Sorting on Clusters of SMPs

David R. Helman Joseph JáJá*

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

August 28, 1997

Abstract

Clusters of symmetric multiprocessors (SMPs) have emerged as the primary candidates for large scale multiprocessor systems. In this paper, we introduce an efficient sorting algorithm for clusters of SMPs. This algorithm relies on a novel scheme for stably sorting on a single SMP coupled with balanced regular communication on the cluster. Our SMP algorithm seems to be asymptotically faster than any of the published algorithms we are aware of. The algorithms were implemented in C using Posix Threads and the SIMPLE library of communication primitives and run on a cluster of DEC AlphaServer 2100A systems. Our experimental results verify the scalability and efficiency of our proposed solution and illustrate the importance of considering both memory hierarchy and the overhead of shifting to multiple nodes.

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

*Supported in part by NSF grant No. CCR-9627210 and NSF HPCC/GCAG grant No. BIR-9318183.

1 Introduction

Clusters of symmetric multiprocessors (SMPs) have emerged as the primary candidates for large scale multiprocessor systems. In spite of this trend, relatively little work has been done to develop techniques for designing algorithms which make effective use of the resources available on such platforms. This task is made difficult by the contrasting requirements of the platform. On the one hand, each SMP must be viewed on its own as a hierarchical shared memory machine. Good performance requires both good load distribution and the minimization of main memory access. On the other hand, from the perspective of the cluster, each node is in effect a superproccessor, and therefore the cluster of SMPs is a collection of powerful processors connected by a communication network. Maximizing the performance of such a distributed memory machine requires both efficient load balancing and regular balanced communication.

In this paper, we examine the problem of sorting on SMP clusters. Sorting is arguably the most studied problem in computer science, due in large part to its pervasiveness. But it is also intrinsically interesting because of its demanding requirements for irregular memory access and interprocessor communication. From the perspective of the cluster, any algorithm which performs well for distributed memory machines would be a reasonable candidate. In particular, we have identified two such algorithms in our previous work [11, 10, 12]. The first is a variation on sample sort and the other is a variation on the approach of sorting by regular sampling. On the other hand, from the perspective of the individual SMP, there are fewer choices for sorting on hierarchical shared memory machines. These include distribution sort [4], greed sort [13], balance sort [14], sharesort [3], simple randomized merge sort [8], radix sort, and the sorting algorithm of Varman et al. [16, 15]. Unfortunately, none of these algorithms by themselves is sufficient to achieve efficient performance on an SMP cluster. Efficient algorithms for distributed memory machines tend to confine interprocessor communication to a minimum number of regular balanced exchanges. By contrast, algorithms for shared memory machines typically capitalize on the fact that accessing data associated with another processor is no more expensive than accessing its own data in the shared memory.

We introduce a sorting algorithm for clusters of SMPs which is a hybrid of our modified algorithms for parallel sorting by random sampling and parallel sorting by deterministic sampling. Our algorithm was implemented in C using Posix Threads and runs on a cluster of DEC AlphaServer 2100A systems. We ran our code using a variety of benchmarks that we have identified to examine the dependence of our algorithm on the input distribution. Our experimental results verify the scalability and efficiency of our proposed solution and illustrate the importance of considering both memory hierarchy and the overhead of shifting to multiple nodes.

The organization of the paper is as follows. **Section 2** presents our computational model for analyzing algorithms for SMP clusters. **Section 3** describes in detail our sorting algorithm for this platform. Finally, **Section 4** describes the experimental performance of our sorting algorithm on an SMP cluster.

2 Computational Model

For our purposes, the cost of an algorithm needs to include a measure that reflects the number and type of memory accesses. A memory access can either be to a local cache, a local main memory, or a remote main memory. It is instructive to start with a brief overview of a number of models that have been proposed to capture the performance of multilevel hierarchical memories.

Many of the models in the literature are specifically limited to two-level memories. Aggarwal and Vitter [4] first proposed a simple model for main memory and disks which recognized the importance of spatial locality. In their uniprocessor model, D blocks of B contiguous records can be transferred between primary and secondary memory in a single I/O. Vitter and Shriver [17] proposed the more realistic D-disk model, in which secondary storage is partitioned into D physically distinct disk drives. In this model, D blocks can be transfered in a single I/O but only if no two blocks are from the same disk. Finally, Vitter and Shriver [17] extended the D-disk model to the parallel domain. In the so-called parallel disk model, the processors are directly connected to one another by some interconnection network which can be modeled by any of the standard models (e.g. PRAM, fixed-connection network). For all these models, the cost of accessing data on disk was substantially higher than internal computation, and, hence, the sole measure of performance used is the number of parallel I/Os.

Alternatively, there are a number of models which allow for any arbitrary number of memory levels. Focusing on the fact that access to different levels of memory are achieved at differing costs, Aggarwal et al. [1] introduced the Hierarchical Memory Model (HMM), in which access to location x requires time $f(x)$, where $f(x)$ is any monotonic nondecreasing function. Taking note of the fact that the latency of memory access makes it economical to fetch a block of data, Aggarwal, Chandra, and Snir [2] extended this model to the Hierarchical Memory with Block Transfer Model (BT). In this model, accessing t consecutive locations beginning with location x requires time $f(x)+t$. Finally. Nodine and Vitter [14] generalized this model to the parallel domain in exactly the same fashion as the D-disk model. The result is the Parallel Hierarchical Model with Block Transfer (P-BT), in which the individual memory hierarchies are linked at the CPUs by some appropriate network topology.

These models all assume that while the buses which connect the various levels of memory might be simultaneously active, this only occurs in order to cooperate on a single transfer. Partly in response to this limitation, Alpern et al. [5] proposed the Uniform Memory Hierarchy Model (UMH). In this model, the l^{th} memory level consists of $\alpha\rho^l$ blocks, each of size ρ^l, and a block of data can be transfered between level $l+1$ and level l in time $\rho^l/b(l)$, where $b(l)$ is the bandwidth. This model has been extended to the parallel domain in two ways as described in [5] and [13]. However, the UMH model is unnecessarily complicated for use with clusters of SMPs and requires significant refinements to capture the corresponding hybrid architecture.

In our complexity model, each SMP is viewed as a two-level hierarchy for which good performance requires both good load distribution and the minimization of secondary memory access. The cluster is viewed as a collection of powerful processors connected by a communication network. Maximizing performance on the cluster requires both efficient load balancing and regular balanced communication. Hence, our performance model combines two separate but complimentary models.

Note that our approach is distinct from two methodologies that are currently being promoted for programming clusters of SMPs. The first approach is to view the platform as a distributed shared memory system, using a software layer to simulate coherent shared memory between nodes by transparently using messages to move around specific data or referenced memory pages. The second approach, based on message passing primitives, enforces a shared-nothing paradigm between tasks, and all communication and coordination between tasks are performed through the exchange of explicit messages, even tasks on a node with physically shared memory. Both of these methodologies accept inefficiencies in order to simplify programmability and portability. However, these current approaches lead to significant inefficiencies that will make them unacceptable for a wide range of problems.

