SPsort: How to Sort a Terabyte Quickly

Jim Wyllie (wyllie@almaden.ibm.com)

February 4, 1999

Abstract

In December 1998, a 488 node IBM RS/6000 SP* sorted a terabyte of data (10 billion 100 byte records) in 17 minutes, 37 seconds. This is more than 2.5 times faster than the previous record for a problem of this magnitude. The SPsort program itself was custom-designed for this benchmark, but the cluster, its interconnection hardware, disk subsystem, operating system, file system, communication library, and job management software are all IBM products. The system sustained an aggregate data rate of 2.8 GB/s from more than 6 TB of disks managed by the GPFS global shared file system during the sort. Simultaneous with these transfers, 1.9 GB/s of local disk I/O and 5.6 GB/s of interprocessor communication were also sustained.

Introduction

The speed of sorting has long been used as a measure of computer systems I/O and communication performance. In 1985, an article in Datamation magazine proposed a sort of one million records of 100 bytes each, with random 10 bytes keys, as a useful measure of computer systems I/O performance [1]. The ground rules of that benchmark require that all input must start on disk, all output must end on disk, and that the overhead to start the program and create the output files must be included in the benchmark time. Input and output must use operating system files, not raw disk partitions. The first published time for this benchmark was an hour [12]. With constant improvements in computer hardware and sort algorithms, this time diminished to just a few seconds [7]. At that point, variations on the basic theme evolved [6]. "MinuteSort" [3, 8] measures how much can be sorted in one minute and "PennySort" [5] measures how much can be sorted for one cent, assuming a particular depreciation period. Recently, several groups reported sorting one terabyte of data [8, 9, 10]. SPsort improves substantially upon the best of these results.

Hardware

The benchmark machine is a 488 node IBM RS/6000 SP, located in the IBM SP system test lab in Poughkeepsie, New York. Figure 1 shows the organization of this machine. Each node contains four 332MHz PowerPC* 604e processors, 1.5 GB of RAM, at least one 32 bit 33 MHz PCI bus, and a 9 GB SCSI disk. The nodes communicate with one another through the high-speed SP switch with a bi-directional link bandwidth to each node of 150 megabytes/second. The switch adapter in each node is attached directly to the memory bus, so it does not have to share bandwidth with other devices on the PCI bus. Of the 488 nodes, 432 are compute nodes, while the remaining 56 are configured as storage nodes. Global storage consists of 1680 4.5 GB Serial Storage Architecture (SSA*) disk drives, organized into 336 twin-tailed 4+P RAID-5 arrays, for a total of just over 6 TB of user-accessible space attached to the storage nodes. Compute nodes are packaged 16 to a rack, while the storage nodes, which have 3 PCI busses and consequently are larger, are packaged 8 to a rack. In total, the CPU and switch hardware occupies 34 racks, and the global disks require another 18 racks.

