CS 4414: Recitation 9

Sagar Jha

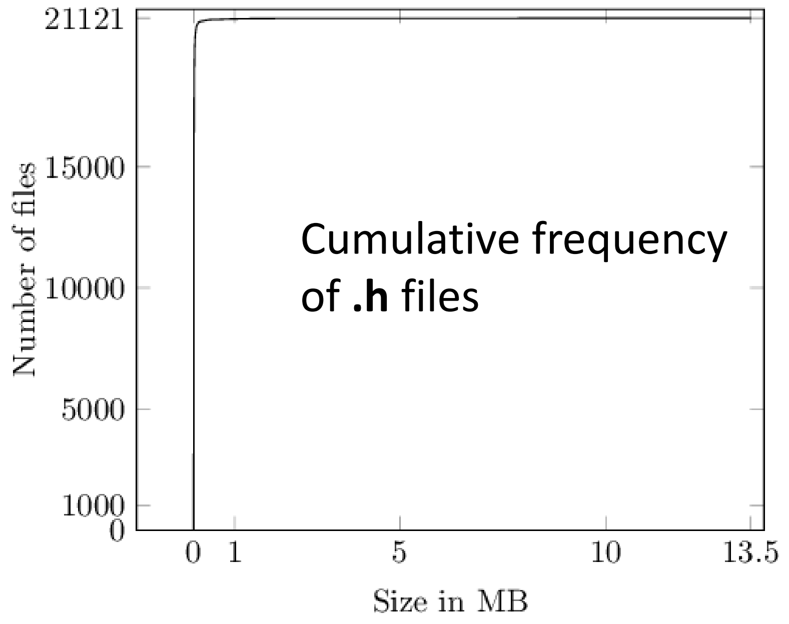
Today: Multithreading Part II

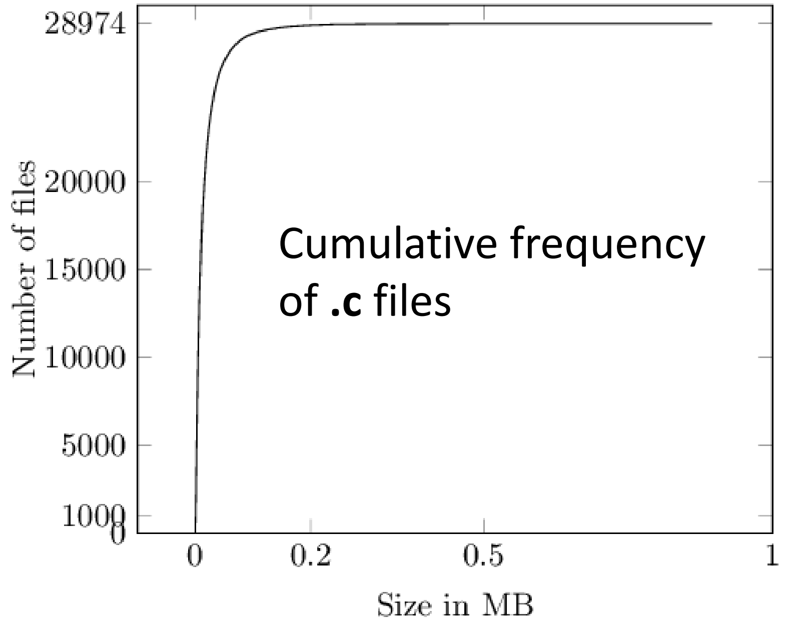
- Surveying Linux source code
- Design of wc++
- Evaluating wc++ on my laptop and Fractus
- Identifying bottlenecks through performance statistics
- Evaluating alternate design choices
- Improving performance guided by performance characteristics

First things first: Survey of linux-5.8-rc7

First things first: Survey of linux-5.8-rc7

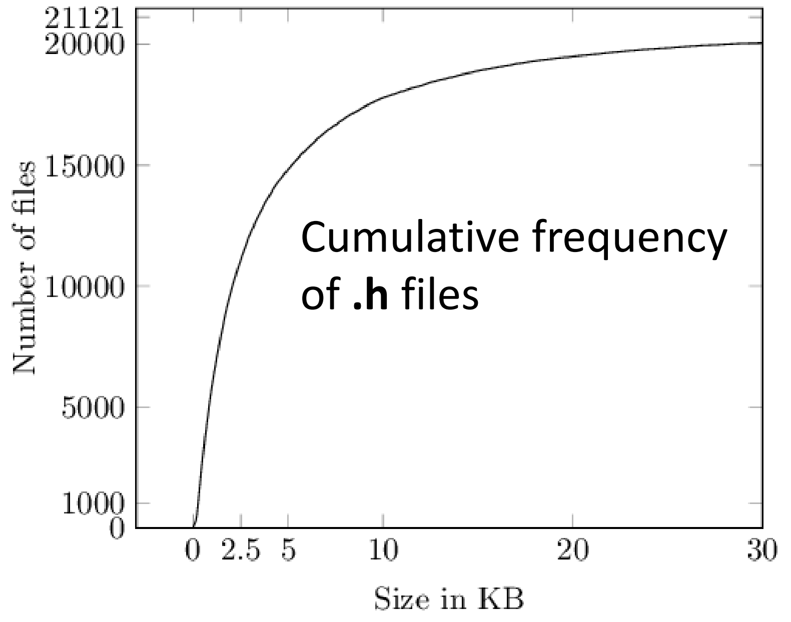
- ~21K header files (.h) and ~29K source files (.c)
- Max header file size is 13.5MB, almost all header files are < 30KB
- Max source file size is ~1MB, almost all source files are < 100KB





