COMP322
HW2 Report
Michelle Pang

(a)
I created a list of futures in which each future returns a list of users. Then, using a for-loop, I iterate through the repo, and for each repo, I create a future, in which I create a list of users(tempUsers) and set it to null. Then I assign the output of getRepContributorsCall to the tempUsers in a try-catch block. If this tempUser is not null, I return it; if it is null, I return null. Outside the future but inside the for-loop, I add the future to the list of futures.

Then I create an async await to wait for the list of futures. In the async await, I create a list of users. Then, using a for-loop, I 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 finishes, I start the aggregation process. Here, I stream the list of users parallelly and collect the data, grouping by the user and summing the user distributions. This creates a map of key type User and value type Integer. Then, I stream this map's entry set parallelly: I first map to a new user with the pair's key's login and the pair's value, which is the summed contributions, then I sort in ascending order and collect to a list. Next, I call updateContributors on this list of users. Then, outside of async await, I return the repo's size.

(b)
My implementation is correct and data-race-free because it always gets the same output, and there are no two or more threads accessing the same memory location concurrently in a single process.

(c)
I expect to see a work of N + 1 from my implementation. This is because I looped through n repositories, doing all calculations concurrently, thus doing only 1 unit of work per repository. And since the async await takes 1 unit of work, it takes N + 1 of work in total.