454 Assignment05 Report Liang Xin

Implementation of parallelizing Delaunay triangulation and Kruskal's algorithm

1. Parallelization of Delaunay triangulation

I created a new thread class **TriThread** extends Thread class which is similar to Worker Thread and use it in the divide-and-conquer step of triangulate(). The strategy is to keep the original sequential triangulate() method and recursively call it as the original code was before, but the difference is to add two more parameter in the triangulate() method to control when to create a new thread and what level to stop creating a new thread. In addition, in each thread run() method will run sequentially, otherwise the resultant mesh cannot stitched correctly.

The two newly-added parameters are boolean isParallel and int level.

The first time to call triangulate() method isParallel is set to be true, and in the TriThread run() method isParallel is set false and passed into the method to guarantee that in each thread it run sequentially.

Parameter level is to control how many threads on which we want to run the code. Because of the design limit, for the triangulation stage, it is only allowed to choose 2, 4, 8, 16, and 32 threads to run with. In other words, the maximal level number is 5.

The recursive call to triangulate() will increase level by one each time, and this increment will be checked by if() clause in the beginning of triangulate() to decide if keep splitting work in more threads.

2. Parallelization of Kruskal's algorithm

Kruskal's algorithm is inherently serial, and it is hard to parallelize this algorithm. I looked up a paper attached at the back of the report and implemented the method described in this paper. I did not follow Dr. Scott's recommendation to implement this parallelization step. This method I used did not help much for two reasons:

1. Creation of the mesh by triangulation takes almost 95% of the total run time, so Kruskal's algorithm takes a very small proportion of the total run time.

2. The method I used might not help much compared the one recommended by Dr. Scott.

The basic idea is if an edge that is going to be examined in a future iteration is found to already form a cycle within the MSF created up to the current iteration, then this edge can be safely discarded immediately. This property essentially allows the out of order rejection of edges.

More specifically, the main thread needs to check only the edges that weren't rejected by the helper threads via checking notCycleEdge if true, thus performing less work compared to the sequential implementation.

And each helper thread keeps looping there and checking edges in its partition if can give rise to a cycle before the main thread enters the partition which has been assigned to this helper thread. Once found one edge is cycle edge, mark the boolean value notCycleEdge as false. When looping through the sorted set of edges, I used the global variable eIndex to remember when to enter a helper partition. However, I found that only eIndex is not enough to force helper thread exit the while() loop, because the number of edges in mesh generated in triangulate() is much bigger than the number of edges Kruskal's algorithm will check to create MST. Thus, in some partitions having longer edges, they can keep looping there forever. So, I added a boolean variable isFinished to tell when to exit the while loop to fix the problem.

At first, I used Hashtable which is synchronized to store the edge detected to be cycle edge; but after Ryan's direction, I realized Hashtable is redundant; notCycleEdge is enough to help the main thread to check if an edge is cycle edge or not.

The command-line argument –t "numThreads" is to control both stages' number of threads.

For triangulate stage, it could be 2, 4, 8, 16, and 32. For Kruskal's algorithm, "numThreads" is the real number of threads, and evenly divide the set of edges into "numThreads" partitions. But only ("numThreads" – 1) threads are woking, because in my code implementation the last threads has no partitions assigned in. However, this would not affect the result much in that the last partition will be taken care of by the main thread.

