Lab 8: Actors

Instructor: Vivek Sarkar

Resource Summary

Course wiki: `https://wiki.rice.edu/confluence/display/PARPROG/COMP322`

Staff Email: comp322-staff@mailman.rice.edu

Clear Login: ssh *your-netid*@ssh.clear.rice.edu and then login with your password

Important tips and links:

edX site : `https://edge.edx.org/courses/RiceX/COMP322/1T2014R`

Piazza site : `https://piazza.com/rice/spring2014/comp322/home`

Java 8 Download : `https://jdk8.java.net/download.html`

IntelliJ IDEA : `http://www.jetbrains.com/idea/download/`

HJ-lib Jar File : `http://www.cs.rice.edu/~vs3/hjlib/habanero-java-lib.jar`

HJ-lib API Documentation : `https://wiki.rice.edu/confluence/display/PARPROG/API+Documentation`

HelloWorld Project : `https://wiki.rice.edu/confluence/display/PARPROG/Download+and+Set+Up`

Sugar Login: ssh *your-netid*@sugar.rice.edu and then login with your password

Linux Tutorial visit `http://www.rcsg.rice.edu/tutorials/`

On SUGAR, JDK8 is already available at `/users/COMP322/jdk1.8.0` and HJ-Lib is already installed at `/users/COMP322/habanero-java-lib.jar`. Run the following command to setup the JDK8 path.

```
source /users/COMP322/hjLibSetup.txt
```

When you log on to Sugar, you will be connected to a *login node* along with many other users. To request a dedicated *compute node*, you should use the following command from a SUGAR login node:

```
qsub -q commons -I -V -l nodes=1:ppn=8,walltime=00:30:00
```

We learned that there may be some issues with SUGAR this week. If you're unable to obtain a compute node, please use your laptop for today's lab instead.

Note that all commands below are CaSe-SeNsItIvE. For example, be sure to use "S14" instead of "s14".

IMPORTANT: please refer to the tutorial on Linux and SUGAR, before staring this lab. Also, if you and others experience long waiting times with the "qsub" command, please ask the TAs to announce to everyone that they should use ppn=4 instead of ppn=8 in their qsub command (to request 4 cores instead of 8 cores).

1 HJ Actors

HJ actors were introduced in Lectures 21–22. An actor class is defined by extending the

```
edu.rice.hj.runtime.actors.Actor<MessageType>
```

class. Concrete subclasses are required to implement the `process(Message)` method. Typical uses will have sequential code inside `process()`, but this implementations allows use of any of the HJ parallel primitives inside the `process()` body.

The following code snippet shows the schema for defining an actor class:

```
import edu.rice.hj.runtime.actors.Actor;
import edu.rice.hj.runtime.actors.Message;

public class EchoActor extends Actor<Message> {
    protected void process(final Message aMessage) {
        ...
}}
```

Method calls can be invoked on actor objects, and they work just like method calls on any other HJlib objects. However, what distinguishes actors from normal objects is that they can be activated by the `start()` method, after which the HJlib runtime ensures that the actor's `process()` method is called in sequence for each message sent to the actor's mailbox. The actor can terminate itself by calling `exit()` in a `process()` call.

Messages can be sent to actors from actor code or non-actor code by invoking the actor's `send()` method using a call as follows, " `someActor.send(aMessage)`". A `send()` operation is *non-blocking* and *asynchronous*. The HJlib Actor library preserves the order of messages with the same sender and receiver, but messages from different senders may be interleaved in an arbitrary order.

As mentioned in the lectures, there are three basic states for an actor:

- **new**: when an instance of an actor is created, it is in the new state. In this state, an HJ actor will receive messages sent to its mailbox but will not process them.
- **started**: in this state, the actor will process all messages in its mailbox, one at a time. It will keep doing so until it decides to terminate. In HJ, an actor is started by invoking its `start()` method: *e.g.,* "`myActor.start()`". Any operation immediately after `start()` should be implemented in the `onPostStart()` method.
- **terminated**: in this state the actor has decided it will no longer process any more messages. Once terminated, an actor cannot be restarted. An actor requests termination by calling its `exit()` method, which changes the actor's state to *terminated* after the `process()` call containing `exit()` returns. Note that the `exit()` call does not itself result in an immediate termination of the `process()` call; it just ensures that no subsequent `process()` calls will be processed. Any operations after `exit()` should be implemented in the `onPostExit()` method.

All async tasks created internally within an actor are registered on the `finish` scope that contained the actor's `start()` operation. The `finish` scope will block until all actors started within it terminate. This is similar to the `finish` semantics while dealing with `asyncs`.

1.1 Tips and Pitfalls

- Use an actor-first approach when designing programs that use actors *i.e.,* think about which actors need to be created and how they will communicate with each other. This step will also require you to think about the communication objects used as messages.
- If possible, use immutable objects for messages, since doing so avoids data races and simplifies debugging of parallel programs.
- You cannot override the `start()` or `exit()` methods in actor classes since they are declared final, any operations you need to do immediately before or after `start()` and `exit()` should be implemented in the methods `onPreStart(), onPreExit(), onPostStart(), onPostExit()` respectively.
- The HJ actor `start()` method is not idempotent. Take care to ensure you do not invoke `start()` on the same actor instance more than once. The `exit()` method on the other hand is idempotent, invoking `exit()` multiple times is safe within the same call to `process()`.
- *Always remember to terminate a started actor* using the `exit()` method. If an actor that has been started is not terminated, the enclosing `finish` will wait forever (deadlock).