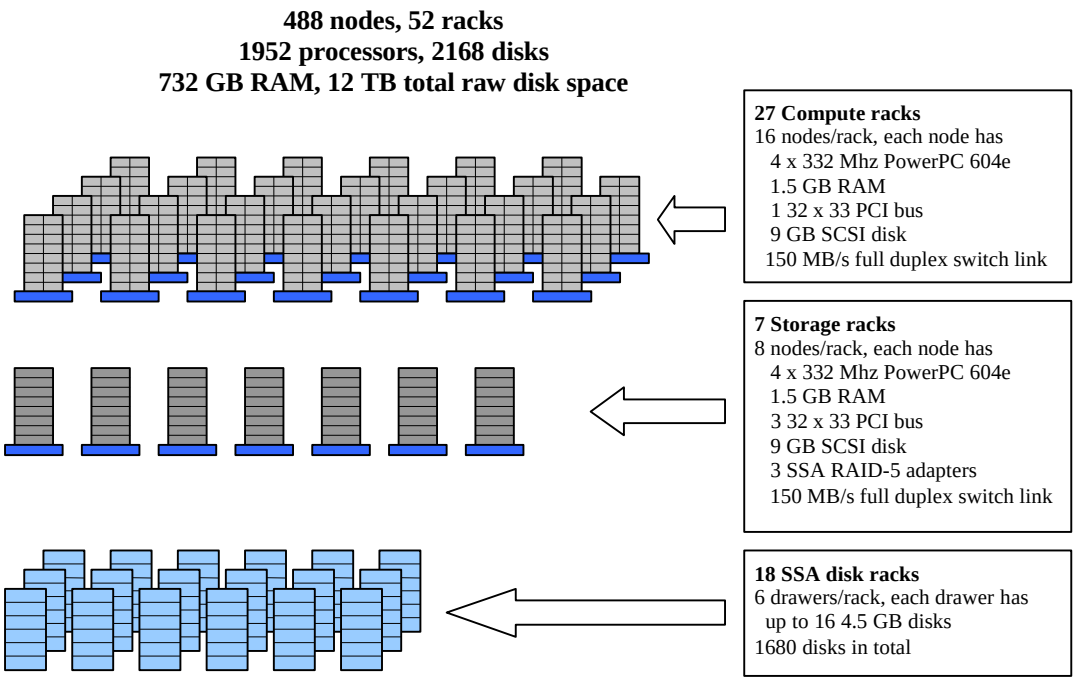

Figure 1. Sort benchmark machine.

System Software

Each node in the SP cluster runs AIX version 4.3.2 to provide X/Open compliant application interfaces. Parallel System Support Program version 3.1 provides the basic cluster management software, including the following functions:

- cluster administration, such as authentication, coordinated software installation, error reporting, and hardware environment monitoring and control,
- high-speed IP communication using the SP switch,
- Virtual Shared Disk (VSD), which makes disks attached to any node appear as if they are attached locally to every node by shipping read and write requests to a node that is directly connected to the disk, and
- group services, to report partial failures of the cluster to other software components, such as VSD and the global file system, so they can initiate their failure recovery procedures.

Parallel Environment version 2.4 (PE) contains the infrastructure to run parallel jobs, including

- starting processes on many nodes,
- collecting and labeling standard output and standard error from all nodes, and
- controlling and interactively displaying the state of parallel processes through an interactive parallel debugger.

PE also includes a high-performance implementation of the Message Passing Interface standard (MPI), by which processes on multiple nodes can communicate [11]. As its underlying transport mechanism, MPI allows applications to send data across the SP switch either through kernel calls that send IP packets or through direct writes into memory on the switch adapter.

LoadLeveler version 3.1 is primarily used on the SP to schedule long-running jobs according to their resource requirements. It also plays the secondary role of managing access to windows into switch adapter RAM to avoid conflicts between multiple processes using user-space MPI on a node at the same time. Although SPsort is the only process active on each node during the sort benchmark, LoadLeveler is still responsible for assigning its switch adapter window.

File System

General Parallel File System version 1.2 (GPFS) manages the 336 RAID arrays as a single mountable file system. GPFS uses wide striping to distribute files across multiple disks and storage nodes. This allows all of the I/O subsystem bandwidth to be brought to bear on a single file when necessary. Distributed file systems such as NFS, AFS, DFS, NetWare**, or Windows NT Server*** all have a client/server structure, in which all disk accesses are made by a single server. The aggregate throughput of these client/server file systems does not scale as nodes are added, since the throughput is limited to that of the central server. In contrast, GPFS is a true cluster file system; there is no central server and therefore no single node that can be saturated with data transfer requests. Instead, GPFS accesses data directly from the VSD server that owns the disk on which they are stored. Thus, the throughput of a GPFS file system scales according to the minimum of the aggregate switch bandwidth of the client nodes and the aggregate I/O or switch bandwidth of the VSD server nodes.

No semantic sacrifices have been made in delivering this level of performance; instances of GPFS coordinate their activity such that X/Open file semantics are preserved[1]. Every node sees the same name space and file contents at all times. GPFS supports cluster-wide atomicity of reads and writes and the proper handling of removal of a file that is still open on some other node.

In addition to data transfers, many of the common control functions in GPFS are also distributed in a scalable way. For example, when SPsort writes its output, many nodes are writing into non-overlapping regions of the same file simultaneously, and each of these nodes must be able to allocate disk blocks. The GPFS space allocation maps are organized in such a way that multiple nodes can concurrently allocate space nearly independently of one another, preventing space allocation from becoming a bottleneck.

