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 SPLIT-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**, **bcast**, **gather**, and **scatter**. Brief descriptions of these are as follows. The **transpose** primitive is an all-to-all personalized communication in which each processor has to send a unique block of data to every processor, and all the blocks are of the same size. The **bcast** primitive is used to copy a block of data from a single source to all the other processors. The primitives **gather** and **scatter** are companion primitives. **Scatter** divides a single array residing on a processor into equal-sized blocks, each of which is distributed to a unique processor, and **gather** coalesces these blocks back into a single array at a particular processor. See [3, 6, 4, 5] for algorithmic details, performance analyses, and empirical results for these communication primitives.

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

2 The Parallel Computation Model

We use a simple model to analyze the performance of our parallel algorithms. 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 τ is the latency of the network and σ 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 σ. 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 $T_{comm}(n, p)$ 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. [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_{comp} as the maximum time it takes a processor to perform all the local computation steps. In general, the overall performance $T_{comp} + T_{comm}$ involves a tradeoff between T_{comp} and T_{comm}. In many cases, it is possible to minimize both T_{comp} and T_{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 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 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 $\left(\frac{n}{p^2}\right)^{th}$ 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 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 $\left(2\frac{n}{p} - \frac{n}{p^2} - p + 1\right)$ 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 $\left(p \leq s \leq \frac{n}{p^2}\right)$, 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 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 $(((k-1) \bmod p) + 1)^{th}$ bin.

- **Step (2):** Each processor P_i routes the contents of bin j to processor P_j, for $(1 \le i, j \le p)$, 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 selects each $\left(k\frac{n}{p^2 s}\right)^{th}$ element as a *sample*, for $(1 \le k \le s)$ and a given value of s $\left(p \le s \le \frac{n}{p^2}\right)$.
- **Step (4):** Processor P_p merges the p sorted subsequences of *samples* and then selects each $(ks)^{th}$ *sample* as Splitter[k], for $(1 \le k \le p-1)$. 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 $T_{(j,k)}$. The set of p subsequences $\{T_{(j,1)}, T_{(j,2)}, ..., T_{(j,p)}\}$ associated with Splitter[j] all contain values which are greater than or equal to Splitter[$j-1$] and less then or equal to Splitter[j], and collectively include at most $\left(\text{Est}[j] \times \frac{n}{p^2 s}\right)$ 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 $(1 \le i, j \le p)$. Since no two processors will exchange more than $\left(\frac{n}{p^2} + \frac{n}{sp}\right)$ elements, this is equivalent to performing a **transpose** operation with block size $\left(\frac{n}{p^2} + \frac{n}{sp}\right)$.
- **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 $\left(\frac{n}{p^2} + \frac{n}{sp} - p\right)$ for $i = p$ and $\left(\frac{n}{p^2} + \frac{n}{sp}\right)$ for $i < p$. Consequently, at the completion of the algorithm, no processor holds more than $\left(\frac{n}{p} + \frac{n}{s} - p\right)$ elements, for $n \ge p^3$ and $\left(p \le s \le \frac{n}{p^2}\right)$.

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 $S_{(j,i)}$ be the subset of samples in S_j originating from processor P_i, and let $c_{(j,i)}$ be the cardinality of $S_{(j,i)}$. Let $V_j = \text{Splitter}[j]$, let $c_{(j,i,1)}$ be the number of samples in $S_{(j,i)}$ 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 $R_{(j,i)} = \{\cup S_{(t,i)} : 1 \leq t < (j-1)\}$, let $b_{(j,i)} = \sum_{t=0}^{j-1} c_{(t,i)}$ 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 Splitter$[j-1] = V_j$. Finally, for simplicity of discussion but without loss of generality, we assume that n is a constant multiple of $p^2 s$.

