HPVM MinuteSort

Xianan Zhang
Department of Computer Science and Enginnering
University of California, San Diego
xianan@csag.ucsd.edu

Luis Rivera
Department of Computer Science
University of Illinois at Urbana-Champaign
lrivera@cs.uiuc.edu

Andrew A. Chien
Department of Computer Science and Engineering
University of California, San Diego
achien@cs.ucsd.edu

Abstract

Using a modified implementation of the HPVM-MinuteSort we benchmarked a heterogeneous configuration of a HPVM cluster. We pinpointed bottlenecks, misconfigurations and the upperbound performance of the several components of the cluster configuration. The benchmark sorted 21.8 GB of data in less than minute, which improves on the previous record for more than 100%. The cluster features two kinds of systems that differs in the amount of memory, IO capability and CPU power. The MinuteSort was designed as a onepass sort, which moves the data from the nodes with more IO capabilities to the nodes with more memory and CPU power to be sorted. After some troubleshooting, the design allowed us to exhaust the memory of the NetServers. To launch the application we used Catapult, a DCOM based tool to start up remote applicatoins.

1 Introduction

Disk-to-disk sorting is a very good benchmark to measure the system capabilities for data movement[6, 1], and well known in the database community. In a parallel implementation, as in the HPVM-MinuteSort[8], the benchmark stresses the IO, communication and memory subsystem. Therefore, we selected the MinuteSort to benchmark a heterogeneous HPVM cluster which features two kinds of systems, those with a rich IO capability given by a four disk stripe set, and those with more CPU power and memory capacity. This presented an interesting problem to exploit both, the large memory capacity and IO power of the cluster. The MinuteSort metric is the amount of data or records sorted in a minute, and the input data is resident on disk and the output data is written back to disk.

The HPVM cluster at UCSD[1] contains 32 Kayak sytems and 32 NetServers, all of them dual Pentium III, for a total of 128 processors. The network is a Myrinet switched interconnect with bidirectional links of 160 MB/s, with 1.2 GB of peak bisection bandwidth as a result of the topology used. The Kayak systems feature 300 MHz processors and 384 MB SDRAM each, whereas the Netservers feature 400 MHz processors and 1 GB SDRAM each. Also, the Kayaks IO capability was expanded with a stripe set of four IDE disks attached to a 3Ware controller which performs hardware striping, plus an IBM IDE disk used as a system disk. The NetServers feature only two logically separated SCSI disks and one of them is used as a system disk.

[1] http://www-csag.ucsd.edu/projects/cluster.html

In order to find the upperbound performance of the IO subsystems of both machine types, we performed tests for sequential IO, the results for the stripe set showed a peak performance for reading of 55 MB/s and 45 MB/s for writing. Each SCSI disk in the NetServers performed equally for reading and writing, peaking at 16 MB/s. Testing also revealed that the switches of the Myrinet network could only achieve about 80% of the total bisection bandwidth, and we realized that the best performance was achieved by minimizing the contention in the network links by making sure that two or more nodes were not sending data to the same machine at the same time. We also rewired the cluster in order to double the bisection bandwidth, which does not imply the addition of new hardware but just a change of topology.

After tuning and troubleshooting the application, it performed close to the expected performance, with some extra overhead in the overlapped IO and communication stage because of the contention mentioned above, which was only reduced enough to fully populate the memory of the sorting nodes. The best perfomrance was measured in 21.8 GB in 56 seconds, 7 seconds for startup and 49 seconds to sort. The aggregate bandwidth for the reading stage was 768 GB/s and 1.37 GB/s for the writing stage.

2 Cluster Performance

To make sure that our design would make the most of the resources we had in our cluster, we first tested the IO subsystems of both IO and sorting nodes, plus the network bandwidth that the network could deliver in an all-to-all and point to point communication pattern. The former takes place in the IO-communication overlap of the first stage, and the later in the last stage when data already sorted is sent back to the IO nodes for writing. Below we described the tests and performance results for IO and communication.

2.1 IO Performance

In Table 1 we show the disk specifications of all the different disks on the cluster. In both cases there is a single disk used for system activity such that all the IO generated by the sorting application was not going to contend for disk use. The selection of disk type, capacity and other specification, was not driven by the sorting application, the goal was to create a terabyte storage server and the focus was price and capacity, not disk speed. Also, the reason for placing most of the disks in the systems with less memory was due to expandability constrains of the PC boxes.