In our SMP model, we recognize that efficient algorithm design requires the efficient decomposition of the problem amongst the available processors, and so, unlike some other models for hierarchical memory mentioned in our introduction, we include the cost of computation in our complexity. But at the same time, our cost model encourages the exploitation of temporal and spatial locality. Specifically, memory at each SMP is seen as consisting of two levels: cache and main memory. A block of w contiguous words can be read from or written to main memory in $\left(\epsilon + \frac{wr}{\kappa}\right)$ time, where ϵ is the latency of the bus, r is the number of processors competing for access to the bus, and κ is the bandwidth. By contrast, the transfer of w noncontiguous words would require $w\left(\epsilon + \frac{r}{\kappa}\right)$ time.

We capture performance on an SMP cluster in exactly the same way as described in [11, 10, 12]. We view a parallel algorithm as a sequence of local SMP computations interleaved with communication steps, where we allow computation and communication to overlap. Assuming no congestion, the transfer of a block consisting of w contiguous words between two nodes takes $\left(\tau + \frac{w}{\beta}\right)$ time, where τ is the latency of the network and β is the bandwidth per node. We assume that the bisection bandwidth is sufficiently high to support block permutation routing amongst the N nodes at the rate of $1/\beta$. In particular, for any subset of r nodes, a block permutation amongst the r nodes takes $\left(\tau + \frac{w}{\beta}\right)$ time, where w is the size of the largest block. Using this cost model, we can evaluate the communication time $T_{comm}(n, N)$ of an algorithm as a function of the input size n, the number of nodes N , and the parameters τ and β. The overall complexity of the cluster $T_{(n,N)}$ is given by the sum of T_{SMP} and $T_{comm}(n, N)$.

Note that our model for clusters of SMPs is similar to the parallel D-disk model for the case of a single disk. It is also similar to the P-BT model, if $f(x)$ is chosen to be a step function equal to zero for all those locations in cache and equal to some positive constant for all other locations in main memory. However, there are certain differences. First, the parallel version of these models considers the processors to be joined by some network topology at the processor level. By contrast, we view the processors at a node to be joined on the level of shared main memory and only the nodes themselves to be joined by an interconnection network, and, moreover, we explicitly include the computation cost.

3 Sorting Algorithms

For simplicity of presentation, we first consider an algorithm for sorting on a single SMP, and then describe an algorithm for sorting on clusters of SMPs.

3.1 Sorting on a Single SMP

Several sorting algorithms that have been proposed in the literature for hierarchical memory models can be considered for possible implementation on an SMP. For example, distribution sort [4] and its more refined version balance sort [14] partition the input elements into buckets using a set of approximately evenly spaced splitters and then sort the contents of each bucket recursively. While they are efficient in memory access, they are not optimal in their computational costs since at each level of recursion they sort the elements in blocks corresponding to the size of the cache. Other sorting algorithms such as greed sort [13] and simple randomized merge sort [8] are bottom-up sorting algorithms. These are sophisticated versions of merge sort, designed to make optimal use of multiple independent disks. However, straightforward merging is inherently sequential, so without modifications these algorithms would not be expected to perform efficiently with multiple processors. Finally, there is sharesort [3], which is a hybrid of the first two approaches. Briefly, the input of n elements is evenly divided into n^{γ} subsets, where γ is some constant between 0 and 1, and the resulting subsets are recursively sorted. A set of precisely evenly-spaced splitters are then computed to divide the sorted lists, after which the k sorted subsets which lie between each pair of splitters are merged to produce the sorted output. The actual algorithm is considerably more complex, since the authors designed it for very general hierarchical memory models.

Since an SMP is considerably simpler than the intended target of these algorithms, there are at least three other algorithms which merit consideration. The first is an adaptation of radix sort, in which each processor starts by computing a histogram for its portion of the input elements. The results of all p histograms are then shared so that each processor can compute its offset values in a global context, which are then used to rearrange the data in parallel. However, depending on the input, this data rearrangement could cause very poor cache performance, which would make this algorithm uncompetitive with other possible choices.

Another approach proposed by Varman et al. [16, 15]. starts by having each processor sorts $\frac{n}{p}$ elements from the input set using C-way merge sort, where C is the size of the cache. Next, a subset of p evenly spaced samples are drawn from each of the p sorted sequences, after which a set of p splitters is identified which partition the subset of samples evenly The size of the sample set is then doubled and the process repeated until the set of samples includes all n input values.

Yet another algorithm is an adaptation of our sorting by regular sampling [11, 12], which we originally developed for distributed memory machines. As modified for an SMP, this algorithm is similar to the parallel sorting by regular sampling (PSRS) algorithm, except that our algorithm can be easily implemented as a stable sort.

The pseudocode for our algorithm is as follows, where C is the size of the cache and L is the cache line size:

- **(1)** Each processor P_i $(1 \leq i \leq p)$ sorts the subsequence of the n input elements with indices $\left(\frac{(i-1)n}{p} + 1\right)$ through $\left(\frac{in}{p}\right)$ as follows:
  - **(A)** Sort each block of m input elements using an appropriate sequential algorithm, where $m \leq C$. For integers we use the radix sort algorithm, whereas for floating point numbers we use the merge sort algorithm.
  - **(B)** For $j = 1$ up to $\left(\frac{\log(n/pm)}{\log(z)}\right)$, merge the sorted blocks of size $\left(mz^{(j-1)}\right)$ using z-way merge, where $z \leq \frac{C}{L}$.
- **(2)** Each processor P_i selects each $\left(k\frac{n}{ps}\right)^{th}$ element as a sample, for $(1 \leq k \leq s)$ and a given value of s $\left(p \leq s \leq \frac{n}{p^2}\right)$.
- **(3)** Processor P_p merges the p sorted subsequences of samples and then selects the $(ks)^{th}$ sample as Splitter[k], for $(1 \leq k \leq p-1)$. By default, the p^{th} splitter is the largest value allowed by the data type used. Additionally, binary search is used to compute for the set of samples with indices 1 through (ks) the number of samples Est[k] which share the same value as Splitter[k].
- **Step (4):** Each processor P_i uses binary search to define an index $b_{(i,j)}$ for each of the p sorted input sequences created in **Step (1)**. If we define $T_{(i,j)}$ as a subsequence containing the first $b_{(i,j)}$ elements in the j^{th} sorted input sequence, then the set of p subsequences $\{T_{(i,1)}, T_{(i,2)}, ..., T_{(i,p)}\}$ will contain all those values in the input set which are strictly less then Splitter[i] and at most $\left(\text{Est}[i] \times \frac{n}{ps}\right)$ elements with the same value as Splitter[i].
- **Step (5):** Each processor P_i merges those subsequences of the sorted input sequences which lie between indices $b_{((i-1),j)}$ and $b_{(i,j)}$ using p-way merge.

