Sorting 39GB for a Penny

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

May 1, 2007

Abstract

I sorted 39GB for a penny using a cheap general purpose processor and two disk drives, running each drive at about 28MB/s sustained. No special-purpose processors, such as GPUs, were employed for the in-memory part of the sort. The sort is a general-purpose sort that can compare records of arbitrary size, and does not rely on the keys being uniformly randomly distributed.

1 The Algorithm

My sort, called *TokuMergeSort*, employs a classic a two-step merge sort, similar to the one used by Gnu Sort [1]. The first step *splits* the data into small sorted files (sometimes called *runs*). The second step *merges* the small sorted files to produce the output. The input file and the final file are stored on one disk, and the temporary files are stored on another disk. TokuMergeSort employs multithreading to keep the different disks busy, and it employs a priority queue to merge hundreds of files in one pass efficiently. In contrast, Gnu Sort is single threaded and uses an inefficient data structure for merging, which means that Gnu Sort requires several passes to merge hundreds of files.

Splitting

The splitting step repeatedly reads a buffer full of data from the input file into main memory, sorts the buffer in main memory, and then writes the sorted data to a temporary file. At the end of the splitting step TokuMergeSort has constructed several hundred sorted files.

As shown in Figure 1, TokuMergeSort creates three threads (using pthreads): a *reader thread*, a *sorter thread*, and a *writer thread*. TokuMergeSort also allocates three buffers. At any given time the reader is reading into a buffer, the sorter is sorting a buffer, and the writer is writing a buffer. TokuMergeSort sets up three queues to communicate between the threads. The reader thread tries to keep the input disk busy, the writer thread tries to keep the output disk busy, and the sorter thread tries to keep the CPU busy sorting.

I built both *Indy* and a *Daytona* implementations of TokuMergeSort. Both versions of the TokuMergeSort splitter employ quicksort [3] for the in-memory sort of a single buffer. My Indy version takes advantage of knowledge that the records are 100 bytes and that the sort key is the first 10 bytes, and that the sort key is random. The Indy sort first sorts small records each comprised of only the sort key and a pointer, and then permutes the 100-byte records in place. This permutation simplifies the writer thread's job, since the buffer is fully sorted.

My 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 sorter thread simply sorts pointers to the records, and does not permute the actual data. Instead, the writer thread must permute the data as it is being written to the temporary file. The reader's complexity also increases slightly, since the end of a record may not align with the end of the buffer, but this complexity has almost no impact on performance.

Figure 2 shows the time breakdown for the two versions of the splitter. The reader thread dominates the performance. The Daytona sorter thread takes about 50% more CPU cycles than the Indy sorter thread, but the sorter thread

*This work is sponsored in part by NSF grants CCP-0621511 and CCF-0541209.

