Homework 5

Instructor: Vivek Sarkar

Homework 5: due by 11:55pm on Friday, April 24, 2015
Automatic extension till May 1, 2015 (slip days can be added to this)
(Total: 100 points for written assignments + 50 points extra credit for optional programming assignment)

All homeworks should be submitted in the directory named hw_5 in svn. In case of problems using the script, you should email a zip file containing the directory to `comp322-staff@rice.edu` before the deadline. See course wiki for late submission penalties.

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 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 Written Assignment: Dining Philosophers Problem (25 points)

Syntactic errors in program text will not be penalized in all written assignments, e.g., missing semicolons, incorrect spelling of keywords, etc. Pseudo-code is acceptable so long as the meaning of your program is unambiguous.

In Lecture 30, we studied the characteristics of different solutions to the Dining Philosophers problem. Both Solution 1 (using Java's synchronized statement) and Solution 2 (using Java's lock library) had the following structure in which each philosopher attempts to first acquire the left fork, and then the right fork:

```
final int numPhilosophers = 5;
final int numForks = numPhilosophers;
final Fork[] fork = ... ; // Initialize array of forks
forall(0, numPhilosophers-1, (p) -> {
  while(true) {
    Think ;
    Acquire left fork, fork[p] ;
    Acquire right fork, fork[(p-1)%numForks] ;
    Eat ;
  } // while
}); // forall
```

Consider a variant of this approach in which any 4 philosophers follow the above structure, but the 1 remaining philosopher first attempts to acquire the right fork and then the left fork.

1. (15 points) Is a deadlock possible in Solution 1 (using Java's synchronized statement) for this variant with 4 philosophers attempting to first acquire the left fork, and then the right fork, and the fifth philosopher doing the opposite? If so, show an execution that leads to deadlock. If not, explain why not.
2. (10 points) Is a livelock possible in Solution 2 (using Java's lock library) for this variant with 4 philosophers attempting to first acquire the left fork, and then the right fork, and the fifth philosopher doing the opposite? If so, show an execution that exhibits a livelock. If not, explain why not. For the purpose of this program, a livelock is a scenario in which all philosophers starve without any of them being blocked.

2 Written Assignment: Locality with Places and Distributions (25 points)

The use of the HJ place construct (Lecture 31) 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 $\neq$ P0. All other reads are local reads. By this definition, the read of A[0] in line 8 in the example code below is a local read and the read of A[1] in line 9 is a remote read, assuming this HJ program is run with 2 places, each place with one worker thread.

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

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 with the overheads of creating `async` tasks.

```
1.      dist d = dist.factory.block([1:N]); // generate block distribution (Lecture 31)
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
```

1. **(10 points)** Estimate the total number of remote reads in this code as a symbolic function of the array size parameter, N, the number of iterations, M, and the number of places P (assuming that the HJ program was executed using P places, 1 worker thread per place).
2. **(10 points)** Repeat part **??** above if line 1 was changed to `dist d = dist.factory.cyclic([1:N]);`
3. **(5 points)** What conclusions can you draw about the relative impact of block vs. cyclic distributions on the number of remote reads in this example?

3 Written Assignment: Load Imbalance with Places and Distributions (25 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.

1. **(10 points)** 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.
2. **(10 points)** Repeat part 1 above if line 1 was changed to `dist d = dist.factory.cyclic([1:N]);`
3. **(5 points)** 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]); // generate block distribution (Lecture 31)
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
```

4 Written Assignment: Message Passing Interface (25 points)

Consider the MPI code fragment shown below when executed with two processes:

1. **(10 points)** What value will be output by the print statement in process 0?
2. **(15 points)** How will the output change if the Irecv() call is replaced by Recv() (and the Wait() call eliminated)?

```
1.        int rank, size, next, prev;
2.        int n1[] = new int[1]; int n2[] = new int[1];
3.        int tag1 = 201, tag2 = 202;
4.        Request request; Status status;

6.        size = MPI.COMM_WORLD.Size();
7.        rank = MPI.COMM_WORLD.Rank();
8.        next = (rank + 1) % size;
9.        prev = (rank + size - 1) %size;
10.       n1[0] = rank*10 + 1; n2[0] = rank*10 + 2;

