Engineering a Cache-Oblivious Sorting Algorithm

Gerth Stølting Brodal, Rolf Fagerberg, Kristoffer Vinther

Presented by William Qian

2020 March 5
6.886 Spring 2020
Overview

1. Lazy \(d\)-Funnelsort

2. Recipe
   - Ingredients
   - Step 1: \(k\)-merger structures
   - Step 2: Tuning basic mergers
   - Step 3: Degree of basic mergers
   - Step 4: Caching for basic mergers
   - Step 5: Base sorting algorithm
   - Step 6: Parameters \(\alpha\) and \(d\)

3. Evaluation

4. Discussion
Food for thought...

1. In what ways did the results of one experiment critically determine the parameters for a later one?

2. What hypotheses did the authors have? Which of these seem sensible but are not supported by the experiments?

3. How do the authors ensure that their experiments are robust, reliable, and reproducible? What do you find unusual?

4. How could some of these results have been discovered with the help of tuning tools like OpenTuner?
1 Lazy $d$-Funnelsort

2 Recipe
   - Ingredients
   - Step 1: $k$-merger structures
   - Step 2: Tuning basic mergers
   - Step 3: Degree of basic mergers
   - Step 4: Caching for basic mergers
   - Step 5: Base sorting algorithm
   - Step 6: Parameters $\alpha$ and $d$

3 Evaluation

4 Discussion
Central to (lazy) funnelsort

Recursively built of \( \sqrt{k} \)-mergers

Outputs of mergers on one level are inputs to parents

When buffers are empty, recursively invoke the filling algorithm

Funnelsort on \( n \) elements incurs \( O(1 + \frac{n}{B}(1 + \log_M n)) \)
Modified \textit{k}-merger

From prior work by Brodal and Fagerberg [1]:

\textbf{Fill}(v)

1. \textbf{while} \(v\)’s output buffer isn’t full
2. \textbf{if} left input buffer empty
3. \hspace{1em} \text{Fill(left child of } v)\]
4. \textbf{if} right input buffer empty
5. \hspace{1em} \text{Fill(right child of } v)\]
6. perform one merge step
Lemma 1. Let $d \geq 2$. The size of a $k$-merger (excluding its output buffer) is bounded by $c \cdot k^\frac{d+1}{2}$ for a constant $c \geq 1$. Assuming $B^\frac{d+1}{d-1} \leq \frac{M}{2c}$, a $k$-merger performs $O \left( \frac{k^d}{B} \log_M(k^d) + k \right)$ I/Os during an invocation.
Modified $k$-merger

Space bound:

$$S(k) = k^{\frac{1}{2}} \cdot k^{\frac{d}{2}} + (k^{\frac{1}{2}} + 1) \cdot S(k^{\frac{1}{2}})$$

\[ \leq c \cdot k^{\frac{d+1}{2}} \]

- $k^{\frac{1}{2}}$ buffers with $k^{\frac{d}{2}}$ size
- Recurse on $k^{\frac{1}{2}}$-size problems
  - 1 recursion “up”
  - $k^{\frac{1}{2}}$ recursions “down”
Modified $k$-merger

**I/O bound:**

1. Largest subtree with $\bar{k}$ leaves
   - Space: $\bar{k}^{\frac{d+1}{2}} \leq \frac{M}{2c}$
2. Parent has $\bar{k}^2$ leaves
   - Space: $(\bar{k}^2)^{\frac{d+1}{2}} > \frac{M}{2c}$
   - Input: “large buffers”
3. Remove large buffer edges
   - Connected *base trees*

\[ D_0 = \left\lceil \frac{\lg k}{2} \right\rceil \]

---

Input buffers

Output buffer

size $k^d$

size $\left\lceil k^{\frac{d}{2}} \right\rceil$

bottom trees

top tree
Modified $k$-merger

1 block $\times \bar{k}$ buffers: $\bar{k}B \leq \left(\frac{M}{2c}\right)^{\frac{2}{d+1}} \cdot \left(\frac{M}{2c}\right)^{\frac{d-1}{d+1}} \leq \frac{M}{2c}$

