A New Deterministic Parallel Sorting Algorithm
With an Experimental Evaluation

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

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

August 15, 1996

Abstract

We introduce a new deterministic parallel sorting algorithm based on the regular sampling approach. The algorithm uses only two rounds of regular all-to-all personalized communication in a scheme that yields very good load balancing with virtually no overhead. Moreover, unlike previous variations, our algorithm efficiently handles the presence of duplicate values without the overhead of tagging each element with a unique identifier. This algorithm was implemented in SP-LT-C and run on a variety of platforms, including the Thinking Machines CM-5, the IBM SP-2-WN, 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 illustrate the efficiency and scalability of our algorithm across different platforms. In fact, the performance compares closely to that of our random sample sort algorithm, which seems to outperform all similar algorithms known to the authors on these platforms. Together, their performance is nearly invariant over the set of input distributions, unlike previous efficient algorithms. However, unlike our randomized sorting algorithm, the performance and memory requirements of our regular sorting algorithm can be deterministically guaranteed.

Keywords: Parallel Algorithms, Generalized Sorting, Integer Sorting, Sorting by Regular Sampling, Parallel Performance.

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

We present a novel variation on the approach of sorting by regular sampling which leads to a new deterministic sorting algorithm that achieves optimal computational speedup with very little communication [15]. Our algorithm exchanges the single step of irregular communication used by previous implementations for two steps of regular communication. In return, our algorithm reduces the problem of poor load balancing because it is able to sustain a high sampling rate at substantially less cost. In addition, our algorithm efficiently accommodates the presence of duplicates without the overhead of tagging each element. And our algorithm achieves predictable, regular communication requirements which are essentially invariant with respect to the input distribution. Utilizing regular communication has become more important with the advent of message passing standards, such as MPI [17], 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 efficiency and scalability of our algorithm across different platforms. In fact, the performance compares closely to that of our random sample sort algorithm, which seems to outperform all similar algorithms known to the authors on these platforms. Together, their performance is nearly indifferent to the set of input distributions, unlike previous efficient algorithms. However, unlike our randomized sorting algorithm, the performance and memory requirements of our regular sorting algorithm can be guaranteed with probability one.

The high-level language used in our studies is Split-C [10], 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 [17], the IBM POWERparallel [7], and the Cray MPP systems [9] and, for example, include the following: transpose, broadcast, 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 broadcast 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 \texttt{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.

\section{The Parallel Computation Model}

We use a simple model to analyze the performance of our parallel algorithms. Our model is based on the fact that current 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, where 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 \((\tau + \sigma m)\) time, where \( \tau \) is the latency of the network and \( \sigma \) is the time per word at which a processor can inject or receive data from the network. Note that the bandwidth per processor is inversely proportional to \( \sigma \). We assume that the bisection bandwidth is sufficiently high to support block permutation routing amongst the \( p \) processors at the rate of \( \frac{1}{\sigma} \). In particular, for any subset of \( q \) processors, a block permutation amongst the \( q \) processors takes \((\tau + \sigma m)\) time, where \( m \) is the size of the largest block.

Using this cost model, we can evaluate the communication time \( T_{\text{comm}}(n, p) \) of an algorithm as a function of the input size \( n \), the number of processors \( p \), and the parameters \( \tau \) and \( \sigma \). The coefficient of \( \tau \) gives the total number of times collective communication primitives are used, and the coefficient of \( \sigma \) 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. [11, 19, 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 \( T_{\text{comp}} \) as the maximum time it takes a processor to perform all the local computation steps. In general, the overall performance \( T_{\text{comp}} + T_{\text{comm}} \) involves a tradeoff between \( T_{\text{comp}} \) and \( T_{\text{comm}} \). In many cases, it is possible to minimize both \( T_{\text{comp}} \) and \( T_{\text{comm}} \) simultaneously, and sorting is such a case.
3 A New Sorting Algorithm by Regular Sampling

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 sorting by regular sampling is to find a set of \( p - 1 \) splitters to partition the \( n \) input elements into \( p \) groups indexed from 1 up to \( p \) 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 \((1 \leq i \leq p-1)\). Then the task of sorting each of the \( p \) groups can be turned over to the corresponding 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 evenly we choose the splitters. One way to choose the splitters is by regularly sampling the sorted input elements at each processor - hence the name Sorting by Regular Sampling.

A previous version of regular sample sort [18, 16], known as Parallel Sorting by Regular Sampling (PSRS), first sorts the \( \frac{n}{p} \) elements at each processor and then selects every \( \left(\frac{p}{n}\right)^{th} \) 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 of these subsequences 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 with as many as \( \left(2\frac{n}{p} - \frac{n}{p} + 1\right) \) elements at the completion of sorting. This could be reduced by choosing more samples, but this would also increase the overhead. And no matter how many samples are chosen, previous studies have shown that the load balance would still deteriorate linearly with the number of duplicates [16]. One could, of course, tag each item with a unique value, but this would also double the cost of both memory access and interprocessor communication. 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 [17].

