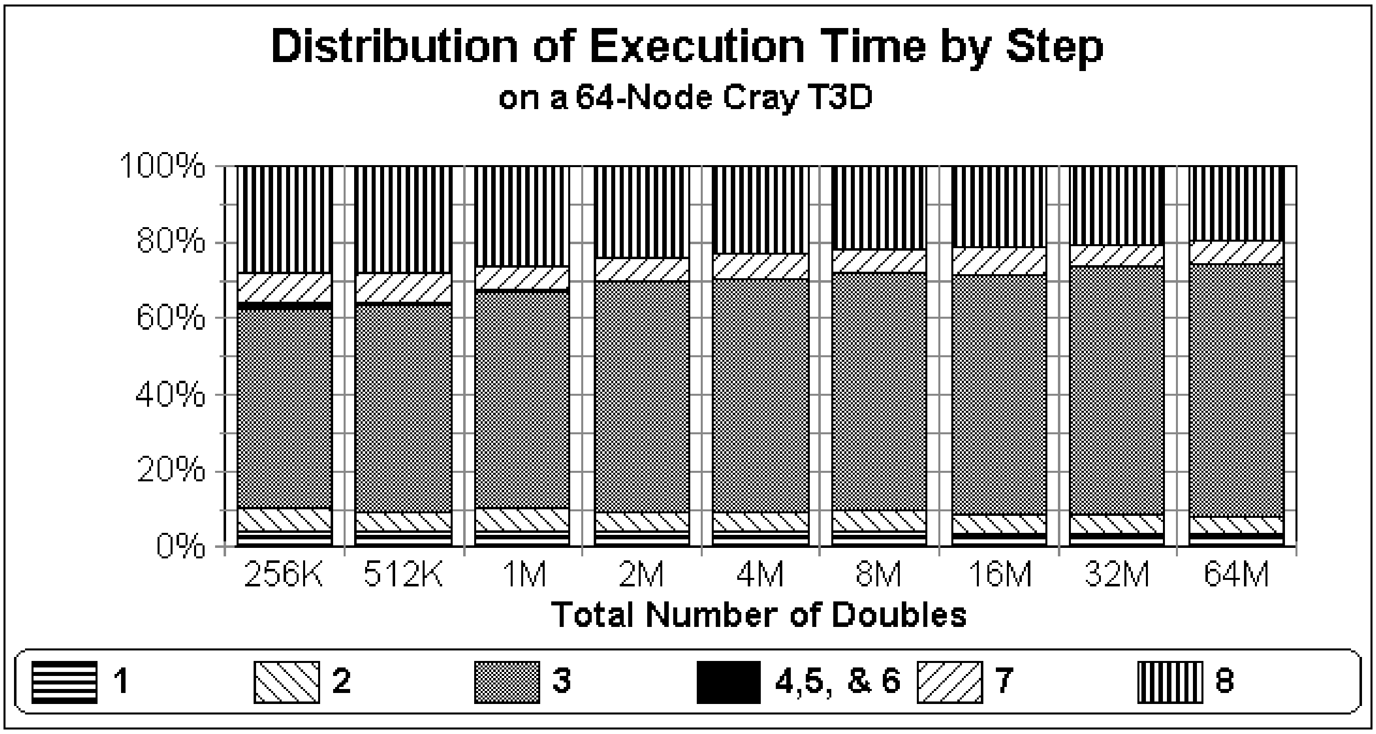


Figure 7: Distribution of execution time amongst the eight steps of sample sort for *doubles*. Times are obtained on a 64 node T3D.

20 times, each time using a different seed for the random number generator. The experimentally derived values for c_1, α_1, c_2, and α_2 agree closely with the theoretically derived values of c_1 (2), $\alpha_1 \leq c_1$, c_2 (2.48), and α_2 (1.33) for $p^2 \leq \frac{n}{3 \ln n}$.