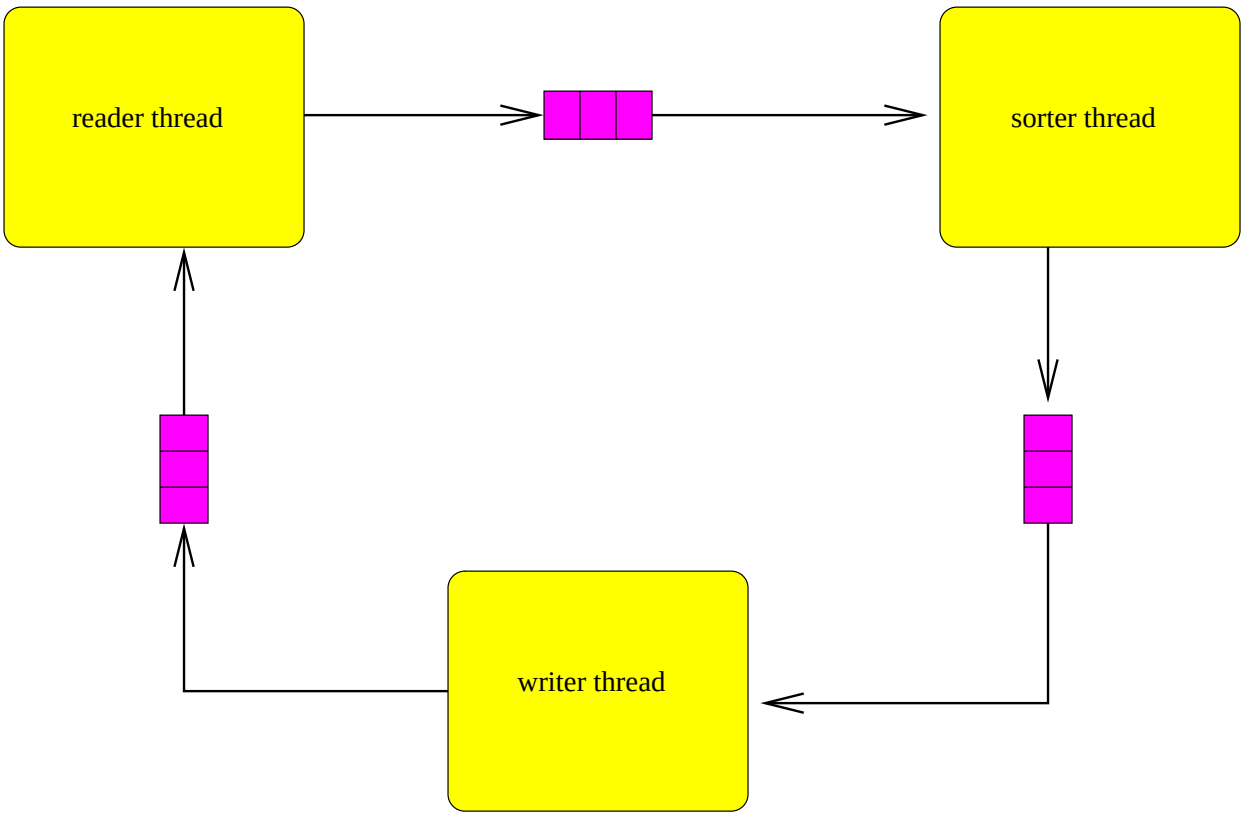

Figure 1: The TokuMergeSort splitter step is implemented by three threads. A reader thread, a sorter thread, and a writer thread. Three buffers of data are cycled between the threads by using three queues. The reader thread reads into a buffer, and then enqueues it for the sorter. The sorter thread dequeues the buffer, sorts it, and enqueues it for the writer. The writer thread dequeues a sorted buffer, writes it to disk and then enqueues the buffer for the reader to refill.

is still idle more than half the time. The writer gets slower, but since the disk I/O for writes occurs asynchronously to the calls to write, it turns out that the writer thread is still idle. (That is, the kernel itself is running threads that actually perform the disk writes.)

Thus the Indy and the Daytona sort run in about the same time. The two disks on this system cannot compete with last year's 59GB Indy sort [2], but they are good enough to outperform last year's Daytona Pennysort winner [4] using only two disks (instead of 4) and 512MiB memory instead of 1GiB memory.

The input file and output file together are too big to fit one disk. Deleting the input file takes about 50 seconds. An alterantive would have been to "trim" the input by 6GB using `ftruncate()`, which takes 13s instead, and gives a total sort size of 40GB for a penny. Sincen Since such trimming is too special purpose for the Daytona, I can only claim the 39GB rate.

Merging

The merging step merges the temporary files to produce the final output file, and deletes the temporary files.

The merger employs a priority queue built using a heap [5]. The merger creates a small buffer for each of the hundreds of temporary files, and then places the buffer into a priority queue sorted by the sort key. The merger repeatedly removes the buffer with the smallest key, writes its line to the output file, adjusts the buffer to contain the next line, and then reinserts the buffer into the priority queue. When the buffer becomes empty, the buffer is refilled from disk. A heap implements a priority queue with $O(\log N)$ comparisons per reinsertion, where N is the number of

	Indy	Daytona
read time	1203.8s	1210.0s
sort CPU time	388.2s	580.6s
write time	748.0s	828.6s

Figure 2: Time breakdown for Indy vs. Daytona splitter. These are the times that the respective threads (reader, sorter, and writer) were busy performing their tasks. Each thread spent the rest of its spent waiting on the queue for the previous step to complete.

step	elapsed time 40GB	39GB
split	1304s	1249
trim input	13s	
delete input		50
merge	1482s	1349
remove temporaries	55s	55
total elapsed time	2854s	2703
time budget	2863s	2863

Figure 3: Elapsed time for 40GB sort (trimming the input just enough to fit the output on disk), and for 39GB sort (deleting the input).

open temporary files.