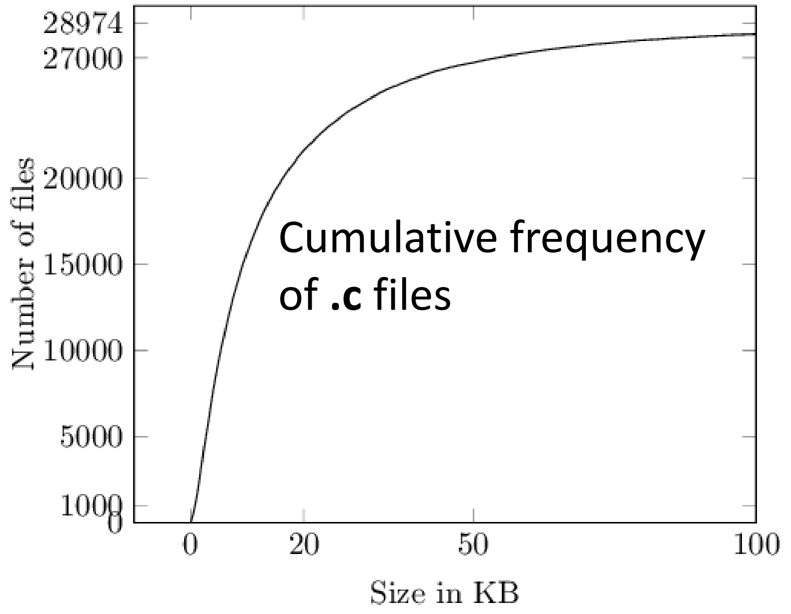
linux-5.8-rc7: Taking a closer look

- Nearly 25% of header files are < 900B
- ~50% and ~75% of header files are < 2.3KB and 6.25KB, resp.
- ~95.0% of header files are < 30KB



linux-5.8-rc7: Taking a closer look

- Nearly 25% of source files are < 3.8KB
- ~50% and ~75% of header files are < 8.8KB and 20.4KB, resp.
- ~97.8% of header files are < 100KB



How did I compute the statistics?

Linux Command Line ☺

- find searches for a file inside a directory
- pipe (|) feeds the output of one command as input to the other
- wc –l counts the number of lines in the input
- xargs sends input lines to the command as arguments
- du –b outputs the size of its argument in Bytes
- awk (or gawk) does pattern scanning and processing on the input
- sort –n sorts lines numerically
- uniq –c counts the number of occurrences of repeated lines

Linux commands to compute the stats

Compute the number of files of a given type

- find ./linux-5.8-rc7 -name *.h | wc -l

Compute the file size frequency

- find ./linux-5.8-rc7 -name *.c | xargs du -b | awk '{print $1}' | sort -n| uniq -c > c_file_stats

Design of wc++

Main thread

- Collects the pathnames of all header and source files in the given directory
- Spawns the given number of worker threads

1

Worker thread 1

- Goes through its set of files and computes results locally
- Merges local results sequentially into the global map

Worker thread 2

- Goes through its set of files and computes results locally
- Merges local results sequentially into the global map

Worker thread 3

- Goes through its set of files and computes results locally
- Merges local results sequentially into the global map

2

Main thread

- Waits for all worker threads to exit
- Prints the results

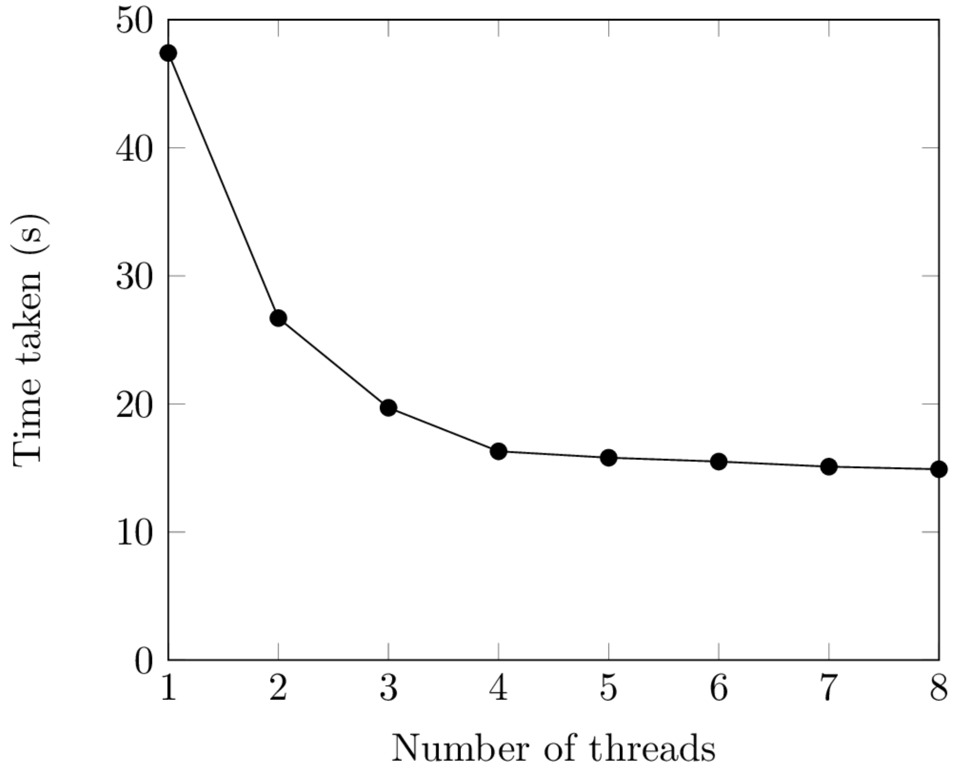
3

Before measuring performance: Not printing the results on the console

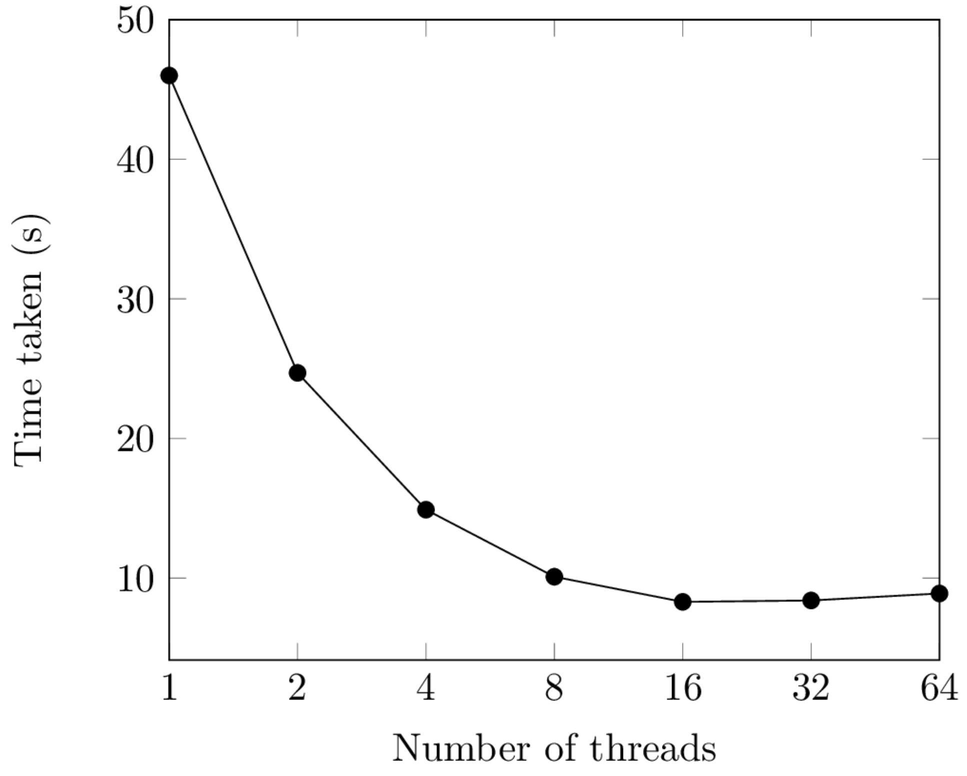
- The final result consists of more than four million words (4067253) with their frequencies
- Specification requires us to print in the order of the most frequent words first, breaking ties alphabetically
- We are not going to 4 million lines to the console! We are still going to compute the results in the right sort order
- Printing has to be single-threaded to preserve the sort order, but this takes more than 10 minutes!
- The rest of the program that computes the results takes less than a minute in all the cases
- This defeats the purpose of multithreading
- No program focused on performance should be printing so many lines