4.3 Comparison with Previous Results

Despite the enormous theoretical interest in parallel sorting, we were able to locate relatively few empirical studies. Of these, only a few were done on machines which either were available to us for comparison or involved code which could be ported to these machines for comparison. In **Tables IV** and **V**, we compare the performance of our sample sort algorithm with two other sample sort algorithms. In all cases, the code was written in SPLIT-C. In the case of Alexandrov et al. [1], the times were determined by us directly on a 32 node CM-5 using code supplied by the authors which had been optimized for a Meiko CS-2. In the case of Dusseau [17], the times were obtained from the graphed results reported for a 64 node CM-5.

	[U]		[G]		[2-G]		[B]		[S]	
int./proc.	**HBJ**	**AIS**	**HBJ**	**AIS**	**HBJ**	**AIS**	**HBJ**	**AIS**	**HBJ**	**AIS**
4K	0.051	0.153	0.050	0.152	0.051	1.05	0.055	0.181	0.049	†
8K	0.090	0.197	0.090	0.192	0.092	1.09	0.094	0.193	0.087	†
16K	0.183	0.282	0.182	0.281	0.184	1.16	0.189	0.227	0.179	†
32K	0.360	0.450	0.359	0.449	0.363	1.34	0.364	0.445	0.361	†
64K	0.725	0.833	0.730	0.835	0.735	1.76	0.731	0.823	0.740	†
128K	1.70	2.02	1.70	2.02	1.70	2.83	1.72	1.99	2.02	†
256K	3.81	4.69	3.80	4.59	3.80	5.13	3.81	4.56	4.69	†
512K	8.12	10.0	8.04	9.91	8.11	9.58	8.10	9.98	10.0	†

Table IV: Total execution time (in seconds) required to sort a variety of benchmarks and problem sizes, comparing our version of sample sort **(HBJ)** with that of Alexandrov et al. **(AIS)** on a 32-node CM-5. †We were unable to run the **(AIS)** code on this input.

	[U]		[B]		[Z]	
int./proc.	**HBJ**	**DUS**	**HBJ**	**DUS**	**HBJ**	**DUS**
1M	16.6	21	12.2	91	10.6	11

Table V: Time required per element (in microseconds) to sample sort 64M *integers*, comparing our results **(HBJ)** with those obtained from the graphed results reported by Dusseau **(DUS)** on a 64 node CM-5.

Finally, there are the results for the NAS Parallel Benchmark [31] for integer sorting (IS). The name of this benchmark is somewhat misleading. Instead of requiring that the integers be placed in sorted order as we do, the benchmark only requires that they be ranked without any reordering, which is a significantly simpler task. **Table VI** compares our results on the Class A NAS Benchmark with the best times reported for the TMC CM-5 and the Cray T3D. We believe that our results, which were obtained using high-level, portable code, compare favorably with the other reported times, which were obtained by the vendors using machine-specific implementations and perhaps system modifications.

Comparison of Class A NAS (IS) Benchmark Times			
Machine	**Number of Processors**	**Best Reported Time**	**Our Time**
CM-5	32	43.1	29.4
	64	24.2	14.0
	128	12.0	7.13
Cray T3D	16	7.07	12.6
	32	3.89	7.05
	64	2.09	4.09
	128	1.05	2.26

Table VI: Comparison of our execution time (in seconds) with the best reported times for the Class A NAS Parallel Benchmark for integer sorting. Note that while we actually place the integers in sorted order, the benchmark only requires that they be ranked without actually reordering.

The only performance studies we are aware of on similar platforms for generalized sorting are those of Tridgell and Brent [33], who report the performance of their algorithm using a 32 node CM-5 on a uniformly distributed random input of signed integers, as described in **Table VII**.

Problem Size	**[U]**	
	(HBJ)	**(TB)**
8M	4.57	5.48

Table VII: Total execution time (in seconds) required to sort 8M signed integers, comparing our results **(HBJ)** with those of Tridgell and Brent **(TB)** on a 32 node CM-5.

