12/15/2017
Aslan Bakri, Won Kuk Lee
abakri@u.rochester.edu
wlee33@u.rochester.edu

CSC 254 Programming Languages
Assignment 6: Concurrency

How to Run:

To run with 1-48 threads:

*In order to use shell scripts compiledata and compile, please run the following commands:
chmod +x compile
chmod +x compiledata

Run with ./compiledata
To change # of vertices, edit compiledata and change vertices value to desired number.
To change seed, edit compiledata and change seed value to desired number.

To run once with custom thread number:
Run with ./compile
To change # of vertices, edit compile and change vertices value to desired number.
To change seed, edit compile and change seed value to desired number.
To change # of threads, edit compile and change threads to desired number.

Methodology

Creating Threads:

We first created a WorkerP class that extends Thread and then changed the DeltaSolve to DeltaSolveP which creates all of the WorkerP objects. This method assigns these workers vertices and starts them. WorkerP initializes on creation and holds a hashset of vertices that it owns based on the indices given to its creation by DeltaSolveP. Each WorkerP object contains a ConcurrentLinkedQueue so other Worker threads can communicate with one another.

Running Delta Stepping:

The run() of WorkerP calculates the DeltaSolve algorithm given to us. However, instead of looping around the buckets, it only loops through the buckets once. Furthermore, it also calls a hold to the CyclicBarrier that we created whenever it wants to satisfy all the requests sent to WorkerP by the ConcurrentLinkedQueue. This keeps going until all the Worker threads complete all the light relaxations and then continues to the heavy relaxations after which the barrier is held again to satisfy the new heavy relaxation requests for each WorkerP ConcurrentLinkedQueue. Each Worker contains a satisfyRequest function that empties its queue and relaxes all the vertices in that queue.

Data Results

Delta Stepping vs. Dijkstras for 20,000 Vertices

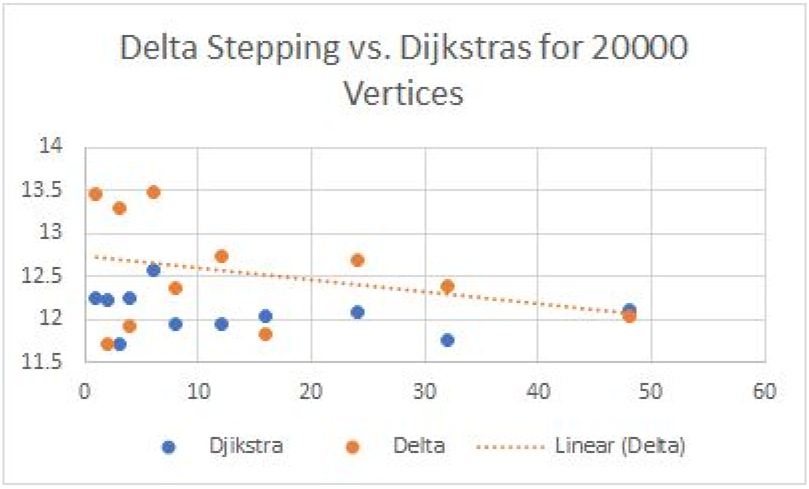
Threads	Djikstra	Delta	Vertices	Speedup
1	12.239	13.457	20000	0.909
2	12.222	11.717	20000	1.043
3	11.707	13.297	20000	0.880
4	12.247	11.927	20000	1.027
6	12.583	13.483	20000	0.933
8	11.946	12.371	20000	0.966
12	11.939	12.736	20000	0.937
16	12.027	11.831	20000	1.017
24	12.091	12.696	20000	0.952
32	11.748	12.378	20000	0.949
48	12.098	12.03	20000	1.006