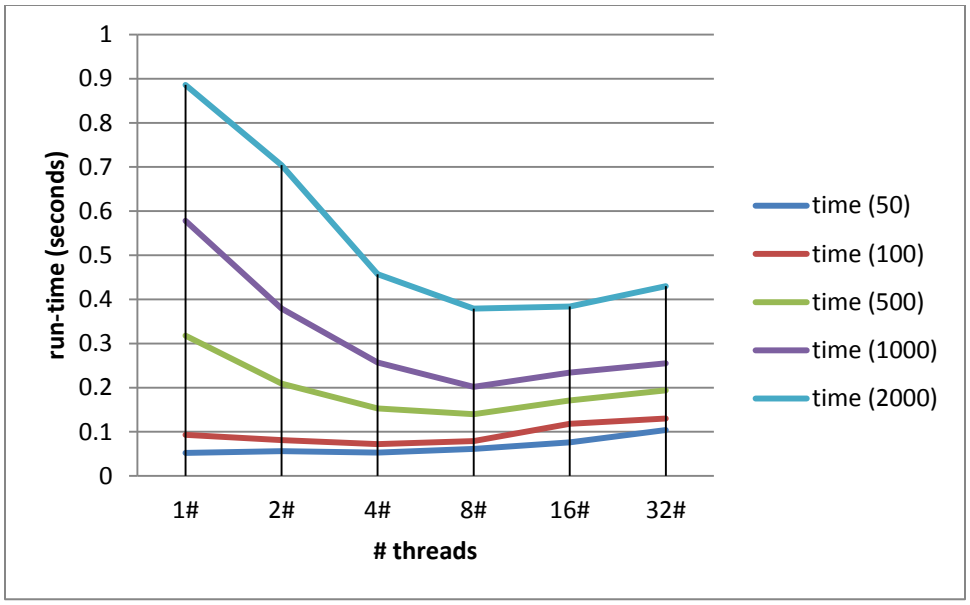
In addition, in order to easily see how the helper thread works. I colored the edges discarded by the helper thread green. For example, you can clearly see while the MST generating by running java MST –a 3 –t 4, the green edges are being drawn simultaneously. I also attached a screen shot of the result in this report.

3. Evaluation and Analysis of parallelization results

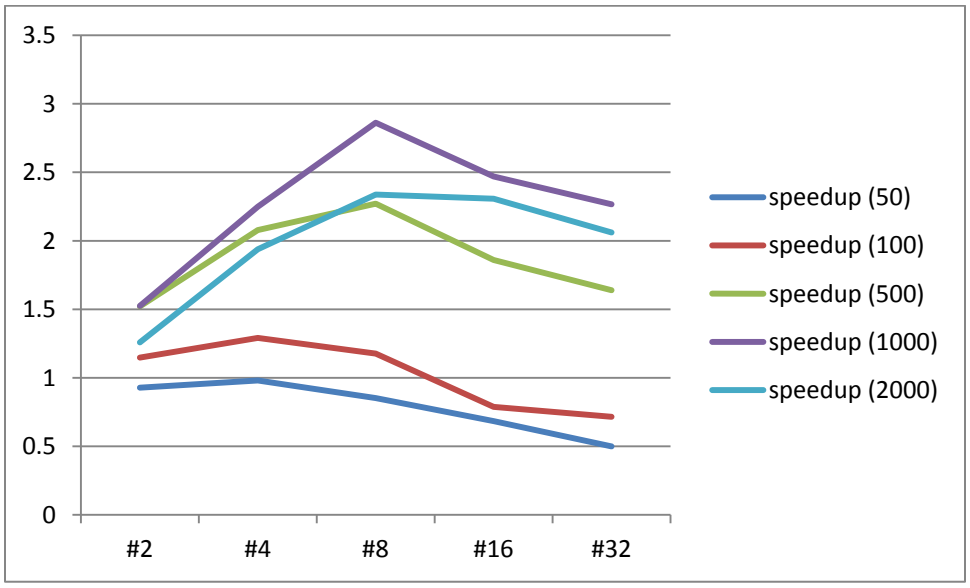
In order to show the speed up clearly, I separated the all my test results into two sets of graphs.

