11-1-1998

A Randomized Parallel Sorting Algorithm With an Experimental Study

D.R. Helman

D.A. Bader

J. JáJá

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SCHOOL OF ENGINEERING
UNIVERSITY OF NEW MEXICO

A New Deterministic Parallel Sorting Algorithm With an Experimental Evaluation

David R. Helman
Institute for Advanced Computer Studies
& Department of Electrical Engineering
The University of Maryland
College Park, MD 20742
e-mail: helman@umiacs.umd.edu

Joseph JáJá [1]
Institute for Advanced Computer Studies
& Department of Electrical Engineering
The University of Maryland
College Park, MD 20742
e-mail: joseph@umiacs.umd.edu

David A. Bader [2]
Department of Electrical and Computer Engineering
The University of New Mexico
Albuquerque, NM 87131
e-mail: dbader@eece.unm.edu

UNM Technical Report: EECE-TR-98-008

Report Date: November 1998

[1]This work was supported by NSF Grants No. CCR-9627210 and No. BIR-9318183.
[2]Supported by NSF CISE Postdoctoral Research Associate in Experimental Computer Science No. 96-25668 and NASA Graduate Student Researcher Fellowship No. NGT-50951.

Abstract

We introduce a new deterministic parallel sorting algorithm for distributed memory machines 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.

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. Our algorithm exchanges the single step of irregular communication used by previous implementations for two steps of regular communication. In return, our algorithm mitigates 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 [16], 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 deterministically.

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 [4], are similar to those of the MPI [16], the IBM POWERparallel [6], 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, 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. This network 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} 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. [1, 11, 18]) 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 samples is by random selection. Examples of this approach include the sample sort algorithm of Blelloch et al. [7] and our own random sample sort algorithm [14]. Alternatively, the splitters may be chosen as we have for this paper by regularly sampling the sorted input elements at each processor - hence the name Sorting by Regular Sampling.

A previous version of regular sample sort [15, 17], 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 [15, 17]. 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 [15]. 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 [16].

In our algorithm, which is parameterized by the number of samples 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. Note that since our algorithm involves three phases of computation punctuated by two rounds of balanced data rearrangement, we distinguish the work of the processors in the three phases by the indices i, j, and k (for initial, intermediate, and final phases of computation). Further, the movement of data between these three phases is distinguished by the same set of indices, so that an arbitrary set of data A indexed with (i, j, k) can be understood as originating at processor P_i in the first round of computation, moving to processor P_j for the intermediate round of computation, and finishing at processor P_k for the final round of computation. While this distinctions might at first appear burdensome, a proper appreciation of the data flow is essential to establishing the correctness of the algorithm.

- Step (1): Each processor P_i $(0 \leq i \leq p-1)$ sorts 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. Each processor P_i then "deals out" its sorted input values amongst a set of p bins so that the element with index x is placed into position $\left\lfloor \frac{x}{p} \right\rfloor$ of bin $(x \bmod p)$.
- Step (2): Each processor P_i routes $I_{(i,j)}$, the contents of bin j, to processor P_j, for $(0 \leq j \leq p-1)$. This 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 $\left((x+1)\frac{n}{p^2 s}\right)^{th}$ element as a *sample*, for $(0 \leq x \leq s-1)$ and a given value of s $\left(p \leq s \leq \frac{n}{p^2}\right)$.
- Step (4): Processor $P_{(p-1)}$ merges the p sorted subsequences of samples and then selects each $((k+1)s)^{th}$ sample as Splitter$[k]$, for $(0 \leq k \leq p-2)$. 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 with indices (ks) through $((k+1)s-1)$, the number Est$[k]$ of samples which share the same value as Splitter$[k]$.
- Step (5): Processor $P_{(p-1)}$ `broadcasts` the two arrays Splitter[ ] and Est[ ] to the other $p-1$ processors.
- Step (6): Each processor P_j uses binary search to define for each sorted sequence $I_{(i,j)}$ and each Splitter$[k]$ a subsequence $T_{(i,j,k)}$, for $(0 \leq i,k \leq p-1)$. The p subsequences $\{T_{(0,j,k)}, T_{(1,j,k)}, \ldots, T_{(p-1,j,k)}\}$ associated with Splitter$[k]$ all contain values which are greater than or equal to Splitter$[k-1]$ and less than or equal to Splitter$[k]$, and they collectively include *at most* $\left(\text{Est}[k] \times \frac{n}{p^2 s}\right)$ elements with the same value as Splitter$[k]$. The term *at most* is used because there may not actually be this number of elements with the same value as Splitter$[k]$.
- Step (7): Each processor P_j routes the set of p subsequences $\{T_{(0,j,k)}, T_{(1,j,k)}, \ldots, T_{(p-1,j,k)}\}$ associated with Splitter$[k]$ to processor P_j, for $(0 \leq k \leq p-1)$. 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_k merges the set of p^2 subsequences received in Step (7) to produce the k^{th} column of the sorted array.

In order to establish the complexity, we need to establish four lemmas, all of which make use of the following set of definitions. Let S_k be the set of samples in the sorted array of samples in Step (4) with indices 0 though $((k+1)s-1)$, inclusively. The last element in S_k is the element chosen as Splitter[k] and is said to have value V_k. Let $L_{(i,k)}$ be the number of samples strictly less than V_k in S_k which originate from the input at processor P_i. Similarly, let $E_{(i,k)}$ be the number of samples equal to V_k in S_k which originate from the input at processor P_i. Note that by definition any particular value of $L_{(i,k)}$ or $E_{(i,k)}$ may be equal to zero, and that $\sum_{u=0}^{p-1}\left(L_{(u,k)} + E_{(u,k)}\right) =$

$(k+1)s$. Note also that $\sum_{u=0}^{p-1} E_{(u,k)} = \sum_{w=t}^{k} \text{Est}[w]$, where t is the smallest index less then or equal to k such that Splitter$[t] = V_k$. Finally, we require throughout this discussion that $n \geq p^3$ and that $\left(p \leq s \leq \frac{n}{p^2}\right)$.

Lemma (1): The following three assertions can be made about the composition of the sorted subsequence $I_{(i,j)}$ received by processor P_j from processor P_i at the end of Step (2):

- (A): At least the first $\left(L_{(i,k)} + E_{(i,k)}\right) \frac{n}{p^2 s}$ elements in $I_{(i,j)}$ are less than or equal to V_k .
- (B): At least the first $\left(L_{(i,k)}\right) \frac{n}{p^2 s}$ elements in $I_{(i,j)}$ are strictly less than V_k.
- (C): The maximum number of elements in $I_{(i,j)}$ which are strictly less than V_k is $\left(\left(L_{(i,k)} + 1\right) \frac{n}{p^2 s} - 1\right)$ if $(j = p-1)$ and $\left(L_{(i,k)} + 1\right) \frac{n}{p^2 s}$ otherwise.