In our algorithm, which is parameterized by a sampling ratio \( s \left( p \leq s \leq \frac{n}{p} \right) \), 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 set of samples used to identify the splitters. This bound holds regardless of the number of duplicate elements present in the input. Moreover, we are able to replace the irregular routing with exactly two calls to our transpose primitive.

The pseudocode for our algorithm is as follows:

- **Step (1):** Each processor \( P_i \ (1 \leq i \leq p) \) 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 $\left(\left\lfloor \frac{k-1}{p} \right\rfloor + 1\right)^{th}$ position of the $((k-1) \mod p) + 1)^{th}$ bin.

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

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

- **Step (4):** Processor $P_p$ merges the $p$ sorted subsequences of samples and then selects each $(ks)^{th}$ sample as Splitter[$k$], for $(1 \leq k \leq p-1)$. By default, the $p^{th}$ splitter is the largest value allowed by the data type used. Additionally, binary search is used to compute for the set of samples $S_k$ with indices $(ks-s+1)$ through $(ks)$ the number of samples $Est[k]$ which share the same value as Splitter[$k$]. 

- **Step (5):** Processor $P_p$ broadcasts the Splitter and Est arrays 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 $T_{\langle j,k \rangle}$. The set of $p$ subsequences $\{T_{\langle j,1 \rangle}, T_{\langle j,2 \rangle}, ..., T_{\langle j,p \rangle}\}$ associated with Splitter[$j$] all contain values which are greater than or equal to Splitter[$j-1$] and less than or equal to Splitter[$j$], and collectively include at most $\left(\text{Est}[j] \times \frac{n}{p^s}\right)$ elements with the same value as Splitter[$j$]. 

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

- **Step (8):** Each processor $P_i$ “unshuffles” all those subsequences sharing a common origin in **Step (2)**. 

- **Step (9):** Each processor $P_i$ merges the $p$ consolidated subsequences to produce the $i^{th}$ column of the sorted array. 

Before establishing the complexity of this algorithm, we need to establish the following theorem.

**Theorem 1:** The number of elements sent by processor $P_i$ to processor $P_j$ in **Step (7)** is at most $\left(\frac{n}{p^i} + \frac{n}{p^j} - p\right)$ for $i = p$ and $\left(\frac{n}{p^i} + \frac{n}{p^j}\right)$ for $i < p$. Consequently, at the completion of the algorithm, no processor holds more than $\left(\frac{n}{p^i} + \frac{n}{p^j} - p\right)$ elements, for $n \geq p^3$ and $\left(p \leq s \leq \frac{n}{p^s}\right)$. 

**Proof:** Let $S_j$ be the set of samples from the sorted array of samples in **Step (4)** with indices $(js-s+1)$ through $(js)$, inclusively. Let $S_{\langle j,i \rangle}$ be the subset of samples in $S_j$ originating from processor $P_i$, and let $c_{\langle j,i \rangle}$ be the cardinality of $S_{\langle j,i \rangle}$. Let $V_j = \text{Splitter}[j]$, let $c_{\langle j,i,1 \rangle}$ be the number of samples in $S_{\langle j,i \rangle}$ with
value less than \( V_j \), and let \( c_{(j,i,2)} = (c_{(j,i)} - c_{(j,i,1)}) \) be the number of samples in \( S_{(j,i)} \) with value \( V_j \). Let \( R_{(j,i)} = \{ \cup S_{(j,i)} : 1 \leq t < (j-1) \} \), let \( b_{(j,i)} = \sum_{t=1}^{j-1} c_{(t,i)} \) be the cardinality of \( R_{(j,i)} \), let \( b_{(j,i,1)} \) be the number of samples in \( R_{(j,i)} \) with value less than \( V_j \), and let \( b_{(j,i,2)} = (b_{(j,i)} - b_{(j,i,1)}) \) be the number of samples in \( R_{(j,i)} \) with value \( V_j \). Obviously, \( b_{(j,i,2)} \) will only be nonzero if \( \operatorname{Splitter}[j-1] = V_j \). Finally, for simplicity of discussion but without loss of generality, we assume that \( n \) is a constant multiple of \( p^2 \).