It is straightforward to see how this algorithm can be implemented as a stable integer sort. **Step (1A)** sorts the input elements block by block. As long as a stable sequential sort is used and the relative order of the blocks is preserved, stability is maintained. The z-way merge in **Step (1B)** can be done using a tree of losers. If ties are resolved by choosing the element from that block which appears first in the input, then stability will be preserved. **Steps (2)** and **(3)** do not directly affect stability. **Step (4)** partitions the p sorted sequences generated by **Step (1B)**. If the $\left(\text{Est}[i] \times \frac{n}{ps}\right)$ elements with the same value as Splitter$[i]$ are chosen in a greedy fashion beginning with the first sequence and proceeding in order, stability will be preserved. Finally, stability can be preserved in the merging of **Step (5)** in exactly the same fashion as **Step (1B)**. Hence, our algorithm results in a stable integer sort.

Before establishing the complexity of this algorithm, we need the results of the following theorem developed in [11, 12]:

Theorem 1: At the completion of the partitioning in **Step (4)**, no more than $\left(\frac{n}{p} + \frac{n}{s} - p\right)$ elements will be associated with any splitter, for $n \geq p^2$ and $\left(p \leq s \leq \frac{n}{p^2}\right)$.

With the results of **Theorem 1**, the analysis of our algorithm for sorting by regular sampling is as follows. The cost of sequentially sorting $\frac{n}{p}$ elements in blocks of size m in **Step (1A)** depends on the data type - sorting integers using radix sort requires $O\left(\frac{n}{p} + \epsilon + \frac{n}{\kappa}\right)$ time, whereas sorting floating point numbers using merge sort requires $O\left(\frac{n}{p}\log m + \epsilon + \frac{n}{\kappa}\right)$ time. **Step (1B)** involves $\left(\frac{\log(n/pm)}{\log(z)}\right)$ rounds of z-way merge beginning with $\left(\frac{n}{pm}\right)$ blocks of size m, which requires only $O\left(\frac{n}{p} log\left(\frac{n}{pm}\right) + \frac{n}{pm}\epsilon + \frac{n}{\kappa}\frac{\log(n/pm)}{\log(z)}\right)$ time because of spatial locality. The reading of s noncontiguous samples in **Step (2)** requires $O\left(s + s\left(\epsilon + \frac{p}{\kappa}\right)\right)$ time. **Step (3)** involves a p-way merge of blocks of size s followed by p binary searches on segments of size s. Since only one processor is active in **Step (3)**, it requires $O\left(sp\log(p) + p\epsilon\log(s) + \frac{ps}{\kappa}\right)$ time. **Step (4)** involves p binary searches on segments of size $\frac{n}{p}$. Since these reads are noncontiguous, they require $O\left(p\log\left(\frac{n}{p}\right) + p\log\left(\frac{n}{p}\right)\left(\epsilon + \frac{p}{\kappa}\right)\right)$ time, Finally, **Step (5)**, involves a p-way merge of p blocks of total maximum size $\left(\frac{n}{p} + \frac{n}{s} - p\right)$, requiring at most $O\left(\left(\frac{n}{p} + \frac{n}{s}\right)\log p + p\epsilon + \left(\frac{n}{p} + \frac{n}{s}\right)\frac{p}{\kappa}\right)$ time. Hence, the overall complexity of our sorting algorithm is given (for floating point numbers) by

$$T(n,p) = O\left(\frac{n}{p}\log n + \left(\frac{n}{pm} + \frac{n}{p^2} + p\log\left(\frac{n}{p}\right)\right)\epsilon + \frac{\log\left(\frac{n}{pm}\right)}{\log(z)}\frac{n}{\kappa}\right) \tag{1}$$

for $n \geq p^2\log\left(\frac{n}{p}\right)$, $\left(p \leq s \leq \frac{n}{p^2}\right)$, $m \leq C$, and $z \leq \frac{C}{L}$. Note that since ϵ for an SMP data bus is small and its coefficient in this expression is far less than n, the complexity can be approximated by:

$$T(n,p) \approx O\left(\frac{n}{p}\log n + \frac{\log\left(\frac{n}{pm}\right)}{\log(z)}\frac{n}{\kappa}\right) \tag{2}$$

The analysis suggests that the parameters m and z should be as large as possible, subject to the stated constraints. Indeed, in such a case we believe our algorithm to be asymptotically faster than the performance achieved by straightforward implementations of the other algorithms on our model. However, our experimental investigation shows that the optimal choices for m and z are actually slightly smaller than suggested by our analysis, since they depend on a variety of factors besides those captured in our model.

3.2 Sorting On a Cluster of SMPs

We have already developed both a deterministic [11, 12] and a randomized [11, 10] sample sort algorithm which we have shown to be very efficient for message passing platforms. Either would be an appropriate choice for a cluster of SMPs, but we chose the randomized sample sort because it proved to be slightly faster in its implementation. We repeat the pseudocode here for convenience, where we replace each sequential step where appropriate by a multithreaded SMP implementation. Note that the communication primitives mentioned are described in detail in [6].

- **Step (1):** Using p threads, each node N_i $(1 \leq i \leq N)$ randomly assigns each of its $\frac{n}{N}$ elements to one of N buckets. With high probability, no bucket will receive more than $c_1 \frac{n}{N^2}$ elements, where c_1 is a constant to be defined later.
- **Step (2):** Each node N_i routes the contents of bucket j to node N_j, for $(1 \leq i, j \leq N)$. Since with high probability no bucket will receive more than $c_1 \frac{n}{N^2}$ elements, this is equivalent to performing a **transpose** operation with block size $c_1 \frac{n}{N^2}$.
- **Step (3):** Using p threads, each node N_i sorts at most $(\alpha_1 \frac{n}{N} \leq c_1 \frac{n}{N})$ values received in **Step (2)** with the appropriate version of our SMP sorting algorithm, depending on the data type.
- **Step (4):** From its sorted list of $(\gamma \frac{n}{N} \leq c_1 \frac{n}{N})$ elements, node N_1 selects each $\left(j\gamma \frac{n}{N^2}\right)^{th}$ element as Splitter$[j]$, for $(1 \leq j \leq N-1)$. By default, Splitter$[N]$ is the largest value allowed by the data type used. Additionally, for each Splitter$[j]$, binary search is used to determine the values $\mathrm{Frac}_{\mathrm{L}}[j]$ and $\mathrm{Frac}_{\mathrm{R}}[j]$, which are respectively the fractions of the total number of elements at node N_1 with the same value as Splitter$[j-1]$ and Splitter$[j]$ which also lie between index $\left((j-1)\gamma \frac{n}{N^2} + 1\right)$ and index $\left(j\gamma \frac{n}{N^2}\right)$, inclusively.
- **Step (5):** Node N_1 **broadcasts** the Splitter, $\mathrm{Frac}_{\mathrm{L}}$, and $\mathrm{Frac}_{\mathrm{R}}$ arrays to the other $N-1$ nodes.
- **Step (6):** Each node N_i uses binary search on its sorted local array to define for each of the N *splitters* a subsequence S_j. The subsequence associated with Splitter$[j]$ contains all those values which are greater than Splitter$[j-1]$ and less than Splitter$[j]$, as well as $\mathrm{Frac}_{\mathrm{L}}[j]$ and $\mathrm{Frac}_{\mathrm{R}}[j]$ of the total number of elements in the local array with the same value as Splitter$[j-1]$ and Splitter$[j]$, respectively.

