ECE 459 W21 Final Exam Solutions

J. Zarnett, P. Lam

April 19, 2021

(1)

(a) Many answers are acceptable, but Monte Carlo integration is an example of an embarrassingly-parallel problem, while linked list traversal needs restructuring. Lock contention is a reason that you might be able to parallelize but not perfectly: sometimes a thread has to wait for another thread to release the lock.

(b) We know $\lambda = 10$ jobs/second and $E[T] = E[S]$ (no queuing) which is 5 seconds; therefore $E[N] = \lambda E[T] = 10 \times 5 = 50$ jobs are in progress at any given time, on average. Compute time costs $0.001 per instance-second, so $50 \times 0.001 = 0.05$ so the compute costs is 5 cents per second.

(This makes the computing bill something on the order of 1.5 million dollars per year. Let's hope it's a company that can afford it.)

(c) One concrete example is if you are trying to compute the average over a sorted array. The average of the first half is not at all the same as the average of the second half in that case. On the other hand, something which is uniformly distributed works well. For instance, you are standing on the street and recording the height of everyone who walks by. (This is not perfect: what if the basketball tournament is happening when you're 75% through with your observations? But it's good enough for this question.)

(d) This is bad because MD5 is not cryptographically secure—it would make it easier to use brute force to guess passwords.

(e) GPUs execute all the code on all the branches, so it's going to execute both `expensive_operation_one` and `expensive_operation_two`, which are, as the name says, expensive. If the operations are actually independent (and they should be), you could run them in two kernels concurrently.

(f) You would be looking for a function that is very short and therefore the overhead of the function call is meaningful in the analysis. Then you benchmark with and without the macro to determine if it's an improvement.

(g) Use profiling to find hotspots and inline them. Measure the results on reasonable loads e.g. using hyperfine.

(h) Rust does make it easier, because one of the problems with thread cancellation in C is ensuring that all resources are released. Rust manages the resources with ownership, so if a thread with a variable owning a piece of memory is cancelled, we can be sure it is freed. It's hard to unlock and deallocate everything in C because some functions might be cancellation points and you might not

get to everything!

(i) If there is redundant I/O between the tasks, reduce it. It can be a winning strategy to buffer the I/O so that you do larger chunks of it at once. One potential disadvantage of buffering is slower response time, if your computation is trying to be interactive.

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

(3) Here is the contents of file `q3-solution-naive-main.rs`. You can also use scopes. I have a solution with scoped-pool in `q3-solution-scoped_pool-main.rs`.

```
diff --git a/W21/code/q3/src/main.rs b/W21/code/q3/src/main.rs
index a6244b3..72fff7a 100644
--- a/W21/code/q3/src/main.rs
+++ b/W21/code/q3/src/main.rs
@@ -1,17 +1,47 @@
+use std::sync::{Arc, Mutex};
+use std::thread;
+
+const NTHREADS:usize = 4;
+const ITERATIONS:usize = 10000000;
+
 fn main() {
-    let mut tally = q3::score::ScoreTally::new(1);
-    let mut a_counter = 0;
-    let mut b_counter = 0;
-    for _ in 0..10000000 {
-       tally.add(vec![("Alice", a_counter), ("Bob", b_counter)]);
-       a_counter = (a_counter * 5 + 13) % 17;
-       b_counter = (b_counter * 9 + 17) % 31;
+    let starting_points = vec![(0,0),(13,16),(13,11),(13,17)];
+
+    let mut tallies = Vec::with_capacity(NTHREADS);
+    for _ in 0 ..= NTHREADS {
+       tallies.push(Arc::new(Mutex::new(q3::score::ScoreTally::new(1))));
     }
+    let mut joins = Vec::with_capacity(NTHREADS);
+    for n in 0..NTHREADS {
+       let tallies = tallies.clone();
+       let (mut a_counter, mut b_counter) = starting_points[n];
+       let handle = thread::spawn(move || {
+            let mut tally = tallies[n].lock().unwrap();
+            for _ in 0..ITERATIONS/NTHREADS {
+               tally.add(vec![("Alice", a_counter), ("Bob", b_counter)]);
+               a_counter = (a_counter * 5 + 13) % 17;
+               b_counter = (b_counter * 9 + 17) % 31;
+            }
+       });

-    let totals = tally.totals();
-    for (candidate, score) in tally.totals().iter() {
-        println!("{} got a score of {}", candidate, score);
+       joins.push(handle);
+    }
+
+    for handle in joins {
+       handle.join().unwrap();
+    }
```

```
+
+    let mut big_tally = q3::score::ScoreTally::new(1);
+    for t in tallies.iter() {
+       for (candidate, score) in t.lock().unwrap().totals().iter() {
+            let mut v:Vec<(&str,usize)> = Vec::with_capacity(1);
+            v.push((*candidate, *score));
+            big_tally.add(v);
+       }
     }

-    assert_eq!(totals, vec![("Bob", 146666680), ("Alice", 84375000)]);
+    for (candidate, score) in big_tally.totals().iter() {
+        println!("{} got a score of {}", candidate, score);
+    }
+    assert_eq!(big_tally.totals(), vec![("Bob", 146666680), ("Alice", 84375000)]);
 }
```

(4) A wins 89.45% of the time; B wins 10.46% of the time; Tie 0.09% of the time.

To mark it, you'll need to run it and verify the correct outcome. But also look to validate that the right work is being done in the GPU and not with the CPU.

If I run it naively it's about 16 seconds on ecetesla0, if it's efficient it's about 2 seconds. If they have a solution that's very slow, -5 marks (so max 15)

(5)

```
diff --git a/src/style.rs b/src/style.rs
index 36853e1..2e846af 100644
--- a/src/style.rs
+++ b/src/style.rs
@@ -6,6 +6,7 @@
 use dom::{Node, NodeType, ElementData};
 use css::{Stylesheet, Rule, Selector, SimpleSelector, Value, Specificity};
 use std::collections::HashMap;
+use rayon::prelude::*;

 /// Map from CSS property names to values.
 pub type PropertyMap =  HashMap<String, Value>;
@@ -61,7 +62,7 @@ pub fn style_tree<'a>(root: &'a Node, stylesheet: &'a Stylesheet) ->
    StyledNode<
             NodeType::Element(ref elem) => specified_values(elem, stylesheet),
             NodeType::Text(_) => HashMap::new()
         },
-        children: root.children.iter().map(|child| style_tree(child, stylesheet)).collect(),
+        children: root.children.par_iter().map(|child| style_tree(child, stylesheet)).collect()
    ,
     }
 }
```

(6) In this case, you're mostly just inspecting the code to see that the right function calls are moved to speculative execution – threads are created and joined to check the data.