2 Exercises for today

2.1 Pi Computation using Bailey-Borwein-Plouffe formula

In the spirit of π Day, our first exercise involves computing π to a specified precision in HJ. The following formula can be used to compute π:

$$\pi = \sum_{n=0}^{\infty} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left(\frac{1}{16} \right)^n$$

The `PiSerial1.java` file contains a simple sequential algorithm for computing π using Java's `BigDecimal` data type, that runs for a fixed number of iterations. The `PiActor1.java` file contains a parallel version of `PiSerial1.hj` using Master-Worker style actors, as explained in Lecture 21.

In contrast, the `PiSerial2.java` file contains a more realistic sequential algorithm that uses a `while` loop to compute more and more terms of the series until a desired precision is reached.

We have already provided a version of `PiActor2.java` with `TODO` comments. For this section, your assignment is to convert the sequential program in `PiSerial2.java` (for computing π to a desired precision) to an actor-based parallel program in `PiActor2.java` by filling in code at the `TODO` segments. Next, you will need to evaluate the performance of the serial and parallel versions, `PiSerial2.java` and `PiActor2.java`, on a Sugar compute node. Also, attempt to run with higher precision values while evaluating the performance. As before, you can compile the program as follows:

```
javac -cp /users/COMP322/habanero-java-lib.jar PiActor2.java PiUtil.java
```

To run the program using 8 cores, use the following command on a *compute node*:

```
java -cp /users/COMP322/habanero-java-lib.jar:.  -Dhj.numWorkers=8 PiActor2
```

2.2 Primes Sieves using a Pipeline

The `SieveSerial.java` file contains a sequential version of the Sieve of Eratosthenes algorithm for generating prime numbers. Your assignment is to convert the sequential program in `SieveSerial.java` (for computing the number of primes in a given range) to an actor-based parallel program in `Sieve.java` (by filling in

code at the `TODO` segments), and to evaluate the performance of the serial and parallel versions on a Sugar compute node (use 8 cores for the parallel version). Also, attempt to run with higher limit values while evaluating the performance.

The basic idea is to create multiple stages of the pipeline that forward a candidate prime number to the next stage only if the stage determines the candidate is locally prime. When the candidate reaches the end of the pipeline, the pipeline may need to be extended. Thus, this is also an example of a dynamic pipeline where the number of stages is not necessarily known in advance. A simple diagrammatic explanation of how the pipeline would work is shown in Figure **??**. Note that to reduce the relative overhead, you will need to increase the amount of work done in each stage by having it store and process multiple prime numbers as a batch.

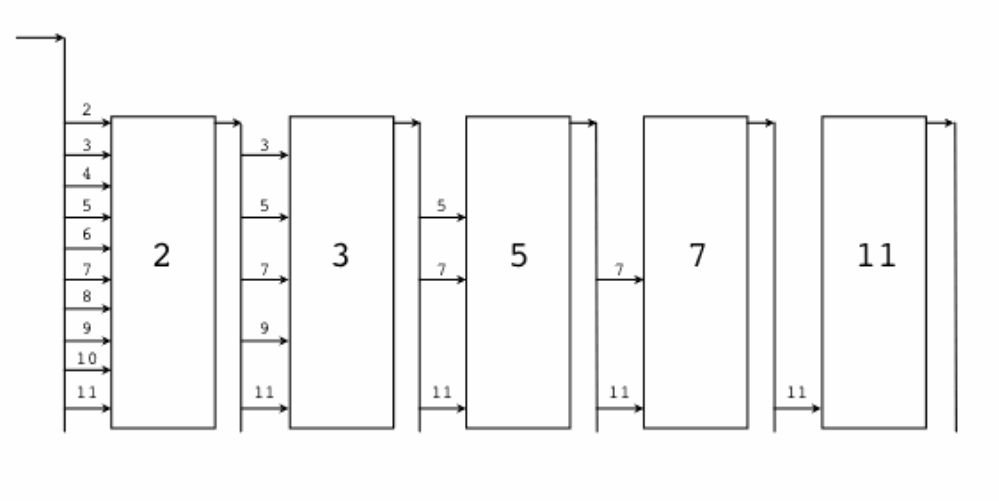

Figure 1: Illustration of Sieve of Eratosthenes algorithm (source: `http://golang.org/doc/sieve.gif`)

3 Turning in your lab work

For each lab, you will need to turn in your work before leaving, as follows.

1. Check that all the work for today's lab is in the `lab_8` directory. If not, make a copy of any missing files/folders there. It's fine if you include more rather than fewer files — don't worry about cleaning up intermediate/temporary files.

2. Use the turn-in script to submit the `lab_8` directory to your turnin directory as explained in the first handout: *turnin comp322-S14:lab_8*. Note that you should *not* turn in a zip file.

   NOTE: Turnin should work for everyone now. If the turnin command does not work for you, please talk to a TA. As a last resort, you can create and email a `lab_8.zip` *file to comp322-staff@mailman.rice.edu.*