Proof: The effect of Steps (1) and (2) is to "deal out" the sorted input at processor P_i amongst the set of sequences $\left\{I_{(i,0)}, I_{(i,1)}, \ldots, I_{(i,p-1)}\right\}$. More precisely, the element with index x in the sorted input at processor P_i is mapped to position $\left\lfloor \frac{x}{p} \right\rfloor$ of sequence $I_{(i,(x \bmod p))}$. Conversely, the element with index y in sequence $I_{(i,j)}$ originates as the element with index $(yp + j)$ in the sorted input at processor P_i. Hence, the z_{th} element to be chosen as a sample from the elements originating at processor P_i, which is actually the element with index $\left(z\frac{n}{p^2 s} - 1\right)$ in $I_{(i,p-1)}$, originated as the element with index $\left(z\frac{n}{ps} - 1\right)$ in the sorted input at processor P_i.

By definition, there are at least $\left(L_{(i,k)} + E_{(i,k)}\right)$ samples originating from processor P_i whose values are less than or equal to V_k. We say *at least* since it is possible that there are samples originating from processor P_i whose value is equal to V_k but who fall after Splitter$[k]$ in the sorted array of samples. Hence, it follows that at least the first $\left(L_{(i,k)} + E_{(i,k)}\right) \frac{n}{ps}$ elements in the sorted input at processor P_i in Step (1) had to be less than or equal to V_k. Since these elements are mapped as described above to the set of sequences $\left\{I_{(i,0)}, I_{(i,1)}, \ldots, I_{(i,p-1)}\right\}$, it follows that at least the first $\left(L_{(i,k)} + E_{(i,k)}\right) \frac{n}{p^2 s}$ elements in each of these sequences are less than or equal to V_k and statement (A) follows.

Next, by definition, there are exactly $L_{(i,k)}$ samples originating from processor P_i whose values are strictly less than V_k. Hence, it follows that at least the first $L_{(i,k)} \frac{n}{ps}$ elements in the sorted input at processor P_i in Step (1) had to be strictly less than V_k. Since these elements are mapped as described above to the set of sequences $\left\{I_{(i,0)}, I_{(i,1)}, \ldots, I_{(i,p-1)}\right\}$, it follows that at least the first $L_{(i,k)} \frac{n}{p^2 s}$ elements in each of these sequences are strictly less than V_k and statement (B) follows.

Finally, since by definition there are exactly $L_{(i,k)}$ samples originating from processor P_i whose values are strictly less than V_k, it follows that the $\left(L_{(i,k)} + 1\right)^{th}$ sample originating from processor P_i must be greater than or equal to V_k. Hence, it follows that the element with index $\left(L_{(i,k)}\right) \frac{n}{ps}$ in the sorted input at processor P_i in Step (1) had to be greater than or equal to V_k, and that at most $\left(\left(L_{(i,k)}\right) \frac{n}{ps} - 1\right)$ elements in this sorted input were strictly less than V_k. Since these elements were mapped as described above to the set of sequences $\left\{I_{(i,0)}, I_{(i,1)}, \ldots, I_{(i,p-1)}\right\}$, it follows that the maximum number of elements in each of these sequences which are strictly less than V_k are $\left(\left(L_{(i,k)} + 1\right) \frac{n}{p^2 s} - 1\right)$ if $(j = p-1)$ and $\left(L_{(i,k)} + 1\right) \frac{n}{p^2 s}$ otherwise and statement (C) follows. □

In order to establish the complexity of our algorithm, we need to establish an upper bound in Step (6) on $\sum_{u=0}^{p-1} \left|T_{(u,j,k)}\right|$, the number of elements selected by processor P_j for routing to processor P_k. However, instead of trying to determine this bound directly, it turns out to be easier if we consider an alternative approach. Let $R_{(j,k)} = \left\{\bigcup T_{(u,j,w)} : 0 \leq u \leq p-1, 0 \leq w \leq k\right\}$ be the number of elements selected by processor P_j for routing

to processors P_0 through P_k. Clearly, $R_{(j,k)}$ includes all those elements that are strictly less than V_k, plus up to $\sum_{w=t}^{k} \text{Est}[w] \frac{n}{p^2 s}$ elements equal in value to V_k (if they can be found), where as before t is the smallest index less than or equal to k such that $\text{Splitter}[t] = V_k$. Then, an equivalent statement of our objective is to establish an upper bound on the difference between $\text{Min}_{(j,k)}$, the minimum possible size of $R_{(j,k-1)}$, and $\text{Max}_{(j,k)}$, the maximum possible size of $R_{(j,k)}$.

Lemma (2): The number of elements in Step (6) which will be selected by processor P_j for routing to processors P_0 through P_k is at least $\text{Min}_{(j,k)} = (k+1)\frac{n}{p^2}$.

Proof: Assertion (A) in Lemma (1) states that at least the first $\left(L_{(i,k)} + E_{(i,k)}\right) \frac{n}{p^2 s}$ elements in the sorted sequence $I_{(i,j)}$ are less than or equal to V_k. Assertion (B) in Lemma (1) states that at least the first $L_{(i,k)} \frac{n}{p^2 s}$ elements in the sorted sequence $I_{(i,j)}$ are strictly less than V_k. Hence, at most $E_{(i,k)} \frac{n}{p^2 s}$ of the first $\left(L_{(i,k)} + E_{(i,k)}\right) \frac{n}{p^2 s}$ elements in the sequence $I_{(i,j)}$ are equal to V_k. Hence, since our algorithm will select all those elements in the set $\left\{I_{(0,j)}, I_{(1,j)}, \ldots, I_{(p-1,j)}\right\}$ which are strictly less than V_k, plus up to $\sum_{u=0}^{p-1} E_{(u,k)} \frac{n}{p^2 s}$ elements which are equal in value to V_k (if they can be found), if follows that it will always be able to select at least

$$\begin{aligned} \text{Min}_{(j,k)} &= \sum_{u=0}^{p-1} \left(L_{(u,k)} + E_{(u,k)}\right) \frac{n}{p^2 s} \\ &= (k+1) s \frac{n}{p^2 s} \\ &= (k+1) \frac{n}{p^2} \end{aligned}$$

elements for routing to processors P_0 through P_k and Lemma (2) follows. □

Lemma (3): The number of elements in Step (6) which will be selected by processor P_j for routing to processors P_0 through P_k is at most

$$\text{Max}_{(i,r)} = \begin{cases} \left((ks+p)\frac{n}{p^2 s} - p\right) & \text{if } k = p-1 \\ (ks+p)\frac{n}{p^2 s} & \text{otherwise} \end{cases}$$