wc++ performance evaluation

- Initially, time reduces sharply with the number of threads, then kind of stagnates. Amdahl’s law explains why performance does not increase proportionally with threads
- Hyperthreading does not increase performance appreciably since threads have to share the same cores. For more on the impact of hyperthreading, read https://medium.com/@ITsolutions/will-hyper-threading-improve-processing-performance-15cba11add74
- Time taken on the Fractus machine is significantly better than on my laptop



Time on my laptop (4 physical, 8 logical cores)



Time on compute28 (32 physical, 64 logical cores)

Taskset: controlling thread to core assignment

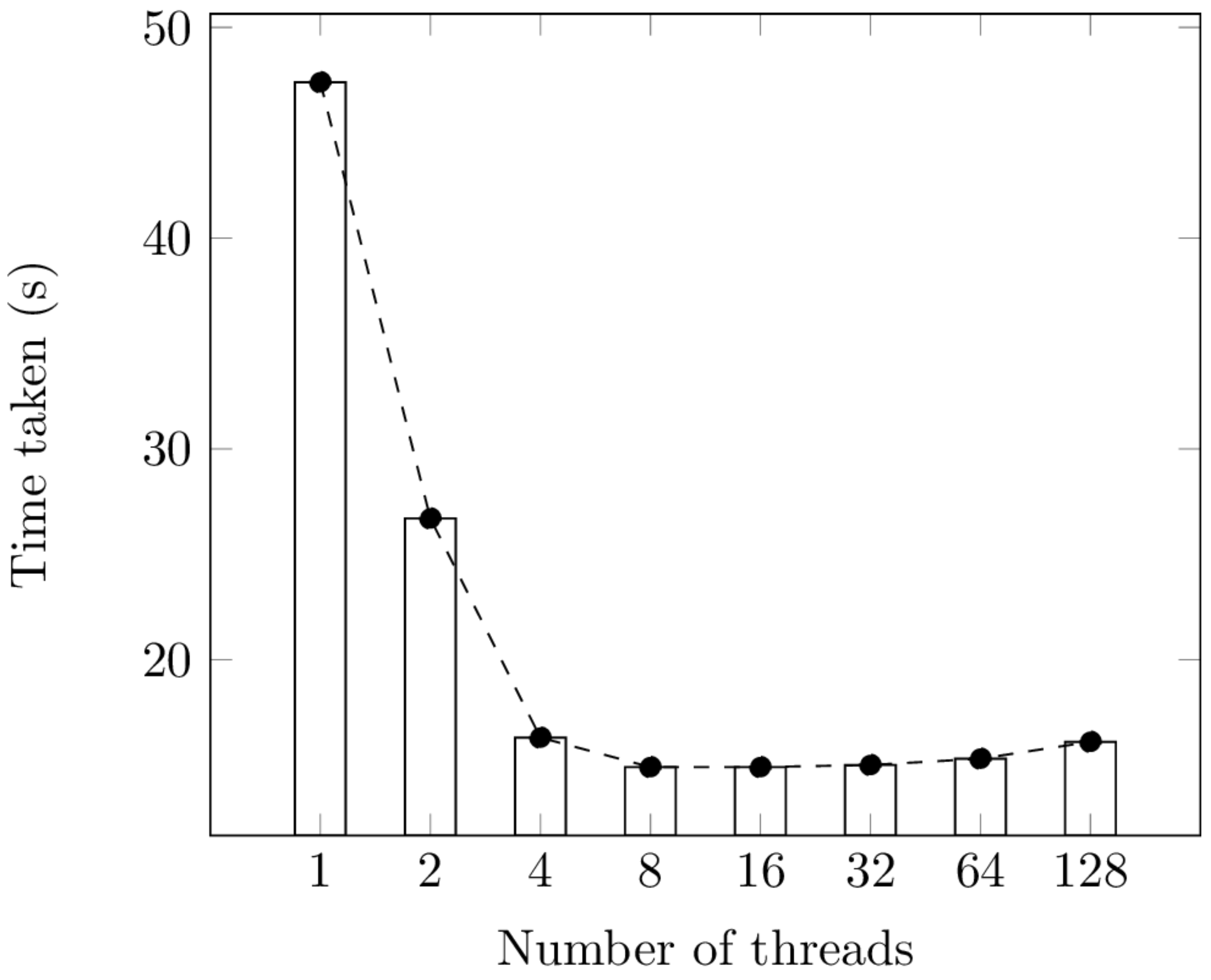
- Format: taskset mask command [arg]…
- How to specify the correct mask? Suppose I have 8 (logical) CPUs. The i^th bit of mask is 1 if I want to use CPU i
- E.g. if I want to use CPUs 0 and 4, the mask will be 0001 0001. Best expressed as a hexadecimal number 0x11
- What happens to performance if
  - 4 threads use four different physical cores?
  - 4 threads use only one logical core?
  - 4 threads use two logical cores, but they are on different physical cores?
  - 4 threads use two logical cores on the same physical core?

Answers (with #threads = 4)

- Running on one logical core is similar to just 1 thread running
- Running on two logical cores which are on separate physical cores is similar to just 2 threads running
- Running on two logical cores sharing the same physical core is somewhat better than just 1 thread running

Experiment	taskset mask	Time taken (s)
Just one logical core	0x1	48.1
Two logical cores on separate physical cores	0x3	26.7
Two logical cores on the same physical core	0x11	42.1
No constraints (Linux will assign the	–	16.3

What happens if we use even more threads?