Type	Controller	Speed (RPM)	Capacity (MB)	Units/Node
Maxtor DiamondMax Plus 5120	3Ware	7200	20.5	4
IBM Deskstar 22GXP	IDE	7200	22.0	1
Barracuda	SCSI	7200	18.2	2

Table 1: Specifications of the disk drives installed in the cluster, the DiamondMax drives form a 4 drive strpeset and the IBM works as a system disk in the Kayak systems. For the NetServers, one of the two SCSI disks is used as system disk and the second one is used for applications IO transactions.

In order to take advantage of all the potential IO bandwidth in the Kayaks, we used the 3Ware controllers[2] for hardware stripping, which scales almost linearly[5] for up to four drives. These allowed us to concentrate on how to use the potential bandwidth instead of trying to generate it by means of software striping. Finally in both kinds of systems, one disk is used as system disk, and the sorting application does not make use of them.

In order to measure the real performance of the stripe set and the SCSI drives, we basically repeated the tests for sequential performance described in the literature[7, 8] for NTFS. In such tests, several IO schemes were tested with different parameters and they were compared in order to find the best way to exploit our IO potential IO bandwidth. The results of those tests are shown in Table 2, the numbers reported are the median of 5 tests performed in 5 machines.

[2] http://www.3ware.com

	Kayak (MB/s)		Netserver (MB/s)	
	Read	Write	Read	Write
Buffered Synchronous	30	26	17	14
Buffered Overlapped	40	23	17	10
Buffered Thread	26	26	17	14
Unbuffered Synchronous	20	10	17	17
Unbuffered Overlapped	56	46	17	17
Unbuffered Thread	60	50	18	18

Table 2: Performance of the NTFS IO schemes tested using the stripe set and the SCSI drive. The results shown correspond to the best values when using the same chunk size as in the application. For the HPVM-MinuteSort we used unbuffered thread and a chunk size of 10 MB.

The results show that for the stripe set, there is a clear difference between Write and Read performance, whereas for the SCSI drive there is almost no difference. In general, unbuffered IO performs better, specially when the chunk size is above the 64 KB chunk size the file system cache uses to write back or from disk. For the stripe set, increasing the chunk size improves performance steadily. Is hardly believable that many applications besides the sorting application would be able to use such chunk size and consequently obtaining the best performance.

2.2 Network Test

For the parallel implementation of the MinuteSort benchmark, communication performance is vital to obtain good performance. In other words, it should be fast enough to prevent it from being the bottleneck. In our particular case IO should never wait for network activity to complete, being IO the slowest component in the chain, it is desirable to perform it without the faster components slowing it down.

The topology of the HPVM cluster at UCSD should be able to achieve 1.6 GB of bisection bandwidth, which corresponds to the theoretical limit of the eight links of 160 MB/s each connecting both halfs of the cluster. Therefore, we decided to measure how much of such bandwidth could be achieved, and designed a point to point communication test using MPI unblocking receive calls, which offers the best communication rate in MPI. The results of the first test when a Netserver or a Kayak is the receiver is shown in Figure 1, it exposes a problem that had been previously noticed for the Kayak systems, the peak performance when a Kayak is the receiver is about 30% slower than a Netserver, topping at only 70 MB/s.

To measure the achievable bisection bandwidth of the cluster we used the same point to point communication tests described above, but using the Kayaks as senders and Netservers as receiver, in order to stress the network as much as possible and find the performance upperbound. The results are shown in Figure 2, it is visible how adding machines to the computation does decrease the average performance perceived per node in a consistent pattern, the efficiency of the switched network reaches 80% of the advertised peak bandwidth.

In order to corroborate this behavior, we analyzed the routes generated by FM and the topology of the cluster, corroborating that the routes where well balanced through the links in the absence of conflicts in both sender and receiver side. Using the routing information we tested the switched behavior under the presence of contention for an specific link, Figure 2 shows the performance drop as perceived for the sender when the number of nodes contending for a single link increases, but the aggregate bandwidth corresponds to 80% or about 128 MB/s of the advertised peak bandwidth regardless of the number of senders contending for a link. Therefore, the achievable bisection bandwidth of the cluster using the existing topology was 1 GB.

