Parallel Algorithms for Personalized Communication and Sorting With an Experimental Study (Extended Abstract)

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

Abstract

A fundamental challenge for parallel computing is to obtain high-level, architecture independent, algorithms which execute efficiently on general-purpose parallel machines. With the emergence of message passing standards such as MPI, it has become easier to design efficient and portable parallel algorithms by making use of these communication primitives. While existing primitives allow an assortment of collective communication routines, they do not handle an important communication event when most or all processors have non-uniformly sized personalized messages to exchange with each other. We first present an algorithm for the **h-relation personalized communication** whose efficient implementation will allow high performance implementations of a large class of algorithms.

We then consider how to effectively use these communication primitives to address the problem of sorting. 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. Another variation using regular sampling for choosing the splitters has similar performance with deterministic guaranteed bounds on the memory and communication requirements. Both of these variations efficiently handle the presence of duplicates without the overhead of tagging each element.

The personalized communication and sorting algorithms presented in this paper have been coded in Split-C and run on a variety of platforms, including the Thinking Machines CM-5, IBM SP-2, Cray Research T3D, Meiko Scientific CS-2, and the Intel Paragon. Our experimental results are consistent with the theoretical analyses and illustrate the scalability and efficiency of our algorithms across different platforms. In fact, they seem to outperform all similar algorithms known to the authors on these platforms, and performance is invariant over the set of input distributions unlike previous efficient algorithms. Our sorting results also compare favorably with those reported for the simpler ranking problem posed by the NAS Integer Sorting (IS) Benchmark.

Keywords: Parallel Algorithms, Personalized Communication, Sorting, Sample Sort, Radix Sort, Parallel Performance.

1 Introduction

A fundamental problem in parallel computing is to design high-level, architecture independent, algorithms that execute efficiently on general-purpose parallel machines. The aim is to be able to achieve portability *and* high performance simultaneously. Note that it is considerably easier to achieve portability alone (say, by using PVM) or high performance (say, by using sophisticated programmers to fine tune the algorithm to the specific machine). There are currently two factors that make this fundamental problem more tractable. The first is the emergence of a dominant parallel architecture consisting of a number of powerful microprocessors interconnected by either a proprietary interconnect or a standard off-the-shelf interconnect. The second factor is the emergence of standards, such as the message passing standard MPI [20], for which machine builders and software developers will try to provide efficient support. Our work builds on these two developments by presenting a theoretical and an experimental framework for designing parallel algorithms. In this abstract, we sketch our contributions in two important problems: personalized communication and sorting. We start with a brief outline of the computation model.

We view a parallel algorithm as a sequence of local computations interleaved with communication steps, and we al-

*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.

low 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 a 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 (see below) 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 can be justified by using our earlier work [17, 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. the LogP [13], BSP [24], and LogGP [1] models) 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 $T_{comp}(n, p)$ as the maximum time taken by any processor to perform all of its local computation steps.

Our algorithms are implemented in SPLIT-C [11], an extension of C for distributed memory machines. The algorithms make use of MPI-like communication primitives but do not make any assumptions as to how these primitives are actually implemented. Our collective communication primitives, described in detail in [6], are similar to those of MPI [20], the IBM POWERparallel [7], and the Cray MPP systems [10] 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 called to broadcast a block of data from a single source to all the remaining processors. The primitives **gather** and **scatter** are companion primitives whereby **scatter** divides a single array residing on a processor into equal-sized blocks which are then distributed to the remaining processors, and **gather** coalesces these blocks residing on the different processors into a single array on one processor.

2 Main Results and Significance

In this section we state the main results achieved for solving the problems of personalized communication and sorting.

Problem 1 (h-Relation Personalized Communication): A set of n messages is to be routed such that each processor is the origin or destination of at most h messages.

Result: A new deterministic algorithm using two rounds of the **transpose** primitive, with optimal complexity and very small constant coefficients, is shown to be very efficient and scalable across different platforms and over different input distributions.

Significance: The importance of this problem has been stated in many papers (e.g. [24, 22]). Our algorithm seems to achieve the best known experimental results for this problem. The general approach was independently described by Kaufmann et al. [18] and Ranka et al. [21] around the same time as our earlier draft [3] appeared, but our algorithm is simpler, has less overhead, and has a tighter bound size for the intermediate collective communication.

Problem 2 (Sorting): Rearrange n equally distributed elements amongst p processors such that they appear in non-decreasing order starting from the smallest indexed processor.

Results: (1) A novel variation of sample sort that uses only two rounds of regular all-to-all personalized communication and that maintains very good load balancing with virtually no overhead. (2) A new deterministic algorithm that achieves almost the same performance as (1) by using regular sampling. (3) An efficient implementation of radix sort that makes use of our personalized communication scheme.

