Homework 2: due by 12noon on Friday, February 12, 2016 (Total: 100 points)

Instructor: Vivek Sarkar, Co-Instructor: Shams Imam.

All homeworks should be submitted in a directory named "hw_2" in your svn repository for this course. In case of problems committing your files, please contact the teaching staff at comp322-staff@rice.edu before the deadline to get help resolving for your issues. No late submissions will be accepted unless you are using your slip days.

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 use slip days, you must submit a SLIPDAY.txt file in your SVN homework folder before the actual submission deadline indicating the number of slip days that 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 do receive an extension from the instructor, please indicate this by placing an EXTENSION.txt file in your SVN homework folder before the actual submission deadline indicating the date that the extension was granted by the instructor as well as the length of the extension.

If you see an ambiguity or inconsistency in a question, please seek a clarification on Piazza 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 `hw_2_written.pdf` *in the hw_2 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 is based on a highly *inefficient* sequential algorithm because it does not use memoization or dynamic programming, but we will use this version for simplicity.)

1. Perform a big-O analysis for the total work performed by a call to fib(n). What expression do you get for WORK(n) as a function of n? (15 points)

   Hint: The closed form for Fibonacci number $Fib_i = (\phi^i - \hat{\phi}^i)/\sqrt{5}$, where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = \frac{1-\sqrt{5}}{2}$ (i.e., the two roots of the equation $x^2 = x + 1$).

2. Perform a big-O analysis for the critical path length for a call to fib(n). What expression do 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

1.2 Finish Accumulators (25 points)

Consider the pseudocode shown below in Listing 2 for a Parallel Search algorithm that is intended to compute C, 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, and why? Write your answers in terms of M, N, and C.

```
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

2 Programming Assignment (50 points)

2.1 Setup

For this homework, you can choose to set up your environment manually (as in Lab 3) or use IntelliJ-Maven configuration to do so automatically. Note that the teaching staff have resolved the Maven repository issues by migrating to a new repository host, so we expect the IntelliJ-Maven configuration to be the simpler option.

For either option, you should start by checking out the project template for HW2 from your SVN turnin folder, located at `https://svn.rice.edu/r/comp322/turnin/S16/`*your-netid*`/hw_2`. You can either do this from the command line:

```
svn checkout https://svn.rice.edu/r/comp322/turnin/S16/NETID/hw_2
```

or use IntelliJ's VCS support by going to File > New > Project from Version Control > Subversion, clicking the + button at the top of the pop up, entering the same https URL, clicking Checkout, and selecting a location on your local machine to place your copy of HW2. Note that if you checkout from the command line, you will then need to import the checked out folder as a new project in IntelliJ.

If you choose to configure your environment automatically, then from the IntelliJ project you just created from Subversion find the included pom.xml file in the file navigator (likely on the left hand side of your screen). This pom.xml will be in the top-level directory of the checked out project. Right-click pom.xml, and select "Add as Maven Project". This option is towards the bottom of the options list. Once you have done that, again right-click on the same pom.xml file and you should now see a "Maven" option towards the bottom of the list. Click on Maven > Reimport to be sure IntelliJ has pulled in the JARs for this homework. Finally, go to File > Project Structure and in the pop up select "Modules" from the left menu. You should see three JARs listed in the central pane: hjlib-cooperative:0.1.8, asm-all:5.0.3, and junit:3.8.2. Check the checkboxes next to each, hit "Apply" at the bottom right of the window, and close the window by clicking "OK".

If you choose to configure your system manually, follow the instructions in the Lab 3 handout starting with the second bullet point in Section 1.1 and using the project you just created from Subversion. Note that for this homework you will need a newer version of HJlib (v1.8) than was used for Lab 3. We have provided a single ZIP on the COMP 322 website containing all three of the necessary JARs to run this homework if you choose to manually install them. Also note that you should *not* modify any folder structure for this homework.

Finally, regardless of whether you created the project manually or automatically, navigate to File > Project Structure and verify that in the "Project" pane you have "1.8" selected as the "Project SDK". You will also need to add the `-javaagent` JVM option to any Run Configurations you create in this lab, as has been described in the other labs and homeworks. If you are unsure what the full path to your HJlib JAR is, you should be able to expand "External Libraries" in the file navigator, right click on the hjlib-cooperative JAR listed under there, select "Open Library Settings", and copy the full path from that pop up window.

Once the project is properly configured, you should be able to go to Build > Rebuild Project and successfully compile the Java classes you have been provided with. You can verify this by running the JUnit tests inside `edu.rice.comp322.MatrixMultiplyCorrectnessTest`. Two of the six tests should fail and will by fixed by you: `testOptimizedParallelMultiplySimple` and `testOptimizedParallelMultiplyRectangular`.

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 an effort to make the model 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.

In Lab 3, we learned that abstract metrics can be enabled by invoking

```
HjSystemProperty.abstractMetrics.setProperty(true);
```

before calling `launchHabaneroApp()`. For this homework, we introduce another call,

```
HjSystemProperty.asyncSpawnCostMetrics.setProperty(C);
```

that sets the abstract async overhead to C, and should also be called before `launchHabaneroApp()`. The abstract async overhead is a cost measured in work that a task pays each time an async is created. 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 limit the effect of that cost on the CPL.

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 $N \times (2 \times log_2(N) + 1)$.

You will need to add calls to

```
HjSystemProperty.abstractMetrics.setProperty(true);
HjSystemProperty.asyncSpawnCostMetrics.setProperty(C);
```

where appropriate (i.e. before `launchHabaneroApp`) in any tests you write to verify your implementation. Also, 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 will use while grading.

2.4 AutoGrader

This homework will be supported on the AutoGrader. As with Lab 1, simply navigate to ananke.cs.rice.edu, and upload a ZIP file of your solution for the COMP322-S16-HW2 assignment. Your ZIP file should be created from the top-level hw_2/ directory. This directory includes your pom.xml and src/ directory. For example, you can create it from the command line using:

```
zip -r hw_2.zip hw_2/
```

or by right-clicking on the hw_2 directory in your file browser and selecting "Send to > Compressed (zipped) folder" (Windows) or "Compress hw_2" (Mac OS).

You are not required to use the AutoGrader, it is simply meant to be a useful tool. It will run several static checks (including checkstyle and FindBugs) to help identify problems in your code. The teaching staff may also upload additional tests to the AutoGrader during the assignment to help verify the correctness of your implementation (if new tests are uploaded, a notification will be sent out on Piazza).

2.5 Submitting

Your submission should include the following in the hw_2 directory and should be turned in via the hw_2 turnin folder you checked out as an IntelliJ project at the start of this section:

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 `hw_2_report.pdf` in the hw_2 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.

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 work excluding abstract overhead.