Searching for the Sorting Record: Experiences in Tuning NOW-Sort

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau,
David E. Culler, Joseph M. Hellerstein, and David A. Patterson

Computer Science Division,
University of California, Berkeley
{dusseau,remzi,culler,jmh,pattrsn}@cs.berkeley.edu

Abstract

We present our experiences in developing and tuning the performance of NOW-Sort, a parallel, disk-to-disk sorting algorithm. NOW-Sort currently holds two world records in database-industry standard benchmarks. Critical to the tuning process was the setting of *expectations*, which tell the programmer both *where* to tune and *when* to stop. We found three categories of useful tools: tools that help set expectations and configure the application to different hardware parameters, visualization tools that animate performance counters, and search tools that track down performance anomalies. All such tools must interact well with all layers of the underlying software (*e.g.*, the operating system), as well as with applications that leverage modern OS features, such as threads and memory-mapped I/O.

1 Introduction

On March 1, 1996, after much debate about how to benchmark our prototype cluster system, we decided to implement an external, or disk-to-disk, parallel sort. External sorting had all of the qualities we desired in a benchmark. First, external sorting is both memory and I/O intensive. Second, sorting stresses many aspects of both local and distributed operating system performance. Third, well-understood sorting algorithms exist for both sequential and parallel environments. Finally, the presence of industry-standard benchmarks allows us to compare our performance to that of other large-scale systems.

At that time, a 12-processor SGI Challenge held the world-records on the two existing benchmarks. For the Datamation benchmark [13], the SGI had sorted 1 million 100-byte records with 10-byte keys from disk to disk in 3.52 seconds. For MinuteSort [25], 1.6 GB of these 100-byte records were sorted within one minute [32]. Our goal was to surpass these records with a cluster of 105 UltraSPARC I workstations connected with a high-speed network.

Over one year later, on April 1, 1997 (April Fool's day), with the help of many people in the U.C. Berkeley NOW project [3], we laid claim to both benchmark records. On 32 machines, we reduced the time to sort 1 million records to 2.41 seconds; more importantly, in 59.7 seconds we sorted over 8.4 GB on 95 machines. These NOW-Sort Datamation and MinuteSort records still stand today.

We described the NOW-Sort algorithms and our performance measurements in a previous paper [5]; we now relate our difficulties in scaling this memory- and I/O-intensive application to 95 machines. Along the way, we learned much about scalable systems, disk performance, cache-sensitive algorithms, operating system interfaces, and memory management. In this paper, we focus on our methodology for tuning NOW-Sort, as well as the tools we found useful.

Our methodology for achieving scalable performance relied on setting and meeting optimistic performance *expectations*. We began by forming a set of performance goals for each phase of the sort. We then repeatedly characterized and implemented progressively larger components of the sort, and measured whether or not our implementation met our expectations. When we found discrepancies, we focused our optimization efforts on the slow phases of the sort, either fixing performance bugs in the application or underlying system, or refining our expectations to be more realistic. With expectations for each phase of our algorithm, we not only knew *where* to start the tuning process, but also *when* to stop.

We found performance tools were useful for three distinct purposes: setting expectations, visualizing measured performance, and searching for anomalies. First, *configuration* tools helped to define expectations for each resource by measuring the best-case performance that could be achieved; the output of these tools was also sometimes used to parameterize the sort to the hardware at hand, allowing the sort to achieve peak utilization on a variety of machine configurations. Second, we found that simple *visualization* tools allowed us to quickly identify performance problems; by animating machine performance counters, we were able to notice the cause and nature of problems that arose. We cannot emphasize enough the importance of performance counters at all levels of the system: the CPU, I/O bus, and even network switches and links. While we were able to easily implement simple tools for these first two areas, we did not implement a hierarchical *search* tool to remove run-time anomalies. This tool would have been helpful in the final stages of our efforts, to help achieve a peak-performance, undisturbed run of NOW-Sort. Without such a tool, we were forced to track down anomalies by hand, first isolating the problem workstation and then identifying the aspect of that machine that differed from the others.

The paper is organized to roughly follow the chronology of our experience. We begin in Section 2 by describing our

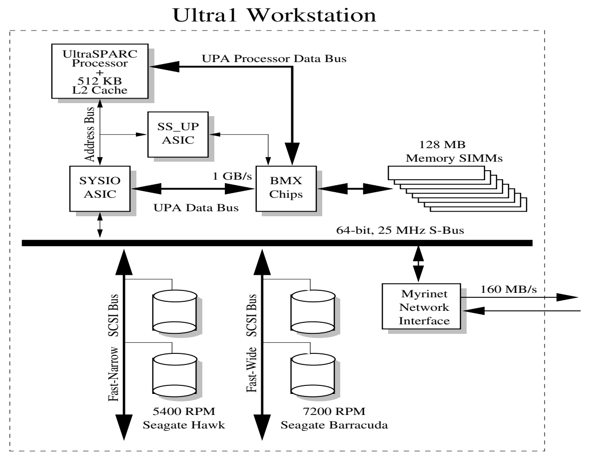

Figure 1: **Ultra1 Workstation.** *The figure depicts the internal architecture of an Ultra1 workstation. BMX is a crossbar connection to memory, and SYSIO is the main I/O controller.*

