Engineering Cache-oblivious Sorting Algorithms

Master’s thesis by

Kristoffer Vinther

Sorting Algorithms

- The Problem

  Given a set of elements, put them in non-decreasing order.

- Motivation

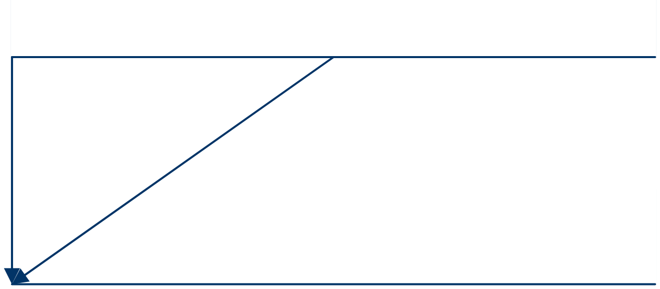
  Very commonly used as a subroutine in other algorithms (such as graph-, geometric-, and scientific algorithms).

  A good sorting implementation is thus important to achieving good implementations of many other algorithms.

  Performance of sorting algorithms seem greatly influenced by many aspects of modern computers, such as the memory hierarchy and pipelined execution.

Sorting Algorithms – Binary mergesort

- Ex. binary mergesort:
  1. Split elements into two halves.
  2. Sort each half recursively.
  3. Make space for sorted elements and *merge* the sorted halves



Engineering Cache-oblivious Sorting Algorithms

Master's thesis by
Kristoffer Vinther

Cache-oblivious – Motivation

- The presence of a memory hierarchy has become a fact of life.
- Accessing non-local storage may take a very long time.
- Good locality is important to achieving high performance.

	Latency	Relative to CPU
Register	0.5 ns	1
L1 cache	0.5 ns	1-2
L2 cache	3 ns	2-7
DRAM	150 ns	80-200
TLB	500+ ns	200-2000
Disk	10 ms	10^7

Cache-oblivious Algorithms – Models

Random Access Memory

- All basic operations take constant time.
- Complexity is the number of operations executed (instruction count), i.e. the total running time of the algorithm.

External Memory

- Computation is done in *main memory*.
- Data is brought to and from main memory in *I/O*s, explicitly controlled by the algorithm
- Complexity is the number of I/Os done by the algorithm.

Cache-oblivious

- Algorithms designed for the RAM model; algorithm does not control the I/Os.
- Algorithms analyzed for the EM model.
- Complexity is *both* the instruction count and the number of I/Os (memory transfers) incurred by the algorithm.

Cache-oblivious Algorithms – Sorting

Random Access Memory

- Complexity of binary mergesort: $\mathcal{O}(N \log N)$.
- Complexity of any (comparison-based) sorting algorithm: $\Omega(N \log N)$.

External Memory

- Complexity of binary mergesort: $\mathcal{O}(N/B \log N/M)$.
- Complexity of any sorting algorithm: $\Omega(N/B \log_{M/B} N/M)$.

- Binary mergesort optimal in External Memory only if $M = 2B$.
- What if $M > 2B$? *Multiway mergesort* incurs $\mathcal{O}(N/B \log_{M/B} N/M)$ I/Os, given the right M and B.
- Multiway mergesort is suboptimal with the wrong M and B.
  - M and B cannot in general be determined.
  - Running the algorithm on a machine different from the one to which it was designed.
- Funnelsort and LOWSCOSA incurs $\mathcal{O}(N/B \log_{M/B} N/M)$ memory transfers, without knowing M and B.

Cache-oblivious Algorithms – Assumptions

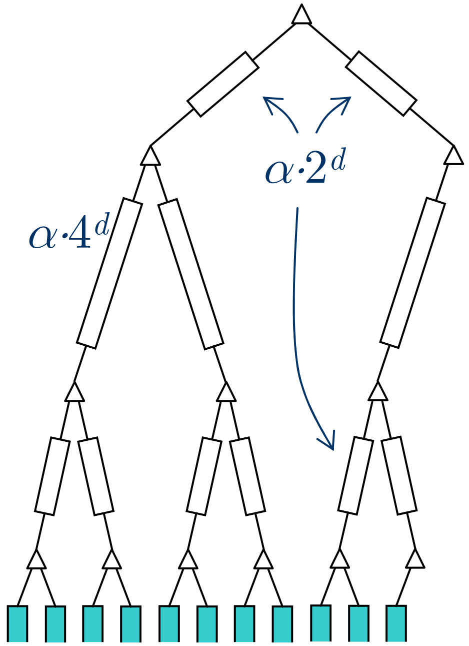
- To analyze the cache complexity of an algorithm that is oblivious to caches, some issues need to be settled:
  - How is an I/O initiated?
  - Where in memory should the block be placed?

Cache-oblivious Algorithms – Ideal Cache

- We analyze in the ideal cache model:
  - Automatic replacement
  - Full associativity
  - Optimal replacement strategy: Underlying replacement policy is the optimal offline algorithm.
  - Two levels of memory
  - Tall cache: $M/B \geq cB^{2/(d-1)}$, for some $c > 0$ and $d > 1$.
- Unrealistic assumptions?

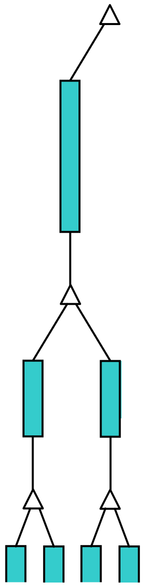
Cache-oblivious Algorithms – Sorting cont’d.

