

# *psort*, yet another fast stable sorting software

Paolo Bertasi, Marco Bressan, and Enoch Peserico

Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Padova, Italy  
psort@dei.unipd.it

**Abstract.** *psort* was the fastest sorting software in 2008 according to the Pennysort benchmark, sorting 181GB of data for 0.01\$ of computer time. This paper details its internals, and the careful fitting of its architecture to the structure of modern PCs-class platforms, allowing it to outperform state-of-the-art sorting software such as *GNUsort* or *STXXL*.

## 1 Introduction

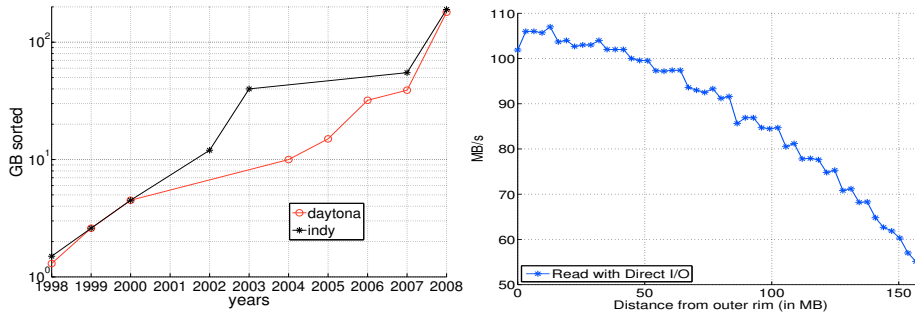
This paper details the internals of *psort*, the fastest sorting software of 2008 according to the Pennysort benchmark [3]. This introduction provides a brief history of Pennysort and related sorting benchmarks (Subsection 1.1), and a simple taxonomy of the “mainstream” sorting techniques for large datasets that helps put our work into perspective (Subsection 1.2), followed a high-level overview of *psort* and of the organization of the rest of this paper (Subsection 1.3).

### 1.1 Datamation, Pennysort, and other sorting benchmarks

Datamation [4] defined the first [15] public sorting benchmark - sort a million 100 byte records, initially in random order, according to the first 10 bytes *from disk to disk*.

In 1985 the time required to complete the Datamation benchmark was almost 1 hour; but within 10 years it had dropped to a few seconds, and it appeared that it would soon become obsolete.

Therefore, in 1995, [22] proposed two new sorting benchmarks, MinuteSort and PennySort. MinuteSort, aimed at supercomputer-class platforms, requires sorting as many records as possible within 1 minute. Pennysort, aimed at PC-class platforms, requires sorting as many records as possible with 0.01\$ of computing time, assuming that the price of a machine is amortized over 3 years (thus, on a  $x$  dollar machine, one is allowed  $\frac{0.01}{x} \cdot 3$  years of computing time). In both cases record format coincides with that of the datamation benchmark. Also, in both cases a distinction is made between “Daytona” software, designed for general purpose sorting, and “Indy” software, specifically optimized for the benchmark. Over the years a number of slight refinements have been added to the rules, and new benchmarks such as TeraSort and JouleSort have been introduced. All details can be found at [3].



**Fig. 1.** The Pennysort Benchmark record history ( $psort = 2008$ ) **Fig. 2.** Disk bandwidth as a function of distance from outer rim

## 1.2 A simple taxonomy of sorting

Hundreds of articles and even entire books (e.g. [18]) have been written on sorting. This subsection provides a simple taxonomy of sorting techniques for large data-sets to help put  $psort$  into perspective compared to existing software.

Virtually all efficient sorting software today is either *distribution*-based, *merge*-based, or a hybrid of the two. Distribution-based sorting distributes the data into two or more bins, in such a way that for each pair of bins all keys of one precede all keys of the other; then it recursively sorts each bin. Merge-based sorting splits the input into two or more runs, sorts each run, and then merges the sorted runs. Distribution and merge-based sorting can obviously be combined. For example, one might use the former to “locally” sort separate runs, that are in turn “globally” merged - the approach of Alphasort [22].