The current release of GPFS limits the maximum file size in the 6 TB global file system to approximately 475 GB. Since 1000 GB are needed to hold the input and output of the sort, SPsort uses three input files, two of size 333 GB and one of 334 GB. There are also three output files of approximately the same sizes. The concatenation of the three output files forms the sorted result.

Sort Program

SPsort is a custom C++ implementation optimized for the extended Datamation sort benchmark. It uses standard SP and AIX services: X/Open compliant file system access through GPFS, MPI message passing between nodes, Posix pthreads, and the SP Parallel Environment to initiate and control the sort job running on many nodes.

The sort employs techniques that have appeared before in other record-setting sorts. Its fundamental structure is a two-phase bucket sort: in the first phase, the program partitions the key space into approximately equal-sized segments and appoints a node to handle each segment. Every node is responsible for reading a portion of the input and distributing records to other nodes according to the key

[1] GPFS release 1.2 does not support the **mmap** system call. There are also minor deviations from the X/Open standard in the handling of file access and modification times.

partition [4]. At the end of the first phase, records have been partitioned by key such that the lowest keys are held by node 0, the next lowest keys by node 1, etc. After this distribution phase is complete, nodes send their counts of how many records they each hold to a master node, which computes where in the output files each node should begin writing and distributes this information to every node. In the second phase of the sort, nodes independently sort the records they were sent during the first phase and write them into the appropriate position in one of the output files. Because the keys are chosen from a uniform random distribution, the number of records sent to each node to be processed during phase 2 is roughly equal.

Given sufficient RAM, all of the records distributed during the first phase of the sort could be cached in memory. In this case, the program would be considered a one-pass sort, since each record would be read from disk and written to disk exactly once. However, the 488 node SP has only 732 GB of RAM, so a one-pass sort is not possible. Therefore, SPsort uses space on the local disks of compute nodes to buffer records distributed during the first phase. Since these local disks contain other data (e.g. the operating system), the availability of free space on the local disks determines how many compute nodes can be used in the sort. A total of 400 nodes had enough local free space to participate in the terabyte sort. In addition to these compute nodes, one node controls the sort, and another runs the GPFS lock manager.

Sorting a terabyte on 400 nodes implies that each node receives about 2.5 GB of data during the distribution phase. It is not necessary to write all of these data to temporary disk (a two-pass sort), since a significant amount of memory is available for buffering. As records arrive at a node during the distribution phase, SPsort further classifies them by their keys into one of 16 local sub-buckets. Buffers full of records belonging to the first 5 of these sub-buckets with lowest key values remain in RAM, while buffers holding records in the other 11 sub-buckets are written to the local SCSI disk. When the output phase of the sort finishes with a resident sub-bucket, the buffers it occupies are used to read buffers from a non-resident sub-bucket with higher key values. Since approximately 11/16 of the records are written to scratch disks, SPsort can be considered to perform a 1.6875-pass sort.

While the first phase of SPsort distributes records to the nodes responsible for each key range, designated master nodes create the three output files. Following an MPI synchronization barrier to insure that the file creations have completed, each node opens one of the output files. Since this all happens in parallel with record distribution, the overhead to create and open the output files does not increase the running time of the sort.

Earlier research has demonstrated the importance of paying close attention to processor cache performance for the CPU-intensive portions of sort algorithms [7]. Generally, this has meant working with a more compact representation of the data to be sorted and organizing the data for good cache locality. One popular representation, and the one that is used here during the second phase of the sort, is to form an auxiliary array of record descriptors. Each descriptor contains the high-order word of the record key and a pointer to the whole 100 byte record in memory. To sort the descriptor array in a cache-friendly way, SPsort uses a radix sort [2]. While building the descriptor array, the program counts the number of times each distinct value occurs for each of the byte positions in the high-order word of the key. This yields four histograms, each with 256 entries. Using 32 bit counters for each entry requires a total of 4K bytes to hold all of the histograms for one local sub-bucket, which fits easily within the 32K L1 cache of a 604e processor.

