Performance Challenge #6

easyperf.net

Task: wordcount

Split a text and count each word's frequency, then print the list sorted by frequency in decreasing order. Ties are printed in alphabetical order.

Example:

```
$ echo "apple pear apple art" | ./wordcount.exe
apple   2
art     1
pear    1
```

Dataset: complete snapshot of Hungarian Wikipedia (5.4 GB).

Announcement: *https://easyperf.net/blog/2022/05/28/Performance-analysis-and-tuning-contest-6*
Code repository: *https://github.com/dendibakh/perf-challenge6*

Motivation

Leaderboard

Full Hungarian Wikipedia

Updated: June 13, 2016

Only the ones that finish are listed. The rest run out of memory.

Rank	Experiment	CPU seconds	User time	Maximum memory	Contributor
1	java -Xms2G -Xmx8G -classpath java:java/zah-0.6.jar WordCountOptimized	140.64	191.44	4232308	
2	java -Xms2G -Xmx8G -classpath java:java/trove-3.0.3.jar WordCountOptimized	183.17	219.39	4391804	
3	rust/wordcount/wordcount	208.86	193.34	6966228	Joshua Holmer
4	cpp/wordcount_clang	224.23	211.76	4933968	
5	cpp/wordcount	227.74	213.86	4934204	Dmitry Andreev, Matias Fontanini, Judit Acs
6	c/wordcount	270.15	255.95	2453228	gaebor

Can we beat it?

From: https://github.com/juditacs/wordcount/

Baseline implementation

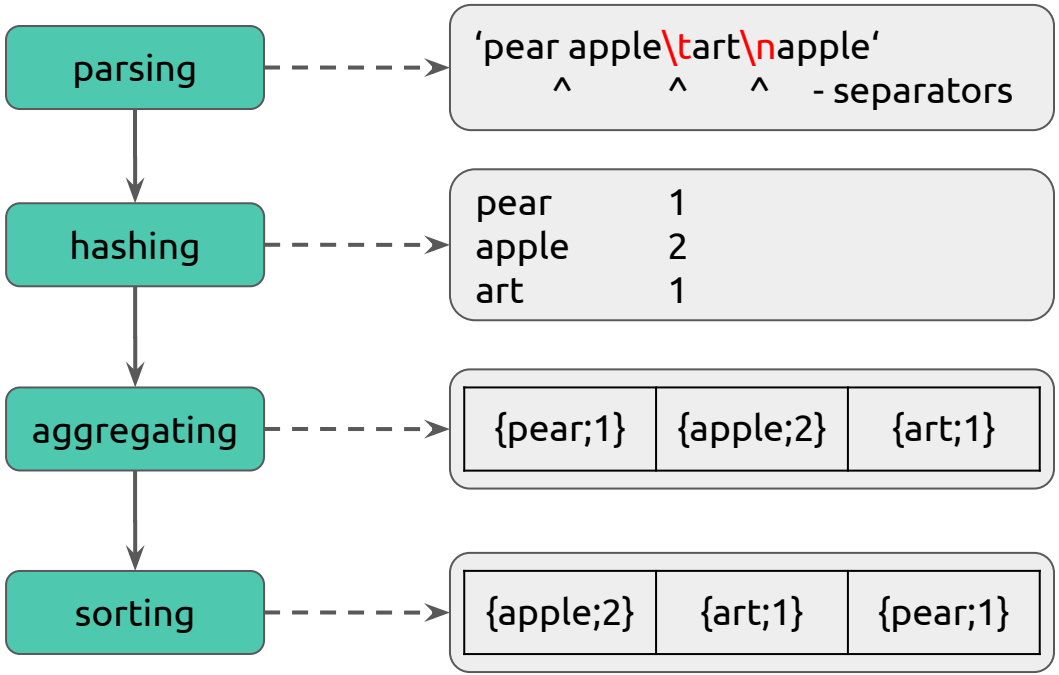
```
std::vector<WordCount> wordcount(std::string filePath) {
  std::unordered_map<std::string, int> m;
  m.max_load_factor(0.5);
  std::vector<WordCount> mvec;

  std::ifstream inFile{filePath};
  std::string s;
  while (inFile >> s)
    m[s]++;

  mvec.reserve(m.size());
  for (auto &p : m)
    mvec.emplace_back(WordCount{p.second, move(p.first)});

  std::sort(mvec.begin(), mvec.end(), std::greater<WordCount>());
  return mvec;
}
```

parsing & hashing
(123.1 sec, 75%)

