Homework 4: due by 11:59pm on Friday, October 12, 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 Assignment (50 points total)

Please submit your solution to this assignment in a plain text file named `hw4_written.txt` *in the submission system.*

1.1 Observationally Cooperative Multithreading (25 points)

The classical Dining Philosophers problem can be stated as follows; this description was taken from the Wikipedia entry for this problem:

> "Five silent philosophers sit at a table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it's not being used by another philosopher. After he finishes eating, he needs to put down both the forks so they become available to others. A philosopher can grab the fork on his right or the one on his left as they become available, but can't start eating before getting both of them.
>
> The problem is how to design a discipline of behavior (a concurrent algorithm) such that each philosopher won't starve, *i.e.,* can forever continue to alternate between eating and thinking, assuming that any philosopher can not know when others may want to eat or think."

A solution to the Dining Philosophers problem using the Observationally Cooperative Multithreading (OCM) model (Lecture 8) is shown below in Listing 1. In this solution, the HJ `forall` construct is used to create a parallel task for each philosopher, and the OCM model is assumed for the execution of each `forall` task/iteration.

```
const int numPhilosophers = 5;
const int numForks = numPhilosophers;
Fork[] fork = ... ; // Initialize array of forks
forall(point [p] : [0:numPhilosophers-1]) {
  think(p);
  yield;
  eat (p, fork[p], fork[(p+1) % numForks);
  yield;
}
```

Listing 1: OCM solution to the Dining Philosophers problem

For simplicity, assume execution of the program in Listing 1 on a single processor, when answering the following questions:

1. *(10 points) What assumptions need to be made for the* `yield` *statement to ensure that no philosopher ever starves?*
2. *(5 points) Convert the code in Listing 1 from the OCM model to a standard HJ program by removing the* `yield` *statements. Can this HJ program exhibit a data race? If so, show where the data race can occur. If not, explain why not.*
3. *(10 points) Now transform the above HJ program (obtained by removing* `yield` *statements) by inserting HJ* `isolated` *statements to mimic the semantics of the* `yield` *statements in the original OCM version. What assumptions need to be made for scheduling of* `isolated` *statements in HJ to ensure that no philosopher ever starves with this version?*

1.2 Mystery Actor Program (25 points)

The main program fragment for initiating a mystery actor computation is shown in Listing 2, and the definition of the process() method for the MysteryActor class is shown in Listing 3. Though the full class definition is not shown, you can assume that it includes the following members:

- isExitMsg(Object msg) — virtual method that checks if msg is the exit message
- seq, val — private Integer variables for an instance of MysteryActor (seq and val are part of the actor's local state)
- isMyMultiple(Integer num) — virtual methods that checks if num is a multiple of val
- MysteryActor(Integer seq, Integer val) — constructor that creates actor with specified values for seq and val
- nextActor — reference to next actor in the chain, if any

```
1        twosFilter = new MysteryActor(new Integer(1), new Integer(2));
2        finish {
3          twosFilter.start();
4          for (int i = 3; i <= limit; i++) twosFilter.send(i);
5          twosFilter.send(EXIT_MSG);
6        }
```

Listing 2: Fragment of main() method for mystery actor program

1. *(10 points) Describe what output should be expected from this program for a given (integer) value of limit in Listing 2. What sequence is embodied in the print statements executed in line 4 of Listing 3?*
2. *(15 points) Consider a sample execution of this program with limit=10. How many actors are created? How many calls to send() are performed?*

2 Programming Assignment (50 points total)

2.1 Habanero-Java Setup

See Section 2.1 of the Homework 0 handout for instructions on setting up a Habanero-Java installation for use in this homework. Also, see Section 3 of this handout for available command-line options when compiling

```java
void process(Object msg) {
  if (isExitMsg(msg)) { // msg is the exit message
    if (nextActor != null) nextActor.send(msg);
    System.out.println("Position " + seq + " has value = " + val);
    exit();
  } else {
    boolean isMultiple = isMyMultiple((Integer) msg);
    if ( !isMultiple ) {
      // msg is not a multiple of val (for this actor)
      if (nextActor == null) {
       // create new actor to filter out multiples of msg
       nextActor = new MysteryActor((Integer) seq+1, (Integer) msg);
       nextActor.start();
      }
      else nextActor.send(msg); // pass it on
    } // if
  } // if-else
} // process()
```

Listing 3: process() method for mystery actor program

and executing HJ programs. In general, the command-line interface (rather than DrHJ) is recommended for performance measurements, since DrHJ incurs additional overhead on the machine by running in a separate JVM from the one used to execute your application. Since HJ's work-stealing schedulers currently do not support actors, you have to use one of HJ's work-sharing schedulers (either -s or -fj) for this assignment. (The -fj fork-join scheduler is likely to perform better.) The input file needed for this assignment can be found in SequentialMain.hj.

