High-Performance Sorting on Networks of Workstations

Andrea C. Arpaci-Dusseau
Computer Science Division
University of California, Berkeley
dusseau@cs.berkeley.edu

Remzi H. Arpaci-Dusseau
Computer Science Division
University of California, Berkeley
remzi@cs.berkeley.edu

David E. Culler
Computer Science Division
University of California, Berkeley
culler@cs.berkeley.edu

Joseph M. Hellerstein
Computer Science Division
University of California, Berkeley
jmh@cs.berkeley.edu

David A. Patterson
Computer Science Division
University of California, Berkeley
patterson@cs.berkeley.edu

Abstract

We report the performance of NOW-Sort, a collection of sorting implementations on a Network of Workstations (NOW). We find that parallel sorting on a NOW is competitive to sorting on the large-scale SMPs that have traditionally held the performance records. On a 64-node cluster, we sort 6.0 GB in just under one minute, while a 32-node cluster finishes the Datamation benchmark in 2.41 seconds.

Our implementations can be applied to a variety of disk, memory, and processor configurations; we highlight salient issues for tuning each component of the system. We evaluate the use of commodity operating systems and hardware for parallel sorting. We find existing OS primitives for memory management and file access adequate. Due to aggregate communication and disk bandwidth requirements, the bottleneck of our system is the workstation I/O bus.

1 Introduction

The past few years have seen dramatic improvements in the speed of sorting algorithms, largely due to increased attention to issues of computer architecture. However, the best sorting results to date have been produced by industrial researchers working on expensive, well-endowed versions of shared-memory parallel computers (SMPs) produced by their parent companies [26, 30]. In this paper we describe how we achieved new records in sorting performance using a relatively modest "shared-nothing" network of general-purpose UNIX workstations. We set a new MinuteSort record of 6.0 GB (*i.e.*, 6×2^{30} bytes) on a 64-node cluster of UltraSPARCs, and a new Datamation Benchmark record of 2.41 seconds on a 32-node cluster.

Our use of a Network of Workstations (NOW) for sorting has a number of benefits. First, NOWs provide a high-degree of *performance isolation*: that is, they allow analysis of behavior on a node-by-node, factor-by-factor basis. By contrast, in an SMP, resources are pooled together, and hence it is more difficult to achieve an equivalent fine-grained analysis. For example, by tuning single-processor performance carefully, we are able to deliver roughly 431 MB/s of disk bandwidth to the 64-processor sorting application. Second, NOWs provide *incremental scalability* of hardware resources. Because additional workstations can always be added to a NOW, well-tuned programs can be easily scaled to large configurations. By contrast, most SMPs have a hard limit on scalability imposed by the size of their box, and are more expensive and complex to scale within that limit.

In this paper we note some lessons for the NOW research community, which is made up largely of researchers in operating systems and computer architecture [2, 3, 5, 9, 22]. In principle NOWs are similar to "shared-nothing" architectures for databases [4, 11, 14, 18, 20, 29, 31, 32] but they have typically been analyzed and tuned in the context of compute-intensive applications. We demonstrate that NOWs can be a state-of-the-art platform for data-intensive applications as well.

The ease of assembling different configurations in a NOW motivated us to investigate a family of solutions for sorting, rather than a single algorithm tuned to a particular machine. Our investigation exposes a range of options for overlapping steps of the sorting algorithm and their implications. In particular, we develop tools to characterize existing hardware, explore the relationship of parallelism and bandwidth constraints, and characterize a number of tradeoffs between pipelining and associated costs in memory utilization.

An additional contribution of this work is to evaluate commodity software and hardware for sorting and, by extension, for other database applications. We find that threads and the current interface for memory management in modern UNIX operating systems do indeed help in developing efficient implementations; this is a cheering update on some of the well-known criticisms of earlier versions of UNIX for database applications [28]. On the other hand, we demonstrate that some important facilities are missing in modern workstations, such as handling data striped across heterogeneous disks and determining available memory. Regarding machine architecture, we find that our clustered UltraSPARC I workstations are limited by insufficient I/O bus bandwidth, which, if left unimproved, would prevent data-intensive applications from enjoying future gains in CPU, memory, network, or disk speed.

The organization of this paper matches the development of our sorting implementations. After briefly reviewing related work in Section 2 and our experimental platform in Section 3, we cover our four versions of NOW-Sort in the next four sections. Figure 1 depicts the derivation of our sorting algorithms,

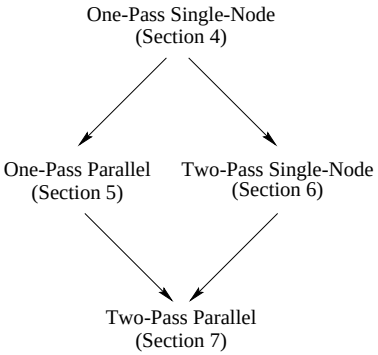

Figure 1: **Development of Sorting Algorithm**. *The sections of the paper follow the same development as the progression of sorting algorithms, from the simplest to our most complex.*

where more complex versions build from the simpler ones. We will show that by first understanding and tuning application performance in simple configurations, we are able to build highly-scalable parallel versions with little effort. Our best Datamation benchmark time is given for the one-pass parallel version in Section 5, and the best MinuteSort results for the two-pass parallel sort in Section 7. We present our conclusions in Section 8.

2 Related Work

The Datamation sorting benchmark was introduced in 1985 by a group of database experts as a test of a processor's I/O subsystem and operating system [17]. The performance metric of this benchmark is the elapsed time to sort one million records from disk to disk. The records begin on disk and are each 100-bytes, where the first 10-bytes are the key. Thus, with one million 100-byte records, 95 MB of data are read and written from disk. The elapsed time includes the time to launch the application, open, create, and close all files, ensure the output resides on disk, and to terminate the program. Price-performance of the hardware and software is computed by pro-rating the five-year cost over the time of the sort. The previous record-holder on this benchmark was a 12 processor SGI Challenge with 96 disks and 2.25 GB of main memory [30] at 3.52 seconds; a single processor IBM RS/6000 with 8 disks and 256 MB of memory has an impressive time of 5.1 seconds and better price/performance, but uses raw disk, which is not allowed in the benchmark [1].

Recognizing that the Datamation benchmark is outdated and is more a test of startup and shutdown time than I/O performance, the authors of AlphaSort introduced MinuteSort in 1994 [26]. The key and record specifications are identical to that of Datamation; the performance metric is now the amount of data that can be sorted in one minute of elapsed time. Price-performance is calculated from the list price of the hardware and operating system depreciated over three years. The SGI system was also the previous record-holder on the MinuteSort benchmark, sorting 1.6 GB. AlphaSort achieved 1.1 GB on only three processors, 36 disks, and 1.25 GB of memory, for better price/performance [26].

Over the years, numerous authors have reported the performance of their sorting algorithms and implementations [1, 6, 7, 15, 16, 21, 25, 26, 27, 30, 35], and we try to leverage many of the implementation and algorithmic lessons that they describe. One difference between most of this work and ours is that we provide measurements for a range of system configurations, varying the number of processors, the number of disks per machine, and amount of memory. Another difference is that our environment is one of the few parallel configurations where each node is a complete system, with its own virtual memory system, disks, and file system.

3 Experimental Environment

3.1 Hardware

Two different cluster environments form our experimental testbed. The first consists of 64 commodity UltraSPARC I workstations, each with 64 MB of memory (however, most measurements only extend to 32 nodes due to time constraints). Each workstation houses two internal 5400 RPM Seagate Hawk disks on a single fast-narrow SCSI bus. Note that with only two disks per machine, we can not afford to dedicate a spare disk for paging activity.