- **Step (7):** Each node N_i routes the subsequence associated with Splitter$[j]$ to processor N_j, for $(1 \le i, j \le N)$. Since with high probability no sequence will contain more than $c_2 \frac{n}{N^2}$ elements, where c_2 is a constant to be defined later, this is equivalent to performing a **transpose** operation with block size $c_2 \frac{n}{N^2}$.
- **Step (8):** Using p threads, each node N_i merges the N sorted subsequences received in **Step (7)** to produce the i^{th} column of the sorted array. Note that, with high probability, no node has received more than $\alpha_2 \frac{n}{N}$ elements, where α_2 is a constant to be defined later.

Recalling that we established in [10] that with high probability $c_1 \ge 2$, $\alpha_2 \ge 2.62$, and $c_2 \ge 5.42$, the analysis of our sample sort algorithm is as follows. The randomization in **Step (1)** is easily done with multiple threads by having each of the p processors at a node simultaneously assign $\frac{n}{Np}$ elements, requiring at most $O\left(\frac{n}{Np} + N\epsilon + \frac{n}{N\kappa}\right)$ time. The sorting at each node in **Step (3)** can be done using p processors with our SMP sorting routine in approximately $O\left(\frac{n}{Np}\log\left(\frac{n}{Nm}\right) + \frac{\log\left(\frac{n}{Npm}\right)}{\log(z)}\frac{n}{N\kappa}\right)$ time for *integers* and $O\left(\frac{n}{Np}\log\left(\frac{n}{N}\right) + \frac{\log\left(\frac{n}{Npm}\right)}{\log(z)}\frac{n}{N\kappa}\right)$ time for doubles. **Steps (4)** and **(6)**are both done using a single thread and both require $O\left(p\log\left(\frac{n}{N}\right) + p\log\left(\frac{n}{N}\right)(\epsilon + 1/\kappa)\right)$ time. **Step (8)** requires that we merge N sorted sequences. This can be easily done using multiple threads by following the same scheme used in **Steps(2)** through **(5)** of our algorithm for sorting on a single SMP, requiring at most $O\left(\frac{n}{Np}\log N + N\epsilon + \frac{n}{N\kappa}\right)$ time. Finally, **Steps (2)**, **(5)**, and **(7)** call the communication primitives **transpose**, **bcast**, and **transpose**, respectively. The analysis of these primitives in [6] shows that with high probability these three steps require $T_{comm}(n,p) \le \left(\tau + \frac{2n(N-1)}{N^2\beta}\right)$, $T_{comm}(n,p) \le \left(\tau + \frac{2(N-1)}{\beta}\right)$, and $T_{comm}(n,p) \le \left(\tau + \frac{5.24n(N-1)}{N^2\beta}\right)$, respectively. Hence, taking note of the modest size of ϵ and its small coefficients in all these steps, with high probability the overall complexity of our sample sort algorithm is given (for floating point numbers) by

$$\begin{aligned} T(n,p) &\approx T_{comp}(n,N,p) + T_{comm}(n,N,p) \\ &= O\left(\frac{n}{Np}\log n + \frac{\log\left(\frac{n}{Npm}\right)}{\log(z)}\frac{n}{N\kappa} + \tau + \frac{n}{N\beta}\right) \end{aligned} \tag{3}$$

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

Note that the analysis indicates that the bus bandwidth is a more serious limiting factor than the cluster interconnect bandwidth, especially in view of the fact that the bus bandwidth is not scalable like the cluster interconnect bandwidth.

4 Performance Evaluation

Our algorithms were implemented using Posix Threads [9] and run on a DEC Alpha Cluster. Our DEC Alpha cluster consists of 10 AlphaServer 2100A systems, each of which holds 4 Alpha 21064A processors running each at 275 MHz. Each Alpha 21064A processor has a 16KB primary data cache and a 4MB secondary data cache. The AlphaServers are connected using the Digital Gigaswitch/ATM and OC-3c adapter cards, which have a peak bandwidth rating of 155.52 Mbps. Internode communication is effected by calls to the SIMPLE collective communication primitives [7].

4.1 Sorting Benchmarks

We tested our code on a variety of benchmarks, each of which had both a 32-bit *integer* version and a 64-bit double precision floating point number (*double*) version. 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 $\mathrm{MAX_D}$, 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 $(2^{-30} \times \mathrm{MAX_D})$.

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 $\left(\left(\left((j-1)g+\frac{p}{2}-1\right) \bmod p\right)+1\right)\frac{2^{31}}{p}$ and $\left(\left(\left(\left((j-1)g+\frac{p}{2}\right) \bmod p\right)+1\right)\frac{2^{31}}{p}-1\right)$, the second $\frac{n}{pg}$ elements at each processor to be random numbers between $\left(\left(\left((j-1)g+\frac{p}{2}\right) \bmod p\right)+1\right)\frac{2^{31}}{p}$ and $\left(\left(\left(\left((j-1)g+\frac{p}{2}+1\right) \bmod p\right)+1\right)\frac{2^{31}}{p}-1\right)$, 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 [10] for a detailed justification of these benchmarks.

4.2 Experimental Results for a Single SMP

We begin by experimentally determining the optimal values of the parameters m and z, where m is the block size in **Step (1A)** and z is the z-way merge used in **Step (1B)**. **Table I** displays the times required to sort 4M *doubles* using four threads as a function of m and z. Clearly, performance suffers dramatically when the block size reaches 4MB (512K eight byte double precision numbers), which is the limit of the secondary cache. This is expected, since sorting a block in **Step (1A)** now requires that data be repeatedly swapped to main memory. It is more difficult to explain why the optimal values for m and z were 32K and 32, respectively, since this block size exceeds the 16KB primary cache but falls well short of the secondary cache size. Part of the explanation might lie in the fact that the overall complexity of $O\left(\frac{n}{p}\log n + \frac{\log\left(\frac{n}{pm}\right)}{\log(z)}\frac{n}{\kappa}\right)$ obscures the fact that the complexity of the block sort in **Step (1a)** is $O\left(\frac{n}{p}\log m + \epsilon + \frac{n}{\kappa}\right)$, which is an increasing function of m. The value of $z = 32$ is reasonable when we note than the actual cache line size is 32 bytes. Our tree of losers is implemented with z 12-byte records, so the entire process could easily take place in the 16KB primary cache.