11.       if ( rank == 0 ) {
12.         request= MPI.COMM_WORLD.Irecv(n1,0,1,MPI_INT,prev,tag1);
13.         MPI.COMM_WORLD.Send(n2,0,1,MPI_INT,next,tag2);
14.         status = MPI.COMM_WORLD.Wait(request);
15.         System.out.println("Output = " + n1[0]);
16.       } else { // rank == 1
17.         MPI.COMM_WORLD.Recv(n1,0,1,MPI_INT,prev,tag2);
18.         n2[0] = n1[0];
19.         MPI.COMM_WORLD.Send(n2,0,1,MPI_INT,next, tag1);
20.       }
```

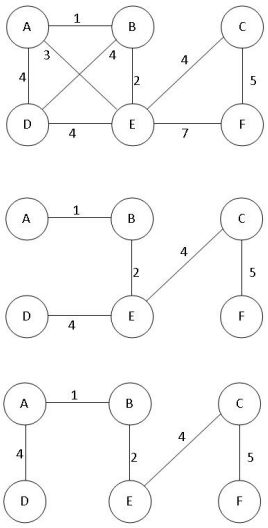

Figure 1: Two possible Minimum Spanning Trees (MSTs) for a given undirected graph (source: `http://en.wikipedia.org/wiki/Minimum_spanning_tree`)

5 Optional Programming Assignment: Minimum Spanning Tree of an Undirected Graph (50 points)

In this homework, we will focus on the problem of finding a *minimum spanning tree* of an undirected graph. Some of you may recall minimum spanning trees from COMP 182. Note that we have studied parallel spanning tree algorithms earlier in COMP 322, but the focus of this assignment is on parallel algorithms for finding the spanning tree with minimum cost.

The following definition is from Wikipedia (`http://en.wikipedia.org/wiki/Minimum_spanning_tree`):

> Given a connected, undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a weight to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A Minimum Spanning Tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of minimum spanning trees for its connected components.

Figure 1 shows there may be more than one minimum spanning tree in a graph. In the figure, two minimum spanning trees are shown for the given graph (each with total cost = 16).

5.1 Boruvka's algorithm to compute the Minimum Spanning Tree of an Undirected Graph

The first known algorithm for finding the MST of an undirected graph was developed by Czech scientist Otakar Boruvka in 1926. Two other commonly used algorithms for computing the MST are Prim's algorithm and Kruskal's algorithm. In this assignment, we will focus on parallelizing Boruvka's algorithm, by providing a reference sequential implementation of that algorithm in Java. *If you prefer to parallelize an alternate MST algorithm, please send email to comp322-staff@rice.edu as soon as possible. You will then be responsible for obtaining or creating a sequential Java implementation of that algorithm as a starting point.*

The following summary of Boruvka's sequential algorithm is from Galois description of Boruvka:

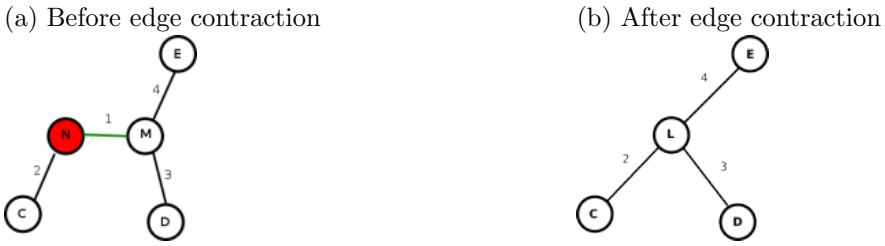

Figure 2: Demonstration of edge contraction

> Boruvka's algorithm computes the minimal spanning tree through successive applications of edge-contraction on an input graph (without self-loops). In edge-contraction, an edge is chosen from the graph and a new node is formed with the union of the connectivity of the incident nodes of the chosen edge. In the case that there are duplicate edges, only the one with least weight is carried through in the union. Figure 2 demonstrates this process. Boruvka's algorithm proceeds in an unordered fashion. Each node performs edge contraction with its lightest neighbor.

In the example in Figure 2, the edge connecting nodes M and N is contracted, resulting in the replacement of nodes M and N by a single node, L.

The Maven project for this lab is located in the following svn repository:

- `https://svn.rice.edu/r/comp322/turnin/S15/`*`NETID`*`/hw_5_boruvka`

Please use the subversion command-line client to checkout the project into appropriate directories locally. You can compile the code and run the unit tests by typing "`mvn clean compile test`" as usual.

5.2 Reference Sequential Implementation of Boruvka's algorithm

The files `SeqBoruvka.java`, `SeqComponent.java`, `SeqEdge.java` contain a sequential implementation of Boruvka's algorithm in Java. To run the code with the simplest input provided, type "`mvn clean compile exec:exec -PSeqBoruvka0`".

You should obtain an output that looks as follows:

```
...
     Inputs:
           # nodes loaded = 264346
           # edges loaded = 733846
             Total weight = 9.4908549E8
     Outputs:
           # result edges = 264345
      result total weight = 3.10150315E8
             SeqBoruvka          Iteration-0:   495.648 ms
...
```

Note that when running any other sequential/parallel implementation of an MST algorithm with the same input, the output MST should always have 264,345 edges (which is 1 fewer than the number of nodes in the input graph, 264346). In addition, the cost should also equal 3.1E8 after rounding (there may be differences in less significant digits due to floating-point rounding errors).

The input file, `Boruvka USA-road-d.NY.gr.gz`, was obtained from `http://www.dis.uniroma1.it/challenge9/download.shtml`. It contains a distance graph that represents the roadways in New York City, where each

edge is labeled with the distance between two points. Other sample inputs can also be obtained from the above URL. The format for specifying the graph is described in `http://www.dis.uniroma1.it/challenge9/format.shtml#graph`.

