Rose Whitt

322 HW 2 Report

Part A

Summary

I created a list of futures with the futures returning a list of users. Then I iterate through the repos and create a future. In this future, I create a list of users (tempUsers) and set it to null. I assign the output of gerRepContributorsCall (like in the sequential version) to the tempUsers in a try catch block. If this is not null, I return it. If it is, I return null. Now, outside of the future, but inside the for loop, I add the future to the list of futures.

Next, I create an async await that waits for the list of futures. I create a list of users and then iterate through the list of futures, safe getting each future and adding all of its users to the list of users.

When the for loop is done, I begin the aggregation process. I stream the list of users (parallel stream) and collect the data, grouping by the user and summing the user contributions. This creates a map of key type user and value type integer. Next, I stream this map's entry set (parallel stream). I first map to a new user with the pair's key's login and the pair's value, which is the summed contributions. I then sort in ascending order and collect to a list. I call the updateContributors function on this list of users, exit the async await, and return the repos' size.

Part B

It is data race free because it consistently gets the same output and the same memory location is not being accessed concurrently from two or more threads in a single process.

Part C

The expected value of work is n + 1.

This is because I loop through n repositories, doing all calculations concurrently, thus only doing one unit of work per repository. The + 1 comes from my async await.