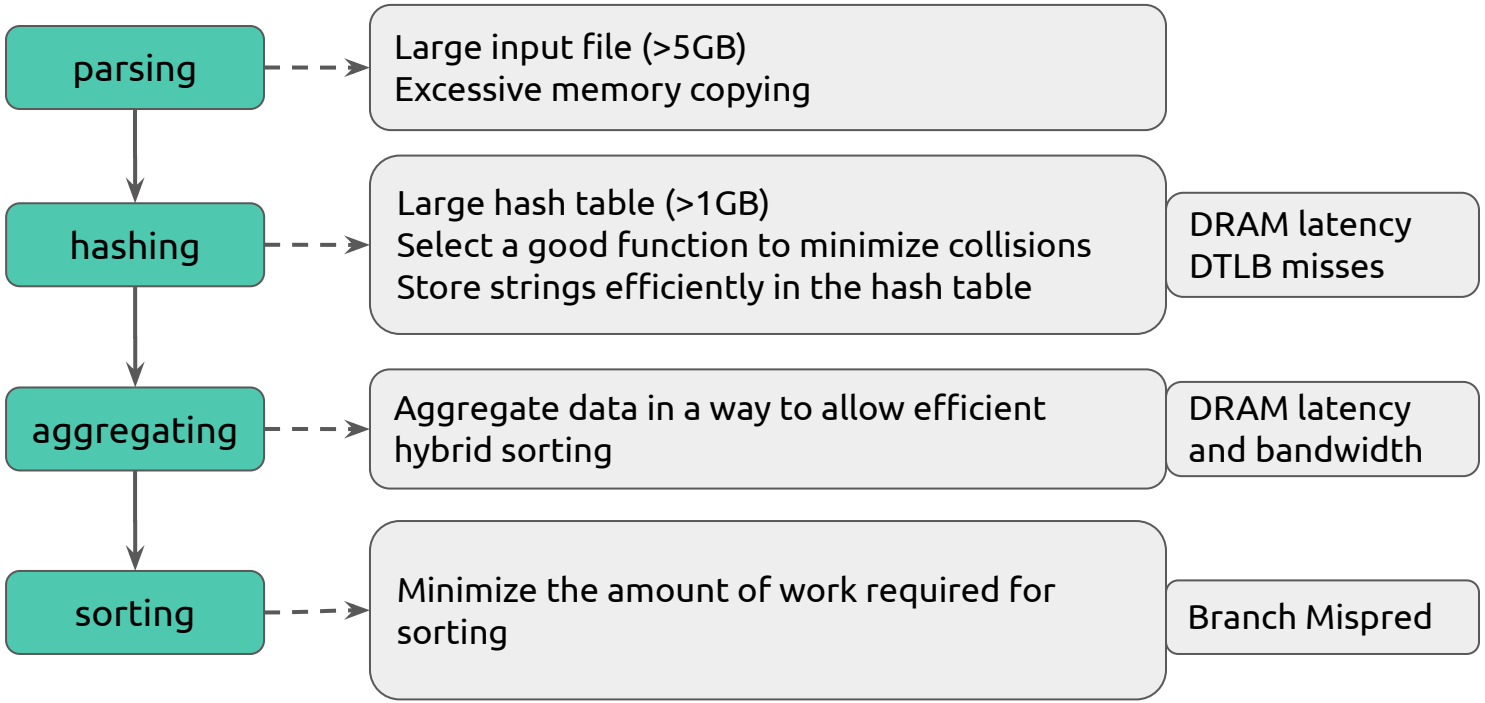
hash_map -> vector copy
(4.3 sec, 2.6%)

sorting (22.6 sec, 13.6%)

High-level view



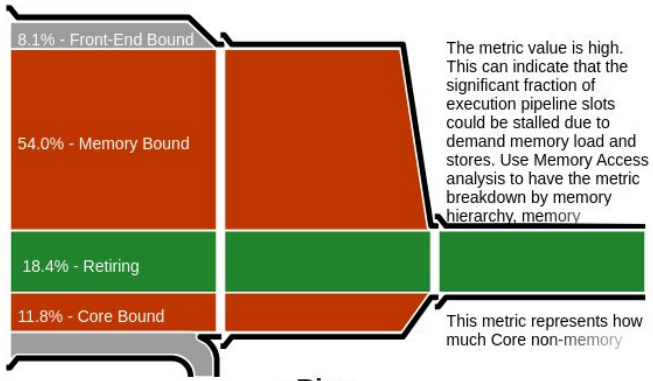
Bottlenecks breakdown



Top-Down Uarch Analysis

Elapsed Time: 178.885s

Clockticks:	645,798,600,000	
Instructions Retired:	451,253,100,000	
CPI Rate:	1.431	
Retiring:	18.4%	of Pipeline Slots
Front-End Bound:	8.1%	of Pipeline Slots
Bad Speculation:	7.8%	of Pipeline Slots
Back-End Bound:	65.7%	of Pipeline Slots
Memory Bound:	54.0%	of Pipeline Slots
L1 Bound:	6.7%	of Clockticks
L2 Bound:	1.2%	of Clockticks
L3 Bound:	5.8%	of Clockticks
DRAM Bound:	48.0%	of Clockticks
Memory Bandwidth:	29.0%	of Clockticks
Memory Latency:	42.5%	of Clockticks
Store Bound:	0.2%	of Clockticks
Core Bound:	11.8%	of Pipeline Slots
Average CPU Frequency:	3.8 GHz	
Total Thread Count:	2	
Paused Time:	0s	

