ECE 459 Winter 2023 Final Assessment

Instructions

General

1. This is a take-home exam and should be treated as such. You are expected to do the exam independently and may not collaborate with other people (classmates or otherwise).
2. This is an open-book exam, so you can consult your notes, the lecture materials, Google, Stack Overflow, man pages, etc.
3. If you take code from a source on the internet, make sure you cite it with a comment including the URL where you found it.
4. We will not answer questions on Piazza or via e-mail; if you need to state an assumption, do so.
5. Submit your PDF with the written questions in Crowdmark, and add and commit your code in gitlab.
6. Be sure to submit your files on time.

Written Questions (Q1, Q2)

1. You can create your PDF using whatever software you like.
2. Answer the questions in the order of the exam, but also make it clear to the reader what question is being answered.
3. Use the amount of marks associated with a question as your guide for how much you should write; we recommend keeping it brief.

Programming Questions (Q3 - Q6)

1. You are allowed to modify the code, packages, and build files as you need, except you cannot change the output formats (for marking consistency).
2. Your code needs to run on `eceubuntu` (or `ecetesla` for CUDA) machines. Code that does not compile will result in a zero for that question.
3. You can use the compiler, code analysis tools, debuggers, etc.
4. If you are having technical issues with the ECE Servers, please e-mail `praetzel@uwaterloo.ca` (course staff are not admins on the machines).
5. As with the assignments, your local machine setup cannot be supported.
6. Please try to distribute your work across servers; `ecetesla` machines are the only ones that can run CUDA problems, so try using `eceubuntu` when the GPU is not needed.
7. Complete the `README.txt` to inform the marker about the server you used for each code question.
8. Please respect the directions to use a maximum of 4 threads; this helps reduce the server load to allow multiple people to work at once.
9. Server resources are limited and extensions will not be granted due to demand issues, so don't procrastinate.

1 Short Answer [30 marks]

Answer these questions using at most three sentences. Each question is worth 3 points.

(a) (Rust.) We've spent lots of time talking about how great Rust is at preventing concurrency problems. Give an example of a concurrency problem that Rust cannot prevent.

(b) (Reduced-resource computation.) One way that we could "do less work" is by ending the voting process as soon as a majority have voted in favour of one option. Give an example of a problem domain where this strategy is reasonable and justify your selections.

(c) (Speedup calculation.) You have a small server and a program that is 21% sequential and 79% parallel. You have some budget to spend. You can either go from 4 CPUs to 6 or you can spend the money on development to reduce the sequential part by 3%. Which do you do?

(d) (Speculation.) In the N-Body problem, we used approximations to trade accuracy for time (but did not always have great results). If we added speculation as to whether an another point is near or far away from the current one, would that help? Justify your answer.

(e) (Devops.) Suppose you have a service that allows people to run some code for free. How would you detect abuse of the system? Your answer should reference some course concepts.

(f) (Order of magnitude estimation, `computers-are-fast.github.io`.) I'm curious how long it takes to lock a `Mutex` in Rust. Write down a simple `main` function that would allow you to test this out using `hyperfine`.

(g) (Queueing Theory.) In queueing theory, Tim's approach for the time average requires that the system is aperiodic, otherwise period samples may not accurately capture the average of the system. You are evaluating a periodic system and your colleague Jordan says that it's still possible to use the time average approach if the sample intervals are random instead of periodic. Is this correct? Justify your answer.

(h) (Profiling.) The Coz profiler discussed in lecture gives us some information about what would happen if we sped up a part of the program. Sometimes speeding up a part of the program makes the overall performance worse. We already gave one example: it may increase lock contention. Give another example of a situation where speeding up a part of the program may make overall performance worse.

(i) (Queueing Theory.) In class we talked about interchangeability of servers offering services. How would you describe the interchangeability of courses in your degree? Justify your answer.

(j) (Compiler Optimizations.) One compiler optimization we did not cover is called *strength reduction*. This is where a compiler replaces an expensive operation and transforms it into one or more less-expensive operations. Provide an example of such a transformation.

2 Compilers versus software engineers [20 marks]

Think of this as the long-answer question, as compared to the short answer questions above. We delve into one topic in more detail.

In this course, we've seen that there are many performance improvements that compilers can't do. That's good for us instructors, at least, since we still have techniques that we can teach you.

[10 marks] Lecture 18 discussed manual optimizations for `rustc`. Pick three `rustc` manual optimizations (either from Lecture 18 or from the cited reference materials) and discuss in 1–3 sentences why each these optimizations could not be automatically done by the compiler. Also, name one compiler optimization that cannot be done by the programmer at the source code level, and (in 1–3 sentences) say why not.

[10 marks] Write down one common pattern that at least 3 different compiler optimizations share; name the pattern, the 3 optimizations, and say why they apply the pattern (in 1 sentence each). Is it always (or almost always) a win to apply this pattern? An example of a pattern is "do less work" (but now you can't choose that one).

3 Buffering, Please Wait... [10 Marks]

Buffering is a technique we discussed that could be used to improve performance when writing data out to disk. In the `buffer` directory is an incomplete implementation of buffering. To complete it, you need to implement the trait `Flush` to have the behaviour below:

The buffer should be flushed whenever it is full, and in the starter code there is already a periodic timer that triggers the flush every 1 second in case the buffer does not fill up in that time. Flushing means writing the contents of the buffer out to the file (`output.txt`).