Clearly, each sample can be mapped in a one-to-one fashion to the sorted input generated during Step (1) (but before being distributed amongst the bins). For example, the first sample in \( S_{(j,i)} \) maps to the \( \left( \left( b_{(j,i)} + 1 \right) \frac{n}{p^2 s} \right)^{th} \) element in the sorted input at processor \( P_i \), the second sample in \( S_{(j,i)} \) maps to the \( \left( \left( b_{(j,i)} + 2 \right) \frac{n}{p^2 s} \right)^{th} \) element in the sorted input at processor \( P_i \), and so forth up to the \( c_{(j,i)}^{th} \) element which maps to the \( \left( \left( b_{(j,i)} + c_{(j,i)} \right) \frac{n}{p^2 s} \right)^{th} \) element in the sorted input at processor \( P_i \).

Hence, it follows that \( L_{(j,i)} \) elements in the sorted input of Step (1) at processor \( P_i \) will be less than \( V_j \), where \( \left( b_{(j,i,1)} + c_{(j,i,1)} \right) \frac{n}{p^2 s} \leq L_{(j,i)} \leq \left( b_{(j,i,1)} + c_{(j,i,1)} + 1 \right) \frac{n}{p^2 s} - 1 \). It is also true that at least \( M_{(j,i)} \) elements in the sorted input of Step (1) will be less than or equal to \( V_j \), where \( M_{(j,i)} = \left( b_{(j,i)} + c_{(j,i)} \right) \frac{n}{p^2 s} \).

The shuffling of Step (1) together with the transpose of Step (2) maps the \( t^{th} \) element at processor \( P_i \) into the \( \left( \left( \frac{t-1}{p} \right) + 1 \right)^{th} \) position of the \( i^{th} \) subarray at processor \( P_i \), a subarray which we will denote as \( \operatorname{SA}_{((i-1) \mod p^2 s) + 1,i} \). Now, \( L_{(j,r,i)} \) elements in \( \operatorname{SA}_{(r,i)} \) will be less than \( V_j \) and will unequivocally route to processors \( P_i \) through \( P_j \), where:

\[
\left( b_{(j,i,1)} + c_{(j,i,1)} \right) \frac{n}{p^2 s} \leq L_{(j,r,i)} \leq \left( b_{(j,i,1)} + c_{(j,i,1)} + 1 \right) \frac{n}{p^2 s} \quad \text{if } r < p
\]

\[
\left( b_{(j,i,1)} + c_{(j,i,1)} \right) \frac{n}{p^2 s} \leq L_{(j,r,i)} \leq \left( b_{(j,i,1)} + c_{(j,i,1)} + 1 \right) \frac{n}{p^2 s} - 1 \quad \text{if } r = p
\]

Furthermore, at least \( M_{(j,r,i)} \) elements in \( \operatorname{SA}_{(r,i)} \) will be less than or equal to \( V_j \), where \( M_{(j,r,i)} = \left( b_{(j,i)} + c_{(j,i)} \right) \frac{n}{p^2 s} \). This means that the \( p \) subarrays at processor \( P_r \) collectively have at least

\[
\sum_{i=1}^{p} M_{(j,r,i)} = \sum_{i=1}^{p} \left( b_{(j,i)} + c_{(j,i)} \right) \frac{n}{p^2 s} = \frac{n}{p^2}
\]

elements which are greater than or equal to \( V_j \). Furthermore,

\[
\sum_{i=1}^{p} \left( M_{(j,r,i)} - L_{(j,r,i)} \right) \leq \sum_{i=1}^{p} \left( \left( b_{(j,i)} + c_{(j,i)} \right) \frac{n}{p^2 s} - \left( b_{(j,i,1)} + c_{(j,i,1)} \right) \frac{n}{p^2 s} \right)
\]

\[
= \sum_{i=1}^{p} \left( b_{(j,i,2)} + c_{(j,i,2)} \right) \frac{n}{p^2 s}
\]

\[
= \left( \operatorname{Est}^*[j] + \operatorname{Est}[j] \right) \times \frac{n}{p^2 s}.
\]
where \((\text{Est}[j] \times \frac{n}{p^2})\) is the number of elements equal to \(V_j\) which the algorithm in \textbf{Step (6)} will seek to route to processor \(P_j\) and \((\text{Est}^*[j] \times \frac{n}{p^2})\) is the number of elements equal to \(V_j\) which the algorithm in \textbf{Step (6)} will seek to route to processors \(P_i\) through \(P_{(j-1)}\). From this it follows that the algorithm will always be able to route a minimum of \(\text{Min}_{(j,r)} = \sum_{t=1}^{p} M_{(j,r,t)} = j \frac{n}{p^2}\) elements to processors \(P_i\) through \(P_j\). On the other hand, the maximum number of elements that will be routed by this algorithm to these processors is:

\[
\text{Max}_{(j,r)} = \left(\sum_{t=1}^{p} I_{(j,r,t)}\right) + (\text{Est}^*[j] + \text{Est}[j]) \times \frac{n}{p^2} s
\]

