4.1 Lecture Summary

4.1 Optimistic Concurrency

Lecture Summary: In this lecture, we studied the *optimistic concurrency* pattern, which can be used to improve the performance of concurrent data structures. In practice, this pattern is most often used by experts who implement components of concurrent libraries, such as *AtomicInteger* and *ConcurrentHashMap*, but it is useful for all programmers to understand the underpinnings of this approach. As an example, we considered how the *getAndAdd()* method is typically implemented for a shared *AtomicInteger* object. The basic idea is to allow multiple threads to read the existing value of the shared object *(curVal)* without any synchronization, and also to compute its new value after the addition *(newVal)* without synchronization. These computations are performed *optimistically* under the assumption that no interference will occur with other threads during the period between reading *curVal* and computing *newVal*. However, it is necessary for each thread to confirm this assumption by using the *compareAndSet()* method as follows. (c*ompareAndSet()* is used as an important building block for optimistic concurrency because it is implemented very efficiently on many hardware platforms.)

The method call *A.compareAndSet(curVal, newVal)* invoked on *AtomicInteger A* checks that the value in *A* still equals *curVal*, and, if so, updates *A*'s value to *newVal* before returning *true*; otherwise, the method simply returns *false* without updating *A*. Further, the *compareAndSet()* method is guaranteed to be performed atomically, as if it was in an object-based isolated statement with respect to object *A*. Thus, if two threads, T1 and T2 call *compareAndSet()* with the same *curVal* that matches *A*'s current value, only one of them will succeed in updating *A* with their *newVal*. Furthermore, each thread will invoke an operation like *compareAndSet()* repeatedly in a loop until the operation succeeds. This approach is guaranteed to never result in a deadlock since there are no blocking operations. Also, since each call *compareAndSet()* is guaranteed to eventually succeed, there cannot be a livelock either. In general, so long as the contention on a single shared object like *A* is not high, the number of calls to *compareAndSet()* that return *false* will be very small, and the optimistic concurrency approach can perform much better in practice (but at the cost of more complex code logic) than using locks, isolation, or actors.

Optional Reading:

1. Wikipedia article on Optimistic concurrency control

2. Documentation on Java's AtomicInteger class

4.2 Lecture Summary

4.2 Concurrent Queues

Lecture Summary: In this lecture, we studied *concurrent queues*, an extension of the popular queue data structure to support concurrent accesses. The most common operations on a queue are *enq(x)*, which enqueues object *x* at the end (tail) of the queue, and *deq()* which removes and returns the item at the start (head) of the queue. A correct implementation of a concurrent queue must ensure that calls to *enq()* and *deq()* maintain the correct semantics, even if the calls are invoked concurrently from different threads. While it is always possible to use locks, isolation, or actors to obtain correct but less efficient implementations of a concurrent queue, this lecture illustrated how an expert might implement a more efficient concurrent queue using the optimistic concurrency pattern.

A common approach for such an implementation is to replace an object reference like *tail* by an *AtomicReference*. Since the *compareAndSet()* method can also be invoked on *AtomicReference* objects, we can use it to support (for example) concurrent calls to *enq()* by identifying which calls to *compareAndSet()* succeeded, and repeating the calls that failed. This provides the basic recipe for more efficient implementations of *enq()* and *deq()*, as are typically developed by concurrency experts. A popular implementation of concurrent queues available in Java is *java.util.concurent.ConcurrentLinkedQueue.*

Optional Reading:

1. Documentation on Java's AtomicReference class
2. Documentation on Java's ConcurrentLinkedQueue class

4.3 Lecture Summary

4.3 Linearizability

Lecture Summary: In this lecture, we studied an important correctness property of concurrent objects that is called *Linearizability*. A concurrent object is a data structure that is designed to support operations in parallel by multiple threads. The key question answered by linearizability is what return values are permissible when multiple threads perform these operations in parallel, taking into account what we know about the expected return values from those operations when they are performed sequentially. As an example, we considered two threads, T1 and T2, performing *enq(x)* and *enq(y*) operations in parallel on a shared concurrent queue data structure, and considered what values can be returned by a *deq()* operation performed by T2 after the call to *enq(y)*. From the viewpoint of linearizability, it is possible for the *deq()* operation to return item *x* or item *y*.

One way to look at the definition of linearizability is as though you are a lawyer attempting to "defend" a friend who implemented a concurrent data structure, and that all you need to do to prove that your friend is "not guilty" (did not write a buggy implementation) is to show one scenario in which all the operations return values that would be consistent with a sequential execution by identifying logical

moments of time at which the operations can be claimed to have taken effect. Thus, if *deq()* returned item *x* or item *y* you can claim that either scenario is plausible because we can reasonably assume that *enq(x)* took effect before *enq(y)*, or vice versa. However, there is absolutely no plausible scenario in which the call to *deq()* can correctly return a code/exception to indicate that the queue is empty since at least *enq(y)* must have taken effect before the call to *deq()*. Thus, a goal for any implementation of a concurrent data structure is to ensure that all its executions are linearizable by using whatever combination of constructs (e.g., locks, isolated, actors, optimistic concurrency) is deemed appropriate to ensure correctness while giving the maximum performance.

