ECE 459 W22 Final Exam Solutions

J. Zarnett, P. Lam

April 21, 2022

(1)

(a) The first run may be unrepresentative of actual performance for more than one reason and if your answer is reasonable that's fine. Example: On the first run, the caches will be cold—that is, they don't contain your code or data, thus resulting in more cache misses. On a subsequent run, they will have data, and this is a better match for what happens when the application is running for real.

(b) I verified that it takes about $s = 90$ seconds to toast bread. Numbers on the Internet suggest that it takes about 150 seconds to take an order, giving $\lambda = 1/150$. T_q is $s/(1 - \rho)$, where $\rho = 90/150$, giving $T_q = 117$ seconds. You can't really toast faster, but you can have two toasters going at once. That makes $\rho = 90/300 = 0.3$. We have $\lambda \times s$ still at 0.6 and $K = 10/13$. That gives $C = 3/10$ and $T_q = 99.3$ seconds.

(c) Any problem domain where you can successfully argue is acceptable. One example might be something like fraud detection in purchases; if the evaluation of whether the transaction is fraudulent takes too long then we choose a default action (deny) and proceed on that basis.

(d) The fundamental assumptions about the Poisson distribution still apply: there's a huge number of users, each of which have a small probability of visiting the site at any given moment. The probability of the individual visit may be higher on Black Friday than on a random day of the year, but the model is still valid.

(e) At first glance this seems like a reasonable plan. The write takes CPU instructions, possibly a cache miss, and may result in other CPUs invalidating their caches unnecessarily because of things like false sharing.

You could also argue that dead-store elimination can also be bad, because if you're using it to overwrite things that shouldn't be in memory anymore (passwords or similar once they've been used!) you're actually harming security.

(f) Reasons for overhead: if running locally, displaying to the screen & scrolling takes time; if running over network, you introduce a bunch of networking overhead. And, even if you redirect output to /dev/null, it still has to compute the output, which has overhead.

(g) Inter-thread communication: if you have a big array, you don't really want to pass it between threads if you don't have to; shared memory is likely more efficient.

(h) Bottlenecks: (i) The allocator might have more locking than you need in your context; (ii) You

can try to pool (reuse) allocated objects to reduce calls to the allocator.

(i) AWS:

- level of support for the shiny new AWS thing on the AWS side
- amount of work you'd need to do to migrate your existing code/how well your need is met by the new thing
- cost of the new AWS thing vs what you're paying now

(j) Humour is subjective. If they followed the instructions and clearly made an effort, full marks.

(2) In this question, the goal is to write a good description so that the reader can understand what changes are supposed to be made. That shouldn't include writing a bunch of code, but some pseudocode and some written description seem sensible. There are a few key elements that need to be present. Each should have some explanation for why this approach is chosen.

Parallelism [5 marks]. The solution must explain some details about how it is supposed to be parallelized. This could be by, for example, creating some number of threads in a pool, where each thread will take items from the batch, or get its own batch.

Retry [4 marks]. The solution described must have a retry mechanism for anything that was rejected because of the rate limit, either by scheduling a retry later (after some delay) or making sure that the invoice is not marked as failed and will be selected in a future batch (and it's okay to make assumptions about how the next batch is retrieved, as long as they're explicit).

Self-Adjusting [6 marks]. The solution described also needs to have a mechanism for reducing the parallelism if the rate limit is reached. This could be by reducing the number of threads in the pool, for example.

(3) 5 marks for implementing the letter frequency analysis; 5 for implementing the scoring. Verify in both cases that 4 threads are used and appropriate concurrency-control mechanisms are present (e.g., atomic types, mutex, etc).

You will also want to run the code to verify that the answers match the expected!

(4) I specifically said that efficiency doesn't count here, so don't look at that. As usual, it has to compile and run, and it has to produce the same answer as the reference result (as printed by cargo run cpu on the starter code). "Solutions that do not compile will earn at most 39 % of the available marks for that part. Solutions that compile but crash earn at most 49%."

Apart from that, look at the code. I expect most students to launch the kernels properly; allocate 5 marks for that. Then the `add` kernel is pretty simple; check that the bounds are right. That's 5 marks. And the `find_max_index` kernel (plus any necessary work to combine results at the end, possibly on CPU side) is worth another 15 points. I'd expect most students to Google and find something pretty similar to what I have. If they return the value and not the index, give them 5/15 (because you can find code to return the value on Google but the index requires you to do work). If something is wrong with the initialization part of the `find_max_index` kernel, that's 5 points, and the reduction itself is 10 points.

(5) The targets are `_print` (7.17%), `search` (52.8%), `insert` (24.6%), and `drop` (6.4%). I'd expect students to get slightly different numbers on their own machines. The speedup calculation for e.g. `_print` goes like this: 100s in the original, 92.83s sped up, speedup is $100/92.83 = 10.7\times$.

`search` is clearly the bottleneck here. The issue is that most of the execution time is in a single call to `strsim::jaro_winkler`. If it were multiple calls we could improve that. Or we could see if the library is being used suboptimally. But I don't think it is. So, the challenge is that the slow code is code that we don't control. One way to speed it up is to create your own fork of the library code. Or, you can decide that you can live with some other result, not `jaro_winkler`.

The solution that I implemented was to print out checksums instead of full results. These can be verified but take much less time to output. Anything else that meets the specification is OK, of course.