Distribution-based sorting has two major advantages over merge-based sorting. First, it can be easily performed completely in parallel, and thus is virtually the only approach used today - at least at the “global” level - for sorting on large PC-clusters (e.g. [13, 10, 21]). Second, it can be very efficient in terms of number of key lookups - this makes it a favorite of all past record holders of the Pennysort benchmark (e.g [14, 25, 19, 23, 5]) at least for some phases of the sort.

Merge-based sorting is instead intrinsically “comparison based” and thus requires at least  $n \lg(n)$  key lookups to sort  $n$  keys ([18]) though this disadvantage is more apparent than real, as we shall see. On the other hand, merge-based sorting always results in well-balanced sub-problems of predictable size; this makes it more “robust” and easier to fine tune to a memory hierarchy. This predictability could in theory be achieved by a careful selection of the thresholds between bins in distribution-based sorting, but only at a high cost (see e.g. [18]).

## 1.3 Our results

$psort$  is a fast stable external sorting software (available as source, binary and/or library) that can sort collections of records of arbitrary size according to an arbitrary infix.  $psort$  sorted 181GBs of data for 0.01\$ of computer time in 2008,

making it the fastest sorting software in 2008 according to the Pennysort benchmark. The careful fitting of its architecture to the structure of modern PC-class platforms (made easier by its pure merge-based nature) allows it to outperform state-of-the-art sorting software such as *GNU sort*[2], *qsort*[1], *C++ STL sort* or *STXXL*[12], even for record and key sizes and distributions quite different from those of the Pennysort benchmark.

In order to understand *psort* optimizations, one has to understand the complex architecture of modern PCs to a greater level of detail than that offered by most theoretical models today. Section 2 provides this information.

Section 3 describes *psort* itself and its tuning to a modern PC.

Section 4 describes our experimental results, comparing *psort* with other state of the art sorting software - both under the Pennysort rules (including the selection of the “best” PC to run it) and under a number of different scenarios.

Finally, Section 5 summarizes our results and discusses their significance before concluding with the bibliography.

## 2 The anatomy of a PC

Current hardware is extremely complex. While a number of abstract models from the past decades attempt to capture the main aspects of modern architectures - hierarchical memory [6], block transfer[7, 9], pipelining[11], parallelism[26] etc. - they are generally insufficient to abandon the ivory tower of big  $O$  notation and squeeze out of a machine at least 50% of its the peak performance ([8]). It is impossible to review all the details of the performance of a modern PC, but this section provides a comprehensive overview of those (often disregarded) factors that can be crucial to data-intensive software such as sorting - starting from disks and filesystem (Subsection 2.1), then moving to motherboard and memory (Subsection 2.2), and finally to the processor chip (Subsection 2.3).

### 2.1 Disks and filesystem

Modern disks provide the abstraction of a (logical) linear array of data blocks, with access to a sequence of contiguous blocks requiring a fixed *seek time* independent of the amount of data (typically of the order of  $10ms$ ), plus a *transfer time* that is directly proportional to it (typically  $10 - 100ms/MB$ ). Obviously, to approach peak performance, transfer time should dominate seek time (contiguous data transfers indicatively of  $1MB$  or more). Transfer time is lower for data logically closer to the beginning of the array, corresponding physically to the area of the disk closer to the outer rim; this should be regarded as the “high performance” portion of the drive (see figure 2).

Multiple disks (usually up to  $4 - 8$ ) can be used in parallel (RAID 0) as a single disk with the same seek time but proportionally larger transfer speed, by splitting data into “stripes” divided, in a round robin fashion, between different disks. This can be done in hardware or by the filesystem/OS - we found the latter approach effective and of minimal CPU cost.

Most software today does not access the disk directly, but through a filesystem. Filesystems offer a lot of functionality over raw disk access, but at a price

in terms of performance (in fact, many applications with high performance disk access, such as DBMSs, bypass the filesystem altogether); however, most sorting benchmarks (including Pennysort) enforce the use of a filesystem for disk access. To minimize the CPU and memory system overhead, it is then crucial to access data through asynchronous, direct I/O from the device directly into user-space.