Delta Stepping vs. Dijkstras for 100 Vertices

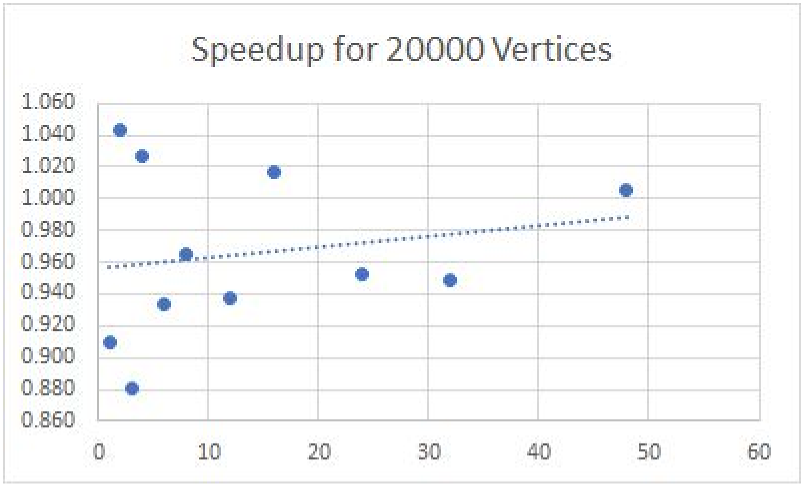
Threads	Djikstra	Delta	Vertices	Speedup
1	0.016	0.02	100	0.800
2	0.013	0.022	100	0.591
3	0.015	0.023	100	0.652
4	0.012	0.033	100	0.364
6	0.012	0.033	100	0.364
8	0.013	0.031	100	0.419
12	0.019	0.052	100	0.365
16	0.013	0.065	100	0.200

24	0.018	0.081	100	0.222
32	0.015	0.102	100	0.147
48	0.013	0.138	100	0.094

<u>Data Analysis</u>



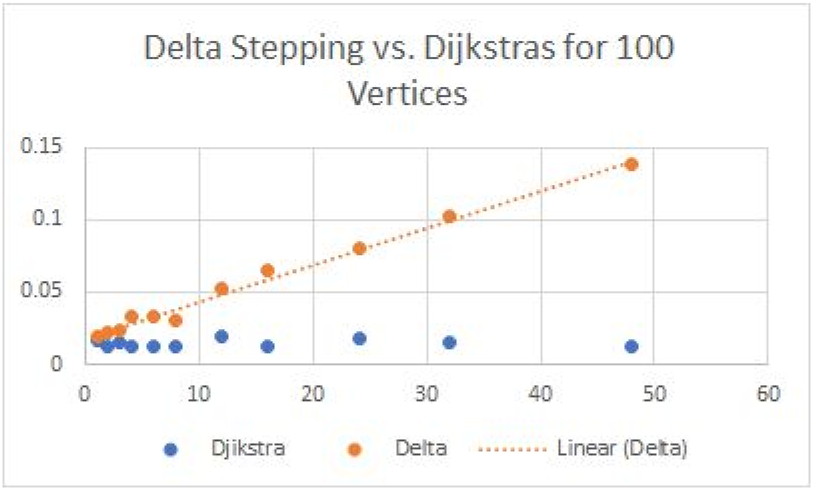
From the above graph, we can see that as the threads increased and the trend of the time elapsed for Delta Stepping algorithm decreased. This shows that as the number of threads increased, the runtime actually decreased, showing an improvement.



For the same data, we can see that the speedup also increases, from approx 0.95 to 0.99, which is a small change, but definitely an improvement. We were not able to obtain a speedup of k with k threads, though we did find an increasing slope. Clearly something about our implementation of the parallel Delta Stepping was flawed because of this large disparity between expected and real speedup. However, it does show the benefits of parallel programming when computing algorithms with high runtime. We suspect that the reason why we did not have an ideal speedup is because we forced the threads to run into a barrier at the end of the light relaxation loop. As a result, this may end up making the light relaxations run at almost identical runtime to Dijkstra's algorithm-- definitely something that we must make sure to avoid next time we implement concurrency and parallelism.

Another interesting find is that generally Delta Stepping ran faster with an even number of threads, as can be seen with the test cases for 1, 2, 3, and 4 threads. This is significant because it hints at the nature of thread communication. We experienced this first hand in workshop with the “pen game”.

We discovered something else when we were testing with a very small number of vertices. In the following experiment we tested the Delta Stepping runtime against the Dijkstra’s runtime for 100 vertices.



As can be seen in the above graph, although we generally expect parallel algorithms to perform better as we add more threads, in this case it actually did significantly worse as more threads were added. We think that this is due to the processing, initialising, and managing of threads and the ConcurrentLinkedQueues taking up a larger proportion of the total number of instructions ran. As a result, this caused the algorithm to run slower as there was an increase in number of threads. This is also reflected in the decreasing rate of speedup as threads increase, shown below.