ECE 459: Programming for Performance
Lab 4 — Profiling [1]

Created by Jeff Zarnett, Stephen Li, & Yuhan Lin, updated by Bernie Roehl & Patrick Lam
Due: March 31, 2025 at 23:59 Eastern Time

One of the keys to improving performance is identifying bottlenecks that are slowing a program down. The key to doing that is the use of profiling tools. In this assignment, you are provided a simulation of a Hackathon. The program is slow, and you will use some profiling tools to analyze the code to find out where it's spending its time. Based on what you find, you can make changes to the program to speed it up.

Request. Before we get started, a small request: please make sure you don't leave leftover processes hanging around. This was an issue in a previous year near the deadline. You can use `ps` and `killall` on your processes.

Background

The Hackathon has three different kinds of threads: Students, Idea Generators, and Package Downloaders. The Idea Generator generates an idea the way many startup companies are pitched, as an equivalent of a known service for a new audience, such as "LinkedIn for Service Animals". Students work on ideas, and working on an idea requires downloading some number of software Packages. The Hackathon simulation runs until all ideas have been built.

Since we have multiple threads, the order of outputs may differ. However, we can still check for correctness by taking the `xor` (exclusive or) of the SHA256 hashes of every idea generated and every package downloaded.

Analyzing and Optimizing the Program

Your goal is to speed up the program. You should therefore do some analysis using profiling tools. You may have some ideas about where to start immediately, but tools help you check your assumptions and find places for improvement. The basic workflow is to use profiling tools to identify what's slow, make changes, and re-evaluate to see how much (or how little) it improved the runtime of your program. If the program is sped up enough, you're all done (and can go on to writing your commit message).

You can use any analysis tools you like, whether or not they were presented in the lectures. However, for the purposes of your commit log message you should use a flamegraph. A flamegraph is a visualization of profiling data, meant to show you where a program is spending most of its time. And they look cool, so there's that.

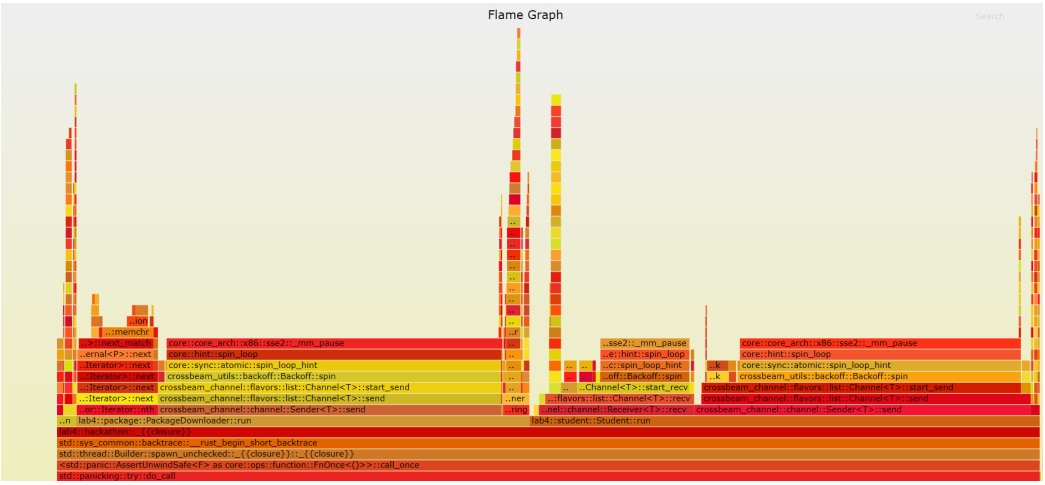
You probably need to know about how flamegraphs work. To read up on them, see `http://www.brendangregg.com/flamegraphs.html`. There are some useful YouTube videos linked there to give you guidance. In particular, the video at `https://youtu.be/D53T1Ejig1Q` is useful. If you're in a rush, just watch from the 10 minute mark to about the 16 minute mark.

For your convenience, we've created a Makefile target that runs your program and generates the flamegraph. You'll have to run `cargo install flamegraph` first. Then, you can create a flamegraph by typing "make". This will generate an SVG (Scalable Vector Graphics) file called "flamegraph.svg" containing the graph, which can be viewed in any graphical viewing program or in your preferred web browser. We believe that you can create flamegraphs on any of the eceubuntu and ecetesla machines.

Tip. Want to view SVGs in VS Code? I hear that `cssho.vscode-svgviewer` is helpful.

[1] v1, 14Mar23

Figure 1: A flamegraph of the starter code



In your commit log message, explain how you used your profiling data to decide what to change. Be sure to include the flamegraph for the final version of the program (if not standard flamegraph generation, say how you generated it) and provide some explanation to the reader as to how you used the flamegraph to improve the starter code. Also let us know which machine you measured your results on (it should be one of the ECE machines so that we can reproduce it). (Hint: ecetesla2 doesn't have many threads available!)