Performance on my laptop. After 8 threads, performance decreases because threads have to share the same cores

Breaking down the process

The following stats are relevant:

- ***t_find_all_files***
  Time to traverse the directory structure and collect all .h and .c files
- ***t_process***
  For each thread, time to process its share of files
- ***t_merge***
  For each thread, time to merge the local results into the global map
- ***n_files_processed***
  For each thread, number of files processed
- ***n_bytes_processed***
  For each thread, the total Bytes processed
- ***t_sort***
  Time to sort the results by frequency of occurrence, breaking ties alphabetically

Can you guess the time breakdown with a single thread?

- Breakdown total time taken (~47s) as the sum,

  47s = t_find_all_files + t_process + t_merge + t_sort

Can you guess the time breakdown with a single thread?

- Breakdown total time taken (~47s) as the sum,

  47s = t_find_all_files + t_process + t_merge + t_sort

- Solution

  46.6s = 0.117s + 43.149s + 1.5s + 1.5s

This already rules out optimization ideas for <u>steps that run sequentially</u>:

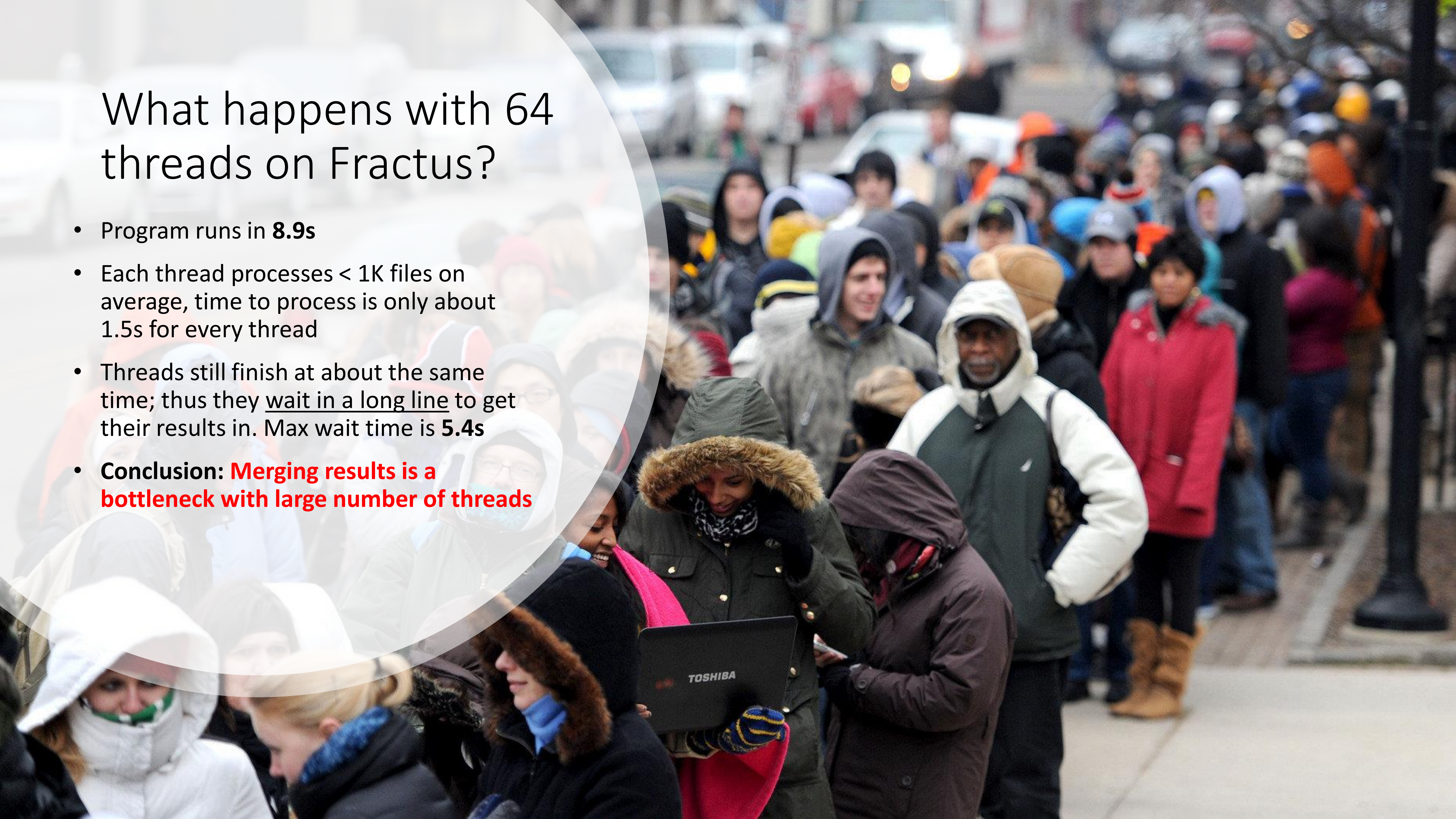
- Time to find files is negligible. No need to worry about pipelining finding files and processing them
- Time to sort is not that high. For small number of threads, this will not be a worry.

Stats with 1-4 threads

#threads	t_find_all_files (s)	t_process (s)			t_merge (s)			t_sort (s)	n_files_processed (K)			n_bytes_processed (MB)		
1	0.1	43.2			1.5			1.5	50.1			797.4		
2	0.1	23.2	23.2		1.0	1.8		1.5	25.2	24.9		402.6	394.8	
3	0.1	15.9	15.9	15.9	1.9	1.3	0.7	1.5	16.8	16.6	16.6	265.5	266.7	265.3
4	0.1	13.2	13.2		0.6	2.3		1.6	12.6	12.6		196.7	201.3	
		13.2	13.2		1.2	1.8			12.2	12.7		200.84	198.55	

Observations:

- Thanks to my dynamic thread to file assignment scheme, all threads finish processing at about the same time. The number of files and bytes processed per thread remains about the same across threads too.
- Merging thread-local results into the global map is in a critical section. Because all threads finish at about the same time, they contend with each other for the merge.
- **Conclusion: High processing time is a bottleneck with small number of threads**