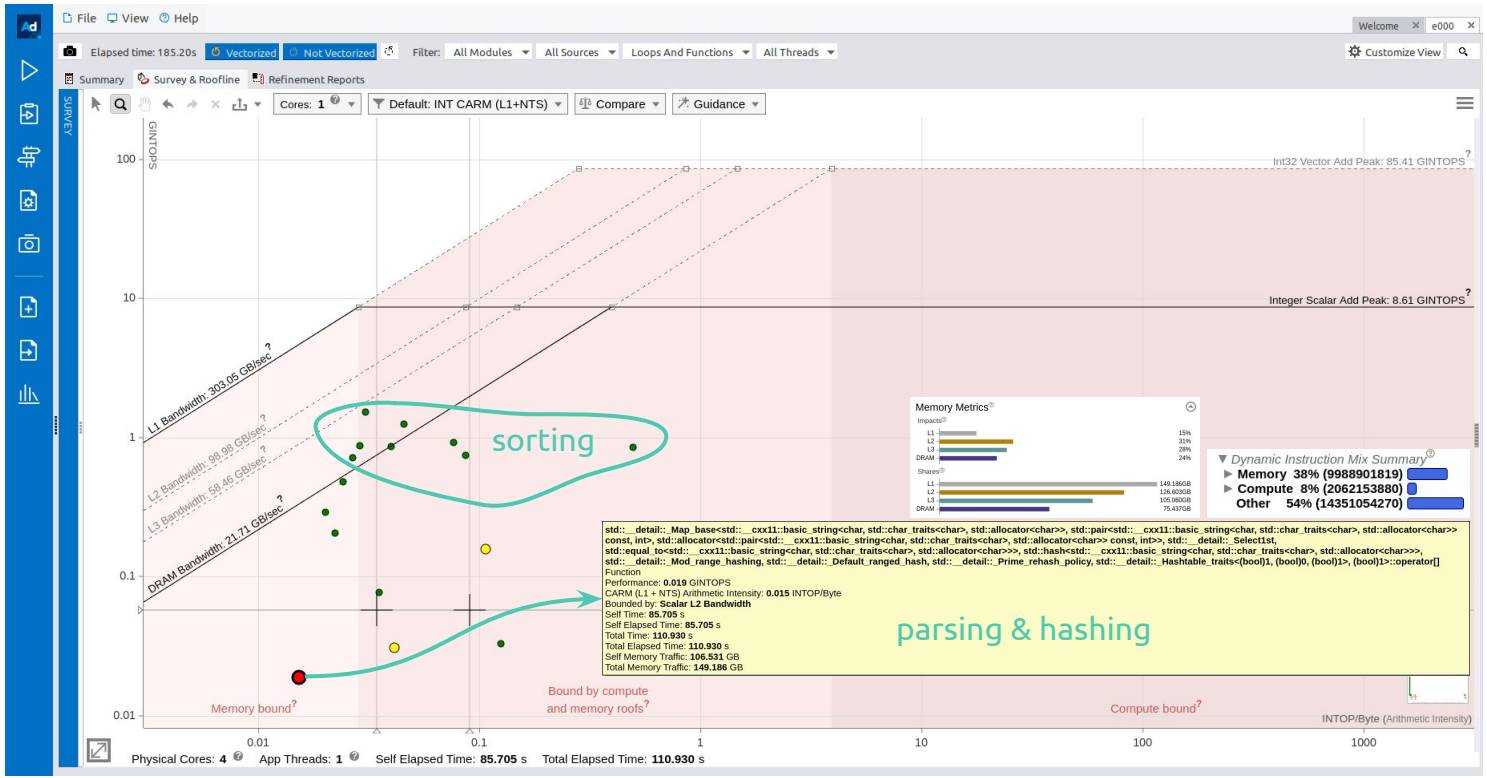


µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Intel Core i5-8259U @3.80GHz Turbo, 6MB L3-cache + 16GB RAM + Ubuntu 20.04 + Clang 14.0

Roofline analysis

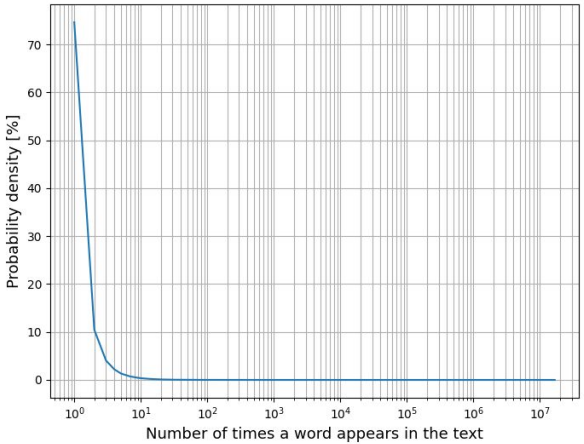


Intel Core i5-8259U @3.80GHz Turbo, 6MB L3-cache + 16GB RAM + Ubuntu 20.04 + Clang 14.0

Observations

Total number of words: ~500M

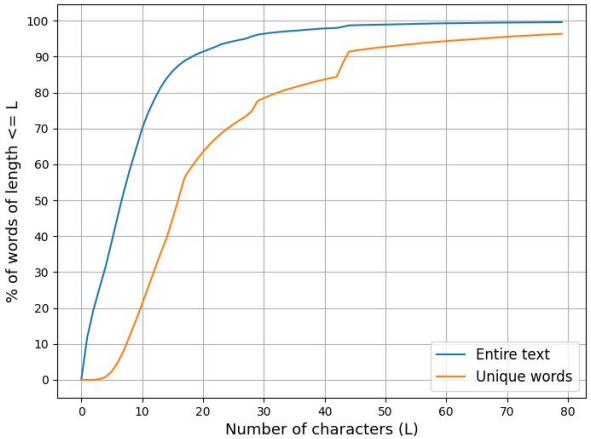
1. Majority of words occur only once



Example: 70% of all words occur only once

This is relevant for sorting since, for example, we can handle unique words separately.

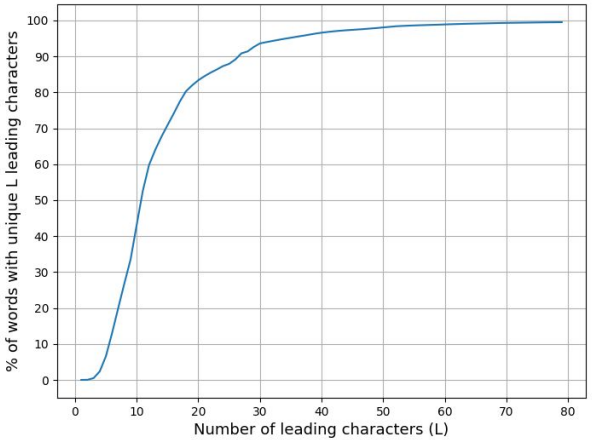
2. Most of the words are shorter than 32 characters



Example: 90% of all words are shorter than 20 characters

This gives us insight into choosing the short string optimization size, e.g. how many characters we want to store in the hash_map nodes.
Unique words info is relevant for sorting.
(in the later slides)