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 $\left(\left(b_{(j,i)} + 1\right) \frac{n}{ps}\right)^{th}$ element in the sorted input at processor P_i, the second sample in $S_{(j,i)}$ maps to the $\left(\left(b_{(j,i)} + 2\right) \frac{n}{ps}\right)^{th}$ element in the sorted input at processor P_i, and so forth up to the $c^{th}_{(j,i)}$ element which maps to the $\left(\left(b_{(j,i)} + c_{(j,i)}\right) \frac{n}{ps}\right)^{th}$ 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 $\left(\left(b_{(j,i,1)} + c_{(j,i,1)}\right) \frac{n}{ps} \leq L_{(j,i)} \leq \left(\left(b_{(j,i,1)} + c_{(j,i,1)} + 1\right) \frac{n}{ps} - 1\right)\right)$. 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 $M_{(j,i)} = \left(b_{(j,i)} + c_{(j,i)}\right) \frac{n}{ps}$.

The shuffling of **Step (1)** together with the **transpose** of **Step (2)** maps the t^{th} element at processor P_i into the $\left(\left\lfloor \frac{t-1}{p} \right\rfloor + 1\right)^{th}$ position of the i^{th} subarray at processor $P_{(((t-1) \bmod p)+1)}$, a subarray which we will denote as $\mathrm{SA}_{((((t-1) \bmod p)+1),i)}$. Now, $L_{(j,r,i)}$ elements in $\mathrm{SA}_{(r,i)}$ will be less than V_j and will unequivocally route to processors P_1 through P_j, where:

$$\begin{array}{rl}
\left(b_{(j,i,1)} + c_{(j,i,1)}\right) \dfrac{n}{p^2 s} \leq L_{(j,r,i)} \leq \left(b_{(j,i,1)} + c_{(j,i,1)} + 1\right) \dfrac{n}{p^2 s} & \text{if } r < p \\
\left(b_{(j,i,1)} + c_{(j,i,1)}\right) \dfrac{n}{p^2 s} \leq L_{(j,r,i)} \leq \left(\left(b_{(j,i,1)} + c_{(j,i,1)} + 1\right) \dfrac{n}{p^2 s} - 1\right) & \text{if } r = p
\end{array}$$

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

$$\begin{aligned}
\sum_{t=1}^{p} M_{(j,r,t)} &= \sum_{t=1}^{p} \left(b_{(j,t)} + c_{(j,t)}\right) \frac{n}{p^2 s} \\
&= j \frac{n}{p^2}
\end{aligned}$$

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

$$\begin{aligned}
\sum_{t=1}^{p} \left(M_{(j,r,t)} - L_{(j,r,t)}\right) &\leq \sum_{t=1}^{p} \left(\left(b_{(j,t)} + c_{(j,t)}\right) \frac{n}{p^2 s} - \left(b_{(j,t,1)} + c_{(j,t,1)}\right) \frac{n}{p^2 s}\right) \\
&= \sum_{t=1}^{p} \left(b_{(j,t,2)} + c_{(j,t,2)}\right) \frac{n}{p^2 s} \\
&= (\mathrm{Est}^*[j] + \mathrm{Est}[j]) \times \frac{n}{p^2 s},
\end{aligned}$$

where $\left(\text{Est}[j] \times \frac{n}{p^2 s}\right)$ is the number of elements equal to V_j which the algorithm in **Step (6)** will seek to route to processor P_j and $\left(\text{Est}^*[j] \times \frac{n}{p^2 s}\right)$ is the number of elements equal to V_j which the algorithm in **Step (6)** will seek to route to processors P_1 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_1 through P_j. On the other hand, the maximum number of elements that will be routed by this algorithm to these processors is:

$$\begin{aligned}
\text{Max}_{(j,r)} &= \left(\sum_{t=1}^{p} L_{(j,r,t)}\right) + (\text{Est}^*[j] + \text{Est}[j]) \times \frac{n}{p^2 s} \\
&\leq \begin{cases}
\begin{array}{r} \sum_{t=1}^{p}\left(\left(b_{(j,t,1)} + c_{(j,t,1)} + 1\right)\frac{n}{p^2 s} + \left(b_{(j,t,2)} + c_{(j,t,2)}\right)\frac{n}{p^2 s}\right) = \\ (js+p)\frac{n}{p^2 s} \end{array} & \text{if } r < p \\
\begin{array}{r} \sum_{t=1}^{p}\left(\left(\left(b_{(j,t,1)} + c_{(j,t,1)} + 1\right)\frac{n}{p^2 s} - 1\right) + \left(b_{(j,t,2)} + c_{(j,t,2)}\right)\frac{n}{p^2 s}\right) = \\ \left((js+p)\frac{n}{p^2 s} - p\right) \end{array} & \text{if } r = p
\end{cases}
\end{aligned}$$

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

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

and Theorem 1 follows.