```
1  // START OF EDGE CONTRACTION ALGORITHM
2  Component n = null;
3  while (!nodesLoaded.isEmpty()) {
4    // poll() removes first element from the nodesLoaded work-list
5    n = nodesLoaded.poll();
6    if (n.isDead)
7      continue; // node n has already been merged
8    Edge e = n.getMinEdge(); // retrieve n's edge with minimum cost
9    if (e == null)
10     break; // done - we've contracted the graph to a single node
11   Component other = e.getOther(n);
12   // mark this component dead, i.e. we have visited this component
13   other.isDead = true;
14   n.merge(other, e.weight); // Merge node other into node n
15   nodesLoaded.add(n); // Add newly merged n back in the work-list
16 }
17 // END OF EDGE CONTRACTION ALGORITHM
```

Listing 1: Sequential version of Boruvka's MST algorithm

Listing 1 shows the main code for Boruvka's algorithm from `SeqBoruvka.java`. The field, `nodesLoaded`, is a reference to a *work-list* that (in the sequential version) is implemented as a `LinkedList` of `Component` objects, where each component refers to a collection of one or more more nodes that have been contracted. Initially, each node is in a component by itself. This implementation assumes that the graph contains no self-loops (*i.e.,* no edge from a node to itself) and that the graph is connected.

Each iteration of the `while` loop in line 5 removes a component, `n`, from the work-list[1]. Line 6 skips the component if it was marked as *dead i.e.,* if it was merged with another component. Line 8 retrieves edge `e` with minimum cost connected to component `n`. If `n` has no adjacent edges, then we must have collapsed all the nodes into a single component and we can exit the loop using the `break` statement in line 10. Line 11 sets `other` to the component connected to `n` at the other end of `e`. Lines 13 and 14 do the necessary book-keeping to merge `other` into `n`. Finally, line 15 adds the newly contracted component, `n`, back into the work-list.

5.3 Your Assignment: Parallel Minimum Spanning Tree Algorithm using Java Threads

Your assignment is to design and implement a parallel algorithm for finding the minimum spanning tree of an undirected graph using whatever parallel Java primitives you have learned in this class — standard Java threads, `java.util.concurrent` libraries, HJ library, or some (carefully selected) combination thereof. You can use the sequential implementation of Boruvka's algorithm described in Section 2.2 as a starting point, and you are free to modify any files in the `edu.rice.comp322.boruvka.parallel` that you choose. Your homework will be evaluated as follows. A single implementation should be submitted for both parts 1 and 2:

1. **[Correctness evaluation (20 points)]** The following two observations about the algorithm in Listing 1 can provide insights on how to parallelize the algorithm:
   - The order in which components are removed (line 5) from and inserted (line 15) in the work-list is not significant i.e., the work-list can be implemented as any unordered collection.
   - If two iterations of the `while` loop work on disjoint (`n`, `other`) pairs, then the iterations can be executed in parallel using a thread-safe implementation of the work-list that allows inserts and removes to be invoked in parallel.

[1]Line numbers here refer to Listing 1, and not the code in `SeqBoruvka.java`.

You will get full credit for this part of the homework with any correct implementation that exploits parallelism among while-loop iterations as indicated above, even if the implementation incurs large overheads and runs slower than the sequential version. Your program should return a current MST solution for the given test input, when executed with at least 2 threads. Include the output from one such run (i.e. the output for a single iteration) in your report.

2. [**Performance evaluation on Sugar compute nodes (15 points)**] The goal of this part of the homework is to evaluate the performance of your parallel implementation and compare it to that of the sequential version. Measure the performance of your program when using 1, 2, 4, and 8 threads, and compare it with the performance of the sequential version that was provided. There are six input files provided (`USA-road-d.NY.gr.gz`, `USA-road-d.BAY.gr.gz`, `USA-road-d.COL.gr.gz`, `USA-road-d.FLA.gr.gz`, `USA-road-d.NW.gr.gz`, and `USA-road-d.NE.gr.gz`), each with progressively larger graphs (i.e. larger numbers of input edges). You may choose any one of these input files to present your results. Clearly mention the input files you have chosen for your results in you report along with the relevant outputs.

   You will be graded on the performance obtained by your parallel implementation with 2, 4 and 8 threads, relative to the performance of the same implementation using 1 thread. Getting good speedups when parallelizing Boruvka's algorithm can be challenging. However, you should see some performance improvements when going from 1 to 2 or 4 threads. Do not be surprised if you see performance degradations with 8 threads.