5 Conclusion

In this paper, we introduced a novel variation on sample sort and conducted an experimental study of its performance on a number of platforms using widely different benchmarks. Our results illustrate the efficiency and scalability of our algorithm across the different platforms and appear to improve on all similar results known to the authors. Our results also compare favorably with those reported for the simpler ranking problem posed by the NAS Integer Sorting (IS) Benchmark.

We have also studied several variations on our algorithm which use differing strategies to ensure that every bucket in **Step (1)** receives an equal number of elements. The results obtained for these variations were very similar to those reported in this paper. On no platform did the improvements exceed approximately 5%, and in many instances they actually ran more slowly. We believe that a significant improvement of our algorithm would require the enhancement of the sequential sorting and merging in **Steps (3)** and **(8)**, and that there is little room for significant improvement in either the load balance or the communication efficiency.

6 Acknowledgements

We would like to thank Ronald Greenberg of UMCP's Electrical Engineering Department for his valuable comments and encouragement.

We would also like to thank the CASTLE/Split-C group at The University of California, Berkeley, especially for the help and encouragement from David Culler, Arvind Krishnamurthy, and Lok Tin Liu.

The University of California, Santa Barbara, parallel radix sort code was provided to us by Mihai Ionescu. Also, Klaus Schauser, Oscar Ibarra, Chris Scheiman, and David Probert of UC Santa Barbara, provided help and access to the UCSB 64-node Meiko CS-2. The Meiko CS-2 Computing Facility was acquired through NSF CISE Infrastructure Grant number CDA-9218202, with support from the College of Engineering and the UCSB Office of Research, for research in parallel computing.

Arvind Krishnamurthy provided additional help with his port of Split-C to the Cray Research T3D [2]. The Jet Propulsion Lab/Caltech 256-node Cray T3D Supercomputer used in this investigation was provided by funding from the NASA Offices of Mission to Planet Earth, Aeronautics, and Space Science. We also acknowledge William Carlson and Jesse Draper from the Center for Computing Science (formerly Supercomputing Research Center) for writing the parallel compiler AC (version 2.6) [11] on which the T3D port of Split-C has been based.

This work also utilized the CM-5 at National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, under grant number ASC960008N.

We would like to acknowledge the use of the UMIACS 16-node IBM SP-2-TN2, which was provided by an IBM Shared University Research award and an NSF Academic Research Infrastructure Grant No. CDA9401151.

Please see `http://www.umiacs.umd.edu/~dbader` for additional performance information. In addition, all the code used in this paper will be freely available for interested parties from our anonymous ftp site, `ftp://ftp.umiacs.umd.edu/pub/dbader`. We encourage other researchers to compare with our results for similar inputs.

References

[1] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One step closer towards a realistic model for parallel computation. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, Santa Barbara, CA, July 1995.

[2] R.H. Arpaci, D.E. Culler, A. Krishnamurthy, S.G. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In ACM Press, editor, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 320–331, Santa Margherita Ligure, Italy, June 1995.

[3] D.A. Bader, D.R. Helman, and J. JáJá. Practical Parallel Algorithms for Personalized Communication and Integer Sorting. CS-TR-3548 and UMIACS-TR-95-101 Technical Report, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, November 1995. Submitted to *ACM Journal of Experimental Algorithmics*.

[4] D.A. Bader and J. JáJá. Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study. Technical Report CS-TR-3384 and UMIACS-TR-94-133, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, December 1994. To appear in *Journal of Parallel and Distributed Computing*.

[5] D.A. Bader and J. JáJá. Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study. In *Fifth ACM SIGPLAN Symposium of Principles and Practice of Parallel Programming*, pages 123–133, Santa Barbara, CA, July 1995.

[6] D.A. Bader and J. JáJá. Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection. Technical Report CS-TR-3494 and UMIACS-TR-95-74, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, July 1995. To be presented at the 10th *International Parallel Processing Symposium*, Honolulu, HI, April 15-19, 1996.

[7] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical Report RNR-94-007, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Moffett Field, CA, March 1994.

[8] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and M. Snir. CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 6:154–164, 1995.