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

$$\begin{aligned}
T(n,p) &= T_{comp}(n,p) + T_{comm}(n,p) \\
&= O\left(\frac{n}{p}\log n + \tau + \frac{n}{p}\sigma\right) \qquad (1)
\end{aligned}$$

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

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 $\mathrm{MAX}_{\mathrm{D}}$, the maximum value allowed for *doubles*, is approximately 1.8×10^{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 $(2^{-30} \times \mathrm{MAX}_{\mathrm{D}})$.
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 $\frac{n}{p^2}$ elements at each processor to be random numbers between 0 and $(\frac{2^{31}}{p} - 1)$, the second $\frac{n}{p^2}$ 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 $((((j-1)g + \frac{p}{2} - 1) \bmod p) + 1)\,\frac{2^{31}}{p}$ and $\left(((((j-1)g + \frac{p}{2}) \bmod p) + 1)\,\frac{2^{31}}{p} - 1\right)$, the second $\frac{n}{pg}$ elements at each processor to be random numbers between

$((((j-1)g+\frac{p}{2}) \bmod p)+1)\,\frac{2^{31}}{p}$ and $\left(((((j-1)g+\frac{p}{2}+1) \bmod p)+1)\,\frac{2^{31}}{p}-1\right)$, 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 $\left((2i-1)\,\frac{2^{31}}{p}\right)$ and $\left((2i)\,\frac{2^{31}}{p}-1\right)$. Otherwise, we set all $\frac{n}{p}$ elements to be random numbers between $\left((2i-p-2)\,\frac{2^{31}}{p}\right)$ and $\left((2i-p-1)\,\frac{2^{31}}{p}-1\right)$. 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 $(\frac{n}{p}+\frac{n}{s}-p)$ elements, whereas the even-indexed processors will hold $(\frac{n}{p}-\frac{n}{s}+p)$ 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 $\left((j-1)\frac{n}{p^2}+1\right)$ through $\left(j\frac{n}{p^2}-1\right)$ are set to random values between $\left((j-1)\frac{2^{31}}{p}+1\right)$ and $\left(j\frac{2^{31}}{p}-1\right)$, the elements with indices $\left(j\frac{n}{p^2}\right)$ through $\left(j\frac{n}{p^2}+\frac{n}{sp}-1\right)$ are set to $\left(j\frac{2^{31}}{p}\right)$, the elements with indices $\left(j\frac{n}{p^2}+\frac{n}{sp}\right)$ through $\left((j+1)\frac{n}{p^2}-1\right)$ are set to random values between $\left(j\frac{2^{31}}{p}+1\right)$ and $\left((j+1)\frac{2^{31}}{p}-1\right)$, and the element with index $\left((j+1)\frac{2^{31}}{p}\right)$ is set to $\left((j+1)\frac{2^{31}}{p}\right)$. At processor P_1, for j equal to $(p-1)$, the elements with indices $\left((j-1)\frac{n}{p^2}+1\right)$ through $\left(j\frac{n}{p^2}-1\right)$ are set to random values between $\left((j-1)\frac{2^{31}}{p}+1\right)$ and $\left(j\frac{2^{31}}{p}-1\right)$, the elements with indices $\left(j\frac{n}{p^2}\right)$ through $\left(j\frac{n}{p^2}+\frac{n}{sp}-1\right)$ are set to $\left(j\frac{2^{31}}{p}\right)$, and the elements with indices $\left(j\frac{n}{p^2}+\frac{n}{sp}\right)$ through $\left((j+1)\frac{n}{p^2}\right)$ are set to random values between $\left(j\frac{2^{31}}{p}+1\right)$ and $\left((j+1)\frac{2^{31}}{p}-1\right)$. At processor P_i $(i>1)$, for odd values of j between 1 and p, the elements with indices $\left((j-1)\frac{n}{p^2}+1\right)$ through $\left(j\frac{n}{p^2}+\frac{n}{sp}-1\right)$ are set to random values between $\left((j-1)\frac{2^{31}}{p}+1\right)$ and $\left(j\frac{2^{31}}{p}-1\right)$, the elements with index $\left(j\frac{n}{p^2}+\frac{n}{sp}\right)$ is set to $\left(j\frac{2^{31}}{p}+i\right)$, and the elements with indices $\left(j\frac{n}{p^2}+\frac{n}{sp}+1\right)$ through $\left((j+1)\frac{n}{p^2}\right)$ are set to random values between $\left(j\frac{2^{31}}{p}+1+i\right)$ and $\left((j+1)\frac{2^{31}}{p}-1\right)$. 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 $\frac{p}{4}$ 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.

