ECE 459: Programming for Performance
Final Exam

Zahin Mohammad

April 19, 2021

1 Question 1

1.1 a

Fully parallelizable tasks benefit directly with the number of CPU's thrown at it. An example of this is the parallel JWT solver. Programs that can't use more CPU's without restructuring are sequential programs, that dont utilize any parallelism. An example of this are P-complete problems such as the Circuit Value problem (computing the output of a Boolean circuit on a given input). Some programs are in between 1 and 2 because they have both parallelizable and serial parts in their execution.

1.2 b

Little's law states:

$$E[N] = \lambda * E[T] \tag{1}$$

Plugging in the numbers we have $\lambda = 10$, $E[T] = \frac{9+1}{2}$. Therefore,

$$E[N] = 10 * \frac{9+1}{2} = 50 \tag{2}$$

If each job is \$0.001, then the total cost is,

$$E[N] * \$0.001 = \$0.05/s \tag{3}$$

1.3 c

An example of where this approach goes badly wrong is if the array is sorted with a high variance. Taking the average for the first half of the array would very wrong as it is ignoring the highest/lowest elements which would skew the average a lot. A data source where this is likely to go well from uniform randomly distributed data.

1.4 d

Basic MD5 is easily crack-able using GPU accelerated rainbow tables.

1.5 e

This is slow because it presents branching that results in two expensive operations being performed. To make this program faster I would remove the need for branching by removing the use of if/else statements.

1.6 f

I would use inline for simple and short methods that are called frequently. To verify these changes made an impact we can use profiling tools such as flamegraph.

1.7 g

This can backfire when dealing with a collection or an array of boxed values. A primitive array is one reference, and looping through this to access the values inside is a single de-reference per item. An array of boxed values requires at least N references where N is the number of items in the array. Looping through this to access the values inside is multiple de-reference per item.

1.8 h

Canceling threads in Rust requires programmers to implement the functionally by informing threads in some manner (ex. message passing) on how to stop execution (not really canceling). In contrast, in C programmers can cancel threads from the main thread given an ID without giving the thread an option to get canceled/cleanup.

1.9 i

Since the the tasks are independent, I would use asynchronous I/O. This lets us create requests even before previous have finished. The disadvantage of using this is the overhead of creating and managing multiple requests; harder to reason about when compared to sequential.

1.10 j

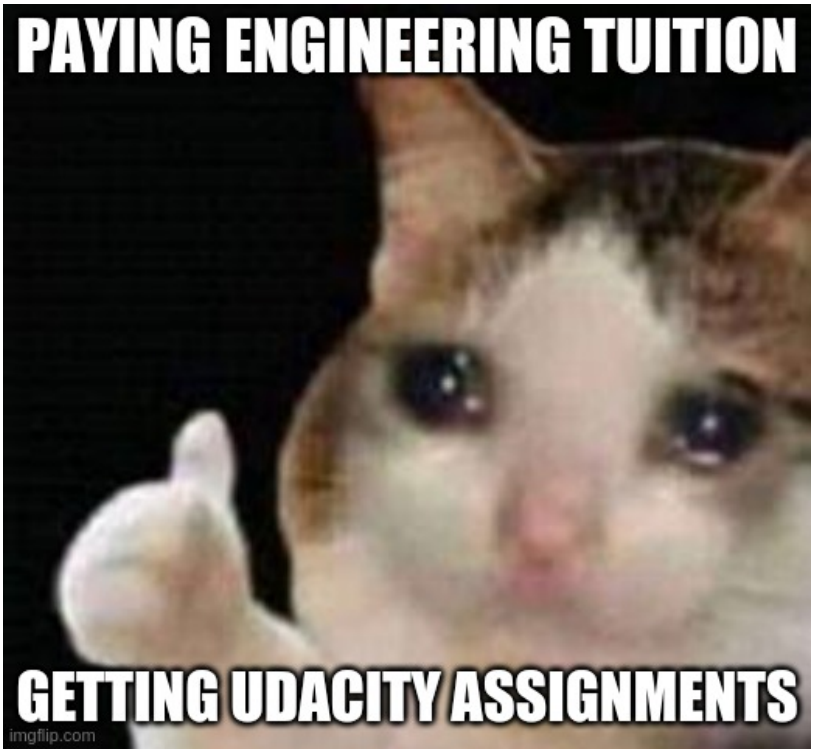

Figure 1: Quality Meme

2 Question 2

```
let some_var = AtomicUsize::new(5);
{
    let some_var = some_var.clone();
    let t = thread::spawn(move || {
        some_var.compare_and_swap(5, 10, Ordering::Relaxed
    }));
}

some_var.compare_and_swap(10, 12, Ordering::Relaxed)

// If t thread executes first, then some_var holds 12
// If main thread executes first, then some_var holds 10
```

3 Question 3

See code.

4 Question 4

See code.

5 Question 5

5.1 Identifying Targets

I would focus on the style_tree first as it has the deepest stack and is also one of the widest in the flamegraph.

5.2 Improvement strategies

One change I could make is using a hahsmap to store rules based on tag, classes and id. This would require refactoring how the Rule type is used in *style.rs* and *css.rs*.

In *doms.rs* I would remove the functions in *ElementData*. These are computations that happen repeatedely but dont change for a node, so they can be part of the struct itself.

Relating to the point above, I would remove *attributes* in *ElementData*. After adding the classes and ID to the struct, this map is no longer being used.

5.3 Implementing an improvement

The code change that was implemented was removing the functions implemented for the *ElementData* struct and have them saved into the struct itself. Additionally the attributes map was removed since the classes and ID are now being stored in the struct itself. This was more efficient as it saves the program from recomputing the classes each time. Below is the before and after results of hyperfine on *ecetesla*0.

```
Before:
Time (mean):      673.0 ms    28.9 ms    [User: 581.9 ms, System: 85.5 ms]
Range (min ... max):    637.4 ms ... 721.8 ms    10 runs
After:
Time (mean ):      605.6 ms    24.6 ms    [User: 520.3 ms, System: 83.8 ms]
Range (min ... max):    562.2 ms ... 633.5 ms    10 runs
```