Homework 5: due by 11:55pm on Monday, April 21, 2014 (Total: 100 points)

Instructor: Vivek Sarkar

All homeworks should be submitted in a directory named hw_5 using the turn-in script. In case of problems using the script, you should email a zip file containing the directory to comp322-staff@mailman.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)

Please submit your solution to this assignment in a plain text file named `hw_5_written.txt` *in the submission system. Syntactic errors in program text will not be penalized in the written assignment 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 29, 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.

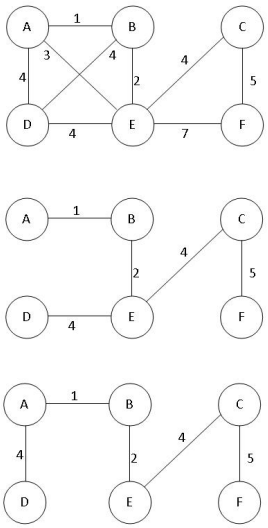

Figure 1: Two possible Minimum Spanning Trees (MSTs) for a given undirected graph (source: `http://en.wikipedia.org/wiki/Minimum_spanning_tree`)

2 Programming Assignment: Minimum Spanning Tree of an Undirected Graph (75 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.
>
> One example would be a cable TV company laying cable to a new neighborhood. If it is constrained to bury the cable only along certain paths, then there would be a graph representing which points are connected by those paths. Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects to every house. There might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost.

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).

Figure 2: Demonstration of edge contraction

2.1 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@mailman.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 `http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/boruvkas_algorithm`.

> 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.

2.2 Reference Sequential Implementation of Boruvka's algorithm

The files `Boruvka.java`, `Component.java`, `Edge.java`, `Pair.java` contain a sequential implementation of Boruvka's algorithm in Java. You can compile the code by typing "`javac *.java`" as usual. To run the code with the test input provided, type "`java -server -Xmx10g Boruvka USA-road-d.NY.gr.gz`". You should obtain an output that looks as follows:

```
Sequential version (seq): Reading /users/COMP322/hw_5/USA-road-d.NY.gr.gz
loaded 264346 - edges 733846- total weight 9.4908549E8
Finished with edges = 264345,  total weight = 3.10134454E8, time = 0.886908 seconds
```

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`.

Listing 1 shows the main code for Boruvka's algorithm from `Boruvka.java`. The field, `Loader.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 2 removes a component, `n`, from the work-list[1]. Line 4 skips the component if it was marked as *dead i.e.,* if it was merged with another component. Line 5 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 6. Line 7 sets `other` to the component connected to `n` at the other end of `e`. Lines 8 and 9 do the necessary book-keeping to merge `other` into `n`. Finally, line 10 adds the newly contracted component, `n`, back into the work-list.

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

Listing 1: Sequential version of Boruvka's MST algorithm

2.3 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 that you choose. Your homework will be evaluated are as follows. A single implementation should be submitted for both parts 1 and 2:

1. **[Correctness evaluation (30 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 2) from and inserted in (line 10) 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.

   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 in your report.

2. **[Performance evaluation on Sugar compute nodes (25 points)]** The goal of this part of the homework is to evaluate the performance of your parallel implementation and compare it to that of

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

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. Include all outputs in your report.

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 25 points for performance evaluation will be broken down as follows:

1-thread execution time — 4 points if the 1-thread execution time is $\leq 2\times$ the sequential time, 2 points 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 — 7 points if the 2-thread execution time is $\leq 2/3\times$ the 1-thread time (speedup ≥ 1.5), 4 points if it is $\leq$ the 1-thread time (speedup ≥ 1), 2 points 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 — 7 points if the 4-thread execution time is $\leq 4/7\times$ the 1-thread time (speedup ≥ 1.75), 4 points if it is $\leq$ the 1-thread time (speedup ≥ 1), 2 points 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 — 7 points if the 8-thread execution time is $\leq 1/2\times$ the 1-thread time (speedup ≥ 2), 4 points if it is $\leq$ the 1-thread time (speedup ≥ 1), 2 points 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 (20 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 the default input

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

   Please place the report file(s) in the top-level `hw_5` directory.

2.4 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 `Boruvka.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.,*

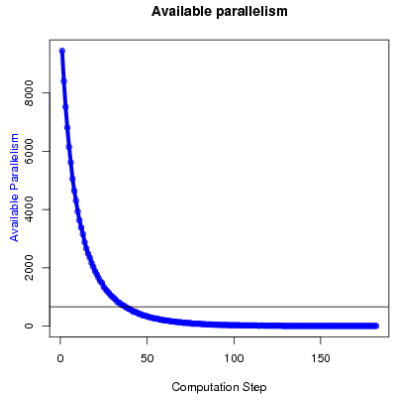

Figure 3: Available parallelism in parallel Boruvka Minimum Spanning Tree algorithm (source: `http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/mst`)

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 `Boruvka.java`, you may find it convenient to modify `Component.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.

2.5 Use of PBS scripts (Optional)

We have provided a PBS script file to simplify running your programs on SUGAR. It is set up to run the sequential implementation provided for this homework, and will need to be updated if you modify the command-line interface for the Boruvka main program (*e.g.,* by adding an extra command-line argument for the number of threads to use.) See `https://docs.rice.edu/confluence/display/ITDIY/Getting+Started+on+SUGAR` for documentation on using PBS scripts.

As an example, issuing the command "`qsub hw_5.pbs`" from a SUGAR login node will output a line as follows that contains your job submission id:

```
5393436.sugarman.rcsg.rice.edu
```

The job will automatically be executed on a compute node when one becomes available, without your having to issue a `qsub -I` command and wait for an interactive session. When the job completes, it will create two files, `hw_5.pbs.o5393436` and `hw_5.pbs.e5393436`, containing the `stdout` and `stderr` outputs respectively from your program. The `stdout` file will contain the output and timing from your program run as shown above, along with some other diagnostic outputs from the `pbs` script. Using this script is optional, but it may be convenient if the waiting times for interactive sessions grow too long.