For the remaining discussion, the times reported are for the experimentally optimal values of m and z and for the number of samples $s = 8$. The relative performance of the algorithm on different benchmarks seems to confirm that this was a reasonable choice for s.

Table I also provides an interesting illustration in the importance of minimizing secondary memory access. If we consider the data for a block size of 2K, the execution time drops as we move from $z = 2$ to $z = 4$ to $z = 8$. This is reasonable since we require 9 rounds of 2-way merge, 5 rounds of 4-way merge, and only 3 rounds of 8-way merge, and each round of z-way merge is obviously another round where all the input elements must be brought in from main memory. Moving from $z = 8$ to $z = 16$ has little effect on the execution time since it does nothing to reduce the memory requirements, but moving to $z = 32$ saves a round of memory access and, hence, the execution time is further reduced. However, the most dramatic illustration of the importance of minimizing secondary memory access can be found by comparing the optimal sorting time of 3.41 seconds for $m = 32K$ and $z = 32$ with the time of 9.86 seconds required to sort using only binary merge sort. Reducing memory access by a combination of block sorting and z-way merging improved performance nearly threefold.

Block	**Denomination of z-Way Merge**								
Size	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
2K	7.24	5.50	4.37	4.24	3.83	3.80	3.98	4.08	3.64
4K	6.57	5.13	4.45	3.91	3.68	3.91	3.86	3.53	
8K	6.25	4.89	4.24	3.76	3.77	3.92	3.43		
16K	5.73	4.53	3.99	3.92	3.86	3.51			
32K	5.41	4.65	3.97	3.99	3.41				
64K	5.14	4.29	4.29	3.78					
128K	5.20	4.81	4.41						
256K	5.98	5.56							
512K	7.68								
1M	9.86 - (No z-way merge is necessary for this block size)								

Table I: Time (in seconds) required to sort 4M *doubles* using four threads as a function of M and z.

Tables II and **III** display the performance of our sorting algorithm as a function of input distribution for a variety of input sizes. Notice that the he performance is essentially similar for benchmarks **[U]**, **[G]**, and **[WR]** and for benchmarks **[Z]**, **[DD]**, and **[RD]**. The reason why the benchmarks in the second group ran significantly faster than the benchmarks in the first group is that the second group of benchmarks all contained values restricted to a very small range ($\{0\}$,$\{0,1,2\}$, and $\{0,1,...,32\}$) which results in significant savings in time required for sorting and merging. Because there was little difference in the performance of the benchmarks in the first group, the remainder of this section will only discuss the performance of our sorting algorithm on the single benchmark **[U]**.

Input Size	**Benchmark**					
	[U]	[G]	[Z]	[WR]	[DD]	[RD]
1M	0.428	0.430	0.319	0.405	0.274	0.311
2M	0.973	0.930	0.586	0.922	0.536	0.597
4M	1.96	1.97	1.25	1.86	1.23	1.30
8M	4.06	4.05	2.66	3.81	2.68	2.74

Table II: Sorting *integers* (in seconds) using 4 threads on a DEC AlphaServer 21000A.

Input Size	**Benchmark**					
	[U]	[G]	[Z]	[WR]	[DD]	[RD]
512K	0.397	0.394	0.320	0.421	0.337	0.348
1M	0.868	0.856	0.741	0.844	0.724	0.710
2M	1.64	1.72	1.39	1.73	1.40	1.51
4M	3.50	3.47	3.00	3.52	3.01	2.98

Table III: Sorting *doubles* (in seconds) using 4 threads on a DEC AlphaServer 21000A.

The results in **Figure 1** examine the scalability of our sorting algorithm as a function of the number of threads. Results are shown for sorting both *integers* and *doubles*. Bearing in mind that these graphs are log-log plots, they show that for a *fixed input size* n the execution time nearly halves when the number of threads p are doubled. The graphs in **Figure 2** examine the scalability of our sorting algorithm as a function of problem size, for differing numbers of threads. They show that for a *fixed number of processors* there is an almost linear dependence between the execution time and the total number of elements n. This might at first appear to exceed our prediction of a $O(n \log n)$ relationship between n and the computational complexity. 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.

4.3 Experimental Results for a Cluster of SMPs

For each experiment, the input is evenly distributed amongst the nodes. The output consists of the elements in non-descending order arranged amongst the nodes so that the elements at each node are

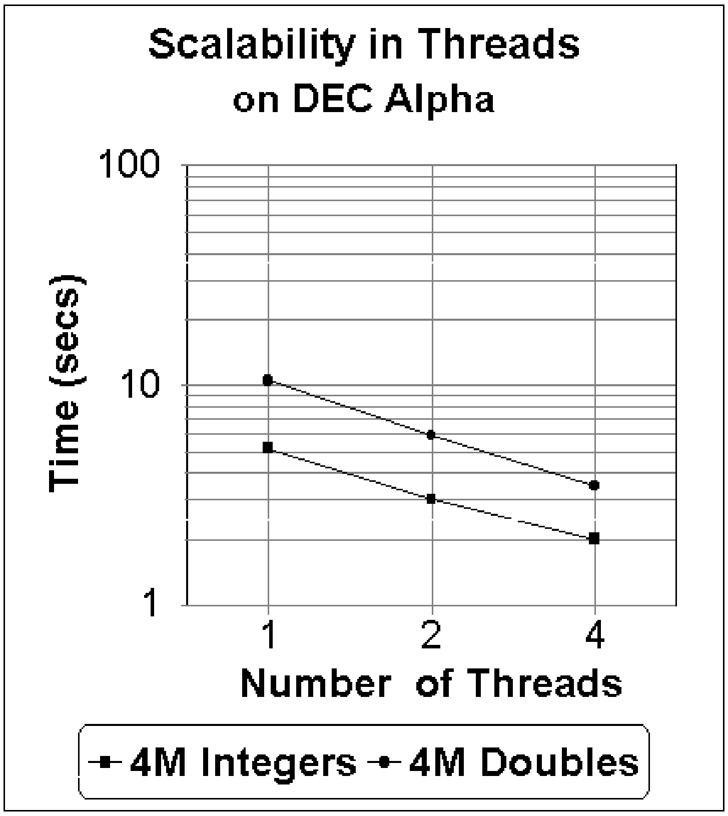


Figure 1: Scalability of sorting *integers* and *doubles* with respect to the number of threads.