Optimal Number of Samples s for Sorting on T3D					
	Number of Processors				
int./proc.	**8**	**16**	**32**	**64**	**128**
16K	128	128	128	128	128
32K	128	128	128	128	128
64K	256	256	256	256	128
128K	256	256	256	256	256
256K	512	512	512	256	512
512K	512	512	512	512	512
1M	1024	512	512	512	1024

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.

Optimal Number of Samples s for Sorting on SP2					
	Number of Processors				
int./proc.	**8**	**16**	**32**	**64**	**128**
16K	256	128	128	128	128
32K	256	256	256	256	256
64K	512	256	256	256	512
128K	512	512	512	512	512
256K	512	512	512	256	512
512K	1024	1024	1024	1024	1024
1M	1024	1024	1024	1024	1024

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 $2^{\lfloor \frac{1}{2}\log(n/p) \rfloor} \approx \sqrt{\frac{n}{p}}$, 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 $O(sp\log p)$, whereas the complexity of **Step (9)** is $O\left(\left(\frac{n}{p}+\frac{n}{s}-p\right)\log p\right)$. 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.

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

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

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

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

Size	[U]	[G]	**[2-G]**	**[4-G]**	[B]	[S]	[Z]	**[WR]**	**[DD]**	**[RD]**
256K	0.056	0.056	0.046	0.046	0.055	0.045	0.044	0.060	0.043	0.050
1M	0.126	0.126	0.113	0.113	0.131	0.111	0.107	0.136	0.018	0.115
4M	0.411	0.411	0.387	0.394	0.416	0.389	0.376	0.435	0.383	0.384
16M	1.60	1.59	1.55	1.55	1.58	1.55	1.49	1.60	1.50	1.49
64M	6.53	6.57	6.44	6.45	6.55	6.49	6.26	6.61	6.26	6.14

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

Size	[U]	[G]	[2-G]	[4-G]	[B]	[S]	[Z]	[WR]	[DD]	[RD]
256K	0.090	0.087	0.082	0.080	0.084	0.080	0.077	0.093	0.081	0.084
1M	0.181	0.184	0.176	0.186	0.176	0.176	0.168	0.198	0.187	0.188
4M	0.598	0.590	0.580	0.576	0.578	0.600	0.570	0.614	0.584	0.589
16M	2.26	2.25	2.35	2.35	2.26	2.40	2.25	2.34	2.29	2.33
64M	9.61	9.61	10.0	10.0	9.57	10.00	9.57	9.74	9.49	9.55

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.

Regular Sorting of 8M Integers [WR]					
	Number of Processors				
Machine	**8**	**16**	**32**	**64**	**128**
CRAY T3D	3.23	1.73	0.976	0.594	0.496
IBM SP2-WN	2.73	1.38	0.761	0.472	0.410
TMC CM-5	-	7.83	3.99	2.29	2.55

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.

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

Regular Sorting of 8M Doubles [WR]					
	Number of Processors				
Machine	**8**	**16**	**32**	**64**	**128**
CRAY T3D	5.25	2.65	1.41	0.827	0.619
IBM SP2-WN	7.95	4.05	2.09	1.18	0.870
TMC CM-5	-	-	6.89	4.39	4.24

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.

