TeraByte TokuSampleSort

Bradley C. Kuszmaul*
MIT CSAIL, Tokutek Inc., Cilk Arts Inc., and MIT Lincoln Laboratories

May 1, 2007

Abstract

Using the tx2500 disk cluster at MIT Lincoln Laboraties, I sorted a terabyte (10^{10} 100-byte records) in 197s using an "Indy" sort, and in 297s using a "Daytona" sort. I sorted 264GB in one minute using an "Indy" sort and 214GB in one minute using an "Daytona" sort. The sort employed a parallel sample sort, and ran on 400 nodes, each containing a 6-disk RAID, and 8GB of memory, all connected by Infiniband. I employed TCP sockets to communicate between the nodes. I used a FUSE module that logically concatenates files distributed across disks into one large file.

My sort, called *TokuSampleSort*, employs a parallel sample sort similar to the one used on supercomputers a decade ago [1]. Borrowing the explanation from [1], the sort works as follows:

> *Assuming n input keys are to be sorted on a machine with p processors, the algorithm proceeds in three phases:*
>
> 1. *A set of $p-1$ "splitter" keys are picked that partition the linear order of key values into p "buckets."*
> 2. *Based on their values, the keys are sent to the appropriate bucket, where the ith bucket is stored on the ith processor.*
> 3. *The keys are sorted within each bucket.*

Just as in [1], I oversampled, picking 64 random *candidate* keys per node, sorting the candidates globally, and then using every 64th candidate as a splitter key.

Unlike the sort of [1], my sort involves I/O. Data starts and ends on disk, and in the case of the Daytona sort, the record boundaries are delimited by newlines. (In the case of the Indy sort, the records are exactly 100 bytes long.) I relied on an operating-system modification to allow the system to view as a single file the concatenation of many files distributed across many nodes.

The Indy version takes advantage of fact that the records are 100 bytes in size, that the sort key is the first 10 bytes, and that the sort key is random. The Daytona version sorts records of varying sizes, uses the entire record as a sort key, and does not rely on the sort key having any particular distribution (e.g., it works just as well on nonrandom keys.)

The machine has 3.2TB of main memory total, allowing me to perform the sort in one I/O pass. The data is read in, sorted, and written out.

Figure 2 shows how the time was spent by the sorting program. The time breakdown is approximate since different nodes finished their jobs at different times. The nodes synchronized only during the sorting of the sample and the permutation steps (as well as at the end when reporting total elapsed time.) These measurements exhibited high variability. For example, I found that the cost of the disk I/O can vary by 30%, which I suspect is due to the disk controller retrying I/O or the RAID controller compensating for bad reads.) It is clear that the local sort is far more expensive for the Daytona than for the Indy, and accounts for almost all of the performance difference. Although the individual numbers in Figure 2 vary greatly, the overall run time varies by only a few percent. One explanation is that although the read time on any given node may vary by 30%, the time for the slowest of the 400 nodes is unlikely to

*This work is sponsored in part by the Department of Defense under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government. This work is also sponsored in part by NSF grants CCP-0621511 and CCF-0541209.

	Daytona	Indy
Minute	214GB	264GB
TeraByte	297s	197s

Figure 1: Overall elapsed time including launch, sort, flushing data to disk, and shutting down the job for TokuSampleSort.

Phase	Minute		TeraByte	
	Daytona	Indy	Daytona	Indy
Startup	4.3s	4.3s	4.3s	9.7s
Read	5.0s	10.0s	30.0s	37.1s
SortSample	6.8s	5.5s	20.7s	44.1s
Bucketize	3.8s	4.4s	13.9s	14.0s
Permute	9.9s	12.2s	45.9s	46.4s
LocalSort	21.7s	5.5s	138.9s	23.9s
Output	4.2s	2.0s	27.4s	11.6s
Fsync	0.0s	7.2s	0.0s	0.0s

Figure 2: Time breakdown by algorithm phase. The time breakdown is approximate, since the various nodes finished their tasks at different times. The Startup step establishes the TCP sockets between each pair of the 400 nodes. The Read step reads each node's data into main memory. The SortSample step selects candidates, sorts them, and broadcasts the splitter keys. The Bucketize step divides each node's data into buckets. The Permute step performs an all-to-all communication, sending the data from Node *j*'s *i*th bucket to Node *i*, for all *i* and *j*. The LocalSort step sorts each node's received data. The Output step writes the sorted data to disk. The Fsync step flushes data to disk.

vary by much. There is some evidence to support this theory: On a single run, the read time varies across processes, but the SortSample step finishes at about the same time on every node. Hence the variability is likely to be in the assignment of time to phases rather than the actual performance of the system. In particular, most of the variability in startup times is probably a measurement artifact.

