psort, yet another fast stable external sorting software

Paolo Bertasi, Marco Bressan and Enoch Peserico
Dept. of Information Engineering, Univ. Padova, Italy

May 25, 2008

Abstract

psort is a fast, stable external sorting software available as a binary executable that can sort a disk file interpreted as a sequence of r byte records according to a key of k bytes starting from the ith byte of each record. *psort* is also available under GPL as a C library, allowing one to define one's own key comparison functions, as well as key preprocessing and postprocessing functions to increase the efficiency of comparisons. *psort* can sort $108 \cdot 2^{24} = 1,811,939,328$ records of 100 bytes each, according to an initial 10 byte random key, on a 392.78 U.S. dollar desktop machine in less than $2408.6766 \leq \frac{3 \cdot 365 \cdot 24 \cdot 3600}{39278}$ seconds - i.e. for less than 1 penny according to the Pennysort Daytona Benchmark. A version of psort hand optimized for the Pennysort (Indy) Benchmark can sort a slightly larger input file of $113 \cdot 2^{24} = 1,895,825,408$ byte records on the same machine in the same time.

1 psort

psort is a fast, stable comparison-based external sorting software available under GPL either as a binary executable (with source code) or as a C library. Subsection 1.1 details its architecture; Subsection 1.2 details its interface.

1.1 Architecture

psort exploits a classic two-pass sorting algorithm. In the first pass, the file is split into *runs* slightly smaller than the main memory size; each run is sorted and written to disk in an intermediate file. This file *can* be the original file, or a newly created one. In the second pass, runs are merged into a single sorted file. This file *must* be different from the intermediate file, but, again, it *can* be the original file. Obviously, if the file fits entirely in the main memory, only the first pass is used.

We now analyze the first pass in greater detail. *psort* uses direct mapped, asynchronous I/O to move data between disk and userspace, interleaving computation with data transfers. Read/write buffers should be sufficiently large to amortize the disk head movement and rotation cost and yet sufficiently small to leave enough space to sort large (and therefore few) runs - which is critical to reach high speed in the second pass (see below). The size of the buffers can be specified manually by the user; otherwise *psort* allocates to them 10% of the total available memory, with a minimum of 16MB per buffer.

From the read buffer data is separated between keys and payloads. Keys are preprocessed so that they can then be compared using integer arithmetic. *Microruns* of keys and payloads are then sorted using MergeSort (the initial pass actually sorts sequences of 4 items with SelectionSort). We

chose MergeSort because it is simple, deterministic, and both efficient and predictable in terms of cache behavior. Microrun size can be specified by the user; otherwise, *psort* computes the size of a microrun (in terms of items) as the square root (rounded up to the nearest power of two) of the number of items in a full run. Ideally, microruns should be approximately the size of the processor cache, so that this phase of the first pass only requires, for each item, a read and a write access to the main memory buffer, and yet microruns are as large (and thus as few) as possible. Microruns are then merged using a $k-$way merge-tree into a single sorted run that is directly written to the output buffer (with records being restored to their original form with the key embedded into the payload). Ideally, the data in the $k-$way merge-tree should also fit entirely in cache; the default choice of microrun size attempts to balance it with the size of the merge-tree. *psort* includes an optional cache-refresh mechanism to prevent the stream of payloads from polluting the cache; this option is turned off by default since in most cases its cost outweighs its benefit.

The second pass is simpler. Runs are read (again using asynchronous, direct-mapped I/O) into a read buffer of contiguously allocated userspace memory; from there they are moved into sort buffers, one for each run, that can dynamically grow and shrink. They are then merged using a $k-$way merge tree and written to a write buffer - and from there to disk. Obviously, one can fit only a small fraction of each run into main memory at the same time. This means that, unlike the first pass which involves long sequential reads and writes, the second pass still involves long sequential writes, but rather short reads. This, in turn, means more time lost positioning the disk head, and a lower effective disk bandwidth.

psort uses sort buffers whose size varies dynamically to store data from first pass runs. Each buffer is formed by a number of *microbuffers*. As records enter the sorted output sequence, they are removed from the microbuffers. As soon as a microbuffer is emptied, it can be recycled for other purposes, effectively reducing the size of the buffer. When the amount of data in a buffer falls below a certain threshold (chosen as a user option) the buffer is "refilled" from the appropriate run. In theory, if data were consumed uniformly from all n runs, one could divide the total available buffer space B in such a way that a newly refilled buffer holds $\approx \frac{n}{n^2+n}2B \approx 2B/n$ bytes of data, the previously refilled one $\frac{n-1}{n^2+n}2B$, and so on. This allows reads of about twice the size achievable with static buffers of size B/n. This approach can be highly ineffective, however, if data are not consumed uniformly, and in particular if they are consumed more rapidly from recently refilled buffers. For this reason, *psort* allows one to specify the geometry of the buffer space as a parameter g between 0 and 1, where $1+g$ specifies the ratio between the maximum buffer size and the buffer size B/n in the presence of static buffers. Then, each run is always guaranteed a minimum buffer of size $(1-g)B/n$. The default value of g is 0.5.