- Funnelsort and LOWSCOSA achieve optimality by merging with *funnels*.
- A funnel is a tree with buffers on the edges. These buffers are *inputs* and *outputs* of the nodes.
- Buffer capacity is determined by following the van Emde Boas recursion; the capacity of the output buffer of a tree with k inputs is αk^d.



Merging – Two-phase funnel with refilling

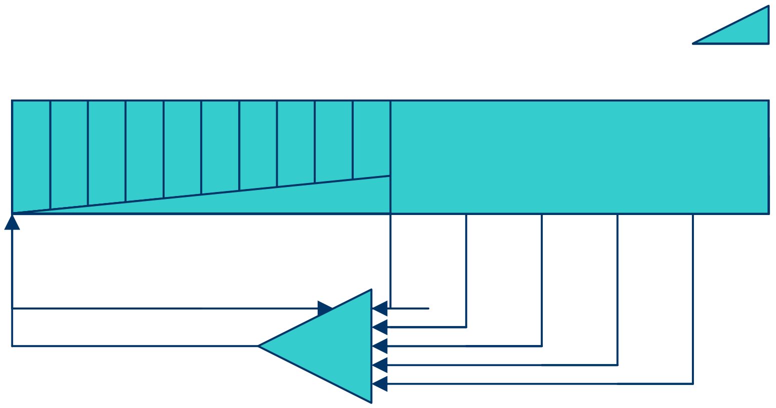
- Elements are merged from the input of a node to the output in a fill() operation.
- In an explicit warm-up phase, fill() is called on all nodes bottom-up. Elements are output from the funnel by then calling fill() on the root.
- When fill() merges at leaf nodes, a custom Refill() function is invoked to signal that elements have been read in from the input of the funnel, so that the space they occupy may be reused.
- fill() merges until either the output is full or one of the inputs is empty. In the latter case, it calls recursively the fill the input. In the first, it is done.

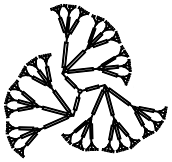


Refill()

LOWSCOSA

- World's first low-order working space cache-oblivious sorting algorithm.
  1. Partition small elements to the back.
  2. Sort recursively (or by using funnelsort).
  3. Attach refiller that moves elements from the front of the array to newly freed space in the input streams.
  4. Sort right half recursively.





Engineering Cache-oblivious Sorting Algorithms

Master's thesis by

Kristoffer Vinther

Algorithm Engineering

It’s all about speed!

- ...and
  - Correctness
  - Robustness
  - Flexibility
  - Portability

Algorithm Engineering – What is speed?

- Theoretician: Asymptotic worst-case running time.
- Algorithm engineer:
  - Good asymptotic performance
  - Low proportionality constants
  - Fast running times on real-world data
  - Robust performance across variety of data
  - Robust performance across variety of platforms

Algorithm Engineering – How to gain speed?

- Optimize low-level data structures.
- Optimize low-level algorithmic details.
- Optimize low-level coding.
- Optimize memory consumption.
- Maximize locality of reference.

A good understanding of the algorithms is extremely important.

Algorithm Engineering
– Pencil & paper vs. implementation

- Moret defines algorithm engineering as ”Transforming ”paper-and-pencil” algorithms into efficient and useful implementations.”

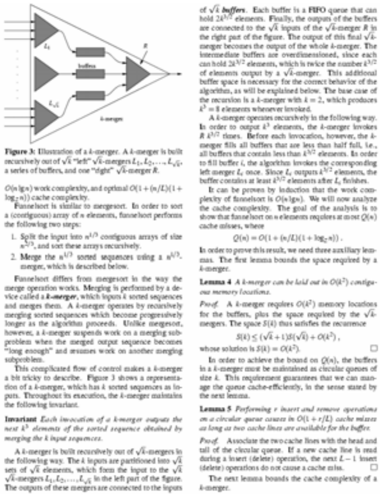


Figure 3: Illustration of a k-merger. A k-merger is built recursively out of $\sqrt{k}$ "left" $\sqrt{k}$-mergers $L_1, L_2, \ldots, L_{\sqrt{k}}$, a series of buffers, and one "right" $\sqrt{k}$-merger R.

$O(n \lg n)$ work complexity, and optimal $O(1+(n/L)(1+\log_Z n))$ cache complexity.

Funnelsort is similar to mergesort. In order to sort a (contiguous) array of n elements, funnelsort performs the following two steps:

1. Split the input into $n^{1/3}$ contiguous arrays of size $n^{2/3}$, and sort these arrays recursively.
2. Merge the $n^{1/3}$ sorted sequences using a $n^{1/3}$-merger, which is described below.

Funnelsort differs from mergesort in the way the merge operation works. Merging is performed by a device called a ***k*-merger**, which inputs k sorted sequences and merges them. A k-merger operates by recursively merging sorted sequences which become progressively longer as the algorithm proceeds. Unlike mergesort, however, a k-merger suspends work on a merging subproblem when the merged output sequence becomes "long enough" and resumes work on another merging subproblem.

This complicated flow of control makes a k-merger a bit tricky to describe. Figure 3 shows a representation of a k-merger, which has k sorted sequences as inputs. Throughout its execution, the k-merger maintains the following invariant.