\[
\leq \left\{ \begin{array}{ll}
\sum_{t=1}^{p} \left((b_{j,t,1} + c_{j,t,1}) + 1\right) \frac{n}{p^2} s + \left(b_{j,t,2} + c_{j,t,2}\right) \frac{n}{p^2} s &= (js + p) \frac{n}{p^2} s - (j - 1)s \frac{n}{p^2} s \\
\sum_{t=1}^{p} \left((b_{j,t,1} + c_{j,t,1}) + 1\right) \frac{n}{p^2} s - 1 + \left(b_{j,t,2} + c_{j,t,2}\right) \frac{n}{p^2} s &= (js + p) \frac{n}{p^2} s - p \\
\end{array} \right. \text{ if } r < p
\]

\[
\text{Max}_{(j,r)} - \text{Min}_{(j-1,r)} = \left\{ \begin{array}{ll}
(js + p) \frac{n}{p^2} s - (js + (j - 1)s) \frac{n}{p^2} s &= (js + p) \frac{n}{p^2} s - (j - 1)s \frac{n}{p^2} s \\
&= \left(\frac{n}{p^2} s + \frac{n}{ps}\right) \text{ if } r < p
\end{array} \right.
\]

Hence, the maximum number of elements send by processor \(P_r\) to processor \(P_j\) is:

\[
\text{Max}_{(j,r)} - \text{Min}_{(j-1,r)} = \left\{ \begin{array}{ll}
(js + p) \frac{n}{p^2} s - (js + (j - 1)s) \frac{n}{p^2} s &= (js + p) \frac{n}{p^2} s - (j - 1)s \frac{n}{p^2} s \\
&= \left(\frac{n}{p^2} s + \frac{n}{ps}\right) \text{ if } r < p
\end{array} \right.
\]

and Theorem 1 follows.

With the results of Theorem 1, the analysis of our algorithm for sorting by regular sampling is as follows. \textbf{Steps (3), (4), (6), (8), and (9)} require \(O(sp)\), \(O(sp \log p)\), \(O(p^2 \log p)\), \(O\left(\frac{n}{p} + \frac{n}{s} + p^2 - p\right)\), and \(O\left(\left(\frac{n}{p} + \frac{n}{s} - p\right) \log p\right)\) time, respectively. The cost of sequential sorting in \textbf{Step (1)} depends on the data type - sorting integers using radix sort requires \(O\left(\frac{n}{p}\right)\) time, whereas sorting floating point numbers using merge sort requires \(O\left(\frac{n}{p} \log \left(\frac{n}{p}\right)\right)\) time. \textbf{Steps (2), (5), and (7)} call the communication primitives \texttt{transpose}, \texttt{beast}, and \texttt{transpose}, respectively. The analysis of these primitives in [6] shows that these three steps require \(T_{\text{comm}}(n,p) \leq \left(\tau + \frac{n}{p}(p-1)\sigma\right)\), \(T_{\text{comm}}(n,p) \leq \left(\tau + 2(p-1)\sigma\right)\), and \(T_{\text{comm}}(n,p) \leq \left(\tau + \left(\frac{n}{p} + \frac{n}{sp}\right)(p-1)\sigma\right)\), respectively. Hence, with high probability, the overall complexity of our sorting algorithm is given (for floating point numbers) by

\[
T(n,p) = \text{Time}(n,p) + T_{\text{comm}}(n,p)
= O\left(\frac{n}{p} \log n + \tau + \frac{n}{p}\sigma\right)
\]

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

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

Our sample sort algorithm was implemented using Split-C [10] and run on a variety of machines and processors, including the Cray Research T3D, the IBM SP-2-WN, and the Thinking Machines CM-5. For every platform, we tested our code 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.

4.1 Sorting Benchmarks

Our nine sorting benchmarks are defined as follows, in which \( n \) and \( p \) are assumed for simplicity but without loss of generality to be powers of two and \( \text{MAX}_D \), the maximum value allowed for doubles, is approximately \( 1.8 \times 10^{308} \).

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 seeded by each processor \( P_i \) with the value \( (21 + 1001i) \). For the double data type, we “normalize” the integer benchmark values by first subtracting the value \( 2^{30} \) and then scaling the result by \( (2^{-30} \times \text{MAX}_D) \).

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 data type, we normalize the integer benchmark values 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 \( \frac{n}{p} \) elements at each processor to be random numbers between 0 and \( (\frac{2^{31}}{p} - 1) \), the second \( \frac{n}{p} \) elements at each processor to be random numbers between \( \frac{2^{31}}{p} \) and \( (\frac{2^{32}}{p} - 1) \), and so forth. For the double data type, we normalize the integer benchmark values in the manner described for [U].

