Computer Sciences Department

Datamation 2001: A Sorting Odyssey

Florentina Popovici
John Bent
Brian Forney
Andrea Arpaci-Dusseau
Remzi Arpaci-Dusseau

Technical Report #1444

August 2002

UNIVERSITY OF
WISCONSIN
MADISON

Datamation 2001: A Sorting Odyssey*

Florentina I. Popovici John Bent Brian Forney
Andrea Arpaci-Dusseau Remzi Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin-Madison

Abstract

We present our experience of turning a Linux cluster into a high-performance parallel sorting system. Our implementation, WIND-SORT, *broke the Datamation record by roughly a factor of two, sorting 1 million 100-byte records in 0.48 seconds. We have identified three keys to our success: developing a fast remote execution service, configuring the cluster properly, and avoiding the potential ill-effects of occasionally faulty hardware.*

1 Introduction

When Datamation [1] was introduced in 1985, sorting 95 Mbyte of data was a time consuming endeavor. Advances in both computing power and software algorithms have reduced the time of the benchmark by three orders of magnitude (see Figure 1), considerably altering the nature of the Datamation sort. Whereas originally intended as an I/O- and memory-intensive benchmark, Datamation is now best considered a benchmark of interactive I/O performance.

Although Datamation's importance as a database benchmark has diminished, we believe the benchmark remains relevant for three reasons. First, Datamation stresses the importance of start-up time. Start-up time has been noted as one of three factors limiting performance of parallel systems [7], and has been a problem for both clusters [2] as well as SMPs [8]. Second, Datamation is an example of an interactive parallel application. For parallelism to become commonplace, interactive applications must become a reality. Third, interactive parallel jobs are particularly sensitive to performance fluctuations that occur in large-scale systems [3]. Thus, if the Datamation sort runs consistently well upon a given platform, conclusions can be drawn on the ability of the platform to avoid or tolerate the presence of such perturbations.

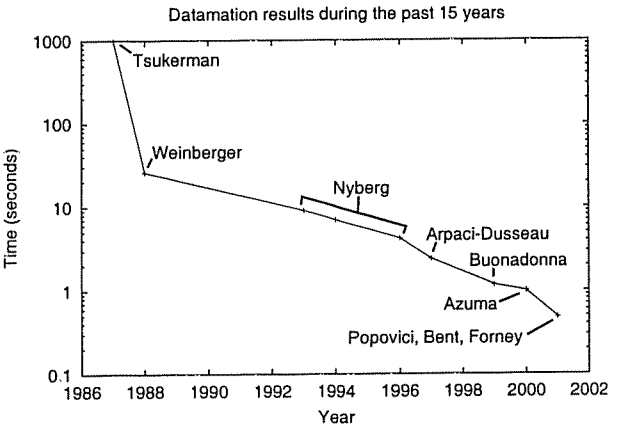

Figure 1: **15 years of Datamation.**

In tuning the sort for our Linux-based cluster-of-PCs platform, we found that three elements were crucial in achieving consistent high-performance. First, we developed a lean parallel remote execution layer in order to minimize start-up time. Second, we configured Linux properly, by enabling SCSI tagged command queueing, altering an overly-pessimistic ARP caching scheme, and increasing socket buffer sizes so as to avoid message-buffer overflows. In all of these cases, the default values were not desirable and led to performance problems that were both unexpected and sometimes difficult to discover. Third, we altered the communication layer to more aggresively resend packets in order to overcome an occasionally faulty

*This technical report describes work completed in the spring of 2001.

network switch module. These *proactive* re-sends are particularly important for jobs that run for extremely short periods of time.

The remainder of the paper is organized as follows. In Section 2, we discuss our base sorting code and in Section 3 the hardware and supporting software. Our efforts to understand and optimize performance for startup, disk I/O, and network I/O are in Section 4. The integration of these efforts into a record breaking sort occur in Section 5. Our conclusions are in Section 6.