3 Design

The design of MinuteSort application was based on the HPVM-Minutesort which sorted 10.3 GB in a minute. The bucket sort it implements is a port of the NOWSort to HPVM with fixes and proper modifications to fit the environment[8]. Since in this implementation IO takes up to 90% of the sorting time, it did not make

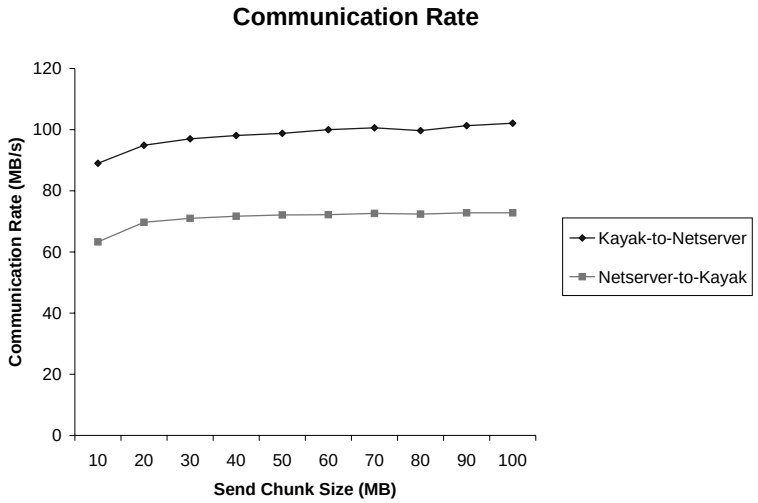

Figure 1: Communication rate of Kayak and Netservers when they act as receivers. In the current implementation of FM it is the receiver that represents the bottleneck, the momory bandwidth of the Kayaks limit the peak performance to 70 MB/s. When a Kayak send and a NetServer receive the performance reaches 100 MB/s.

sense to try to implement a new algorithm and we only improved the efficiency of the algorigthm by taking advantage of the two processors of the sorting nodes, by using two thread to sort the buckets. The plan was to use the same bucket sort algorithm and concentrate on exploit the existing IO power in the best possible way, which in this case means to do it as fast as possible.

Figure 3 shows the data flow of our design, which basically moves the data from the IO nodes to the computing nodes, after the records are sorted, half of the data is sent back to the IO nodes and the other half is written locally, in order to take advantage of some parallelism when writing. The IO nodes perform a double buffering to hide the communication cost in both cases, when sending data and when receiving data. The next subsections describe the measured performance in both IO and sorting nodes, that suggested the design described here.

Given the data obtained from our tests described in Section 2, we generated the following performance projections according to the application architecture explained in above. The goal of this model is to make the best use of the memory of the cluster and to proof that the proposed model is the best possible it. The total amount of memory in the cluster is 44 GB, but two thirds of it is in the NetServers, which happen to have around a third of the IO power of the Kayaks. The first goal in mind is to take advantage of the amount of memory present in the NetServers, which could happen

3.1 IO Nodes

Reading the input data from local disk in the sorting nodes (Netservers), 1 GB of data could be read in 51 seconds, using remotely the IO nodes (Kayaks), we could read the same amount of data at 21 seconds. In the former case, a sorting node should read, send and receive at the same time, the IO rate is 20 MB/s, hence communication should consume 40 MB/s for a total of 60 MB/s required of PCI bandwidth. In the latter case, the IO node should only read and send and the sorting nodes should only receive, considering a 50 MB/s read rate, it would demand 100 MB/s of PCI bandwidth and 50 MB/s in the receiver side. In theory, all these numbers should be achievable and considering that the benchmark on stake in the MinuteSort, there should be enough time left to launch, sort, and write back to disk.

