Homework 2: due by 11:59pm on Wednesday, February 24, 2021 (Total: 100 points)

Instructor: Mackale Joyner

Commit and push all work to the GitHub repo at https://classroom.github.com/a/ckHt-AFN. In case of problems committing your files, please contact the teaching staff at comp322-staff@mailman.rice.edu before the deadline to get help resolving your issues.

Your solution to the written assignment should be submitted as a PDF file named `hw2_written.pdf` in the hw2 directory. This is important — you will be penalized 5 points if you place the file in some other folder or with some other name. The PDF file can be created however you choose. If you scan handwritten text, make sure that the writing is clearly legible in the scanned copy. Your solution to the programming assignment should be submitted in the appropriate location in the hw2 directory.

The slip day policy for COMP 322 is similar to that of COMP 321. All students will be given 3 slip days to use throughout the semester. When you use a slip day, you will receive up to 24 additional hours to complete the assignment. You may use these slip days in any way you see fit (3 days on one assignment, 1 day each on 3 assignments, etc.). If you plan to use a slip day, you need to say so in an svn committed README.md file before the deadline. You should specifically mention how many slip days you plan to use.

Other than slip days, no extensions will be given unless there are exceptional circumstances (such as severe sickness, not because you have too much other work). Such extensions must be requested and approved by the instructor (via e-mail, phone, or in person) before the due date for the assignment. Last minute requests are likely to be denied.

If you see an ambiguity or inconsistency in a question, please seek a clarification on Piazza (remember not to share homework solutions in public posts) or from the teaching staff. If it is not resolved through those channels, you should state the ambiguity/inconsistency that you see, as well as any assumptions that you make to resolve it.

Honor Code Policy: All submitted homeworks are expected to be the result of your individual effort. You are free to discuss course material and approaches to problems with your other classmates, the teaching assistants and the professor, but you should never misrepresent someone else's work as your own. If you use any material from external sources, you must provide proper attribution.

1 Written Assignments (50 points total)

Submit your solutions to the written assignments as a PDF file named `hw2_written.pdf` in the hw2 directory. Please note that you be penalized 10 points if you misplace the file in some other folder or if you submit the report in some other format.

1.1 Parallel Fibonacci using Futures (25 points)

Consider the HJlib code shown below in Listing 1 to compute the Fibonacci function in parallel using futures. (Note that this parallel algorithm is based on a *highly inefficient* sequential algorithm because it does not use memoization or dynamic programming; however, we will use this version for simplicity.)

1. Provide an exact answer for the *total work* performed by a call to fib(n). Include an explanation of the analysis, and state what expression you get for WORK(n) as a function of n. (15 points)
2. Perform a theoretical big-O analysis for the *critical path length* for a call to fib(n). Include an explanation of the analysis, and state what expression you get for CPL(n) as a function of n. (10 points)

```
1     public static int fib(int n) throws SuspendableException {
2         if (n <= 0) return 0;
3         else if (n == 1) return 1;
4
5         HjFuture<Integer> f1 = future(() -> fib(n - 1));
6         HjFuture<Integer> f2 = future(() -> fib(n - 2));
7
8         Integer f1Val = f1.get();
9         Integer f2Val = f2.get();
10        doWork(1); // only count addition in abstract metrics
11        return f1Val + f2Val;
12    }
```

Listing 1: Parallel Fibonacci using Futures

Hints based on common errors from past years: Note that an empirical analysis of CPL obtained with different inputs for the code in Listing 1 is not a substitute for a theoretical big-O analysis. Also, pay attention to where constants matter in a big-O analysis, e.g., $O(2^n)$ and $O(3^n)$ are not the same. Finally, be sure to use the big-O notation if your analysis includes big-O approximations; you will not get full credit if you include a correct big-O answer without the big-O notation.

1.2 Finish Accumulators (25 points)

Consider the pseudocode shown below in Listing 2 for a Parallel Search algorithm that is intended to compute Q, the number of occurrences of the pattern array in the text array. What possible values can variables *count0*, *count1*, and *count2* contain at line 16? Write your answers in terms of M, N, and Q, and explain your answers.

```
1  // Assume that count0, count1, count2 are declared
2  // as object/static fields of type int
3  . . .
4  count0 = 0;
5  accumulator a =  new accumulator(SUM, int.class);
6  finish (a) {
7    for (int i = 0; i <= N - M; i++)
8    async {
9       int j;
10      for (j = 0; j < M; j++) if (text[i+j] != pattern[j]) break;
11      if (j == M) { count0++; a.put(1); } // found at offset i
12      count1 = a.get().intValue();
13   } // for-async
14 } // finish
15 count2 = a.get().intValue();
16 // Print count0, count1, count2
```

Listing 2: Parallel Search using Finish Accumulators

Hints based on common errors from past years: Be sure to include exactly all possible values for the variables, not a subset or superset of the values. Remember to use Q in your answers, even though Q can have different values for different values of the `text[]` and `pattern[]` arrays (even for the same values of M and N.) Finally, don't forget to explain your answers.