## 2.2 Motherboard and main memory

Disks (or rather their on-board caches) communicate with the motherboard through an interface generally designed to have higher throughput (e.g.  $300\text{MB/s}$  for SATA2) than the drives themselves, to ensure interface longevity. Motherboards typically support 4 to 6 (rarely 8) of these interfaces, all connected to a chip known as the *south bridge*. The south bridge aggregates the traffic of disks and other devices (e.g. keyboard, network card) and usually directs it to another chip, the *north bridge*, that controls traffic to/from/between the main memory, the processor chip and graphics hardware (in some architectures at least part of the north bridge is incorporated in the processor).

Modern south and north bridges can manage transfers of large data aggregates directly between memory and disk with only minimal CPU involvement (“Direct Memory Access”). Bandwidth between bridges is typically lower than the sum of the disk interface bandwidths supported by the motherboard, but is rarely a bottleneck for all but the fastest and largest same-generation disk RAIDs (typical values are  $350 - 800\text{MB/s}$ ). The bandwidth between the other hardware connected to the north bridge is higher still, typically several  $\text{GB/s}$ .

There are a few more issues to consider in terms of processor/memory communication. First, over a hundred processor cycles can typically pass between a request for a datum in main memory and its availability on the processor chip. Second, data is transferred to the processor chip only in multiples relatively large *cache lines* (typically 128 to 512 bytes); and if 2 bytes of a datum belong to 2 different cache lines, both lines are accessed. Third, virtual memory addresses have to be translated to physical addresses; while the most recent translations are cached in the processor chip, very sparse memory accesses will force additional accesses to the translation tables [24]. This problem can be minimized using large memory pages, and thus smaller translation tables for the same space.

## 2.3 The processor chip

The basic components of the processor chip are its CPU(s) and its cache(s). Most modern processors have 2 levels of cache, the first smaller (up to a few hundred  $\text{KB}$  vs. one or more  $\text{MB}$ ), but faster to access (typically a few cycles vs.  $10 - 30$ ). Each memory block can only be placed in a small number of cache locations, the *associativity* of the cache (typically  $4 - 16$  for L1 caches,  $8 - 32$  for L2 caches). According to [17] associativity above 8 provides few benefits, but we have found this to be true only if data layout is very carefully planned [20]. All decisions on which data to keep in the cache are made by the processor, typically replacing data used furthest in the past with data more recently accessed. This can result

in undesired behaviours: for example large streams of read-once data can “pollute” the cache, evicting data used repeatedly but somewhat infrequently. The programmer can sometimes attempt to influence cache replacement by issuing extra memory requests to “nail” or prefetch data into the cache.

In general modern processors employ sophisticated circuitry to reorganize instructions, “guess” yet unavailable data, and backtrack from incorrect guesses. This makes it extremely difficult to understand whether, and how, a code snippet can be redesigned to improve efficiency: “optimizations” that increase performance on one system often decrease it on others. Fine tuning through experimentation on the target architecture can net substantial performance gains.

### 3 The anatomy of *psort*

*psort* is a fast, stable sorting software designed for large data sets on PC-class platforms. *psort* is a simple, merge-based sorter that first sorts individual data runs approximately the size of main memory, and then merges them into a single sorted output. In fact, *psort*’s high-level simplicity is probably the source of its performance, allowing careful, low-level tuning to the complex structure of today’s PCs. Subsection 3.1 provides a “global overview” of *psort*. Subsections 3.2 and 3.3 provide the details of the first and second phase of the sort.

#### 3.1 A theorist’s view of *psort*

*psort* is a merge-based sorter tuned to the memory hierarchy. For each pair of consecutive hierarchy layers (e.g. main memory and disk) the structure of *psort* depends on the size  $S_0$  of the smaller and on the size  $S_1$  of the larger.