Next, the program scans through the descriptor array and copies descriptors into a second descriptor array of the same size. The position where SPsort moves a descriptor is based on the low-order byte of its key word. The program maintains an array of 256 pointers, one for each byte value, giving the location in the second descriptor array where the next descriptor should be moved for each key byte value. This array is initialized by computing cumulative sums of the histogram counts of the low-order key bytes. During this part of the algorithm, the cache must hold the array of pointers (256 x 4 bytes), as well as one cache line for each key byte value (256*32 bytes). This requires only 9K, which again is well within the L1 cache size.

After the descriptor array has been sorted by the low-order byte of the high-order key word, SPsort repeats the process for the next most significant byte of the key. This sort is stable, so sorting on the next most significant byte of the key preserves the order of the least significant byte within equal values for the next most significant key byte. Once this has been done for all four bytes, the original descriptor array is ordered by the high-order 32 bits of the 10 byte record keys.

In the next part of phase two, SPsort outputs the records in the sub-bucket whose descriptor array was just sorted. Since there are 10 billion records in a terabyte, but only about 4.3 billion possible values for the high-order word of the sort key, ties are common in the descriptor array. The output phase locates runs of records in the descriptor array with the same value of the high-order word of their keys. When it finds a run, it sorts the descriptors of those records using bubble sort, comparing all 10 bytes of their keys. Since the runs are very short on average (2.3 records), the quadratic complexity of bubble sort does not adversely impact the running time of the sort. After SPsort generates a sorted run, it gathers the records in the run into an output buffer and writes them to disk asynchronously.

After all sub-buckets have been sorted and written, each node closes its output file and performs a barrier synchronization operation. When the master node knows that all nodes have finished writing, it insures that all buffered output data are safely on disk to satisfy the rules of the benchmark.

Each node running the sort program is a 4-way SMP. To overlap as much computation as possible, SPsort uses multiple threads on each node during both phases of the sort. At least one thread is dedicated to each activity that might possibly block, such as GPFS or local disk input/output or MPI message sends and receives. Threads pass buffers to one another through simple producer/consumer queues. If a consumer thread attempts to dequeue a buffer from an empty queue, it blocks until data become available or until the producer signals end-of-file on the buffer queue.

To prove that the output files really are sorted, a parallel validation program reads overlapping sequential segments of the output and verifies that the keys are in sorted order. From the point of view of SPsort, the 90 bytes following the key field in each record are just uninterpreted data. In fact, the program that generates the input files places the file name, record number, several words of random data, and a record checksum here. The validation program uses this checksum to make sure that SPsort corrupts no records. It also computes the checksum of all record checksums. Comparing this value with the same calculation performed on the input files guards against lost or repeated records in the output. For the terabyte sort, the validation program ran on 64 nodes and required 12 minutes to confirm that the output was correct.

Performance Analysis

Figure 2 shows the aggregate data rates across all nodes for both global disk and local disk, each separated into read and write bandwidth. Some cautions are in order before interpreting these data. The numbers on the graph were obtained by running the AIX **iostat** utility on only two nodes: one of the VSD servers and one of the nodes executing the SPsort program. The data rates obtained from **iostat** were then extrapolated to 56 and 400 nodes, respectively, to produce the graphs. Although the workloads seen by each VSD server and sort node are statistically similar due to round-robin striping by GPFS, extrapolation is not an ideal way to measure aggregate throughput. The peak in the VSD read throughput above 3 GB/s early in the run is probably an artifact of this extrapolation. Nevertheless, there is evidence in the form of thread timestamps from all 400 nodes that corroborates the overall shape of the throughput curves. The **iostat** output was not synchronized with the execution of the sort program. Thus, elapsed time zero in Figure 2 was arbitrarily assigned to the sample just before the I/O rate began to rise.

During the distribution phase of the sort, SPsort read 1 TB of input data from GPFS. Once the sort program reached steady state, it read from GPFS at a rate of about 2.8 GB/s. Almost all (399/400) of the data read during this phase passed through the SP switch twice: once from the VSD server to the GPFS client reading the data, and again when the sort program sent each record to the node responsible for its portion of the key space. Thus, the sustained switch throughput during this interval was 5.6 GB/s. The GPFS read rate tailed off near the end of the first phase because some nodes finished reading before others. As records arrived at a node, SPsort classified them into one of 16 sub-buckets, and wrote 11 of these sub-buckets to local disk. Local writes during the distribution phase occurred at an aggregate rate of over 1.9 GB/s in steady state. This is close to 11/16 of the input rate, as expected. The distribution phase ended after about 380 seconds.