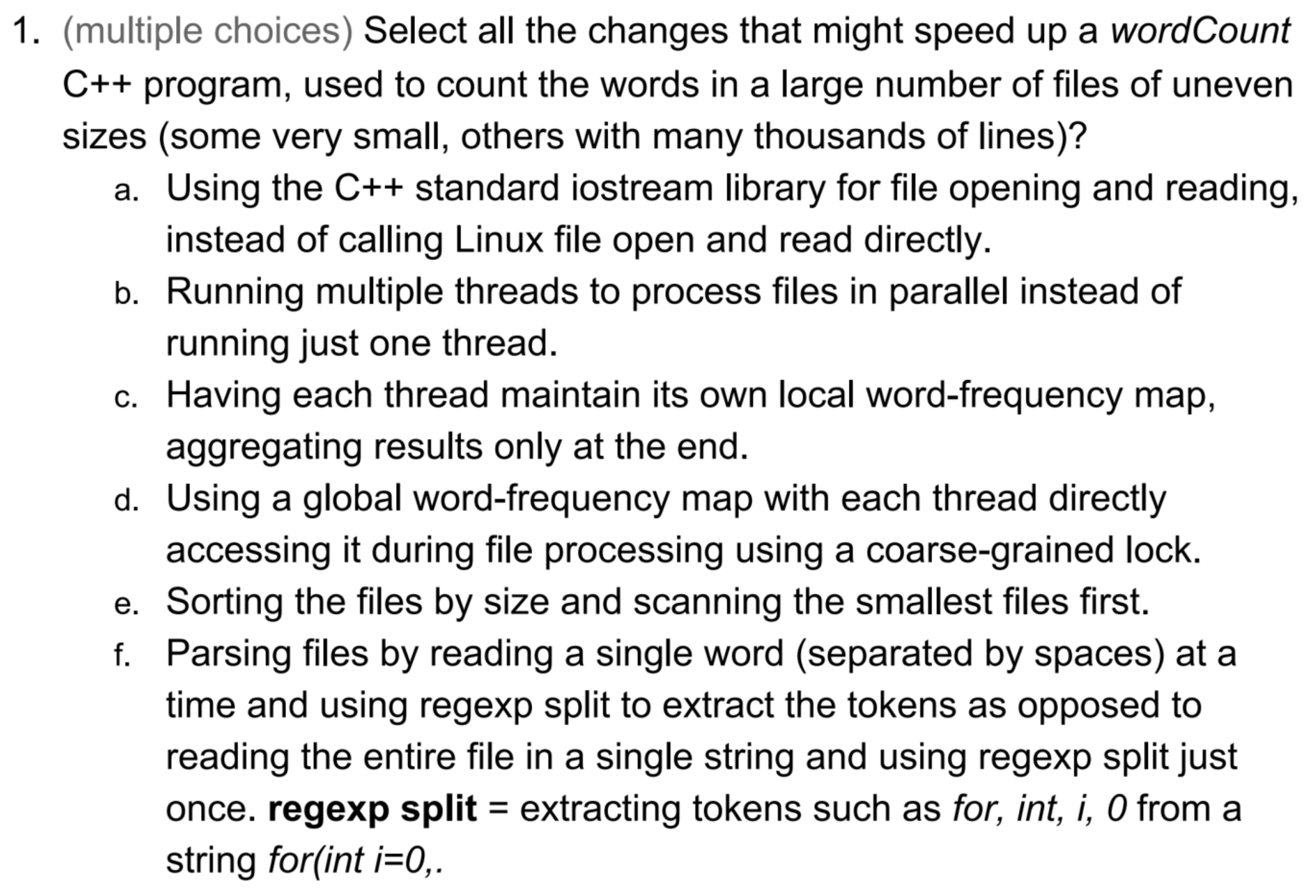


What happens with 64 threads on Fractus?

- Program runs in **8.9s**
- Each thread processes < 1K files on average, time to process is only about 1.5s for every thread
- Threads still finish at about the same time; thus they wait in a long line to get their results in. Max wait time is **5.4s**
- **Conclusion: Merging results is a bottleneck with large number of threads**

Evaluating alternate design choices

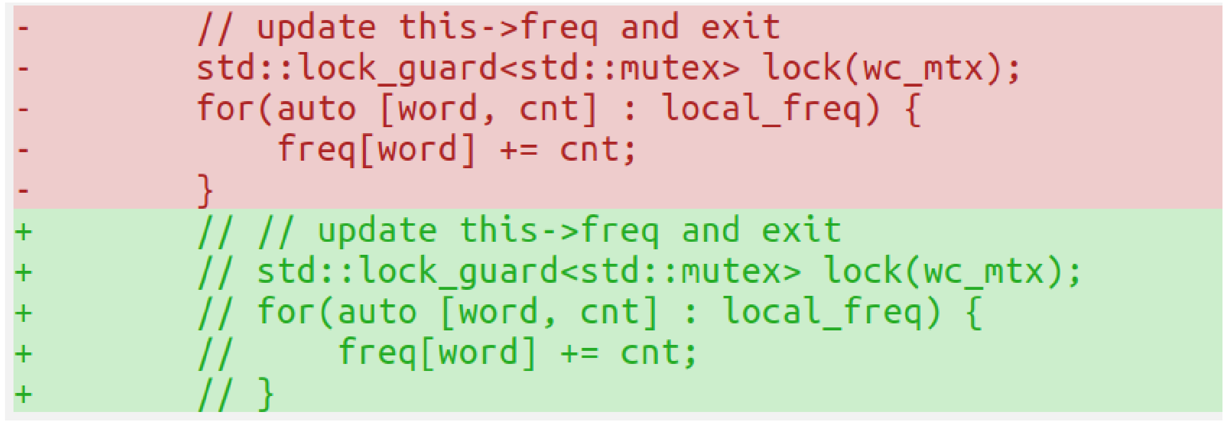
- Remember this quiz question?
- a. and e. are easy to dismiss, b. is an obvious choice
- c. and d. are opposite. Avoiding synchronization overheads through c. is crucial. Let's evaluate d.
- f. is the worst choice you could make! Let's evaluate f.