   The 15 points for performance evaluation will be broken down as follows:

   1-thread execution time — 3 points if the 1-thread execution time is $\leq 2\times$ the sequential time, 1 point if it terminates correctly (regardless of performance), and 0 points if the 1-thread execution does not terminate with the correct answer.

   2-thread execution time — 4 points if the 2-thread execution time is $\leq 2/3\times$ the 1-thread time (speedup ≥ 1.5), 2 points if it is $\leq$ the 1-thread time (speedup ≥ 1), 1 point if it terminates correctly (regardless of performance), and 0 points if the 2-thread execution does not terminate with the correct answer.

   4-thread execution time — 4 points if the 4-thread execution time is $\leq 4/7\times$ the 1-thread time (speedup ≥ 1.75), 2 points if it is $\leq$ the 1-thread time (speedup ≥ 1), 1 point if it terminates correctly (regardless of performance), and 0 points if the 4-thread execution does not terminate with the correct answer.

   8-thread execution time — 4 points if the 8-thread execution time is $\leq 1/2\times$ the 1-thread time (speedup ≥ 2), 2 points if it is $\leq$ the 1-thread time (speedup ≥ 1), 1 point if it terminates correctly (regardless of performance), and 0 points if the 8-thread execution does not terminate with the correct answer.

3. [**Homework report (15 points)**]
   You should submit a brief report summarizing the design and implementation of the parallel constructs used in your parallel MST algorithm, and explain why you believe that the implementation is correct, including why it is free of data races, deadlocks, and livelocks.

   Your report should also include the following measurements for both parts 1 and 2:

   (a) Performance of the sequential version with your input of choice.

   (b) Performance of the parallel version with the same input, executed with 1, 2, 4 and 8 threads.

   Please place the report file(s) in the top-level `hw_5_boruvka` directory. Remember to commit all your source code into the subversion repository during your assignment submission.

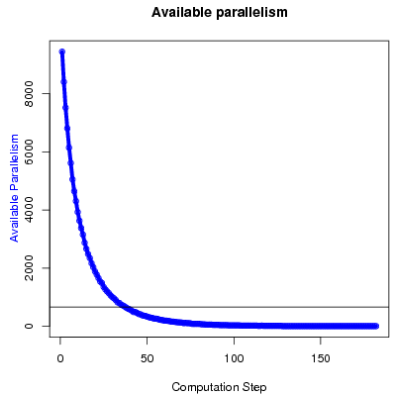

Figure 3: Available parallelism in parallel Boruvka Minimum Spanning Tree algorithm (source: Galois MST description).

5.4 Some Implementation Tips

Here are some tips to keep in mind for this homework:

- Given the large overhead of creating Java threads, you should try and get by with just creating a small number of threads (*e.g.,* one thread per processor core), or a small number of workers if you use HJlib. Each thread can then share the work of the while-loop in Listing 1.
- Figure 3 illustrates how the available parallelism decreases as more and more merge steps are performed. This suggests that the actual parallelism exploited (e.g., number of threads) should be reduced as the contracted graph reaches certain threshold sizes.
- The work-list is implemented as a `LinkedList` in the sequential version of `SeqBoruvka.java`. It will need to be implemented as a thread-safe collection when multiple `while` iterations are executed in parallel. Two `java.util.concurrent` classes that can be useful for this purpose are `ConcurrentLinkedQueue` and `ConcurrentHashMap`. You're welcome to implement your own data structure if you choose *e.g.,* by using `AtomicInteger` operations to manage indexing in a shared array. (Note that the number of insertions in the work-list is bounded by the initial number of nodes.)
- In addition to changes in `ParBoruvka.java`, you may find it convenient to modify `ParComponent.java` to add some form of mutual exclusion constructs to manage cases when two threads collide on the same node when trying to contract a pair. For mutual exclusion, you can try using Java's built-in locks with the `synchronized` statement and the use of `wait-notify` operations as needed, or you can explicitly allocate a `java.util.concurrent.locks.ReentrantLock` for each node/component. One advantage of `ReentrantLock` is that it supports a `tryLock()` method that allows a thread to query the status of a lock without blocking on the request. The potential disadvantage is that it incurs extra space overhead, which may indirectly impact execution time.

You can run your programs locally by using the following maven command: `mvn clean compile exec:exec -PBoruvka0`. There are variants for this command for all six input files (`-PBoruvka0`, `-PBoruvka1`, ..., `-PBoruvka5`). The commands are also available in the README file in your project.

To run the program using 8 cores on STIC, use the provided `slurm` file (myjob.slurm) and commands as available in the README. Please note that when you are logged in STIC, you are logged in a `login node`. These nodes are intended for users to compile software and prepare data files. You should run your program on a `compute node` by submitting a job into the job queue.