Significance: We have developed a suite of benchmarks and conducted extensive experimentations related to Problem 2. Our algorithms have consistently performed very well across the different benchmarks and the different platforms. Algorithm (1) has outperformed all similar results known to the authors on our platforms. It even compares favorably to the machine-specific implementations reported by some of the machine vendors for the Class A NAS IS Benchmark, which only requires the easier task of ranking. Algorithm (2) has almost the same performance as (1) while guaranteeing memory and communication bounds, and Algorithm (3) achieves the best known execution times for a stable integer sorting algorithm. Additionally, all of our algorithms efficiently handle the presence of duplicates without the overhead of tagging each element.

3 Personalized Communication

For ease of presentation, we describe the personalized communication algorithm for the case when the input is initially evenly distributed amongst the processors. The reader is directed to [4] for the general case. Consider a set of n elements evenly distributed amongst p processors in such a manner that no processor holds more than $\frac{n}{p}$ elements. Each element consists of a pair $\langle data, dest \rangle$, where $dest$ is the location to where the $data$ is to be routed. There are no assumptions made about the pattern of data redistribution, except that no processor is the destination of more than h elements and thus $h \geq \frac{n}{p}$. We assume for simplicity (and without loss of generality) that h is an integral multiple of p.

3.1 Algorithm

In our solution, we use two rounds of the **transpose** collective communication primitive. In the first round, each element is routed to an intermediate destination, and during the second round, it is routed to its final destination.

The pseudocode for our algorithm is as follows:

- **Step (1):** Each processor P_i, for $(0 \leq i \leq p-1)$, assigns its $\frac{n}{p}$ elements to one of p bins according to the following rule: if element k is the first occurrence of an element with destination j, then it is placed into bin $(i+j) \bmod p$. Otherwise, if the last element with destination j was placed in bin b, then element k is placed into bin $(b+1) \bmod p$.

- **Step (2):** Each processor P_i routes the contents of bin j to processor P_j, for $(0 \leq i, j \leq p-1)$. Since we established that no bin can have more than $\frac{n}{p^2} + \frac{p}{2}$ elements [4], this is equivalent to performing a **transpose** communication primitive with block size $\frac{n}{p^2} + \frac{p}{2}$.

- **Step (3):** Each processor P_i rearranges the elements received in **Step (2)** into bins according to each element's final destination.

- **Step (4):** Each processor P_i routes the contents of bin j to processor P_j, for $(0 \leq i, j \leq p-1)$. Since we established that no bin can have more than $\frac{h}{p} + \frac{p}{2}$ elements [4], this is equivalent to performing a **transpose** primitive with block size $\frac{h}{p} + \frac{p}{2}$.

The overall complexity of our algorithm is $T_{comp}(n,p) + T_{comm}(n,p) = \mathrm{O}(h) + 2\left[\tau + \left(h + p^2\right)\sigma\right]$, which is asymptotically optimal with very small constants for $p^2 \leq n$.

3.2 Summary of Experimental Results

We examine the performance of our h-relation algorithm on various configurations of the IBM SP-2 and Cray T3D, using four values of h: $h = \frac{n}{p}, 2\frac{n}{p}, 4\frac{n}{p}$, and $8\frac{n}{p}$. More results

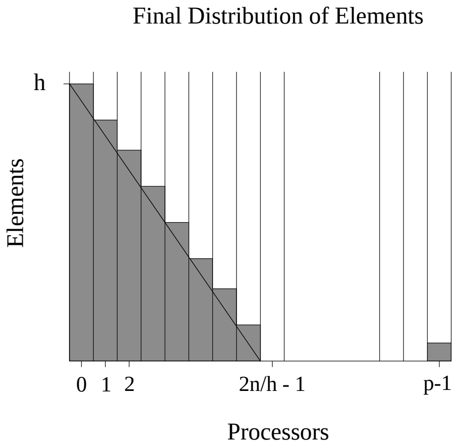


Figure 1: Final distribution of the keys corresponding to our input data sets

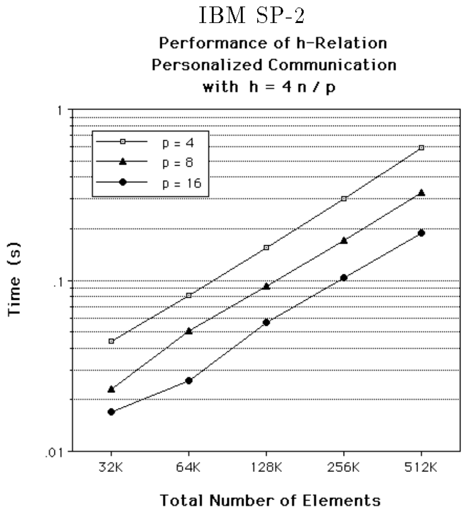
are given in [4]. The data sets used in these experiments are defined as follows. Our input of size n is initially distributed cyclically across the p processors such that each processor P_i initially holds $\frac{n}{p}$ keys, for $(0 \leq i \leq p-1)$. For $h = \frac{n}{p}$ the input consists of $v_0 = \frac{n}{p}$ keys labelled for P_0, followed by $v_1 = \frac{n}{p}$ keys labelled for P_1, and so forth, (with $v_i = \frac{n}{p}$ keys labelled for P_i), with the last $v_{p-1} = \frac{n}{p}$ keys labelled for P_{p-1}. Note that this results in the same data movement as the **transpose** primitive[1]. For $h > \frac{n}{p}$, instead of an equal number of elements destined for each processor, the function v_i, for $(0 \leq i \leq p-1)$, is characterized by