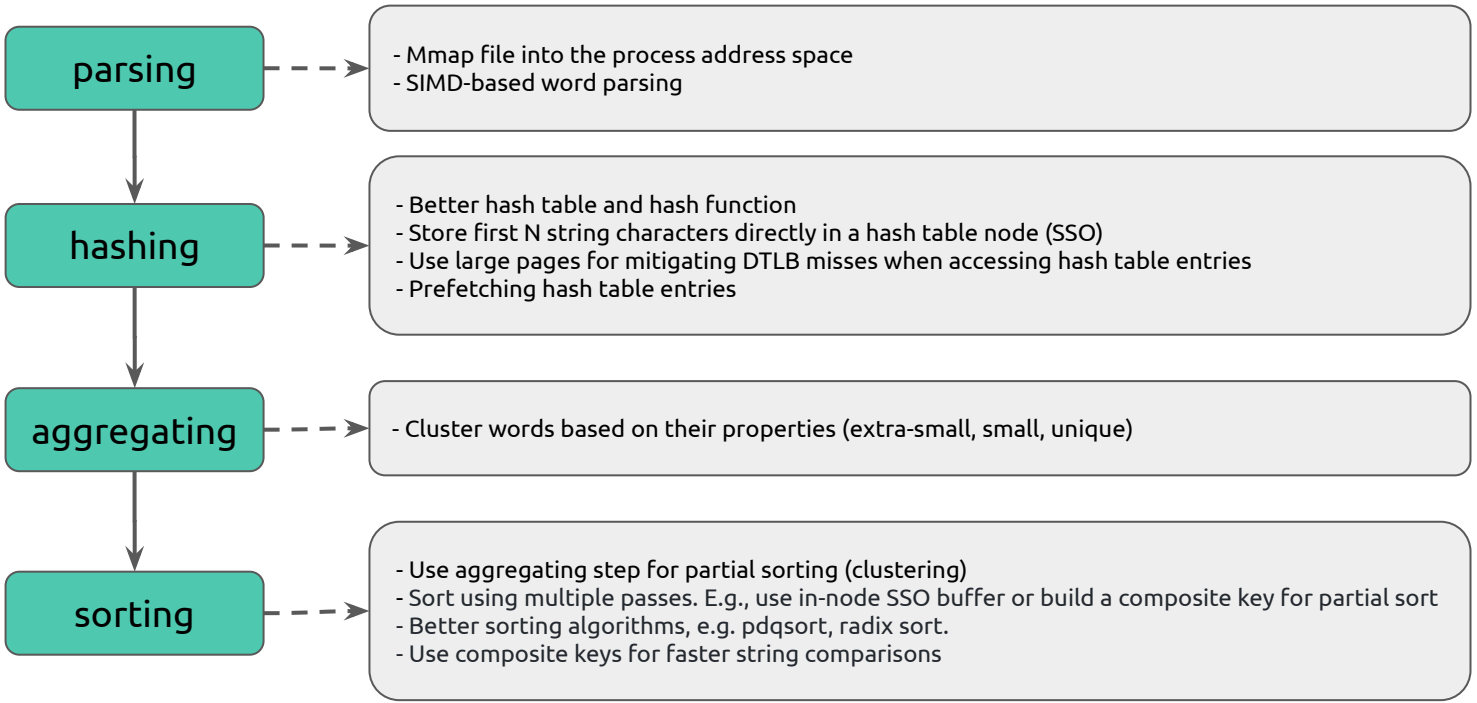
3. Most of the words differ in the first 32 characters



Example: 45% of words differ with all other words within the first 10 characters

This is relevant for comparing strings (used when resolving hash collisions and in sorting).

Optimizations



Disclaimer: no guarantee those optimizations will help on different inputs.

File access

Access via fstream is slow. The file contents can instead be mmap'ed into the address space of the process. This allows for direct access to the data and optimal parsing (see next slide).

fstream -> mmap

Baseline

```
std::ifstream inFile{filePath};
std::string word;
while (inFile >> word)
  // process word
```

Optimized (Linux)

```
int fd =open(filePath.c_str(), O_RDONLY);
const char* mm = static_cast<const char*>(mmap(nullptr, fs, PROT_READ,
MAP_PRIVATE, fd, 0U));
struct stat sb;
fstat(fd, &sb);
std::string_view file_contents{mm, static_cast<size_t>(sb.st_size)};
// Parse raw contents
```

>2x speedup

File access (windows)

Baseline

```
std::ifstream inFile{filePath};
std::string word;
while (inFile >> word)
   // process word
```

Optimized (Windows)

```
void* getDataPtr(const std::string& filePath) {
   HANDLE hFile;    HANDLE hMap;

   OFSTRUCT buffer;
   hFile = CreateFile(filePath.c_str(), GENERIC_READ,
                      FILE_SHARE_READ, 0, OPEN_EXISTING,
                      FILE_ATTRIBUTE_NORMAL, 0);

   hMap = CreateFileMapping(hFile,
          NULL,                    // Mapping attributes
          PAGE_READONLY,           // Protection flags
          0,                       // MaximumSizeHigh
          0,                       // MaximumSizeLow
          NULL);                   // Name
   return MapViewOfFile(hMap,
          FILE_MAP_READ,           // dwDesiredAccess
          0,                       // dwFileOffsetHigh
          0,                       // dwFileOffsetLow
          0);                      // dwNumberOfBytesToMap
}
```

Text parsing

Finding whitespace separators

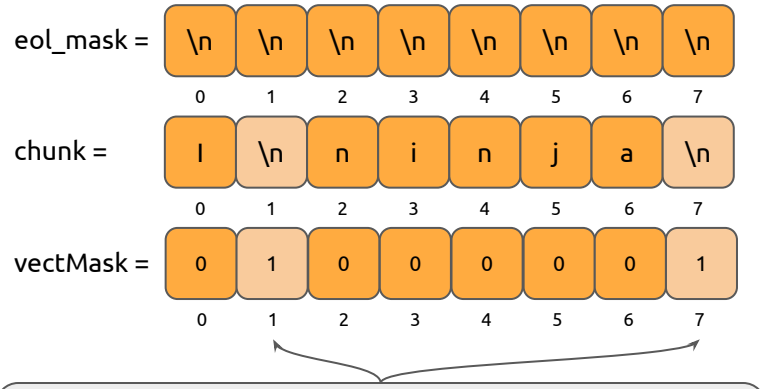
Baseline

```
const uint8_t* buf = ...;
const uint8_t* word_begin = buf;
while (*buf == ' ' || *buf == '\t' || *buf == '\n')
   ++buf;
const uint8_t* word_end = buf;
```

Optimized

```
// process 32 characters at a time
const __m256i eol_mask = _mm256_set1_epi8('\n');
__m256i vectMask = _mm256_cmpeq_epi8(chunk, eol_mask);
uint32_t mask = _mm256_movemask_epi8(vectMask);

while (mask) {
 uint32_t maskPos = _tzcnt_u32(mask);
 // maskPos has position of a next separator
 mask >>= maskPos;
 // ...
}
// !!! repeat the same for ' ' and '\t' separators
```

The individual words can be parsed out using AVX2 vector intrinsics, checking for whitespace 32 characters at a time.