2 Background

This section describes the sort algorithm we used. The implementation is derived from NOW-Sort [2], which assumes that the input records are evenly distributed across P nodes, numbered 0 through $P-1$. The sorted output file is range-partitioned across the node such that the lowest-valued keys are on node 0 and highest-valued keys on node $P-1$; however, the number of records per node will only be approximately equal and depends on the exact distribution of key values within the input files.

The algorithm consists of four steps. In the first step, each of the nodes opens its input file and reads from the file in fixed-sized buffers. After each buffer is read, we copy each record into a send-buffer allocated for each destination node. As in NOW-Sort, the destination node is determined by the top so-many bits of each key; this approach assumes that the key values are uniformly distributed.

In the distribution phase, each node sends the records it read to the node that will sort records within its range-partition. At the receiver site, the keys are separated from their records and placed in one of $B = 2^b$ buckets based on the next b high-order bits of the key; a pointer is kept to the full 100-byte record stored in memory. This step ends with a synchronization across all nodes to ensure that all records have been sent and received.

In the sort step, each node performs an in-core sort on its records. A conventional quicksort algorithm sorts the data.

In the final phase, each node follows the sorted keys and pointers to gather the records into a contiguous buffer. The sorted keys and accompanying records are written to a local file. The concatenated output files on all nodes represent the sorted version of the concatenated input files.

There are two main differences from NOW-Sort that are both fall-outs of our emphasis on the Datamation sort. First, NOW-Sort and other sorts previously used overlap between the read and distribution phases. We discovered that due to the small amount of data per node there is no benefit to overlapping these two steps of the algorithm – the cost of starting the extra threads render this technique undesirable for small amounts of data. Second, we utilize a quicksort instead of a highly specialized partial radix/bubble sort hybrid. The differences among these algorithms at small data sizes is negligable, and therefore we chose to use the conceptually simpler approach.

3 Environment

3.1 Hardware

WIND-SORT runs on a 32 node Linux cluster. Each node contains two Intel 550 MHz Pentium III processors with 896 Mbyte of available memory and five 8.5 GB IBM 9LZX disks. The disks are configured on two Ultra II SCSI buses, rated at 80 Mbyte/s each. Three disks, including the operating system root disk, are on one bus. The remaining two are connected to the other bus. WIND-SORT's data files reside on the four non-root disks.

The nodes are networked by 100 Mbit and gigabit Ethernet. The gigabit Ethernet is a private network for the cluster. The 100 Mbit network provides job launching from an external system and a link to the departmental network. The gigabit Ethernet cards are Intel PRO/1000 F Server Adapters. A 32-port Intel NetStructure 6000 Gigabit Ethernet switch connects all of the gigabit NICs.

3.2 System software

Our cluster's system software is primarily commodity-based, with a few additions from the research community, as well as the development of our own job launching software. Each node in our cluster runs Linux 2.2.19 and uses the ext2 file system and Linux RAID software. Jobs are started in parallel using our own software, `ice`, described in more detail in the next section. Our sorting code also depends on two pieces of research software: Split-C and a UDP-based implementation of Active Messages. Split-C [6] is a parallel extension to C that supports efficient access to a global address space on distributed memory machines. For communication, Split-C uses Active Messages [9], a restricted, lightweight version of remote procedure calls. Since Active

Messages in our cluster is layered upon UDP [4], the implementation is not as efficient as previous highly-tuned timplementations.

4 From 0.996 to 0.48

4.1 Startup cost

The launch time of a parallel program on a cluster can add considerable overhead to the program run time. We found that the currently available job launch packages impose considerable overhead. REXEC and SSH are two common packages which we initially considered. Launching a null job, which exits immediately, on a remote node took 180 ms for REXEC and 920 ms for SSH. In measurements not shown here, we found that REXEC and SSH do not scale well as more jobs are launched. These numbers reveal the need for a high performance launcher. For our sort, we developed a special launcher called ice (Interactive Cluster Execution).