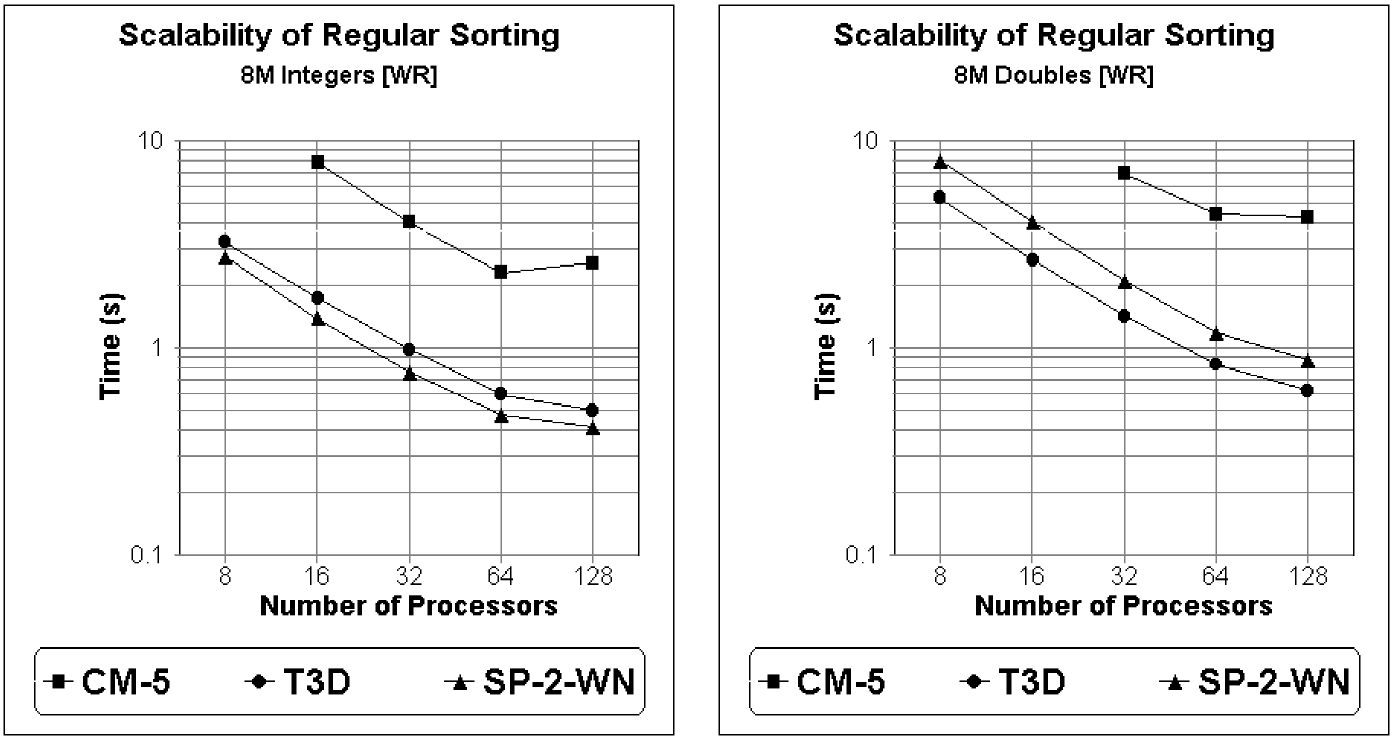


Figure 1: Scalability of sorting *integers* and *doubles* with respect to machine size.

of τ 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 $2^{\lfloor \frac{1}{2}\log(n/p)\rfloor}$, s is halved every other time p is doubled. **Steps (6)** and **(9)** display the expected $O\left(p^2\log p\right)$ and $O\left(\left(\frac{n}{p}+\frac{n}{s}\right)\log p\right)$ $\left(\approx O\left(\left(\frac{n}{p}+\sqrt{np}\right)\log p\right) \text{ for } s\approx\sqrt{\frac{n}{p}}\right)$ 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 τ 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