$$\begin{cases} \left\lfloor h\left(1 - \frac{h}{2n-h} i\right)\right\rfloor, & \text{if } i < \frac{2n}{h},\ i \neq p-1, \\ 0, & \text{if } \frac{2n}{h} \leq i < p-1, \\ n - \sum_{j=0}^{\frac{2n}{h}-1} \left\lfloor h\left(1 - \frac{h}{2n-h} j\right)\right\rfloor, & \text{if } i = p-1. \end{cases} \tag{1}$$

The result of this data movement, shown in Figure 1, is that processor 0 receives the largest imbalance of elements, i.e. h, while other processors receive varying block sizes ranging from 0 to at most h. For $h = 8\frac{n}{p}$, approximately $\frac{3p}{4}$ processors receive no elements, and hence this represents an extremely unbalanced case.

As shown in Figure 2, the time to route an h-relation personalized communication for a given input size on a varying number of processors (p) scales inversely with p whenever n is large compared with p. However, for inputs which are small compared with the machine size, the communication time is dominated by $\mathrm{O}\left(p^2\right)$ as shown in the case of the 128-processor Cray T3D with $n = 256K$.

[1] Note that the personalized communication is more general than a **transpose** primitive and does not make the assumption that data is already held in contiguous, regular sized buffers.



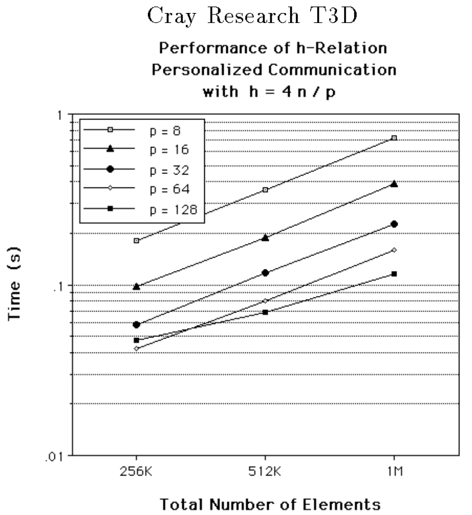


Figure 2: Performance of personalized communication ($h = 4\frac{n}{p}$) with respect to machine and problem size.

4 Sorting

4.1 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 [16, 8, 12] 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 second 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 [20]. The final difficulty with the original approach is that duplicate values are accommodated by tagging each item with some unique value [8]. This, of course, doubles the cost of both memory access and interprocessor communication.

In our version of sample sort, 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, and we are able to efficiently accommodate the presence of duplicates without resorting to tagging. 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 $\left(\alpha_1\frac{n}{p} \leq c_1\frac{n}{p}\right)$ 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 $\left(\beta \frac{n}{p} \leq c_1 \frac{n}{p}\right)$ elements, processor P_0 selects each $\left(j\beta \frac{n}{p^2}\right)^{th}$ element as Splitter$[j]$, for $(1 \leq j \leq p)$. By default, Splitter$[p]$ is the largest value allowed by the data type used. Additionally, for each Splitter$[j]$, binary search is used to define a value Frac$[j]$ which is the fraction of the total elements at processor P_0 with the same value as Splitter$[j]$ which also lie between index $\left((j-1)\,\beta \frac{n}{p^2}\right)$ and $\left(j\beta \frac{n}{p^2} - 1\right)$, inclusively.
- **Step (5):** Processor P_0 **broadcasts** the p *splitters* and their *frac* values to the other $p-1$ processors.
- **Step (6):** Each processor P_i uses binary search on its sorted local array to define for each of the p *splitters* a subsequence S_j. The subsequence associated with Splitter$[j]$ contains values which are greater than or equal to Splitter$[j-1]$ and and less then or equal to Splitter$[j]$, and includes Frac$[j]$ of the total number of elements in the local array with the same value as Splitter$[j]$.
- **Step (7):** Each processor P_i routes the subsequence associated with Splitter$[j]$ 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 $\geq \left(1 - n^{-\epsilon}\right)$ for some positive constant ϵ. But before doing this, we need the results of the following theorem, whose proof has been omitted for brevity [14].

Theorem 1: The number of elements in each bucket at the completion of **Step (1)** is at most $c_1 \frac{n}{p^2}$, the number of elements received by each processor at the completion of **Step (7)** is at most $\alpha_2 \frac{n}{p}$, and the number of elements exchanged by any two processors in **Step (7)** is at most $c_2 \frac{n}{p^2}$, all with high probability for any $c_1 \geq 2$, $\alpha_2 \geq 1.77$ ($\alpha_2 \geq 2.62$ for duplicates), $c_2 \geq 3.10$ ($c_2 \geq 4.24$ for duplicates), and $p^2 \leq \frac{n}{3 \ln n}$.