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 from 1 up to \( \frac{p}{g} \), then for group \( j \) we set the first \( \frac{n}{pg} \) elements to be random numbers between \( (((j - 1)g + \frac{p}{2} - 1) \mod p) + 1 \) \( \frac{2^{31}}{p} \) and \( (((j - 1)g + \frac{p}{2}) \mod p) + 1 \) \( \frac{2^{31}}{p} - 1 \), the second \( \frac{n}{pg} \) elements at each processor to be random numbers between
(((j - 1)g + \frac{n}{2}) \mod p) + 1) \frac{2^n}{p} \text{ and } (((((j - 1)g + \frac{n}{2} + 1) \mod p) + 1) \frac{2^n}{p} - 1), \text{ and so forth.}

For the double data type, we normalize the integer benchmark values in the manner described for [U].

6. **Staggered** [S], created as follows: if the processor index \( i \) is less than or equal to \( \frac{p}{2} \), then we set all \( \frac{n}{p} \) elements at that processor to be random numbers between \( ((2i - 1) \frac{2^n}{p} + 1) \) and \( ((2i) \frac{2^n}{p} - 1) \). Otherwise, we set all \( \frac{n}{p} \) elements to be random numbers between \( ((2i - p - 2) \frac{2^n}{p} + 1) \) and \( ((2i - p - 1) \frac{2^n}{p} - 1) \). For the double data type, we normalize the integer benchmark values in the manner described for [U].

7. **Worst-Load Regular** [WR] - an input consisting of values between 0 and \((2^{31} - 1)\) designed to induce the worst possible load balance at the completion of our regular sorting. Specifically, at the completion of sorting, the odd-indexed processors will hold \( (\frac{n}{p} + \frac{n}{p} + p) \) elements, whereas the even-indexed processors will hold \( (\frac{n}{p} - \frac{n}{p} + p) \) elements. The benchmark is defined as follows. At processor \( P_1 \), for odd values of \( j \) between 1 and \((p - 2)\), the elements with indices \( ((j - 1) \frac{2^{31}}{p} + 1) \) through \( ((j \frac{2^{31}}{p} - 1) \) are set to random values between \( ((j - 1) \frac{2^{31}}{p} + 1) \) and \( ((j \frac{2^{31}}{p} - 1) \), the elements with indices \( ((j \frac{2^{31}}{p} + 1) \) through \( ((j + 1) \frac{2^{31}}{p} - 1) \) are set to random values between \( ((j \frac{2^{31}}{p} + 1) \) and \( ((j + 1) \frac{2^{31}}{p} - 1) \), and the element with index \( ((j + 1) \frac{2^{31}}{p} + 1) \) is set to \( ((j + 1) \frac{2^{31}}{p} - 1) \). At processor \( P_1 \), for \( j \) equal to \((p - 1)\), the elements with indices \( ((j - 1) \frac{2^{31}}{p} + 1) \) through \( ((j \frac{2^{31}}{p} - 1) \) are set to random values between \( ((j - 1) \frac{2^{31}}{p} + 1) \) and \( ((j \frac{2^{31}}{p} - 1) \), the elements with indices \( ((j \frac{2^{31}}{p} + 1) \) through \( ((j + 1) \frac{2^{31}}{p} - 1) \) are set to random values between \( ((j \frac{2^{31}}{p} + 1) \) and \( ((j + 1) \frac{2^{31}}{p} - 1) \). At processor \( P_1 \), \( (i > 1) \), for odd values of \( j \) between 1 and \( p \), the elements with indices \( ((j - 1) \frac{2^{31}}{p} + 1) \) through \( ((j \frac{2^{31}}{p} + 1) \) are set to random values between \( ((j - 1) \frac{2^{31}}{p} + 1) \) and \( ((j \frac{2^{31}}{p} - 1) \), the elements with index \( ((j \frac{2^{31}}{p} + 1) \) is set to \( ((j \frac{2^{31}}{p} + 1) \) of the \( P_1 \), the next \( ((j \frac{2^{31}}{p} + 1 + i) \) and \( ((j + 1) \frac{2^{31}}{p} - 1) \). For the double data type, we normalize the integer benchmark values in the manner described for [U].

8. **Deterministic Duplicates** [DD], an input of duplicates in which we set all \( \frac{n}{p} \) elements at each of the first \( \frac{p}{2} \) processors to be \( \log n \), all \( \frac{n}{p} \) elements at each of the next \( \frac{p}{2} \) processors to be \( \log \left( \frac{n}{2} \right) \), and so forth. At processor \( P_p \), we set the first \( \frac{n}{2p} \) elements to be \( \log \left( \frac{n}{p} \right) \), the next \( \frac{n}{4p} \) elements to be \( \log \left( \frac{n}{2p} \right) \), and so forth.