Proof: Assertion (C) in Lemma (1) states that the maximum number of elements in $I_{(i,j)}$ which are strictly less than V_k is at most $\left(\left(L_{(i,k)}+1\right)\frac{n}{p^2 s} - 1\right)$ if $(j = p-1)$ and $\left(L_{(i,k)}+1\right)\frac{n}{p^2 s}$ otherwise. Hence, since our algorithm will select all those elements in the set $\left\{I_{(i,0)}, I_{(i,1)}, \ldots, I_{(i,p-1)}\right\}$ which are strictly less than V_k, plus up to $\sum_{u=0}^{p-1} E_{(u,k)} \frac{n}{p^2 s}$ elements which are equal in value to V_k (if they can be found), if follows that it will always be able to select at most

$$\text{Max}_{(j,k)} = \begin{cases} \sum_{u=0}^{p-1} \left(\left(L_{(u,k)} + 1 + E_{(u,k)}\right)\frac{n}{p^2 s} - 1\right) = \\ \qquad \left(((k+1)s+p)\frac{n}{p^2 s} - p\right) & \text{if } k = p-1 \\ \sum_{u=0}^{p-1} \left(L_{(u,k)} + 1 + E_{(u,k)}\right)\frac{n}{p^2 s} = \\ \qquad ((k+1)s+p)\frac{n}{p^2 s} & \text{otherwise} \end{cases}$$

elements for routing to processors P_0 through P_k and Lemma (3) follows. □

Lemma (4): The number of elements sent by processor P_j to processor P_k is at most $\left(\frac{n}{p^2} + \frac{n}{sp} - p\right)$ for $j = p-1$ and $\left(\frac{n}{p^2} + \frac{n}{sp}\right)$ otherwise.

Proof: As noted earlier, the maximum number of elements selected by processor P_j for processor P_k is simply the difference between $\text{Min}_{(j,k-1)}$, the minimum possible number of elements selected for processors P_0 through $P_{(k-1)}$, and $\text{Max}_{(j,k)}$, the maximum possible number of elements selected for processors P_0 through P_k. Subtracting the lower bound of Lemma (2) from the upper bound of Lemma(3), we get:

$$\text{Max}_{(j,k)} - \text{Min}_{(j,k-1)} = \begin{cases} \left(((k+1)s+p)\frac{n}{p^2 s} - p\right) - ks\frac{n}{p^2 s} = \\ \qquad \left(\frac{n}{p^2} + \frac{n}{ps} - p\right) & \text{if } j = p-1 \\ ((k+1)s+p)\frac{n}{p^2 s} - ks\frac{n}{p^2 s} = \\ \qquad \left(\frac{n}{p^2} + \frac{n}{ps}\right) & \text{otherwise} \end{cases} \quad \square$$

Theorem 1: The overall complexity of our algorithm for sorting by regular sampling is $O\left(\frac{n}{p}\log\frac{n}{p} + \tau + \frac{n}{p}\sigma\right)$, for $n \geq p^3$ and $\left(p \leq s \leq \frac{n}{p^2}\right)$.

Analysis: 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. Step (3) involves selecting sp samples, and Step (4) involves merging p sorted sequences followed by p binary searches. Since only processor $P_{(p-1)}$ is active in these two steps, they require $O(sp)$ and $O(sp\log p)$ time, respectively. Step (6) involves a total of $O(p^2)$ binary searches on sequences of size $\frac{n}{p^2}$ and therefore requires $O(p^2\log\frac{n}{p^2})$ time. Lemma (4) guarantees that at the end of Step (7) no processor holds more than $\left(\frac{n}{p} + \frac{n}{s} - p\right)$ elements. Hence, the consolidation of Step (8) and the comparison-based merge of Step (9) require $O\left(\frac{n}{p} + \frac{n}{s} + p^2 - p\right)$, and $O\left(\left(\frac{n}{p} + \frac{n}{s} - p\right)\log p\right)$ time, respectively. Steps (2), (5), and (7) call the communication primitives `transpose`, `bcast`, and `transpose`, respectively. The analysis of these primitives in [4] 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

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

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

Clearly, our algorithm is asymptotically optimal and appears to have 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.

Before proceeding to this discussion, however, we describe first a modification of our algorithm which we used in the actual implementation. This modification has has no impact on the overall complexity, but it does result in a more efficient implementation. The basis for the modification is the observation that, in Step (8), the p^2 subsequences to be merged at each processor can be reduced to p subsequences based simply on the initial origin of their elements. Specifically, the set of p subsequences $\{T_{(i,0,k)}, T_{(i,1,k)}, \ldots, T_{(i,p-1,k)}\}$ at processor P_k share a a common origin in the sorted input at processor P_i in Step (1). They are the elements form the set of sorted sequences $\{I_{(i,0)}, I_{(i,1)}, \ldots, I_{(i,p-1)}\}$ which fall between Splitter$[k-1]$ and Splitter$[k]$. Recall also that the set of of sequences $\{I_{(i,0)}, I_{(i,1)}, \ldots, I_{(i,p-1)}\}$ were generated by "dealing out" the sorted input at processor

P_i, and that this mapping is clearly reversible. Hence, with a little bit of information, it is possible to reverse this mapping and reassemble the p subsequences $\{T_{(i,0,k)}, T_{(i,1,k)}, \ldots, T_{(i,p-1,k)}\}$ into a single sorted sequence $U_{(i,k)}$ without actually making any comparison between the values. The advantage of this process of reassembling is that it can be done in $O\left(\frac{n}{p}\right)$ time instead of time $O\left(\frac{n}{p}\log p\right)$ required for binary merge. Note that, while this has no effect on the overall complexity as binary merging is still required to merge the resulting set of p sequences $\{U_{(i,0)}, U_{(i,1)}, \ldots, U_{(i,p-1)}\}$, it does make an appreciable difference in the experimentally measured performance.