Ultra Enterprise I Model 170 (64 MB, 1 5400 RPM Disk)	$15,495
Main Memory	$16 per MB
Internal 5400 RPM Seagate Hawk	$925
External 7200 RPM Seagate Barracuda	$1,100
Enclosure for 8 External Disks	$2,000
Fast-Wide SCSI Card	$750
Myrinet 4.1-M2F Card	$1,200
Myrinet M2F-SW8 Switch	$2,400
8-node System *(8 x 128 MB, 8 x 2 5400 RPM Disks, 8 x 2 7200 RPM Disks, 8 SCSI Cards, 4 Disk Enclosures, 8 Myrinet Cards, 1 Switch)*	$183,152
64-node System *(64 x 64 MB, 64 x 2 5400 RPM Disks, 64 Myrinet Cards, 26 Switches)*	$1,190,080

Table 1: **Hardware List Prices.** *October 1996 list prices.*

The second cluster connects eight more fully-equipped UltraSPARC I Model 170 workstations. Each contains 128 MB of main memory and an extra fast-wide SCSI card, with two 7200 RPM Seagate Barracuda external disks attached. Thus, while this cluster only contains one eighth of the processors in the other configuration, it contains one quarter of the number of disks and amount of memory.

The main lesson taught by the authors of AlphaSort [26] is that even large sorting problems should be performed in a single pass, since only half the amount of disk I/O is performed and the price of memory is relatively low. All previous record-holders on both the Datamation and MinuteSort benchmarks were able to sort the records in a single-pass. However, as we shall see, our NOW configuration is memory-starved, so we perform the MinuteSort benchmark in two passes.

In addition to the usual connection to the outside world via 10 Mb/s Ethernet, every workstation contains a single Myrinet network card. Myrinet is a switch-based, high-speed, local-area network, with links capable of bi-directional transfer rates of 160 MB/s [10]. Each Myrinet switch has eight ports, and the 64-node cluster is constructed by connecting 26 of these switches in a 3-ary tree.

3.2 Software

Each machine in our cluster runs Solaris 2.5.1, a modern, multi-threaded version of UNIX [24]. The disparate resources in the cluster are unified under GLUnix, the prototype distributed operating system for NOW [19]. GLUnix monitors nodes in the system for load-balancing, can co-schedule parallel programs, and provides full job control and I/O redirection. In our experiments, we primarily use GLUnix as a parallel program launcher.

The parallel versions of NOW-Sort are written in Split-C [12]. Split-C is a parallel extension to C that supports efficient access to a global address space on distributed memory machines.

Split-C is built on top of Active Messages [33], a communication layer designed to take advantage of the low latency and high bandwidth of switch-based networks. An Active Message is essentially a restricted, lightweight version of a remote procedure call. When a process sends an Active Message, it specifies a handler to be executed on the remote node. When the message is received, the handler executes atomically with respect to other message arrivals. Active Messages over the Myrinet has the following performance characteristics: the round-trip latency is roughly 20 μs, and the layer can sustain a uni-directional bandwidth (one node sending, another receiving) of 35 MB/s [13].

3.3 Input Key Characterization

In this study, we make a number of simplifying assumptions about the distribution of key values and the layout of records across processors and disks, allowing us to focus on the OS and architectural issues involved in sorting. Clearly, implementations used for sorting data sets in the real world would need to be more robust.

Following the precedent set by other researchers, we measure the performance of NOW-Sort only on key values with uniform distributions. This assumption has implications for our method of distributing keys into local buckets and across processing nodes. With a non-uniform distribution, we would need to modify our implementations to perform a sample sort [8, 15]. By adding an early phase where we sample the data to determine the range of keys targeted for each processor, we could ensure that each processor receives a similar amount of records; we plan on investigating this further in the future.

We also assume that the initial number of records on each workstation is equal, although the performance of our parallel implementations would not be greatly affected by small imbalances. If records are located on only a subset of the workstations, then our current read phase could only utilize the processors attached to those disks, with severe performance implications. Restructuring our algorithms to better deal with this situation is unfortunately beyond the scope of this paper.

4 One-Pass Single-Node Sorting

In this section we describe the basic one-pass version of NOW-Sort for a single workstation, *i.e.*, when all records fit in main memory after being read from disk. We discuss the impact of different interfaces for performing disk I/O and buffer management, and the in-memory sorting algorithm. As indicated in Figure 1, the one-pass single-node version forms the basis of all of our sorting algorithms. Each of the components of the single-node sort is used again in the other sorts, and so is worth understanding and tuning in detail.

At the highest level, the one-pass single-node sort contains three steps (which will be described in more detail shortly):

1. **Read:** Read the N 100-byte records from disk into main memory. Keys may be partially sorted by simultaneously distributing into $B = 2^b$ buckets.
2. **Sort:** Sort the 10-byte keys in memory. If keys have been placed in buckets, then each bucket is sorted individually with either quicksort or a partial-radix sort with cleanup.
3. **Write:** Gather and write sorted records to disk.

In an implementation, these three steps may be overlapped or kept synchronous. For example, we investigate the benefits of overlapping sorting with reading, by copying keys into buckets while reading. Since the majority of the execution time is spent in the phases performing I/O, we begin by describing our approach to reading from disk.

4.1 Reading and Writing from Disk

NOW-Sort must work well in a variety of cluster configurations: namely, differing numbers of disks and amounts of memory. In order for an application to configure itself for best performance in its environment, it must first gather relevant information, which the OS does not always provide.

4.1.1 User-Level Striping

Software does not currently exist to stripe files across multiple local disks with different speeds. To fully utilize the aggregate bandwidth of multiple disks per machine, we implemented a user-level library for file striping on top of each local Solaris file system (i.e., we are not building on top of raw disk). Similar to the approach described in [26], each striped file is characterized by a stripe definition file, which specifies the size of the base stripe in bytes, the names of the files (on different disks) in the stripe, and a multiplicative factor associated with each file/disk.

To determine the proper ratio of stripe sizes across disks, we developed `diskconf`. This tool, given a list of SCSI buses and a list of disks on those buses, creates a large data file on each of the disks; it reads first from each file independently, and then reads simultaneously from all files on the same bus. By measuring each of the achieved bandwidths and determining whether or not each bus is saturated, the tool calculates the multiplicative factor of the base stripe for each disk to achieve its maximum transfer rate. The tool performs an analogous chore for writes.

As described in Section 3, we have two disk configurations to consider: two disks on a fast-narrow SCSI bus, and an additional two disks on a fast-wide SCSI. We first verified that the striping library does not degrade the achievable bandwidth when there is only one disk. Table 2 shows the performance of the striped file system on the simple and more complex disk configurations. In the first two rows, we see that the two 5400 RPM disks saturate the fast-narrow SCSI bus. Since the fast-narrow SCSI bus has a peak bandwidth of 10 MB/s, we measure only 8.3 MB/s from two disks capable of a total of 11 MB/s. Thus, 25% of potential performance is lost due to this architectural oversight. The second two rows indicate that the fast-wide SCSI bus adequately handles the two faster disks. Finally, the last three rows show the superiority of using `diskconf` to configure the striping library, compared to naively striping with equal-sized blocks on disks of different

Seagate Disks	SCSI Bus	Read (MB/s)	Write (MB/s)
1 5400 RPM Hawk	Narrow	5.5	5.2
2 5400 RPM Hawk	Narrow	8.3	8.0
1 7200 RPM Barracuda	Wide	6.5	6.2
2 7200 RPM Barracuda	Wide	13.0	12.1
2 of each (naive striping)	Both	16.0	15.5
2 of each (with disk tool)	Both	20.5	19.1
2 of each (peak aggregate)	Both	21.3	20.1

Table 2: **Bandwidths of Disk Configurations.** *The Read and Write columns show the measured bandwidth using the striping library. The last three rows give the performance with naive striping of same-sized blocks to each disk, with the disk tool, and the peak aggregate, calculated as the sum of the maximum bandwidths seen over the two SCSI buses.*

speeds. By reading (or writing) *two* blocks of data from each slower disk on the fast-narrow SCSI, and *three* from each on the fast-wide SCSI, we achieve 20.5 MB/s or 96% of the peak aggregate bandwidth seen from the four disks.

