CSC254 Assignment 5

Concurrency

Thu Hoang & Jiahao Lu

1. Parallelize Delauney triangulation:

For the triangulate() method, we notice it follows the divide and conquer mechanism: The set of points is divided into half, and triangulate() is recursively called for the left and right set of points when i >= l and j <=r. Thus, we decided to parallelize that call to optimize time. We created a class called TriangulateThread that extends Thread to demonstrate the thread running in parallel. We then let the left part run on the current thread, and the right part run on the new thread concurrently.

Since each point might influence the whole triangulation, we needed to apply synchronization to protect the shared resources. Here, we use ReentrantLock() to lock the resources used when accessing the points in each triangulation.

Another problem we encountered when creating the threads for recursive functions is that the number of threads created might be spawned exponentially. Thus, we put a limit on the number of threads created and only implement parallelization when

```
Thread.activeCount() < numThreads
```

2. Parallelize Kruskal's algorithm:

In order to parallelize Kruskal, we looked into the paper "An approach to Parallelize Kruskal's algorithm using Helper Threads" by Katsigiannis A. et. al.

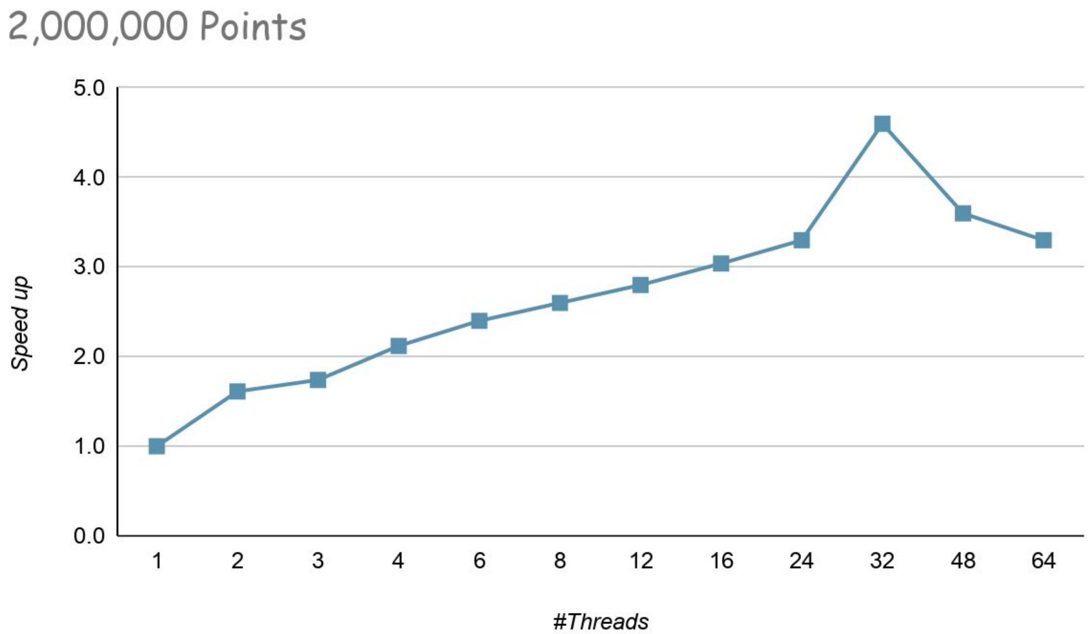
We parallelized the find() part by dividing the set of points into multiple even parts. Then, we created helper threads to find cycles in the later sets while having the main thread starting from the first edge. As invalid points are omitted before running union, the runtime improves significantly.

In this part, we also use Reentrant Locks for the helper threads to protect the shared points when running each encircled method.

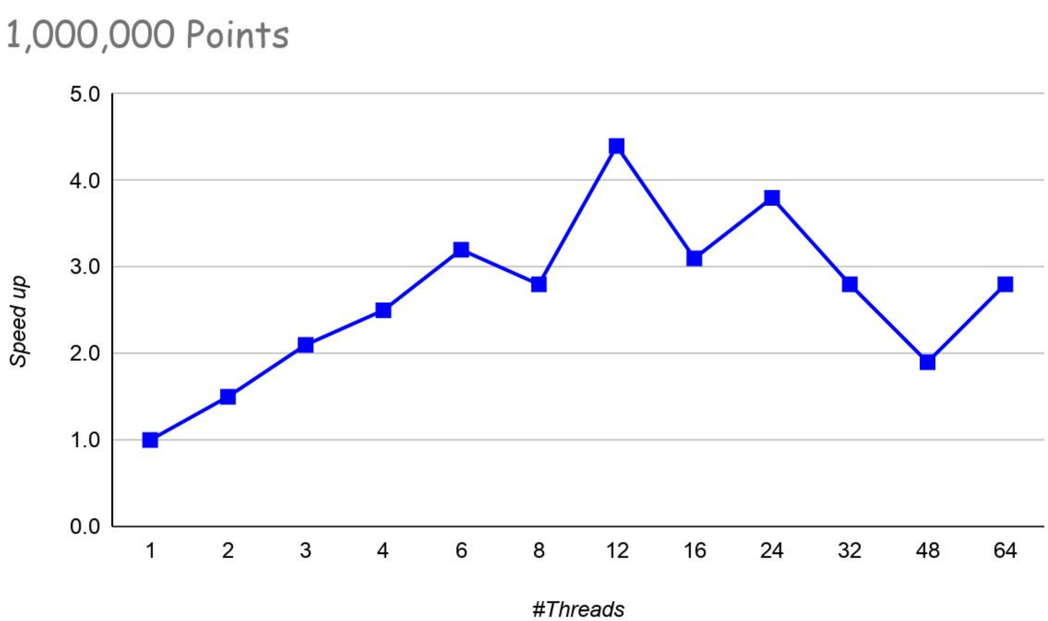
3. Results:

