Homework 1: due by 11:59pm on Tuesday, September 18, 2012 (Total: 100 points)

Instructor: Vivek Sarkar
Co-Instructor: Ran Libeskind-Hadas

All homeworks should be submitted using the submission system at http://cs.hmc.edu/submit. In case of problems using the system, please email your homework to cs181ehelp@cs.hmc.edu.

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 a plain text file named `hw1_written.txt` *in the submission system.*

1.1 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 only finish and (non-future, non-await) async constructs 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.

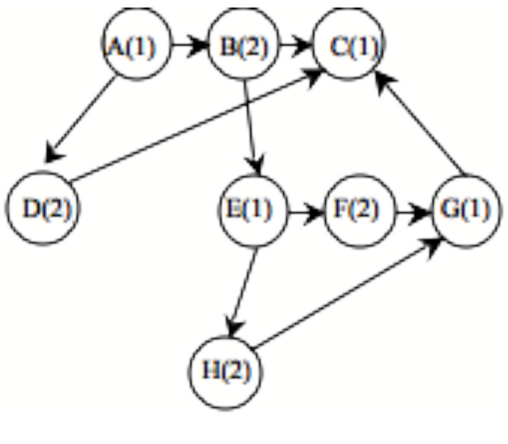

Figure 1: Sample Computation Graph

1.2 Future Tasks and Data-Driven Futures (25 points)

1. (10 points) *Summarize the similarities and differences between futures and data-driven futures in HJ.* In your summary, you should state if it is possible to create a race condition and/or a deadlock when accessing the value in the future container when using either construct.

2. (15 points) Consider the HJ code fragment below that operates on an array of DataDrivenFutures. (Note that there are no `get()` operations in this example, so the only purpose of the `put()` operation is to synchronize with `await` clauses. For simplicity, we just use an empty string `""` as the object being put into a DataDrivenFuture.)

   Is it possible for any instance of the async statement in line 6 to be indefinitely blocked on its await clause? If so, explain how. If not, explain why not.

```
1.       DataDrivenFuture[] A = new DataDrivenFuture[n];
2.         for (int i = 0 ; i < n ; i++) {
3.             A[i] = new DataDrivenFuture();
4.         }
5.         for (int i = n-1 ; i >= 1 ; i--) {
6.             async await(A[i-1]) {
7.                 . . .
8.                 A[i].put();
9.             } // async
10.      } // for
11.      A[0].put();
```

2 Programming Assignments (50 points total)

2.1 Habanero-Java Setup

Please see Section 2.1 of the Homework 0 handout for instructions on setting up a Habanero-Java installation for use in this homework.

2.2 Parallel Quicksort (25 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 given in quicksort.hj. For abstract execution metrics, this version includes a call to `addLocalOps(1)` each time a key comparison is performed. As in Homework 0, 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.*

Include the parallel program in your submission, along with ideal parallel times for the sequential and parallel versions as comments at the top. Also include one or two sentences as comments at the top of the file with your opinion on whether you feel that the ideal speedup for your program is satisfactory or not. The submission system will ask for a file named `quicksort.hj`*,which should be your parallel solution.*

NOTE: See Section 12 (Parallel Quicksort) in the Module handout for technical details on sequential and parallel variants of the Quicksort algorithm. Approach 1 is the simplest of the three parallelization approaches listed in this section, but you are welcome to use any approach that you choose.

2.3 Parallel N-Queens and Finish Accumulators (25 points)

Download the nqueens.hj program.

This program uses a recursive method, `nqueens_kernel()`, to enumerate all solutions found for the nqueens problem. The goal of the program is to compute the total number of solutions found for a given value of n (the default value is $n = 12$). Currently, program execution reports "Incorrect answer" since a finish accumulator is provided but not properly computed.

Your task is twofold:

1. *Use the finish accumulator provided to fix the* `nqueens.hj` *program so that its output is correct. Be sure to see all FYI and TODO comments in the* `nqueens.hj` *file.*
2. *Compare the performance of sequential and parallel versions of your* `nqueens` *program. You can obtain a sequential version by commenting out the "*`async`*" keywords. The performance is automatically timed.*

   To report the results form this performance comparison, please obtain measurements for different command-line options ("hj nqueens 12 5 3", "hj nqueens 12 5 4", ...) by varying the third parameter, which is the cutoff_value threshold used for efficiency. You can keep increasing the threshold until the execution time becomes too long (> 2 minutes, for example). Further, please make sure that all performance measurements are obtained on the same machine (which should have 2 or 4 cores.)

   Include the parallel program in your submission, along with parallel times for different cutoff values as comments at the top. The submission system will ask for a file named `nqueens.hj`*.*

NOTE: See Section 5 and Section 8.5 in the Module 1 handout for technical details on "finish accumulators" and the "seq" clause respectively.