In contrast, Gnu Sort [1] creates a buffer for each open file, and then repeatedly scans all the buffers to find the smallest key, and then outputs that line. Thus Gnu Sort implements a priority queue where each operation takes time $O(N)$ for each line, and so Gnu Sort constructs a merge tree in which 16 files are merged into a bigger file, and then 16 of those larger files are merged into a bigger file, and so forth. Hence the merge phase of a terabyte sort would require two to three passes over the data. Because TokuMergeSort uses an efficient priority queue, it can merge many hundreds of files in a single pass.

The merger is organized as two threads, a merge-reader, and a merge-writer. The merge-writer removes lines from the priority queue, and writes them to the output. The merge-reader refills the buffers.

The merger performs double-buffering. For each open temporary, two buffers are allocated. One buffer is being consumed by the merge-writer, and the other buffer is being refilled by the merge-reader. The merge-reader fills one buffer at a time, and is idle for a total of only 10.2s (out of about 20 minutes for the merge phase), which indicates that the reader the performance-limiting thread.

The elapsed time for TokuMergeSort is broken down as shown in Figure 3.

2 Conclusion

Figure 4 shows my hardware configuration. The system comprised a dual-core AMD 64 processor and two ATA disk drives. I placed the two disk drives on different ATA channels, so they showed up in Linux as `/dev/sda` and `/dev/sdc`. I purchased the cheapest processor I could find (which surprisingly is dual-core), the cheapest disks I could find (80GB Deskstar with 2MB caches), and the cheapest infrastructure I could find (motherboards, cases, fans, and power supplies.)

The TokuMergeSort Daytona penny sort program demonstrates that sorting performance on simple hardware is very cost effective. No special purpose hardware (such as a GPU) is needed to speed up the in-memory part of the sort. Only two disk drives are needed to achieve good performance, and using additional disk drives to amortize the cost of the processor and power supply would provide only a small performance advantage. The cheap microprocessors of the future are likely to provide 4, 8 or more cores and increased memory bandwidth, further reducing the impact of processor performance on sorting.

Component	ewiz.com price
AMD Athlon 64 X2 Dual Core Processor 4200+ 2GHz (512KB cache/core)	$93.73
512MB DDR 500MHz PC3200	30.39
Hitachi HDS728080PLAT20 80GB disk drive, 2MB cache (2 at $39.50 each)	79.00
ASUS a8v-xe socket 939 k8t890 atx amd motherboard	52.02
CPU Fan Cooler Master RR-KEEE-LEE1-GP	6.18
Case 11.99 (POWMAX MM3800)	22.66
powersupply 9.99 (JPAC ATX500 ATX 5000 W Power Supply)	9.99
fan 1.99 Logicsys Computer 80mm case fan	1.44
Software: Redhat Fedora Core 6, Linux 2.6.19-1.2895.fc6 SMP	0.00
Assembly	35.00
Total price	$330.31
Time budget: 94608000s/32908cents =	2863s/cent

Figure 4: Hardware used for the TokuMergeSort Penny Daytona entry

References

[1] Gnu sort. `http://www.gnu.org/software/coreutils/`, 2006.

[2] Naga K. Govindaraju, Jim Gray, Ritech Kumar, and Dinesh Manocha. GPUTeraSort: High performance graphics co-processor sorting for large database management. Technical Report TR-2005-183, Microsoft, November 2005. Revised March 2006.

[3] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[4] Xing Huang and BinHeng Song. Bytes-split-index sort (bsis). `http://research.microsoft.com/barc/SortBenchmark/BSIS-PennySort_2006.pdf`, April 2006.

[5] J. W. J. Williams. Algorithm 232 (heapsort). *Communications of the ACM*, 7:347–348, 1964.