Invariant *Each invocation of a k-merger outputs the next k^3 elements of the sorted sequence obtained by merging the k input sequences.*

A k-merger is built recursively out of $\sqrt{k}$-mergers in the following way. The k inputs are partitioned into $\sqrt{k}$ sets of $\sqrt{k}$ elements, which form the input to the $\sqrt{k}$ $\sqrt{k}$-mergers $L_1, L_2, \ldots, L_{\sqrt{k}}$ in the left part of the figure. The outputs of these mergers are connected to the inputs of $\sqrt{k}$ ***buffers***. Each buffer is a FIFO queue that can hold $2k^{3/2}$ elements. Finally, the outputs of the buffers are connected to the $\sqrt{k}$ inputs of the $\sqrt{k}$-merger R in the right part of the figure. The output of this final $\sqrt{k}$-merger becomes the output of the whole k-merger. The intermediate buffers are overdimensioned, since each can hold $2k^{3/2}$ elements, which is twice the number $k^{3/2}$ of elements output by a $\sqrt{k}$-merger. This additional buffer space is necessary for the correct behavior of the algorithm, as will be explained below. The base case of the recursion is a k-merger with $k = 2$, which produces $k^3 = 8$ elements whenever invoked.

A k-merger operates recursively in the following way. In order to output k^3 elements, the k-merger invokes R $k^{3/2}$ times. Before each invocation, however, the k-merger fills all buffers that are less than half full, i.e., all buffers that contain less than $k^{3/2}$ elements. In order to fill buffer i, the algorithm invokes the corresponding left merger L_i once. Since L_i outputs $k^{3/2}$ elements, the buffer contains at least $k^{3/2}$ elements after L_i finishes.

It can be proven by induction that the work complexity of funnelsort is $O(n \lg n)$. We will now analyze the cache complexity. The goal of the analysis is to show that funnelsort on n elements requires at most $Q(n)$ cache misses, where

$$Q(n) = O(1+(n/L)(1+\log_Z n)) .$$

In order to prove this result, we need three auxiliary lemmas. The first lemma bounds the space required by a k-merger.

Lemma 4 *A k-merger can be laid out in $O(k^2)$ contiguous memory locations.*

Proof. A k-merger requires $O(k^2)$ memory locations for the buffers, plus the space required by the $\sqrt{k}$-mergers. The space $S(k)$ thus satisfies the recurrence

$$S(k) \le (\sqrt{k}+1)S(\sqrt{k}) + O(k^2) ,$$

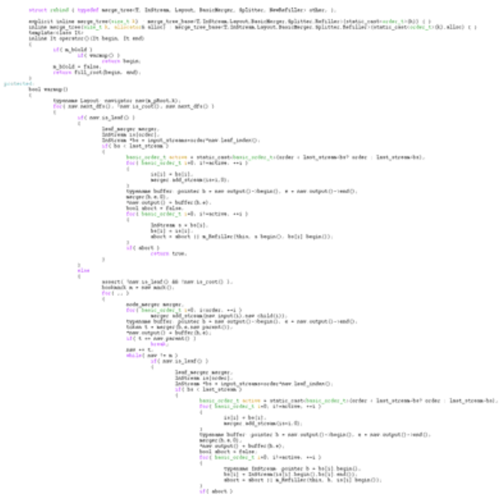
whose solution is $S(k) = O(k^2)$. □

In order to achieve the bound on $Q(n)$, the buffers in a k-merger must be maintained as circular queues of size k. This requirement guarantees that we can manage the queue cache-efficiently, in the sense stated by the next lemma.

Lemma 5 *Performing r insert and remove operations on a circular queue causes in $O(1+r/L)$ cache misses as long as two cache lines are available for the buffer.*

Proof. Associate the two cache lines with the head and tail of the circular queue. If a new cache line is read during a insert (delete) operation, the next $L-1$ insert (delete) operations do not cause a cache miss. □

The next lemma bounds the cache complexity of a k-merger.



```
struct rebind { typedef merge_tree<T, InStream, Layout, BasicMerger, Splitter, NewRefiller> other; };

explicit inline merge_tree(size_t k) : merge_tree_base<T,InStream,Layout,BasicMerger,Splitter,Refiller>(static_cast<order_t>(k)) { }
inline merge_tree(size_t k, allocator& alloc) : merge_tree_base<T,InStream,Layout,BasicMerger,Splitter,Refiller>(static_cast<order_t>(k),alloc) { }
template<class It>
inline It operator()(It begin, It end)
{
    if( m_bHold )
        if( warmup() )
            return begin;
    m_bHold = false;
    return fill_root(begin, end);
}
protected:
    bool warmup()
    {
        typename Layout::navigator nav(m_pRoot,k);
        for( nav.next_dfs(); !nav.is_root(); nav.next_dfs() )
        {
            if( nav.is_leaf() )
            {
                leaf_merger merger;
                InStream is[order];
                InStream *bs = input_streams+order*nav.leaf_index();
                if( bs < last_stream )
                {
                    basic_order_t active = static_cast<basic_order_t>(order < last_stream-bs? order : last_stream-bs);
                    for( basic_order_t i=0; i!=active; ++i )
                    {
                        is[i] = bs[i];
                        merger.add_stream(is+i,0);
                    }
                    typename buffer::pointer b = nav.output()->begin(), e = nav.output()->end();
                    merger(b,e,0);
                    *nav.output() = buffer(b,e);
                    bool abort = false;
                    for( basic_order_t i=0; i!=active; ++i )
                    {
                        InStream s = bs[i];
                        bs[i] = is[i];
                        abort = abort || m_Refiller(this, s.begin(), bs[i].begin());
                    }
                    if( abort )
                        return true;
                }
            }
            else
            {
                assert( !nav.is_leaf() && !nav.is_root() );
                bookmark m = nav.mark();
                for( ;; )
                {
                    node_merger merger;
                    for( basic_order_t i=0; i<order; ++i )
                        merger.add_stream(nav.input(i),nav.child(i));
                    typename buffer::pointer b = nav.output()->begin(), e = nav.output()->end();
                    token t = merger(b,e,nav.parent());
                    *nav.output() = buffer(b,e);
                    if( t == nav.parent() )
                        break;
                    nav += t;
                    while( nav != m )
                        if( nav.is_leaf() )
                        {
                            leaf_merger merger;
                            InStream is[order];
                            InStream *bs = input_streams+order*nav.leaf_index();
                            if( bs < last_stream )
                            {
                                basic_order_t active = static_cast<basic_order_t>(order < last_stream-bs? order : last_stream-bs);
                                for( basic_order_t i=0; i!=active; ++i )
                                {
                                    is[i] = bs[i];
                                    merger.add_stream(is+i,0);
                                }
                                typename buffer::pointer b = nav.output()->begin(), e = nav.output()->end();
                                merger(b,e,0);
                                *nav.output() = buffer(b,e);
                                bool abort = false;
                                for( basic_order_t i=0; i!=active; ++i )
                                {
                                    typename InStream::pointer b = bs[i].begin();
                                    bs[i] = InStream(is[i].begin(),bs[i].end());
                                    abort = abort || m_Refiller(this, b, is[i].begin());
                                }
                                if( abort )
```

