Homework 1: due by 5pm on Wednesday, January 23, 2013 (Total: 100 points)

Instructor: Vivek Sarkar

All homeworks should be submitted in a directory named hw_1 using the turn-in script. In case of problems using the script, you should email a zip file containing the directory to `comp322-staff@mailman.rice.edu` **before the deadline. See course wiki for late submission penalties.**

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 elses work as your own. If you use any material from external sources, you must provide proper attribution.

1 Written Assignments (50 points total)

Please submit your solutions to the written assignments in either a plain text file named `hw_1_written.txt` *or a PDF file named* `hw_1_written.pdf` *in the hw_1 directory.*

1.1 Finish Synchronization (25 points)

Consider the sequential and incorrect parallel versions of the HJ code fragment included below. Assume rows in the two-dimensional arrays are distinct objects (subarrays). *Your task is to only insert finish statements in the incorrect parallel version so as to make it correct i.e., to ensure that the parallel version computes the same result as the sequential version, while maximizing the potential parallelism.*

```
// SEQUENTIAL VERSION:
 for (int i = 0 ; i < n ; i++)
    for (int j = 0 ; j < n ; j++)
      c[i][j] = 0;
  for (int i = 0 ; i < n ; i++)
    for (int j = 0 ; j < n ; j++)
      for (int k = 0 ; k < n ; k++)
        c[i][j] += a[i][k] * b[k][j];
  System.out.println(c[0][0]);

// INCORRECT PARALLEL VERSION:
  for (int i = 0 ; i < n ; i++)
    for (int j = 0 ; j < n ; j++)
      async c[i][j] = 0;
  for (int i = 0 ; i < n ; i++)
    for (int j = 0 ; j < n ; j++)
      async
        for (int k = 0 ; k < n ; k++)
          c[i][j] += a[i][k] * b[k][j];
  System.out.println(c[0][0]);
```

1.2 Analyzing Finish-Async Programs (25 points)

Consider the computation graph in Figure 1. Each node is labeled with the steps name and execution time e.g., B(2) refers to step B with an execution time of 2 units. Program execution starts with step A(1) and ends with step C(1).

1. (10 points) *Calculate the total WORK and CPL (critical path length) for this task graph.*

2. (15 points) *Write a Habanero-Java program with basic finish and async constructs (no futures) that can generate this computation graph.* The steps should be clearly identified in the program. The CG edges are not labeled below as spawn, continue, and join; you can make whatever assumptions you like about the edges when writing your program. It is okay if your program results in the addition of redundant transitive edges to the CG, but it should have exactly the same parallelism structure as the computation graph in Figure 1.

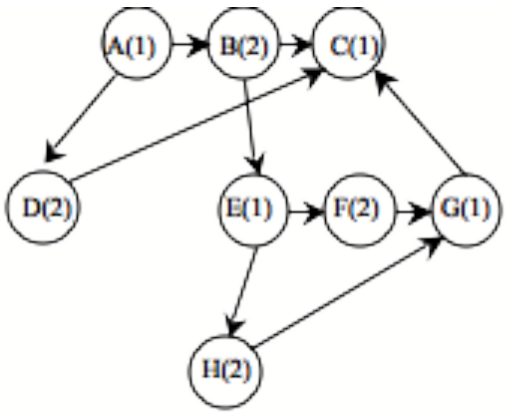

Figure 1: Sample Computation Graph

2 Programming Assignment (50 points)

2.1 Habanero-Java Setup

See Lab 1 for instructions on setting up a Habanero-Java installation for use in this homework, and Lab 2 for obtaining abstract execution metrics.

2.2 Parallel Quicksort (50 points)

Quicksort is a classical sequential sorting algorithm introduced by C.A.R. Hoare in 1961, and is still very much in use today. A sequential version of the Quicksort algorithm is provided in the course wiki in the `quicksort.hj` file, along with the Homework 1 document. For abstract execution metrics, this version includes a call to `doWork(1)` each time a key comparison is performed. As you will see in Lab 2, you can execute an HJ program with an option to generate abstract performance metrics by selecting "Show Abstract Execution Metrics" in DrHJ's Compiler Option preferences, or (if you are not using DrHJ) typing the following command on the command line, "`hj -perf=true` ...". For the sequential version, the `WORK` and `CPL` metrics will be identical.

Your assignment is to convert the sequential program to a correct parallel program with a smaller critical path length (ideal parallel time) than the sequential version. A correct parallel program will generate the same output as the sequential version and will also not exhibit any data races. Your parallel solution should only choose from `async`*,* `finish`*, and* `future` *constructs, since HJ abstract performance metrics are currently only supported for these three constructs.*

Your submission should include the following in the lab_1 directory:

1. (25 points) A complete parallel solution for Quicksort as outlined above in a modified quicksort.hj file.

We will only evaluate its performance using abstract metrics, and not its actual execution time. All code should include basic documentation for each method in each class.

2. (15 points) A report file formatted either as a plain text file named `hw_1_report.txt` or a PDF file named `hw_1_report.pdf` in the hw_1 directory. The report should summarize the design of your parallel solution, and explain why you believe that your implementation is correct, data-race-free, and maximally parallel (to the best of your effort).
3. (10 points) The report file should also include test output for sorting arrays of size n = 10, 100, and 1000. The test output should include both the result value and the WORK, CPL, and IDEAL SPEEDUP (= WORK/CPL) from each run.

NOTE: See Section 2.3 (Parallelizing Quicksort, Approach 1) in the Module 1 handout for technical details on sequential and parallel variants of the Quicksort algorithm. Approach 1 is the simplest of the three parallelization approaches discussed in Module 1. The other two approaches are discussed in Sections 8.2 and 8.3. You will not lose points for using Approach 1, but you are welcome to try Approach 2 or 3 if you're interested in doing so.