I experienced some difficulty ensuring that the data wasn't in cache, especially for the minute sort. The main memory on the nodes is large enough to hold the 300GB input data set, the 300GB output data set, and the two additional in-memory copies used by the sorting algorithm. Hence, on every run I had to take steps to force the input data out of the cache so that the disk I/O would actually take place. I tried a second consecutive run without the intermediate cache flush, and it runs about 5s to 10s faster for the Minute sort, and can run up to 25s faster for the Terabyte sort. Here I report numbers for the case where the cache was flushed before starting.

Unlike the Penny Sort benchmark, the Terabyte sort benchmark does not require that the input and output data appear to be one file to the operating system. Instead the Terabyte sort rules allow the file to be broken up into many files, that when concatenated contain the correct data. My approach was to divide the terabyte file into 400 files, each of size 2.5GB, and to put each file on the local filesystem of a single node. At the end of the sort, there are 400 files (of slightly varying sizes), that when concatenated contain the sorted data. Initially, only the input files exist. (That is, any output files from previous runs are deleted before the run starts.) Although one could imagine that preallocating the output file would save time, that preallocation actually slows down the performance. So even if the rules allowed preallocation, I would not do it.

Although it was not required, Andy Funk (of MIT Lincoln Labs) and I built a file system that allows us to view the 400 files as a single files. I employed the Linux FUSE file system module [2] to produce a file system in which files that are distributed across nodes can be accessed as a single file through the OS interface. I used two different FUSE modules: `SSHFS` and `CATFS`.

The `SSHFS` FUSE module [3] mounts a remote file system locally using the SSH File Transfer Protocol. This allowed us to mount 400 files from 400 different nodes without requiring us to configure NFS mounting of 400 nodes.

Module	lines	characters	semicolons
Parallel sort	434	12904	283
Daytona quicksort	59	1503	44
Indy quicksort	123	3179	74
TCP socket management	371	11379	268
Logging and instrumentation	92	2156	46

Figure 3: The code size for TokuSampleSort. I counted the number of lines, the number of characters, and the number of semicolons. The parallel sort module implements the code specific to parallel sorting. Two different quicksort modules are implemented for the Daytona and Indy problems. The TCP socket management module implements functionality such as all-to-all communication and computing the sum of values from every node. The logging and instrumentation module provides logging and measures where time is spent. The code is written in C.

Andy Funk and I implemented the `CATFS` FUSE module for this project. The `CATFS` (concatenation file system) logically concatenates files together. Conceptually, when the OS requests a read of the *i*th byte of the concatenated file, the `CATFS` module looks at the sizes of all the individual files and reads the proper byte from the proper file. The `CATFS` module is 264 lines of code.

It is not clear why the `fsync()` operation at the end sometimes takes 0s and sometimes takes 7.2s. The daytona verion uses `fwrite()`, whereas the Indy version uses `write()`.

To verify the sort I performed a

```
LC_ALL=C sort -c
```

on the catted output file. I also checked to make sure that every record (looking at bytes 10–20) appeared in the output exactly once, and that for each record, bytes 20–100 were correct, given bytes 10–20. (Recall that bytes 10-20 are the sequence number in the input order, and that bytes 20-100 can be computed from bytes 10–20.)

Elapsed cpu times were measured using the `time` command-line interface. The program reported times for each substep using `gettimeofday()`.

I used TCP sockets running infiniband instead of MPI. First I implemented an MPI version, but I had performance difficulties. The MPI implementation took over 60s to perform an `MPI_init()`, and over 30s to perform `MPI_finalize()`. So I built a TCP version. It is possible that I configured MPI wrong.

I measured the performance on the MIT Lincoln Labs (MIT-LL) tx2500 disk cluster, which comprises 400 nodes, each of which is a dual-socket hyperthreaded Intel Xeon 3.2GHz with 8GB of memory and 6 local disk drives organized as a RAID 5. Thus there are 1600 hardware thread contexts on the 800 processors distributed among the 400 nodes. The nodes are connected using Infiniband and run Linux. The tx2500 went online around April 1, 2007, and I was given exclusive access to the system whenever I needed it during April. MIT-LL paid about $1.9M for the hardware, however Dell contributed significant cost sharing. The estimated value of the tx2500 is about $5.5M. The MIT-LL team and I encountered many problems, including failing disks, crashing nodes, failing RAID controller batteries, and system configuration issues. Whenever I encountered hardware or software problems, the MIT-LLs personnel resolved the problems immediately. This level of support from Lincoln Labs was exceptional.

I plan to web-publish this software under the GPL sometime during May 2007.

Acknowledgments

Jeremy Kepner at MIT-LL provided me with access to the tx2500. Albert Reuther at MIT-LL helped me get started using the tx2500. Pete Michaleas at MIT-LL kept the tx2500 hardware and system software working. Andy Funk at MIT-LL helped build the FUSE module.

References

[1] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167, March/April 1998.

[2] Filesystem in userspace (fuse). `http://fuse.sourceforge.net`, 2006.

[3] Ssh file system. `http://fuse.sourceforge.net/sshfs.html`, 2006.