From the numbers above, that only reading from the IO nodes or a combination of remote-local reading would do the job fast enough, needless to say it also makes it more interesting. Since remote reading represents enough speed to populate all the memory of the sorting nodes we model the sort performance as shown in Table 3, where a time-breakdown per stage using the performance numbers is shown. The time

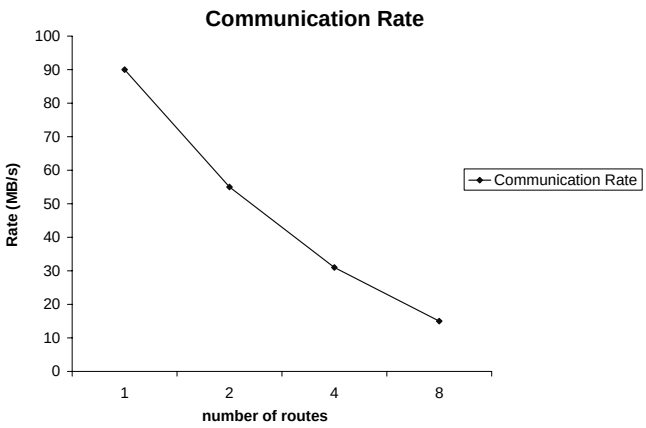

Figure 2: Results of the bisection bandwidth test showed that the communciation performance perceived by the sender degrades as contention increases in the switches.

labeled as launch time takes into account the time to start to load the application to memory in every node, plus the time for FM to synchronize the jobs.

Stage	Time (sec)
Start up	7
Read + Dist	16.6
In-core Sort	5
Write	20

Table 3: This performance takes into account the numbers obtained from the IO tests and ignore the bisecton bandwidth bottleneck, the memory consumed by the sorting algorithm and the limited PCI bandwidth. In this ideal world, the total amount of data sorted per node is 1000 MB for a total of 31 GB.

The reading stage represents a partial sort, the input record are in random order and they are distributed by range to the sorting nodes, since the data follows a uniform distribution, the amount of data sent to each sorting node by each IO node will be roughly the same. A double buffering scheme is implemented to allow the overlap of IO and communication, which means that the bisection bandwidth demand when the 32 IO nodes are sending to the 32 sorting nodes is 1.56 GB.

3.2 Sorting Nodes

The sorting nodes implement the same bucket sort of the NOWSort [3, 4] with the appropriate modifications [8] for the HPVM-MinuteSort. The memory requirements of this implementation increase linearly with the number of records sorted. This overhead comes from two sources, the use of unblocked receive operations which require preallocated buffers, and the fact that the distribution of records is not uniform, which means that the records buffer and buckets need some free space to avoid overflows.

Considering a 10% overhead factor for the record's buffer, 30%, around 100 MB of temporal buffers for unblocking receives and about 50 MB that the NT operating system takes away, we have around 700 MB of memory to fill up with records. The sorting nodes are capable of sorting 2 million keys/sec with the current bucket sort algorithm, since 700 MB represents around 7.3 million records, the in-core sorting should not take more than 4 seconds. Once the records are sorted, they are shipped back to an IO node in a one to one communication, given that the records are already sorted.