More specifically, in Step (5), each sequence $T_{(i,j,k)}$ is tagged with a set of values $< a,b,c,d >$, where $a_{(i,j,k)}$ specifies the number of elements in $T_{(i,j,k)}$ with the same value as Splitter$[k-1]$, $b(i,j,k)$ is the starting index in $I_{(i,j)}$ of the $c(i,j,k)$ values in $T_{(i,j,k)}$ which are strictly greater than Splitter$[k-1]$ and strictly less than Splitter$[k]$, and $d(i,j,k)$ specifies the number of elements in $T_{(i,j,k)}$ with the same value as Splitter$[k]$. Then, Step (8) can be rewritten into two steps as follows.

- Step (8*): Each processor P_k "unshuffles" the set of p subsequences $\{T_{(i,0,k)}, T_{(i,1,k)}, \ldots, T_{(i,p-1,k)}\}$ originating with the sorted input at processor P_i in Step (1) to form the sequence $U_{(i,k)}$, for $(0 \leq i \leq p-1)$. This procedure involves the following three substeps, in which both $T_{(i,j,k)}$ and $U_{(i,k)}$ are most easily thought of as stack data structures:
  - (A): For each subsequence in the set $\{T_{(i,0,k)}, T_{(i,1,k)}, \ldots, T_{(i,p-1,k)}\}$, pop the first $a_{(i,j,k)}$ elements (which have the same value as Splitter$[k-1]$) and push them onto $U_{(i,k)}$.
  - (B): Identify the smallest-indexed subsequence from the set whose associated value of $b_{(i,j,k)}$ is also less than or equal to all the other values of $b_{(i,j,k)}$ in the set. Beginning at this subsequence and proceeding in ascending order of index j (with wrap around), pop one element off each subsequence and push it onto $U_{(i,k)}$. The procedure continues in this round-robin fashion until all $c_{(i,j,k)}$ elements which are strictly between Splitter$[k-1]$ and Splitter$[k]$ have been removed from each subsequence $T_{(i,j,k)}$.
  - (C): For each subsequence in the set $\{T_{(i,0,k)}, T_{(i,1,k)}, \ldots, T_{(i,p-1,k)}\}$, pop the remaining $d_{(i,j,k)}$ elements (which have the same value as Splitter$[k]$) and push them onto $U_{(i,k)}$.
- Step (9*): Each processor P_k merges the set of p consolidated subsequences $\{U_{(0,k)}, U_{(1,k)}, \ldots, U_{(p-1,k)}\}$ to produce the k^{th} column of the sorted array.

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 MAX_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 $\left(2^{31}-1\right)$, 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 $\left(2^{-30} \times \text{MAX}_\text{D}\right)$. Note that the same procedure is used throughout to generate the double benchmark from the integer benchmark.

2. **Gaussian [G]**, a Gaussian distributed random input, approximated by adding four calls to `random()` and then dividing the result by four.
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.
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 0 up to $\left(\frac{p}{g}-1\right)$, then for group j we set the first $\frac{n}{pg}$ elements to be random numbers between $\left(\left(\left(jg+\frac{p}{2}-1\right) \bmod p\right)+1\right)\frac{2^{31}}{p}$ and $\left(\left(\left(\left(jg+\frac{p}{2}\right) \bmod p\right)+1\right)\frac{2^{31}}{p}-1\right)$, the second $\frac{n}{pg}$ elements at each processor to be random numbers between $\left(\left(\left(jg+\frac{p}{2}\right) \bmod p\right)+1\right)\frac{2^{31}}{p}$ and $\left(\left(\left(\left(jg+\frac{p}{2}+1\right) \bmod p\right)+1\right)\frac{2^{31}}{p}-1\right)$, and so forth.
6. **Staggered [S]**, created as follows: if the processor index i is less than or equal to $\left(\frac{p}{2}-1\right)$, 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 $\left((2i+2)\frac{2^{31}}{p}-1\right)$. Otherwise, we set all $\frac{n}{p}$ elements to be random numbers between $\left((2i-p)\frac{2^{31}}{p}\right)$ and $\left((2i-p+1)\frac{2^{31}}{p}-1\right)$.
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 sample sorting algorithm described in Section 3. Specifically, 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 the available code for generating this benchmarks.
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.

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-Load Regular benchmark was developed to induce the worst possible load distribution at the completion of sorting and was included to study the behavior of our algorithm in the guaranteed worst-case scenario. Finally, the Deterministic Duplicates and the Randomized Duplicates benchmarks were included to assess the performance of the algorithms in the presence of duplicate values.

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) [6] 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.

The results in **Tables I** and **II** together with their graphs in **Figures 1** and **2** 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 Steps (3) and (4) are respectively $O(sp)$ and $O(sp\log p)$, whereas the complexity of Step (9) is $O\left(\left(\frac{n}{p}+\frac{n}{s}-p\right)\log p\right)$. Hence, the first terms are

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.

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.

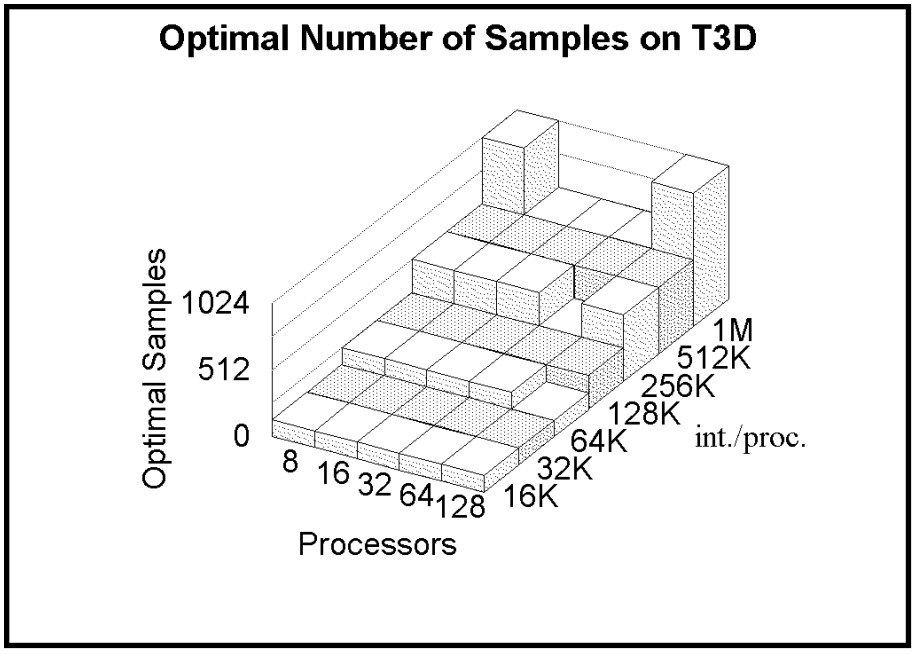

Figure 1: Optimal number of samples *s* for sorting the [WR] integer benchmark on the Cray T3D, for a variety of processors and input sizes.