9. **Randomized Duplicates** [RD], an input of duplicates in which each processor fills an array \( T \) with some constant number \( range \) (\( range \) is 32 for our work) of random values between 0 and \(( range - 1) \) whose sum is \( S \). The first \( \frac{T[1]}{S} \times \frac{n}{p} \) values of the input are then set to a random value between 0 and \(( range - 1) \), the next \( \frac{T[2]}{S} \times \frac{n}{p} \) values of the input are then set to another random value between 0 and \(( range - 1) \), and so forth.
See [14] for a detailed justification of these benchmarks.

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) [7] and MPI. As one might expect, they were faster than the high level Split-C implementation.

<table>
<thead>
<tr>
<th>Optimal Number of Samples ( s ) for Sorting on T3D</th>
<th>Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>int./proc.</td>
<td>8</td>
</tr>
<tr>
<td>16K</td>
<td>128</td>
</tr>
<tr>
<td>32K</td>
<td>128</td>
</tr>
<tr>
<td>64K</td>
<td>256</td>
</tr>
<tr>
<td>128K</td>
<td>256</td>
</tr>
<tr>
<td>256K</td>
<td>512</td>
</tr>
<tr>
<td>512K</td>
<td>512</td>
</tr>
<tr>
<td>1M</td>
<td>1024</td>
</tr>
</tbody>
</table>

Table I: Optimal number of samples \( s \) for sorting the [WR] integer benchmark on the Cray T3D, for a variety of processors and input sizes.

<table>
<thead>
<tr>
<th>Optimal Number of Samples ( s ) for Sorting on SP2</th>
<th>Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>int./proc.</td>
<td>8</td>
</tr>
<tr>
<td>16K</td>
<td>256</td>
</tr>
<tr>
<td>32K</td>
<td>256</td>
</tr>
<tr>
<td>64K</td>
<td>512</td>
</tr>
<tr>
<td>128K</td>
<td>512</td>
</tr>
<tr>
<td>256K</td>
<td>512</td>
</tr>
<tr>
<td>512K</td>
<td>1024</td>
</tr>
<tr>
<td>1M</td>
<td>1024</td>
</tr>
</tbody>
</table>

Table II: Optimal number of samples \( s \) for sorting the [WR] integer benchmark on the IBM SP-2-WN, for a variety of processors and input sizes.

Tables I and II examine the preliminary question of the optimal number of samples \( s \) for sorting on
the Cray T3D and the IBM SP-2-WN. They show the value of \( s \) which achieved the best performance on the Worst-Load Regular [WR] benchmark, as a function of both the number of processors \( p \) and the number of keys per processor \( \frac{n}{p} \). The results suggest that a good rule for choosing \( s \) is to set it to \( 2^{\frac{1}{2} \log(n/p)} \approx \sqrt{\frac{n}{p}}, \) which is what we do for the remainder of this discussion. To compare this choice for \( s \) with the theoretical expectation, we recall that the complexity of Step (3) is \( O(s p \log p) \), whereas the complexity of Step (9) is \( O \left( \left( \frac{n}{p} + \frac{n}{p} - p \right) \log p \right) \). Hence, the first term is an increasing function of \( s \), whereas the second term is a decreasing function of \( s \). It is easy to verify that the expression for the sum of these two complexities is minimized for \( s = O \left( \sqrt{\frac{n}{p}} \right) \), and, hence, the theoretical expectation for the optimal value of \( s \) agrees with what we observe experimentally.

Table III: Total execution time (in seconds) required to sort a variety of integer benchmarks on a 64-node Cray T3D.

<table>
<thead>
<tr>
<th>Size</th>
<th>[U]</th>
<th>[G]</th>
<th>[2-G]</th>
<th>[4-G]</th>
<th>[B]</th>
<th>[S]</th>
<th>[Z]</th>
<th>[WR]</th>
<th>[DD]</th>
<th>[RD]</th>
</tr>
</thead>
<tbody>
<tr>
<td>256K</td>
<td>0.047</td>
<td>0.046</td>
<td>0.040</td>
<td>0.040</td>
<td>0.046</td>
<td>0.042</td>
<td>0.036</td>
<td>0.051</td>
<td>0.037</td>
<td>0.042</td>
</tr>
<tr>
<td>1M</td>
<td>0.104</td>
<td>0.102</td>
<td>0.094</td>
<td>0.092</td>
<td>0.103</td>
<td>0.094</td>
<td>0.080</td>
<td>0.113</td>
<td>0.081</td>
<td>0.089</td>
</tr>
<tr>
<td>4M</td>
<td>0.309</td>
<td>0.305</td>
<td>0.299</td>
<td>0.291</td>
<td>0.310</td>
<td>0.303</td>
<td>0.245</td>
<td>0.325</td>
<td>0.250</td>
<td>0.261</td>
</tr>
<tr>
<td>16M</td>
<td>1.09</td>
<td>1.08</td>
<td>1.09</td>
<td>1.06</td>
<td>1.10</td>
<td>1.11</td>
<td>0.903</td>
<td>1.13</td>
<td>0.904</td>
<td>0.930</td>
</tr>
<tr>
<td>64M</td>
<td>4.18</td>
<td>4.11</td>
<td>4.22</td>
<td>4.09</td>
<td>4.15</td>
<td>4.31</td>
<td>3.52</td>
<td>4.21</td>
<td>3.52</td>
<td>3.59</td>
</tr>
</tbody>
</table>

Table IV: Total execution time (in seconds) required to sort a variety of integer benchmarks on a 64-node IBM SP-2-WN.