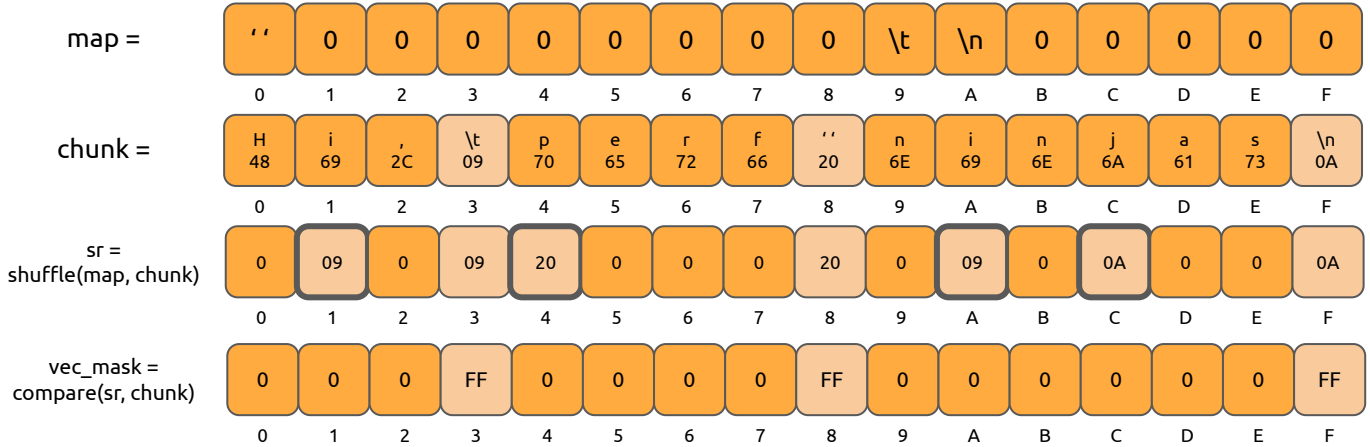


`tzcnt` gives the position of the leftmost bit set. We use it to determine begin and end offsets of a word. Then we shift right the mask to find the next bit set.

~2.5x speedup

The vpshufb trick

_mm256_shuffle_epi8 (__m256i a, __m256i ctrl) shuffles the 8-bit integers within the 128-bit lanes of the first argument according to the low 4 bits of the corresponding element in the second (control) argument. Let us therefore consider a single lane. The following technique lets us compare for all whitespace characters at once



' ' == 0x2**0** '\t' == 0x0**9** '\n' == 0x0**A**

If the low 4 bits of a character match the whitespace character C, it is mapped to C, otherwise to 0. Note that since the high 4 bits are ignored, some extra characters get matched.

To ensure the high 4 bits also match C, compare the shuffle result to the original chunk. Only whitespace characters get mapped to themselves!

1. Create map vector by setting the entries corresponding to the low 4 bits of the whitespace characters to those characters
2. Load text chunk
3. Shuffle the mask using the chunk as control
4. Compare the result of the shuffle with the original chunk
5. Whitespace bitmask obtained, proceed as usual

Note both the shuffle and compare instructions have throughput 0.5 CPI
Speedup compared to fstream: **10x**

```
uint32_t getWhitespaceMask(__m256i chunk) {
  const auto shuffle_map = _mm256_setr_epi8(
    ' ', 0, 0, 0, 0, 0, 0, 0, '\t', '\n', 0, 0, 0, 0, 0,
    ' ', 0, 0, 0, 0, 0, 0, 0, '\t', '\n', 0, 0, 0, 0, 0);
  auto shuf_res = _mm256_shuffle_epi8(shuffle_map, chunk);
  auto ws_mask = _mm256_cmpeq_epi8(shuf_res, chunk);
  return _mm256_movemask_epi8(ws_mask);
}
```

COMPILER EXPLORER

Hashing

Hash tables used:

- Robin-hood
- Boost:flat_hash_map
- Phmap::flat_hash_map
- own implementations

Design goals and tradeoffs:

- You definitely want open-addressing hash table implementation.
- Others?

Hash functions used:

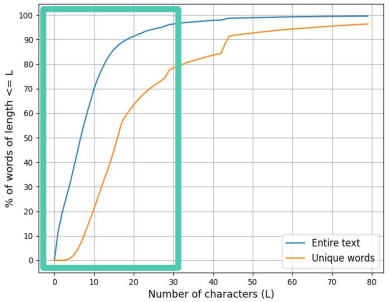
- crc32
- FNV-1a
- Murmur2
- fx_hash
- wyhash
- xx3hash

Design goals and tradeoffs:

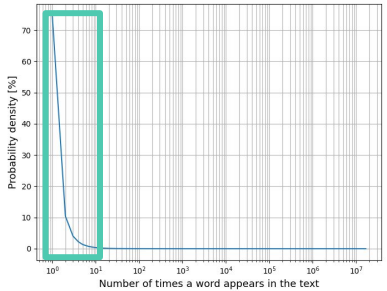
- Fewer hash collisions
- Faster hash computation
- Less bits for the hash in case you want to store it within a node