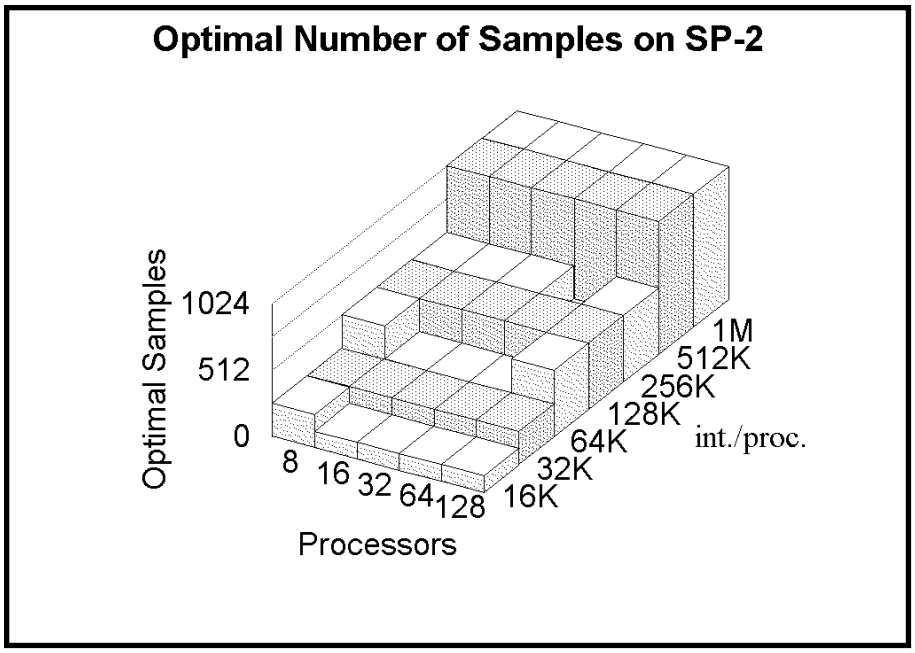

Figure 2: Optimal number of samples s for sorting the [WR] integer benchmark on the IBM SP-2-WN, for a variety of processors and input sizes.

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.

The results in **Tables III** through **VI** together with their graphs in **Figures 3** through **6** 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 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 7** 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

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.

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.

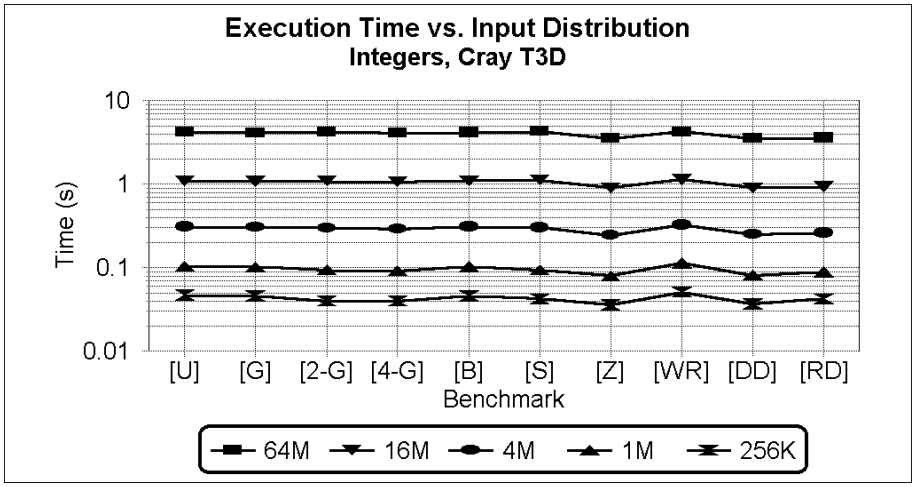

Figure 3: Total execution time (in seconds) required to sort a variety of integer benchmarks on a 64-node Cray T3D.

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,

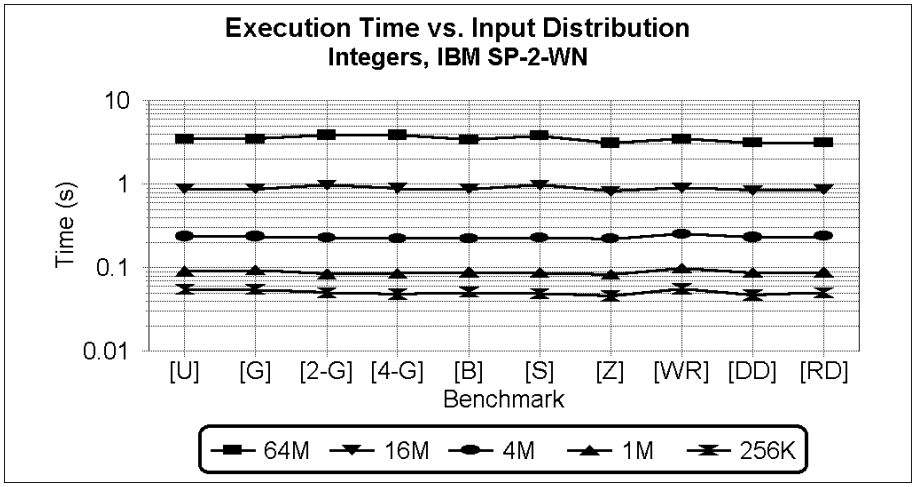

Figure 4: Total execution time (in seconds) required to sort a variety of integer benchmarks on a 64-node IBM SP-2-WN.

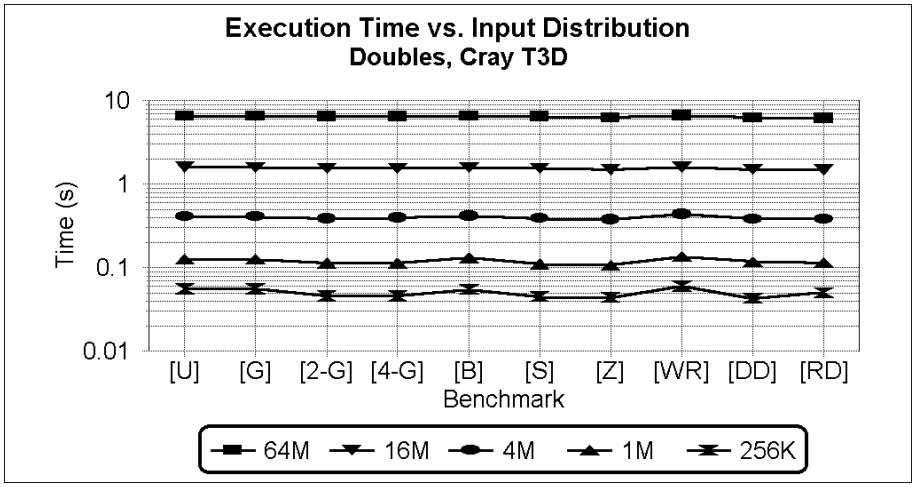

Figure 5: Total execution time (in seconds) required to sort a variety of double benchmarks on a 64-node Cray T3D.