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 $O\left(\frac{n}{p}\right)$ time, whereas sorting floating point numbers using merge sort requires $O\left(\frac{n}{p} \log \frac{n}{p}\right)$ time. **Step (8)** requires $O\left(\frac{n}{p} \log p\right)$ 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 $T_{comm}(n,p) \leq \left(\tau + 2\frac{n}{p^2}(p-1)\,\sigma\right)$, $T_{comm}(n,p) \leq (\tau + p\sigma)$, and $T_{comm}(n,p) \leq \left(\tau + 4.24\frac{n}{p^2}(p-1)\,\sigma\right)$, respectively. Hence, with high probability, the overall complexity of our sample sort algorithm is given (for floating point numbers) by

$$\begin{aligned} T(n,p) &= T_{comp}(n,p) + T_{comm}(n,p) \\ &= O\left(\frac{n}{p} \log n + \tau + \frac{n}{p}\sigma\right) \end{aligned} \quad (2)$$

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

Clearly, our algorithm is asymptotically optimal with very small coefficients. But it is also important to perform an empirical evaluation of our algorithm using a wide variety of benchmarks. Our algorithm was implemented and tested on nine 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. The details and the rationale for these benchmarks are described in Appendix A. Table I displays the performance of our sample sort as a function of input distribution for a variety of input sizes. The results show that the performance is essentially independent of the input distribution. There is a slight preference for those benchmarks which contain high numbers of duplicates, but this is attributed to the reduced time required for sequential sorting in **Step (3)**.

Random Sample Sorting of Integers				
Sorting Benchmark	**Problem Size**			
	1M	**4M**	**16M**	**64M**
[U]	0.0698	0.276	1.07	4.29
[G]	0.0701	0.272	1.06	4.21
[2-G]	0.0716	0.275	1.08	4.24
[4-G]	0.0716	0.277	1.06	4.27
[B]	0.0707	0.272	1.08	4.27
[S]	0.0713	0.277	1.08	4.23
[Z]	0.0551	0.214	0.888	3.49
[WR]	0.0710	0.269	1.07	4.22
[RD]	0.0597	0.230	0.883	3.48

Table I: Total execution time (in seconds) for sample sorting a variety of benchmarks on a 64 node Cray T3D.

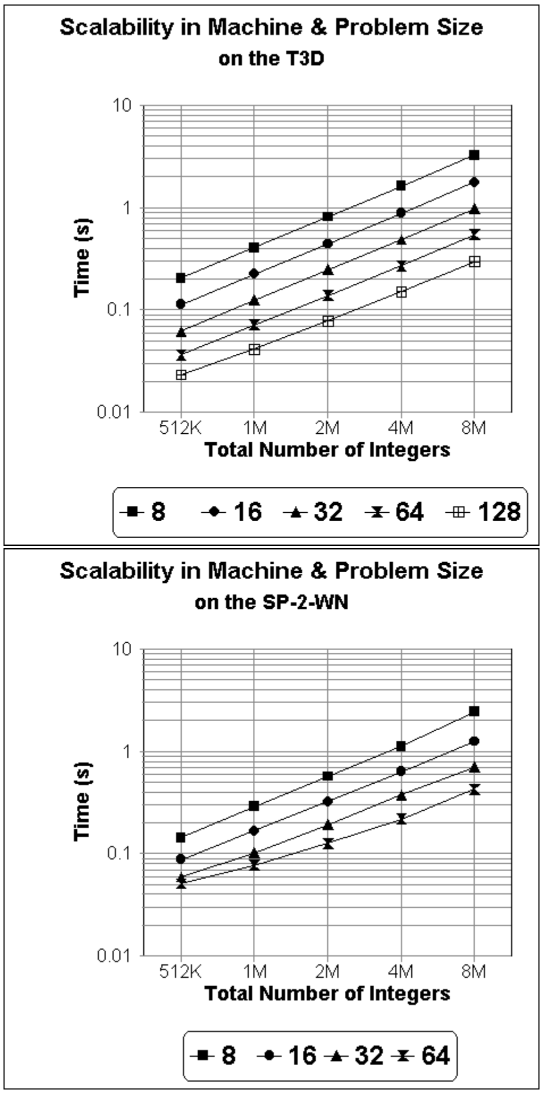


Figure 3: Scalability with respect to problem size of sample sorting *integers* from the **[U]** benchmark, for differing numbers of processors, on the Cray T3D and the IBM SP-2-WN.