In a nutshell *psort* implements a  $p$ -pass merge between the two layers as follows. Choose a block size  $B$  such that  $(\frac{S_0}{B})^p \approx \frac{S_1}{B}$ . To implement the 1<sup>st</sup> pass, split the data in the larger layer into  $\approx \frac{S_1}{S_0}$  data runs of size  $\approx S_0$  and sort each run in the smaller layer. Data are sorted in the smallest layer, the L1 cache, using a simple mergesort. To implement the 2<sup>nd</sup> pass, split the sorted runs into sets of  $w = \frac{S_0}{B}$  runs each, and merge each set into a sorted run of size  $wS_0$  using a  $w$ -way selection tree merger [18], where each way is assigned a buffer of size  $B$ . The entire tree then always resides in cache, so that, during the pass, each item is brought into, and evicted from, the cache only once - and each transfer is amortized over a block of size  $B$ . Similarly, to implement the  $i$ <sup>th</sup> pass, merge sets of  $w$  runs from the  $(i - 1)$ <sup>th</sup> pass into sorted runs of size  $w^{i-1}S_0$ .

The minimum block size that still allows an efficient transfer places an upper bound on the number of ways of a merge, and thus a lower bound on the number of passes. It is chosen to minimize  $(p - 1)(t_B + t_{peak})$  where  $t_B$  is the amortized read time per bit using blocks of size  $B$ , and  $t_{peak}$  is the amortized write time per bit at the peak transfer rate using blocks of size comparable to  $S_0$  (since during a merge phase, the  $w$  input buffers must be small, but there is only one write buffer, that can be large). Note that the first term of the product is  $(p - 1)$ : in the first pass, one can essentially perform both reads and writes of a size comparable to  $S_0$ . In practice, on modern computing platforms,  $p$  almost

always equals 2 or (more rarely) 3 at all layers of the memory hierarchy, with the possible exception of the lowest two (memory and disk), where data sets small enough to fit in memory can be sorted with a single read/write pass of the disk(s). This is not the case for the Pennysort benchmark, where data is usually one to two orders of magnitude larger than memory - making  $p = 2$  and thus entailing 2 “large writes” passes, 1 “large reads” pass (the first), and 1 “small reads” pass (the second). In this case,  $B$  is chosen to match the size of a few disk tracks (i.e. a few *MBs* - to minimize the overhead of seek time) times the number of disks in the RAID.

Two things are worth noting. First, the first pass may always be performed in place, with at most a “slack” of a single run; this is generally impossible for other passes. Second, as noted e.g. in [12], utilizing RAID at the disk layer is asymptotically suboptimal (as the number of disks grows to infinity) compared to (randomized) strategies that control disks independently. However, such strategies have the potential of actually being asymptotically slower than pure RAID on pathological inputs. Perhaps more importantly, their “sweet spot” requires larger disk arrays and sorts than those encountered in practice when dealing with today’s PC-class platforms, where they can actually end up being slower even on “average” inputs due to their small (in fact *asymptotically* negligible) fluctuations in disk utilization (see Subsection 4.3) and their reduced ability to exploit the faster zones of the disk (see Subsection 4.2).

### 3.2 The first phase

The first phase of *psort* essentially involves reading a data run approximately the size of the main memory from disk, sorting it in memory, and writing it back to disk. The devil is in the details.

For I/O efficiency, *psort* makes use of direct asynchronous I/O that transfers data between the disks and a set of userspace buffers (dimensioned so as to achieve near peak transfer rate without consuming too much memory): this requires minimal CPU involvement (even with software RAID, less than 4% of CPU time) and bypasses the space and time costs of moving data through kernelspace buffers. With two buffers one can guarantee that the disks never fall idle: while the CPU operates (reading or writing data) on one, disks exchange data with the other, thus completely overlapping I/O with computation.

As soon as a read buffer is filled *psort* must transfer data to its main memory space. Since this requires moving the data through the processor’s cache, it piggybacks some computation onto the transfer, acting on data “microruns” of size roughly equivalent to the L2 cache. More precisely, if keys are sufficiently small compared to the records’ remaining payload, it separates keys and payloads, attaching to each key a pointer to the corresponding payload. *psort* at this stage also offers the possibility of restructuring keys (e.g. from big to small endian) to make later comparisons more efficient. *psort* then sorts those keys (and eventually reshuffles the payloads) and finally writes the whole “microrun” to a new memory area. This involves, for each datum, at most one write and one read in memory that would be performed anyway to empty the read buffers.

