Engineering a Cache-Oblivious Sorting Algorithm

Gerth Stølting Brodal, Rolf Fagerberg, Kristoffer Vinther

Introduction

Cache

- smaller, faster memory
- stores copies of the data from frequently used main memory locations
- hierarchy of cache levels (registers, level 1 cache, level 2 cache, level 3 cache, main memory, and disk)
- time for accessing a level increases for each new level (most dramatically when going from main memory to disk)

Why make a cache-oblivious sorting algorithm?

- Accessing items in closer cache is more efficient
- Memory access time has major influence on program efficiency

Classic asymptotic analysis of algorithms in the RAM model is unable to capture this

Past work to deal with cache

Other models have been proposed to deal with this

- I/O model introduced by of Aggarwal and Vitter in 1988 (paper from last week) (2 level memory hierarchy)
- A lot of subsequent analysis done in this area

Fundamental finding:

In the I/O model, comparison based sorting takes $\Theta(\mathrm{Sort}_{M,B}(N))$ I/Os, where

$\mathrm{Sort}_{M,B}(N) = \frac{N}{B} \log_{M/B} \frac{N}{B}$ (worst case)

Algorithms are assumed to know the characteristics of the memory hierarchy (not oblivious)

What is a cache-oblivious algorithm?

An algorithm that takes advantage of a processor cache without having the size of the cache (or the length of the cache lines, etc.) as an explicit parameter

The concept of cache-oblivious algorithms was introduced by Frigo et al.

What is a cache-oblivious algorithm?

Designates algorithms formulated in the RAM model, but analyzed in the I/O model for arbitrary block size B and memory size M

I/Os are assumed to be performed automatically by an offline optimal cache replacement strategy

- analysis holds for any block and memory size
- holds for *all* levels of a multilevel memory hierarchy
- characteristics of the memory hierarchy do not need to be known, and do not need to be hardwired into the algorithm for the analysis to hold

By optimizing an algorithm to one unknown level of the memory hierarchy, it is optimized to each level automatically

Previous work on cache-oblivious algorithms

Frigo et al. introduced the concept of cache obliviousness, and presented optimal cache-oblivious algorithms for matrix transposition, FFT, and sorting

Since then many groups have made cache-oblivious algorithms for:

- Tree search
- Computational geometry
- Scanning dynamic sets
- Layout of static trees
- Partial persistence
- Priority queues
- Graph algorithms

Assumptions for cache-oblivious algorithms

Some of these results, in particular those involving sorting and algorithms to which sorting reduces are proved under the assumption **$M \geq B^2$ (tall cache assumption)** (cache much larger than block size)

In particular, this applies to the **Funnelsort** algorithm of Frigo et al.

A variant called lazy funnelsort improved upon this slightly **$M \geq B^{1+\varepsilon}$** with some added assumptions

This paper’s angle

In contrast to the abundance of theoretical results described, empirical evaluations of the merits of cache-obliviousness are more scarce

Current ‘tentative’ results conclude that the efficiency of cache oblivious algorithms lies between that of classic RAM algorithms and that of algorithms exploiting knowledge about the specific memory hierarchy present (cache-aware algorithms)

Here, the authors investigate the practical value of cache-oblivious methods in the area of sorting

Approach

Authors focus on the Lazy Funnelsort algorithm (they believe it has the biggest potential for an efficient implementation among the current proposals for I/O-optimal cache-oblivious sorting algorithms)

- explore a number of implementation issues and parameter choices for the cache-oblivious sorting algorithm Lazy Funnelsort
- settle the best choices through experiments
- compare the final algorithm with tuned versions of Quicksort

Lazy Funnelsort

Binary mergers

- takes as input two sorted streams of elements and delivers as output the sorted stream formed by merging of these
- a merge step moves an element from the head of one of the input streams to the tail of the output stream
- heads of the input streams and the tail of the output stream reside in **buffers** holding a limited number of elements
    - buffer is simply an array of elements, plus fields storing the capacity of the buffer and pointers to the first and last elements in the buffer

