<u>COMP 322 Homework 5 Report</u>

Ideal Parallel Scoring

Summary:

For the first checkpoint, I parallelized the Smith-Waterman alignment algorithm by iterating over the diagonals and computing them in parallel. I came to this implementation by writing out the algorithm a lot until I saw a pattern of dependencies, which is the anti-diagonals. I noticed that there are always width of the matrix + height of the matrix - 1 diagonals in a matrix, so I sequentially iterated over that first using a *forseq* with variable *k*. Then I iterated over each of the elements in this diagonal in parallel by using a *forall* that iterated from 1 to the index of the *forseq* loop (*k*) with variable *j*. To get the x index, I subtracted *j* from *k* and added one. I checked that *(i, j)* were within the bounds of the lengths of the strings and, if they were, did the body of the sequential scoring algorithm.

Correct & Data-Race Free:

My solution is data-race free because no two iterations are reading or writing to the same location/element in the matrix. I ensured this through my anti-diagonal iterating implementation that avoided any dependency conflicts.

My solution is correct because I did not change any of the core computational behavior given in the Sequential Scoring algorithm.

I know that my implementation is correct and data-race free because I pass all of the correctness tests and consistently get the same results.

Useful Paralell Scoring

Summary:

I parallized the Smith-Waterman alignment algorithm with > 33x speedup by using a two dimensional Data Driven Future array, chunking, finish, Futures, and an asyncAwait.

Basically, I divided the matrix into a bunch of "baby matrices" that ran in parallel with each other, baring dependencies, but the baby matrix itself was calculated using the given sequential algorithm.

I first calculated the size of partitions. It took me a while to figure out the perfect divisor/sweet spot, but I ultimately decided on making the x and y partitions the length of that string divided by 80, rounded up, and converted to an integer. I then calculated the number of partitions that the matrix will contain by just dividing the string length by its partition size.

I then intialized the Data Driven Future array (*HjDataDrivenFuture<Integer>[][]*) to be the number x partitions wide and the number of y partitions tall (calculated above). The setting up of this DDF matrix was my biggest challenge. I filled the first row and first column of the DDF matrix with zeros in order to kick start the async await (see below).

The next step was the actual scoring. Everything from here on is wrapped in a finish, except for the return. I iterated from 1 through the number of partitions (x and then y), then created a Future for all three dependencies in the DDF matrix (top, diagonal, left). I registered the asyncAwait with these. I then "converted" the DDF indices that I am iterating through to their corresponding scoring matrix indices. I created a helper that was the exact sequential algorithm that took in the x and y strings and the calculated row/column start and end indices. For testing purposes, I had this helper return the score and I put this baby matrix's score into the DDF matrix, but I really could've put any arbitrary number in there because this is just signalling to the asyncAwait that we are ready to continue. After the finish ends, I return the bottom left corner and I am done!

Correct & Data-Race Free:

My solution is correct and data race free because no two parallel asyncAwait tasks are reading/writing to the same location.

Within each chunk or "baby matrix", the matrix scoring is computed sequentially. Therefore, no data races are possible within these baby matrices that make up the scoring matrix. The asyncAwait always waits for its dependencies to ensure that, as discussed in my Ideal Parallel Scoring algorithm, there are no data race conflicts. This ensures that when it is time to calculate a baby matrix, we know that the baby matrices above, diagonally to the left, and to the left have already been calculated and can be used as dependencies for this current scoring.

My solution is correct because I did not change any of the core computational behavior given in the Sequential Scoring algorithm. I still calculate the score of each block using the dependencies (top, left, and diagonally left) and return the bottom right element that contains the final score. In this algorithm, I just do this in chunks, but still with the same logic and results!

I know that my implementation is correct and data-race free because I pass all of the correctness tests and consistently get the same results.

Measurements:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running edu.rice.comp322.Homework5PerformanceTest

HABANERO-AUTOGRADER-PERF-TEST 48T
edu.rice.comp322.Homework5PerformanceTest.testUsefulParScoring 1389 41
testUsefulParScoring ran in 41 ms, 33.8780487804878x faster than the sequential (1389 ms)

HABANERO-AUTOGRADER-PERF-TEST 48T
edu.rice.comp322.Homework5PerformanceTest.testUsefulParScoring2 1386 40
testUsefulParScoring ran in 40 ms, 34.65x faster than the sequential (1386 ms)
```

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 39.578 sec - in
edu.rice.comp322.Homework5PerformanceTest

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 44.511 s
[INFO] Finished at: 2022-04-24T23:07:27-05:00
[INFO] Final Memory: 22M/320M
[INFO] ------------------------------------------------------------------------
```