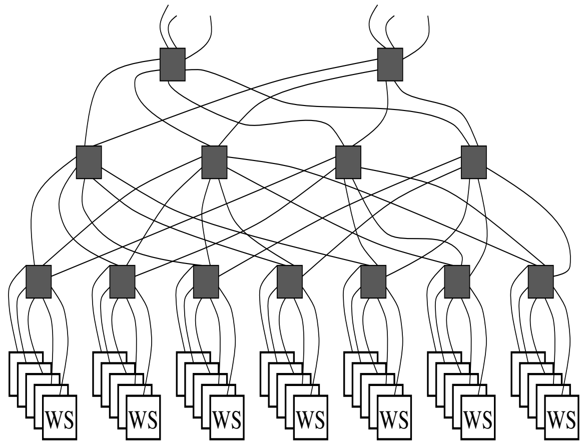

Figure 2: **Berkeley NOW Cluster.** *The figure shows 35 workstations connected with 13 eight-port Myrinet switches. Three such groups comprise the entire 105-node cluster.*

cluster hardware and software. In Section 3, we discuss the development and tuning of the single-node version of the algorithm, which forms the core of the parallel implementation. Section 4 develops the model and basic algorithm of the parallel version of NOW-Sort. The difficulties of scaling to the full-sized system are detailed in Section 5. In Section 6, we relate the experience of isolating performance problems in large systems, what we call "finding the needle in the NOW-stack." Finally, we conclude with some reflections on our experience.

2 The NOW Cluster

2.1 Hardware

The Berkeley NOW cluster consists of 105 commodity Ultra1 workstations [33]. Each Ultra1 workstation contains a single UltraSPARC I processor, with 16 KB on-chip instruction and data caches, and a unified 512 KB second-level cache. At the base of the memory hierarchy is 128 MB of DRAM. A diagram of the internal architecture of the Ultra1 workstation is shown in Figure 1.

The I/O system of the Ultra1 centers around the S-Bus I/O bus, with a peak theoretical bandwidth of 80 MB/s. [1] On the I/O bus, each workstation houses two internal 5400 RPM Seagate Hawk disks, on a single fast-narrow SCSI bus (peak 10 MB/s). To experiment with more disk bandwidth per node, we added two additional 7200 RPM Seagate Barracudas disk on an extra fast-wide SCSI card (peak 20 MB/s) in some configurations. Thus, while the cluster can generally be viewed as a homogeneous collection of workstations, the extra disks run at faster rates, providing yet another challenge for the I/O-conscious programmer.

In addition to the usual connection to the outside world via 10 Mb/s Ethernet, every workstation contains a single Myrinet network card, also attached to the S-Bus. Myrinet is a switch-based, high-speed, local-area network, with links capable of bi-directional transfer rates of 160 MB/s [7]. Each Myrinet switch has eight ports. Our 105-node cluster is comprised of three 35-node clusters, each containing 13 of these switches, connected in a 3-ary, tree-like structure, as shown in Figure 2.

[1] The 64-bit S-bus actually has a higher peak rate, but because our devices only use 32 bits of the bus, 80 MB/s is the peak.

2.2 Support Software

Though we were tuning a user-level application, the performance of NOW-Sort was directly affected by a number of other important software sub-systems, including the multi-layer operating system and communication layer.

Each machine in our cluster runs a copy of Solaris 2.5.1, a multi-threaded version of UNIX [17]. Some of the strengths of Solaris include its efficient support for kernel-level threads, a well-developed Unix file system, and sophisticated memory management.

However, 105 machines running Solaris does not a cluster make. To present the illusion of a single, large-scale system to the end user, our cluster employs GLUnix, the prototype distributed operating system for the Berkeley NOW [14]. GLUnix monitors nodes in the system for load-balancing, coschedules parallel programs, and provides job control and I/O redirection. Since all of our measurements took place in a dedicated environment, we primarily used GLUnix as a parallel program launcher.

The parallel versions of NOW-Sort are written with the support of the Split-C library [10]. Split-C is a parallel extension to C that supports efficient access to a global address space on distributed memory machines. The Split-C library provides many useful group synchronization and communication primitives, including barriers and reductions.

For communication, we utilized Active Messages [34], a communication layer designed for the low latency and high bandwidth of switch-based networks. An Active Message is essentially a restricted, lightweight 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. The version of Active Messages that we used, GAM, supports only a single communicating process per workstation at a time. GAM over the Myrinet has a round-trip latency of roughly 20 μs and a uni-directional bandwidth (one node sending, another receiving) of 35 MB/s [11].

3 Single-Node Performance

When tuning an application, most programmers have a set of performance expectations they hope to achieve, whether a MFLOPS rating, an absolute execution time, or a certain level of speed-up. Without these expectations, the programmer cannot determine at what point he or she has reached sufficient performance. However, many times, expectations are only vaguely defined, or worse, omitted entirely. Our experience has shown that explicitly defining expectations for the important components of the system can greatly simplify the tuning of complex applications.

In this section, we describe our experience developing and meeting expectations for the primary building-block of NOW-Sort: the single-node algorithm. The path that we followed contained three simple steps. First, we carefully chose our initial expectations, based on models and measurements of the system when operating under ideal circumstances. Second, we implemented the most straight-forward algorithm with potential for achieving these goals. Finally, we measured the performance of all phases of our implementation, and focused our efforts on only those portions that did not meet our expectations.

3.1 Defining Expectations

Since sorting is a benchmark which stresses the I/O and memory sub-systems of a machine, our expectations focused on I/O and memory system performance. Before designing our algorithm, we enumerated four goals for the single-node sort:

1. Perform the minimum amount of I/O required.
2. Avoid paging of virtual memory.
3. Ensure that I/O dominates the total run-time.
4. Transfer data from disk at peak (*i.e.* sequential) rates.

Our first expectation is the most basic: given that disks are the slowest component of the system, their use should be minimized. In general, better overall performance can be achieved by performing extra computation or additional memory copies rather than accessing disk. For sorting, the minimum amount of required I/O is determined by the amount of memory in the system. When the records to be sorted fit into available memory, a one-pass sort can be used, where each record is read and written from disk exactly once. Otherwise a two-pass sort is required, where each record is read and written from disk twice.[2]

Second, the use of memory should be explicitly managed such that paging of virtual memory to disk does not occur. Paging is especially damaging in our environment because one of the disks that holds records for sorting is also used for swap space. Therefore, not only does paging hurt memory performance, it also interferes with the sequential access patterns of the disk transfers.