- Filling in the details.

Experimental Methodology

Methodology – Algorithmic details

- How should the funnel be laid out in memory?
- How do we locate nodes and buffers?
- How should we implement merge functionality?
- What is a good value for z and how do we merge multiple streams efficiently?
- How do we reduce the overhead of the sorting algorithm?
- How do we sort at the base of the recursion?
- What are good values for α and d?
- How do we handle the output of the funnel?
- How do we best manage memory during sorting?
- ...

Methodology – Algorithmic details cont’d.

- Inspired by knowledge of the memory hierarchy and modern processor technology, we develop several solutions to each of these questions.
- All solutions are implemented and benchmarked to locate the best performing combination of approaches.
- It turns out, the simpler the faster (except perhaps memory management).
- Increasing α and d is a cheap way of decreasing the overhead of the funnel.

Methodology – What answers do we seek?

- Are the assumptions of the ideal cache model too unrealistic, i.e. are the algorithms *only* optimal/competitive under ideal conditions?
- Will the better utilization of caches improve running time of our sorting algorithms?
- Will the better utilization of virtual memory improve running time of our sorting algorithms?
- Can our algorithms compete with classic instruction count optimized RAM-based sorting algorithms and memory-tuned cache-aware EM-based sorting algorithms?

Methodology – Platforms

- To avoid ”accidental optimization,” we benchmark on several different architectures:
  - MIPS R10k: Classic RISC; short pipeline, large L2 cache, low clock rate, software TLB miss handling. 64-bit.
  - Pentium 3: Classic CISC; twice as deep a pipeline as MIPS, good branch prediction, many execution units.
  - Pentium 4: Extremely deep pipeline, compensated by very good branch prediction. Very high clock rates.
- Several different operating systems supported: IRIX (64-bit), Linux (32-bit), Windows (32-bit). Benchmarks run on IRIX and Linux.
- Tested with several different compilers: GCC, MSVC, MIPSPRO, ICC.

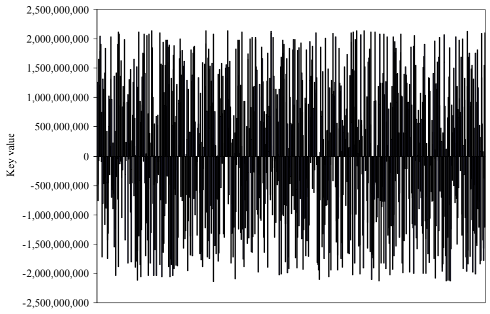
Methodology – Data types

- To demonstrate robustness, we benchmark several different data types:
  - Key/pointer pairs: class { long key; void *p; }
  - Simple keys: long.
  - Records: class { char record[100]; }
    - Inspired by the official sorting benchmark, Datamation Benchmark.
    - Order determined by strncmp().

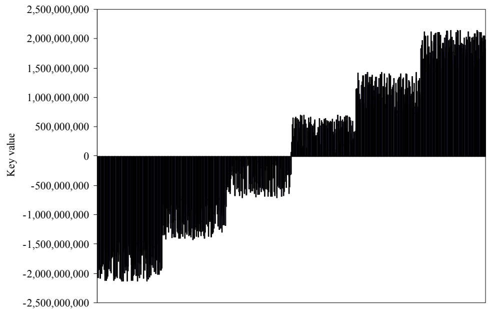
Methodology – Input data

- To demonstrate robustness, we benchmark several different input distributions:
  - Uniformly distributed.
  - Almost sorted.
  - Few distinct elements.

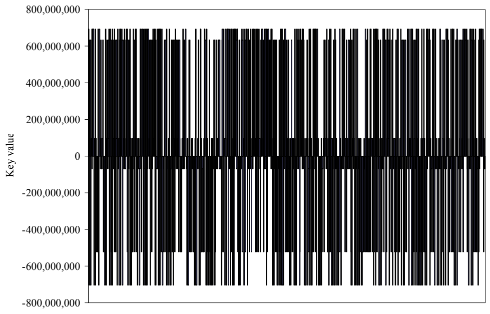
Uniform key distribution



Almost sorted



Few disitinct keys



Methodology – What to measure

- Primarily wall clock time.
- CPU time is no good, since it does not take into account the time spent waiting for page faults to be serviced.
- L2 cache misses.
- TLB misses.
- Page faults.