The sorting algorithm used in *L1* cache for the microruns is a simple mergesort, with some tweaks. First, a single pass on the data (the same that possibly detaches keys and records) sorts sets of 4 consecutive elements using a simple selection sort. Second, careful placement of data ensures that only  $5/4$  of the total space  $s$  occupied by the records (or by the detached keys) is used, rather than the “common” factor 2 of mergesort: this is achieved by first sorting half of a microrun in space  $\frac{s}{2} + \frac{1}{2} \cdot \frac{s}{2}$  (see [18]), then the other half (for a total space of  $\frac{s}{2} + \frac{s}{2} + \frac{1}{2} \cdot \frac{s}{2} = \frac{5s}{4}$ ), and finally piggybacking the merge of the two halves over the transfer out of the read buffer. Third, *psort* offers the opportunity to minimize program branches. As a first option, it can use bitwise — rather than logical — ANDs and ORs when comparing multi-word keys. As a second option, it can use the result of a comparison (and its inverse) as an offset to a pointer to the new positions of the keys, saving a branch at the cost of a few extra operations. The data sorted in *L1* are then merged in *L2*.

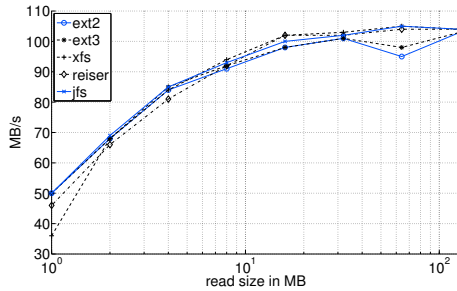
Sorted microruns are then merged in a second pass (and possibly a third, depending on the ratio between the size of the memory and of the *L2* cache, and the associativity of the latter). Two important potential hurdles at this stage are associativity misses (in early experiments these reduced performance by as much as 20%) and “stream pollution” of the cache (see Section 2). *psort* employs a careful data layout to minimize the former, and offers against the latter the option of a periodic cache refresh through dummy reads.

The output of the final pass is directly written to the I/O buffers, potentially recombining keys and payloads, and possibly (if there is no second phase) inverting the initial restructuring of the keys. Again, double buffering allows full overlap between I/O and computation.

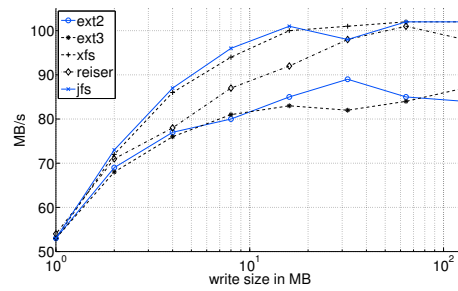
### 3.3 The second phase

The second phase (which only takes place if the dataset does not fit in main memory) is much simpler:  $w$  sorted runs at a time are streamed from disk and merged (with the same code that merges microruns in the first phase), and the output is streamed back to disk. Recalling Subsection 3.1, it is reasonable to use a single merge pass if the number of data runs (i.e. the ratio between data and memory size) does not exceed the ratio between the size of the memory and that of one “efficient” read from disk.

Data is read with direct asynchronous I/O into (userspace) dynamically sized buffers, one per run. When the amount of data in a buffer falls below a threshold the buffer is “refilled” from the appropriate run. In theory, if data were consumed uniformly from all  $w$  runs, one could divide the total available buffer space  $B$  in such a way that a newly refilled buffer held  $\approx \frac{w}{w^2+w} 2B \approx 2B/w$  bytes of data, the previously refilled one  $\frac{w-1}{w^2+w} 2B$ , and so on. This would allow reads of about twice the size achievable with static buffers of size  $B/n$ . This 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 the user to specify buffer geometry, choosing a tradeoff between safety and optimization.