Our third expectation is motivated by previous work in disk-to-disk sorting. Previous researchers found clever ways to hide the cost of internal sorting by overlapping it with disk I/O[2, 25]. Therefore, our goal is do the same in our implementation.

Finally, transfers to and from disk should proceed at the maximum rate. To achieve maximum throughput, disk seeks must be minimized; this can be achieved by accessing disk sequentially or with sufficiently large transfer sizes. Since sorting accesses data on disk in a regular and deterministic fashion, this goal should be attainable.

These four expectations can be used to predict the desired time for the sort as a function of the number of records, N, and the size, S, of each record:

$$\text{Expected Time} = \begin{cases} \frac{N \cdot S}{B_{read}} + \frac{N \cdot S}{B_{write}} & \text{if } N \cdot S < M \\ 2\frac{N \cdot S}{B_{read}} + 2\frac{N \cdot S}{B_{write}} & \text{otherwise} \end{cases}$$

where M is the amount of available memory,and B_{read} and B_{write} are the peak read and write disk bandwidths.

Determining the amount of available memory and peak disk bandwidth in our system required that we implement two simple configuration tools.

Memory Configuration Tool: Our first two expectations require that NOW-Sort be aware of the amount of available memory. In the ideal case, this amount would be reported by a simple system call to the operating system. However, because Solaris 2.5.1 does not give an accurate assessment of free memory, we developed a user-level tool. By accessing various amounts of memory while monitoring paging activity, the memory configuration tool accurately estimates the amount a user program can safely allocate. For example, on a machine with 64 MB of physical memory, the configuration tool determined that 47 MB of memory were available for user applications. While our tool has a number of limitations (notably that available memory must remain constant while the tool and the application execute), it is sufficient for forming both expectations: it determines whether one or two passes are required over the records on disk and it specifies the number of records that can be sorted in a pass without paging.

Seagate Disks	SCSI Bus	B_{Read} (MB/s)	B_{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 Hawks + 2 Barracudas	Both	20.5	19.1

Table 1: **Disk Bandwidth Expectations.** *The Read and Write columns show the peak bandwidth as reported by the disk configuration tool. Measurements of the I/O system were required to determine that the SCSI-bus limits peak performance.*

Disk Configuration Tool: To concretely set our expectations for peak disk bandwidth, we developed a simple disk configuration tool. This program serves two purposes; not only does it automatically adjust the amount of data striped to each disk (to handle disks with differing performance characteristics), but it also reports the maximum achievable bandwidth when the processor is performing no other work. Without this tool, we would have been unaware of bottlenecks in the I/O system and would have set unrealistically high expectations for the achievable bandwidth from multiple disks. For example, although we could read from one 5400 RPM disk at 5.5 MB/s, the disk configuration tool revealed that we could read from two disks at only 8.3 MB/s, due to saturation of the fast-narrow SCSI bus. With a naive model, our extrapolated expectation would have been 11 MB/s, a difference of 30%. Our full set of

[2] An optimization can be performed on the two-pass model: the last sorted run of the first pass does not need to be temporarily written to disk and then read back.

expectations for disk performance, as determined by the disk configuration tool, are shown in Table 1.

3.2 One-Pass Algorithm

With our expectations well-defined, we began developing the NOW-Sort code for a single workstation. Our implementation contains two distinct code paths, depending upon whether one or two passes are required to sort the records in available memory. We describe our experience tuning first the one-pass sort, and then the two-pass sort.

Our initial version of the one-pass, single-node sort contained three simple steps.

- **Read:** The program begins by opening the input file; the `read()` system call transfers the 100-byte records from disk into main memory.
- **Internal Sort:** A quicksort is performed over all of the keys and records in memory. Before the quicksort, each 10-byte key is separated from the 100-byte record and a pointer is set up to the full record. This separation allows the sort to operate more efficiently, swapping keys and pointers instead of full records during the quicksort [25].
- **Write:** The list of sorted keys and pointers is traversed in order to gather the sorted records into an output buffer. When the output buffer fills, it is written to disk with the `write()` system call. Finally, `fsync()` is called to ensure all data is safely on disk, and the file is closed.

Our measurements revealed that this initial implementation failed to meet our expectations in two ways. First, fewer records than expected could be sorted without paging to disk. Second, the time for the internal sort was a noticeable percentage of the total execution time. We briefly discuss how we located and fixed these two performance problems.

3.2.1 Optimizing Memory Management

To track down the unexpected paging activity, we monitored disk traffic while increasing the number of records sorted. We visually observed this traffic by implementing a small performance meter, `diskbar`, described in more detail below. With this tool, we found that when sorting more than 20 MB of records on a machine with 47 MB of available memory, a small amount of paging traffic occurred. As we increased the input size, the traffic steadily worsened, until the machine was thrashing throughout the sort and write phases.

We readily diagnosed the cause of this classic problem [31]: when using the `read()` system call, the file system was unnecessarily buffering the input file. When the size of the file reached half of available memory, the operating system was forced to discard something from memory; unfortunately, the pages that the replacement policy chose to discard were from the running sort program, rather than from the buffered (and now useless) input file. By switching to the `mmap()` interface and using `madvise()` to inform the file system of our sequential access pattern, we found that we could sort 40 MB of records without paging, as desired.

3.2.2 Optimizing Internal Sorting

To evaluate if we could shorten internal sorting time we monitored CPU utilization with `cpubar`, another home-grown visualization tool, while executing NOW-Sort. The low CPU utilization that we observed during the read phase implied that we could overlap part of the internal sort with the read phase. As suggested in [2], we performed a bucket sort while reading records from disk, followed by a partial-radix sort. With this optimization, we reduced the in-memory sort time for one-million records with four disks from 20% of the total execution time down to only 4%.