An ice daemon runs on each node of the cluster, and is responsible for receiving messages from clients and spawning jobs in response. To launch a parallel job, a user can simply type: `ice <nodelist> <port> <job> [arguments]`. The ice command contacts ice daemon on the specified node or nodes requesting the launch of a job. The request includes the environment variables of the ice command. In response to a request, each ice daemon creates, by default, connections back to the requesting ice command to redirect program output and to control the program. After creating the connections, each ice daemon forks a new process and waits for the completion of the job. At completion the exit status of the remote job is sent back to the spawning node.

In addition to the basic operation described, ice allows for several optimizations. First, the program output connection can be turned off. The creation of this connection can be expensive as TCP is used. WIND-SORT only used this optimization after debugging was completed. Second, the control connection can use UDP instead of the default of TCP. Finally, the ice daemon can cache environment variables and hostname to IP translations. This caching scheme works well for our static environment. The results of each of these optimizations is shown in Figure 2. ice with optimizations minimizes the launch time of WIND-SORT to less than 80 ms on a 32 node cluster.

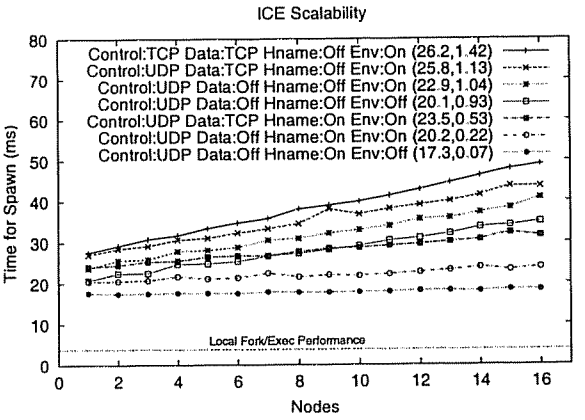

Figure 2: **Ice scaling comparison.** *This graph plots the start-up performance of the ice remote execution layer. Each line has the following four parameters: control protocol (TCP or UDP), data protocol (TCP or off) for program output, hostname cache (off or on), and environment cache (off or on). The numbers in parentheses specify the parameters of a best-fit line through the data points (the b and a of $y = ax + b$).*

4.2 Disk I/O

To understand disk I/O in our cluster, we gathered disk performance data over a range of file sizes. All data was gathered using software RAID 0 with 8 KB stripes and four disks. Reads were performed in 3600 byte chunks, while writes were performed in 40,000 byte increments. These sizes were chosen for coding convience after determining Linux 2.2.19 is relatively insensitive to I/O chunk sizes. The file system caches were flushed before each measurement. These results are shown in Figure 3 for reads and Figure 4 for writes. The figures only show the data for less than 10 Mbyte of data, which is the area of interest for Datamation on a 32 node cluster.

We made four observations from the data we collected. Read bandwidth across the four disks nearly reaches the aggregate rated performance of 77.6 Mbyte/s. The small amount of transfered data in Figure 3 does not show this due to software and hardware overheads. In other measurements, not shown here, we found that read bandwidth peaks at 74.4 Mbyte/s when the amount of data being read exceeds 90 Mbyte. We also observed the addition of more disks increases performance. With the small amount of data transfered during WIND-SORT (approximately 3 Mbyte per node) we were concerned that adding more disks might actually reduce performance. Performance can decrease due to the overhead of striping, due to variations in perfor-

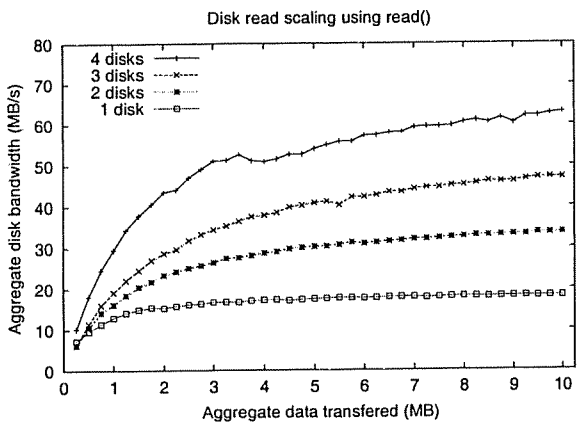

Figure 3: **Disk read performance scaling.** *This graph shows how disk performance scales for data transfered and number of disk used. Read performance scales well. Each point represents the average of 30 trials.*