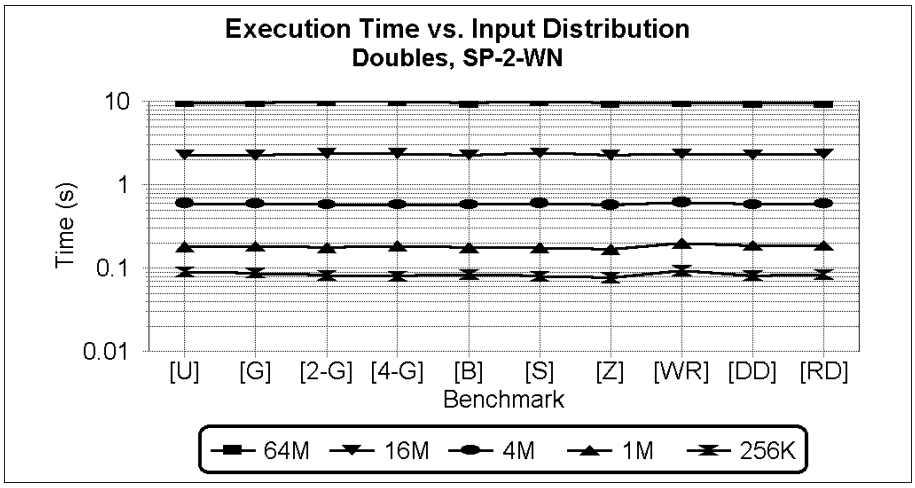

Figure 6: Total execution time (in seconds) required to sort a variety of double benchmarks on a 64-node IBM SP-2-WN.

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.

Regular Sorting of 8M Doubles [WR]					
Machine	**Number of Processors**				
	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.

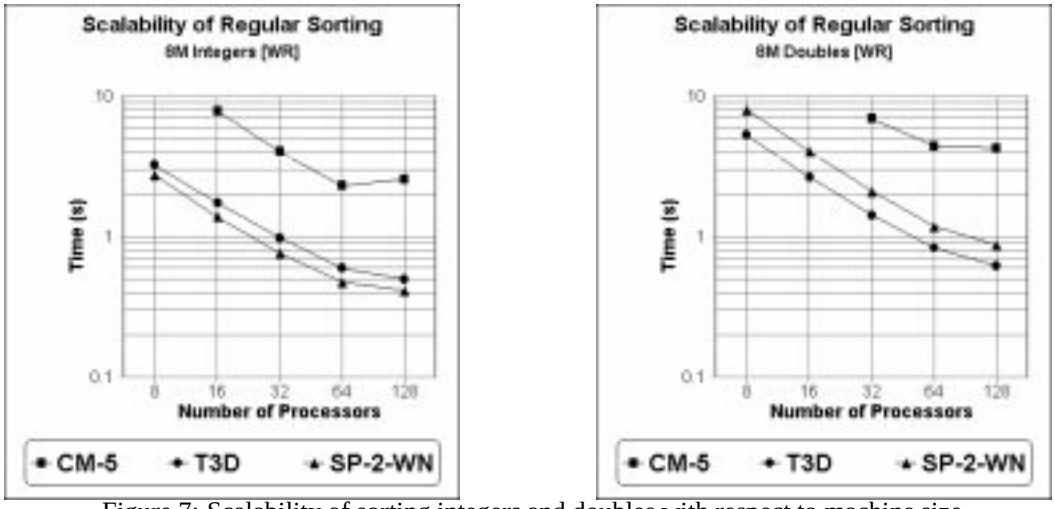

Figure 7: Scalability of sorting integers and doubles with respect to machine size.

However, the results in Tables VII and VIII together with their graphs in Figure 7 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 of τ on p simply becomes more pronounced as

Step by Step Breakdown of Sorting 8M Integers					
Step	**Number of Processors (Number of Samples)**				
	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.

p increases and $\frac{n}{p}$ decreases. Step (3) exhibits the $O(sp)$ complexity we anticipated, since for $2^{\left\lfloor \frac{1}{2}\log(n/p)\right\rfloor}$, s is halved every other time p is doubled. Steps (6) and (9) display the expected $O\left(p^2\log p\right)$ and $O\left(\left(\frac{n}{p}+\frac{n}{s}\right)\log p\right)$ $\left(\approx O\left(\left(\frac{n}{p}+\sqrt{np}\right)\log p\right)\right.$ for $\left.s\approx\sqrt{\frac{n}{p}}\right)$ 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 exhibit $O\left(\frac{n}{p}+\frac{n}{s}\right)$ $\left(\approx O\left(\frac{n}{p}+\sqrt{np}\right)\right.$ for $\left.s\approx\sqrt{\frac{n}{p}}\right)$ 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 8** 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 9** 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 only about 10% of the execution time on the T3D and about 20% of the execution time on the SP-2. The difference in the distribution between these two platforms is likely due in part to the fact that an integer is 64 bits on the T3D while only 32 bits on the SP-2. By contrast, doubles are 64 bits on both platforms. For $n = 64M$ doubles, the sequential sorting, unshuffling, and merging performed in Steps (3), (8), and (9) consume approximately 80% of the execution time on both platforms, whereas the two `transpose` operations in Steps (2) and (7) together consume only about 15% of the execution time. Together, these results show that our algorithm is extremely efficient in its communication performance.

4.3 Comparison with Previous Results

Despite the theoretical interest in deterministic sorting algorithms, we were able to locate relatively few empirical studies. All but one of these studies used machines which are no longer available. The single exception is the recent work of Gerbessiotis and Siniolakis [12, 13]. **Table X** compares the performance of their deterministic algorithm with that of our regular sampling algorithm on an IBM SP-2.