Cache Configuration Tool: To reduce the time spent stalled for memory, the keys touched in each partial-radix sort should be made to fit into the second-level cache. Therefore, the optimal number of keys in each bucket is determined by the size of the L2 cache. Our current implementation is optimized for the 512 KB L2 cache in the UltraSPARC-I. Running NOW-Sort with a different cache architecture would require two changes. First, the size of the second-level cache must be determined, either via look-up in a hardware parameter table, or with a cache configuration tool, similar to the micro-benchmarks described in [30]. Second, the sort must be trivially modified to accept the cache size as a parameter instead of as a hard-coded constant.

3.3 Two-Pass Algorithm

The two-pass version of NOW-sort leverages much of the code that was already optimized for the one-pass sort. At the highest-level, there are two phases:

- **Create Runs**: The application repeatedly reads in a portion of the input file, sorts it, and writes the sorted run out to disk; two kernel-threads are used to overlap reading and writing from disk. In effect, the one-pass sort is called multiple times, each time using half of available memory.
- **Merge Runs:** The sorted runs are read in from disk and merged into a single, sorted output file. First, each run created in the first phase is mapped into memory. Then, until all records have been written out to disk, the record with the lowest-valued key is picked from the top of each run and copied to an output buffer; when the output buffer fills, it is written out to disk.

3.3.1 Optimizing Disk Bandwidth

Measurements of our initial two-pass implementation revealed that creating the runs took the expected amount of time, but that merging was slower than desired. Observing the delivered read disk bandwidth (again, with the simple `diskbar` tool) quickly illuminated the problem: although we were reading from each sorted run sequentially, the layout of multiple runs per disk resulted in a disk seek between every read, significantly reducing the achieved bandwidth.

To achieve near-peak bandwidth from disks, NOW-Sort amortizes disk seek-time by carefully managing I/O during the merge; that is, a large amount of data is explicitly prefetched from one run before reading anything from the next. Empirically, we found that 512 KB to 1 MB prefetch-buffers were sufficiently large to achieve near-sequential performance.

Three kernel threads are required to successfully prefetch data while also writing to disk. A reader thread begins by copying 1 MB from each run into a merge buffer. A merger thread waits until a set of buffers are full and then copies the record with the lowest-valued key to an output buffer. After an output buffer fills, a writer thread writes the buffer to disk, marking the buffer empty when finished.

Seek Configuration Tool: In general, the buffer size for the merge phase should be calculated automatically as a function of the disk characteristics. Therefore, the disk configuration tool should return disk seek time, s, as well as peak disk bandwidth, B. In any application with non-sequential transfers, a simple calculation reveals the required size of the transfer, T, to obtain a desired fraction, f ($0 \leq f < 1$), of the peak disk bandwidth.

$$T = B \cdot s \cdot (\frac{f}{1-f})$$

3.4 Discussion

The ability to focus on single-node performance is an essential tool for both debugging and performance analysis on parallel machines. In NOW-Sort, we were able to optimize the cache, memory, and processor performance of the single-node algorithm and know that this behavior would hold in the parallel algorithm as well.

We found that automatic configuration is necessary for high-performance applications to adapt to a wide variety of system parameters. Adapting NOW-Sort to system parameters was simplified by concentrating on the behavior of a single node. Towards this end, we developed memory capacity and disk bandwidth configuration tools; cache and disk seek time tools might also be useful in a more dynamic environment.

These tools served a dual purpose: while they were useful in adapting the sort to varying hardware parameters (differing speed disks, differing amounts of memory), they also were crucial in setting our expectations. Without these expectations, we would not have been able to determine if we were achieving acceptable transfer rates from the disks, or whether we were making effective use of available memory. While traditional tools report where the code is spending its time, they do not report when to stop tuning. For example, profiling our final code would inform us that our code is I/O bound; for NOW-Sort this implies that our tuning is completed, whereas for most applications this would imply that more tuning is needed!

Identifying performance problems on the single node was simplified with feedback from a small set of visualization tools that we implemented. The `cpubar` tool displays the current CPU utilization, divided into user, system, wait, and idle time. `diskbar` displays delivered bandwidth from each local disk, including the swap partition. While current text-based tools, such as `top` and `iostat`, report similar information, they do not update statistics more than once per second. We found update rates on the order of five to ten times a second were necessary to capture the bursts of resource usage. The CPU tool in particular is more useful than its textual counterpart because it enabled us to visualize the summation of multiple components while simultaneously looking for anomalies. As simple as these tools are, we found them incredibly useful at giving an instantaneous and continuous view of system behavior; today, we rarely run applications without concurrent monitoring.

Tuning the in-memory behavior of the sort was easier than expected, largely due to the development of cache-sensitive algorithms by previous researchers [2, 25]. Early in our experience, we ran the Shade instruction set simulator to evaluate our implementation [9]; however, largely because we could not easily match the reported statistics back to specific lines of our user-level code, we were not able to use this information for tuning. Further, Shade does not trace kernel code, which has important interactions with the NOW-Sort algorithm.

If our internal sort had consumed a larger fraction of the total execution time, then tools specifically aimed at finding memory bottlenecks, such as Cprof [18] and MemSpy [21], could have been useful for fine-tuning. However, for codes that interact with underlying software systems (especially the OS and communication layer), it is crucial that these tools give information on more than just user-level code. Modifying tools such as these to work in tandem with a complete machine simulator (*e.g.*, SimOS [29]) would thus be more useful.

4 Parallel Performance

By focusing on single-node performance, we developed an excellent building block for a scalable sorting algorithm. Our next step towards large-scale parallelism was to define a set of expectations for parallel performance. After defining these expectations, we implemented the algorithm on a small number of machines. On this small cluster configured with a full set of disks, we found that our implementation did not meet our initial expectations.