Optional Reading:

1. Wikipedia article on the Linearizability

4.4 Lecture Summary

4.4 Concurrent HashMap

Lecture Summary: In this lecture, we studied the *ConcurrentHashMap* data structure, which is available as part of the *java.util.concurrent* standard library in Java. A *ConcurrentHashMap* instance, *chm*, implements the *Map* interface, including the *get(key)* and *put(key, value)* operations. It also implements additional operations specified in the *ConcurrentMap* interface (which in turn extends the *Map* interface); one such operation is *putIfAbsent(key, value)*. The motivation for using *putIfAbsent()* is to ensure that only one instance of *key* is inserted in *chm*, even if multiple threads attempt to insert the same key in parallel. Thus, the semantics of calls to *get()*, *put()*, and *putIfAbsent()* can all be specified by the theory of linearizability studied earlier. However, it is worth noting that there are also some aggregate operations, such as *clear()* and *putAll()*, that cannot safely be performed in parallel with *put(), get()* and *putIfAbsent()*.

Motivated by the large number of concurrent data structures available in the *java.util.concurrent* library, this lecture advocates that, when possible, you use libraries such as *ConcurrentHashMap* rather than try to implement your own version.

Optional Reading:

1. Documentation on Java's ConcurrentHashMap class

2. Wikipedia article on Java's ConcurrentMap interface

4.5 Lecture Summary

4.5 Minimum Spanning Tree Example

Lecture Summary: In this lecture, we discussed how to apply concepts learned in this course to design a concurrent algorithm that solves the problem of finding a *minimum-cost spanning tree* (MST) for an undirected graph. It is well known that undirected graphs can be used to represent all kinds of networks, including roadways, train routes, and air routes. A spanning tree is a data structure that contains a subset of edges from the graph which connect all nodes in the graph without including a cycle. The cost of a spanning tree is computed as the sum of the weights of all edges in the tree.

The concurrent algorithm studied in this lecture builds on a well-known sequential algorithm that iteratively performs *edge contraction* operations, such that given a node *N1* in the graph, *GetMinEdge(N1)* returns an edge adjacent to *N1* with minimum cost for inclusion in the MST. If the minimum-cost edge is (*N1*,*N2*), the algorithm will attempt to combine nodes *N1* and *N2* in the graph and replace the pair by a single node, *N3*. To perform edge contractions in parallel, we have to look out for the case when two threads may collide on the same vertex. For example, even if two threads started with vertices *A* and *D*, they may both end up with *C* as the neighbor with the minimum cost edge. We must avoid a situation in which the algorithm tries to combine both *A* and *C* and *D* and *C*. One possible approach is to use unstructured locks with calls to *tryLock()* to perform the combining safely, but without creating the possibility of deadlock or livelock situations. A key challenge with calling *tryLock()* is that some fix-up is required if the call returns false. Finally, it also helps to use a concurrent queue data structure to keep track of nodes that are available for processing.

Optional Reading:

1. Wikipedia article on Borvka’s algorithm for finding a minimum cost spanning tree of an undirected graph

Mini Project 4: Parallelization of Boruvka's Minimum Spanning Tree Algorithm

---

miniproject_4.zip

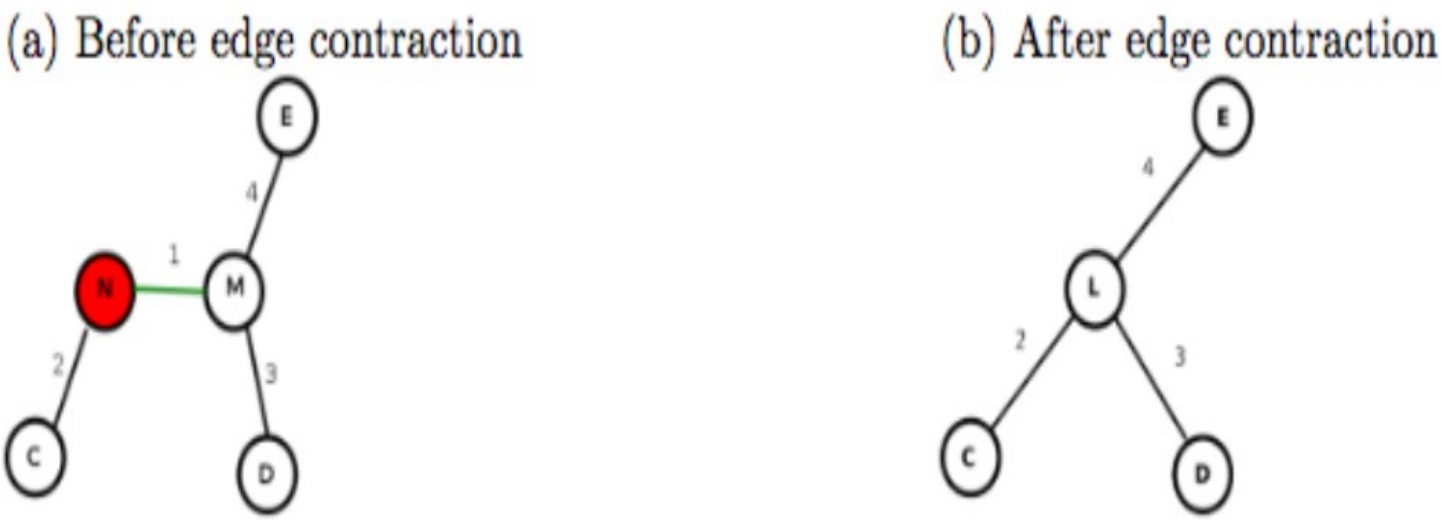
Project Goals and Outcome

In this assignment, we will focus on parallelizing a reference sequential version of Boruvka's algorithm. The following summary of Boruvka's sequential algorithm is from the Galois project's description of Boruvka's 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 below, the edge connecting nodes M and N is contracted, resulting in the replacement of nodes M and N by a single node, L.



In this assignment, we'll explore the use of concurrent queues, threads, and unstructured locks, with calls to *tryLock()*, to produce a concurrent and data race-free implementation of Boruvka's algorithm. Your parallel implementation will be evaluated on real datasets representing the road networks of several United States regions. These datasets are pre-packaged with the miniproject_4.zip above.

Project Setup

Please refer to Mini-Project 0 for a description of the build and testing process used in this course.

Once you have downloaded and unzipped the project files using the gray button labeled miniproject_4.zip at the top of this description, you should see the project source code file at

miniproject_4/src/main/java/edu/coursera/concurrent/ParBoruvka.java

and the project tests in

miniproject_4/src/test/java/edu/coursera/concurrent/BoruvkaPerformanceTest.java

It is recommended that you review the demo video and lecture videos for this week before starting this assignment. An example sequential implementation is also provided in SeqBoruvka.java.

Project Instructions

Your modifications should be made entirely inside of ParBoruvka.java. You should not change the signatures of any public or protected methods inside of ParBoruvka.java, but you can edit the method bodies and add any new methods that you choose. We will use our copy of

BoruvkaPerformanceTest.java in the final grading process, so do not change that file or any other file except ParBoruvka.java.

Your main goal for this assignment is to complete the computeBoruvka method at the top of ParBoruvka. The testing infrastructure will call computeBoruvka from a certain number of Java threads. You do not need to create any additional threads if you do not wish to (indeed, it is recommended that you do not). computeBoruvka is passed two objects:

1. nodesLoaded: A ConcurrentLinkedQueue object containing a list of all nodes in the input graph for which you are to compute a minimum spanning tree using Boruvka's algorithm. Because this Queue<ParComponent> object is a ConcurrentLinkedQueue, it is safe to access nodesLoaded from multiple threads concurrently without additional synchronization.
2. solution: A BoruvkaSolution object on which you will call BoruvkaSolution.setSolution once your parallel Boruvka implementation has collapsed the input graph down to a single component. You must only call setSolution once, and the testing infrastructure will use the provided result to verify your output.

Inside of ParBoruvka, there are two additional inner classes: ParComponent and ParEdge. ParComponent represents a single component in the graph. A component may be a singleton node, or it may be formed from collapsing multiple nodes into each other. A ParEdge represents an edge between two ParComponents. You may not change the signatures for the existing methods in ParComponent or ParEdge. However, you are free to modify their method bodies, to add new methods, or to add new fields. An efficient implementation will likely require modifications to ParComponent and/or ParEdge.

Project Evaluation

Your assignment submission should consist of only the ParBoruvka.java file that you modified to implement this mini-project. As before, you can upload this file through the assignment page for this mini-project. After that, the Coursera autograder will take over and assess your submission, which includes building your code and running it on one or more tests. Your submission will be evaluated on Coursera’s auto-grading system using 4 CPU cores. Note that the performance observed for tests on your local machine may differ from that on Coursera's auto-grading system, but that you will only be evaluated on the measured performance on Coursera. Also note that for all assignments in this course you are free to resubmit as many times as you like. See the Common Pitfalls page under Resources for more details. Please give it a few minutes to complete the grading. Once it has completed, you should see a score appear in the “Score” column of the “My submission” tab based on the following rubric:

- 50% - Performance on 4 cores while processing the FLA dataset
- 50% - Performance on 4 cores while processing the NE dataset

NOTE: Unlike past mini-projects, this mini-project loads datasets from files. If during your local testing you encounter java.io.FileNotFoundException errors, modify the paths to USA-road-d.FLA.gr.gz and USA-road-d.NE.gr.gz at the top of BoruvkaPerformanceTest.java to be the absolute paths on your local machine to the respective files. These files are provided with the mini-project ZIP, under src/main/resources/boruvka. This will not affect grading on the Coursera auto-grader.