4.1.2 Buffer Management

Depending upon the system interface used to read data from disk, the application may or may not be able to effectively control its memory usage and prevent double-buffering by the operating system. In this section, we compare three approaches to reading records from disk: `read`, `mmap`, and `mmap` with `madvise`. For demonstration purposes, we use a very simple implementation of the sorting algorithm which performs all steps sequentially and quicksorts the keys when they are in memory.

The left-most graph of Figure 2 shows that when the application uses the `read` system call to read records into memory from disk, the total sort time increases severely when more than 20 MB of records are sorted, even though 64 MB of physical memory are available. This performance degradation occurs due to thrashing in the virtual memory system. With `read`, the file system performs its own buffering, and thus the user program is unable to control the total amount of memory in use.

To avoid the double-buffering of `read` while leveraging the convenience of the file system, we investigate the `mmap` interface. Applications use memory mapped files by opening the desired file, calling `mmap` to bind the file into a memory segment of the address space, and accessing the memory region as desired. As shown in the middle graph, performance with `mmap` also degrades after about 20 MB of records, due to the page replacement policy of the virtual memory subsystem. LRU replacement throws away soon-to-be-needed sort-buffers during the read phase; again, the ensuing thrashing degrades performance.

Fortunately, `mmap` has an auxiliary system call, `madvise`, which informs the operating system of the intended access pattern for a particular region of memory. For example, one call to `madvise` notifies the kernel that a region will be accessed sequentially, thus allowing the OS to fetch ahead of the current page and to throw away pages that have already been accessed. The right-most graph of Figure 2 shows that with both `mmap` and `madvise` the sorting program has linear performance up to roughly 40 MB, when it has used all available memory.

4.1.3 Determining Available Memory

The amount of available memory on a workstation determines the number of records that can be sorted with a one-pass sort, and the number of records per run in a multi-pass sort. Some previous work has addressed the issue of adapting sorting algorithms to memory constraints at run-time [34, 36]; however, we must first know the amount of free memory available to the sorting application. Because there is no existing Solaris interface that gives an accurate estimate, we developed `memconf`.

The memory tool allocates a buffer and fetches it into main memory using one of two methods: by writing an arbitrary value to the first word on each page or by copying words from a memory-mapped, sequentially-advised input file. After prefetching, the tool again touches each page of the buffer, recording CPU utilization. If the buffer fits in main memory, then CPU utilization is high (at least 85% by our current definition), and a larger buffer is tried. If the buffer does not fit, CPU utilization is low due to paging activity, and the tool backs off to a smaller buffer. A binary search is used to refine our estimate of usable memory.

Running the basic version of `memconf` showed us that on machines with 64, 128, and 256 MB of real memory, only 47, 104, and 224 MB, respectively, are available to user applications after the operating system and daemon processes are factored out. Applications using `mmap` and `madvise`, such as NOW-Sort, have approximately 10% less available memory (42, 93, and 195 MB).

Finally, we verified that our predictions of available memory matched the number of records we were able to sort without memory thrashing. Since a sharp increase in sort time occurs when we sort just a few more records than fit in available memory, we must be conservative; thus, we scale our estimate of available memory by 95%. Our predictions are a comfortable distance from the "memory wall", while not wasting available memory.

4.2 In-Core Sorting

This section quantifies the performance of various main-memory sorting techniques. If performed correctly, the in-core sort within the disk-to-disk single-node sort comprises only a small portion of the total execution time. For example, the in-core sort consumes only 0.6 of the 5.1 seconds required to sort one million records on an IBM RS/6000 with 8 disks [1], i.e., 12% of the total time. In this section we investigate the programming complexity needed to achieve this range of performance by measuring three different in-core sorting algorithms.

Quicksort: The first in-core sort is a simple quicksort over all of the keys in memory [23]. Previous work has indicated that swapping only the key and a pointer to the full record is faster than swapping the entire 100-byte record, even though extra memory and work is required to set up the pointers [26]. Comparisons between keys begin with the most-significant word, and only examine the remaining words if the previous ones are identical.

The top line in the left-side graph of Figure 3 shows the time for the incore quicksort as a function of the number of keys. For reference, in a system with four disks, approximately 12 seconds are required to read and write one million records; thus, with quicksort, 20% of the execution time is spent in the in-core sort.

Bucket + Quicksort: The second in-core sort performs a quicksort after the keys have been distributed into buckets; keys are placed in one of $B = 2^b$ buckets based on the high-

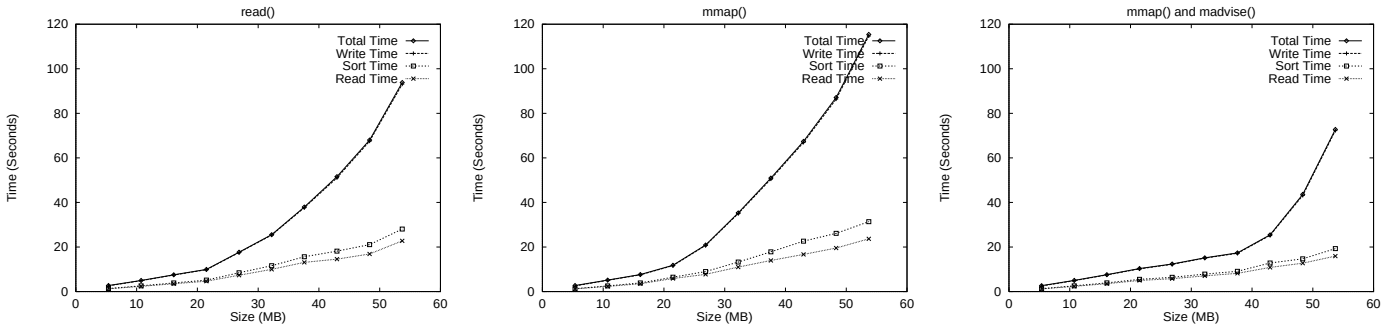

Figure 2: **Read() versus Mmap().** *All three graphs show cumulative numbers for an UltraSPARC I with 64 MB of memory and one Seagate 5400 RPM disk. The first two graphs show that when read or mmap is used, the total sort time increases super-linearly before nearing the memory limits of the machine. The right-most graph shows that with memory mapping and advising, available memory can be used more effectively.*

order b bits of the key. Placing keys into the appropriate bucket is easily overlapped with the read phase due to the simple interface provided by `mmap`. We see no degradation in read bandwidth when overlapping, since the CPU is under-utilized. In our implementation, each bucket contains the most significant 32-bits of the key after removing the top b-bits, and a pointer to the full record. In the common case, only these 32-bits of the key are examined; only when ties between keys occur are random accesses to memory necessary.

The number of buckets is determined at run-time such that the average number of keys per bucket fits into the second-level cache, which is 512 KB on the UltraSPARC. Since quicksort is performed in-place and each partial key plus pointer requires 8 bytes, there should be 64K keys per bucket. Note that this approach is highly dependent on a uniform distribution of keys; with a skewed distribution, certain buckets may contain more keys than fit in cache, degrading performance.

As shown by the middle line of Figure 3, performing quicksort on buckets that fit in the second-level cache is faster than using quicksort on all of memory; with this approach only 14% of the total time to sort one million records is spent in the in-core sort.

Bucket + Partial-Radix: The third in-core sort performs a partial-radix sort with clean-up on the keys within each bucket, as suggested in [1]. Once again, the most-significant 32-bits of the key (after removing the top b-bits) and a pointer to the full record are kept in the bucket.

Radix sort relies on the representation of keys as n-bit numbers. We perform two passes over the keys and use a radix size of 11, examining a total of 22-bits. We refer to this as a partial-radix sort since we do not radix sort on all $80 - b$ bits. On each pass over the keys, a histogram is constructed that contains the count of each of the 2^{11} digits. Then, the histogram is scanned from the lowest entry to the highest to calculate the rank of each key in the sorted-order. Finally, the keys are permuted to their destinations according to their rank in the histogram. After the partial radix sort, a clean-up phase is performed, where keys with ties in the top 22+b bits are bubble-sorted.