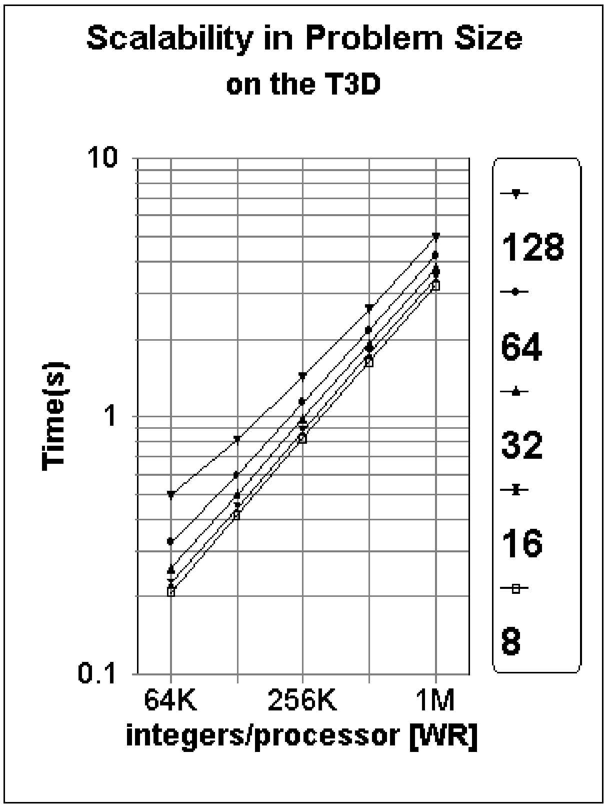
Step by Step Breakdown of Sorting 8M Integers					
	Number of Processors (Number of Samples)				
Step	**8 (1024)**	**16 (512)**	**32 (512)**	**64 (256)**	**128 (256)**
1	2.320220	1.172284	0.591670	0.299310	0.151576
2	0.132129	0.069106	0.045686	0.029076	0.019693
3	0.008468	0.010606	0.026364	0.026372	0.053686
4	0.000015	0.000019	0.000028	0.000047	0.000085
5	0.000052	0.000078	0.000082	0.000128	0.000226
6	0.000390	0.001303	0.004339	0.012499	0.028225
7	0.130839	0.070650	0.050185	0.039518	0.076284
8	0.148714	0.077050	0.042443	0.034429	0.059114
9	0.485934	0.332238	0.215449	0.152325	0.107410
Total	3.226760	1.733333	0.976246	0.593705	0.496300

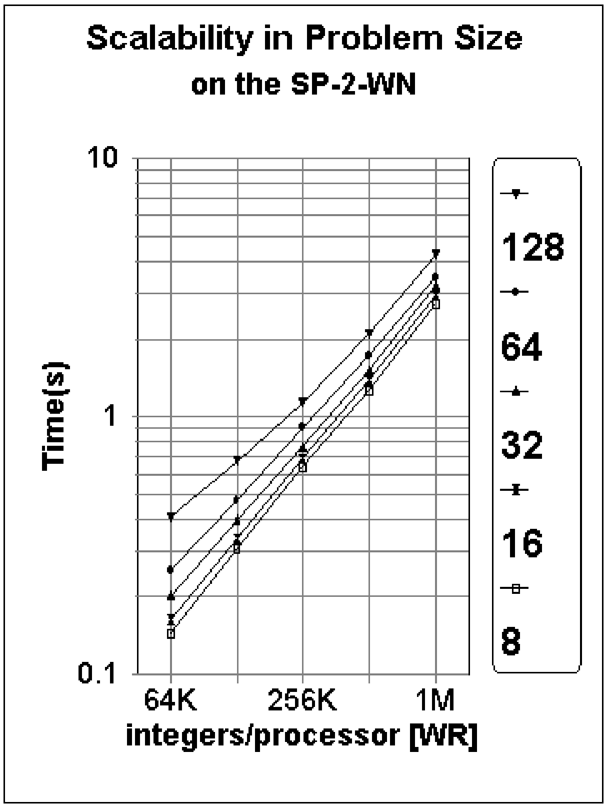
Table IX: Time required (in seconds) for each step of sorting 8M *integers* on the Cray T3D using the [**WR**] benchmark.