1.2 Interface

psort is written in C and is available both as a full fledged binary executable and as a C library. The C library allows one to provide one's own comparison functions in addition to the default one (that treat keys as strings of characters). Also, it allows one to provide preprocessing and postprocessing functions that transform (and, reverse the transformation of) the key into a form more amenable to comparisons using the CPUs integer arithmetic. For example, the 10 byte key of the Pennysort Benchmark is transformed into a pair of (padded) 8 byte integers. Finally, this allows one to use separately the code for the first and second pass of the software.

The executable allows the user to specify the file to sort, interpreted as a sequence of r byte records with a key of k bytes starting from the ith byte of each record. It also allows the user to

specify where the temporary file and final file should be located (including whether one of them overlaps with the original file). Finally, the executable allows the knowledgeable user to tinker with a number of parameters of the sorting code, including microrun size, second pass buffer geometry etc. (see subsection 1.1, above).

2 Pennysort

We tested *psort* using the Pennysort Benchmark. Subsection 2.1 provides the details of the our hardware and OS. Subsection 2.2 provides the actual results of our tests. Guided by our results, we tried to improve the performance of *psort* by tailoring it to the benchmark and to the specific architecture. The performance of this "indy" version of *psort* is also detailed in subsection 2.2.

2.1 Hardware

We attempted to acquire a hardware platform that would deliver the maximum performance at the minimum cost. This was done not only to achieve a good performance of *psort* under the Pennysort Benchmark, but also to understand what are the bottlenecks in today's PC architectures for tasks dealing with large amounts of data that are both CPU and I/O intensive. Our choice for a motherboard was an ASRock ALive NF6P-VSTA with an Nvidia nForce 430 Southbridge - an inexpensive but high performance "Linux friendly" motherboard which supports 4 SATA 3.0Gb channels, for a maximum aggregated traffic of over $420MB/s$. We paired it with 4 Western Digital WD1600AAJS hard drives, which can individually deliver an whopping $100MB/s$ peak read/write rate for extremely large sequential reads/writes on the outer rim of the disk. We configured them with GNU/Linux (Gentoo) "vanilla" software RAID. As a filesystem, we tested XFS, JFS, ReiserFS and ext3FS. The best performers where XFS and JFS (with the former outperforming the latter by an almost negligible amount). In both cases the CPU (see below) usage to saturate the disk transfer rate was negligible - less than 3%. Thus, we finally settled for XFS. The best performance was achieved with a stripe size of $128KB$. Note that this is a very "disk heavy" PC, with about half the total cost being taken by the 4 disks.

RAM choice must take into account three parameters: size, speed and price. A RAM that is twice as large approximately (in fact, more than) doubles the size of runs in the first pass. This, in turn allows reads that are four times as long during the second pass. Of course, memory today is only available in sizes that are powers of 2. It turned out that the best compromise was $2GB$. RAM speed is another important parameter. PC4200 RAM is about a dozen times faster than the southern bridge *in theory.* In practice, we found that accessing RAM can have a large number of "hidden" costs - e.g. due TLB lookups and to the fact that it is accessed in whole "cache lines". Even just two or three read+write passes can consume the majority of the available RAM bandwidth, and it is extremely difficult to coax the compiler to overlap RAM to cache transfers with processor operations. In practice, it turned out that even using 2 banks of OCZ 800MHz PC6400 RAM with CAS 4 latency almost half the "CPU" time taken during the first pass was spent accessing the RAM.

The choice of the actual processor strongly depends on that of the other components - probably more than on the "number crunching" power of the CPU itself. In particular we chose a cheap, single core Athlon LE running at 2.4Ghz, with $128KB$ of L1 cache and $1MB$ of L2 cache. The total cost of the hardware at NewEgg.com on May 19th 2008 was 357.78 \$. Under the Pennysort

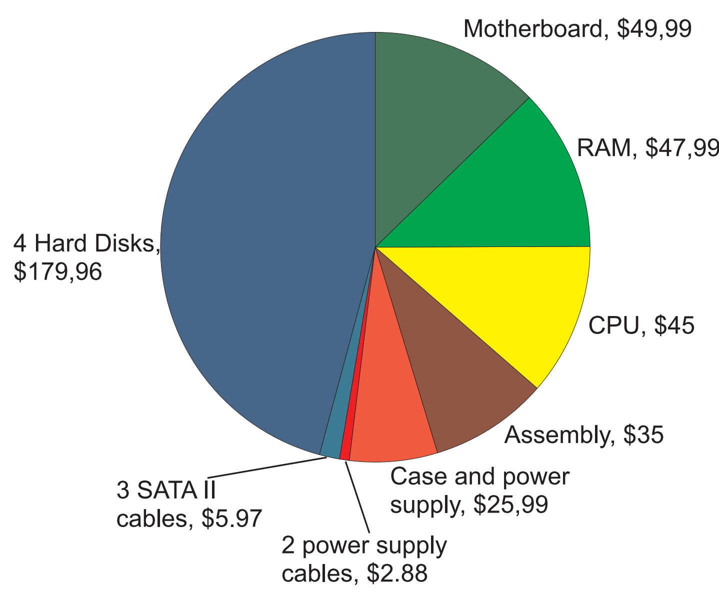

Figure 1: Hardware costs

formula, adding the mandatory 35$ "assembly fee", this allowed us a total time budget of slightly more than 2408.6766 seconds.