As with quicksort, the number of buckets is selected such that the average number of keys per bucket fits in the second-level cache. However, since radix sort requires a source and a destination buffer, only half as many keys fit in cache as compared to quicksort, i.e., 32K keys.

The bottom line in the left-side graph of Figure 3 shows that radix sort on each bucket is greatly superior to quicksort. Only

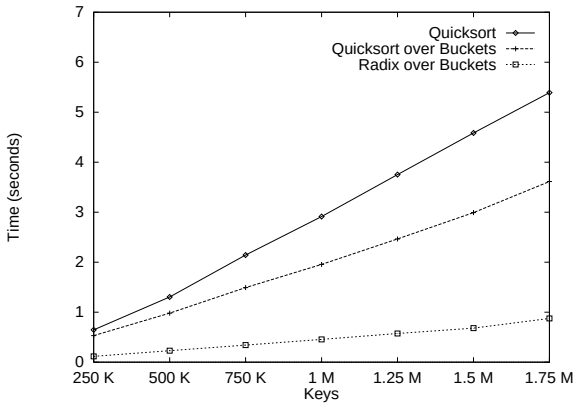

Figure 3: **Comparison of In-Core Sorting Algorithms**. *The graph compares the sorting time using quicksort over all keys in memory, quicksort independently on each bucket, and a radix sort on each bucket. The graph is not cumulative.*

4% of the total time to sort one million records is now spent in the in-core sort. Clearly, the combination of distributing keys into buckets and performing a radix sort on the keys within each bucket is worth the added complexity. We should note that we implemented some of the optimizations found in [1], including restricting the number of buckets to be less than the number of TLB entries, overlapping of sorting and writing, and a small optimization on the clean-up phase; however, we did not see a significant gain from these methods.

4.3 Discussion

To summarize, Figure 4 shows the total time as returned by the UNIX `time` command for the one-pass single-node version of NOW-Sort on systems with two and four disks. Before timing each run, we flush the file cache by unmounting and re-mounting each of the striped disks. As expected, the in-core sorting time is negligible in both configurations compared to the time for reading and writing the records from disk. Furthermore, the read bandwidth is roughly balanced with the write bandwidth and matches the transfer times found with the disk configuration tool within 5-10%. Finally, the performance is linear in the number of records when the records fit in main memory.

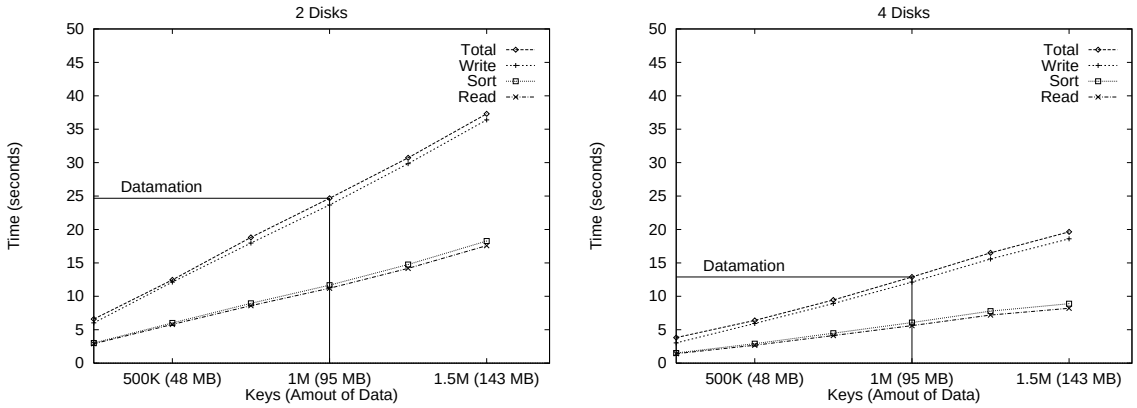

Figure 4: **Single-Node, Single-Pass Sort Time**. *Both graphs are cumulative. The time for the sort is shown as a function of the number of records. The graph on the left depicts a system using two disks connected to a fast-narrow SCSI bus; the graph on the right is for a system that uses an additional two disks on a fast-wide SCSI. Both systems contain 256 MB of real memory.*

Optimizing the single-node, single-pass sort has shed light on a number of machine architecture and operating system issues. First, our disk configuration tool has shown that the UltraSPARC internal fast-narrow SCSI bus saturates with more than one 5400 RPM disk. A fast-wide SCSI bus would have enabled us to achieve 11 MB/s from our two internal disks instead of 8.3 MB/s, a 25% loss in performance. We also found that striping to disks with different bandwidths requires different stripe sizes across disks.

Second, our evaluation of file system interfaces has shown that `mmap` provides simple and efficient fine-grained access to files. With the `mmap` and `madvise` interfaces, copying keys into buckets is completely hidden under the disk transfer time. Obtaining this same performance with the `read` system call would require more programming complexity: because the cost of issuing each `read` is high, users must use threads to prefetch data in large chunks. Further, the file system buffering that occurs with `read` wastes an inordinate amount of memory, which is not acceptable for data-intensive applications.

Third, we have seen that with a simple memory configuration tool, an application can effectively determine and use most of available memory. The memory tool discovered the buffer requirements of `mmap`, roughly a 10% tax applied to free memory. However, an OS interface that gave programs an accurate run-time estimate of free memory would be preferable.

5 One-Pass Parallel Sorting

We have detailed the intricacies of single-node sorting when the records fit into memory, specifically disk striping, memory management, and the in-core sorting algorithm. We now examine the one-pass parallel sort and the issues that arise when communication is added. We also present our best performance on the Datamation benchmark.

Assuming that the records begin evenly distributed across P workstations, numbered 0 through $P-1$, the three steps of the single-node sort extend very naturally into the four-step parallel algorithm:

1. **Read:** Each processor reads records from its local disk into main memory.
2. **Communicate:** Key values are examined and the records are sent to one of $P * B$ local or remote buckets.
3. **Sort:** Each processor sorts its local keys.
4. **Write:** Each processor gathers and writes its records to local disk.

Each of the workstations memory-maps its input files and calculates the processor containing the remote bucket for each key. Our current implementation determines the destination processor with a simple bucket function (i.e., the top $\lg_2 P$ bits of each key) and copies the key from the input file to a *send-buffer* allocated for each destination processor. Once again, this approach for bucketizing assumes that the key values are from a uniform distribution. The specifics of when the records in a send-buffer are sent vary across our implementations and are described in more detail below.

When a message containing records arrives at a processor, an Active Message handler is executed. The handler moves the records into a sort-buffer and copies the partial keys and pointers into the correct local buckets; these operations are directly analogous to distributing partial keys into buckets in the single-node sort. The most significant difference from the single-node version is that rather than distributing records into *local* buckets, each processor distributes records into *remote* buckets. This computation is naturally decoupled with Active Messages: the sending processor determines the destination processor owning a range of buckets; the receiving processor determines the final bucket, performing calculations identical to the single-node version within the message handler.

After synchronizing across all processors to ensure that all records have been sent and received, each node performs an in-core sort on its records and writes out its local portion to local disk. The sort and write steps are identical to the single-node version. At the end, the data is sorted across the disks of the processors, with the lowest-valued valued keys on processor 0 and highest-valued keys on processor $P-1$. Note that the number of records per node will only be approximately equal, and currently depends on the distribution of key values.