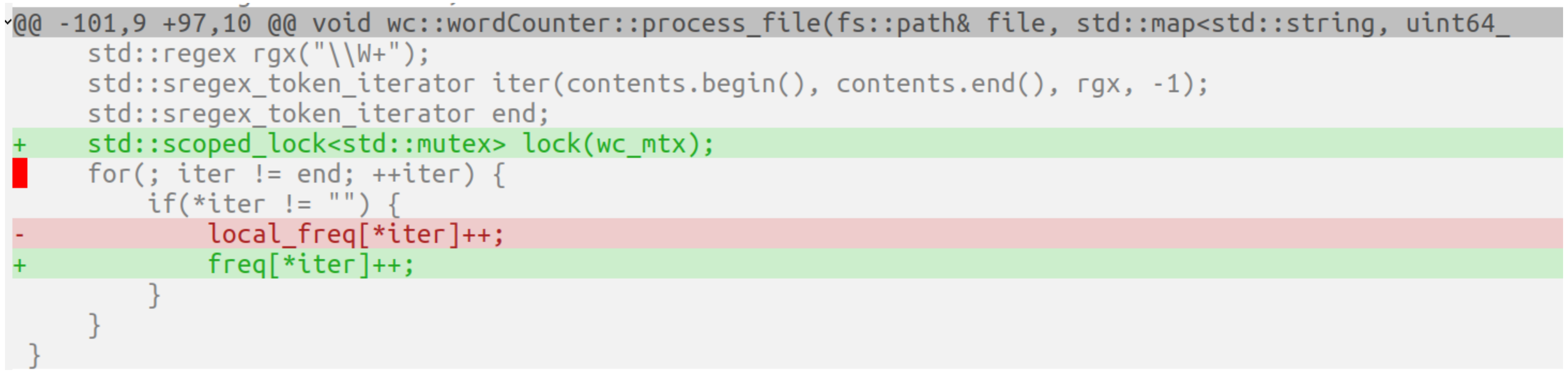
1. (multiple choices) Select all the changes that might speed up a *wordCount* C++ program, used to count the words in a large number of files of uneven sizes (some very small, others with many thousands of lines)?
   a. Using the C++ standard iostream library for file opening and reading, instead of calling Linux file open and read directly.
   b. Running multiple threads to process files in parallel instead of running just one thread.
   c. Having each thread maintain its own local word-frequency map, aggregating results only at the end.
   d. Using a global word-frequency map with each thread directly accessing it during file processing using a coarse-grained lock.
   e. Sorting the files by size and scanning the smallest files first.
   f. Parsing files by reading a single word (separated by spaces) at a time and using regexp split to extract the tokens as opposed to reading the entire file in a single string and using regexp split just once. **regexp split** = extracting tokens such as *for, int, i, 0* from a string *for(int i=0,*.

Directly modifying the global frequency map

- In process_file, we directly modify global freq in a critical section
- We don’t need the merge step anymore



```
-         // update this->freq and exit
-         std::lock_guard<std::mutex> lock(wc_mtx);
-         for(auto [word, cnt] : local_freq) {
-             freq[word] += cnt;
-         }
+         // // update this->freq and exit
+         // std::lock_guard<std::mutex> lock(wc_mtx);
+         // for(auto [word, cnt] : local_freq) {
+         //     freq[word] += cnt;
+         // }
```



```
@@ -101,9 +97,10 @@ void wc::wordCounter::process_file(fs::path& file, std::map<std::string, uint64_
     std::regex rgx("\\W+");
     std::sregex_token_iterator iter(contents.begin(), contents.end(), rgx, -1);
     std::sregex_token_iterator end;
+    std::scoped_lock<std::mutex> lock(wc_mtx);
     for(; iter != end; ++iter) {
         if(*iter != "") {
-            local_freq[*iter]++;
+            freq[*iter]++;
         }
     }
 }
```

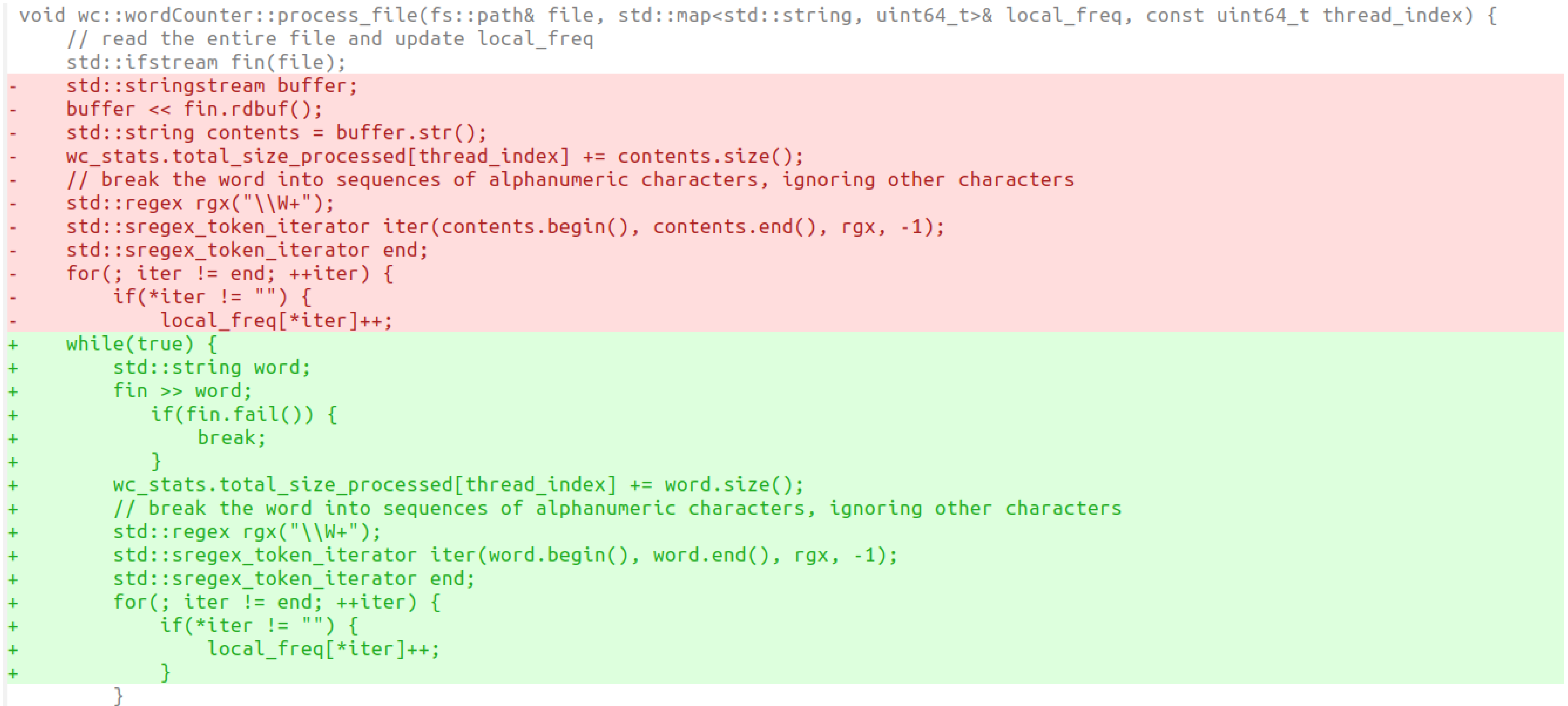
Program performs much worse because of increased contention!