Base tree + 1 block $\times \bar{k}$ buffers $\leq \frac{M}{c}$ space

If $k$-merger is a base tree, output $k^d$ items in $O \left(\frac{k^d}{B} + k\right)$ I/Os

Otherwise, for Fill($v =$ root node of a base tree)

a. Loads $\Omega(\bar{k}^d)$ elements to output buffer
b. Base tree + 1 block/buffer = $O \left(\frac{1}{B} \bar{k}^{\frac{d+1}{2}} + \bar{k}\right)$

c. $\bar{k}^{d+1} > \frac{M}{2c} \implies \bar{k}^{d-1} > \left(\frac{M}{2c}\right)^{\frac{d-1}{d+1}} \geq B \cdot \frac{1}{B} \bar{k}^d \geq \bar{k}$

d. Casework: recursive calls may cause base tree reloads

e. Charge $O \left(\frac{1}{B}\right)$ I/O per large buffer insert
f. $\geq \left(\frac{M}{2c}\right)^{\frac{1}{d+1}}$ leaves means $O(\log_M k^d)$ large buffer inserts/item
Modified $k$-merger

- Per invocation:
  - $k$-merger is base tree: $O\left(\frac{k^d}{B} + k\right)$ I/Os
  - $k$-merge is not: $O\left(\frac{k^d}{B} \log M k^d\right)$ large buffer I/Os
  - Overall I/O cost bounded by $O\left(\frac{k^d}{B} \log M k^d + k\right)$
- Proof based on buffer size, any memory layout works!
- This paper: $D_0/D_0+1$ buffers have size $\alpha \lceil d^{\frac{3}{2}} \rceil$, $\alpha > 0$
Lazy $d$-Funnelsort

Recipe

- Ingredients
- Step 1: $k$-merger structures
- Step 2: Tuning basic mergers
- Step 3: Degree of basic mergers
- Step 4: Caching for basic mergers
- Step 5: Base sorting algorithm
- Step 6: Parameters $\alpha$ and $d$

Evaluation

Discussion
Ingredients

Cache-oblivious sorting implementation

- 1 baseline cache-oblivious sorting algorithm, theoretically efficient
- 1 state-of-the-art sorting algorithm, not necessarily cache-oblivious
- 1 or more workloads, aiming to cover useful sorting applications
- 1 or more data distributions, to simulate different workload types
- 1 consistent method for accurately measuring time
- Several machines and architectures, optional but recommended
- Many hypotheses that can translate into experiments
## Step 1: $k$-merger structures

<table>
<thead>
<tr>
<th>Allocator</th>
<th>Invocation pattern</th>
<th>Navigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom</td>
<td>Recursive</td>
<td>Pointers</td>
</tr>
<tr>
<td>Standard*</td>
<td>Iterative</td>
<td>Implicit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layout</th>
<th>Merger nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>stored with output buffer</td>
</tr>
<tr>
<td>DFS</td>
<td></td>
</tr>
<tr>
<td>vEB</td>
<td>stored separately</td>
</tr>
</tbody>
</table>

- BFS
- DFS
- vEB

William Qian

Engineering a C/O Sorting Algorithm

2020 March 5
Step 1: $k$-merger structures

Experiments

- Cartesian product of factors
- 28 experiments on 3 machines

Workload

- $k$ streams of $k^2$ items
- $k$-merger: $(\alpha, d) = (1, 2)$
- Basic merger degree($z$): 2
- $k \in [15, 270]$
- Measure $\left\lceil \frac{20000000}{k^3} \right\rceil$ merges
Step 1: $k$-merger structures

 Allocator
  - Custom
  ✓ Standard*

 Invocation pattern
  ✓ Recursive
  • Iterative

 Navigation
  ✓ Pointers
  • Implicit

 Layout
  • BFS
  • DFS
  ✓ vEB

 Merger nodes
  • stored with output buffer
  ✓ stored separately
Step 2: Tuning basic mergers

Idea: improve on the merge step of the basic mergers

- Basic merger
- Coarse bound-checking
- Hybrid bound-checking
- Hybrid Hwang-Lin

Experiments:
- Same as step 1, but with three additional \((\alpha, d)\) pairs:
  - \((1, 3)\)
  - \((4, 2.5)\)
  - \((16, 1.5)\)
Step 2: Tuning basic mergers