In the single-node version of NOW-Sort, when we found discrepancies between our expectations and our measured performance, we found and removed inefficiencies in our implementation. However, in the parallel version, the differences we found were due to unrealistic expectations, based on an overly simplistic model of the system. By refining our expectations by measuring a small cluster, we were able to determine the appropriate hardware configuration for the entire cluster of 105 machines.

4.1 Defining Expectations

In the ideal case, each process executing in the parallel application sorts the same number of records in the same amount of time as the single-node version. That is, the scalability of NOW-Sort should be perfect as the problem size is increased linearly with the number of workstations.

Microbenchmarks of Active Message communication performance indicate that 35 MB/s can be transferred between nodes [11]. The implication of this level of bandwidth is that sending a set of records to another node requires noticeably less time than reading those records from disk. Therefore, we add a "parallel" expectation to the four enumerated for the single-node sort:

5. Overlap communication with other phases of the sort.

In the next sections, we present our algorithm that attempts to hide the cost of communication under the read phase of the sort, and then discuss why this expectation could not be met as the amount of disk bandwidth per node increased.

4.2 Algorithm Design

To reduce the complexity of tuning our parallel sort we chose a programming model that provides explicit control over the operations performed on each node (single-program-multiple-data, or SPMD). Under the SPMD programming model and Active Messages, the one-pass and the two-pass parallel sorts were direct extensions to the single-node algorithms. The ease of our design was due largely to the Active Messages paradigm, which allows communication to be integrated into the on-going computation in a natural fashion.

The only change from the single-node versions occurs when records are initially read from disk. As a record is read from disk, it is range-partitioned across the set of workstations; each machine receives a roughly equivalent fraction of the data, with workstation 0 receiving the lowest-valued keys, workstation 1 the next lowest, and so forth. Records are temporarily buffered before transmission so that multiple records can be sent with a single Active Message, achieving better communication bandwidth. Upon receipt of a message, the Active Message handler splits the keys into buckets, and copies each record to a record array. After this, the internal sort and the write steps, as well as the entire merge phase of the two-pass sort, are identical to the single-node versions, and require no communication amongst the nodes.

4.3 Modifying Expectations

Our evaluation of both the one- and two-pass parallel sorts began with measurements on a four-node cluster of UltraSPARC workstations with one disk each. The initial experiments revealed the desired properties of our algorithm: communication costs completely hidden under I/O wait time, no unnecessary I/O, no paging, data transferred from disk at peak bandwidth, and negligible internal sorting time. However, as we added more disks to the cluster, we found that the performance did not increase as expected.

We found the most direct method for isolating a performance problem was to run the application, and use hardware performance counters to measure the behavior of the program. The difficulty of this methodology in our environment is that the Ultra workstation has only CPU performance counters, which cannot capture all memory traffic or any I/O bus traffic. Our solution was to employ a more sophisticated machine, the UltraEnterprise 5000, which has an array of performance counters on both the memory bus and I/O bus. Though the machine is an 8-processor SMP, we shut down all but one of the processors and utilized the machine as an ordinary workstation with extraordinary counters.

Running the sorting code with counters produced a "performance profile" revealing that with more than two disks per workstation, both the CPU and I/O bus are saturated [6]. The CPU reaches its peak utilization due to the cost of initiating reads from disk while simultaneously sending records to remote nodes and copying keys into buckets. The I/O bus saturates from handling disk, network send, and network receive traffic; for every byte read from disk, roughly three bytes must travel over the I/O bus. Once these architectural bottlenecks are reached, there is no benefit to overlapping reading from disk with communication. Thus, communication costs cannot be hidden when there are more than two disks per workstation, and our performance expectations must be mitigated.

Our previous models, based on a more naive understanding of the machine architecture rather than actual measurements, did not foresee these bottlenecks. In fact, we originally projected that four disks per machine would be an ideal purchase for the entire cluster. However, by running on a small number of well-configured machines, we were able to understand the architectural limitations of our hardware environment, and correctly shape the future hardware purchases for the entire 105-node cluster.

To sort the most keys for the least cost, a one-pass sort should be used and the disk system should be able to fill memory in half the desired sort time (*e.g.*, for MinuteSort this is 30 seconds). Since two disks give the best cost-performance, each workstation in our system can read or write at approximately 8 MB/s. Therefore, each machine should contain enough memory for roughly 240 MB of records. Accounting for the memory requirements of the operating system and the application, each workstation should have 325 MB of physical memory. As a result of these calculations, the hardware of the entire cluster was upgraded to two disks per workstation, but to only 128 MB of memory per workstation, due to budget (actually donation) limitations. With insufficient memory for a one-pass attempt at the MinuteSort benchmark, most of our measurements continued to rely on the two-pass sort.

4.4 Discussion

Accurate models and predictions of performance are necessary to correctly purchase hardware for large systems. However, modeling a memory- and I/O-intensive parallel program is a non-trivial task. We were able to accurately predict resource usage by running NOW-Sort on a small, well-configured cluster, and thus could gauge worthwhile hardware purchases for the entire system.

Such models could be easily constructed from a more comprehensive set of hardware counters at all levels of the machine. Though most modern processors have a reasonable set of performance counters [12, 22] that have been shown to be useful for detailed performance profiling [27], other components of the machine are ignored. For example, researchers have shown that network packet counters can be extremely useful [8, 20]. However, just monitoring in-coming and out-going packets is not enough. Minimally, 32-bit counters should be available for every interconnection in the system, from the memory and I/O bus of each workstation, out into the switches of the network. These counters should track the number of bytes that pass through the monitored part of the system, as well as the resource utilization. More specific information per device would also be helpful. For example, disks should report seeks and transfers per second and include tags to differentiate traffic (*e.g.*, one PID from another).

