Engineering a Cache-Oblivious Sorting Algorithm (Brodal et al.)

Kliment Serafimov, 6.827, Spring 2022
Motivation

- Sorting is a fundamental problem.
- Cache obliviousness provides guarantees regardless of cache spec.
  - No need to fine tuning.
  - No need for parameter dependence.
Set up

- Working in the RAM model, assuming input sizes that fit in RAM.
- Target runtime (in cache misses): $O\left(\frac{N}{B \log(M)} \right) (N)$
  - $M$ is size of cache
  - $B$ is size of the cache line
  - $N$ is size of input
- Need the (weaker) tall cache assumption: $M > B^{1+c}, c > 0$ (Brodal et al)
  - Pays additional cost factor of $1/c$
  - Standard tall cache assumption: $M > B^2$
Algorithm

Main algorithm:
1. Split \( N \) into \( N^{(1/d)} \) segments of size \( N^{(1-1/d)} \) each
2. Sort each segment recursively (use standard sort for base case)
3. Apply a \((N^{(1/d)})\)-merger on the sorted segments.

K-merger algorithm:

1. Construct a k-merger tree (16-merger tree shown in figure)
   a. With carefully constructed buffer sizes
2. Apply the fill procedure enough times to sort
   a. Each invocation sorts \( k^d \) elements.

Buffer sizes of a k-merger:

1. Buffer size at the middle (depth \( d/2 \)) of the tree are: \( a^d (3/2) \)
2. Recurse on top and bottom trees.
   a. ‘Van Emde Boas’-style recursion

Procedure Fill:\( v \)

\[\text{while } v\text{'s output buffer is not full}\]
\[\text{if left input buffer empty}\]
\[\text{Fill(left child of } v)\]
\[\text{if right input buffer empty}\]
\[\text{Fill(right child of } v)\]
\[\text{perform one merge step}\]
Analysis:

High level idea:

- Sorting is constrained within one subset of the buffers at a time due to the Van Emde Boas-style recursion of setting buffer sizes.
- See board for intuition
Implementation and Experiments

- Machines used for evaluation:
  - Pentium 4, Pentium III, MIPS 10000, AMD Athlon, Itanium 2
- Merger implementation
  - Recursive vs iterative
- Memory navigation:
  - Pointer based, index arithmetic
- Memory layout:
  - BFS, DFS, Van Emde Boas.
  - Lay out nodes and buffers separately, vs together
- Memory allocation:
  - Custom allocator, standard allocator (only used with pointer-based navigation)
- Results of 28 experiments: best choice of parameters:
  - (1) recursive invocation, (2) pointer-based navigation, (3) vEB layout
  - (4) nodes and buffers laid out separately, and (5) allocation by the standard allocator.
More parameters!

- Varying degree of the merger: \( z = \{2..9\} \)
  - Best choice: 4 or 5
- Merger construction caching
  - Gave speedup of 3-5%
- Buffer size scaling parameter \( a \), and \( d \).
  - Best choice for \( a = 16 \), and \( d = 2 \)
- Base-case sorting algorithm
  - Use std::sort
Comparisons and baselines

- Compared against cache aware sorting algorithms as well as quicksort.
- See paper for charts!
- Main takeaway: performance depends on the architecture and the input size
  - In some cases the overhead of Funnelsort is not worth the gain
  - For architectures with fast CPUs (where cache misses are costlier in comparison) and large input sizes, Funnelsort wins!
Discussion question

- Can we have an even simpler algorithm?