5.1 Exploiting Overlap

Simultaneously reading records from local disk while sending records to other processors has the potential to overlap disk-wait time and network latency. However, overlapping these operations is only useful if shared resources, such as the CPU

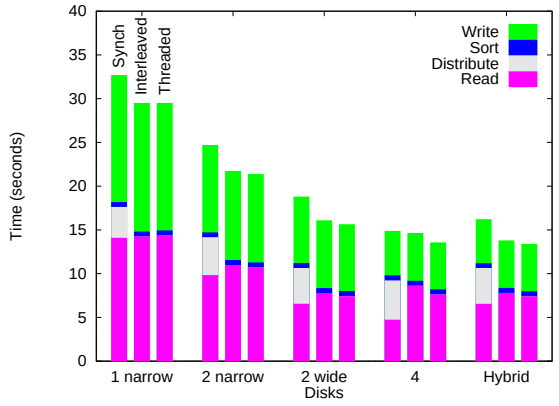

Figure 5: **Comparison of Algorithms.** *The left-most bar of each group shows the performance breakdown of the synchronous implementation, the middle-bar the interleaved implementation, and the right-most bar the threaded version. For the interleaved and threaded versions, read and distribution time is collapsed into the read category. The Hybrid implementation reads from two fast-wide disks, and writes to all four disks.*

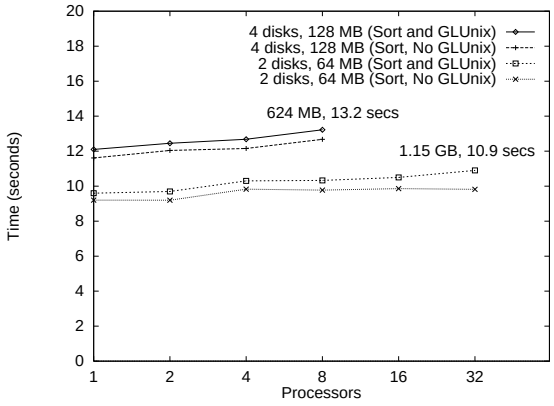

Figure 6: **Scalability of the One-Pass Parallel Sort.** *360,000 records per processor (37 MB) are sorted on the cluster with two disks and 64 MB of memory per machine. 800,000 records per processor (78 MB) are sorted with four disks and 128 MB of memory per machine.*

and the I/O bus, are under-utilized. In this subsection, we evaluate three versions of the one-pass parallel sort that vary the degree to which they overlap disk I/O and communication.

Synchronous: In the synchronous version each processor reads, communicates, sorts, and then writes, with no overlap across the four steps. Each buffer contains $\epsilon * N/P$ records, where ϵ is an expansion factor related to the distribution of key values. The drawback of a straight-forward implementation is that it requires twice as much memory as the records to be sorted: one buffer is needed for the records read and sent by a processor and another buffer is needed for the records received. Our implementation reserves only one extra buffer for communication (*i.e.*, $P + 1$ buffers), but requires extra synchronization across processors. On iteration i, processor p sends to processor $(p + i)$ mod P, to minimize contention. After sending the complete buffer to the first processor, the processors synchronize, waiting until the records have been received. These steps continue until all records have been distributed.

Interleaved: The second implementation alternates reading and communicating, using one thread. As before, there is a send-buffer for each destination, but now it is relatively small (4 KB in the current implementation). Rather than waiting until all records have been read, records are sent after the send-buffer is full. Alternating reading and sending not only has the advantage of overlapping both, but also uses less memory and is more robust to the distribution of key values than the synchronous version. This algorithm only synchronizes across processors once, ensuring that all records have been sent and received, before beginning the in-core sort.

Threaded: The final implementation also overlaps reading and sending, but uses two threads. A *reader-thread* reads the records and moves them into local send-buffers and a *communicator-thread* sends and receives messages. Two sets of send-buffers now exist for each destination; when the reader finishes filling any of the buffers in the set, it signals the communicator which sends all of the records in the current set.

Our measurements, shown in Figure 5, found that with only one or two disks per node, the interleaved versions (both single and dual threaded) outperform the synchronous version by roughly 10 %. Two threads perform slightly better than one because they are better able to stagger read and send requests smoothly over time. With more disks, the difference between the synchronous and overlapped versions diminishes. With four disks in the system, all three algorithms perform nearly identically.

Interestingly, the interleaved versions actually read at a slower rate with four disks than with only two 7200 RPM disks. The reduction occurs because the UltraSPARC I/O Bus, the S-Bus, is saturated long before its theoretical peak of 80 MB/s. We find that we are able to read only about 13 MB/s from the four disks, while simultaneously sending and receiving at approximately 11.5 MB/s, for an aggregate of 36 MB/s over the S-Bus. To make good use of four disks capable of transferring 20 MB/s, the S-Bus would need to be able to sustain roughly 60 MB/s aggregate.

Due to the limited bandwidth of the S-Bus, we found that a *hybrid* system has the best performance when interleaving reading and communication. The hybrid system reads from the two disks on the fast-wide bus, and writes to all four. The write phase still profits from four disks since there is no concurrent communication, and hence the I/O bus is devoted solely to disk activity. Since the threaded parallel sort equals or outperforms the other implementations, we focus on it exclusively for the remainder of our experiments.

5.2 Discussion

Our one-pass parallel NOW-Sort is almost perfectly scalable as the number of processors is increased and the number of records per processor is kept constant, as shown in Figure 6. In other words, we can sort twice as many keys in about the same amount of time by simply doubling the number of processors. The slight increase in time with the P is mostly due to the overhead of GLUnix, our distributed operation system. Remote process start-up increases with more processors, taking approximately 0.6 seconds on eight nodes and 1.1 seconds on 32.

Our performance on the Datamation benchmark is shown in Figure 7. Each processor sorts an equal portion of the one million records, so as more processors are added, each

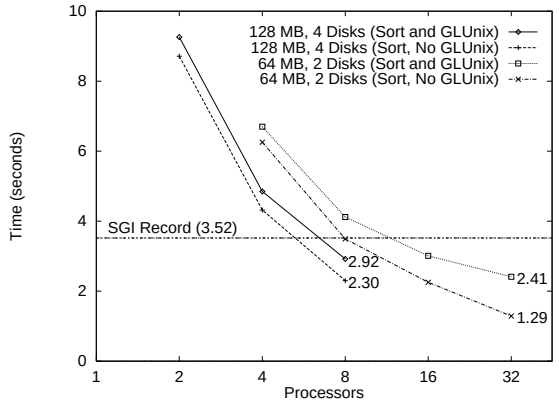

Figure 7: **Performance on the Datamation Benchmark.** *The graph shows the time to sort 1 Million records with four disks and 128 MB per processor and with two disks and 64 MB per processor. Data points are not shown when one million records will not fit in the available memory.*

node sorts fewer records. With the resulting small problem sizes, remote process start-up is a significant portion of our total sorting time. In fact, on 32 processors, the total time is almost equally divided between process start-up and our application. However, this is probably more a function of the lack of maturity of our own cluster OS, than the fundamental costs of a distributed operating system, and is currently the focus of optimization.

Interestingly, process start-up can also be time-consuming on SMPs: both [26] and [30] indicate that `bzero`'ing the address-space is a significant, if not dominant, cost for one-pass external sorts, most likely because the SMP OS does not parallelize the process. On NOWs, this aspect of process creation is not a problem, since each local address space is initialized in parallel.

The parallel single-pass sort revealed another I/O-system bottleneck in the UltraSPARC I architecture: the S-Bus. Because the S-Bus can achieve only about 36 MB/s, a workstation can not effectively use more than two disks when simultaneously communicating over the network. To remain a viable alternative to SMPs, UltraSPARC I/O bandwidth must improve dramatically or, more radically, provide communication over the memory bus without coherence. The corollary to this suggests that a system with two disks on a fast-wide SCSI bus is the sweet-spot in the cost/performance curve.

In conclusion, we found developing the parallel single-pass sort to be a natural extension from the single-node version: the lessons described in the previous section regarding disk striping, memory management, and in-core sorting were all directly applicable. In the NOW environment, we were able to isolate performance effects on a per-processor basis; the only difference from the single-node version was additional communication traffic over the S-Bus. In contrast, we believe that in an SMP system the pooling of resources obscures this type of analysis. Finally, the Active Message interface not only provided high bandwidth and low overhead communication, but also facilitated an algorithm which differed only slightly from the single-node version; instead of copying keys into local buckets, each processor now copies keys into buckets spread across processors.