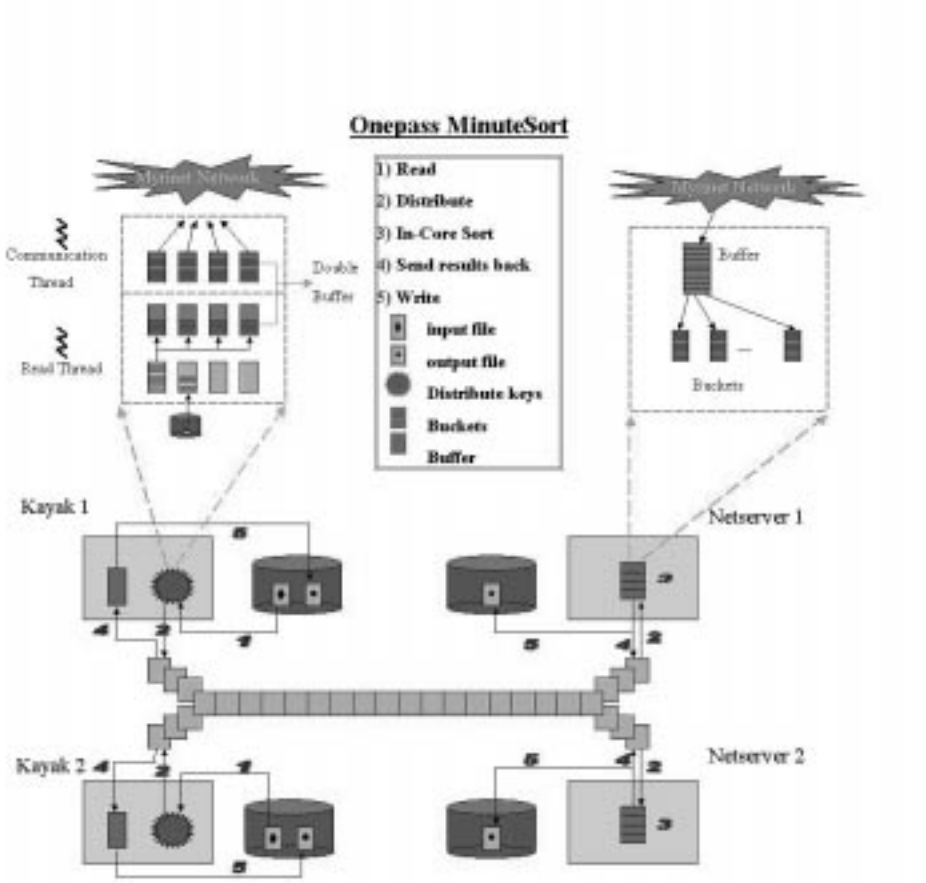

Figure 3: The graph represents the HPVM-MinuteSort design, the IO nodes send data to the sorting nodes using a double buffering scheme, the sorting nodes implement a bucket sort, finally they send write part of the data to disk locally and send the rest to the IO nodes to be written back to disk.

4 Performance

We present the performance obtained with the implementation described above. Since this is a MinuteSort benchmark, the metric is the amount of data sorted in a minute, but there are several interesting aspects that we would like to discuss in this section, as the problems and bottlenecks that we faced and attacked on the process.

4.1 Benchmark Performance

After implementation was complete, we proceeded to test the program using the whole cluster and obtained some disappointing results shown in Figure 4. The performance shows a worse than expected performance in the overlapped stages, where IO and communication takes place. Sorting time behaves as expected and launching time at this point is considered irrelevant, we would like the performance to provide around 10 seconds for launching and synchronization time. The amount of data sorted was 18 GB, the time was under a minute but excluding launching time.

As mentioned above, the peak bisection bandwidth is 25% smaller than the required one and if that is not enough, the achievable one was measured smaller, setting the peak to less than 60% the required

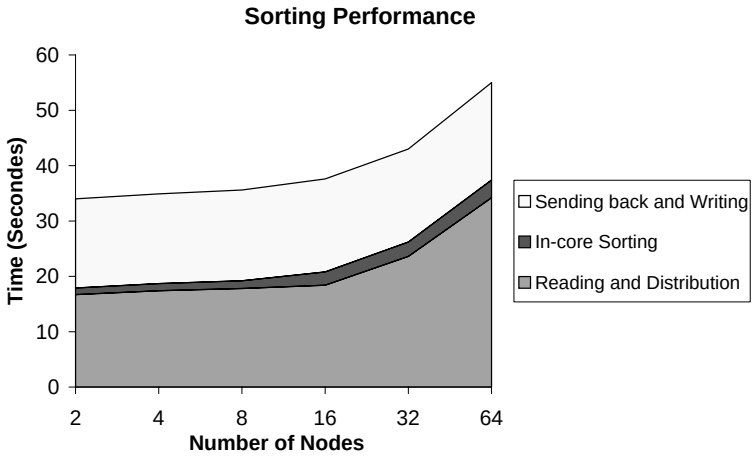

Figure 4: After initial testing, this is the time per stage of the application as the number of nodes is increased. Perfomrance is particularly bad after 16 nodes, when more than two machines are sharing a network link.

amount of bisection bandwidth. However, we are hardly reaching this peak, and after some analysis and experimentation we realized that most of the reason was poor communication scheduling. To schedule communication the carterpillar algorithm is implemented, which would work better if the same nodes kept synchronized all the time and no more than one IO node sends data to a sorting node at the same time.