Your code should do a better job than *no* buffering, which should be easy to evaluate with `hyperfine`. You can play around with different buffer sizes by changing the constant `BUFFER_SIZE` to different values. To see what the impact is with no buffering at all, set this constant to 1. You should use the **debug** version of the code, to prevent optimizations from the compiler from interfering with the measurements. Test via:

```
hyperfine "target/debug/buffer input.txt loremipsum.txt"
```

Add your code changes to the repository.

Commentary, Compile Release. In the `README.md` file in the `buffer` folder, write down your testing parameters and the outcomes as reported by `hyperfine`.

Also, try running the different buffer sizes with `hyperfine` using a **release** build. How does that affect your observations? Add this to the `README.md` file as well.

4 Rust fundamentals [10 marks]

This question gives you a chance to demonstrate some of what you've learned about the Rust borrow checker.

a. Removing clones [5 marks] Your task here is to remove the calls to `.clone()`. *The Rust Programming Language* says that it's OK, as a beginning Rust programmer, to use `clone()` here just to get things working, but you can indeed rewrite this code to not use `clone()` while still populating the struct `S` with the args. You may not rewrite the struct `S`. You may not use `unsafe`.

```rust
use std::env;

// must call with two command-line parameters
fn main() {
    let some_vec : Vec<String> = env::args().collect();
    let s = S::new(&some_vec);

    println!("got␣struct␣S␣with␣foo␣{}␣and␣bar␣{}", s.foo_string, s.bar_string);
}

// do not change this struct definition
struct S {
    foo_string: String,
    bar_string: String,
}

// you will need to change the impl
impl S {
    fn new(args: &[String]) -> S {
        let foo_string = args[1].clone();
        let bar_string = args[2].clone();

        S { foo_string, bar_string }
    }
}
```

b. Lifetimes [5 marks] Consider the following short program.

```rust
fn main() {
    println!("{}", g(&59));
}

// may not change the signature of g()
fn g<'a>(w:&'a i32) -> &'a i32 {
    // may not remove 'static on rv's type annotation
    let rv:&'static i32 = f(&4, w);
    rv
}

fn f(i:&i32, _j:&i32) -> &i32 {
    i
}
```

This won't compile. Add lifetime annotations to make it compile. As it says in the comment, you may not remove the `'static` annotation on `rv`, nor may you change the signature of function `g()`. Also, you may not add `'static` anywhere. I really don't know how you might use `unsafe` here, but don't do that.

5 Password Cracking [10 Marks]

You are a hacker, and you are currently on a mission to crack people's passwords. However, the only way you know how to crack a password is to compare the hash values between the one generated by the password and the one generated by your guess. If they are equal, you assume the guessed password is the one you want to crack. However, you do not want to recalculate the hashes of every guess nor store every `(password, hash)` pair. Thus, you decided to build a rainbow table to balance time and space.

There are two tasks: one is to build the rainbow table, and the other is to use the table. However, we focus on the first task: building the rainbow table.

In this question, you can assume every possible password is a 6-digit number, such as `123456`. The hashing algorithm is MD5 (`https://docs.rs/md-5/latest/md5/`).

Implement the `TODO`s in the starter code, ensuring that the rainbow table is built efficiently. You can ignore collisions. Your code should run faster than hashing passwords sequentially.

6 Typst compiler is now open source! [20 Marks]

Typst is a new markup-based typesetting system which aims to match the power of LaTeX. After several months of closed preview testing, the Typst compiler is now open source: `https://github.com/typst/typst`.

However, it turns out to be slow under some circumstances. Your task is to profile and optimize it.

For this question, we've provided you with a modified version of the Typst compiler and a document `documents/notes.typ` to compile into a pdf.

To build the `typst` binary:

```
cargo build --release
```

To compile the document into a pdf:

```
./target/release/typst compile documents/notes.typ documents/notes.pdf
```

6.1 Identifying targets (5 marks)

We've created the flamegraph for you[1]. You can find `flamegraph.svg` in the `typst` directory. Your job in this part is to investigate it. Consider the two largest components under `counter::Counter::at`, then calculate how much speedup for the *entire* program you could potentially achieve by reducing each of these parts' runtimes to 0 (in isolation). Please put your answer in the `README.md` under the `typst` folder.

The flamegraph could be created using:

```
cargo flamegraph -- compile documents/notes.typ documents/notes.pdf
```

but you don't really want to.

6.2 Optimization (15 marks)

You may notice the `comemo` prefixes. This is a crate designed to enable incremental compilation in Typst (`https://github.com/typst/comemo`). In summary, it allows Typst to memorize a function by caching its return values, storing the hash value so that the function only needs to be executed once per set of unique arguments. We cannot optimize this crate directly, but we can determine how to optimize Typst with the knowledge it provides.

Choose one of the two components, then:

Explain its logic (5 marks) Briefly explain what it is doing and how it is used in the program. An IDE can help you explore the call structure. Please put your answer in the `README.md` under the `typst` folder.

Do the optimization (10 marks) Briefly explain your optimization in the `README.md` under the `typst` folder (a few sentences should suffice), implement the optimization and measure its impact on performance. You can measure the performance using:

```
hyperfine "./target/release/typst compile documents/notes.typ documents/notes.pdf"
```

Adjust the default `hyperfine` parameters if needed.

[1]There is a sad story here about how eceubuntu and ecetesla seem to need 25 minutes to create the flamegraph; when I [PL] tried to run it on my machine, I found that I needed Rust 1.64, and Debian/Ubuntu only provide 1.63 except in experimental; so we're providing that file for you just in case.