6 Two-Pass Single-Node Sorting

Having described the one-pass versions of NOW-Sort, we now detail the extensions needed for a single-node when there is insufficient memory and two passes must be made over the records. As described in Section 4.1.3, we use our memory configuration tool to determine the amount of available memory so that we can choose between the one-pass and two-pass algorithms.

1. **Create Runs:** The one-pass sort (**Read**, **Sort**, and **Write**) is repeated to create multiple sorted runs on disk.
2. **Merge:** In this new phase, the sorted runs are merged into a single sorted file.

Create Sorted Runs: In the first phase, two threads are created: a *reader-thread* and a *writer-thread*. Available memory (M) is divided into B_c buffers, where B_c equals one or two, depending upon whether the reader and writer are synchronous or overlapped, as discussed in the next section. Each run contains M/B_c records, requiring $R = \frac{N}{M/B_c}$ runs, where N is the number of records. The reader copies the records from disk and moves keys and pointers into buckets, and then signals that the buffer is full. The reader is then free to fill the next empty buffer with the next run. The writer waits for a buffer to be filled, sorts each bucket, writes the records to disk, and then signals that the buffer is empty. This process is repeated until the R runs are sorted and written to disk as separate files.

Merge Sorted Runs. In the second phase, multiple runs are merged into a single output file. Our implementation contains three threads: a *reader*, a *merger* , and a *writer*. The reader-thread memory-maps the R run files and reads the first chunk of records from each run into one of B_m sets of R merge buffers; B_m may be one or two, depending upon whether or not reading and merging are synchronous or overlapped, also discussed in the next section. Prefetching records in large chunks (currently about 512 KB) amortizes the seek time across runs, and thus obtains nearly the same read bandwidths as sequential accesses. After reading the R buffers, the reader signals to the merger that these buffers are full, and continues to prefetch into the next set of empty merge buffers. The merger selects the lowest-valued key from the top of each run, and copies the record into a write buffer. When the write buffer is full, the writer it signaled, which writes the records to disk.

Note that the merge phase is an instance where we found that a simple implementation using `mmap` and `madvise` does not attain sequential disk performance. When accessing multiple read streams from the same disk, `mmap` does not prefetch data in sufficiently large blocks to amortize the seek costs. Thus, our merge phase must explicitly manage prefetching with multiple threads and buffers.

6.1 Exploiting Overlap

In both phases, reading from disk may be overlapped with writing and with computation. As in the one-pass parallel sort, overlapping phases by pipelining is only beneficial when resources, such as the CPU and I/O bus, are under-utilized; Overlapping now has additional implications for the layout of records across disks and for memory usage, and pipelining one phase implies pipelining both phases, since the output from the first phase is the input of the second. Therefore, we investigate the impact of run length and the layout of records across disks on both phases.

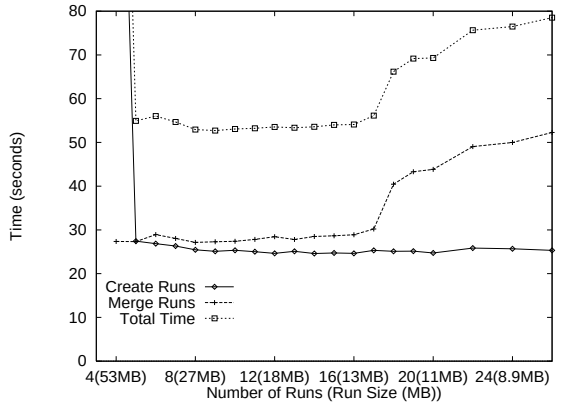

Figure 8: **Effect of Run Size.** *Non-cumulative time for both phases of the two-phase sort are shown, as well as the total time, for 2M keys on an UltraSPARC 1 with 128 MB of memory and four disks. When creating runs in phase one, each run must fit in memory. When merging runs in phase two, the number of runs should be 17 or less.*

Run Length: The synchronous version of phase one uses all of memory to create each run; thus, it generates half as many runs as the pipelined version. More runs may adversely affect the performance of the merge phase, since reading from more disk streams may raise the seek time beyond that which we hide by prefetching large chunks. The number of runs is also constrained such that there exists sufficient memory for the prefetch buffers of each run in the merge phase (*i.e.*, $512kb \cdot R \cdot B_m < M$).

Disk Layout: The synchronous and pipelined versions must manage the layout of records across disks differently. To maintain sequential access, each disk must be dedicated to either reading or writing at any given time. Since the pipelined implementations are both reading and writing at nearly all times, half of the disks must be used for reading, and the other half for writing. The synchronous implementations can utilize all the disks for reading and then all of the disks for writing. Thus, when both phases are synchronous, the input file, the temporary sorted runs, and the output file are striped across all disks; when both phases are pipelined, each file is striped across only half the disks. When one phase is synchronous but the other is not, the synchronous phase suffers because it can only use that half of the disks employed by the pipelined phase.

Measurements on both two-disk and four-disk systems reveal that the best two-pass single-node version occurs when both phases are pipelined. The pipelined merge phase exhibits a dramatic improvement over the synchronous version because it effectively prefetches large chunks of each merge run and hides the cost of seeking. The pipelined merge improves slightly when the first phase is synchronous, since it has fewer runs to merge; however, the penalty to the first phase of executing synchronously while only writing to half of the disks outweighs this improvement.

6.2 Determining Run Size

Within the pipelined algorithm, the *length* of each run impacts the performance of the first phase, while the *number* of runs impacts the second phase. In this section, we determine a compromise in the number of runs and their lengths that results in good performance for both phases. The tension between the two phases when sorting a fixed amount of data and changing the number of runs is shown in Figure 8.

When few runs are created (four or less), the time for the first phase is large since the size of each run is greater than the amount of available memory (M/B_c); thus, the memory manager pages data to and from disk. Beyond the point where each run fits in memory, the time for the first phase continues to drop slightly as the run size decreases, due to the lower cost of filling and draining the read/write pipeline. That is, with less data per run, the time the writer must wait for the reader to read the first run decreases, and the time for the writer to write the last run decreases.

Conversely, as the number of runs increases up through 17, the time for the second phase increases. The merge phase reads records from disk in 512 KB chunks to amortize the cost of seeking across independent runs, creating a similar start-up effect. The start-up time increases with the number of runs because the reader prefetches a fixed chunk for each of the runs. The net result between the "memory wall" and 17 runs is that increasing the number of runs for a given problem set has no overall effect on performance, slightly decreasing the time to generate the runs, while increasing the time to merge the runs. After 17 runs, performance degrades rather sharply again, with merge time increasing by 30 to 50 percent. Currently, we are not sure of the cause of this behavior, but suspect that it is related to cache performance, since the performance drop-off is mirrored by an increase in user-CPU time, and not disk-wait time.

6.3 Discussion

We conclude by summarizing the MinuteSort performance of the two-pass single-node sort with two and four disks in Table 3. The time for the merge phase is slightly larger than the time to create the runs because the records are divided into twelve runs, which concurs with the data in Figure 8. We use these results as a comparison case for the two-pass parallel sort in the next section.

	2 Disks, 110 MB	4 Disks, 229 MB
Create	28.6 s (7.7 MB/s)	26.7 s (17.2 MB/s)
Merge	29.4 s (7.5 MB/s)	32.5 s (14.1 MB/s)
Total	58.4 s	60.4 s

Table 3: **Single-node MinuteSort Results.** *The amount of data that can be sorted in one minute on a single node is shown for a UltraSPARC I with 2 disks and 64 MB of memory versus one with 4 disks and 128 MB of memory. 110 MB corresponds to 1,150,000 records; 229 MB to 2,400,000 records.*

In this section, we again saw that NOW-Sort must have an accurate estimate of available memory: to choose between the one-pass or two-pass algorithm for a given number of records, and to determine the maximum run size in the two-pass algorithm.