4.2 Trouble Shooting

Since the data is only close to a uniform distribution, avoiding the problem mentioned above is really hard without any extra help. One solution is the use of barriers before the IO nodes start sending data to a different sorting node, with the corresponding price in overhead for the use of barriers. Another way of reducing this problem is an increase in the size of the buffers used in the double buffering algorithm. However, the memory in the IO nodes is limited and such buffers cannot be make too big, for instance, a buffer of 10 MB would demand a total of 640 MB considering the size of the cluster.

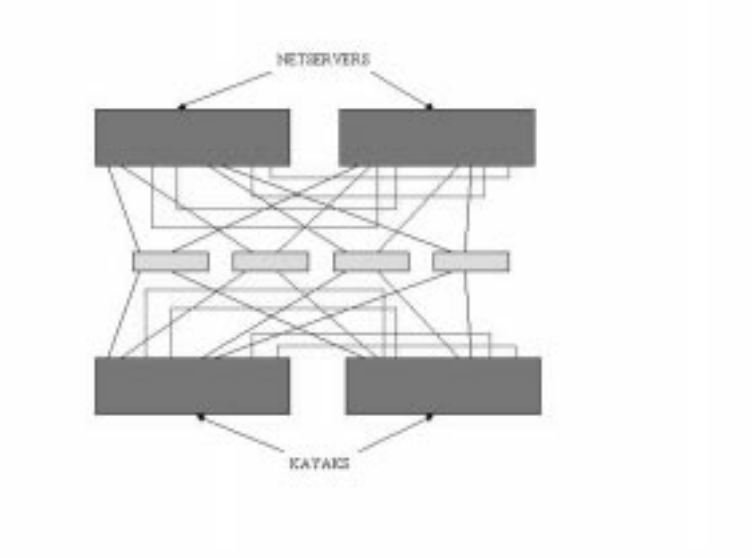

Figure 5: After rewiring the cluster, this was the new topology designed to double the bisection bandwidth by increasing the number of links.

A third solution which would work in combination with any of the two described above relies on the increase of the bisection bandwidth, which requires a reconfiguration of the network topology. The topology at this point offered only 8 different links from the IO nodes to the sorting nodes. Therefore, we reconfigured the cluster to double the bisection bandwidth by doubling the number of links. In the process we did not add hardware, we sacrificed some links in between the interclusters with care of not affecting the overall performance of the cluster. The final topology of the cluster is shown in Figure 5.

Finally, we decided to use the sorting nodes local disk to write half of the record. Initially, we planned to write only the last part of the data in order to speed up the writing of the records to disk, but the performance obtained in the Kayaks was about the same bandwidth that could be obtained using the sorting nodes local disk. Therefore, we decided to write half of the data locally and half of the data remotely, which basically doubles the aggregate IO write bandwidth.

4.3 Final Performance

After rewiring the network, we decided to increase the size of the buffers as a second optimization, with the hope to reduce contention enough to fully populate the memory of the sorting nodes and avoiding the cost of using after each buffer was emptied and the IO nodes switched to a different target. This indeed increase performance and we sorted 21.8 GB in one minute, breaking the previous Minutesort records of 10.3 GB. The time breakdown without launching time is shown in Figure 6, where it is visible that the price paid for reading and communication still has scaling problems when the number of nodes participating in the sort increases. As expected sorting time and write time scale linearly with the number of nodes.

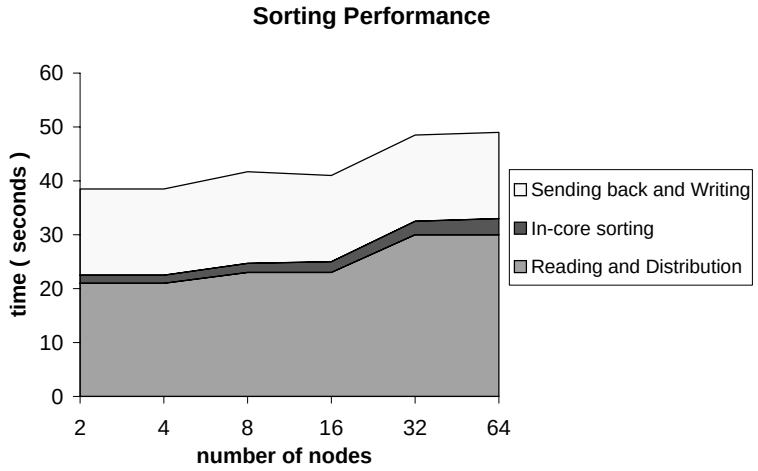

Figure 6: Time per stage after optimizations, as the number of nodes is incrased, the amount of data sorted this time was 21.8 GB with 64 nodes. The performance persist but was minimized enough to allow the application to ran out of memory in the sorting nodes.