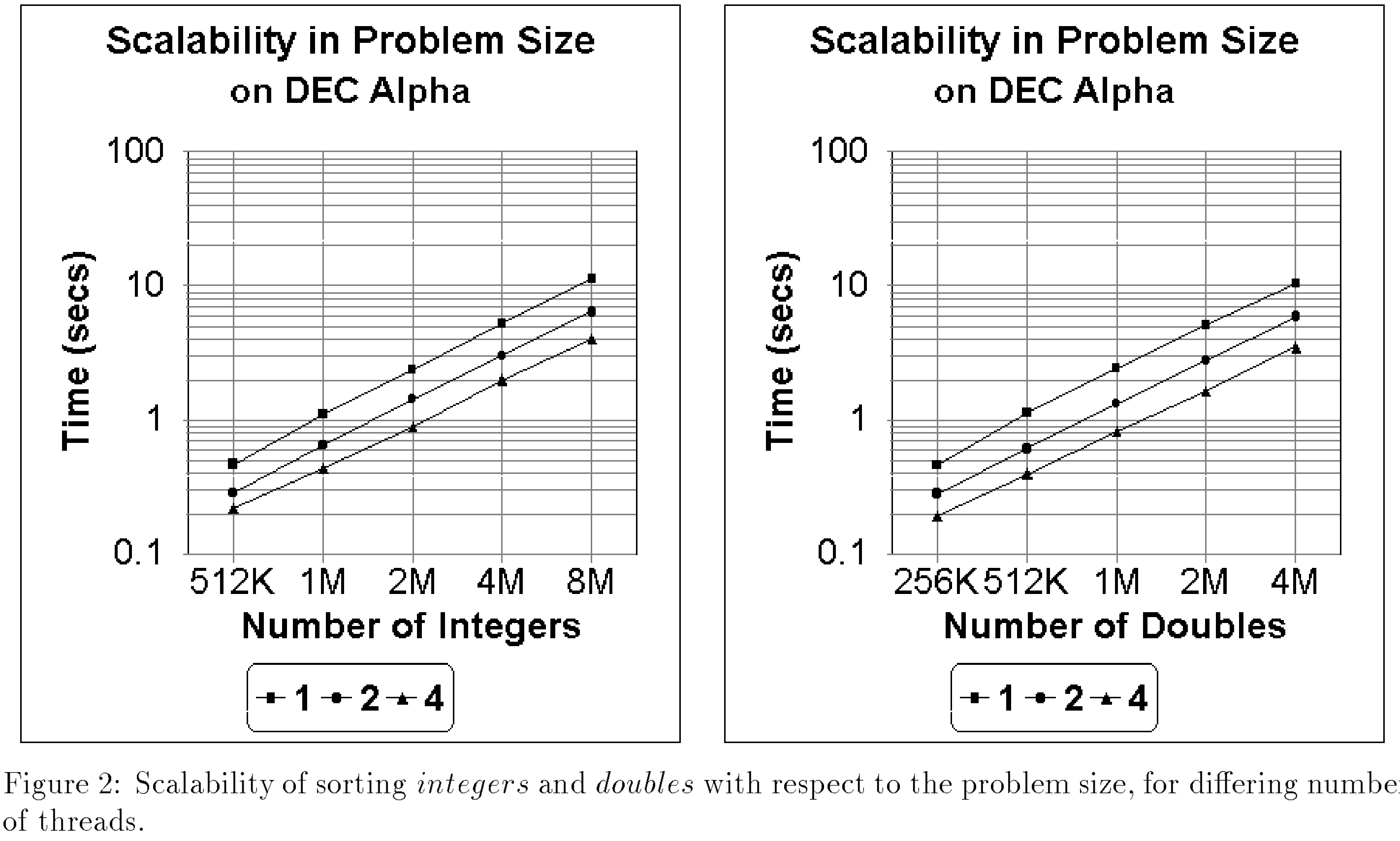


Figure 2: Scalability of sorting *integers* and *doubles* with respect to the problem size, for differing numbers of threads.

in sorted order and no element at node N_i is greater than any element at processor N_j, for all $i < j$. Note that in all cases *the results shown for a single node were obtained using the sorting algorithm for a single SMP.*

Tables IV and **V** display the performance of our sorting algorithm as a function of input distribution for a variety of input sizes. In each case, the performance is essentially independent of the input distribution. Because of this independence, the remainder of this section will only discuss the performance of our sorting algorithm on the single benchmark **[U]**.

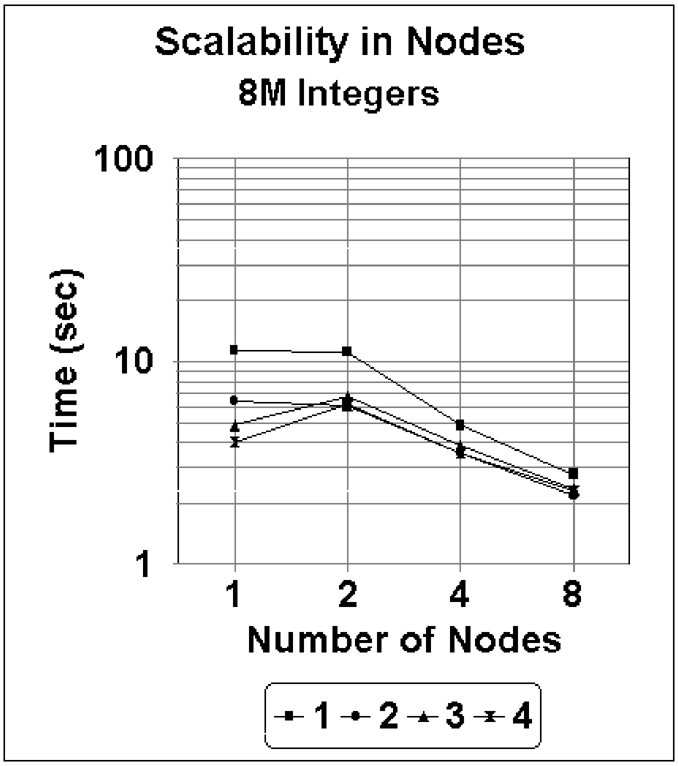
Input Size	Benchmark								
	[U]	[G]	[2-G]	[B]	[S]	[Z]	[WR]	[DD]	[RD]
8M	2.28	2.31	2.23	2.21	2.36	2.11	2.27	2.16	2.13
16M	4.36	4.31	4.42	4.41	4.33	4.17	4.31	4.09	4.08
32M	9.22	9.53	9.26	9.17	9.31	8.93	9.36	8.99	8.98
64M	16.49	17.01	16.86	17.21	17.09	16.07	16.79	16.26	16.03

Table IV: Total execution time (in seconds) required to sort a variety of *integer* benchmarks on an 8 node cluster using 4 threads.

