ECE 459: Programming for Performance
Assignment 4

Jeff Zarnett, Stephen Li (Part 1) & David Cheng (Part 2)

March 15, 2019 (Due: April 5, 2019)

This assignment is choose-your-own-adventure, to a limited extent. You may choose to do *either* Option 1: Profiling, or Option 2: Load Balancing. We will mark whichever one you choose to do, but not both. You do not need to declare or announce in any way what option you plan to do; just submit the normal way and we'll figure out from your report which option you chose. And before you ask—there are no bonus marks for doing both.

1 Profiling

In this assignment, you are provided a simulation of a Hackathon. The program is slow, however, and you will use some profiling tools to analyze the program. Based on what you find, you can make changes to the program to speed it up.

To give you an overview of the program, the Hackathon has three different kinds of threads: students, idea generators, and package downloaders. The idea 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 of the SHA256 hashes of every idea generated and package downloaded.

Getting started. Get your starter code from `git.uwaterloo.ca`.

To submit, push the C++ files containing your code, along with your report.

1.1 Analyzing and Changing the Program

You need to speed up the program. You should accordingly 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: 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 done (and can go on to writing your report).

When you want to analyze the program, you can use any analysis tools you like, whether or not they were presented in lecture. However, for the purposes of your report you should use a Flamegraph. A flamegraph is a visualization of profile data, meant so show you efficiently 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.

A makefile target has been created that runs the program and generates the Flamegraph, for your convenience. Use the command `make fg_fast` or `make fg_slow` to create them for your program (with and without optimizations). These generate a svg (scalable vector graphics) image called `fg_fast.svg` and `fg_slow.svg` respectively. These can be viewed in any graphical viewing program or your preferred web browser.

In your report, explain how you used your profiling data to decide what to change. Be sure to include the flame graph for the final version of the program and provide some explanation to the reader as to how your updated Flamegraph demonstrates improvement over the starter code.

Restrictions. To prevent trivializing the problem, and making it so we can compare your code against the baseline, here is a list of restrictions:

- `main.cpp` cannot be modified.
- Cannot introduce bugs (memory leaks, deadlocks, etc.).
- Checksums at end of execution (before/after optimizations) must match.
- Cannot use STL libraries such as vector/list/map/etc. (but you are allowed to replace `Container` class with your own data implementation).
- Cannot trivialize the threads e.g. moving everything to 1 thread to eliminate synchronization. You can rewrite how the threads communicate (different mutex/constructs usage, for example).
- All public observable behaviour/output for each thread must be preserved. For example, `IdeaGenerator` must add `NEW_IDEA` Events to the queue. It must also be responsible for inserting the poison pills for the `Student` threads.
- `Student` threads may only be terminated with the poison pill (i.e. you cannot pass an “expected number of ideas" to `Student`).
- Do not hardcode any values read from files. We will be testing with different files of various sizes.
- No other changes that trivialize the simulation or execution (e.g., deleting functionality or things that consume CPU time).

1.2 Evaluation

Implementation (35 marks) Your code must preserve the original behaviour and can’t break any of the restrictions specified above. The changes you make should be supported by a profiling tool of some sort and not just random changes.

Report (15 marks) 12 marks for explaining the changes that you’ve made, including your final Flamegraph and some explanation for the reader; 3 marks for clarity of exposition.

- **Sample Solution Speedup:** 24
- **Minimum Speedup:** 20

2 Load Balancing

In this assignment, you will be asked to simulate a multi-server application to investigate the impact of Load Balancing on the system's overall performance. The goal here is both to implement load balancing, and also to make some determinations about what impact load balancing has (big or small, positive or negative) based on the parameters of the system.

A basic overview of the program as provided is: some number of jobs will arrive periodically (according to a Poisson distribution) to the system and they will be assigned to one of N queues either by (a) random assignment or (b) round-robin (i.e., job i is assigned to queue i modulo N). Each queue will have an associated `pthread` that will dequeue the first job, if any is available, and dispatch it to a processing function. When processing is complete, it outputs some timing data to the results file in a specified format, giving us some information about how long it takes to complete the processing of the task.

Getting started. Get your starter code from `git.uwaterloo.ca`.