Examining the two-pass single-node sort, we found that pipelined versions of both phases performed better than synchronous implementations, regardless of the number of disks. We found that `mmap` with `madvise` does not prefetch sufficiently large blocks to amortize seek costs across multiple streams, forcing us to explicitly use a prefetching thread in the

merge phase. By prefetching large blocks, sequential performance is maintained even when the merge phase must read from R runs, where R is greater than the number of disks in the system.

Finally, we have observed the importance of carefully managing the layout of records across disks; by dedicating half the disks for reading and half for writing, we do not disrupt the sequential access pattern generated by each stream.

7 Two-Pass Parallel Sorting

In this section we describe our two-pass parallel algorithm and our results on the MinuteSort benchmark. This final sorting algorithm extends naturally from the one-pass parallel and two-pass single-node algorithms.

1. **Create Runs:** Processors create sorted runs across all of the nodes of the cluster. Runs are created by repeating the **Read**, **Send**, **Sort**, and **Write** steps of the one-pass parallel sort.

2. **Merge:** Each processor merges its local sorted runs into a single local file. This phase is identical to the merge on a single-node described in the previous section and is not discussed further.

7.1 Exploiting Overlap

Creating multiple sorted runs across processors in the first phase contains several opportunities for overlap. In the one-pass parallel algorithm, reading and sending could be overlapped; in the two-pass single-node sort, reading and writing could be overlapped. In the two-pass parallel sort, we may overlap none, one, or both of these pairs of operations. To understand the tradeoffs, we describe the actions of the three threads in more detail: the *reader*, the *communicator*, and the *writer*.

The read thread is responsible for mapping the input file and copying keys and records into per-destination send-buffers. When any of the send-buffers fill, the reader signals the communicator. As in the one-pass parallel sort, the reader and communicator can be overlapped (using two sets of send buffers) or synchronous (with one set).

When signaled, the communicator-thread picks up the entire set of send-buffers and sends each to the appropriate destination processor. When a message arrives, the Active Message handler is invoked, which copies the records into the sort-buffer and the partial keys into the appropriate bucket. After the communicator has sent all of the records in the current run, the processors synchronize and wait to receive the records.

After a run has been sent and received, the writer-thread is signaled. The writer sorts the keys within each bucket and then gathers and writes the records to disk. Similar to the two-pass single-node sort, the reading of one run may be overlapped with the writing of the previous run. Overlapping implies that two sets of sort-buffers and buckets exist, each set consuming approximately half of memory.

Our measurements indicate that the number of disks in the system determine which operations should be overlapped. With two disks, the best implementation overlaps all four operations. With four disks, the best overlaps reading and sending, but performs reading and writing synchronously. When all operations are overlapped, the CPU is saturated at nearly 95% utilization. Performing the read and write operations synchronously not only minimizes CPU and I/O bus contention, but by creating only one run at a time, stripes the input file across all four disks and creates less runs to merge in phase two.

7.2 Discussion

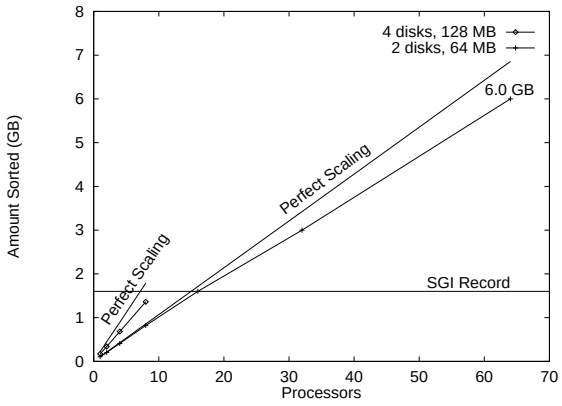

Figure 9: **Parallel MinuteSort Results.** *This graph shows MinuteSort results on two different cluster configurations: an 8-node cluster with 4 disks and 128 MB per processor and a 64-node cluster with 2 disks and 64 MB per processor. Perfect scaling is relative to single-node performance.*

The performance of the two-pass parallel NOW-Sort on the MinuteSort benchmark is shown in Figure 9. With eight machines and four disks per machine, our implementation sorts 1.4 GB in a minute. We experience a large drop in the number of keys that can be sorted when we move from the single-node sort to the parallel version running on one processor because reading and writing are now performed synchronously instead of overlapped; however, compared to the parallel version on one processor, the algorithm scales perfectly linearly with more processors.

On the cluster with two disks per machine, the parallel version of the sort on one machine sorts as many keys as the single-node algorithm, since both implementations overlap all phases. With 16 processors, we sort 1.6 GB of data, which ties the previous MinuteSort record by the more fully-loaded SGI system: 12 processors, 96 disks, and 2.25 GB of memory. With 32 processors, we sort 3.0 GB in 57.9 seconds, and on 64 nodes, we sort 6.0 GB in 59.3 seconds. The algorithm does not scale quite linearly with the number of processors; the drop is largely due to start-up bottlenecks, costing us 3 seconds on 64 nodes.

Unfortunately, due to the relative dearth of memory on our workstations, the MinuteSort is a two-pass algorithm on our cluster; on previous record-holding systems, MinuteSort was performed in a single pass. For example, to sort 1.6 GB of data, we are performing twice the I/O of the SGI XFS system in the same amount of time.

8 Conclusions

In this paper we presented NOW-Sort, a collection of configurable sorting algorithms for networks of workstations. We have shown that NOWs are well-suited to I/O-intensive applications such as sorting. The cluster environment is ideal for the development of efficient parallel applications because

System	Year	Procs	Disks	Mem	Datamation		MinuteSort	
				(GB)	Time (s)	Price/Perf (cents)	Data (GB)	Price/Perf ($/GB)
DEC Alpha AXP 3000	1993	1	10	0.256	13.7	0.9	–	–
DEC Alpha AXP 7000	1993	3	28	0.256	7.0	1.4	–	–
DEC Alpha AXP 7000	1993	3	36	1.25	–	–	1.1	0.47
SGI Challenge XL	1995	12	96	2.25	3.5	0.4	1.6	0.61
IBM RS/6000 39H	1996	1	8	0.256	5.1	0.2	–	–
NOW UltraSPARC	1996	8	32	1.00	2.92	0.32	1.4	0.13
NOW UltraSPARC	1996	16	32	1.00	3.01	0.57	1.6	0.19
NOW UltraSPARC	1996	32	64	2.00	2.41	0.91	3.0	0.20
NOW UltraSPARC	1996	64	128	4.00	–	–	6.0	0.20
Ideal NOW UltraSPARC	*1996*	*28*	*56*	*8.76*	–	–	*6.2*	0.10

Table 4: **Summary of Datamation and MinuteSort Results.** *Note that the price of the SGI Challenge is estimated and the IBM RS/6000 uses raw disk, which is not allowed in the Datamation specification. Note that 1 GB is* 2^{30} *bytes.*

it provides performance isolation: by monitoring the behavior of the application on each node, the program-developer can isolate performance bottlenecks. Clusters also enable incremental scalability: hardware, whether it be processors, disks, or memory, can be added to the system and used in an effective manner.

This work was enabled by two key pieces of software on the NOW cluster. The first of these is Active Messages, a high-speed communication layer providing low latency and high throughput to parallel programs [13, 33]. The second key software component is GLUnix, a distributed operating system for NOWs [19].

By studying disk-to-disk sorting, we qualitatively and quantitatively assessed the workstation operating system and machine architecture. One of the underlying goals of the NOW project is to utilize existing commodity software and hardware; the assumption is that general-purpose workstations form solid building blocks for clusters connected via a high-speed network. In this paper, we demonstrated some of the strengths and weaknesses of this assumption.

For NOW-Sort, the most important function of the operating system is to support efficient file access. We find that `mmap` with `madvise` in Solaris 2.5.1 provides fast, fine-grained access to files, with the file system usually providing adequate prefetching and buffer management; however, it consumes 10% of available memory. The older `read` interface still incurs the double-buffering problem, and should not be used by programs with large memory requirements. We did find that `mmap` with `madvise` was not sufficient when reading from multiple independent streams in the merge phase of the two-pass sort; there, we needed to use multiple threads to manage prefetching ourselves.