Methodology – Validity

- For time considerations, we run benchmarks only once.
- Benchmarks are run on such massive datasets that they each take several minuets, even several hours.
- Periodic external disturbances affect all algorithms, are always present, and cannot be eliminated by e.g. averaging.

Methodology – Competitors

- To answer the question of whether our cache-oblivious algorithms can compete with RAM-based and memory-tuned cache-aware sorting algorithms, we compare them with
  - *Introsort*, developed by SGI as part of STL.
  - *Multiway mergesort*, a part of TPIE, tuned for disk.
  - *Multi-mergesort*, developed by Kubricht *et al.*, tuned for L2 cache.
  - *Tiled mergesort*, developed by Kubricht *et al.*, tuned for L2 cache.

Methodology – The problem

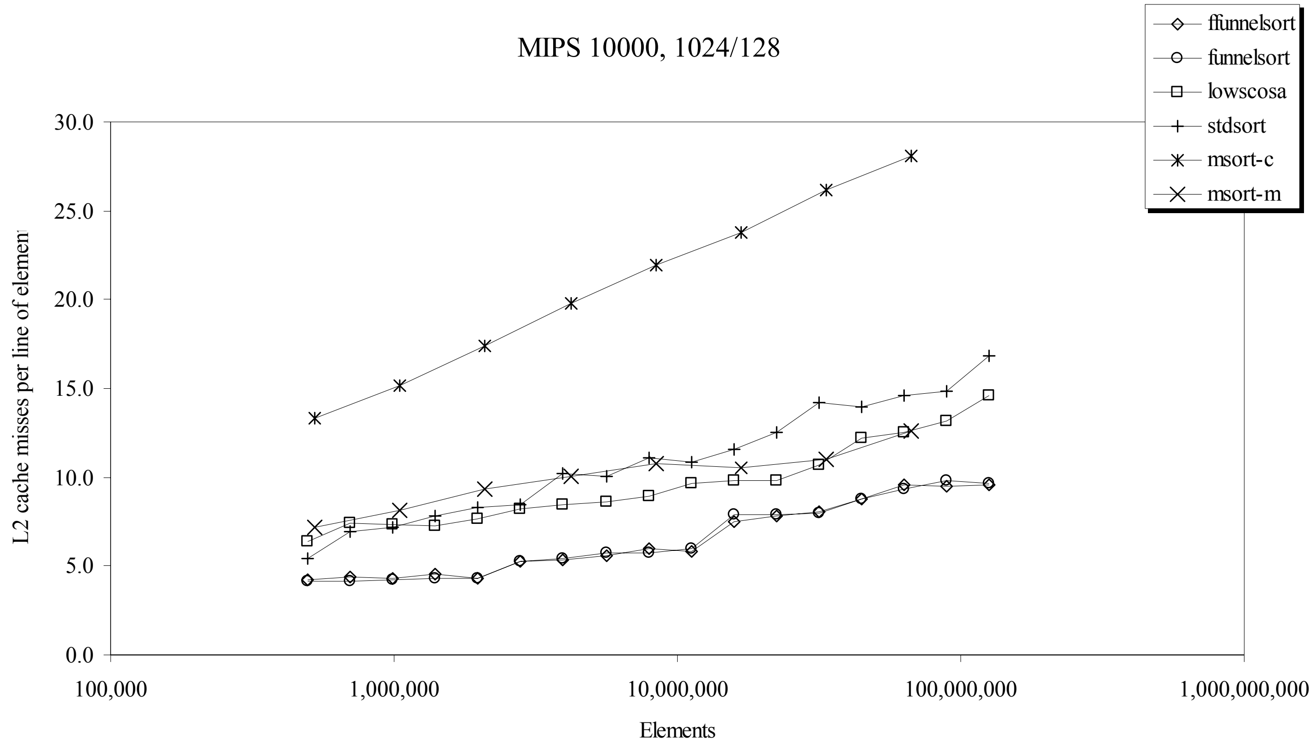
- A file stored on a local disk in a native file system contains a number of contiguous elements.
- The problem is solved when there exist a (possibly different) file with the same elements stored contiguously in non-decreasing order.
- No part of the (original) file is in memory when sorting begins.

Motivation: We don’t want to favor any particular initial approach; we believe that real-life applications of sorting doesn’t.