- Time taken increases with the number of threads!
- This is entirely due to the increased processing time

Number of threads	Time (in s)
1	46.3
2	47.5
3	49.0
4	55.1
64 (Fractus)	123.1

Parsing string word by word

- Original implementation reads an entire file in a string
- It then tokenizes the entire string in one shot



```
 void wc::wordCounter::process_file(fs::path& file, std::map<std::string, uint64_t>& local_freq, const uint64_t thread_index) {
     // read the entire file and update local_freq
     std::ifstream fin(file);
-    std::stringstream buffer;
-    buffer << fin.rdbuf();
-    std::string contents = buffer.str();
-    wc_stats.total_size_processed[thread_index] += contents.size();
-    // break the word into sequences of alphanumeric characters, ignoring other characters
-    std::regex rgx("\\W+");
-    std::sregex_token_iterator iter(contents.begin(), contents.end(), rgx, -1);
-    std::sregex_token_iterator end;
-    for(; iter != end; ++iter) {
-        if(*iter != "") {
-            local_freq[*iter]++;
+    while(true) {
+        std::string word;
+        fin >> word;
+           if(fin.fail()) {
+               break;
+           }
+        wc_stats.total_size_processed[thread_index] += word.size();
+        // break the word into sequences of alphanumeric characters, ignoring other characters
+        std::regex rgx("\\W+");
+        std::sregex_token_iterator iter(word.begin(), word.end(), rgx, -1);
+        std::sregex_token_iterator end;
+        for(; iter != end; ++iter) {
+            if(*iter != "") {
+                local_freq[*iter]++;
+            }
         }
```

This decision impacts performance the most

- Time taken goes through the roof – almost 30 minutes with 1 thread!
- Again, the processing time increases disproportionately
- What goes wrong?
  (possibly) Reading a file takes longer and processing a single word has constant overheads

Number of threads	Time (in minutes)
1	29.7
2	15.3
3	10.7
4	8.6
64 (Fractus)	12.75

Bonus: Static thread to file assignment

- Thread ***i*** processes files indexed ***n * t + i*** for all t, where n is the number of threads
- Threads don’t need to coordinate using an std::atomic<uint64_t> to find unique file indexes
- But the solution is less robust to delays – what if a thread runs slower than others (artifact of scheduling decisions) or it has to process significantly more data due to disparity in file sizes?
- For our test-case, results are marginally better for small number of threads because of the very large number of small files and decreased synchronization requirements
- For 64 threads, threads finishing processing at slightly different times results in reduced contention for merging results, giving a 5% improvement

Static partitioning gives comparable results

- The table does not account for variance in results
- I would not trade small performance-gains for a less robust solution
- The original solution can be improved to have better characteristics

#threads	Time (s)	Original Time(s)
1	47.1	47.4
2	26.5	26.7
3	19.2	19.7
4	15.8	16.3
64 (Fractus)	8.5	8.9

How can we improve performance?

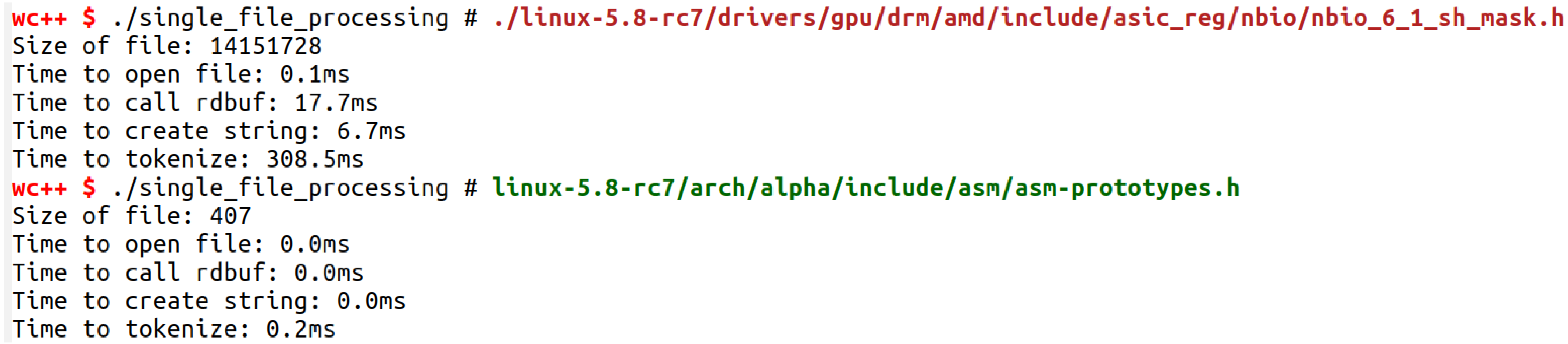
- Two of the three alternate choices are downright bad
- Static partitioning scheme is less robust to delays

Recall the bottlenecks we identified earlier:

- High processing time is a bottleneck with small number of threads
- Merging results is a bottleneck with large number of threads