Between the distribution phase and the output phase nodes had to determine where in their output file to begin writing. While this synchronization between nodes went on, nodes sorted their first resident sub-bucket and collected the first few output buffers. Once all nodes finished reading and were allowed to

begin their output, they very quickly achieved nearly 2 GB/s write bandwidth to GPFS, then stabilized at about 2.25 GB/s for the bulk of the output phase. The steady state output rate was lower than the steady state input rate because each node read from all three input files during the distribution phase, but only wrote into one of the three output files during the output phase. Thus, there was more total sequential prefetching going on during the distribution phase than there was sequential writebehind during the output phase. The overhead of allocating space for the output files was not a major factor in the difference between read and write throughput. The output phase finished after about 860 seconds of elapsed time.

The most striking of the four curves in Figure 2 is the throughput of reading from the local disk. Recall that this curve was obtained by extrapolating the reads done by a single node. It is easier to explain this curve by describing the activity of a single node, which requires scaling the peak throughput back by a factor of 400, to a little over 9 MB/s. At the end of the distribution phase, SPsort started a thread on each node to read non-resident sub-buckets back into RAM. This thread consumed all available buffers as quickly as possible at about time 380 seconds, then had to wait for more buffers to become available. At this point, only a portion of the first non-resident sub-bucket had been read back into RAM. Once the first resident sub-bucket was sorted and written to the output file, SPsort freed the buffers it was using, allowing the local read thread to wake up and use the buffers to read the remaining part of the first non-resident sub-bucket and the beginning of the next one. In the hardware configuration used, GPFS output bandwidth was less than that of the 400 local disks, so the rate of filling buffers with data from non-resident sub-buckets was greater than the rate at which they could be emptied into GPFS. Thus, the *read fast, run out of buffers, block* pattern repeated for all 11 of the non-resident sub-buckets, each one producing a peak in the local read throughput graph. By elapsed time 700 seconds, no more non-resident data remained, so no further local reads were required while SPsort sorted and wrote out the last 5 sub-buckets..

In addition to the external measurements taken by **iostat**, SPsort also gathers internal statistics. Counts of how often consumer threads had to wait for data from their upstream producer threads confirmed that the sort program was able to drive GPFS to its full throughput. The program also captures high-resolution timestamps at some of the interesting points during its execution. The clocks of different nodes are synchronized to within about one millisecond, so these timestamps can reliably sequence events across nodes. Besides acting as an independent confirmation of the length of the two sort phases, these data also yield other interesting facts. For example, the actual sort of the descriptor array for a single sub-bucket

