ECE 459: Programming for Performance
Lab 2—Building Dictionaries (in parallel)[1]

Patrick Lam
Due: February 12, 2024 at 23:59 Eastern Time

Learning Objectives:

- Become familiar with safely and quickly passing information between threads.

The most related lecture content for this lab is in Lectures 9 and 10, though there is somewhat-relevant content in appendix C and lecture 16.

Background

Log files can be a key source of data about long-running executables like web servers and operating system kernels. Here's a sample line from a log file.

```
58717 2185 boot_cmd new 1076865186 1 Targeting domains:node-D1 and
   nodes:node-[40-63] child of command 2176
```

Analyzing log files can help you troubleshoot undesired behaviour (both performance and crash related). Unfortunately, log files get huge (tens of gigabytes!). Really, you need to use tools to have any hope of deriving meaning from sufficiently large log files. Even more unfortunately, it seems like every piece of software uses its own log format. And, regardless of format, different software is going to log different information using different messages. So, how to write software that analyzes log files? Well, you could write a set of regular expressions to capture relevant data for each program. But that's a lot of work.

[**?**] propose a mechanical way to extract structured data from unstructured logs. The core part of the approach is building dictionaries of so-called n-grams, or tuples of tokens, and identifying low-frequency n-grams as dynamic parts of log messages. Before reading further you should read the reference provided and make sure you understand at least Sections 2 and 4. Ask on Piazza if you do not. The rest of this assignment document is written with the understanding that you have read the paper.

I've provided a sequential sequential implementation. Your task in this Programming for Performance assignment is thus to speed up the dictionary construction. You'll modify the starter code to use concurrency to read log files and build dictionaries of n-grams. Once you've built the dictionaries, you can use them to identify that tokens `nodes:node-[40-63]` and `2176` in the example above are dynamic, and the rest of the line is static.

[1] v0, 19Jan23

Running the code

The implementation takes a number of arguments. The most important ones are a log file (which is the source for the dictionaries), a declaration of its format, and a line to analyze for dynamic versus static tokens.

```
cargo run --release -- --raw-hpc data/HPC.log \
  --to-parse "58717 2185 boot_cmd new 1076865186 1 Targeting domains:node-D1 and nodes:node-[40-63] child of command 2176" \
  --before-line "58728 2187 boot_cmd new 1076865197 1 Targeting domains:node-D2 and nodes:node-[72-95] child of command 2177" \
  --after-line "58707 2184 boot_cmd new 1076865175 1 Targeting domains:node-D0 and nodes:node-[0-7] child of command 2175" \
  --cutoff 106
```

Here, we specify that the raw log file is in format HPC (`-raw-hpc`) and located in `data/HPC.log`. You'll find a number of other log files in the `data` subdirectory. In case you're curious about experimenting with even more logs, you can find a bunch at https://github.com/logpai/loghub.. The list of formats more-or-less supported is: linux, openstack, spark, hdfs, hpc, proxifier, healthapp.

The `-to-parse` argument specifies the line to be parsed. The logram approach also uses the previous and subsequent lines; if available, you can specify them with `-before-line` and `-after-line`. You can also specify just the two last and two first tokens with `-before` and `-after`.

Logram identifies low-frequency ngrams as likely dynamic tokens. The `-cutoff` argument is the cut-off for dynamic versus static tokens.

I've put in a `-num-threads` option which specifies how many threads to start. Since the starter code is sequential, that option doesn't do anything yet. Your task is to make it do something.

The `README.md` file contains a number of other invocations of the code. logram is more effective on some log types than others. You can run your code on the longer-running examples and see how fast you've made it.

Understanding the code

There's less going on here than in the Lab 1 code, but there is enough so that you can get practice with using concurrency, getting speedups, and avoiding race conditions.

We haven't talked about how to do profiling yet so I can't ask you to find what the bottleneck is. So, the bottleneck is `dictionary_builder` in `packages/parser.rs`. There's a lot of other code; if you read the paper you can understand what it does.

This function sequentially iterates over the lines in the log file and hands them off to the function `process_dictionary_builder_line`. What's important about that function is that it updates entries in its two `HashMaps`: `dbl` and `trpl` (for 2-grams and 3-grams respectively).

Your task

Since this is Programming for Performance, your task is to make the code faster. The strategy will be to have multiple threads, each of which is responsible for analyzing a part of the log file (in parallel).

1. For this assignment, you can load the entire log file into memory at once so that you can give each thread a different segment of the log file. (Yes, that wouldn't work for gargantuan log files; to scale up more I would use a different approach.)
2. Start a number of threads. Each thread processes its segment of the log file. First, give each thread a separate hash map, and merge the maps at the end. Evaluate this solution and write about it in the separate-maps commit log. Keep this implementation available under the `-single-map` command-line option (`args.single_map` will be `Some(true)` when the option is specified). We do not expect this implementation to be faster than the original.
3. Finally, look around on the Internet for crates that implement concurrent hash maps, and use a single concurrent hash map for `dbl` and one for `trpl`. Evaluate the performance of that—we are expecting it to be faster than the original, but we are not setting a specific performance target. Use this implementation when `-single-map` is not specified.

   The first Google result for me is `chashmap`, but note that it is not under active maintenance. We suggest finding a concurrent hashmap crate that is being actively maintained.

Have fun!

Rubric

The general principle is that correct solutions earn full marks. However, it is your responsibility to demonstrate to the TA that your solution is correct. Well-designed, clean solutions are therefore more likely to be recognized as correct.

Solutions that do not compile will earn at most 39% of the available marks for that part. Segfaulting or otherwise crashing solutions earn at most 49%. We are going to run your code on some testcases to make sure it works and look at the changes to see that you used concurrent hash maps.

Implementation (85 marks) Your code must preserve the original behaviour and must use multiple threads. The separate-maps implementation is worth 40 points and the concurrent-maps implementation is worth 45 points.

Commit Log (15 marks) 12 marks for explaining the changes that you've made. 3 marks for clarity of exposition.

What goes in a commit log

Here's a suggested structure for your commit log message justifying the pull request.

- Pull Request title explaining the change (less than 80 characters)
- Summary (about 1 paragraph): brief, high-level description of the work done.
- Tech details (up to 3 paragraphs): anything interesting or noteworthy for the reviewer about how the work is done.
- Something about how you tested this code for correctness and know it works (about 1 paragraph).
- Something about how you tested this code for performance and know it is faster (about 1 paragraph).

Write logs in files `commit-log/separate-maps.md` and `commit-log/concurrent-hashmap.md`.

Clarifications

I'm sure I'll have some. But I don't have any yet.