Input Size	Benchmark								
	[U]	[G]	[2-G]	[B]	[S]	[Z]	[WR]	[DD]	[RD]
4M	2.76	2.79	2.72	2.74	2.73	2.61	2.81	2.64	2.60
8M	4.89	4.86	4.80	4.76	4.85	4.57	4.80	4.61	4.54
16M	9.36	9.54	9.30	9.19	9.28	8.94	9.25	9.01	8.90
32M	18.71	19.31	18.68	18.27	18.54	18.16	18.98	18.23	18.31

Table V: Total execution time (in seconds) required to sort a variety of *double* benchmarks on an 8 node cluster using 4 threads.

The results in **Figure 3** examine the scalability of our sorting algorithm as a function of the number of nodes, for a variety of threads. Results are shown for sorting both *integers* and *doubles*. To understand these results, consider the step by step breakdown of the execution times shown in **Table VI** for sorting 8M *integers* with both 1 and 4 threads. Moving from one node to two introduces the overhead of **Steps 1-2** and **4-8**, which together account for approximately 35% and 50% of the total execution time on 1 and 4 threads, respectively. This consumes the majority of the time we could hope to save by sharing the work of sorting amongst two nodes. The effect is more pronounced for multiple threads because as our model predicts internode communication is independent of the number of threads. The effect of this overhead would be even more pronounced were it not for the fact that the time required for **Step 3** for both 1 and 4 nodes is considerably higher than we would expect from sorting 4M *integers* on a single node. But moving between 1 and 4 nodes and 1 and 8 nodes, the time required for **Step (3)** scales inversely with the number of nodes, which is the expectation of our model. The failure of communication in Steps 2 and **7** to scale inversely with the number of nodes might at first appear surprising. However, this performance is actually quite reasonable if we recall that for 2, 4, and 8 nodes, each node has to send approximately 2M, 1.5M, and 0.875M *integers* across the network, respectively. The clear implication of the results shown here in **Figure 3** and **Table VI** is that an algorithm must be both efficient and scalable to justify the use of multiple nodes.



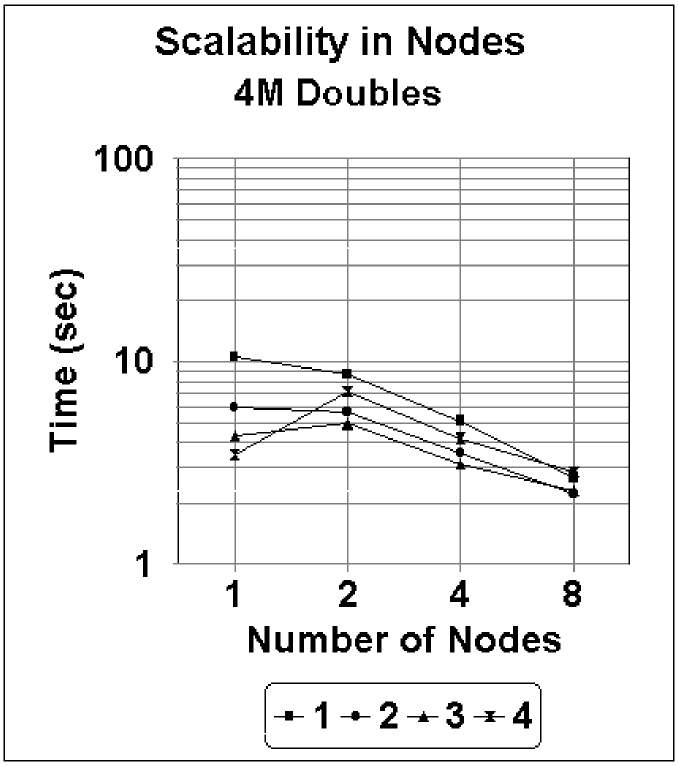


Figure 3: Scalability of sorting *integers* and *doubles* with respect to the number of nodes, for differing numbers of threads.

	One Thread				Four Threads			
Step(s)	**1**	**2**	**4**	**8**	**1**	**2**	**4**	**8**
1	0.00	0.87	0.41	0.22	0.00	0.56	0.26	0.11
2	0.00	1.28	0.92	0.56	0.00	1.11	0.88	0.58
3	11.19	7.17	2.39	1.17	3.99	3.11	0.87	0.73
4-6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
7	0.00	1.24	0.84	0.49	0.00	1.08	1.39	0.62
8	0.00	0.49	0.27	0.32	0.00	0.23	0.13	0.25
Total	11.19	11.05	4.83	2.76	3.99	5.09	3.53	2.29

Table VI: Time required for each step of sorting 8M *integers* with respect to the number of nodes using 1 and 4 threads.

The graphs in **Figure 4** examine the scalability of our sorting algorithm as a function of problem size, for differing numbers of nodes and for 1 and 4 threads. For one thread, they show that for a *fixed number of nodes* there is an almost linear dependence between the execution time and the total number of elements n. This agrees with our prediction of a $O(n \log n)$ relationship between n and the computational complexity since for the range of values shown $\log n$ differs by only a factor of 1.2.

The results for 4 threads in **Figure 4** are seemingly more problematic. To understand these results, consider the step by step breakdown of the execution times shown in **Table VII** for sorting varying numbers of integers using 8 nodes and 4 threads. Clearly, the communication costs of **Steps (2)** and **(7)** dominate the overall execution time. Strangely, increasing the problem size from 1M to 2M actually decreases the communication costs, after which the communication costs scale relatively linearly with the problem size n as expected. This immediately suggests that there is some sort of change in the communication protocol with packets larger than 64K bytes. The exact same behavior is seen in **Table VIII**, which shows the the step by step breakdown of the execution times for varying numbers of nodes for 1M *integers* and 4 threads. Increasing the number of nodes from 4 to 8 while holding the problem size constant decreases the packet size from 256K to 64K and creates an unexpected spike in the time required for communication. Finally, if we vary the number of threads while using 8 nodes to sort 1M integers, the exact same abnormality is encountered. At first, this would seem unexpected, since according to our model communication costs should not depend on the the number of threads. However, in our implementation of a transpose, each node exchanges p roughly equal size packets, one for each thread. Thus, it would seem that increasing the packet size above 16384 bytes causes a change in the communication protocol. Once the protocol is switched, the relative costs of communication decline and the execution time scales with problem size as our model anticipates.

	Problem Size			
Step(s)	**1M**	**2M**	4M	**8M**
1	0.01	0.03	0.06	0.11
2	0.47	0.26	0.38	0.58
3	0.07	0.12	0.36	0.73
4-6	0.00	0.00	0.00	0.00
7	0.51	0.35	0.45	0.62
8	0.18	0.16	0.12	0.25
Total	1.27	0.96	1.37	2.29

Table VII: Time required for each step of sorting *integers* on the DEC Alpha Cluster using 8 nodes and 4 threads.