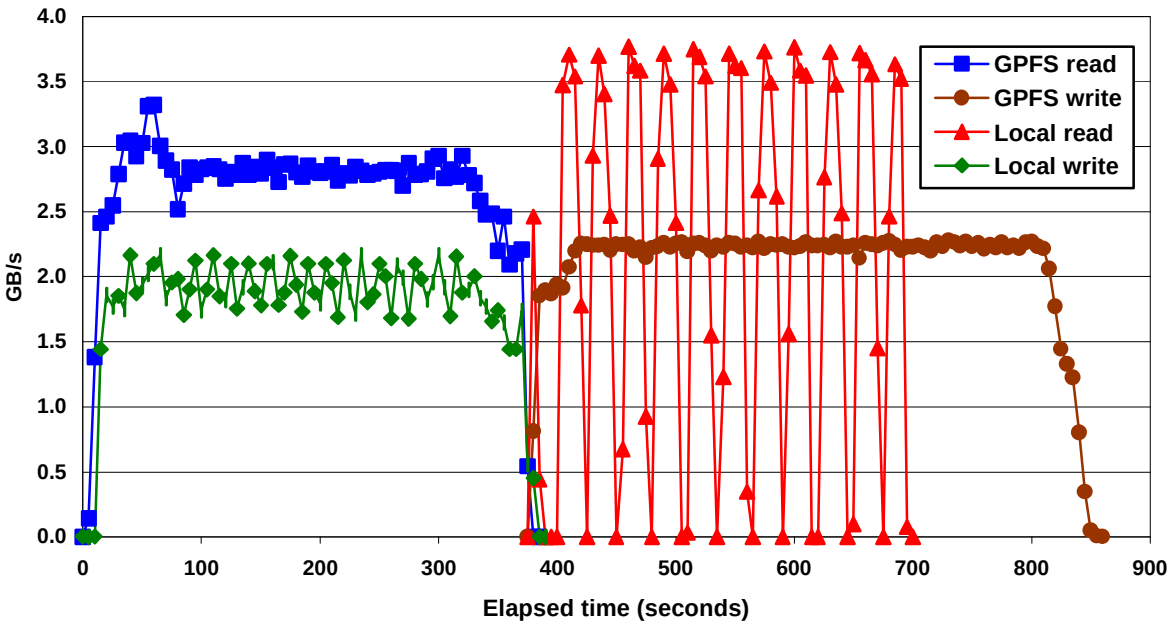

Figure 2. Extrapolated aggregate disk bandwidth.

took only about 0.8 seconds. Furthermore, this time is always completely overlapped with other processing. The sort of the first sub-bucket occurs while nodes are exchanging record count data to determine where to start writing in their output file. Sorts of all subsequent sub-buckets occur while the records of the previous sub-bucket are being gathered into buffers and written to the output file. Thus, at least for the tested configuration, further improvements to CPU performance in the second phase of the sort are unnecessary.

By far the most interesting number to come out of the timestamp data is the length of program startup and shutdown. The duration between the earliest time when SPsort started on some node until the time after the final **fsync** on the master node was 872 seconds. This is only slightly longer than the time during which I/O was occurring on VSD servers according to Figure 2. However, the total time required to run the shell script for the 1 TB benchmark was 1057 seconds. The ground rules of the sort benchmark require reporting the longer time, but 185 seconds of overhead to start up and shut down a program on 400 nodes seems puzzling, and probably easily improved. Possible reasons for this long time include the following:

- Setting up the PE environment. To run SPsort under PE, a remote execution daemon may need to be started on each node, and once running it needs to set up TCP connections for the standard input, output, and error of the sort program on that node.
- Assigning switch adapter resources. The sort program appears as an interactive job to the LoadLeveler job scheduler, but it still needs to be assigned a window into switch adapter RAM on every node before it can use user-space MPI message passing.
- Loading the program. The SPsort program binary was resident in GPFS, and had never been run on most of the 400 nodes since the last time the nodes were rebooted. Since VSD does no caching, the disk block containing the program was read from disk many times before the nodes started running.
- Outputting the timestamps. Each node collects several hundred timestamps while it is running. These calls are quite cheap, probably under one microsecond each. However, after SPsort finishes all of its sorting work on a node, it converts and outputs all of its timestamps. This output gets funneled back to the primary node, which labels it and redirects it into another GPFS file. This file ended up containing over 138,000 lines of output.
- Tearing down the PE environment. Everything that was set up to run the sort needs to be cleaned up when it completes.

Analysis of the performance of SPsort is hampered by the fact that the author had only one opportunity to run it on 400 nodes with 1 TB of input data. Despite careful preparation, not all measurements that should have been taken were made during that one run. As a result, it is not possible to assess the relative magnitudes each of the preceding reasons. It is not even possible to say whether most of the overhead happened before or after the I/O shown in Figure 2. Furthermore, just a few hours after the sort ran, the machine was disassembled for shipment to a customer site.

Future Work

Clearly, the first piece of future work would be to run the sort with additional instrumentation to quantify and then improve the overhead of initiation and termination. Aside from simply turning off the gathering of timestamp data, there are also options to disable unnecessary parts of the PE setup that were not known to the author when he ran the test in December.

There are a large number of tunable parameters in SPsort that were not explored thoroughly due to time pressure. These include the number and size of various kinds of buffers, the number of resident and non-resident sub-buckets, the style of MPI communication used, and the number of MPI send and receive threads. Increasing the depth of writebehind in GPFS should bring the write bandwidth up to that achieved for read.