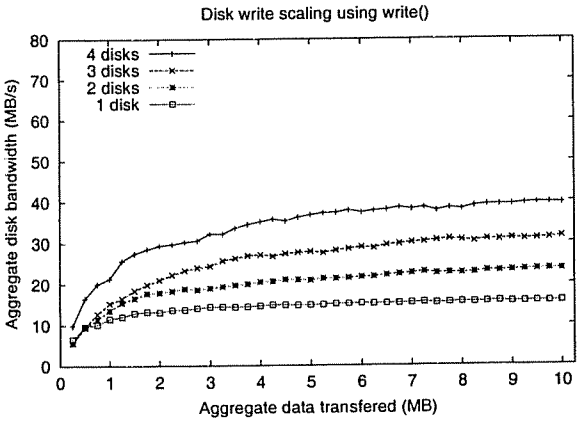

Figure 4: **Disk write performance scaling.** *Write performance with SCSI tagged command queuing enabled, as shown in this graph, scales to less than half the expected bandwidth of 80 Mbyte/s. Each point represents the average of 30 trials.*

mance characteristics between the disks or even due to such a seemingly minor factor as the location of the disk arm when the read begins. This was not the case in WIND-SORT but is seen in very small transfers in Figures 3 and 4 when the amount of data transfered is less than 0.5 Mbyte.

In terms of writes, we found write performance for four disks was much lower than expected. The manufacturer rates the disks at 19.4 Mbyte/s for sustained transfers from the outer tracks [5], yet the disks originally delivered only 6 Mbyte/s of bandwidth.

Several attempts were made to improve this performance. One of these was enabling SCSI tagged command queuing in the SCSI driver. Tagged command queuing allows more than one outstanding SCSI request per device for better request scheduling. Performance increased for a single disk from 6 Mbyte/s to 18 Mbyte/s. (Figure 4 has tagged command queuing enabled.) However, even with tagged command queuing enabled, write performance does not scale as disks are added; write bandwidth for four disks is less than 40 Mbyte/s. Using user-level striping did not show significant differences in performance from the kernel-level RAID software. Additionally, efforts to determine if the disk write limitation was due to the kernel's structure and policies revealed little[1].

Unlike NOW-Sort, our disk I/O performance was not improved by the use of implicit I/O. WIND-SORT used explicit I/O using the `read()` and `write()` system calls rather than performing implicit I/O using `mmap()`. Through measurement, we found that aggregate read performance using `mmap()` sustained about half of the bandwidth seen using `read()`. The Linux 2.2 implementation of `mmap()` is not as well-tuned as other Unix implementations; for example, it lacks `madvise()` which is used in other systems to provide hints to the virtual memory system that can trigger page prefetching.

4.3 Network

The distribution time was the largest component of our sort times. In the ideal case of a node-to-node transfer using UDP, our network delivered 37.5 Mbyte/s as shown in Figure 5. When all-to-all communication occurs in a UDPAM-based performance test, the network delivered roughly half of that rate per-node. Unfortunately, both node-to-node and all-to-all are much less than the theoretically possible 125 Mbyte/s peak bandwidth. We discovered that the bottleneck is receiver overhead; the receiver utilizes one processor completely while the sending processors are mostly idle[2].

Figure 7 shows the scaling of the sort's all-to-all distribution phase. The lowest of 30 trials at each point is shown in the figure. This figure shows

[1]Running NetBSD on a cluster node produced the same results. The only common components when NetBSD and Linux when run on our cluster are the core of the SCSI driver and the hardware. It is therefore our suspicion that the limitation is due to one of these two components.

[2]The cluster's gigabit Ethernet NICs do not support jumbo frames. Jumbo frames may improve the performance of the cluster's network.