Step(s)	Number of Nodes			
	1	**2**	**4**	**8**
1	0.00	0.06	0.03	0.01
2	0.00	0.14	0.13	0.47
3	0.43	0.30	0.14	0.07
4-6	0.00	0.00	0.00	0.00
7	0.00	0.15	0.13	0.51
8	0.00	0.03	0.05	0.18
Total	0.48	0.69	0.49	1.27

Table VIII: Time required for each step of sorting 1M *integers* on the DEC Alpha Cluster using 4 threads.

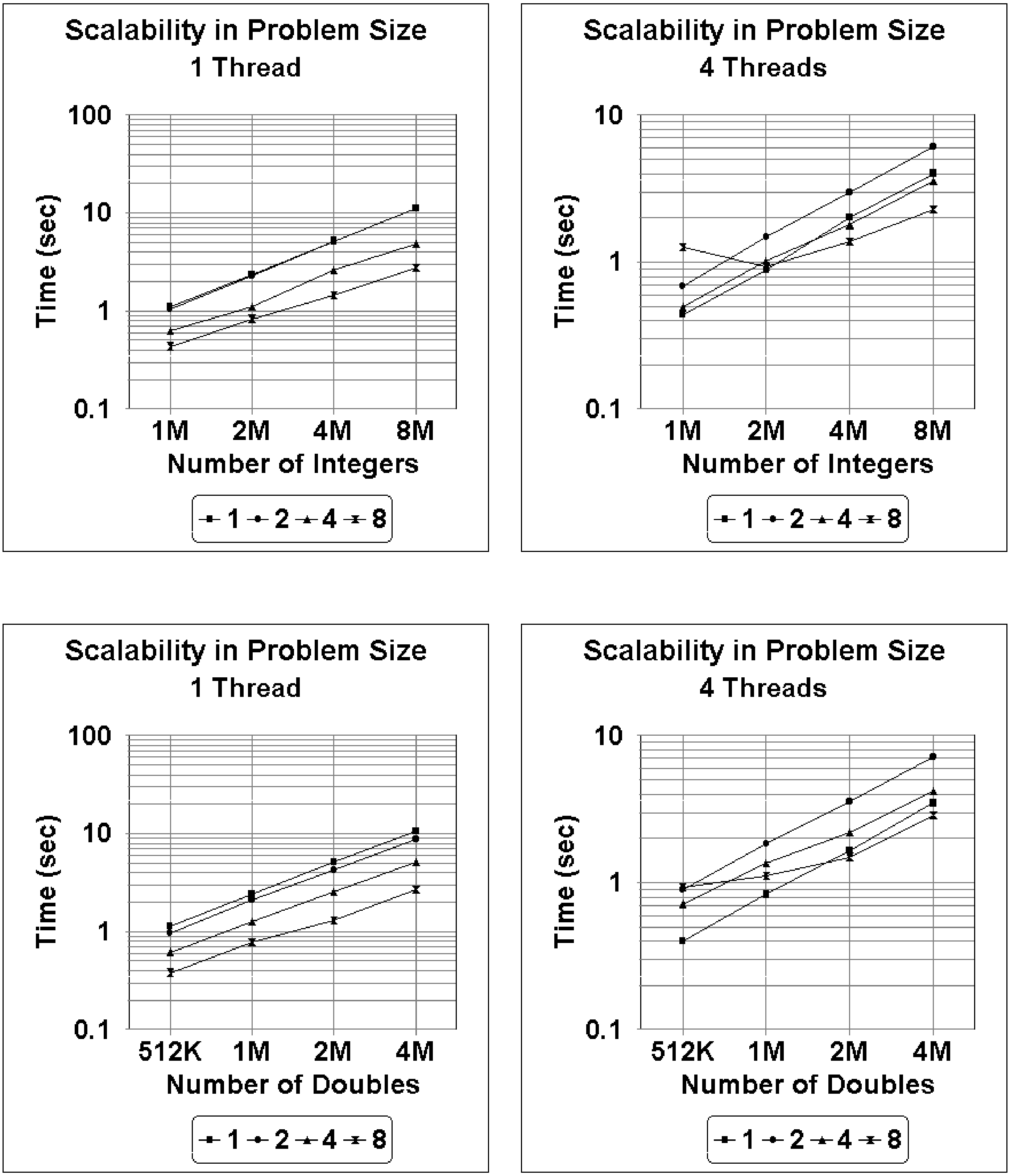


Figure 4: Scalability of sorting *integers* and *doubles* with respect to the problem size, for differing numbers of nodes and threads.

References

[1] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A Model for Heirarchical Memory. In *Proceedings of the 19th Annual ACM Symposium of Theory of Computing*, pages 305–314, May 1987.

[2] A. Aggarwal, A. Chandra, and M. Snir. Heirarchical Memory with Block Transfer. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, October 1987.

[3] A. Aggarwal and G. Plaxton. Optimal Parallel Sorting in Multi-Level Storage. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 659–668, 1994.

[4] A. Aggarwal and J. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31:1116–1127, 1988.

[5] B. Alpern, L. Carter, E. Feig, and T. Selker. The Uniform Memory Hierarchy Model of Compuatation. *Algorithmica*, 12:72–109, 1994.

[6] D.A. Bader and J. JáJá. Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection. Technical Report CS-TR-3494 and UMIACS-TR-95-74, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, July 1995. Presented at the 10th *International Parallel Processing Symposium*, pages 292-301, Honolulu, HI, April 15-19, 1996.

[7] D.A. Bader and J. JáJá. SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors. CS-TR-3798 and UMIACS-TR-97-48 Technical Report, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, May 1997.

[8] R. Barve, E. Grove, and J. Vitter. Simple Randomized Mergesort on Parallel Disks. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 109–118, Padua, Italy, June 1996.

[9] 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.

[10] D.R. Helman, D.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. Submitted to jpdc.

[11] D.R. Helman, D.A. Bader, and J. JáJá. Parallel Algorithms for Personalized Communication and Sorting With an Experimental Study. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 211–220, Padua, Italy, June 1996.

[12] D.R. Helman, J. JáJá, and D.A. Bader. A New Deterministic Parallel Sorting Algorithm With an Experimental Evaluation. Technical Report CS-TR-3670 and UMIACS-TR-96-54, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, August 1996. Submitted to jea.

[13] M. Nodine and J. Vitter. Large-Scale Sorting in Parallel Memories. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 29–39, Newport, RI, June 1991.

[14] M. Nodine and J. Vitter. Deterministic Distribution Sort in Shared and Distributed Memory Multiprocessors. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, Velen, Germany, June 1993.

[15] P. Varman, B. Iyer, D. Haderle, and S. Dunn. Parallel Merging: Algorithm and Implementation Results. *Parallel Computing*, 15:165–177, 1990.

[16] P. Varman, B. Iyer, and S. scheufler. A Multiprocessor Algorithm for Merging Multiple Sorted Lists. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 22–26.

[17] J. Vitter and E. Shriver. Algorithms for Parallel Memory I: Two-Level Memories. *Algorithmica*, 12:110–147, 1994.