**Fig. 3.** Filesystem read speed as a function of read size



**Fig. 4.** Filesystem write speed as a function of write size

## 4 *psort* vs. the competition

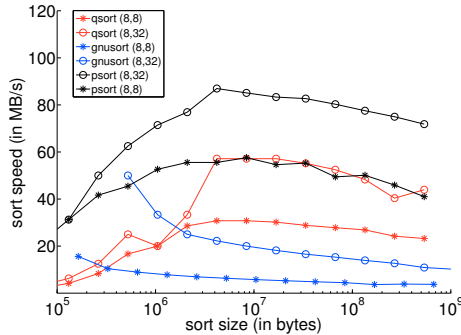
This section compares *psort* to its competitors, in terms of the Pennysort benchmark (Subsections 4.1 and 4.2) and also in a wider variety of sorting scenarios (Subsection 4.3).

### 4.1 Choosing and configuring the hardware

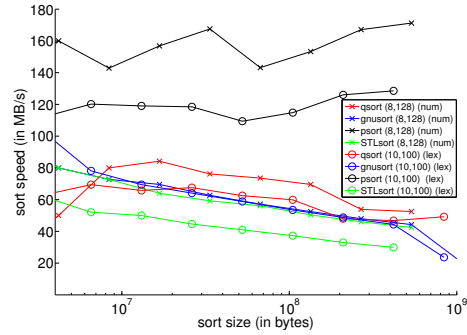
To test *psort* we looked for a hardware platform delivering maximum performance at the minimum cost, to maximize our results under the Pennysort benchmark, but also to understand the bottlenecks in today’s PC architectures for data-intensive tasks.

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 supporting 4 SATA2 channels, for a maximum aggregated traffic of  $500MB/s$ . We paired it with 4 Western Digital WD1600AAJS drives, which can individually deliver a whopping  $100MB/s$  peak read/write rate. We configured them with GNU/Linux (Gentoo) “vanilla” software RAID. As a filesystem, we tested SGI’s XFS, IBM’s JFS, ReiserFS and ext2fs. The best performers where XFS and JFS, with JFS slightly better overall but XFS outperforming it very slightly for the read and write sizes of interest to us (see figures 3 and 4). In both cases CPU usage to saturate the disk transfer rate was negligible - less than 2%. 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 (see figure 8).

RAM choice must take into account three parameters: size, speed and price. A RAM that is twice as large more than doubles the size of runs in the first pass. This, in turn, allows reads that are over 4 times longer during the second pass. It turned out that the best compromise was  $2GB$ . RAM speed is another important parameter. PC4200 RAM has a *theoretical* “peak” transfer rate of  $4.2GB/s$  - an order of magnitude faster than the southern bridge. 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



**Fig. 5.** *psort* vs *qsort* and *GNU sort* for small record sizes



**Fig. 6.** *psort* vs *qsort*, *STL sort* and *GNU sort* for large record sizes

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 during the first pass was spent accessing 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. We chose a cheap, single core Athlon 1620LE 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 19<sup>th</sup> 2008 was 357.78\$. Under the Pennysort formula, adding the mandatory 35\$ “assembly fee”, this allowed us a total time budget of slightly more than 2408.6766 seconds.

## 4.2 Pennysort results

We tested *psort* on the hardware described in Section 4.1. We compiled it with: `-march=k8 -O3 -funroll-loops -funsafe-loop-optimizations -B /usr/share/libhugetlbfs/ -Wl,--hugetlbfs-link=B`.

We positioned the input file into an appropriately sized partition on the outer rim of the disks, overwrote it with the output of the first pass, and had the second pass create the output file in a second partition. This guaranteed 3 of the 4 passes took place on the fastest partition of the disk, and only 1 on a slower partition. Note that, had we been using independent disks for the intermediate files (RAID would still have been necessary for the initial input and final output, since the rules of the Pennysort benchmark enforce a single file for each), at most 2 passes could have taken place on the faster partition.