Results:

- Hwang-Lin has a large overhead
- Bound-checking is ineffective
- Hybrids work better
- Straightforward works best

CPU branch prediction is really good, hand-coding is just extra overhead
Step 3: Degree of basic mergers

Idea: multiway mergers: less data movement, more complex

- Basic mergers
- Various multiway mergers
- Looser trees [3]

Experiments:

- \((k, \alpha, d) = (120, 16, 2)\)
- 8 mergings of 1728000 elements
- \(z \in [2, 9]\)
Step 3: Degree of basic mergers

Results:
- 4- and 5-way mergers work best
- Looser trees don’t show inflection, but have high overhead
Step 4: Caching for basic mergers

Idea: construct one $k$-merger per level
- Each level uses the same size $k$-merger
- Reset and reuse the $k$-merger for merging in the same level

Experiments:
- $(\alpha, d, z) = (4, 2.5, 2)$
- Straightforward binary basic mergers
- Base case uses `std::sort()` for sizes $< \alpha z^d = 23$
- Workloads: between $[5000000, 200000000]$ elements

Results: 3-5% speedup across the board
Step 5: Base sorting algorithm

Idea: choose a good base case for sorting a small number of elements
Experiments:

- Insertion, selection, heap, shell, and \texttt{std::sort()} sorts
- Workload: input sizes from 10 to 100

Results: \texttt{std::sort()} is fastest
Step 6: Parameters $\alpha$ and $d$

Idea: choose good $\alpha$ and $d$ parameters

Experiments:
- $\alpha \in [1, 40]$
- $d \in [1.5, 3]$
- Workloads: various sizes

Results:
- $\alpha < 10$ produces a longer running time
- $d$ does not have a large impact at reasonable sizes
- Small $(\alpha, d)$ correspond to small buffer sizes
  - Cost of navigation and invocation spread over fewer merge steps
- Optimal $(\alpha, d) \approx (16, 2.5)$
1 Lazy $d$-Funnelsort

2 Recipe
   - Ingredients
   - Step 1: $k$-merger structures
   - Step 2: Tuning basic mergers
   - Step 3: Degree of basic mergers
   - Step 4: Caching for basic mergers
   - Step 5: Base sorting algorithm
   - Step 6: Parameters $\alpha$ and $d$

3 Evaluation

4 Discussion
Setup

Benchmarks:
- Funnelsort2 (binary basic mergers)
- Funnelsort4 (four-way basic mergers)
- Quicksort (GCC)
- Quicksort (Bentley & McIlroy)
- msort-c (cache-aware)
- msort-m (cache-aware)
- R-merge

Workloads (RAM):
- Inputs of sizes in RAM range
- Median of 21 trials

Workloads (Disk):
- Inputs on-disk
- Single-trial
Data

Disk-based experiments omitted for brevity
Takeaways

1. Arbitrary memory layouts can still hold asymptotic properties
   - But vEB structure still has practical benefits
2. Iterative optimizations can lead to a competitive algorithm
3. Cache-obliviousness overhead can be worth it!


Lazy $d$-Funnelsort

Recipe

- Ingredients
- Step 1: $k$-merger structures
- Step 2: Tuning basic mergers
- Step 3: Degree of basic mergers
- Step 4: Caching for basic mergers
- Step 5: Base sorting algorithm
- Step 6: Parameters $\alpha$ and $d$

Evaluation

Discussion
“Constructive feedback”

- Graphs
  - Too dense and somewhat poorly organized
  - Legend labels are inconsistent between graphs
- “std::sort() is really good” isn’t very novel
- Memory layout observation is exciting, but is eventually disappointing
- Methodology and experiment setups are fairly detailed and precise
- Engineering phase pattern is useful
  - Though visuals (e.g. graphs) would have been helpful
1. In what ways did the results of one experiment critically determine the parameters for a later one?

2. What hypotheses did the authors have? Which of these seem sensible but are not supported by the experiments?

3. How do the authors ensure that their experiments are robust, reliable, and reproducible? What do you find unusual?

4. How could some of these results have been discovered with the help of tuning tools like OpenTuner?