Xuan Thuong Le

Alex Hankin

CSC 254 Assignment 6: Concurrency

Write-Up (and README)

Runtime instructions:   
  
We did not change any of the means in order to run the program. It should still compile and run as a normal java file, and still take in the same arguments and parameters as outlined in the assignment (-n for vertices, -t for threads, etc.) Using “-a 0” to remove animation is recommended. We were running into trouble with the program taking a longer time to shut down after the program had calculated elapsed time, if this happens use the “Ctrl-C” command.

List of Files:

Writeup.pdf: This file

SSSP.java: Where a majority of the code is kept. Most of the classes and methods went untouched as the code itself was already functional as a sequential program, but we needed to manipulate aspects of the Surface class in order to make delta stepping parallel. Methods involving Animation, UI, manipulation of the arguments, Worker thread, as well as aspects of the surface class that involved creating vertices and edges were left untouched. However, the primary DeltaSolve() algorithm and surrounding functions were altered to make parallyzation. More specifics can be seen in the code/comments itself, but as general summary it mainly revolved around our new thread that we created for this purpose, deltaThread. To do this in deltaSolve we create a set number of deltaThreads in an ArrayList and gave each Thread an ArrayList of vertices which is essentially an apx. even split of the total vertices. Within the threads we apply the delta stepping algorithm where we loop through our buckets as normal. Once a bucket has been fully examined, we encounter a barrier, this is to ensure that each thread is working on the same respective bucket, and no threads jumps to the next bucket without the rest. After, a second barrier is encountered, this is after a check is performed to see if both all buckets are empty, and all message queues are empty for every thread. If this is the case, we are done and the program stops, else the program continues. Message queues are also applied in order for the threads to interact with one another.

Coordinator.java: This code has remained unchanged from the original sample code provided to us from the assignment. However, it remained essentially to our management of the threads that we handled. We also directly referenced it once when we registered and unregistered our newly created thread, “deltaThread”.

Time analysis

Disclaimer: Due to time constraints we were only able to have a functional build of our code ready in time for on 20-min session on the nodes. We did as much testing as we could given the time, with hopefully enough data being acquired. You are absolutely welcome to test our code on the nodes when grading. We also had one bug correction noticed after the testing regarding barrier. In between barriers we were analyzing the wrong set of buckets when checking for emptiness. This was adjusted and fixed however, due to the similarity of the values of these two variables in our code, it didn’t adjust the times greatly, and so we are reporting on the original time data calculated on the csug node. Please feel free to test the time calculation when grading.

Number of Threads vs Time

For the sake of ease of legibility there are two execution time graphs, one containing Dijkstra’s runtime average (the point at apx. 1.2 seconds at 0 threads) and one with Dijkstra as a reference.

There are a few things that we can take away from these two graphs. Namely for one the delta stepping methods works far superior to Dijkstra in this case, as the difference in the structure of the two graphs alone show. Second, there’s also the fact that there is not much runtime difference in between thread calculations. Now, this could be attributed to either, not enough threads, not enough vertices, or sufficient optimization on our part, for the lack of variance. However, some patterns were noticeable, namely, that around 6 or 8 threads the time seemed to reach its maximum in every series of tests, while continuing to drop afterward. Presumably, on a more powerful computer, that if we extrapolate, we can keep lowering this run time with more and more threads.

Sequential/Parallel Time Comparison

Unfortunately we do not appear to have optimized this aspect very well, as there is no really noticeable improvement among threads once the method is actually paralyzed to begin with. We don’t necessarily know particularly why, as far as we’re concerned we have a functioning program. However, we may have done too much working with the filler, that being aspects such as searching through hash maps and buckets for checks. We check through the hash maps often for update queues, as well as buckets and queues for if our algorithm is done. Thus, even if we achieve a faster algorithm with more threads the amount of time searching and checking grows with number of threads, and probably balances it out. When looking to optimize our code we will almost definitely start with trying to improve how we handle barriers and message queues.