# of points vary from 50 to 2000 in the 1st set

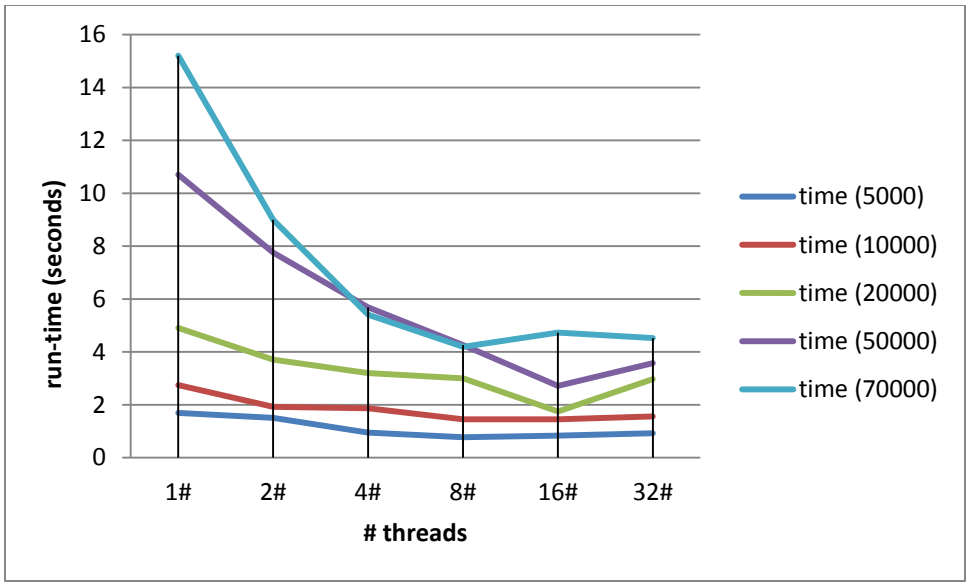
# of points vary from 5000 to 7000s in the 2nd set



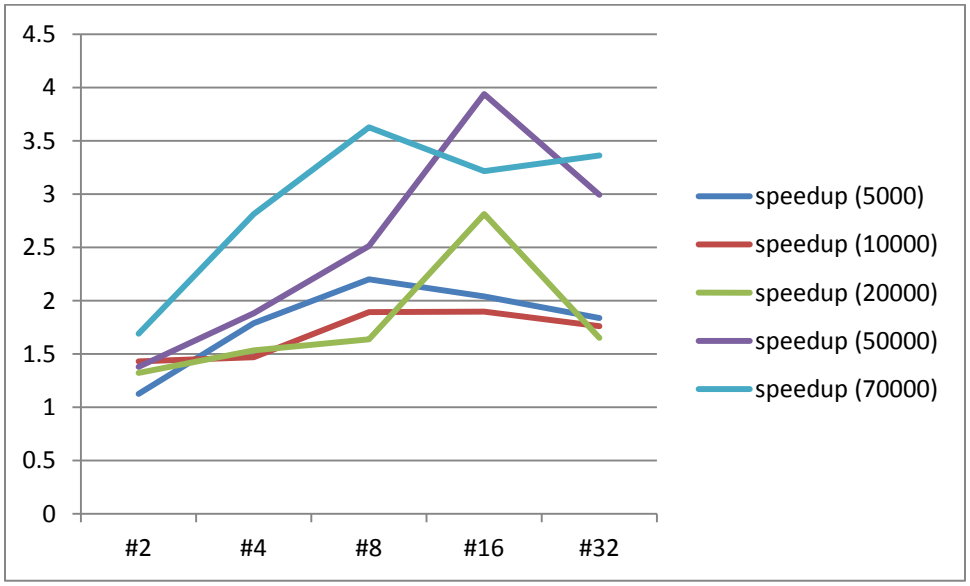
Execution time as a function of the number of threads Graph



Speed Up Graph



Execution time as a function of the number of threads Graph



Speed Up Graph

Ideally, in the speedup graph, you'd see a speedup of *k* with *k* threads. From the graph based on my tests, the linear relationship of speedup with #threads is clear, but the coefficient is not k.

Dr. Scott gave a test result on the assignment web site: for 10,000 points, running 8 threads on a niagara1 machine, the sequential execution takes about 3.1 seconds; the parallel execution takes about 1.3, a speedup of about 2.4.

My version is: For 10,000 points, running 8 threads on a niagara1 machine, the sequential execution takes about 2.746 seconds; the parallel execution takes about 1.45, a speedup of about 1.90, similar to 2.4.