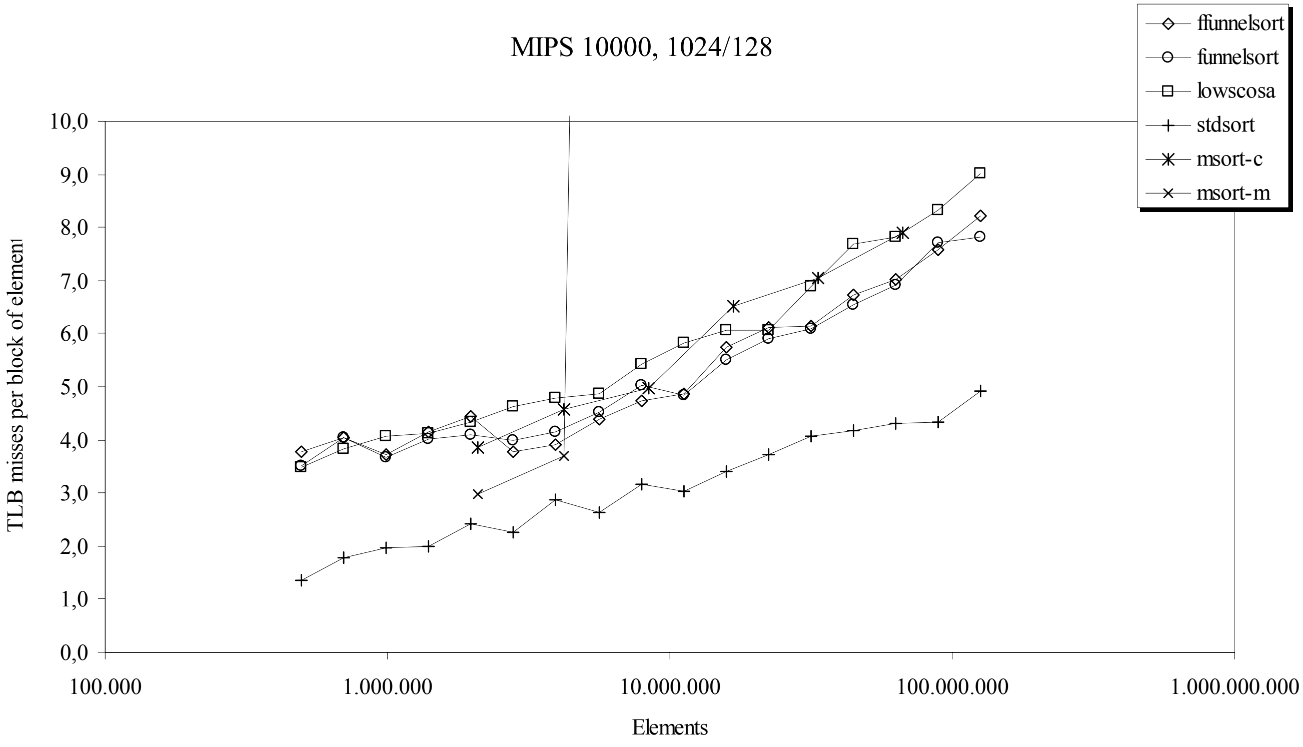
Inspiration: Datamation Benchmark.