Table II examines the scalability of our sample sort as a function of machine size. It shows that for a given input size n the execution time scales almost inversely with the number of processors p. Figure 3 examines the scalability of

Random Sample Sorting of 8M Integers					
	Number of Processors				
Machine	**8**	**16**	**32**	**64**	**128**
Cray T3D	3.21	1.74	0.966	0.540	0.294
IBM SP2-WN	2.41	1.24	0.696	0.421	-
TMC CM-5	-	7.20	3.65	1.79	0.849

Table II: Total execution time (in seconds) for sorting 8M *integers* from the **[WR]** benchmark on a variety of machines and processors. A hyphen indicates that that particular platform was unavailable to us.

our sample sort as a function of problem size, for differing numbers of processors. It shows that for a fixed number of processors there is an almost linear dependence between the execution time and the total number of elements n. Finally, Table III compares our results on the Class A NAS Benchmark for integer sorting (IS) with the best times reported for the TMC CM-5 and the Cray T3D. Note that 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. 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 III: 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.

See [14] for additional performance data and comparisons with other published results.

4.2 A New Sorting Algorithm by Regular Sampling

A disadvantage of our random sample sort algorithm is that the performance bounds and the memory requirements can only be guaranteed with high probability. The alternative to this is to choose the *samples* by regular sampling. A previous version of regular sample sort [23, 19], known as Parallel Sorting by Regular Sampling (PSRS), first sorts the $\frac{n}{p}$ elements at each processor and then selects every $\left(\frac{n}{p^2}\right)^{th}$ element as a *sample*. These *samples* are then routed to a single processor, where they are sorted and every p^{th} *sample* is selected as a *splitter*. Each processor then uses these *splitters* to partition the sorted input values and then routes the resulting subsequences to the appropriate destinations, after which local merging is done to complete the sorting process. The first difficulty with this approach is the load balance. There exist inputs for which at least one processor will be left at the completion of sorting with as many as $\left(2\frac{n}{p} - \frac{n}{p^2} - p + 1\right)$ elements. This could be reduced by choosing more *splitters*, but this would also increase the overhead. And no matter what is done, previous workers have observed that the load balance would still deteriorate linearly with the number of duplicates [19]. 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 [20].

In our algorithm, which is parameterized by a sampling ratio s $\left(p \leq s \leq \frac{n}{p^2}\right)$, we guarantee that at the completion of sorting, each processor will have at most $\left(\frac{n}{p} + \frac{n}{s} - p\right)$ elements, while incurring no overhead in gathering the *samples* to identify the *splitters*. This bound holds regardless of the number of duplicates present in the input. Moreover, 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 $(0 \leq i \leq p-1)$ sorts each of its $\frac{n}{p}$ input values 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. The sorted data is then "dealt" into p bins so that the k^{th} item in the sorted order is placed into the $\left\lfloor \frac{k}{p} \right\rfloor^{th}$ position of the $(k \bmod p)^{th}$ bin.

- **Step (2):** Each processor P_i routes the contents of bin j to processor P_j, for $(0 \leq i, j \leq p-1)$, which is equivalent to performing a **transpose** operation with block size $\frac{n}{p^2}$.

- **Step (3):** From each of the p sorted subsequences received in **Step (2)**, processor P_{p-1} selects each $\left(k\frac{n}{p^2 s}\right)^{th}$ element as a *sample*, for $(1 \leq k \leq s-1)$, and a given value of s, for $\left(p \leq s \leq \frac{n}{p^2}\right)$.

- **Step (4):** Processor P_{p-1} merges the p sorted subsequences of *samples* and then selects each $(ks)^{th}$ *sample* as a *splitter*, for $(1 \leq k \leq p-1)$. By default, the last splitter is the largest value allowed by the data type used. Additionally, binary search is used to compute for the set of samples SA_k with indices $(k-1)s$ through $(ks-1)$ the number of samples $\mathrm{Est}[k]$ which share the same value as $\mathrm{Splitter}[k]$. Note that $\left(\mathrm{Est}[k] \times \frac{n}{ps}\right)$ is an upper bound on the total number of duplicates amongst the samples in SA_k and the $\left(\frac{n}{ps} - 1\right)$ elements which proceed each sample in the sorted order of **Step (1)**. After **Step (2)**, the input is redistributed so so that each processor holds between $\left(\mathrm{Est}[k] \times \frac{n}{p^2 s}\right)$ and $\left((\mathrm{Est}[k]-1) \times \frac{n}{p^2 s} + 1\right)$ of those duplicates associated with the set SA_k in **Step (1)**.

- **Step (5):** Processor P_{p-1} **broadcasts** the p *splitters* and their *est* values to the other $p-1$ processors.

- **Step (6):** Each processor P_i uses binary search to define for each of the p sorted sequences received in **Step (2)** and each of the p splitters a subsequence $S_{(j,k)}$. The p subsequences associated with $\mathrm{Splitter}[j]$ all contain values which are greater than or equal to $\mathrm{Splitter}[j-1]$ and and less then or equal to $\mathrm{Splitter}[j]$, and collectively include at most $\left(\mathrm{Est}[j] \times \frac{n}{p^2 s}\right)$ elements with the same value as $\mathrm{Splitter}[j]$.