<table>
<thead>
<tr>
<th>Size</th>
<th>[U]</th>
<th>[G]</th>
<th>[2-G]</th>
<th>[4-G]</th>
<th>[B]</th>
<th>[S]</th>
<th>[Z]</th>
<th>[WR]</th>
<th>[DD]</th>
<th>[RD]</th>
</tr>
</thead>
<tbody>
<tr>
<td>256K</td>
<td>0.055</td>
<td>0.055</td>
<td>0.050</td>
<td>0.048</td>
<td>0.051</td>
<td>0.049</td>
<td>0.046</td>
<td>0.056</td>
<td>0.047</td>
<td>0.050</td>
</tr>
<tr>
<td>1M</td>
<td>0.091</td>
<td>0.094</td>
<td>0.085</td>
<td>0.086</td>
<td>0.089</td>
<td>0.087</td>
<td>0.083</td>
<td>0.099</td>
<td>0.087</td>
<td>0.089</td>
</tr>
<tr>
<td>4M</td>
<td>0.237</td>
<td>0.236</td>
<td>0.229</td>
<td>0.223</td>
<td>0.224</td>
<td>0.228</td>
<td>0.222</td>
<td>0.253</td>
<td>0.231</td>
<td>0.239</td>
</tr>
<tr>
<td>16M</td>
<td>0.873</td>
<td>0.878</td>
<td>0.974</td>
<td>0.886</td>
<td>0.868</td>
<td>0.969</td>
<td>0.819</td>
<td>0.904</td>
<td>0.835</td>
<td>0.851</td>
</tr>
<tr>
<td>64M</td>
<td>3.45</td>
<td>3.46</td>
<td>3.83</td>
<td>3.86</td>
<td>3.38</td>
<td>3.79</td>
<td>3.09</td>
<td>3.45</td>
<td>3.11</td>
<td>3.12</td>
</tr>
</tbody>
</table>

Table V: Total execution time (in seconds) required to sort a variety of double benchmarks on a 64-node Cray T3D.

Tables III, IV, V, and VI 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 and a 64 node IBM.
Table VI: Total execution time (in seconds) required to sort a variety of double benchmarks on a 64-node IBM SP-2-WN.

SP-2; results obtained from other platforms 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 **Worst-Load Regular** benchmark [WR].

The results in Tables VII and VIII together with their graphs in Figure 1 examine the scalability of our sample sort as a function of machine size. Results are shown for the T3D, the SP-2-WN, and the CM-5. Bearing in mind that these graphs are log-log plots, they show that, for a given input size \( n \), the execution time scales inversely with the number of processors \( p \) for \( (p \leq 64) \). While this is certainly the expectation of our analytical model for doubles, it might at first appear to exceed our prediction of an \( O\left(\frac{n}{p} \log p\right) \) computational complexity for integers. However, the appearance of an inverse relationship is still quite reasonable when we note that, for values of \( p \) between 8 and 64, \( \log p \) varies by only a factor of two. Moreover, this \( O\left(\frac{n}{p} \log p\right) \) complexity is entirely due to the merging in **Step (9)**, and in practice, **Step (9)** never accounts for more than 30% of the observed execution time. Note that the complexity of **Step (9)** could be reduced to \( O\left(\frac{n}{p}\right) \) for integers using radix sort, but the resulting execution time would, in most cases, be slower.

### Table VII: Total execution time (in seconds) required to sort 8M integers on a variety of machines and processors using the [WR] benchmark. A hyphen indicates that particular platform was unavailable to us.

