Lab 8: Actors and Places

Instructor: Vivek Sarkar

1 Update your HJ/DrHJ Installation

The performance measurements for today's lab should be done on Sugar, and we've already updated the HJ installation there. (See Lab 4 handout on setup instructions to access the HJ installation in the COMP 322 userid on Sugar.)

There is a new release of HJ which includes actors as a library. If you are also working with a local installation, please update it from the HJ download page, https://wiki.rice.edu/confluence/display/PARPROG/HJDownload, to make sure that you have the actor library along with the latest updates and bug fixes.

2 HJ Actors

HJ actors were introduced in Lectures 21–23. An actor class is defined by extending the `hj.lang.Actor` class. Concrete sub-classes of `hj.lang.Actor` are required to implement the `process()` method. A limitation of the current HJ implementation is that it does not support abstract methods that use generic type parameters, so *please restrict yourselves to extending `Actor<Object>` instead of extending `Actor<T>`* .

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

```
1 import hj.lang.Actor;
2 public class EchoActor extends Actor<Object> {
3     protected void process(Object aMessage) {
4         ...
5 } }
```

Method calls can be invoked on actor objects, and they work just like method calls on any other HJ objects. However, what distinguishes actors from normal objects is that they can be activated by the `start()` method, after which the HJ 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 HJ 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()`".
- **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.

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`.

The following HelloWorld example was discussed in Lecture 22 (slide 10), and is also available in `HelloWorld.hj`:

```
1  import hj.lang.Actor;
2
3  public class HelloWorld {
4      public static void main(final String[] args) {
5          EchoActor actor = new EchoActor();
6          finish {
7              actor.start(); // actor attaches itself to finish scope
8              // we are guaranteed ordered sends, i.e. though Hello and ↩
                   World will be
9              // processed asynchronously, they will be processed in that ↩
                   order
10             actor.send("Hello");
11             actor.send("World");
12             actor.send(EchoActor.STOP_MSG);
13         } // wait until actor terminates
14         System.out.println("EchoActor has terminated");
15 } }
16
17 class EchoActor extends Actor<Object> {
18     static final Object STOP_MSG = new Object();
19     protected void process(final Object msg) {
20         if (STOP_MSG.equals(msg)) {
21             exit();
22         } else {
23             System.out.println(msg);
24 } } }
```

Other examples that were discussed in Lecture 22 include `Pipeline.hj` (slides 15–17), and `ThreadRingMain.hj` (slide 18).

2.1 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.
- When overriding the `start()` or `exit()` methods in actor classes, remember to make the appropriate calls to the parent's implementation with `super.start()` or `super.exit()`, 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).

3 Exercises for today

3.1 Pi Computation using Gregory-Leibniz series

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:

$$\pi = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots$$

The `Piserial.hj` file contains a simple sequential algorithm for computing π using Java's `BigDecimal` data type, that runs for a fixed number of iterations. The `Pi.hj` file contains a parallel version of `Piserial.hj` using actors, as explained in slides 12–14 of Lecture 22.

In contrast, the `PiSerial2.hj` 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.

For this section, your assignment is to convert the sequential program in `PiSerial2.hj` (for computing π to a desired precision) to an actor-based parallel program in `Pi2.hj`, and to evaluate the performance of the serial and parallel versions,`PiSerial2.hj` and `Pi2.hj`, on a Sugar compute node (using the "-places 1:8" option for the parallel version).

3.2 Primes Sieves using a Pipeline

The `SieveSerial.hj` file contains a sequential version of the Sieve of Eratosthenes algorithm for generating prime numbers. You already studied a similar example in Lab 1. For this section, your assignment is to convert the sequential program in `SieveSerial.hj` for computing π to an actor-based parallel program in `Sieve.hj` (say), and to evaluate the performance of the serial and parallel versions on a Sugar compute node (using the "-places 1:8" option for the parallel version).

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 available at `http://golang.org/doc/sieve.gif`. 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.

3.3 Places

For this section, your assignment is to extend your solution to the previous assignment (`Sieve.hj`) to use places to increase data locality. To evaluate the performance of the place-based version on a Sugar compute node, you should use the "-places 8:1" option (for 8 places).

Please make your best effort to bind actors to `places` for locality. HJ actors allow you to specify the `place` where the actor is started. Using this configured value, the HJ actor will always run at the designated `place`. A simple example in which an actor, `worker`, is started at the next place relative to the current place is as follows:

```
1 ...
2 place workerHome = here.next(); // next place relative to current place ↩
      (see slide 4 in Lecture 23)
3 worker.start(workerHome);
4 ...
```