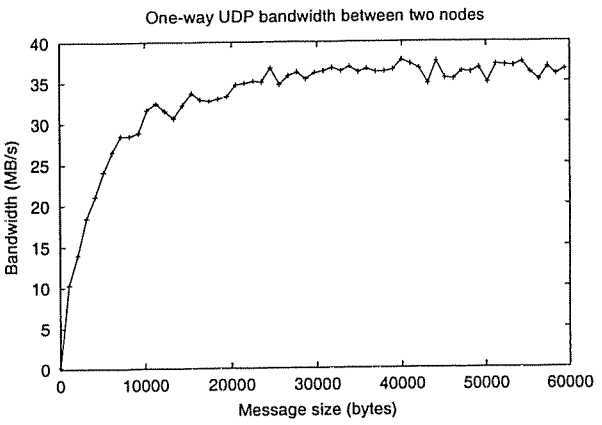

Figure 5: **One-way UDP performance.** *This graph shows the achieved bandwidth when sending 10,000 messages from one node of our cluster to another as the size of the UDP message increases. Performance reaches 37.5 Mbyte/s which is significantly less than the theoretical peak bandwidth of 125 Mbyte/s.*

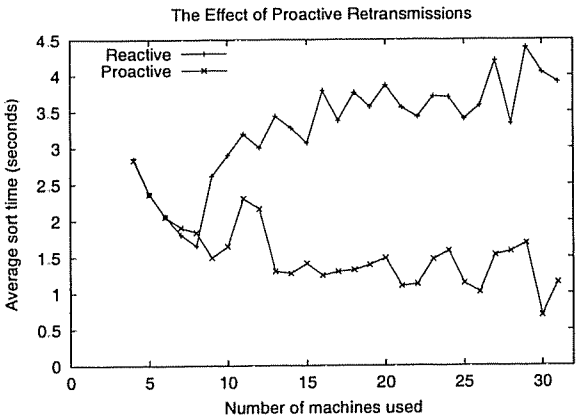

Figure 6: **Proactive retransmissions.** *This graph shows how proactive retransmissions constrain the variability imposed by the faulty hardware. The reactive strategy in which dropped packets are requested only after the timeout expires incurs a much greater penalty. The average of 30 trials is shown.*

a margin of dimishing returns as the number of utilized nodes grows beyond 20; as the amount of data transferred decreases, the fixed overhead dominates.

The distribution time was influenced by several factors. One of these was that the gigabit switch had a faulty module, which was observed to drop packets ocassionally under heavy load. To overcome this problem we used proactive retransmits.

Our transport protocol, UDPAM, insures reliable delivery through the use of a timeout thread which resends any dropped packets. This thread checks at regular time intervals whether there are any packets not yet acknowledged by the receiver and resends these packets. Although reliable, we found that waiting for timeouts is not aggressive enough to provide a good distribution time.

For this reason, we added a function to the UDPAM library which allows us to resend immediately all unacknowledged packets. This allows us to retransmit packets proactively as soon as the distribute phase of the sort is complete. Figure 6 shows the difference between proactive retransmits and the default UDPAM reactive technique, which uses timeouts. Proactive sends reduced the average sort time as the sort scales to one-eight of the reactive time at 30 nodes.

Another factor that originally had an adverse affect on communication time was the ARP (Address Resolution Protocol) traffic. The default ARP timeout values caused ARP traffic to occasionally flood the network in order to renew their cached entries. We found that even such a seemingly minor disruption could have a significant impact on our sort performance; this was especially so due to the sensitivity of our switch as described above. Given the static nature of our environment, such frequent ARP traffic is unnecessary and was turned off resulting in less performance variation and a tighter distribution of our sort times.

Finally, to avoid dropped packets because of lack of buffer space in the kernel, we increased the size of the kernel network receive buffer to accommodate the amount of data that can be received by each node.