2.2 Parallelization of a Filter Bank pipeline

The SequentialMain.hj. file contains a port of the Filter Bank StreamIt program to sequential HJ. (You do not need to know the StreamIt language to do this assignment.) This program performs multi-rate signal processing and consists of multiple pipeline branches. On each branch, the pipeline involves multiple stages including multiple delay stages, multiple FIR filter stages, and sampling. Since Filter Bank represents a pipeline, it can easily be implemented using actors, with one actor per stage. The FIR filter stage is stateful, appears early in the pipeline, and can be a bottleneck in the pipeline because of the time required to process each input. Parallelizing the computation of the weighted sum in the FIR filter stage will further help improve the performance of this program.

Your assignment is to design and implement a parallel version of the Filter Bank program, using the provided sequential implementation as a starting point. Your solution must use actors to implement the pipeline stages, but you can use additional HJ constructs as well. Your homework deliverables are as follows:

1. **[Parallel version (35 points)]**
   Create a new parallel version of `SequentialMain.hj` that is designed to achieve the *smallest execution time*. You will be graded on the real speedup achieved by your parallel version relative to the sequential version.

   Please place your solution in a file named "`ParallelMain.hj`". You are allowed to add or modify method definitions in any class, so long as your implementation generates the same output as the sequential version (for any given input).

2. **[Homework report (15 points)]**
   You should submit a brief report summarizing the design of your parallel algorithm, explaining why you believe that each implementation is correct and data-race-free.

Your report should also include the following measurements

(a) Performance of the sequential version with the default input

(b) Performance of the parallel version with the default input, executed with the "`-places 1:1`" option to run with one worker and the "`-places 1:k`" option to run with k workers (where k is the number of cores available in the machine that you used).

If you obtained a speedup due to parallelism, your report should explain why it occurred. On the other hand, if you did not obtain a speedup due to parallelism, your report should explain why that did not occur.

Please submit your report in a plain text file named `hw4_report.txt` *in the submission system. If you prefer to submit a PDF file, that's okay too. (Send email to cs181ehelp@cs.hmc.edu with your report, if you have problems submitting a PDF file.)*

3 Compile-time and Runtime options for Habanero Java

These options are described assuming that HJ programs are compiled and executed using the command-line interface. See DrJava's Preferences panes to select some of these options when using DrJava.

To compile an HJ program, Foo.hj, type "*hjc [options] Foo.hj*". The following command-line options are currently available for hjc (type "*hjc -help*" for a summary):

-racedet Enable race-detection when program executes (off by default, only works for basic async and finish constructs)

-rt s Compile program for work-sharing runtime (on by default, supports all HJ constructs)

-classpath ⟨**path**⟩ Search path for class files

-sourcepath ⟨**path**⟩ Search path for hj source files (must include -classpath if this option is used)

-destdir ⟨**path**⟩ Set the location where output classes from hjc should be placed

-version Print version number of the HJ compiler. Please include this version number when reporting any problems to comp322-staff.

To execute a compiled HJ program, Foo.hj, type "*hj [options] Foo [args]*", where *args* are the command-line arguments for your program'. The following command-line options are currently available for hj (type "hj -help" for a summary):

-places ⟨**p**⟩**:**⟨**w**⟩ Set number of places and workers per places. The default value is $p = 1$ and $w =$ number of processors in the system.

For now, you will be working with 1 place, so you can use *-places 1:n* to run an HJ program with n workers (for n cores).

-fj Use Fork-Join variant of work-sharing runtime (off by default, assumes that program has been compiled for work-sharing execution)

-version Print version number of the HJ runtime system. Please include this version number when reporting any problems to comp322-staff.

-mx ⟨**size**⟩ set max heap size, e.g., -mx 8000M sets the max heap size to 8GB

-classpath ⟨**path**⟩ Search path for class files

-J⟨**arg**⟩ Pass ⟨arg⟩ directly to Java runtime (use with caution)