Lazy Funnelsort

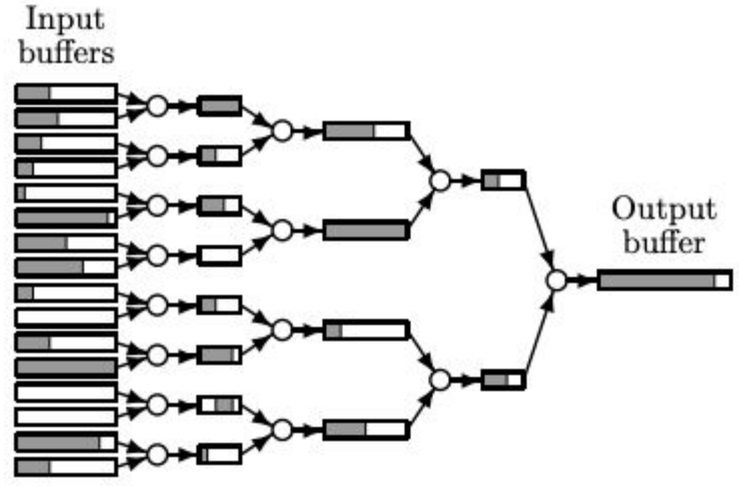
Binary merge trees

- Binary mergers can be combined to binary merge trees by letting the output buffer of one merger be an input buffer of another
- binary trees with mergers at the nodes and buffers at the edges. The leaves of the tree contain the streams to be merged
- An *invocation* of a merger is a recursive procedure which performs merge steps until its output buffer is full or both input streams are exhausted

Lazy Funnelsort

K-merger

For k = a power of two, a k-merger is a perfect binary tree of k − 1 binary mergers with appropriate sized buffers on the edges, k input streams, and an output buffer at the root of size k d , for a parameter d > 1



Lazy Funnelsort

K-merger

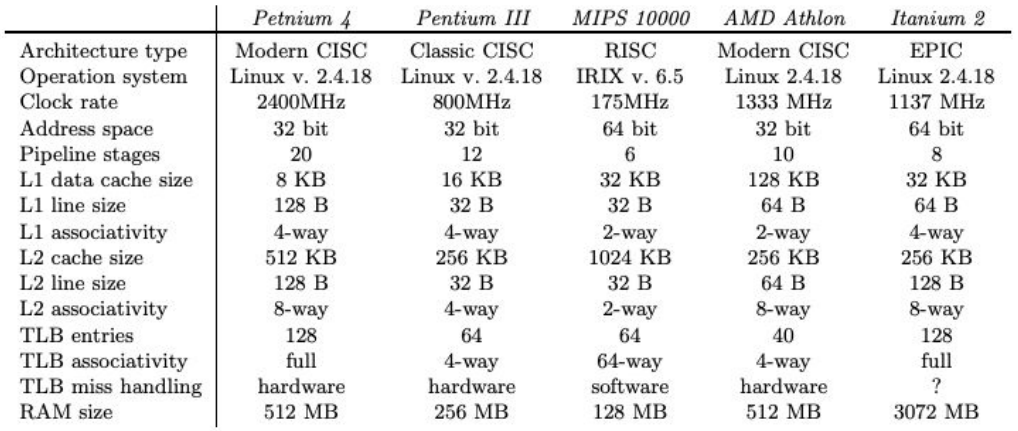
Sizes of the buffers are defined recursively:

- Let the top tree be the subtree consisting of all nodes of depth at most $\lceil i/2 \rceil$, and let the subtrees rooted by nodes at depth $\lceil i/2 \rceil + 1$ be the bottom trees
- The edges between nodes at depth $\lceil i/2 \rceil$ and depth $\lceil i/2 \rceil + 1$ have associated buffers of size $\alpha \lceil d\ 3/2 \rceil$, where α is a positive parameter, and the sizes of the remaining buffers are defined by recursion on the top tree and the bottom trees

Methodology

Goal: first to develop a good implementation of Funnelsort by finding good choices for design options and parameter values through empirical investigation, and then to compare its efficiency to that of Quicksort

Test on five different architectures



	Petnium 4	*Pentium III*	*MIPS 10000*	*AMD Athlon*	*Itanium 2*
Architecture type	Modern CISC	Classic CISC	RISC	Modern CISC	EPIC
Operation system	Linux v. 2.4.18	Linux v. 2.4.18	IRIX v. 6.5	Linux 2.4.18	Linux 2.4.18
Clock rate	2400MHz	800MHz	175MHz	1333 MHz	1137 MHz
Address space	32 bit	32 bit	64 bit	32 bit	64 bit
Pipeline stages	20	12	6	10	8
L1 data cache size	8 KB	16 KB	32 KB	128 KB	32 KB
L1 line size	128 B	32 B	32 B	64 B	64 B
L1 associativity	4-way	4-way	2-way	2-way	4-way
L2 cache size	512 KB	256 KB	1024 KB	256 KB	256 KB
L2 line size	128 B	32 B	32 B	64 B	128 B
L2 associativity	8-way	4-way	2-way	8-way	8-way
TLB entries	128	64	64	40	128
TLB associativity	full	4-way	64-way	4-way	full
TLB miss handling	hardware	hardware	software	hardware	?
RAM size	512 MB	256 MB	128 MB	512 MB	3072 MB

Table 1: The specifications of the machines used in this paper.