- **Step (7):** Each processor P_i routes the p subsequences associated with $\mathrm{Splitter}_j$ to processor P_j, for $(0 \leq i, j \leq p-1)$. Since no two processors will exchange more than $\left(\frac{n}{p^2} + \frac{n}{sp} - 1\right)$ elements, this is equivalent to performing a **transpose** operation with block size $\left(\frac{n}{p^2} + \frac{n}{sp} - 1\right)$.

- **Step (8):** Each processor P_i "unshuffles" all those subsequences sharing a common origin in **Step (2)**. The p consolidated subsequences are then merged to produce the i^{th} column of the sorted array.

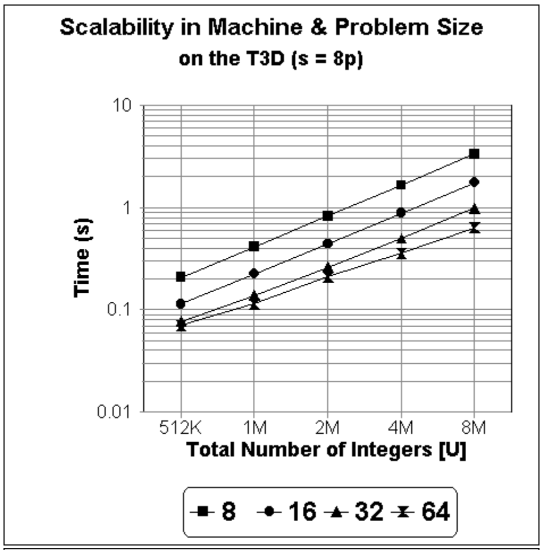
Before establishing the complexity of this algorithm, we need the results of the following theorem, whose proof has been omitted for brevity [15].

Theorem 2: The number of elements exchanged by any two processors in **Step (7)** is at most $\left(\frac{n}{p^2} + \frac{n}{sp} - 1\right)$. Consequently, at the completion of **Step (7)**, no processor receives more than $\left(\frac{n}{p} + \frac{n}{s} - p\right)$ elements, for $n \geq p^3$.

Hence, the analysis of our regular sample sort algorithm is similar to that of our sample sort algorithm and is given (for floating point numbers) by

$$T(n,p) = O\left(\frac{n}{p}\log n + \tau + \frac{n}{p}\sigma\right) \tag{3}$$

for $n \geq p^3$ and $\left(p \leq s \leq \frac{n}{p^2}\right)$.



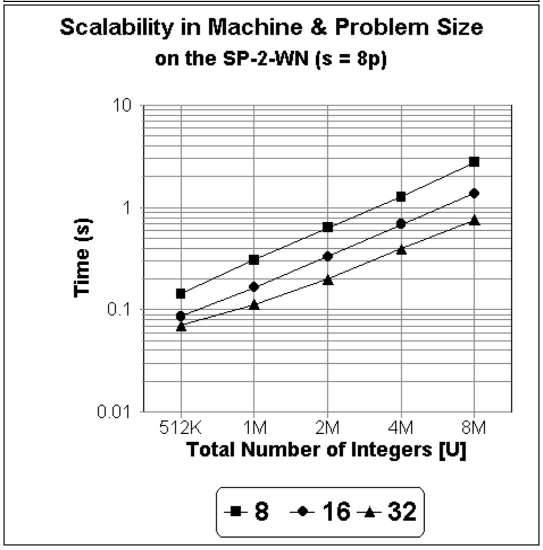


Figure 4: Scalability with respect to problem size of regular sorting *integers* from the **[U]** benchmark, for differing numbers of processors, on the Cray T3D and the IBM SP-2-WN.

Like our random sample sort algorithm, our regular sample sort algorithm is asymptotically optimal with very small coefficients. Once again, our algorithm was implemented and tested on the nine benchmarks. Table IV displays the performance of our regular sort as a function of input distribution for a variety of input sizes. It shows that the performance is essentially independent of the input distribution. Table V examines the scalability of our regular sort as a function of machine size. It shows that for a given input size n the execution time scales almost inversely with the number of processors p. Finally, Figure 4 examines 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. See [15] for additional performance data and comparisons with other published results.

Regular Sample Sorting of Integers, $s = 8p$				
Sorting Benchmark	**Benchmark**			
	1M	**4M**	**16M**	**64M**
[U]	0.114	0.355	1.17	4.42
[G]	0.114	0.341	1.16	4.38
[2-G]	0.0963	0.312	1.11	4.33
[4-G]	0.0970	0.320	1.12	4.28
[B]	0.117	0.352	1.17	4.35
[S]	0.0969	0.304	1.10	4.42
[Z]	0.0874	0.273	0.963	3.75
[WR]	0.129	0.365	1.35	5.09
[RD]	0.0928	0.285	0.997	3.81

Table IV: Total execution time (in seconds) for regular sorting a variety of benchmarks on a 64 node Cray T3D.