To launch the application we started using LSF which represents a lot of unnecessary overhead for this benchmark. What is needed here is only a remote start up tool, hence we changed to Catapult [2], which decreased the launch time dramatically but did not eliminate the high variance of the total launching time due to the FM synchronization. After extensive testing of the launch time, it varied from 5 seconds to 17 seconds, with more consistent reading of 7 and 9 seconds. These readings suggest a sorting time of 52 seconds or less in order to allow room for launch time.

5 Conclusions

We used the disk-to-disk MinuteSort to benchmark a heterogeneous HPVM cluster, which had two types of systems differing in several configuration aspects as IO, memory and CPU speed. For instance half of the cluster can generate at a rate of 1.8 GB/s of aggregate bandwidth and the other half only 640 MB/s.

This particularity allowed an interesting design for the MinuteSort benchmark by partitioning the cluster in IO nodes and sorting nodes. Before writing down a design we measured the performance of each of the components that could represent a bottleneck for the performance of this application, as it is the IO and communication subsystems performance.

Initial testing using the new version of the HPVM version and the old HPVM-MinuteSort code used to establish the MinuteSort record of 10.3 GB, we obtained a peak performance of 15 GB under a minute. In order to improve performance we changed the design to take advantage of the heterogeneous resources existing in the cluster, namely the vast memory of the sorting nodes and the huge IO capability of the IO nodes. The new design showed some bottlenecks that would hardly show with the previous homogeneous approach given the imbalance of resources of the new cluster.

Initial testing showed poor performance which after some analysis was attributed to the shortage of bisection bandwidth and the lack of good communication scheduling. We decided to rewire the cluster being careful to avoid any drop of performance in other part of the cluster. Also, in order to decrease the contention at the receiver nodes for the first stage or the sort, we increased the size of buffers of the double buffering scheme and allow the application to fully populate the memory of the sorting nodes.

After optimizations and tuning, the benchmark was run again and this time we sorted 21.8 GB in 49 seconds, plus 7 seconds of launching time for a total of 56 seconds. At this point we ran out of memory on the sorting nodes. The application generates an aggregate bandwidth for the reading stage of 768 MB/s and 1.37 GB/s of aggregate bandwidth for the writing stage.

6 Bibliography

References

[1] Anon and Others. A mesaure of transaction processing power, 1985.

[2] Philip Buonadonna, Joshua Coates, and Spencer Low. Millenium sort: A cluster-based application for NT using River primitives, VIA and DCOM. In *Proceedings of the 3rd USENIX Windows NT Symposioum*, 1999. Available at http://www.cs.berkeley.edu/ philipb/cs262report.doc.

[3] Andrea C. Dusseau, Remzi H. Arpaci, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on Network of Workstations. In *SIGMOD'97 Tucson, Arizona*, pages 234–254, May 1997.

[4] Andrea C. Dusseau, Remzi H. Arpaci, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. Searching for the sorting record: Experiences in tunning NOW-Sort. In *Symposioum on Parallel and Distributed Tools (SPDT'98)*, August 1998.

[5] R. Horst, J. McDonald, and R. Thompson. Pennysort results with 3Ware DiskSwitch controllers. Research Report TR-1999-1, 3Ware, Palo Alto, California, April 1999.

[6] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. Alphasort: A RISC machine sort. In *Proceedings of 1994 ACM SIGMOD Conference*. ACM, 1943.

[7] Erik Riedel, Catharine van Ingen, and Jim Gray. A performance study of sequential I/O on Windows NT 4. In *Proceedings of 2nd USENIX Windows NT*, pages 1–10, 1998.

[8] Luis Rivera. Disk-to-disk parallel sorting on HPVM clusters running Windows NT. Master's thesis, University of Illinois, Department of Computer Science, 1304 W. Springfield Avenue, Urbana, Illinois., May 2000.