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 cache merge and a  $2^8$ -way merger-tree, 50MB read/write buffers, overwriting the initial file with the intermediate file)

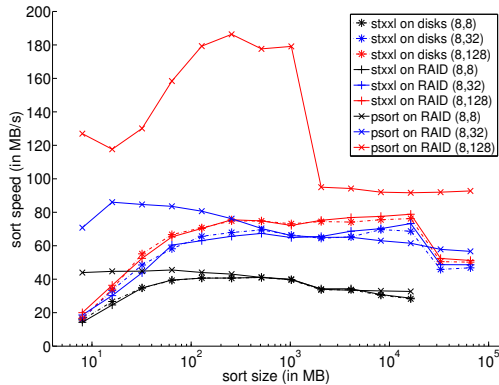


Fig. 7. *psort* vs *STXXL*

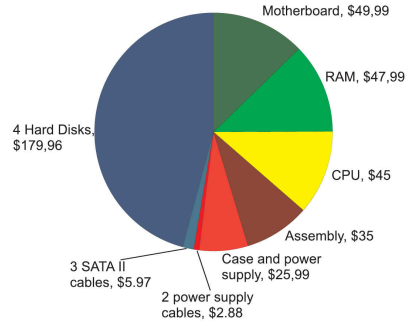


Fig. 8. Budget of the testbed machine

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.

### 4.3 Other scenarios

In order to evaluate *psort* outside the Pennysort context, we compared it to some state-of-the-art sorting libraries under different scenarios. We had *psort* compete against the high performance (non-stable) external sorting *STXXL* library [12] on sorting 128, 32 and 8 byte records according to the first 8 bytes for a variety of input sizes, from 10MB to 100GB (we only managed to run *STXXL* with power-of-2 key size, which seems in line with [12]). *psort* speed was always 20% higher or more for 128 byte records, and either higher (for small sort sizes) or essentially identical for 32 byte and 8 byte records (see figure 7). It should be noted that *STXXL* has a large number of tuning parameters, that we tweaked trying to achieve optimal performance. We found the most crucial one to be block size - in our scenario 32MB (vs. the default value of 2MB) offered the best performance. Interestingly, and contrary to what one might expect from [12], using independent disks never increased performance significantly, and sometimes even slightly decreased it, due to the slight fluctuations it introduces in disk usage. Apparently the theoretical advantage of using independent disks translates into a practical advantage only for significantly larger sorts.

We also had *psort* compete *in main memory* against both *qsort* [1], *GNU sort* [2] and the C++ STL *sort* (which do not support external memory sorting), on sorting 128, 32 and 8 byte records according to the first 8 bytes, as well as 100 byte records according to the first 10 bytes (the datamation record format), again for a variety of input sizes (see figures 5 and 6). *psort* speed was always higher, from 20% to 300%.

## 5 Conclusions

This section briefly summarizes our results, discusses their significance, and looks at future directions both in terms of efficient sorting and other data-intensive software (Subsection 5.2) and of the Pennysort benchmark (Subsection 5.1).

### 5.1 10 years of Pennysort

It is interesting to compare our results with the prediction of 10 years ago by Gray et al. [16] 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 an average factor of about 1.6/year (see figure 1). This almost exactly matches Moore’s Law - but looking at the 1998 winners it is easy to see that improvement is not due solely to better hardware. We sorted more than 120 times the data of Gray et al. [16] using 3 times the time budget, a set of disks with about 15 times the peak (total) bandwidth, and a  $2.4GHz$  Athlon 1620LE vs. a  $266MHz$  Pentium II - the latter having a lesser comparative gap to the memory. Software engineering advances are then responsible for at least a factor 2 – 3 of improvement (note that *psort*’s basic algorithms are decades old).

After 10 years Pennysort is still an excellent benchmark for the lower levels of the memory hierarchy - less so for the processor. As reflected by our budget (see figure 8) our machine had superb disks, an excellent motherboard, good memory, and one of the cheapest processors of the market. While the Pennysort “spirit” has remained the same over the years, the rules keep changing slightly every year. We believe this reduces the value of Pennysort as a benchmark for the *evolution* of PCs and sorting software. On the other hand, we advocate one change: stipulating that the prices used to compute the time budget be taken from a list made public a few months before the submission deadline. This would avoid last minute shopping (and coding) frenzies and/or heavy impact on the relative results of different entries caused by fluctuations of hardware prices.

