COMP 322: Fundamentals of Parallel Programming (Spring 2012)
Instructor: Vivek Sarkar

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

Homework 5: due by 5pm on Friday, April 6, 2012
(2 questions totaling 40 points)

All homeworks should be submitted using the turn-in script. In case of problems using the script, please email your homework to comp322-staff at mailman.rice.edu.

Name: _____

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

1) Places and Distributions [Lectures 17 & 18] (10 points for 1a + 10 points for 1b)

The use of the HJ place construct is motivated by improving locality in a computer system's memory hierarchy. We will use a very simple model of locality in this problem by focusing our attention on *remote reads*. A remote read is a read access on variable V performed by task T0 executing in place P0, such that the value in V read by T0 was written by another task T1 executing in place P1 ≠ P0. All other reads are *local reads*. By this definition, the read of A[0] in line 11 in the example code below is a local read and the read of A[1] in line 12 is a remote read, assuming this HJ program is run with the "`-places 2:1`" option

```
1.   finish {
2.     place p0 = place.factory.place(0);
3.     place p1 = place.factory.place(1);
4.     double[] A = new double[2];
5.     finish {
6.       async at(p0) { A[0] = … ; }
7.       async at(p1) { A[1] = … ; }
8.     }
9.     finish {
10.      async at(p0) {
11.        … = A[0]; // Local read
12.        … = A[1]; // Remote read
13.      }
14.    }
15.  }
```

1a) Locality impact of distributions (10 points)

Consider the following variant of the one-dimensional iterative averaging example studied in the lectures. We are only concerned with local vs. remote reads in this example and not about the (in)efficiency of this code.

i) Your task is to estimate the total number of remote reads in this code as a symbolic function of the array size parameter, N, the number of iteations, M, and the number of places P (assuming that the HJ program was executed using the "`–places P:1`" option).

ii) Repeat part i) above if line 1 was changed to "`dist d = dist.factory.cyclic([1:N]);`"

iii) What conclusions can you draw about the relative impact of block vs. cyclic distributions on the number of remote reads in this example?

```
1. dist d = dist.factory.block([1:N]);
2. for (point [iter] : [0:M-1]) {
3.   finish for(int j=1; j<=N; j++)
4.     async at(d[j]) {
5.       myNew[j] = (myVal[j-1] + myVal[j+1]) / 2.0;
6.     } //finish-for-async-at
7.   double[] temp = myNew; myNew = myVal; myVal = temp;
8. } // for
```

1b) Load balance impact of distributions (10 points)

Consider the example code below that also uses places and distributions. In this example, we are only concerned with estimating the total number of operations performed *at each place* by adding up the contributions from calls to perf.addLocalOps() in line 6, as is done in HJ's abstract performance metrics.

i) Your task is to estimate the total number of operations performed at place q (0<= q<P) as a symbolic function of N, P, and q. For example, if P=1, then the total number of operations performed at place q=0 must be N*(N+1)/2.

ii) Repeat part i) above if line 1 was changed to "`dist d = dist.factory.cyclic([1:N]);`"

iii) What conclusions can you draw about the relative impact of block vs. cyclic distributions in improving the load balance in this example?

```
1.  dist d = dist.factory.block([1:N]);
2.  finish for(int j=1; j<=N; j++)
3.     async at(d[j]) {
4.       for (int i=1; i<=j; i++) {
5.         …
6.         perf.addLocalOps(1)
7.       }
8.  } //finish-for-async-at
```

2) Conversion of compareAndSet() calls to isolated statements [Lectures 23 & 24] (10 points for 2a + 10 points for 2b)

Part 2a) (10 points) Rewrite the two do-while loops in class **IQueue** below to use **isolated** statements instead of the **compareAndSet()** calls in method **enq()** and **deq()**, thereby making explicit the meaning of the **compareAndSet()** calls. (See expansion of AtomicInteger operations in slide 18 of Lecture 20.)

```
1.   import java.util.concurrent.atomic.*;
2.   class IQueue {
3.     AtomicInteger head = new AtomicInteger(0);
4.     AtomicInteger tail = new AtomicInteger(0);
5.     Object[] items = new Object[Integer.MAX_VALUE];
6.     public void enq(Object x) {
7.       int slot ;
8.       // Loop till enqueue slot is found
9.       do slot = tail.get();
10.      while (!tail.compareAndSet(slot,slot +1));
11.      items[slot] = x;
12.    } // enq
13.    public Object deq() throws EmptyException {
14.      Object value; int slot;
15.      // Loop till dequeue slot is found
16.      do {
17.        slot = head.get(); value = items[slot];
18.        if (value == null) throw new EmptyException();
19.      } while (!head.compareAndSet(slot,slot+1));
20.      return value;
21.    } // deq
22.  } // Iqueue
```

Part 2b) (10 points) Analyze the Linearizability of instances of class IQueue as though they were concurrent objects. Either prove that all executions with calls to enq() and deq() will be linearizable, or provide a sample execution that is not linearizable.