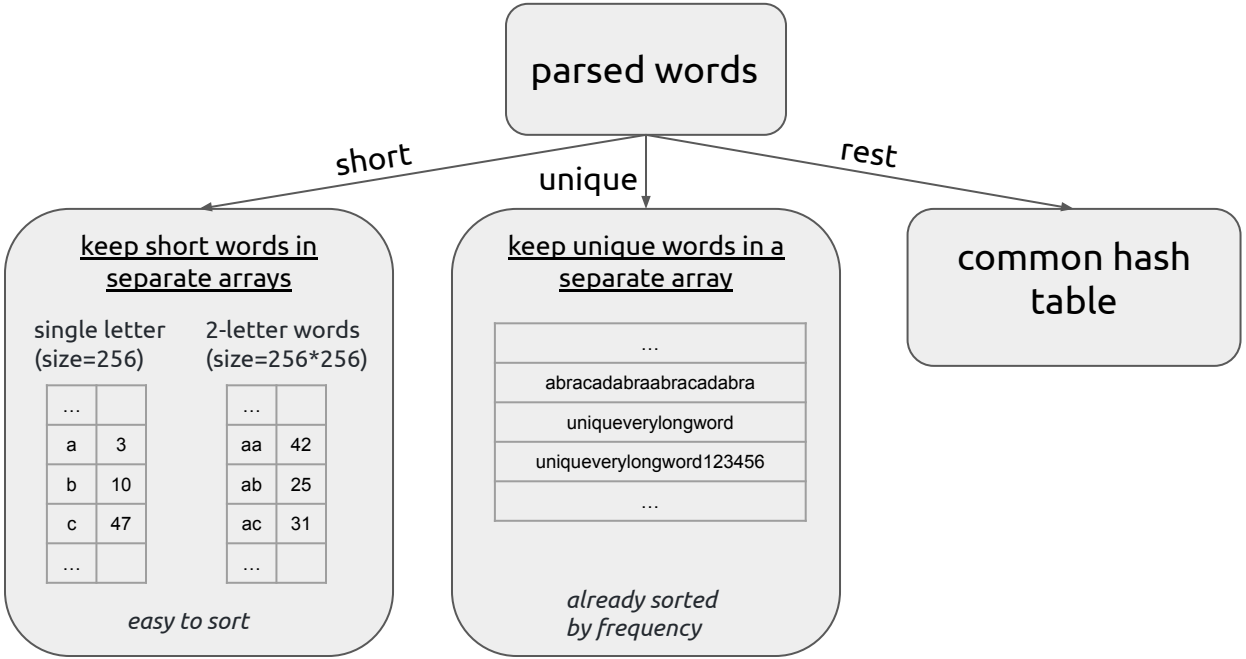
Hybrid Hashing

Most of the words are short…



… and unique.





Small String Optimization (SSO)

Idea 1: store the first N characters of a string directly in the hash table nodes. N < 32 to make it fit into the YMM register (256 bit) to allow faster SIMD-based strcmp.

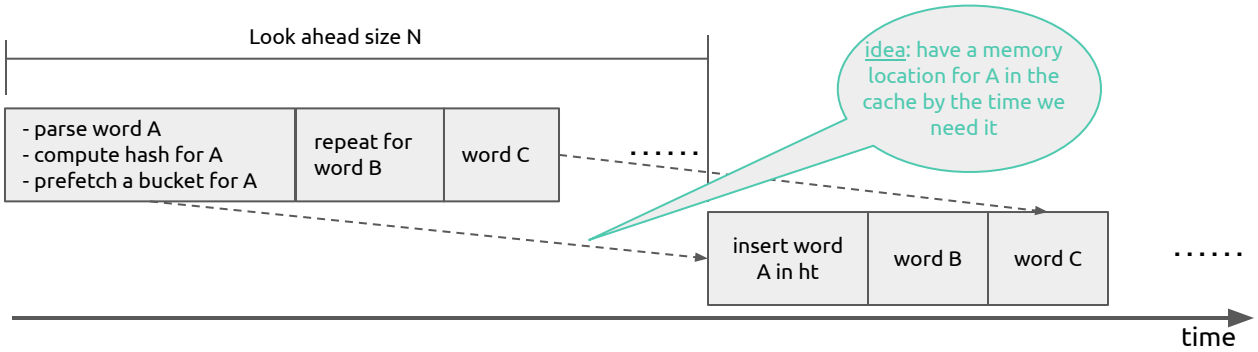
```
struct hash_tabe_node {
  // ...
  __m256i str;
};

// string_bytes - a chunk of 32 bytes;
// str_length - lenght of a string within that chunk

// zero out all the bytes after str_length
str_YMM = _mm256_and_si256(string_bytes,
make_mask(str_length));

if (_mm256_movemask_epi8(_mm256_cmpeq_epi8(node->str,
str_YMM)) == 0xFFFFFFFF) {
  // strings are equal
}
```

Idea 2: use a reversible hash for strings of length <=8, then you don't have to store a string pointer in the hash table entry because you can rematerialize it from the hash.

```
// treat strings <= 8 characters as uint64 x
uint64 x = *((uint64*)str);
hash = x * 0x517cc1b727220a95ull;

uint64 str = hash * 0x2040003d780970bdull;

// x == (x * 0x517cc1b727220a95ull) *
                0x2040003d780970bdull
```

Hash table prefetching

Cache misses when accessing hash table buckets are inevitable due to the fact that buckets are accessed in random order. But the penalty of those cache misses can be hidden if we batch them together or overlap them with execution. Assumption: parsing a word and computing its hash is faster than the L3 cache miss latency (300+ cycles).



Pseudocode:

```
// for a chunk of 256KB
auto words = parseWords(chunk);
auto hashes = computeHashes(words);
__builtin_prefetch(hashes);
auto buffer = storeIntoBuffer(words, hashes);

for (auto word: buffer)
  ht.insert(word);
```

Considerations:

1. Prefetching helps more in cases when there are many words in a memory chunk.
2. The size of a chunk must not be large to fit into L1/L2 cache (tunable).
3. Instead of processing in chunks, you could try to have a single loop and overlap execution of a word X with prefetching for a word X+N.

Utilizing Large Pages

Reduce the frequency of missing in virtual -> physical page translation cache for data (dTLB), when performing random accesses to entries of a hash table.