Also, when there are more threads up to 16-32, the increase rate of speedup is decreasing. I guess the cost to create more threads and the might context switch offset the speedup gained from multiple threads.

Thirdly, my second parallelization of Kruskal's algorithm is to create the number of threads before sequentially running Kruskal's algorithm, and after all threads starting, main thread has to wait for them to join finally. I think this is the bottleneck keeping my code from performing better. Helper threads accessing edges to find the cycle edge may encounter cache miss a lot. All these takes much more time than time gained by mere parallelization.

4. Existing Problem

a. java MST

After all printout, the main threads still needs some time to totally return. At first, I thought it may be because the main did not wait for the helper threads; but even after helper threads join, this time latency still exits.

b.

java MST –a 1

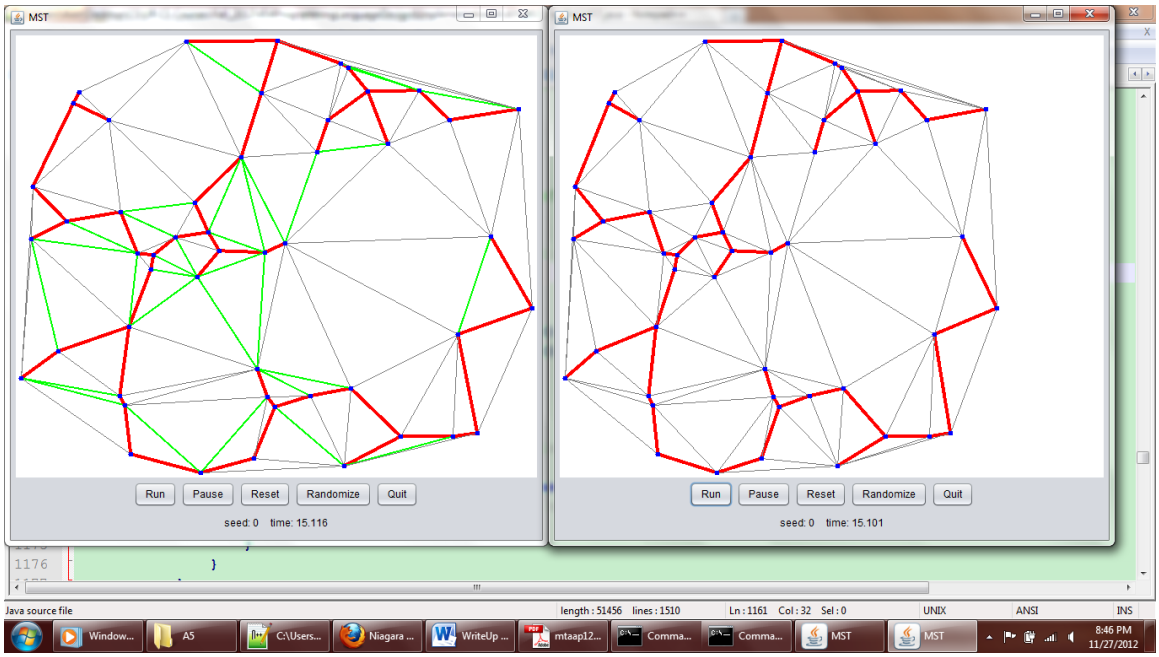
can guarantee that we enter the helper threads. But if I run with java MST –a 0, sometimes it shows we enter helper threads, but most of the times helper threads seemed not to help discard the cycle edge. I am not clear about this problem.

c. Threads join exception, no time to fix them.

5. Conclusion

Parallelization of sequential code is definitely hard. There is more for me to learn and practice on.

Please check java MST –a 3 –t 4, the green edges are being drawn simultaneously. I also attached a screen shot of the result in this report. This feature is the only place I feel confident at.



Left is MST parallel version Right is Sequential version

Reference:

“An Approach to Parallelize Kruskal's Algorithm Using Helper Threads”

Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International