Problem Size	2 Processors		4 Processors		8 Processors	
	HJB	GS	HJB	GS	HJB	GS
128K	0.117	0.709	0.066	0.399	0.041	0.217
512K	0.499	2.87	0.285	1.54	0.144	0.844
1M	1.01	5.73	0.566	3.06	0.307	1.61

Table X: Total execution time (in seconds) to sort the [U] integer benchmark on the IBM SP-2, comparing our results (HJB) with those reported by Gerbessiotis and Siniolakis (GS)

.

4.4 Comparison With Our Random Sample Sort Algorithm

Table XI compares the performance of our sorting by regular sampling algorithm with that of our random sample sort algorithm [14] on both the T3D and the SP-2-WN using the [WR] benchmark. If $T_{\mathrm{RS}}(n,p)$ represents the time required by our regular sample sort algorithm and $T_{\mathrm{SS}}(n,p)$ represents the time required by our random sample sort algorithm, then the corresponding entry in Table XI is $(T_{\mathrm{RS}}(n,p) - T_{\mathrm{SS}}(n,p))$ as a percentage of $T_{\mathrm{SS}}(n,p)$. Thus, a negative entry indicates that the regular sample sort runs faster than the random sample sort algorithm. The results largely confirm what we would expect: large values of p together with relatively small values of n make the performance of our regular sampling algorithm uncompetitive when compared with our sample sort algorithm. The reason for this is that, unlike our regular sampling algorithm, none of the steps in our sample sort algorithm exhibit such strong dependence on p. But aside from this subset of problems, the performance of the two algorithms is comparable. Here, regular sampling would seem to have the advantage, because it deterministically guarantees the performance bounds and the memory requirements for any input.

Comparison of Regular Sampling with Sample sort						
Machine (Type)	Keys/ Proc.	Number of Processors				
		8	16	32	64	128
T3D (integers)	64K	-1.9 %	0.4 %	6.2 %	24.5 %	74.6 %
	256K	-2.2 %	-1.2 %	0.8 %	10.8 %	29.1 %
	1M	-3.0 %	-2.8 %	-0.8 %	4.5 %	13.8 %
SP-2-WN (integers)	64K	1.2 %	-1.2 %	5.8 %	16.6 %	39.3 %
	256K	12.5 %	8.9 %	9.5 %	10.8 %	-1.6 %
	1M	13.3 %	11.8 %	10.7 %	6.5 %	-6.4 %
T3D (doubles)	64K	-2.8 %	-3.2 %	3.0 %	19.5 %	57.1 %
	256K	-2.4 %	-3.0 %	0.0 %	4.6 %	21.6 %
	1M	-3.0 %	-2.7 %	-1.7 %	1.5 %	8.2 %
SP-2-WN (doubles)	64K	-1.6 %	-1.3 %	-1.9 %	5.9 %	31.0 %
	256K	-1.1 %	-0.5 %	0.5 %	4.0 %	9.6 %
	1M	0.1 %	1.9 %	2.1 %	1.3 %	5.0 %

Table XI: Comparison of time required by our regular sampling algorithm with the time required by our sample sort algorithm using our [WR] benchmark. If $T_{\mathrm{RS}}(n,p)$ represents the time required by our regular sampling algorithm and $T_{\mathrm{SS}}(n,p)$ represents the time required by our random sample sort algorithm, then the corresponding entry is $(T_{\mathrm{RS}}(n,p) - T_{\mathrm{SS}}(n,p))$ as a percentage of $T_{\mathrm{SS}}(n,p)$. Thus, a negative entry indicates that the regular sample sort runs faster than the random sample sort algorithm.

Acknowledgments

We would like to thank Ronald Greenberg of the Department of Mathematical and Computer Sciences at Loyola University, Chicago, 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.

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) [8] on which the T3D port of SPLIT-C has been based.

We also thank the Numerical Aerodynamic Simulation Systems Division of NASA's 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 related work by the authors.