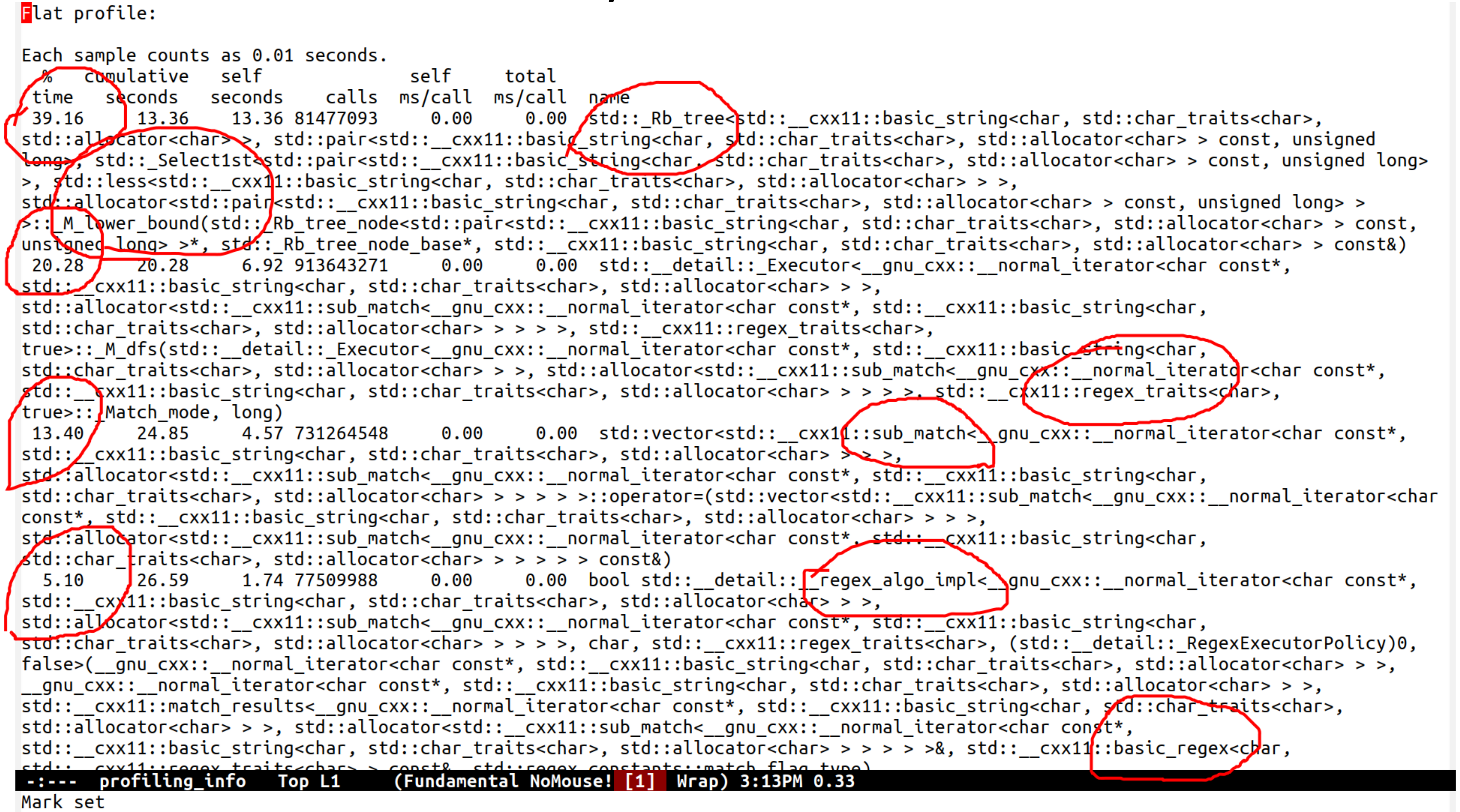
How can we process files faster?

- I profiled the file processing function in a separate program
- Found that regex split is the most expensive step
- This may be one reason Ken's C program is faster



```
wc++ $ ./single_file_processing # ./linux-5.8-rc7/drivers/gpu/drm/amd/include/asic_reg/nbio/nbio_6_1_sh_mask.h
Size of file: 14151728
Time to open file: 0.1ms
Time to call rdbuf: 17.7ms
Time to create string: 6.7ms
Time to tokenize: 308.5ms
wc++ $ ./single_file_processing # linux-5.8-rc7/arch/alpha/include/asm/asm-prototypes.h
Size of file: 407
Time to open file: 0.0ms
Time to call rdbuf: 0.0ms
Time to create string: 0.0ms
Time to tokenize: 0.2ms
```

gprof output supports the theory (-pg –O1, num threads = 1)



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 39.16     13.36    13.36 81477093     0.00     0.00  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >, std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const, unsigned
long>, std::_Select1st<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const, unsigned long>
>, std::less<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >,
std::allocator<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const, unsigned long> >
>::_M_lower_bound(std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const,
unsigned long> >*, std::_Rb_tree_node_base*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
 20.28     20.28     6.92 913643271     0.00     0.00  std::__detail::_Executor<__gnu_cxx::__normal_iterator<char const*,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >,
std::allocator<std::__cxx11::sub_match<__gnu_cxx::__normal_iterator<char const*, std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > > > > >, std::__cxx11::regex_traits<char>,
true>::_M_dfs(std::__detail::_Executor<__gnu_cxx::__normal_iterator<char const*, std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > >, std::allocator<std::__cxx11::sub_match<__gnu_cxx::__normal_iterator<char const*,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > > > >, std::__cxx11::regex_traits<char>,
true>::_Match_mode, long)
 13.40     24.85     4.57 731264548     0.00     0.00  std::vector<std::__cxx11::sub_match<__gnu_cxx::__normal_iterator<char const*,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >,
std::allocator<std::__cxx11::sub_match<__gnu_cxx::__normal_iterator<char const*, std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > > > > >::operator=(std::vector<std::__cxx11::sub_match<__gnu_cxx::__normal_iterator<char
const*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >,
std::allocator<std::__cxx11::sub_match<__gnu_cxx::__normal_iterator<char const*, std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > > > > > const&)
  5.10     26.59     1.74 77509988     0.00     0.00  bool std::__detail::__regex_algo_impl<__gnu_cxx::__normal_iterator<char const*,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >,
std::allocator<std::__cxx11::sub_match<__gnu_cxx::__normal_iterator<char const*, std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > > > > >, char, std::__cxx11::regex_traits<char>, (std::__detail::_RegexExecutorPolicy)0,
false>(__gnu_cxx::__normal_iterator<char const*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >,
__gnu_cxx::__normal_iterator<char const*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >,
std::__cxx11::match_results<__gnu_cxx::__normal_iterator<char const*, std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > >, std::allocator<std::__cxx11::sub_match<__gnu_cxx::__normal_iterator<char const*,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > > > >&, std::__cxx11::basic_regex<char,
std::__cxx11::regex_traits<char> > const&, std::regex_constants::match_flag_type)
-:---  profiling_info   Top L1     (Fundamental NoMouse! [1] Wrap) 3:13PM 0.33
Mark set
```

Idea 1: Let’s batch file processing

- Read contents from multiple files into a single string until it is large enough (*max_size*). Only then tokenize.
- Regexp split is linear in the size of the string. Processing large strings does not provide any clear improvement.
- With max_size ranging from 1KB to 50MB, performance with a single thread remains somewhat constant at about 48-50s.
- Does not work therefore ☹

How can we merge faster?

- Idea 2: Ken’s idea of parallel merge!
- I didn’t try it, but it should give speedups with large #threads
- If you are interested, extend my solution to support parallel merge

Summary

- Increasing the number of threads does not automatically increase performance
- Threads can work in parallel as long as they run on separate cores
- The more the threads need to synchronize, the less they can work in parallel
- Steps that run sequentially can bite us (sequential vs. parallel merge)
- String processing is a pain
- Measuring time/performance statistics can guide us in understanding and improving performance