Counters that allow tracing of short periods of activity would facilitate detailed studies of portions of the code. Collecting all of this data, however, is not of much use without techniques for feeding the results back to the user. Standard performance monitoring tools (such as `cpubar` and `diskbar`) would be the first step, allowing the user to see bottlenecks in the system; more advanced systems could use the counters to pinpoint troublesome sections in the program of interest.

5 Scaling the System

After our expectations had been set and fulfilled on a small number of machines, we were ready to scale NOW-Sort to the full cluster size. In this section, we discuss the difficulties encountered when scaling up to 105 processors. In general, we had little difficulty debugging or scaling the algorithmic aspects of the sort. Most of the challenge arose from the scaling characteristics of the underlying sub-systems, such as the distributed operating system and communication layer. With these sub-systems, we found that performance did not slowly degrade with more workstations, but instead fell off a cliff; this radical drop in performance usually occurred around 32 workstations – the cluster size at which system developers had stopped their evaluations.

5.1 The Distributed Operating System

The Datamation and MinuteSort benchmarks specify that the start-up time of the application must be included in all measurements. This essential task within GLUnix [14] was a bottleneck on large cluster sizes. While we encountered start-up times usually between one and two seconds on medium-sized clusters of 32 nodes, starting a job on more than about 40 nodes often required between 20 and 30 seconds!

Together with the GLUnix developers we tracked down this interesting performance bug. To set the group ID of the job, each GLUnix daemon was performing an NIS request to a centralized NIS server. For a large, parallel job, this meant that many clients were simultaneously requesting data from the NIS server over a shared Ethernet, which responds to each client sequentially. Removing the call to the NIS server was trivial: a local system call to set the group ID was all that was necessary. Thus, we had found what we thought was the cause of our performance problem.

Unfortunately, removing the group-ID bug *slowed down* the start-up time. Further manual instrumentation of the code revealed a new bottleneck when each new member of the parallel program set up TCP connections with the start-up process. To redirect standard input, output, and error in GLUnix, each client in an P-way parallel job opens three socket connections to the home node. For large parallel jobs, the burst of $3 \cdot P$ socket-connection requests overflowed the default socket-accept queue of 128 entries. This overflow forced some of the clients to timeout and retry the connection multiple times, drastically increasing the start-up time for large applications. The time-out and retry did not occur previously because the NIS request serialized this portion of the code; with the serialization removed, the new bottleneck appeared. Fortunately, the size of the Solaris TCP accept-queue can be readily modified. Removing these two bottlenecks in GLUnix improved start-up time into the acceptable (but not ideal) range of a few seconds.

5.2 The Communication Layer

Running the NOW-Sort application on large configurations also stressed the communication sub-system in new and distinct ways. Two basic issues arose: errors in the communication hardware and insufficient bisection bandwidth. In general, problems in the communication layer were difficult to isolate because they were almost indistinguishable from performance problems within the application itself.

The Myrinet local-area network was designed to provide an "MPP-like backplane" in a local area network setting [7]. In its goal to export a low-overhead, high-throughput communication layer, the design of Generic Active Messages (GAM) for Myrinet assumed complete hardware messaging reliability. On medium-sized clusters, this assumption was reasonable, and bit errors were rarely encountered. However, on 105 processors (the largest existing Myrinet cluster to this day), bit errors occurred on many runs of the NOW-Sort application. The errors manifested themselves as CRC errors, which the communication layer recognized, but could not recover from, thus terminating the application.

The current version of Active Messages performs reliable message transfer on top of this usually reliable medium [19]; however, the temporary solution in GAM was to reduce the number of outstanding messages allowed in the flow-control layer, limiting the load placed on the network. While this solution had the desired effect of reducing the chance of bit errors, it had the disastrous side-effect of almost halving the achievable bandwidth. Even worse, this change was made without our knowledge and shortly before the April 1st deadline for sorting results. Though we could have theoretically sorted 9 GB in 30 seconds, we sorted "only" 8.41 GB in 59.7 seconds, which stands as the current MinuteSort record.

On large configurations, we also saw that the bisection bandwidth of the network was insufficient for the all-to-all communication pattern of NOW-Sort. For development reasons in earlier phases of the project, it had been useful to have three distinct clusters each containing roughly 32 machines. While the network topology had been designed such that the bandwidth within each of these three clusters was sufficient, there were insufficient links across clusters.

Tracking down this bandwidth deficiency in the network was more difficult than we anticipated. First, due to deadline pressures, we did not sufficiently micro-benchmark the network before running NOW-Sort. Therefore, we assumed that the larger network continued to meet the expectations we had developed on the smaller cluster. Second, there is currently no way to observe the utilization of the links and switches internal to the network, since there are no performance counters. Consequently, we were not able to readily differentiate poor performance due to insufficient bandwidth and poor performance due to algorithmic characteristics. Once found, though, adding more links (and thus bandwidth) solved the problem.

5.3 Discussion

From these difficulties, we learned a number of valuable lessons. First, application writers should be able to debug programs running on a large number of nodes in an interactive environment – it is not realistic to assume that programmers will be able track down all bugs on a smaller number of nodes. Second, even though the NOW-Sort algorithm was scalable, in many cases, the underlying system was not. In large-scale systems, system designers must have access to tools that are normally restricted to application-level programs; without such support, system-level performance tuning is an ad hoc, laborious process. Parallel profiling tools, such as Quartz [4], modified for our cluster environment, would have been useful for the developers of GLUnix. A tool such as Paradyn [24] would also have been helpful for avoiding lengthy recompiles by allowing dynamic instrumentation of "interesting" pieces of code and providing immediate visual feedback.