Baseline

```
auto size = sizeof(hashNodeT) * NUM_BUCKETS;
HashTableT* ht = (HashTableT*)malloc(size);
```

```
$ perf stat -e 'dTLB-loads,dTLB-load-misses' ./wordcount large.data

15,982,080,125   dTLB-loads
   299,612,763   dTLB-load-misses  # 1.87% of all dTLB accesses

      31.594642156 seconds time elapsed
```

Optimized (Linux)

```
auto size = sizeof(hashNodeT) * NUM_BUCKETS;
void hgtblAllocPtr = mmap(0, size, (PROT_READ | PROT_WRITE),
(MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB), 0, 0);
HashTableT* ht = (HashTableT*)hgtblAllocPtr;
```

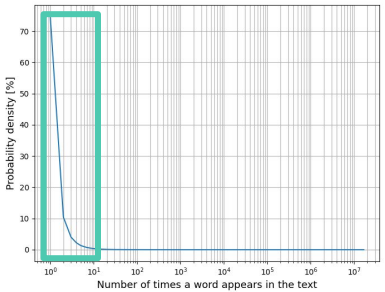
```
$ perf stat -e 'dTLB-loads,dTLB-load-misses' ./wordcount large.data

13,926,563,026   dTLB-loads
     9,879,928   dTLB-load-misses  # 0.07% of all dTLB accesses

      21.374012605 seconds time elapsed
```

~1.5x speedup

Considering that hash table is quite big (~1.5GB), default page size (4KB on x86) requires 250'000 entries, while large page size (2MB) requires only 500 entries.

Hybrid Sorting 1

Most of the words are unique



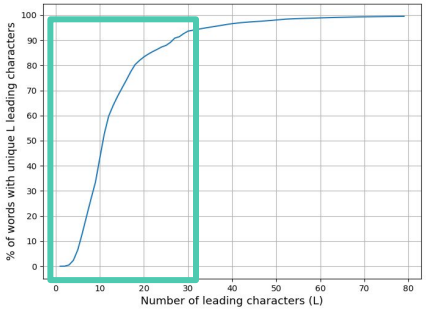
Idea: Use the aggregate phase to “bucketize” the words based on the number of their occurrence. After that only lexicographical comparison remains.

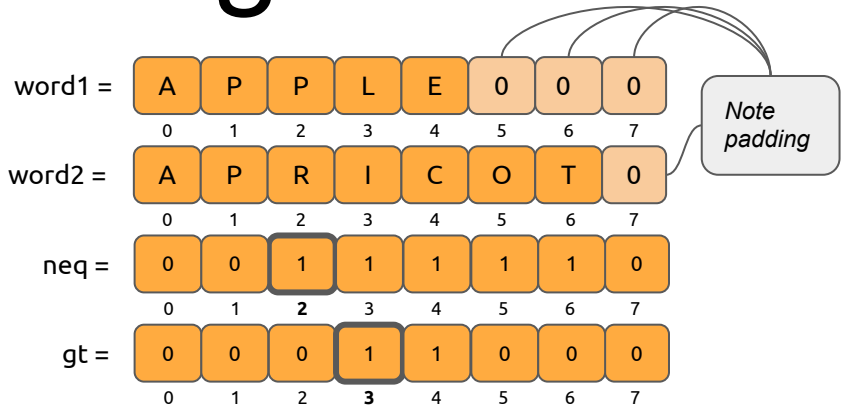
Put words with frequency less than N=32 into separate buckets indexed by frequency. This way you only need to sort them lexicographically.

Hybrid Sorting 2

Idea: Use in-node SSO buffers for faster lexicographical comparisons. Buffers avoid indirection. Use SIMD instructions instead of strcmp.

Most of the words differ in <32 characters






```
bool cmpLess(__m256i v1, __m256i v2) {
  auto eq_result  = _mm256_cmpeq_epi8(v1, v2);
  auto leq_result = _mm256_cmpeq_epi8(_mm256_min_epu8(v1, v2), v1);
  auto neq_mask = ~_mm256_movemask_epi8(eq_result);
  auto gt_mask  = ~_mm256_movemask_epi8(leq_result);
  auto neq_ind  = __builtin_ctz(neq_mask);
  auto gt_ind   = __builtin_ctz(gt_mask);
  return gt_ind > neq_ind;
}
```

COMPILER EXPLORER

Definition: w1 lexicographically compares less to w2 iff at the position of the first character that they differ, the character from w1 is smaller than the character from w2.
Rephrased: the position where w1 and w2 first differ is smaller than the first position where w1 > w2.
Implementation: get comparison bitmasks, obtain the requisite positions using tzcnt.
Notes:

- Pad buffers with zeros
- Unsigned comparison - no direct HW support

Benefits:

- 13 instructions for 32 characters
- No branches!

Other Sorting Ideas

Idea: Use integer keys for lexicographical string comparison.

Example: View first 4 bytes of a string as int32.
w1 = ‘agfq’ (0x 61 67 66 71)
w2 = ‘agos’ (0x 61 67 6F 73)

We can infer that w1 < w2 if we compare those strings as integers.

Need to convert to big-endian when reading ints from memory using bswap builtin/instruction.

Idea: Better sorting algorithm.

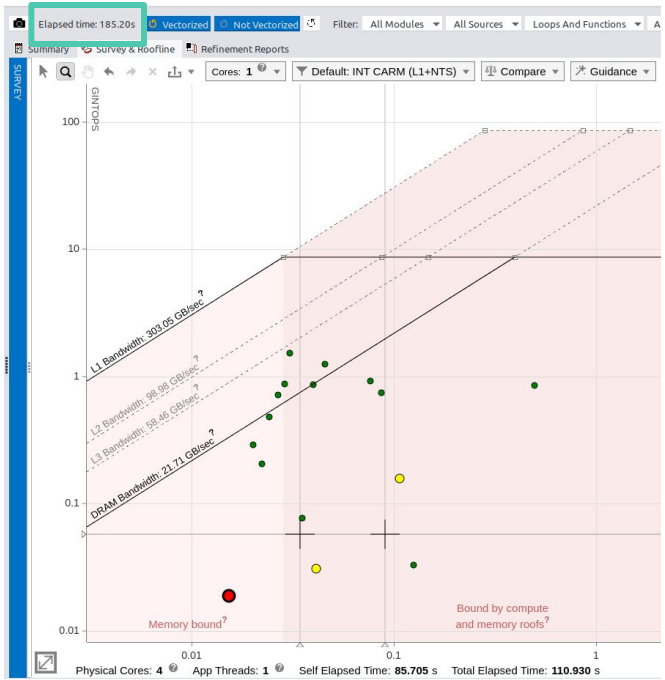
- pdqsort
- Radix sort

Idea: use hashes and string length for faster comparisons.