2.2 The actual test

We tested *psort* on the hardware described above, compiling it with gcc with the flags `-O3 -m64 -msse2 -mtune=k8 -march=k8 -funroll-loops -mfpmath=sse -funsafe-loop-optimizations -B /usr/share/libhugetlbfs/ -Wl,--hugetlbfs-link=B`. We positioned the input file on an appropriately sized partition on the outer rim of the disks, overwrote it during the first pass, and created the output file in a second partition during the second pass. The first pass was slightly limited by the CPU, or, more correctly, by the combination of CPU and RAM. More expensive CPUs did not yield sufficient increases in performance to justify their use. The second pass was entirely limited by the disks. *psort* (using 2^{16} record Mergesort and a 2^8-way merger-tree, $50MB$ read/write buffers, overwriting the initial file with the intermediate file) sorted $108 \cdot 2^{24} = 1,811,939,328$ records taking less than 2405 seconds. We then manually "retooled" *psort* into an "indy" version adapted solely for the Pennysort benchmark (eliminating unnecessary "general purpose sorting" code, manually unrolling loops etc.). This yielded a small, but observable gain in performance. *psort* indy managed to sort $113 \cdot 2^{24} = 1,895,825,408$ records in less than 2407 seconds.

It is interesting to compare our results with the prediction of 10 years ago by Gray et al. (http://www.rrsd.com/psort/ms/ps.htm) that price-performance would double yearly for the next 10 years, yielding $1.50TB$ for 1 penny by 2008. Instead, price-performance has "only" increased by slightly over two orders of magnitude, i.e. a factor of about 1.6/year (almost exactly matching Moore's Law).

Shopping Cart

Print

Qty.	Product Description	Savings	Total Price
1	Rosewill RCW-305 6" /Serial ATA I, II 5.25" Male to 15P Serial ATA Female Power Adapter Cable /Multi-Color - Retail Item #: N82E16812119024 Return Policy: Standard Return Policy		$0.99
3	OKGEAR 36" SATA II cable Model GC36AKM22 - Retail Item #: N82E16812123174 Return Policy: Standard Return Policy		$5.97 ($1.99 each)
1	Nippon Labs 4-Pin PC power to 2 x SATA Converter Cables Model POW-SATA-2 - Retail Item #: N82E16812816015 Return Policy: Standard Return Policy		$1.89
1	Linkworld 3230-09-C2222U Black Steel ATX Mid Tower Computer Case 430W Power Supply - Retail Item #: N82E16811164092 Return Policy: Standard Return Policy		$25.99
4	Western Digital Caviar SE WD1600AAJS 160GB 7200 RPM SATA 3.0Gb/s Hard Drive - OEM Item #: N82E16822136075 Return Policy: Limited 30-Day Return Policy Select An Optional Extended Warranty Plan		$179.96 ($44.99 each)
1	ASRock ALiveNF6P-VSTA AM2+/AM2 NVIDIA GeForce 6150SE / nForce 430 Micro ATX AMD Motherboard - Retail Item #: N82E16813157125 Return Policy: Limited 30-Day Return Policy Select An Optional Extended Warranty Plan		$49.99
1	OCZ Platinum Revision 2 2GB (2 x 1GB) 240-Pin DDR2 SDRAM DDR2 800 (PC2 6400) Dual Channel Kit Desktop Memory Model OCZ2P800R22GK - Retail Item #: N82E16820227139 Return Policy: Memory (Modules, USB) Return Policy Select An Optional Extended Warranty Plan		$47.99
1	AMD Athlon 64 LE-1620 2.4GHz Socket AM2 45W Single-Core Processor Model ADH1620DHBOX - Retail Item #: N82E16819103198 Return Policy: Processors (CPUs) Return Policy Select An Optional Extended Warranty Plan		$45.00
		Subtotal:	$357.78
	Calculate Shipping Zip Code: 02108 UPS Guaranteed 3 Day Service -- $44.67	Shipping:	$44.67
	Redeem Gift Certificates Claim Code: Security Code:	Gift Certificates:	$0.00
	Apply Promo Code	Promo Code:	$0.00

Figure 2: The NewEgg.com shopping cart for our hardware, not including the 35$ “assembly fee”.