2 Programming Assignment (50 points)

2.1 Setup

See the Lab 1 handout for instructions on HJlib installation for use in this homework, and Lecture 3 for information on abstract execution metrics. The provided code for HW 2 can be found in your GitHub repository at https://classroom.github.com/a/ckHt-AFN.

2.2 Abstract Overhead

Thus far, our abstract metrics have assumed an idealized execution in which there is no overhead in creating *async* or *future* tasks. In the real world, asyncs are not actually free: they consume processor cycles and system memory. In this homework, we will simulate that cost using abstract metrics, and try to implement a parallel algorithm in such a way as to obtain the best CPL value when taking abstract overheads into account.

In an effort to make abstract metrics a bit more realistic, we will add an *abstract overhead* for the programming assignment in Homework 2. The idea behind abstract overhead is to charge a certain cost, C, to a parent task whenever it creates a child task. This cost will be added as sequential work to the parent just before the child task is created. For example, if a task creates N async child tasks, it will incur an overhead of $N \times C$ units of work which will be added to other work that the task is doing.

2.3 Parallel Matrix Multiply with Abstract Overhead (50 points)

The goal of this assignment is to implement a parallel matrix multiply program with the smallest critical path length, when taking abstract async overhead into account. If you need to brush up on matrix-matrix multiplies, see the sample code in Worksheet 1 Question 2 or https://www.mathsisfun.com/algebra/matrix-multiplying.html. Your solution should work for matrices of all sizes (within the limits of the memory capacity of your machine), but you will be graded by multiplying two $N \times N$ matrices for $N = 1024$ with an abstract async overhead cost of $C = N = 1024$. The abstract metrics should count one unit of work for each multiply operation, and assume that all other operations (other than the abstract async overhead) are free. When $C = N$, we expect the best solution to have a critical path length of approximately $N \times (2 \times log_2(N) + 1)$.

Make sure to add calls to `doWork(1)` for each multiply operation performed as part of your matrix-matrix multiply implementation. If you do not, your results will be misleading.

We have provided a basic template of a `Matrix` class which can be multiplied by another matrix class. `Matrix` includes sample sequential and parallel implementations of matrix-matrix multiply. Note that the CPL of the parallel implementation is much higher than the best solution of $N \times (2 \times log_2(N) + 1)$.

You should complete the `optimizedParallelMultiply` method, with the goal of minimizing the CPL of your matrix-matrix multiply solution for $N = C = 1024$. You are free to add any tests or other code you like under the main/ and test/ directories, but please do not modify the folder structure of the project.

A correct parallel program should generate the same output as the sequential version, and should not exhibit any data races. The parallelism in your solution should be expressed using only `async`, `finish`, and/or `future` constructs. It should pass the unit tests provided, and other tests that the teaching staff may use while grading.

2.4 Submitting

As with HW 1, you will need to add, commit, and push all work to the GitHub repository. The repository is located at https://classroom.github.com/a/ckHt-AFN. Please open a browser, navigate to the url, and verify that you successfully committed your homework. Don't forget to add a README.md file if you plan to use slip days.

Your submission should include the following in the hw2 directory:

1. (25 points) Your completed solution to the parallel matrix multiply problem, that attempts to minimize CPL in the presence of an abstract overhead implemented in the `Matrix.optimizedParallelMultiply` method. We will only evaluate the performance of your solution using abstract metrics, and not its actual execution time.

   15 points will be allocated based on the ideal parallelism that you achieve and the correctness of your implementation. You will get the full 15 points if you achieve a CPL of $N \times (2 \times log_2(N) + 1)$ or better, when the abstract overhead is $C = N$.

   10 points will be allocated for coding style and documentation. At a minimum, all code should include basic documentation for each method in each class. You are also welcome to add additional unit tests to test corner cases and ensure the correctness of your implementations.

2. (15 points) A report file formatted as a PDF file named `hw2_report.pdf` in the hw2 directory. The report should contain the following:

   (a) A summary of your parallel algorithm, and the steps that you had to take to minimize CPL in the presence of abstract overehad.

   (b) An explanation as to why you believe that your implementation is correct and data-race-free.

   (c) An explanation of what value of CPL (as a function of N) you expect to see from your implementation, and why.

   Hints based on common errors/omissions from past years: Be sure to explain what parallel constructs you used, why, and what subcomputations ran in parallel as a result. Also, justify why you believe that your parallel solution will always produce the same output as the sequential version. In addition, you should explain why there are no data races in your solution. Finally remember to explain what value of CPL (as a function of N) you expect to see from your implementation, and why.

3. (10 points) The report file should also include test output for the CPL value obtained for matrix multiply with N = C = 1024 as inputs. Also, include the IDEAL PARALLELISM (= N^3/CPL) value obtained from your CPL value. Note that N^3 is included in the numerator, since that is the total useful work (excluding abstract overhead).