[9] K. Batcher. Sorting Networks and Their Applications. In *Proceedings of the AFIPS Spring Joint Computer Conference 32*, pages 307–314, Reston, VA, 1968.

[10] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[11] W.W. Carlson and J.M. Draper. AC for the T3D. Technical Report SRC-TR-95-141, Supercomputing Research Center, Bowie, MD, February 1995.

[12] H. Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations. *Annals of Math. Stat.*, 23:493–509, 1952.

[13] Cray Research, Inc. *SHMEM Technical Note for C*, October 1994. Revision 2.3.

[14] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Portland, OR, November 1993.

[15] D.E. Culler, A.C. Dusseau, R.P. Martin, and K.E. Schauser. Fast Parallel Sorting Under LogP: From Theory to Practice. In *Portability and Performance for Parallel Processing*, chapter 4, pages 71–98. John Wiley & Sons, 1993.

[16] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[17] A.C. Dusseau. Modeling Parallel Sorts with LogP on the CM-5. Technical Report UCB//CSD-94-829, Computer Science Division, University of California, Berkeley, 1994.

[18] T. Hagerup and C. Rüb. A Guided Tour of Chernoff Bounds. *Information Processing Letters*, 33:305–308, 1990.

[19] W.L. Hightower, J.F. Prins, and J.H. Reif. Implementations of Randomized Sorting on Large Parallel Machines. In *Proceedings of the 4th Symposium on Parallel Architectures and Algorithms*, pages 158–167, San Diego, CA, July 1992.

[20] J.S. Huang and Y.C. Chow. Parallel Sorting and Data Partitioning by Sampling. In *Proceedings of the 7th Computer Software and Applications Conference*, pages 627–631, November 1983.

[21] J. JáJá and K.W. Ryu. The Block Distributed Memory Model. Technical Report CS-TR-3207, Computer Science Department, University of Maryland, College Park, January 1994. To appear in *IEEE Transactions on Parallel and Distributed Systems*.

[22] J.F. JáJá and K.W. Ryu. The Block Distributed Memory Model for Shared Memory Multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 752–756, Cancún, Mexico, April 1994. (Extended Abstract).

[23] F.T. Leighton. Tight Bounds on the Complexity of Parallel Sorting. *IEEE Transactions on Computers*, C-34:344–354, 1985.

[24] H. Li and K.C. Sevcik. Parallel Sorting by Overpartitioning. Technical Report CSRI-295, Computer Systems Research Institute, University of Toronto, Canada, April 1994.

[25] X. Li, P. Lu, J. Schaeffer, J. Shillington, P.S. Wong, and H. Shi. On the Versatility of Parallel Sorting by Regular Sampling. *Parallel Computing*, 19:1079–1103, 1993.

[26] J.M. Marberg and E. Gafni. Sorting in Constant Number of Row and Column Phases on a Mesh. *Algorithmica*, 3:561–572, 1988.

[27] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, June 1995. Version 1.1.

[28] C.G. Plaxton. Efficient Computation on Sparse Interconnection Networks. Technical Report STAN-CS-89-1283, Department of Computer Science, Stanford University, Stanford, CA, September 1989.

[29] M.J. Quinn. Analysis and Benchmarking of Two Parallel Sorting Algorithms: Hyperquicksort and Quickmerge. *BIT*, 29:239–250, 1989.

[30] J.H. Reif and L.G. Valiant. A Logarithmic Time Sort for Linear Sized Networks. *Journal of the ACM*, 34:60–76, 1987.

[31] S. Saini and D.H. Bailey. NAS Parallel Benchmarks Results 12-95. Report NAS-95-021, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Moffett Field, CA, December 1995.

[32] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.

[33] A. Tridgell and R.P. Brent. An Implementation of a General-Purpose Parallel Sorting Algorithm. Techical Report TR-CS-93-01, Computer Sciences Laboratory, Australian National University, Canberra, Australia, February 1993.

[34] L.G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[35] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.