Rules/Restrictions. To prevent trivializing the problem, and to make it possible for us to compare your code against the starter code, here is a list of rules:

- You may change how threads communicate.
- You may change (or remove) output of intermediate checksums, but the checksums printed out at the end of execution must be identical to the ones generated by the starter code.
- You can move IO into main.
- You may not change the numbers of student, idea, and package downloader threads (but you can add helper threads).
- You can change the provided structs and add, modify, or remove members and their visibility. You can change method signatures as well, e.g. by adding more parameters.
- The arguments of the program must still be respected (so you can't change the the number of students/ideas/-packages managed by each thread), and you must read the data at runtime.
- You can change the lock structure of the program. But don't, for instance, create a single thread to handle checksum updates. Checksum updates stand in for download time.
- Must not introduce bugs (memory leaks, deadlocks, etc.).
- Must not trivialize the threads e.g. moving everything to 1 thread to eliminate synchronization. You may rewrite how the threads communicate (different mutex/constructs usage, for example).
- All publicly observable final behaviour/output for each thread must be preserved. You may change intermediate prints (I/O) but not final ones. For example, IdeaGenerator must add `NewIdea` Events to the queue. It must also be responsible for inserting the poison pills (`OutOfIdeas`) for the Student threads.
- Any threads you create may only be terminated with the `OutOfIdeas` event (no passing "expected number of ideas").

- You must not hardcode any values that are read from files. We will be testing with different files of various sizes.
- You can and we encourage you to use any crates that don't trivialize the problem. For instance, you can use `lazy_static` and `once_cell`. You can use Rayon, though you aren't required to, and it makes the flamegraph harder to read.
- You can change the makefile, e.g. to add new targets or change the build flags.
- No other changes that trivialize the simulation or execution (e.g. deleting functionality that consumes CPU time).

Rubric

We're continuing to require a commit log message instead of a report. Commit messages should still argue for why your change should be merged (including the flamegraph) but imply less boilerplate.

Implementation (35 marks) Your code must preserve the original behaviour and cannot violate any of the rules specified above. You should make changes, and the changes you make should be supported by a profiling tool of some sort—they can't just be random changes.

Commit Log (15 marks) 12 marks for explaining the changes that you've made, including your expected speedup results, final flamegraph, and some explanation for the reader. 3 marks for clarity of exposition.

Performance (50 marks) For this lab, we are finally measuring performance (with hyperfine). Your code must run faster than the starter code. The faster, the better. Marking guidelines:

Speedup ($\times$)	(% of marks for performance)
≥ 13.5	100
$[12, 13.5)$	90
$[11, 12)$	80
$[10, 11)$	70
$[9, 10)$	60
$[8, 9)$	50
$[7, 8)$	40
$[6, 7)$	30
$[5, 6)$	20
$[4, 5)$	10
< 4	0

We are going to try to run your code on unloaded systems, but if the TA says that they get a different time than you do, please contact the TA.

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 (3–5 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 3 paragraphs, referring to the flamegraph as appropriate).

Write your message in the file `commit-log/message.md` and commit the final generated flamegraph as the file `commit-log/flamegraph.svg`. Don't forget to `git add commit-log/flamegraph.svg`.

Clarifications

Helgrind complains, even on the unmodified code. Good that you're running Helgrind. Yep, you're not required to fix this.

Which times do you care about? Hyperfine mean time, taking range and deviations into account.

Is a shared memory queue OK instead of crossbeam? Yes.

I don't know, man. [unknown] is 50% of my flamegraph. Set `num_pkgs` to something big, like 100,000.

Just one commit message? If you want, you can cast your changes as multiple unrelated changes, but you can also present them as a single change. Probably easier for the TAs as a single change.

I don't like Events. You can restructure the program to get rid of the `Event` type if you feel that is appropriate, e.g. you want to change the channel structure. (I'm not sure this helps!)

5 isn't enough: I want to write more than 5 paragraphs in the commit log. Yeah, just don't go overboard, but do what makes sense.

I don't like 0s. We aren't going to test your program with any arguments set to 0.

Oh Nondeterminism! no! You may assume that the number of students $>=$ number of idea generators. It should be determinstic then. Also, you can assume number of ideas $>$ number of idea generators.

A Blooper to Avoid. Be sure to use `hyperfine -i "target/release/lab4"` and not `hyperfine -i "cargo run -release"`.

I don't like doing unnecessary work. Me neither. But please don't remove calls to `hex::decode` and `hex::encode` in `Checksum::update`. They represent download time.

How many packages? This is set using a command-line option.

What testscases are you going to use for performance testing? We'll use a variety of test cases. The one in the Makefile is one of the harder ones. We've found increasing the idea count, the pkg gen count, and the student count to large numbers to also be difficult.

What prints do I need to keep? Everything from "Global checksums" to the end. Do not rename Global checksums.

Do I make a pull request for you to mark? No, please merge everything to main.