Regular Sample Sorting of 8M Integers, $s = 8p$, [WR]				
Machine	**Number of Processors**			
	8	**16**	**32**	**64**
Cray T3D	4.07	2.11	1.15	0.711
IBM SP2-WN	3.12	1.57	0.864	-
TMC CM-5	-	8.04	4.34	2.42

Table V: Total execution time (in seconds) for regular sorting 8M *integers* from the **[WR]** benchmark on a variety of machines and processors. A hyphen indicates that that particular platform was unavailable to us.

4.3 An Efficient Radix Sort

We now consider the problem of sorting n integer keys in the range $[0, M-1]$ that are distributed equally over a p-processor distributed memory machine. An efficient and well known stable algorithm is **radix sort** that decomposes each key into groups of r-bit blocks, for a suitably chosen r, and sorts the keys by sorting on each of the r-bit blocks beginning with the block containing the least significant bit positions. Here we only sketch the algorithm **Counting Sort** for sorting on individual blocks.

The **Counting Sort** algorithm sorts n integers in the range $[0, R-1]$ by using R counters to accumulate the number of keys equal to i in bucket B_i, for $0 \le i \le R-1$, followed by determining the rank of the each element. Once the rank of each element is known, we can use our h-relation personalized communication to move each element into the correct position; in this case $h = \frac{n}{p}$. Counting Sort is a **stable** sorting routine, that is, if two keys are identical, their relative order in the final sort remains the same as their initial order.

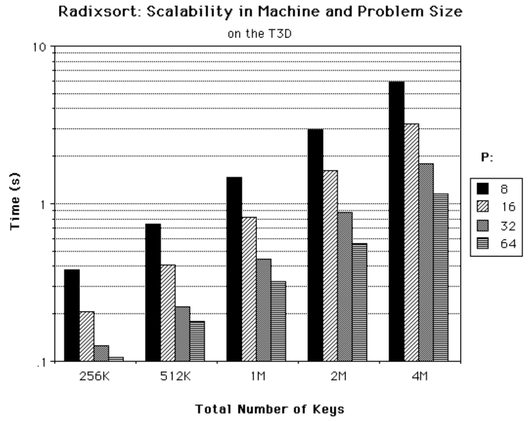
The pseudocode for our Counting Sort algorithm uses six major steps and is as follows.

- **Step (1):** For each processor i, count the frequency of its $\frac{n}{p}$ keys; that is, compute $I[i][k]$, the number of keys equal to k, for $(0 \le k \le R-1)$.
- **Step (2):** Apply the **transpose** primitive to the I array using the block size $\frac{R}{p}$. Hence, at the end of this step, each processor will hold $\frac{R}{p}$ consecutive rows of I.
- **Step (3):** Each processor locally computes the prefix-sums of its rows of the array I.
- **Step (4):** Apply the **(inverse) transpose** primitive to the R corresponding prefix-sums augmented by the total count for each bin. The block size of the **transpose** primitive is $2\frac{R}{p}$.
- **Step (5):** Each processor computes the ranks of local elements.
- **Step (6):** Perform a personalized communication of keys to rank location using our h-relation algorithm for $h = \frac{n}{p}$.

Input, $\frac{n}{p}$	SP-2 $p = 16$		CM-5 $p = 32$		CS-2 $p = 16$	
	[AIS]	[BHJ]	[AIS]	[BHJ]	[AIS]	[BHJ]
[R], 4K	0.474	0.107	1.63	0.163	0.664	0.083
[R], 64K	0.938	0.592	3.41	1.91	1.33	0.808
[R], 512K	4.13	4.03	19.2	15.1	7.75	7.33
[C], 4K	0.479	0.107	1.64	0.163	0.641	0.081
[C], 64K	0.958	0.584	3.31	1.89	1.23	0.790
[C], 512K	4.13	4.02	16.4	14.9	6.86	6.65
[N], 4K	0.475	0.109	1.63	0.163	0.623	0.085
[N], 64K	0.907	0.613	3.55	1.89	1.22	0.815
[N], 512K	4.22	4.12	18.2	15.0	6.34	7.29

Table VI: Total execution time for radix sort on 32-bit integers (in seconds), comparing the AIS and our implementations.

Table VI presents a comparison of our radix sort with another implementation of radix sort in SPLIT-C by Alexandrov et al. [1] This other implementation, which was tuned for the Meiko CS-2, is identified the table as AIS, while our algorithm is referred to as BHJ. The input [R] is random, [C] is cyclically sorted, and [N] is a random Gaussian approximation [4]. Additional performance results are given in Figure 5 and in [4].



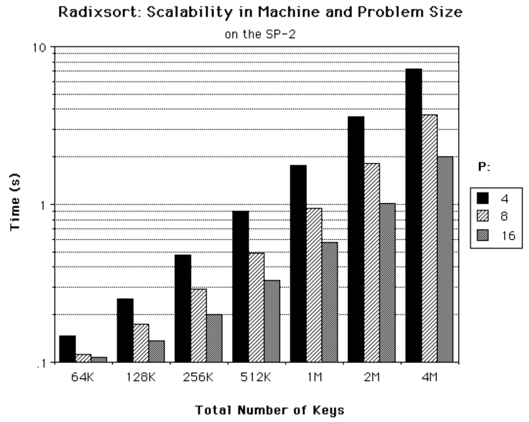


Figure 5: Scalability of radix sort with respect to machine and problem size, on the Cray T3D and the IBM SP-2-TN

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. Bader. Randomized and Deterministic Routing Algorithms for h-Relations. ENEE 648X Class Report, April 1, 1994.