Methodology

Try three element types

Integers: commonly used in experimental papers, but not particularly realistic, as keys normally have associated information

Records containing one integer and one pointer: models sorting small records directly, as well as key-sorting of large records

Records of 100 bytes: models sorting medium sized records directly, and is the data type used in the Datamation Benchmark originating from the database community

Methodology

Variations

- mainly consider uniformly distributed keys, but also try skewed inputs such as almost sorted data or data with few distinct key values

Performance

- Measured wall clock time

Optimizing Lazy Funnelsort

k-Merger Structure

While no particular layout is needed for the theoretical analysis of Lazy Funnelsort to hold, *some* layout has to be chosen, and the choice could affect the running time

- consider BFS, DFS, and vEB **layout**
- consider having a merger node **stored** along with its output buffer, or storing nodes and buffers separately (each part having the same layout)
- consider both pointer based and implicit **navigation** (shown to be possible on these trees)
- try two **coding styles** for the invocation of a merger, namely the straight-forward recursive implementation, and an iterative version
- try make own **allocation** function, which starts by acquiring enough memory to hold the entire merger, or using default allocator

k-Merger Structure

- test combinations of the choices described
- One experiment consists of merging k streams of k^2 elements in a k-merger with $z = 2$, $\alpha = 1$, and $d = 2$
- For each choice, for values of k in [15: 270] measure the time for $\lceil 20{,}000{,}000/k^3 \rceil$ such mergings

k-Merger Structure

- best combination on all architectures is recursive invocation of a pointer based vEB layout with nodes and buffers separate, allocated by the standard allocator
- time used for the slowest combination is up to 65% more
- difference is biggest on the Pentium 4 architecture
- largest gain occurs by choosing the recursive invocation over the iterative
- vEB layout ensures around 10% reduction in time
    - spatial locality of the layout is not entirely without influence in practice, despite its lack of influence on the asymptotic analysis

Tuning the Basic Mergers

The **inner loop** in the Lazy Funnelsort algorithm is the code performing the merge step in the nodes of the k-merger

Explore ideas for efficient implementation of this code

Idea: compute the minimum of the number of elements left in either input buffer and the space left in the output buffer. Merging can proceed for at least that many steps without checking the state of the buffers, thereby eliminating one branch from the core merging loop.

Tuning the Basic Mergers

Not beneficial (rather, the minimum computation will constitute an overhead) in situations where one input buffer stays small for many merge steps

-> also implement the optimal merging algorithm of Hwang and Lin, which has higher overhead, but is an asymptotical improvement when merging sorted lists of very different sizes

also try a hybrid solution which invokes it only when the contents of the input buffers are skewed in size

Tuning the Basic Mergers

Results:

- Hwang-Lin algorithm has, as expected, a large overhead (3x for non hybrid version)
- heuristic calculating minimum sizes is not competitive, being between 15% and 45% slower than the fastest
- Several hybrids fare better, but the straightforward solution is consistently the winner in all experiments

Interpretation: hand-coding just constitutes overhead

Degree of Basic Mergers

No need for the k-merger to be a binary tree

If we for instance base it on four-way basic mergers, we effectively remove every other level of the tree. This means less element movement and less tree navigation.

reduction in data movement seems promising—part of Quicksorts speed can be attributed to the fact that for random input, only about every other element is moved on each level in the recursion, whereas e.g. binary Mergesort moves all elements at each level.

price to pay is more CPU steps per merge step, and code complication due to the increase in number of input buffers that can be exhausted

Degree of Basic Mergers

Results:

- As degree z goes from 2 to 9, the time first decreases, and then increases again, with minimum attained around 4 or 5
- maximum is 40– 65% slower than the fastest