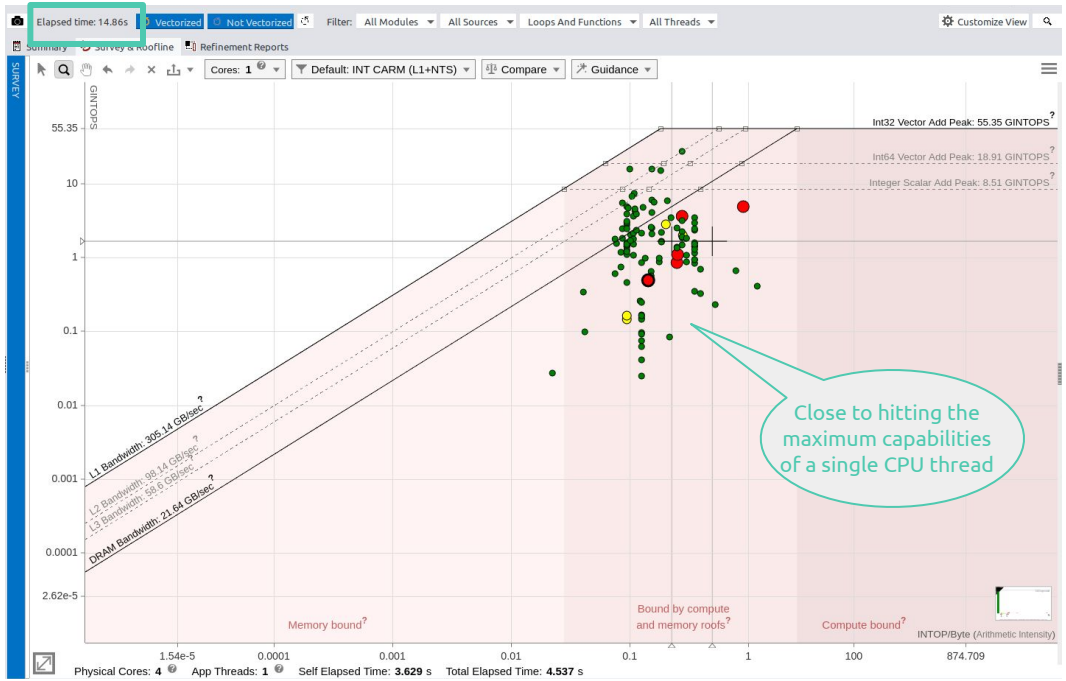
TODO

Roofline analysis

Baseline



Top solution



Scoretable

Baselines on target machines
Linux: 163 sec
Windows: 220 sec

Score = Linux speedup + Windows speedup

Name	Solution Linux (sec)	Linux speedup	Solution Windows (sec)	Windows speedup	Parsing	Hashing	Sorting
Robert Burke	13	12.5x	16	13.7x	- Mmap file into the process address space - SIMD-based word parsing (pshufb)	- Robin-hood hash-table - wyhash hash function - SSO - Large pages - prefetching hash_map buckets	- Radix sort with insertion sort for short subarrays
Stellaris62	25	6.5x	27	8.1x	- Mmap file into the process address space - SIMD-based word parsing (non-pshufb)	- Own impl of a hash table w/ open addressing - hash function - crc32 - SSO - Large pages	- Hybrid sorting (spec for freq=1, freq<32,freq>=32) - pdqsort
Andrey Evstyukhin	47	3.5x	55	4x	- Reading in 1MB chunks - SIMD-based word parsing (blends)	- boost:flat_hash_map - FNV-1a hash function - prefetching hash_map buckets	- Use hashes and string length for faster comparisons
Jakub Gałecki	52	3.1x	55	4x	- Mmap file into the process address space - SIMD-based word parsing (pshufb)	- Robin-hood hash map - Murmur2 hash function - SSO	- Sort the buckets of similar counts - Sort in multiple passes
Mansur Mavliutov	52	3.1x	59	3.7x	- Mmap file into the process address space	- Separate data structure for short strings - Hash table w/ open addressing - Big block allocations	- Hybrid sort (spec for freq=1)
Adam Richardson	44	3.7x	77	2.8x	- fread with a large buffer - SIMD-based word parsing	- Linear probing hashmap - Arena allocation with hugepages for strings	- Composite integer sort key - Sort in multiple passes
Franek Korta	64	2.5x	83	2.6x	- Reading in 2MB chunks	- xx3 hash - Separate data structure for short strings - SSO up to 8 characters - prefetching hash_map buckets	
Alexey Shmelev	82	2x	96	2.3x	- fread with a large buffer	- Separate data structure for short strings and a separate one for and unique strings - Own impl of a hash table	- Composite integer sort key - Sort in multiple passes - Sort the buckets of similar counts
Adam Folwarczny	94	1.7	85	2.6x	- fread with a large buffer	- Robin-hood hash table	- Sort the buckets of similar counts - Sort in multiple passes
Ole Schulz-Trieglaff	103	1.6x	210	1.1x	- Mmap file into the process address space	- phmap::flat_hash_map hash table	

Code available: https://github.com/dendibakh/perf-challenge6

SSO = Short String Optimization

Q&A

Would love to hear your feedback!