To submit, push the C file containing your code, along with your report. The report will explain your load balancing strategy, and discuss when load balancing is effective and when it is not.

This program takes a lot of command line arguments so you have lots of space to experiment with the program and evaluate it. The parameters are:

Character	Meaning
`n`	Number of "servers" (queues)
`a`	Assignment policy (1 for random, 2 for round robin)
`j`	Number of jobs
`b`	Load balancing (0 for off, 1 for on)
`l`	Lambda (arrival rate parameter)
`m`	Max Rounds (controls variability of job size)

The output file will be comma separated values ("results.csv"), with one line per completed job. The column order:

1. Job ID
2. Arrival time (the time at which the job enters the queue)
3. Execution time (the amount of time the job took in the `execute` function)
4. Departure time (when the job's execution is finished)
5. Response time (departure time minus arrival time)

2.1 Investigating the Impact

The implementation side of load balancing is writing the functionality into the program. A high level explanation is that you need to migrate jobs from over-loaded queues to under-loaded ones. This keeps things "balanced" within some margin. You should (probably) run the load balancing technique periodically. How often you wish to run it is a design decision: if you run it too often, you'll waste a lot of time shuffling jobs around; not often enough and you do not see enough benefit.

Each queue uses the First-Come First-Served (FCFS) policy (i.e., all work is considered to be of equal priority).

One strategy for implementation follows, though you are welcome (encouraged) to come up with your own. At each period, measure the average length of the queues, where the length of each queue is the total number of jobs currently waiting in it. If the length of a particular queue is higher than the average, then it is considered as

an over-loaded queue and as such you need to migrate some jobs from the end of that queue to the under-loaded queues of that tier. This process continues till you have balanced queues (all queues are approximately equal in length, depending on how many jobs there are).

List of things I don't want you to do: (1) have a single queue for incoming jobs (as this trivializes the load balancing), (2) have > 1 thread per queue (as this can mean that threads sit with the work items and get blocked, meaning load balancing does nothing).

A naive implementation may have an infinite loop, frequently moving the same job between the queues over and over. This is something to avoid...

This last column of the output file is important; the real goal is to minimize the **average 90th percentile response time** of all jobs (through load balancing). You will want to modify the provided `loadbalance.c` to implement the load balancing and do it efficiently. Then it is time for the report. There's even a helpful python script to do some of the calculation for you.

To write the report, there are four cases that form the baseline scenarios, shown below:

1. Random assignment, no load balancing
2. Round robin assignment, no load balancing
3. Random assignment, with load balancing
4. Round robin assignment, with load balancing

This covers the variables `a` and `b` in the parameters. Find the average 90th percentile response time of all jobs in each case, and investigate how they are affected by changes in the arrival rate (`l`) and variability of job size (`m`). Choose a reasonable value of `j` for your testing.

The content of the report will then say some things like "load balancing has an increasing positive impact correlated with an increase in `x`...". Consider also the combinations of variables `l` and `m` (i.e, are there any synergistic effects).

Your report should support your findings with some hard (measured) numbers and the use of some graphs to illustrate is recommended. Intuition or logic might say that a change in `x` results in the outcome of `y`, but it must be quantified. A graph that charts the effectiveness of load balancing (compared to baseline) with some increasing or decreasing variable is very convincing to the reader, for example. Tastefully-presented data should make up about half of the report and your discussion and explanations the rest. Avoid chartjunk[1], and don't imprison your numbers[2].

2.2 Evaluation

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. Segfaulting or otherwise crashing solutions earn at most 49%.

Your program will be evaluated over several runs to prevent bad luck on a single run from being the deciding factor. The program should not leak memory, nor should it have data races that result in invalid or incorrect output.

Implementation (15 marks) A correct implementation is important. You may use any techniques to implement the load balancing, as long as it produces the right output, does not trivialize the work, and balances the load.

Report (35 marks) 30 marks for explaining the impact of load balancing (4ish pages is appropriate length); 5 marks for clarity of exposition.

[1] `https://en.wikipedia.org/wiki/Chartjunk`
[2] `http://betterposters.blogspot.ca/2012/08/the-data-prison.html`