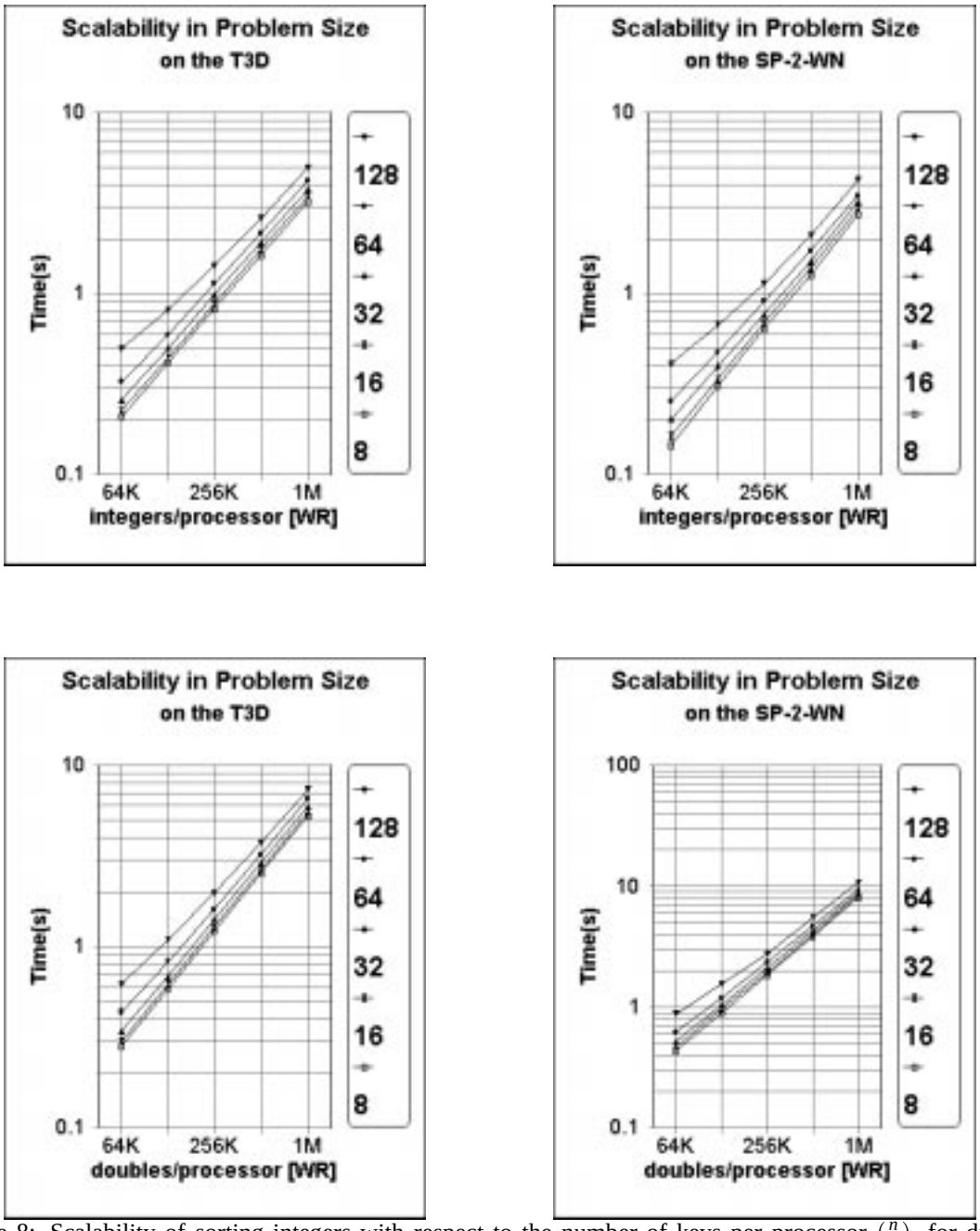

Figure 8: Scalability of sorting integers with respect to the number of keys per processor $(\frac{n}{p})$, for differing numbers of processors.

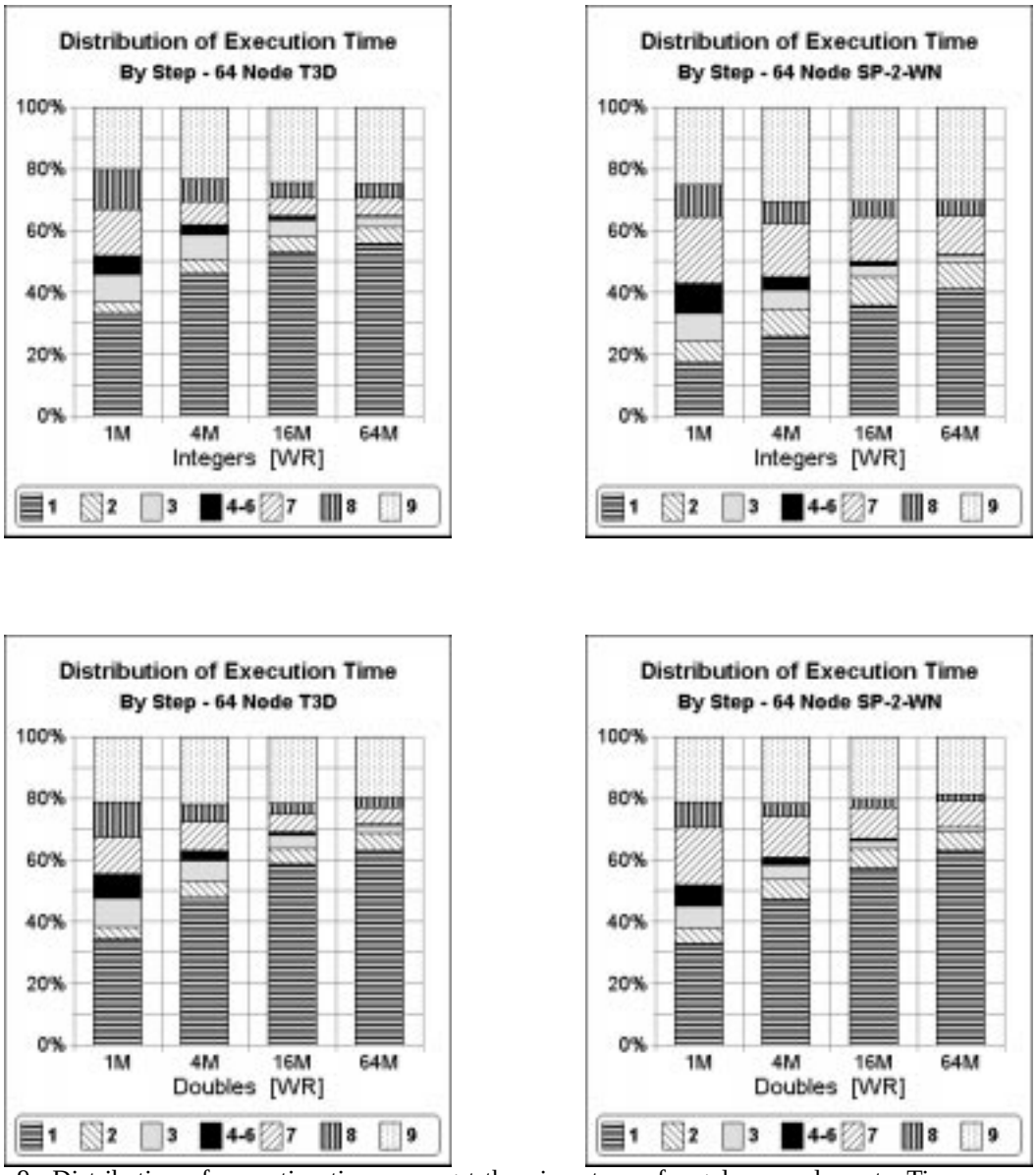

Figure 9: Distribution of execution time amongst the nine steps of regular sample sort. Times are obtained for both a 64 node T3D and a 64 node SP-2-WN using both the integer and the double versions of the [WR] benchmark.

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] David A. Bader, David R. Helman, and J. JáJá. Practical Parallel Algorithms for Personalized Communication and Integer Sorting. *ACM Journal of Experimental Algorithmics*, 1(3):1–42, 1996. http://www.jea.acm.org/1996/BaderPersonalized/.

[4] David 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. Presented at the 10th *International Parallel Processing Symposium*, pages 292-301, Honolulu, HI, April 15-19, 1996.

[5] David A. Bader and J. JáJá. Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study. *Journal of Parallel and Distributed Computing*, 35(2):173–190, June 1996.

[6] 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.

[7] 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. To appear in the *Communications of the ACM.*

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

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

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

[11] 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.

[12] A.V. Gerbessiotis. Data for Regular Sorting. Personal Communication, July 1996.

[13] A.V. Gerbessiotis and C.J. Siniolakis. Deterministic Sorting and Randomized Median Finding on the BSP Model. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 223–232, Padua, Italy, June 1996.

[14] David R. Helman, David A. Bader, and J. JáJá. A Randomized Parallel Sorting Algorithm With an Experimental Study. Technical Report CS-TR-3669 and UMIACS-TR-96-53, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, August 1996. To appear in the *Journal of Parallel and Distributed Computing.*

[15] 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.

[16] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 1994. Version 1.0.

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

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