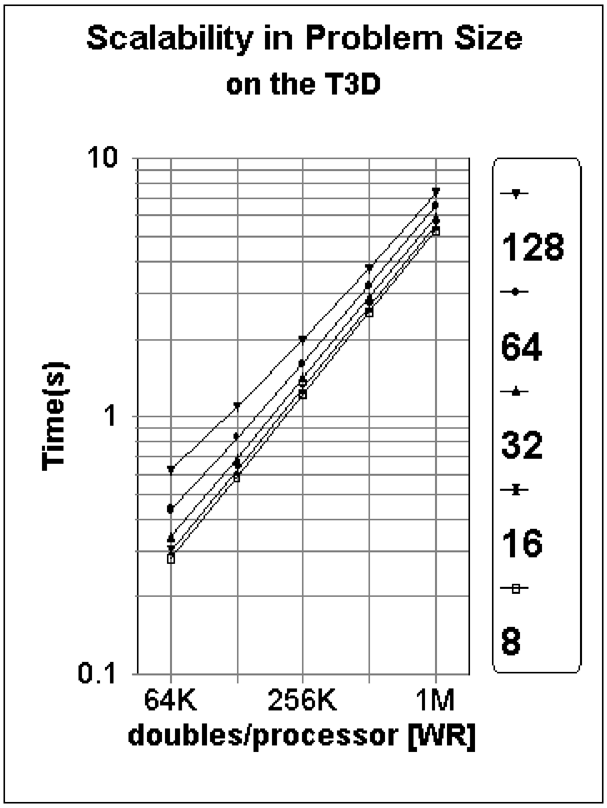
exhibit $O\left(\frac{n}{p}+\frac{n}{s}\right)$ $\left(\approx O\left(\frac{n}{p}+\sqrt{np}\right)\text{ for } s \approx \sqrt{\frac{n}{p}}\right)$ complexity. But the scheme chosen for unshuffling also involves an $O(p)$ amount of overhead for each group of p subsequences to assess their relationship so that they can be efficiently unshuffled. For sufficiently large values of p, this overhead begins to dominate the complexity. While the data of **Table IX** was collected for sorting *integers* on the T3D, the data from the SP-2-WN and the T3D support the same analysis for sorting both *integers* and *doubles*.

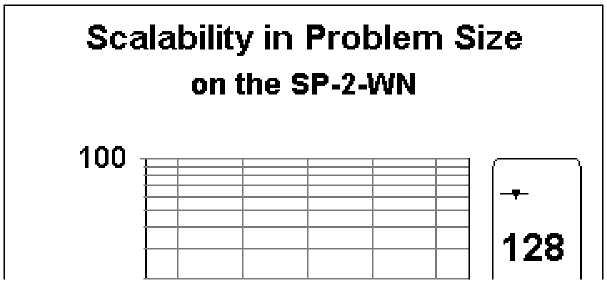
The graphs in **Figure 2** examine the scalability of our regular sample sort as a function of keys per processor $\left(\frac{n}{p}\right)$, for differing numbers of processors. They show that for a fixed number of up to 64 processors there is an almost linear dependence between the execution time and $\frac{n}{p}$. 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 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 $O\left(p^2 \log p\right)$, $O\left(\frac{n}{p}+\sqrt{np}\right)$, and $O\left(\left(\frac{n}{p}+\sqrt{np}\right)\log p\right)$ 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


Scalability in Problem Size
on the T3D
10
1
0.1
Time(s)
64K
256K
1M
integers/processor [WR]
128
64
32
16
8


Scalability in Problem Size
on the SP-2-WN
10
1
0.1
Time(s)
64K
256K
1M
integers/processor [WR]
128
64
32
16
8


Scalability in Problem Size
on the T3D
10
1
0.1
Time(s)
64K
256K
1M
doubles/processor [WR]
128
64
32
16
8


Scalability in Problem Size
on the SP-2-WN
100
128