During the first sort phase, some nodes finish reading and distributing their portion of the input significantly before the last node. Since GPFS makes all input data globally accessible, nodes that finish

early could be assigned another segment of the input to process. This would require restructuring the division of labor among nodes, but would not be difficult. The effect on performance would be to sharpen the knee of the GPFS read throughput curve in Figure 2, leading to a shorter distribution phase. Applying the same technique to the output phase is problematic, since the data that need to be written to GPFS are only available on the node to which they were sent during the distribution phase.

Running the sort program on a more balanced configuration would improve its price-performance substantially. The ultimate bottleneck proved to be the VSD servers. Without changing the hardware configuration, it should be possible to drive data through GPFS just as fast using 256 nodes instead of 400. This would increase the amount of temporary storage required on each sort node to 4 GB, still much smaller than the 9 GB local disks. It would also increase the required bandwidth to the local disks. By increasing the amount of RAM used to cache resident sub-buckets from 780 MB to 1000 MB, it should be possible to handle this higher throughput on the existing local disks. Adding an additional local disk to each node would permit a further significant reduction in the number of nodes.

It is possible to scale the sort still further using a super-cluster of SPs interconnected with High Performance Gateway Nodes (HPGN). The sectors of such a super-cluster are interconnected through multiple parallel paths with a total bandwidth measured in GB/s, and MPI communication is supported across these paths. Thus, if the startup overhead can be reduced, it should be possible in principle to modify the sort program described here to take advantage of all of the resources of a super-cluster and improve the sort time for a terabyte by a factor proportional to the size of the super-cluster. An alternate approach would be to sort a larger input dataset, thus amortizing the startup cost across more data and driving up the sort rate as measured in GB/s.

References

[1] Anon., Et-Al. (1985). "A Measure of Transaction Processing Power." *Datamation*. V.**31**(7): pp. 112-118. Also in *Readings in Database Systems*, M.J. Stonebraker ed., Morgan Kaufmann, San Mateo, 1989.

[2] Agarwal, R.C. "A Super Scalar Sort Algorithm for RISC Processors." ACM *SIGMOD '96*, pp. 240-246, June 1996.

[3] Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Culler, D.E., Hellerstein, J.M., and Patterson, D.A. "High-Performance Sorting on Networks of Workstations." ACM *SIGMOD '97*, Tucson, Arizona, May, 1997. Available at http://now.cs.berkeley.edu/NowSort/nowSort.ps.

[4] DeWitt, D.J., Naughton, J.F., and Schneider, D.A. "Parallel Sorting on a Shared-Nothing Architecture Using Probabilistic Splitting," Proc. First Int. Conf. on Parallel and Distributed Info Systems, IEEE Press, January 1992, pp. 280-291.

[5] Gray, J., Coates, J., and Nyberg, C. "Performance/Price Sort and PennySort." Technical Report MS-TR-98-45, Microsoft Research, August 1998. Available at http://research.microsoft.com/barc/SortBenchmark/PennySort.doc.

[6] http://research.microsoft.com/barc/SortBenchmark/ Sort benchmark home page, maintained by Jim Gray.

[7] Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., and Lomet, D. "AlphaSort: A Cache-Sensitive Parallel External Sort." VLDB Journal **4**(4), pp. 603-627 (1995). Available at http://research.microsoft.com/barc/SortBenchmark/AlphaSort.rtf.

[8] http://www.ordinal.com/ Home page of Ordinal Corp., whose NSORT program holds a record for MinuteSort and performed the first reported terabyte sort.

[9] http://www.sandia.gov/LabNews/LN11-20-98/sort_story.htm "Sandia and Compaq Computer Corp. team together to set world record in large database sorting." Description of terabyte sort at Sandia Lab on 11/20/98 in "under 50 minutes."

[10] http://www.sgi.com/newsroom/press_releases/1997/december/sorting_release.html "Silicon Graphics ccNUMA Origin Servers First to Break Terabyte Data Sort Barrier." SGI press release from 12/97 announcing the first terabyte sort: 2.5 hours for 1 TB.

[11] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. *MPI: The Complete Reference*, The MIT Press, 1995.

[12] Tsukerman, A., "FastSort– An External Sort Using Parallel Processing," *Tandem Systems Review*, **3**(4), Dec. 1986, pp. 57-72.