<table>
<thead>
<tr>
<th>Machine</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRAY T3D</td>
<td>3.23</td>
<td>1.73</td>
<td>0.976</td>
<td>0.594</td>
<td>0.496</td>
</tr>
<tr>
<td>IBM SP2-WN</td>
<td>2.73</td>
<td>1.38</td>
<td>0.761</td>
<td>0.472</td>
<td>0.410</td>
</tr>
<tr>
<td>TMC CM-5</td>
<td>-</td>
<td>7.83</td>
<td>3.99</td>
<td>2.29</td>
<td>2.55</td>
</tr>
</tbody>
</table>

However, the results in Tables VII and VIII together with their graphs in Figure 1 also show that for \( p \) greater than 64, the inverse relationship between the execution time and the number of processors begins to deteriorate. **Table IX** explains these results with a step by step breakdown of the execution times reported for the sorting of integers on the T3D. **Step (1)** clearly displays the \( O\left(\frac{n}{p}\right) \) complexity expected for radix sort, and it dominates the total execution time for small values of \( p \). The **transpose** operation in **Step (2)** displays the \( \left(\tau + \frac{n}{p}\sigma\right) \) complexity we originally suggested. The dependence
Table VIII: Total execution time (in seconds) required to sort 8M doubles on a variety of machines and processors using the [WR] benchmark. A hyphen indicates that particular platform was unavailable to us.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>CRA Y T3D</td>
<td>5.25</td>
</tr>
<tr>
<td>IBM SP2-W N</td>
<td>7.95</td>
</tr>
<tr>
<td>TMC CM-5</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 1: Scalability of sorting integers and doubles with respect to machine size.

of $\tau$ on $p$ simply becomes more pronounced as $p$ increases and $\frac{n}{p}$ decreases. **Step (3)** exhibits the $O(sp)$ complexity we anticipated, since for $2 \left[ \frac{1}{p} \log(n/p) \right]$, $s$ is halved every other time $p$ is doubled. **Steps (6) and (9)** display the expected $O(p^2 \log p)$ and $O \left( \left( \frac{n}{p} + \frac{n}{s} \right) \log p \right)$ ($\approx O \left( \left( \frac{n}{p} + \sqrt{n/p} \right) \log p \right)$ for $s \approx \sqrt{\frac{n}{p}}$) complexity, respectively. **Steps (7) and (8)** exhibit the most complicated behavior. The reason for this is that in **Step (7)**, each processor must exchange $p$ subsequences with every other processor and must include with each subsequence a record consisting of four integer values which will allow the unshuffling in **Step (8)** to be performed efficiently. Hence, the $O \left( \frac{n}{p^2} + \frac{n}{sp} + 4p \right)$ transpose block size in the case of 128 processors is nearly half that of the the case of 64 processors (1280 vs. 2816). This, together with the fact that $\tau$ increases as a function of $p$, explains why the time required for **Step (7)** actually increases for 128 processors. **Step (8)** would also be expected to
Step by Step Breakdown of Sorting 8M Integers

<table>
<thead>
<tr>
<th>Step</th>
<th>Number of Processors (Number of Samples)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 (1024)</td>
</tr>
<tr>
<td>1</td>
<td>2.3202920</td>
</tr>
<tr>
<td>2</td>
<td>0.132129</td>
</tr>
<tr>
<td>3</td>
<td>0.008468</td>
</tr>
<tr>
<td>4</td>
<td>0.000015</td>
</tr>
<tr>
<td>5</td>
<td>0.000052</td>
</tr>
<tr>
<td>6</td>
<td>0.000390</td>
</tr>
<tr>
<td>7</td>
<td>0.130839</td>
</tr>
<tr>
<td>8</td>
<td>0.148714</td>
</tr>
<tr>
<td>9</td>
<td>0.485934</td>
</tr>
<tr>
<td>Total</td>
<td>3.226760</td>
</tr>
</tbody>
</table>

Table IX: Time required (in seconds) for each step of sorting 8M integers on the Cray T3D using the [WR] benchmark.

The graphs in Figure 2 examine the scalability of our regular sample sort as a function of keys per processor \( \frac{n}{p} \), for differing numbers of processors. They show that for a fixed number of up to 64 processors there is an almost linear dependence between the execution time and \( \frac{n}{p} \). While this is certainly the expectation of our analytic model for integers, it might at first appear to exceed our prediction of a \( O\left(\frac{n}{p} \log n\right) \) 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. For \( p > 64 \), the relationship between the execution time and \( \frac{n}{p} \) is no longer linear. But based on our discussion of the data in Table IX, for large \( p \) and relatively small \( n \) we would expect a sizeable contribution from those steps which exhibit \( O\left(p^2 \log p\right) \), \( O\left(\frac{n}{p} + \sqrt{n}p\right) \), and \( O\left(\left(\frac{n}{p} + \sqrt{n}p\right) \log p\right) \) complexity, which would explain this loss of linearity.

Finally, the graphs in Figure 3 examine the relative costs of the nine steps in our regular sample sort algorithm. Results are shown for both a 64 node T3D and a 64 node SP-2-WN, using both the integer and the double versions of the [WR] benchmark. Notice that for \( n = 64M \) integers, the sequential sorting, unshuffling, and merging performed in Steps (1), (8), and (9) consume approximately 85% of the execution time on the T3D and approximately 75% of the execution time on the SP-2. By contrast, the two transpose operations in Steps (2) and (7) together consume...