[4] 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. To appear in *ACM Journal of Experimental Algorithmics.*

[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. To appear in *Journal of Parallel and Distributed Computing.*

[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] 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.

[8] 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.

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

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

[11] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, S. Luna, T. von Eicken, and K. Yelick. *Introduction to Split-C.* Computer Science Division - EECS, University of California, Berkeley, version 1.0 edition, March 6, 1994.

[12] 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.

[13] 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.

[14] D.R. Helman, D.A. Bader, and J. JáJá. A Parallel Sorting Algorithm With an Experimental Study. Technical Report CS-TR-3549 and UMIACS-TR-95-102, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, December 1995.

[15] D.R. Helman, D.A. Bader, and J. JáJá. A Parallel Regular Sorting Algorithm With an Experimental Study. Technical report, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, June 1996. In Preparation.

[16] 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.

[17] 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. To appear in *IEEE Transactions on Parallel and Distributed Systems.*

[18] M. Kaufmann, J.F. Sibeyn, and T. Suel. Derandomizing Algorithms for Routing and Sorting on Meshes. In *Proceedings of the 5th Symposium on Discrete Algorithms*, pages 669–679. ACM-SIAM, 1994.

[19] 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.

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

[21] S. Ranka, R.V. Shankar, and K.A. Alsabti. Many-to-many Personalized Communication with Bounded Traffic. In *The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 20–27, McLean, VA, February 1995.

[22] S. Rao, T. Suel, T. Tsantilas, and M. Goudreau. Efficient Communication Using Total-Exchange. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 544–550, Santa Barbara, CA, April 1995.

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

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

A Sorting Benchmarks

Our nine 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 $\left(2^{31}-1\right)$, 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 $\frac{\text{MAX}}{(2^{31}-1)}$.

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 $\left(\frac{\text{MAX}}{p}-1\right)$, the second $\frac{n}{p^2}$ elements at each processor to be random numbers between $\frac{\text{MAX}}{p}$ and $\left(2\frac{\text{MAX}}{p}-1\right)$, 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 $\left(\left(jg+\frac{p}{2}\right) \bmod p\right)\frac{\text{MAX}}{p}$ and $\left(\left(\left(jg+\frac{p}{2}+1\right) \bmod p\right)\frac{\text{MAX}}{p}-1\right)$, the second $\frac{n}{pg}$ elements at each processor to be random numbers between $\left(\left(jg+\frac{p}{2}+1\right) \bmod p\right)\frac{\text{MAX}}{p}$ and $\left(\left(\left(jg+\frac{p}{2}+2\right) \bmod p\right)\frac{\text{MAX}}{p}-1\right)$, 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 $(2i+1)\frac{\text{MAX}}{p}$ and $\left((2i+2)\frac{\text{MAX}}{p}-1\right)$, and so forth. Otherwise, we set all $\frac{n}{p}$ elements to be random numbers between $\left(i-\frac{p}{2}\right)\frac{\text{MAX}}{p}$ and $\left(\left(i-\frac{p}{2}+1\right)\frac{\text{MAX}}{p}-1\right)$, and so forth.

7. **Worst-Case Regular [WR]** - an input consisting of values between 0 and MAX designed to induce the worst possible load balance at the completion of our regular sorting. At the completion of sorting, the even-indexed processors will hold $(\frac{n}{p}+\frac{n}{s}-p)$ elements, whereas the odd-indexed processors will hold $(\frac{n}{p}-\frac{n}{s}+p)$ elements. See [15] for additional details.

8. **Randomized Duplicates [RD]** an input of duplicates in which each processor fills an array T with some constant number *range* of random values between 0 and $(range-1)$ (*range* is 32 for our work) whose sum is S. The first $\frac{T[0]}{S}\frac{n}{p}$ values of the input are then set to a random value between 0 and $(range-1)$, the next $\frac{T[1]}{S}\frac{n}{p}$ values of the input are then set to another random value between 0 and $(range-1)$, and so forth.

We selected these nine 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. The **Worst Case Regular** benchmark was included to empirically evaluate both the worst case running time expected for our regular sorting algorithm and the effect of the sampling rate on this performance. Finally, the the **Randomized Duplicates** benchmark was included to assess the performance of the algorithms in the presence of duplicate values

B 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.

We 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.

12 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) [9] on which the T3D port of Split-C has been based.

We also thank the Numerical Aerodynamic Simulation Systems Division of the NASA Ames Research Center for use of their 160-node IBM SP-2-WN.

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

Please see `http://www.umiacs.umd.edu/research/EXPAR` 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.