Sorting on Clusters of SMPs [*]
(Extended Abstract)

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

Abstract

We introduce an efficient algorithm for sorting on clusters of symmetric multiprocessors (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. 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.

1 Introduction

Clusters of symmetric multiprocessors (SMPs) have emerged as a primary candidate for large scale multiprocessor systems. In spite of this trend, relatively little work has been done to develop techniques for designing algorithms that 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 the overhead incurred by simultaneous main memory access. On the other hand, from the perspective of the cluster, each node is in effect a super-processor, 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 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 [8, 7, 10]. 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 the algorithms described in [3, 11, 12, 2, 6, 13]. Unfortunately, none of these algorithms by itself 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 our abstract is as follows. **Section 2** presents our computational model for analyzing algorithms for SMP clusters. **Section 3** describes 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. A number of models have already been proposed to capture the performance of multilevel hierarchical memories, including the D-disk model of Vitter and Shriver [14], the Hierarchical Memory with Block Transfer Model of Aggarwal et al. [1], and the Uniform Memory Hierarchy Model of Alpern et al. [4]. However, we believe these models are unnecessarily complicated for use with clusters of SMPs and would require 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 the overhead incurred by 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 complementary models.

More specifically, in our SMP model, 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 possibly distinct blocks in main memory, 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 [8, 7, 10]. 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 $T_{comm}(n, N)$. See [9] for a more extensive discussion of computational models.

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

Our algorithm is an adaptation of our sorting by regular sampling [8, 10], 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 integer 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 $\left(\frac{(i-1)n}{p} + k\frac{n}{ps}\right)^{th}$ 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 pth 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 jth 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 than 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 can be shown [8, 10] that no processor will merge more than $\left(\frac{n}{p} + \frac{n}{s} - p\right)$ elements.

It is straightforward to see how this algorithm can be implemented as a stable integer sort, and [9] contains an informal proof. The analysis of our algorithm 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. It can be shown that the overall complexity can be approximated by:

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

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 any of the other known 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 [8, 10] and a randomized [8, 7] 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 [9].

- **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, for $(c_1 \geq 2)$.

- **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 $\alpha_1 \frac{n}{N}$ $(\alpha_1 \leq c_1)$ 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 $\gamma \frac{n}{N}$ $(\gamma \leq \alpha_1)$ 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 Frac$_L[j]$ and Frac$_R[j]$, 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, Frac$_L$, and Frac$_R$ 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 Frac$_L[j]$ and Frac$_R[j]$ 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 $(1 \leq i, j \leq N)$. Since with high probability no sequence will contain more than $c_2 \frac{n}{N^2}$ elements, for $(c_2 \geq 5.42)$, 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 ith column of the sorted array. Note that, with high probability, no node has received more than $\alpha_2 \frac{n}{N}$ elements, for $(\alpha_2 \geq 2.62)$.

A detailed analysis of this algorithm can be found in [9], and we merely repeat its conclusion here for convenience. With high probability, the overall complexity of our sample sort algorithm is given (for floating point numbers) by

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

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

4 Performance Evaluation

Our algorithms were implemented using POSIX threads 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 [5].

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. See [9] for a detailed description and justification of these benchmarks.

4.1. 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 1** 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.

Block Size	Denomination of z-Way Merge								
	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 1. Time (in seconds) required to sort 4M doubles using four threads as a function of M and z.

Table 1 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.

Tables 2 displays 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]** (uniform distribution).

Input	Benchmark					
Size	[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 2. Sorting integers (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 is doubled.

The graph in **Figure 2** examines 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 threads* there is an almost linear dependence between the execution time and the total number of elements n.

4.2. 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 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.*

Table 3 displays 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]** (uniform distribution).
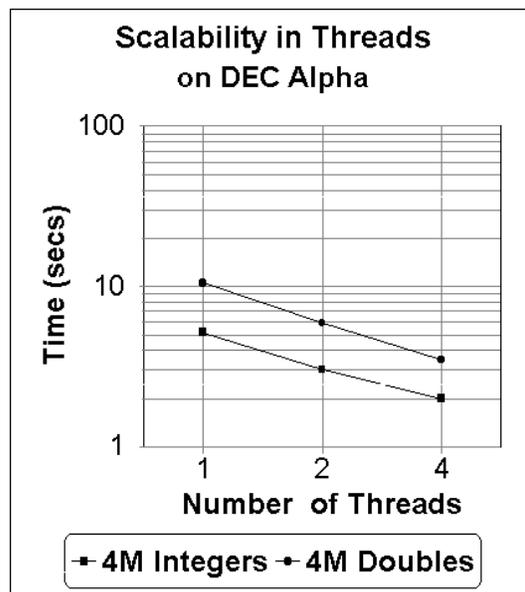

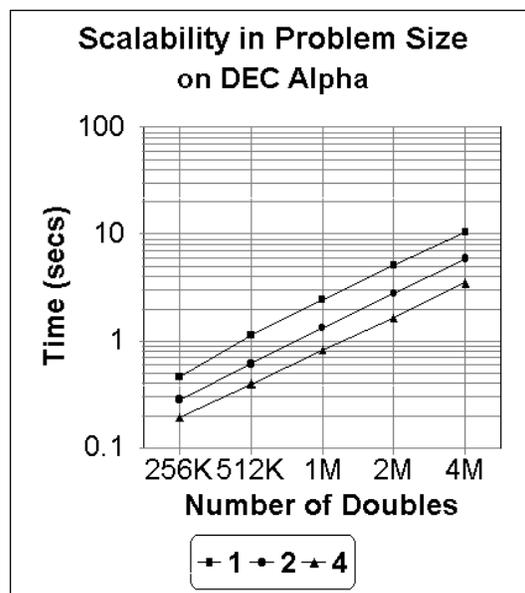**Figure 1. Scalability of sorting integers and doubles with respect to the number of threads.**

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

5

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 3. Total execution time (in seconds) required to sort a variety of doubles benchmarks on an 8 node cluster using 4 threads.