5 Putting it all together

By developing a fast remote execution layer, configuring the cluster properly for disk and network performance, and developing simple mechanisms to deal with occasionally faulty hardware, we were able to transform the cluster into a high-performance parallel sorting system. Even though the rate of scale decreases as nodes are added, we were able to post minimal gains to the limits of our cluster's capacity (32 nodes). Eventually, by carefully tuning the disks and the networks (and with some patience) we were able to set a new record of 0.48 seconds for the Datamation challenge.

Figure 7 shows how each phase of the sort scales as more nodes are utilized[3]. This graph under-

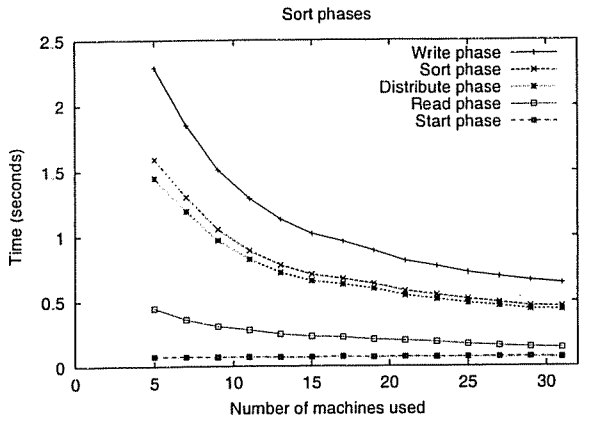

Figure 7: **Sort scaling.** *This graph shows the performance of each of the sort phases as an increasing number of machines is used. Each point is the best time observed from a run of thirty iterations and the lines are drawn stacked. The lowest of 30 trials at each plotted point is shown.*

scores as well the disproportionate amount of time the sort program spends in the distribution phase of its operation.

5.1 Validation

To validate the correctness of our sort, we developed some additional tools. We noticed additionally that there are three aspects of correctness: size, sequencing and validity. A correct sort is one which has sorted correctly every key without corrupting any values.

To verify this, our check program uses `ssh` to collect both the distributed input and outfile files onto one machine. It then concatenates the files together and verifies both validity and size by doing a checksum of each 100-byte record in each file. Sequencing is checked by ensuring that each key follows the previous in the concatenated output file.

We would like to stress the importance of regularly checking all three factors when writing sort programs. We found that the likliest opportunity for bugs to enter our code was when we had disabled the validation phase of our sort.

[3]The sort results as presented here are inflated because of the overhead incurred by gathering the timing data for the different sort phases. Timing each of the phases added an extra global communication to compute the maximum time spent in a specific phase over all the nodes.

6 Conclusion

We have presented our experiences in tuning a cluster to optimize its performance on the Datamation benchmark. Even though the Datamation benchmark has changed quite a bit over time, its value as a tool for measuring and understanding system performance has not diminished, as it stresses different aspects of systems than other sorting benchmarks. Through careful engineering of a remote execution layer, proper configuration, and simple techniques to overcome hardware misbehaviors that may be commonplace in large-scale clusters, we have demonstrated that Linux-based PC clusters have the potential to become the basis for interactive parallelism in the years to come.

Acknowledgements

We would like to thank the Computer Systems Lab, especially H. Paul Beebe and David Parter, for keeping our cluster running and David DeWitt for encouraging us to start this project and motivating us to finish it. Finally, we thank Jim Gray for his excellent and sustained encouragement and advice.

References

[1] Anonymous. 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.

[2] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David P. Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings of the 1997 ACM SIGMOD Conference*, pages 243–254, 1997.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *The 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[4] Brent Chun. UDPAM: Active Messages over UDP. Network of Workstations Project Retreat, January 1996. http://now.cs.berkeley.edu/Retreat/Winter96/udpam-slides.ps.

[5] IBM Corporation. *Ultrastar 9LZX/18ZX Hardware/Functional Specification*, December 1998.

[6] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, 1993.

[7] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[8] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. AlphaSort: A RISC Machine Sort. In *Proceedings of 1994 ACM SIGMOD Conference*, May 1994.

[9] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.