Results

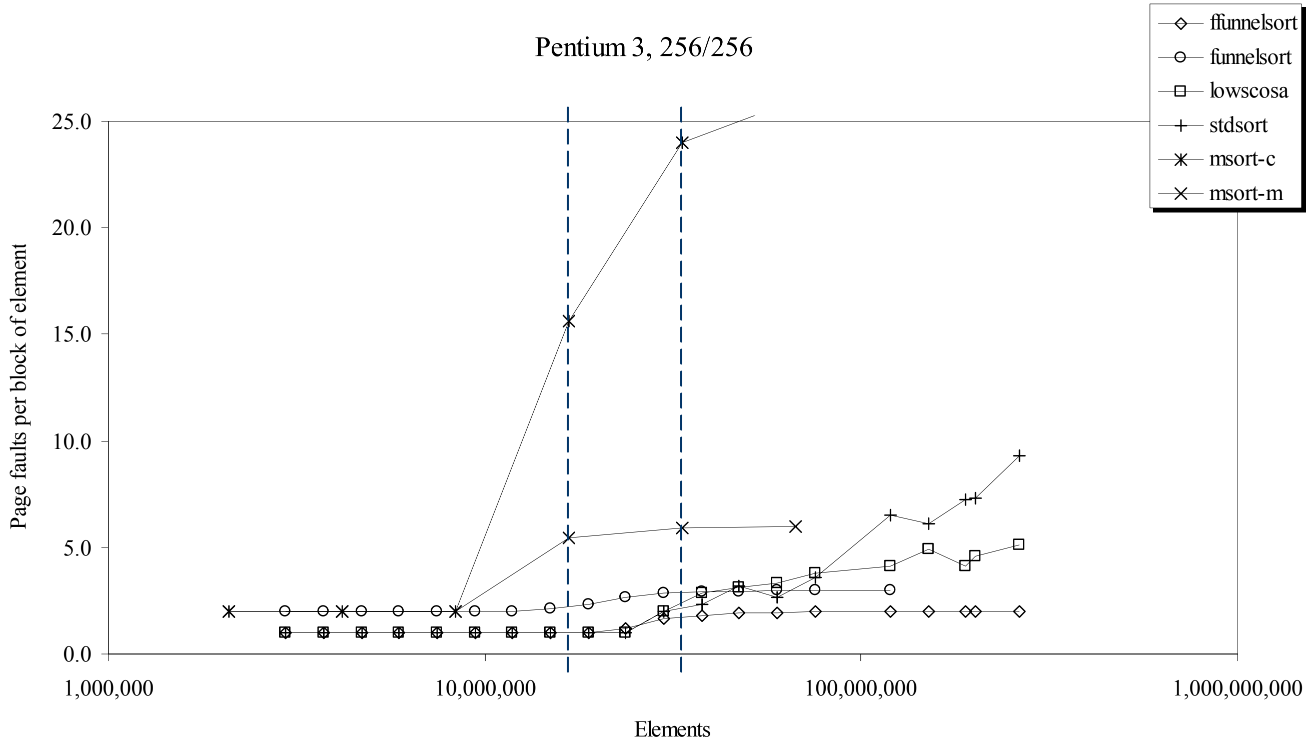
Results
– L2 cache misses



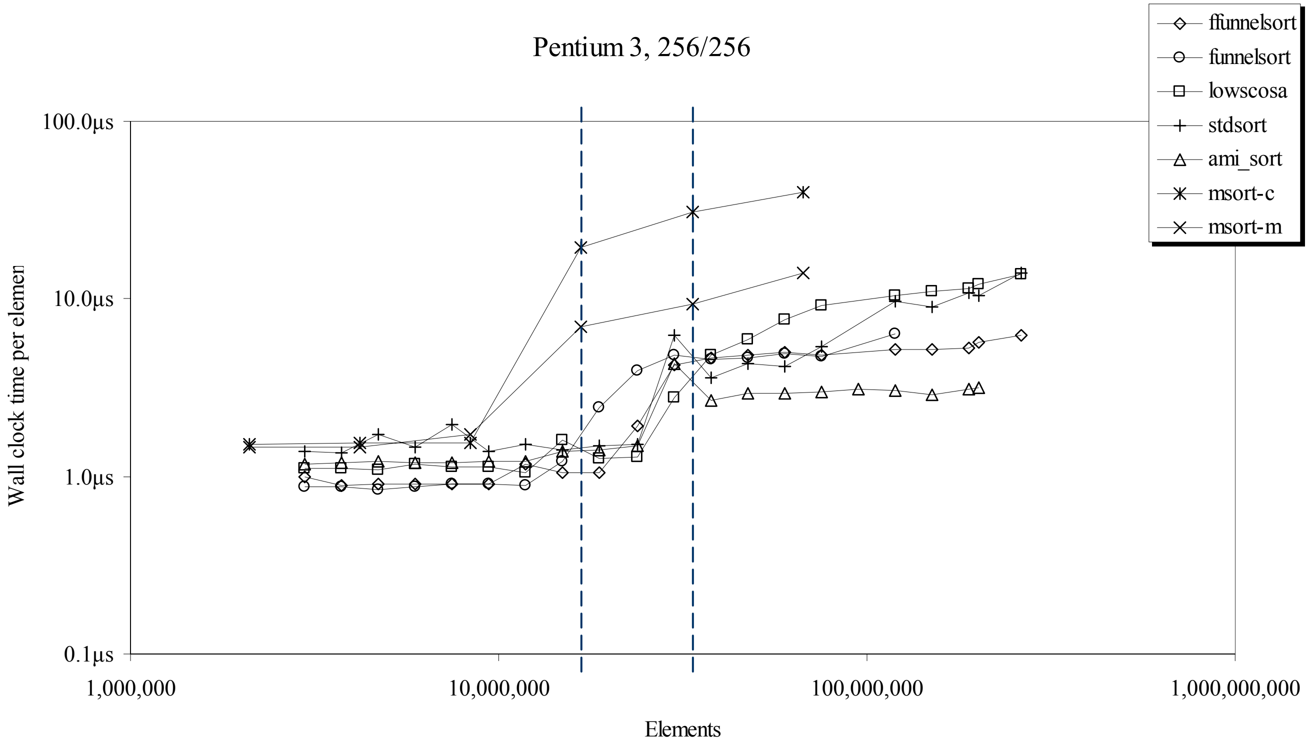
Results – TLB misses



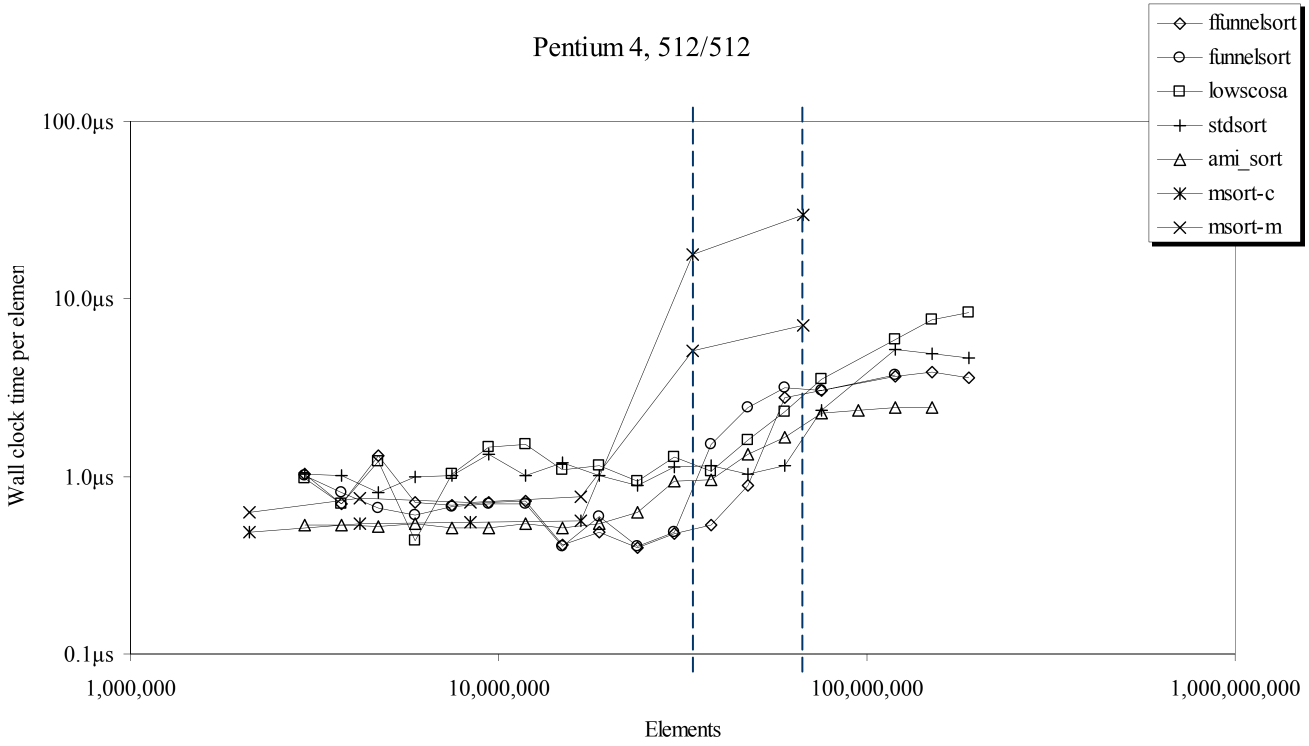
Results – Page faults



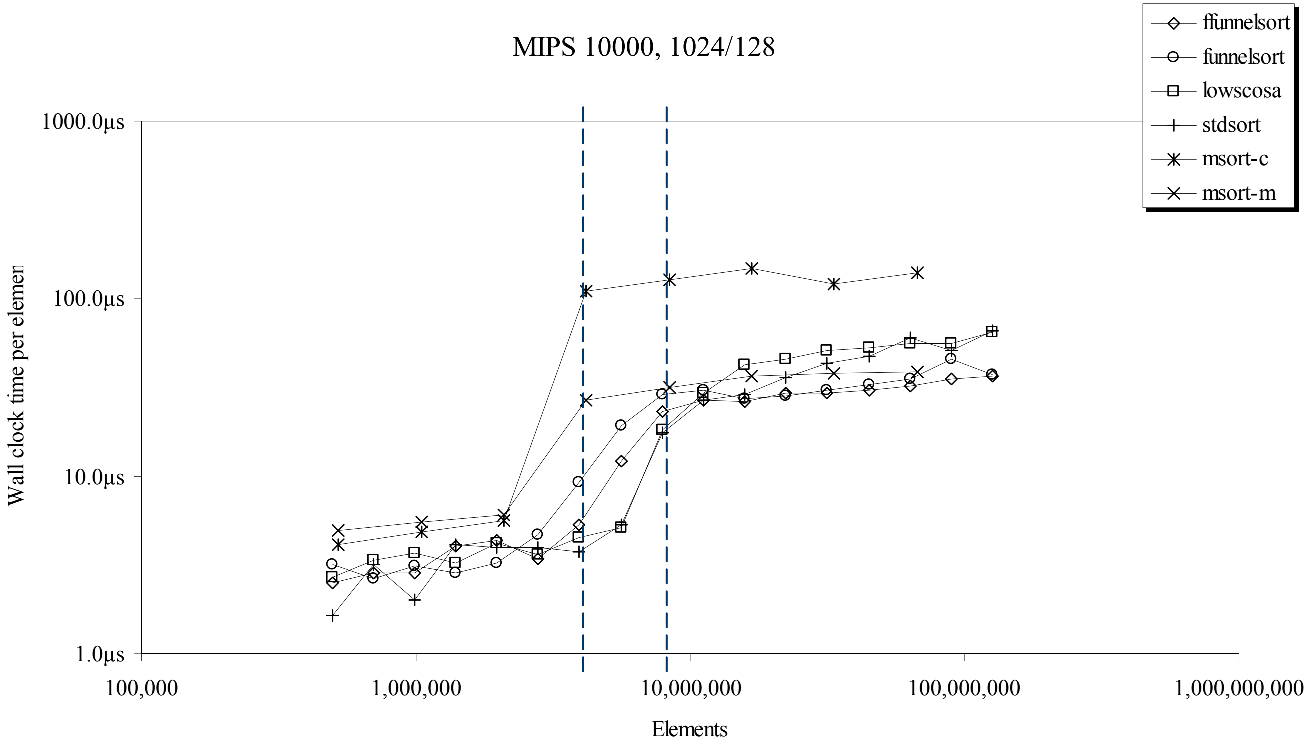
Results
– Wall clock time, Pentium 3



Results
– Wall clock time, Pentium 4



Results
– Wall clock time, MIPS R10000



Conclusion

Conclusion

- Very high performing generic sorting algorithm.
- Unique to our algorithms, performance remains robust
  - across wide range of input sizes.
  - on several different data types.
  - on several different input distributions.
  - across several different processor and operating system architectures.

References

- Bernard M.E. Moret and Henry D. Shapiro. Algorithms and Experiments: The New (and Old) Methodology, Journal of Universal Computer Science, 7:434-446, 2001.
- Alok Aggarwal and Jeffery S. Vitter. The input/output complexity of sorting and related problems. Communications of the ACM, 1988.
- Matreo Frigo, Charles E. Leiserson, Harald Prokop, and Shidhar Ramachandran. Cache-oblivious algorithms. Proceedings of the 40th Annual Symposium on Foundations of Computer Science, New York, 1999.
- David S. Johnson. A Theoretician's Guide to the Experimental Analysis of Algorithms. Proceedings of the 5th and 6th DIMACS Implementation Challenges. Goldwasser, Johnson, and McGeoch (eds), American Mathematical Society, 2001.
- Datamation Benchmark. Sort Benchmark Home Page, hosted by Microsoft. World Wide Web document, http://research.microsoft.com/barc/SortBenchmark/, 2003.
- Duke University. A Transparent Parallel I/O Environment, World Wide Web Document, http://www.cs.duke.edu/TPIE/, 2002.
- Li Xiao, Xiaodong Zhang, and Stefan A. Kubricht. Improving Memory Performance of Sorting Algorithms, ACM Journal of Experimental Algorithms, Vol. 5, 2000.