Isolating performance problems is particularly frustrating in a dynamic environment, where libraries, daemons, and other system services change without altering the application binary. In our case, a one-line change in the Active Message library lead to devastating performance results. Tools that understand the complete dependence tree of an application and notify the user when changes occur would be helpful in developmental systems; perhaps configuration management tools are a good match to this problem [16]. One difficulty with this approach is following the chain of dependencies. For example, in our environment, applications link with the GLUnix library to access GLUnix. Though the GLUnix library may not have changed, the GLUnix user-level daemon, which is contacted through the library, may have.

Finally, large-scale systems must be commonly tuned on the "target" system size. In our research environment, the largest common platform for testing was usually about 32 machines, due to contention for resources across developers. As a result, the system behaved well up to that size. However, we quickly found our system did not scale to the desired full cluster size. The implication for system development groups is that when investigating how to build systems consisting of N workstations, the entire system should contain $C \cdot N$ workstations, where C is greater than 1.

6 Needle in the NOW-Stack

Once the system has been scaled, the final step in chasing the sorting record was a successful, unperturbed run of the sort. The performance of NOW-Sort is quite sensitive to various disturbances and requires a dedicated system to achieve "peak" results. In this section, we discuss how we solved a set of particular run-time performance problems where a foreign agent (such as a competing process, inadequate memory, or a full disk) on one or more machines slowed down the entire application. In general, this was a hierarchical process, similar to the search process described in [15]. In the first step, "finding the needle in a NOW-stack", the slow workstations in the cluster were identified. In the second step, "removing the needle", the specific problem on those slow workstations was identified and solved.

6.1 Finding the Needle

Because each process in our parallel sort performs the same amount of work, any one machine with a disturbance slows down the entire application. When NOW-Sort exhibited lower than expected performance, we isolated the nodes with disturbances in one of two ways. First, we ran a GLUnix status tool, `glustat`, that reports the load average and available virtual memory for each workstation in the cluster; nodes with anomalous values were easily singled out. If all nodes appeared homogeneous to GLUnix, we ran NOW-Sort again with additional timing statistics displayed for each workstation and every phase. At least one of the workstations was always noticeably slower during some phase, pointing us to the machine with the needle.

6.2 Identifying and Removing the Needle

After the slow machine was isolated, the key was to find what was *different* about that machine compared to the others in the cluster. The four most common "needles" we found on machines were faulty hardware, extra processes competing for the CPU, less available memory, and extra data on disks.

Faulty Hardware: The first needle that we encountered consisted of machines with faulty or misbehaving hardware. Before beginning, we were aware that a few machines in the 105-node cluster were unavailable, due to faulty power supplies or motherboards. However, some of the machines that had been adequate for running compute-intensive, sequential processes had either network-interface or disk problems. Thus, in our cluster of 105 possible machines, we found that only 95 were usable for NOW-Sort.

CPU Stealing: The most trivial and most common needle was a foreign job running on one of the workstations. Competing against another process obviously slows down the sorting process on the workstation. Once the slow workstation was identified, standard process monitoring tools such as `ps` and `top` identified the culprit to be killed.

Memory Hog: Another common performance problem was caused by a difference in free memory across machines. In our most notable needle scenario, the GLUnix master daemon had been placed on one of the workstations running the sort, instead of on a separate administrative machine. Though the CPU requirements of the master were negligible, the memory footprint was not. As a result, the sort process running with the master began to page, interfering with the sequential I/O activity. Tracking down this aberrant GLUnix process proved more difficult than would be expected, due to the small differences involved.

Disk Layout: The performance of NOW-Sort is quite sensitive to the layout of files on each local disk. Because multi-zone disks give significantly higher bandwidth when data is allocated on outer tracks [23], full disks lead to less than expected data rates. Our disk configuration tool verified that read performance varies significantly (30%), depending on disk layout. Several times, after tracking down a slow node, a quick inspection revealed a scratch disk full of another user's temporary files. After we removed the files, the sort performance returned to its expected level.

6.3 Discussion

Our hierarchical methodology of first isolating problematic workstations and then identifying specific disturbances required performance isolation across workstations. The hard boundary between workstations allowed us to "drill-down" and focus on the performance of a particular component, whether it was a CPU, memory system, or disk. Other large-scale systems where resources are aggregated behind a single interface, such as SMPs or RAID-style disk systems, do not lend themselves naturally to this hierarchical approach.

Our strategy allowed us to successfully solve performance problems, but required a fair amount of intelligent user intervention. A more automatic approach is clearly desirable, especially for users not as familiar with all components of the system. For example, problematic workstations could be flagged with a more sophisticated tool, such as Performance Assertion Checking [26]. With such a tool, simple assertions about performance expectations are placed in the code, and flagged when they do not succeed.

While standard parallel tools could have been useful in the search process, some limitations prevented our use of them. Most of these limitations arise due to the "advanced" features of Solaris that we leverage: kernel threads and memory-mapped files. For example, Paradyn does not work with threaded applications, which eliminates all but our single-node, one-pass sort [24]. Our codes also make heavy use of `mmap()` to access files, whereas many tools only track the `read()` and `write()` system calls when reporting I/O behavior. One such system, Pablo, tracks read and write calls, and uses them to give breakdowns of I/O requests, sizes, and times of access [28]. With `mmap()`, this is much more difficult, requiring the instrumentation of all loads and stores to mapped file regions.

Identifying specific problems was simplified with a few visualization tools that we developed, as described in Section 3. The only problem with this feedback is that we could not run the meters on all 95 nodes, due to the large amount of X-window traffic that they generate. A more advanced system could collect this data locally on each node, and then collate it and present it as an animation afterwards. Likewise, the specific problem could be automatically identified by constantly monitoring the processor, memory, and disk utilization on each machine, reporting anomalies to the user.