We found the existing Solaris support for I/O- and memory-intensive applications to be lacking in two key areas. First, efficiently employing multiple disks of differing speeds on a single workstation is not straight-forward. Second, there is no reasonable way to determine available memory, so that an application does not over-extend resources and thrash. To solve these problems, we developed two small configuration tools. The first of these evaluates the speed of the disks in the system and feeds this information into our striping library, allowing us to get maximal performance from our disks. The second tool measures available memory on each node of the cluster. These two tools combine to enable the sorting application to tune itself to the available cluster and attain peak efficiency.

We found that the I/O bus structure of the UltraSPARC I workstation provides inadequate support for the disk and communication needs of sorting. For example, we found that with four disks, an implementation with synchronous reading and writing was sometimes superior to a pipelined version. Indeed, we were surprised to find that a machine with just two disks and a connection to a high-speed network is the most effective building block, as adding more disks leads to little benefit. There are two solutions to this pending problem: an improved I/O infrastructure, with enough bandwidth to support disks and network, or communication facilities that function over the *memory* bus (perhaps without coherence), leaving only the disks to consume precious I/O bus bandwidth. We also found that the internal fast-narrow SCSI bus prevents full utilization of the bandwidth of the two internal disks, a 25% performance loss.

The performance of our system relative to previous record-holders on both the Datamation and MinuteSort benchmarks is shown in Table 4. Our best-performing configuration of 32 processors sorts one million records in one-pass in 2.41 seconds, but at a relatively high cost. The 8 processor system with four disks per node is slightly slower, at 2.92 seconds, but has significantly better cost/performance. The IBM RS/6000 still achieves the best cost/performance of all known results, but uses raw disk, which is not allowed by the benchmark specification.

Our MinuteSort results were obtained on machine configurations with little memory, and thus required two-pass external algorithms. The previous record-holders contained enough main memory to hold the data. Even performing twice the amount of I/O, we were able to sort 6.0 GB (6 x 2^{30} bytes) in one minute on 64 processors. In this configuration, with only 128 disks, we are able to deliver roughly 431 MB/s to the sorting application. The previous MinutesSort record-holder needed 96-disks to deliver only 54 MB/s. For the best price/performance, we project that if we had 28 processors with two disks and 320 MB of memory each, we could fit 6.2 GB of records in memory; thus, by sorting in only one-pass, we could use much less hardware to obtain slightly better performance.

Visit our home page to find out more about NOW-Sort and the NOW Project: `http://now.cs.berkeley.edu`.

Acknowledgments

NOW-Sort was the combined effort of many members in the the Berkeley NOW Project. We would first like to thank Richard P. Martin for implementing and supporting Split-C and Active Messages 1.0 on

Myrinet. This work would not have been finished without his help. We also thank Doug Ghormley and David Petrou for their efforts in making GLUnix a fast, reliable system. We thank the members of the Tertiary Disk group for lending us the Barracuda disks, and Ken Lutz and Eric Fraser for making both clusters a working reality. We extend special thanks to Jim Gray and Chris Nyberg for comments on early drafts as well as helpful conversations about sorting. Finally, we thank the anonymous reviewers for providing useful feedback.

This work was funded in part by DARPA F30602-95-C-0014, DARPA N00600-93-C-2481, NSF CDA 94-01156, NASA FDNAGW-5198, and the California State MICRO Program. Andrea Arpaci-Dusseau is supported by an Intel Graduate Fellowship.

References

[1] R. C. Agarwal. A Super Scalar Sort Algorithm for RISC Processors. In *Proceedings of the 1996 ACM SIGMOD Conference*, pages 240–246, June 1996.

[2] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, Feb. 1994.

[3] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of SIGMETRICS/Performance '95*, pages 267–78, 1995.

[4] C. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhnigran, S. Padmanabhan, and W. Wilson. An Overview of DB2 Parallel Edition. In *Proceedings of 1995 SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995.

[5] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, Dec. 1995.

[6] B. Baugsto, J. Greipsland, and J. Kamerbeek. Sorting Large Data Files on POMA. In *Proceedings of COMPAR-90 VAPPIV*, pages 536–547, Sept. 1990. Springer Verlag Lecture Notes No. 357.

[7] M. Beck, D. Bitton, and W. K. Wilkinson. Sorting Large Files on a Backend Multiprocessor. Technical Report 86-741, Department of Computer Science, Cornell University, Mar. 1986.

[8] G. Blelloch, C. Leiserson, and B. Maggs. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Symposium on Parallel Algorithms and Architectures*, July 1991.

[9] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the International Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

[10] N. Boden, D. Cohen, R. E. Felderman, A. Kulawik, and C. Seitz. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, Feb. 1995.

[11] H. Boral, W. Alexander, L. Clay, G. Copeland, et al. Prototyping Bubba, a Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, Mar. 1990.

[12] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Supercomputing '93*, 1993.

[13] D. Culler, L. T. Liu, R. Martin, and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, Feb. 1996.

[14] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, et al. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, Mar. 1990.

[15] D. Dewitt, J. Naughton, and D. Schneider. Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting. In *Proceedings of the International Conference on Parallel and Distributed Information Systmes*, 1991.

[16] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin. Fast Parallel Sorting Under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):791–805, Aug. 1996.

[17] A. et. al. A Measure of Transaction Processing Power. *Datamation*, 31(7):112–118, 1985. Also in *Readings in Database Systems*, M.H. Stonebraker ed., Morgan Kaufmann, San Mateo, 1989.

[18] B. Gerber. Informix Online XPS. In *Proceedings of 1995 SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995.

[19] D. P. Ghormley, D. Petrou, A. M. Vahdat, and T. E. Anderson. GLUnix: A Global Layer Unix for NOW. http://now.cs.berkeley.edu/Glunix/glunix.html.

[20] G. Graefe. Volcano: An Extensible and Parallel Dataflow Query Processing System. Technical report, Oregon Graudate Center, June 1989.

[21] G. Graefe. Parallel External Sorting in Volcano. Technical Report CU-CS-459, Computer Science, University of Colorado at Boulder, June 1990.

[22] M. D. Hill, J. R. Larus, S. Reinhardt, and D. A. Wood. Cooperative-Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, 1993.

[23] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[24] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, D. Williams, M. Smith, S. Barton, and G. Skinner. Symmetric Multiprocessing in Solaris 2.0. In *Proceedings of COMPCON Spring '92*, 1992.

[25] X. Li, G. Linoff, S. Smith, C. Stanfill, and K. Thearling. A Practical External Sort for Shared Disk MPPs. In *Proceedings of SUPERCOMPUTING '93*, pages 666–675, Nov. 1993.

[26] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *Proceedings of 1994 ACM SIGMOD Conference*, May 1994.

[27] B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren, and B. Vaughna. FastSort; A Distributed Single-Input Single-Output External Sort. *SIGMOD Record*, 19(2):94–101, June 1990.

[28] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[29] M. Stonebraker. The Case for Shared Nothing. *Database Engineering*, 9(1), 1986.

[30] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Annual Technical Conference*, Jan. 1996.

[31] Tandem Performance Group. A Benchmark of NonStop SQL on Debit-Credit Transactions. In *Proceedings of SIGMOD International Conference on Managament of Data*, Chicago, IL, June 1988.

[32] Teradata Corporation. *DBC/1012 Data Base Computer System Manual*, release 2.0 edition, Nov. 1985. Document Number c10-0001-02.

[33] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[34] H. Young and A. Swami. The Parameterized Round-Robin Partitioned Algorithm for Parallel External Sort. In *Proceedings 9th International Parallel Processing Symposium*, pages 213–219, Santa Barbara, CA, Apr. 1995.

[35] M. Zagha and G. Blelloch. Radix Sort for Vector Multiprocessors. In *Supercomputing*, 1991.

[36] W. Zhang and P. Larson. A Memory-Adaptive Sort (MASORT) for Database Systems. In *Proceedings of CASCON '96*, Toronto, Nov. 1996.