Functional and Parallel Programming Final Exam

Date: Thursday, December 3rd, 2020
Instructor: Rachata Ausavarungnirun

Problem 1 (25 Points):	
Problem 2 (20 Points):	
Problem 3 (25 Points):	
Problem 4 (30 Points):	
Extra Credit (5 points):	
Total (100+5 points):	

Instructions:

1. This is a 48-hour exam.
2. Submit your work as a zip file on Canvas.
3. Because this is a 48-hour exam, I expect everyone to `test your code`. Hence, you must **submit the test cases for all the coding questions** along with the code and explain what each test case is used for. Test cases, along with the explanation of how you test your code, will factor into 10% of all the questions.
4. If not specified, input and output types are a part of the question. Please use appropriate input and output types that make sense for the purpose of the question.
5. Please clearly comment your code, especially if your code do not work perfectly,
6. Clearly indicate your final answer for each conceptual problem.
7. Our typical restriction for Scala and Rust packages are similar to Assignments 3 and 4.
8. **DO NOT CHEAT.** If we catch you cheating in any shape or form, you will be penalized based on **my plagiarism policy** ($N*$ 10% of your total grade, where N is the number of times you plagiarized previously).

Tips:

- **Read everything.** Read all the questions on all pages first and formulate a plan.
- **Be cognizant of time.** The submission site will close at 00.01AM on Saturday.
- **Show work when needed.** You will receive partial credit at the instructors' discretion.

1. Potpourri [25+5 points]

For this questions, please put your answers here and submit the pdf, or put an answer in a separated pdf named `potpourri.pdf`. Please be as brief as possible. Long answer that has a mixture of correct and incorrect statement can result in a lower grade than a brief but correct statement. Be fundamental.

(a) **Parallelism** **[5 points]**

Define parallelism.

(b) **Ownership System Benefits** **[5 points]**

Please list two benefits of the ownership system.

(c) **Shared Memory vs. Message Passing** **[5 points]**

What are the key differences between the shared memory model and message passing? Feels free to list just one.

(d) **I am Lazy** [**5 points**]

What is the key benefit of Lazy evaluation when you are dealing with a stream?

(e) **Reads vs. Writes** [**5 points**]

In parallel programming using Rust, we have been saying that you should avoid using `mut` as much as possible. Why? Please be fundamental.

(f) **EXTRA CREDIT: GPUs vs. OpenMP** [**5 points**]

What is the type of parallel program that the GPU utilizes? Briefly explain why.

What is the type of parallel program that the OpenMP tool utilizes? Briefly explain why.

2. mergeSort in Scala [20 points]

In class, we show you how to implement a mergesort in parallel in Scala. Use the same algorithm and the concept of Scala's Await and Future, implement a function *mergeSort*. This function must take in any list that its elements have an `Ordered` trait (See `https://www.scala-lang.org/api/2.5.1/scala/Ordered.html` for more information), and produce a sorted list.

Once you are done with the implementation, please provide the analysis of the work and depth of your design.

Please save the file with the name `mergeSort.scala`

3. Playing with Rust [25 points]

This question ask you on two unrelated topics.

(a) **Fibonacci Optimization [15 points]**

Write a function `fib(n)` that calculate the Fibonacci number of n. However, please 1) parallelize your code as much as possible (or make your code as fast as possible) and 2) explain what is the work and depth of your implementation.

Note that our grading scheme for this question will also depend on the speed of your implementation (total of 5 points). I will test your code and rank all the submissions' performance using a relatively large value of n (as long as the output fits within `uint64`. Ranking will be done in the increment of 20% (i.e., if you are in the top 80%, you gets full credit, if you are in the next 60% - 80% you get 4 out of 5 points, etc.

Please save the file with the name `fib.rs`

(b) **Newton's Method to Compute 'e' [10 points]**

In class, we went over the concept behind the Newton's method to approximate the value of the Euler's number (we did this during the in-class exercise 2). In this problem, you must come up with an implementation of a *Euler's number* using the newtonian method, and calculate the work and depth of your implementation.

Please save the file with the name `valueOfE.rs`

4. Binary Search Tree Strikes Again [30 points]

In this question, you are going to implement a binary search tree in **Rust**.

Please limit the use of mutable. You may get points taken off if your answers use lots of mutable unnecessarily.

Below is the definition of a binary search tree:

A binary tree is a tree that consist of up to two childs: left and right subtrees. The left and right subtrees either be a binary tree or a leaf node (which contains no child).

A binary search tree (BST) is a binary tree where every single element on the left subtree is lower or equal to the value of the current node, and the values of every single element on the right subtree is higher than the current node.

An in-order traversal is a method to traverse your tree in a way that it first visits all the left subtree first, follow by the middle (current node), then the right subtree. Performing an in-order traversal on a binary search tree will result in a list that is sorted from lowest to highest.

(a) Unlike the Scala version you implemented during the midterm, you are now going to use a struct in Rust. Please create a struct called Node that consist of an 64-bit unsigned integer value (`u64`), the left Node and the Right Node. More information on Rust's struct can be found here: `https://doc.rust-lang.org/rust-by-example/custom_types/structs.html`

(b) Implement an `insertVal` function that **takes an input value (`u64`) and an input tree (`Node`)**, and insert this value as a new node to your tree while ensuring you preserve the BST's property. **This function should return a resulting tree after you insert the node.**

(c) Implement an `walkTree` function that performs an in-order traversal **on the input tree** and **return a vector of u64 values** (which should now be sorted from lowest to highest). **Please refrain from using mutable here.**

(d) Implement an `heightCheck` function that **takes an input tree and return the height of the tree**. We define height as the maximum depth of all the branches in your tree. **Please refrain from using mutable here.**

(e) Implement a function called `parallelLookup` that takes in a vector of unsigned 64-bit integer. The function performs a **parallel lookup** to check whether each elements is in the tree or not and then return the total number of elements that are in the tree. (For example, if our tree contains [1, 2, 5, 9, 10, 100, 10101] and our input is [2, 9, 11, 100, 101], then, the function `parallelLookup` returns 3 because three of the values (2, 9, 100) in the input arrays are in our tree.

(f) Once everything is all done, please write your own test cases that thoroughly test your code **so that it also works in corner cases**. Your test should shows, as best as possible, that your implementation works as intended. (**Hint:** I am sure you have these from the midterm ...)

Please save the file with the name `bst.rs`