Once found, fixing the problems was usually easy for us because we have a reasonable understanding of all of the components of our system. However, users less familiar with the Berkeley NOW cluster would have a more difficult time isolating anomalies. Further, fixing the problems sometimes required interfering with the rights of other users (*e.g.*, killing jobs and removing files). Not only do most users not have permission to perform such actions, it seems unlikely that this process could be safely automated.

7 Conclusions

The paper has presented our experiences of writing and tuning NOW-Sort, the current world-record holder in external sorting. Crucial to attaining peak performance was the setting of performance *expectations*, which allowed us to find and fix performance problems in the sort. We believe that tools to help users set expectations are of great value; our case was probably simpler than most, in that the main expectation was to achieve peak disk bandwidth.

Throughout the process, we made use of a number of tools to help attain peak performance. Configuration tools helped us set our expectations, as well as automate parameterization of the sort to various disk systems and memory sizes. Visualization tools gave us instant visual feedback on the performance of the sort, allowing quick deductions as to the location and possible cause of performance problems. Finally, hierarchical search tools would have been helpful in automating the search for performance anomalies.

In the end, with a mix of simple tools and ad hoc practices, we achieved the performance we desired. On the most challenging external sorting benchmark, MinuteSort, we sorted a total of 6 GB within a minute on 64 nodes using two passes. Thus, we delivered over 409 MB/s of sustained I/O performance to the sort application, roughly 80% of the peak I/O rate that could be delivered in that time. The final record we achieved was 8.41 GB in just under a minute on 95 nodes, this time in one pass. The delivered I/O bandwidth to the application is not as high in this case, due to GLUnix start-up difficulties and a one-line change in the underlying communication layer. Perhaps this final, imperfect total serves as an appropriate reminder as to the nature of our journey.

Acknowledgments

NOW-Sort would not have been possible without the combined effort of many members in the Berkeley NOW Project; we thank all of those who contributed to our success, especially Rich Martin and Doug Ghormley. We extend special thanks to Jim Gray and Chris Nyberg for sparking our interest in sorting, and for all of their encouragement and sage advice along the way. Finally, we thank our shepherd, Margaret Martonosi, and the anonymous reviewers, for many pieces of 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. Remzi Arpaci-Dusseau is supported by an Intel Graduate Fellowship.

References

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. K. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of I/O intensive parallel applications. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 15–27, Philadelphia, May 1996. ACM Press.

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

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

[4] T. E. Anderson and E. Lazowska. Quartz: A Tool for Tuning Parallel Program Performance. In *Proceedings of the 1989 ACM SIGMETRICS and PERFORMANCE Conference on Measurement and Modeling of Computer Systems*, pages 115–125, May 1989.

[5] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-Performance Sorting on Networks of Workstations. In *SIGMOD '97*, May 1997.

[6] R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. The Architectural Costs of Streaming I/O: A Comparison of Workstations, Clusters, and SMPs. In *HPCA '98*, February 1998.

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

[8] J. Chapin, S. Herrod, M. Rosenblum, and A. Gupta. Memory System Performance of UNIX on CC-NUMA Multiprocessors. In *1995 ACM SIGMETRICS/Performance Conference*, pages 1–13, May 1995.

[9] B. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator For Execution Profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference*, pages 128–37, May 1994.

[10] 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.

[11] D. E. Culler, L. T. Liu, R. P. Martin, and C. O. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, 2/1996.

[12] DEC. DECChip 21064 RISC Microprocessor Preliminary Data Sheet. Technical report, Digital Equipment Corporation, 1992.

[13] 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.

[14] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson. A Global Layer Unix for a Network of Workstations. *To appear in Software Practice and Experience*, 1998.

[15] J. K. Hollingsworth. *Finding Bottlenecks in Large-scale Parallel Programs*. PhD thesis, University of Wisconsin, Aug. 1994.

[16] J. K. Hollingsworth and E. L. Miller. Using Content-Derived Names for Configuration Management. In *1997 ACM Symposium on Software Reusibility*, Boston, May 1997.

[17] 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.

[18] A. R. Lebeck and D. A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE COMPUTER*, pages 15–26, October 1994.

[19] A. M. Mainwaring. Active Message Application Programming Interface and Communication Subsystem Organization. Master's thesis, University of California, Berkeley, 1995.

[20] M. Martonosi, D. W. Clark, and M. Mesarina. The SHRIMP Hardware Performance Monitor: Design and Applications. In *Proceedings of 1996 SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, February 1996.

[21] M. Martonosi, A. Gupta, and T. E. Anderson. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proceedings of the 1992 ACM SIGMETRICS and PERFORMANCE Conference on Measurement and Modeling of Computer Systems*, pages 1–12, May 1992.

[22] T. Mathisen. Pentium Secrets. *Byte*, pages 191–192, July 1994.

[23] R. V. Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the 1997 USENIX Conference*, Jan. 1997.

[24] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11), 1995.

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

[26] S. Perl and W. E. Weihl. Performance Assertion Checking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 134–45, December 1993.

[27] S. E. Perl and R. L. Sites. Studies of Windows NT Performance Using Dynamic Execution Traces. In *OSDI 2*, pages 169–184, October 1996.

[28] D. A. Reed, C. L. Elford, T. Madhyastha, W. H. Scullin, R. A. Aydt, and E. Smirni. I/O, Performance Analysis, and Performance Data Immersion. In *Proceedings of MASCOTS '96*, pages 5–16, February 1996.

[29] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modelling and Computer Simulation (TOMACS)*, January 1997.

[30] R. H. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*. PhD thesis, U.C. Berkeley, Computer Science Division, February 1992.

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

[32] 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.

[33] M. Tremblay, D. Greenley, and K. Normoyle. The Design of the Microarchitecture of UltraSPARC-I. *Proceedings of the IEEE*, 83(12):1653–63, December 1995.

[34] 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.