The results in **Figure 3** examines the scalability of our sorting algorithm as a function of the number of nodes, for a variety of threads. To understand these results, consider the step by step breakdown of the execution times shown in **Table 4** 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 one node with 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 these results is that an algorithm must be both efficient and scalable to justify the use of multiple nodes.

Step(s)	One Thread				Four Threads			
	1	2	4	8	1	2	4	8
Total	11.19	11.05	4.83	2.76	3.99	5.09	3.53	2.29
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

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

The graphs in **Figures 4** and **5** examine the scalability of our sorting algorithm as a function of problem size, for
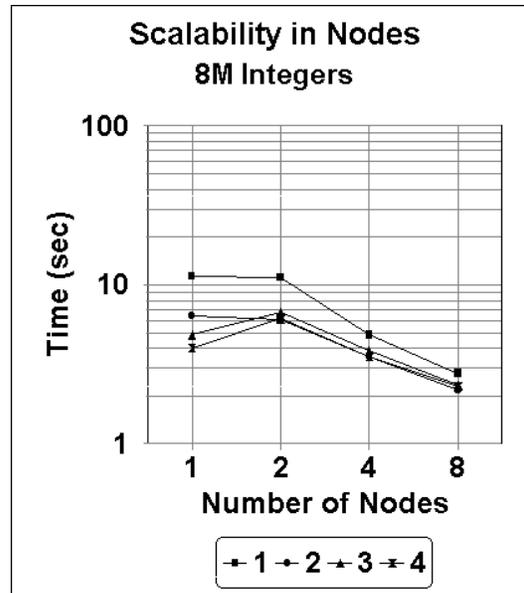

Figure 3. Scalability of sorting integers with respect to the number of nodes, for differing numbers of threads.

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. The results for 4 threads in **Figure 5** are seemingly more problematic. However, a step by step breakdown of the running times in [9] shows that the communication costs dominate the execution time and that as the problem size increases from 1M to 2M integers *the communication costs actually decrease*, presumably because of 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.

References

[1] 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.

[2] 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.

[3] A. Aggarwal and J. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31:1116–1127, 1988.
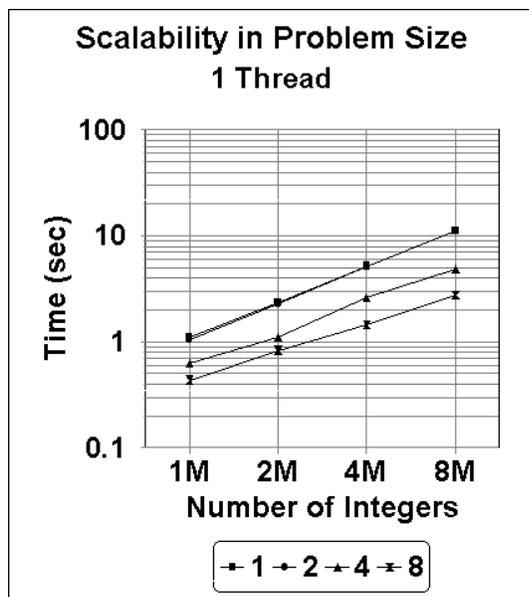
Figure 4. Scalability of sorting integers with respect to the problem size, for differing numbers of nodes and a single thread.
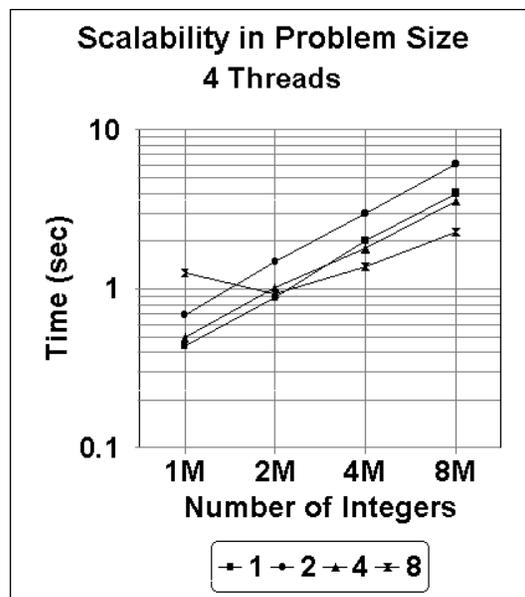

Figure 5. Scalability of sorting integers with respect to the problem size, for differing numbers of nodes and four threads.

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

[5] D.A. Bader and J. JáJá. SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors. UMIACS-TR-97-48 Technical Report, UMIACS, University of Maryland, 1997.

[6] 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.

[7] D.R. Helman, D.A. Bader, and J. JáJá. A Randomized Parallel Sorting Algorithm With an Experimental Study. Technical Report UMIACS-TR-96-53, UMIACS, University of Maryland, 1996. To appear in the *Journal of Parallel and Distributed Computing*.

[8] 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.

[9] D.R. Helman and J. JáJá. Sorting on Clusters of SMPSs. Technical Report UMIACS-TR-97-69, UMIACS, University of Maryland, 1996.

[10] D.R. Helman, J. JáJá, and D.A. Bader. A New Deterministic Parallel Sorting Algorithm With an Experimental Evaluation. Technical Report UMIACS-TR-96-54, UMIACS, University of Maryland, 1996.

[11] 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.

[12] 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.

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

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