### 5.2 Some (ugly?) lessons from *psort*

Unlike all previous winners of the Pennysort benchmark, *psort* is completely merge-based, rather than a distribution-based hybrid. This might be one reason of its success. Merge-based software tends to require more key lookups at the highest levels of the memory hierarchy - but these levels are no longer the bottleneck. On the other hand, merge-based sorting software is somewhat more predictable - and thus it can be fitted more carefully to the lower levels of the memory hierarchy, avoiding performance losses where they count.

*psort* exploits many simple tricks that can be expected to boost the performance of any data-intensive software. Perhaps the ultimate lesson of *psort* is that a lot of ugly work is necessary to transform any simple, elegant algorithm into a software that preserves at least 50% of the performance potential of today’s PCs. This explains why sorting, despite its relative simplicity, its long history, and its great practical importance, can still see non-trivial improvements through simple algorithm engineering (rather than algorithmic breakthroughs).

## References

1. *The GNU C library - Array Sort Function*. <http://www.gnu.org/>.
2. *GNU Coreutils - sort*. <http://www.gnu.org/>.
3. *Sort Benchmark Home Page*. <http://www.hpl.hp.com/hosted/sortbenchmark/>.
4. A measure of transaction processing power. *Datamation*, 31(7):112–118, 1985.
5. A. M. Aaron Darling. *DMSort: A PennySort and Performance/Price Sort*. <http://www.hpl.hp.com/hosted/sortbenchmark/DMSort.pdf>.
6. A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. of ACM STOC 1987*, pages 305–314.
7. A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. *Proc. of IEEE FOCS 1987*, pages 204–216.
8. A. Ailamaki, D. J. Dewitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go. In *Proc. of VLDB 1999*.
9. B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:12–2, 1994.
10. A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-performance sorting on networks of workstations. *Proc. of ACM SIGMOD 1997*, 26(2):243–254.
11. G. Bilardi, K. Ekanadham, and P. Pattnaik. Optimal organizations for pipelined hierarchical memories. In *Proc. of ACM SPAA 2002*, pages 109–116.
12. R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *Proc. of ACM SPAA 2003*, pages 138–148.
13. D. J. Dewitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proc. of PDIS 1991*.
14. N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: High performance graphics coprocessor sorting for large database management. In *Proc. of ACM SIGMOD ICMD 2006*.
15. J. Gray. A measure of transaction processing 20 years later. *CoRR*, abs/cs/0701162, 2007.
16. J. Gray, J. Coates, and C. Nyberg. Price/performance sort and 1998 pennysort performance / price sort and pennysort, 1998.
17. J. Hennessy, J. L. Hennessy, D. Goldberg, and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1st edition.
18. D. E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley, 1998.
19. Z. W. Lei Yang, Hui Huang and T. Song. *SheenkSort: 2003 Performance / Price Sort and PennySort*. <http://www.hpl.hp.com/hosted/sortbenchmark/SheenkSort.pdf>.
20. K. Mehlhorn and P. Sanders. Scanning multiple sequences via cache memory. *Algorithmica*, 1(35):75–93, 2003.
21. M. H. Nodine and J. S. Vitter. Greed sort: optimal deterministic sorting on parallel disks. *J. ACM*, 42(4):919–933, 1995.
22. C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet. Alphasort: A cache-sensitive parallel external sort. *VLDB J.*, 4(4):603–627, 1995.
23. L. Z. Peng Liu, Yao Shi. *2002 Performance / Price Sort and PennySort*. <http://www.hpl.hp.com/hosted/sortbenchmark/THSort.pdf>.
24. N. Rahman, R. Cole, and R. Raman. Optimised predecessor data structures for internal memory. In *Proc. of WAE 2001*.
25. R. Ramey. *Postman's Sort*. <http://www.rrsd.com/>.
26. L. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, 1990.