Since the number of levels for elements to move through evolves as 1/ log(z), while the number of comparisons for each level evolves as z, a likely explanation is that there is an initial positive effect due to decrease in element movements, which soon is overtaken by increase in instruction count per level

findings are rather consistent across the three architectures

Merger Caching

- In the outer recursion of Funnelsort, the same size k-merger is used for all invocations on the same level of the recursion
- A natural optimization would be to precompute these sizes and construct the needed k-mergers once for each size
- These mergers are then reset each time they are used

On all architectures, merger caching gave a 3–5% speed-up

Base Sorting Algorithm

Test different sorting algorithms for base case (when you can’t use tree structure anymore)

Tried Insertionsort, Selectionsort, Heapsort, Shellsort, and Quicksort

Insertionsort is the best

Parameters α and d

α = factor in buffer size expression

d = main parameter defining the progression of the recursion, in the outer recursion of Funnelsort, as well as in the buffer sizes in the k-merger

The sizes of the buffers are defined recursively: Let the top tree be the subtree consisting of all nodes of depth at most $\lceil i/2 \rceil$, and let the subtrees rooted by nodes at depth $\lceil i/2 \rceil + 1$ be the bottom trees. The edges between nodes at depth $\lceil i/2 \rceil$ and depth $\lceil i/2 \rceil + 1$ have associated buffers of size $\alpha \lceil d^{3/2} \rceil$, where α is a positive parameter, and the sizes of the remaining buffers are defined by recursion on the top tree and the bottom trees

Parameters α and d

Results:

- marked rise in running time when α drops below 10, increasing to a factor of four for α = 1
- effect is particularly strong for d = 1.5
- Smaller α and d give smaller sizes of buffers, and the most likely explanation seems to be that the cost of navigating to and invoking a basic merger is amortized over fewer merge steps when the buffers are smaller
- different values of d appear to behave quite similarly

A sensible choice appears to be α around 16, and d around 2.5

Evaluating Lazy Funnelsort

Evaluating Lazy Funnelsort

Competitors:

- Use quicksort to compare implementations of lazy funnelsort
- Also compare with recent implementations of cache-aware sorting algorithms aiming for efficiency in either internal memory or external memory by tunings based on knowledge of the memory hierarchy

Evaluating Lazy Funnelsort

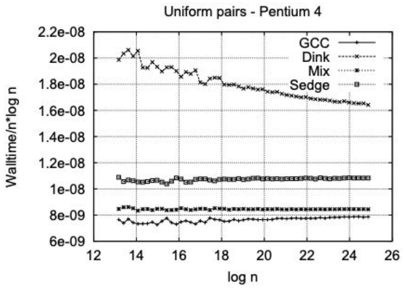
Experiments

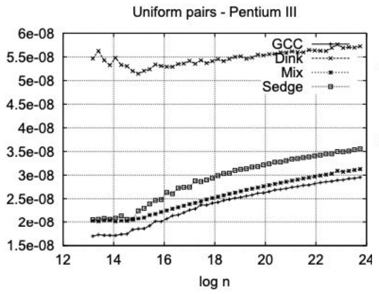
- test the algorithms described above on inputs of sizes in the entire **RAM** range, as well as on inputs residing on **disk**
- All experiments are performed on machines with no other users
- influence of background processes is minimized by running each experiment in internal memory 21 times, and reporting the median
- In external memory experiments are rather time consuming, and we run each experiment only once, believing that background processes will have less impact on these

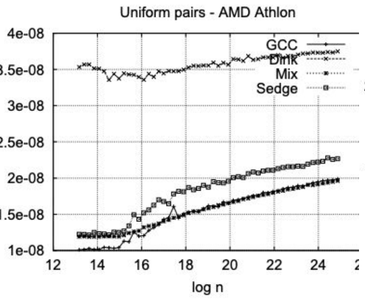
Evaluating Lazy Funnelsort

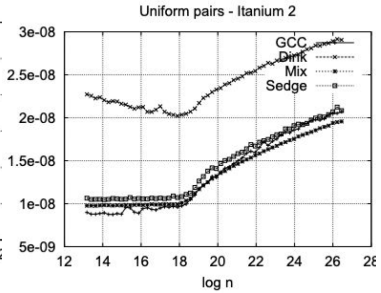
Results:

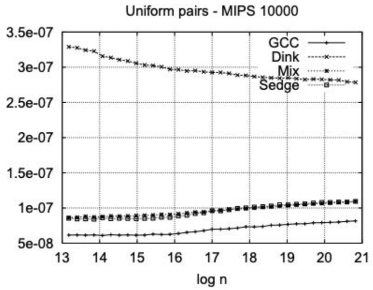
- comparison of Quicksort implementations showed that three contestants ran pretty close, with the GCC implementation as the overall fastest
  - GCC uses a compact two-way partitioning scheme, and simplicity of code here seems to pay off
- closely followed by our own implementation (denoted Mix), based on the tuned three-way partitioning of Bentley and McIlroy











Evaluating Lazy Funnelsort

Results:

- Funnelsort algorithm with four-way basic mergers are consistently better than the one with binary basic mergers, except on the MIPS architecture, which has a very slow CPU
- reduced number of element movements really do outweigh the increased merger complexity, except when CPU cycles are costly compared to memory accesses

Evaluating Lazy Funnelsort

Results:

- For the smallest input sizes, the best Funnelsort loses to GCC Quicksort (by 10-40%), but on three architectures gains as n grows, ending up winning (by the approximately the same ratio) for the largest instances in RAM
    - Except on MIPS 10000 (slow CPU), PC800 bus (has a large cache line size (reducing effects of cache latency when scanning data in cache))
- On these two architectures, CPU cycles, not cache effects, are dominating the running time for sorting
- on architectures where this is not the case, the theoretically better cache performance of Funnelsort actually shows through in practice, at least for a tuned implementation of the algorithm

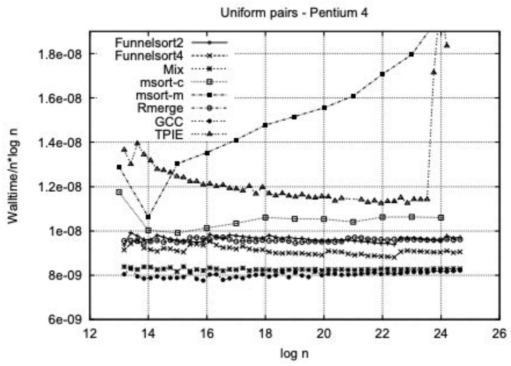
Evaluating Lazy Funnelsort

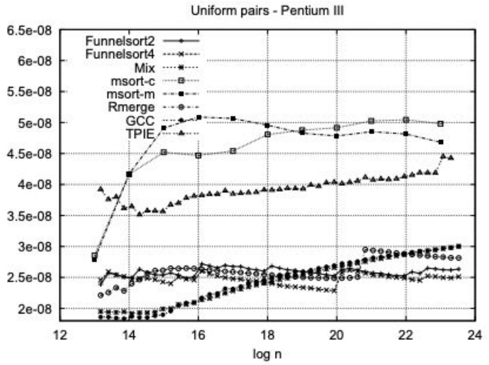
Results:

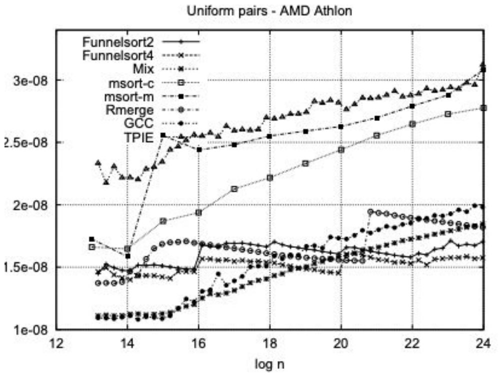
- R-merge algorithm competes well
- TPIEs algorithm is not competitive in RAM (TPIE is a library for external memory computations, and includes highly optimized routines for e.g. scanning and sorting)

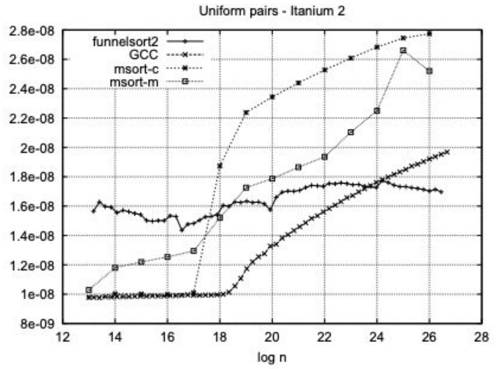
Evaluating Lazy Funnelsort

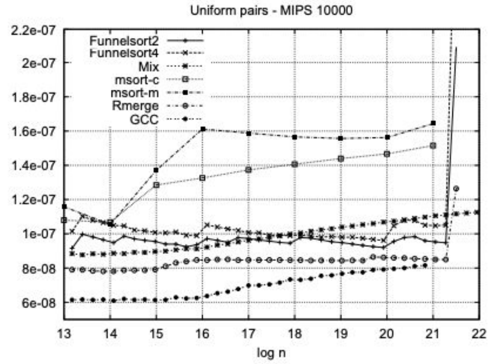
Results:











Evaluating Lazy Funnelsort

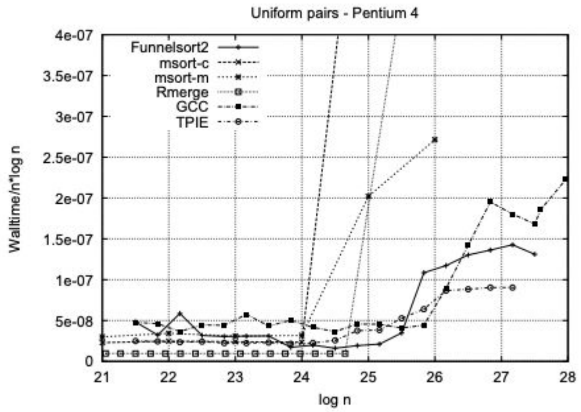
Results:

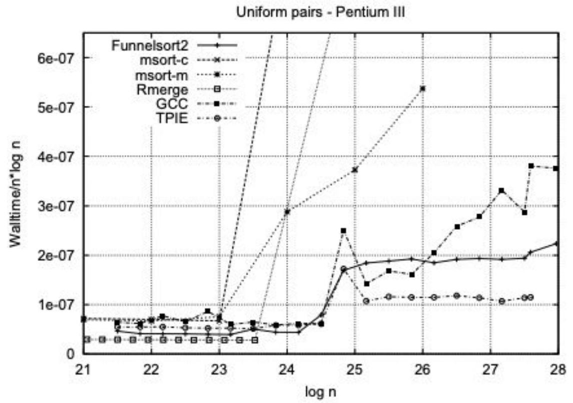
Disk experiments

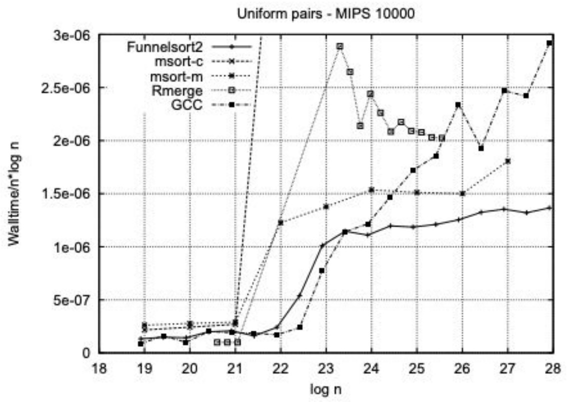
- TPIE is the clear winner (optimized for external memory)
  - its use of double-buffering (something which seems hard to transfer to a cache-oblivious setting) gives it an unbeatable advantage
- Funnelsort comes in as a second, and outperforms GCC quite clearly
  - gain over GCC seems to grow as n grows larger, which is in good correspondence with the difference in the base of logarithms in the I/O complexity of these algorithms
- algorithms tuned to cache perform notably badly on disk

Evaluating Lazy Funnelsort

Results:







Conclusion

Summary

- Developed a tuned implementation of Lazy Funnelsort
- Compared empirically with efficient implementations of other comparison based sorting algorithms
- implementation is competitive in RAM as well as on disk, in particular in situations where sorting is not CPU bound
- Funnelsort was almost always among the two fastest algorithms, and clearly the one adapting most gracefully to changes of level in the memory hierarchy

For sorting, the overhead involved in being cache-oblivious can be small enough for the nice theoretical properties to actually transfer into practical advantages

Assessment

Strengths

- Extensive testing on multiple different types of machines
- Comparison to state of the art existing algorithms

Weaknesses

- Parameter search was sequential (not a grid search) -> could miss some optimal combinations
- Implementation universally better than existing ones

Future work

- Sequential vs parallel cache-oblivious sorting algorithms (eg Panda and Sajith 2022)
- Is cache-oblivious overhead also advantageous for other algorithms besides sorting in practice?