In order to effectively collect the results, we've used the shell script shared on Piazza.

Down below are the results that we have found when running our program in node2x18a in csug machine:

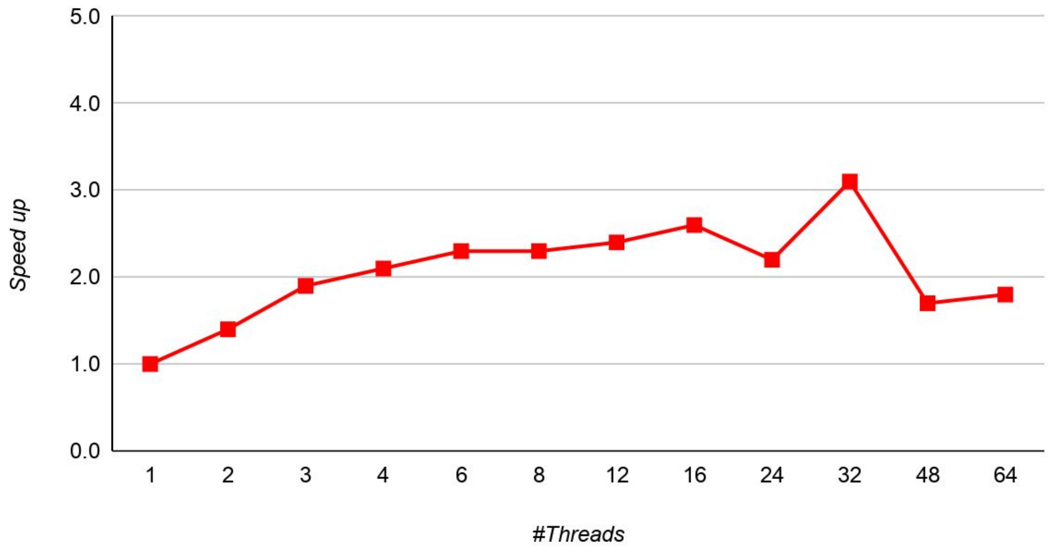


1: 44.191s	2. 27.407s	3. 25.385s	4. 20.757s	6. 18.458s	8. 16.731
12. 16.060s	16. 14.527s	24. 13.277s	32. 9.683s	48. 12.421s	64. 13.481s



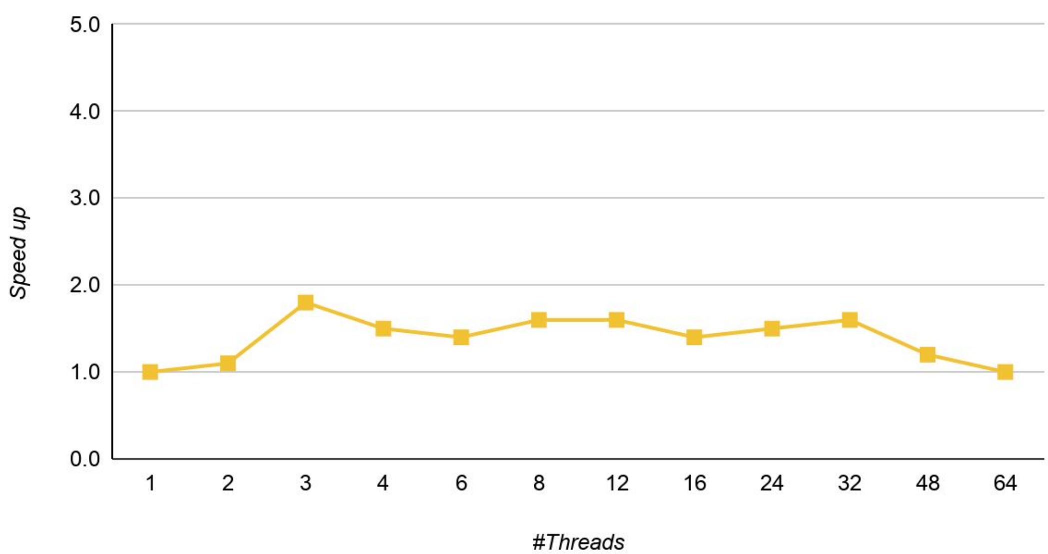
1. 15.542s	2. 10.408s	3. 7.364s	4. 6.197s	6. 4.808s	8. 5.452s
12. 3.536s	16. 5.089s	24. 4.075s	48. 8.281s	64. 5.596s	

500,000 Points



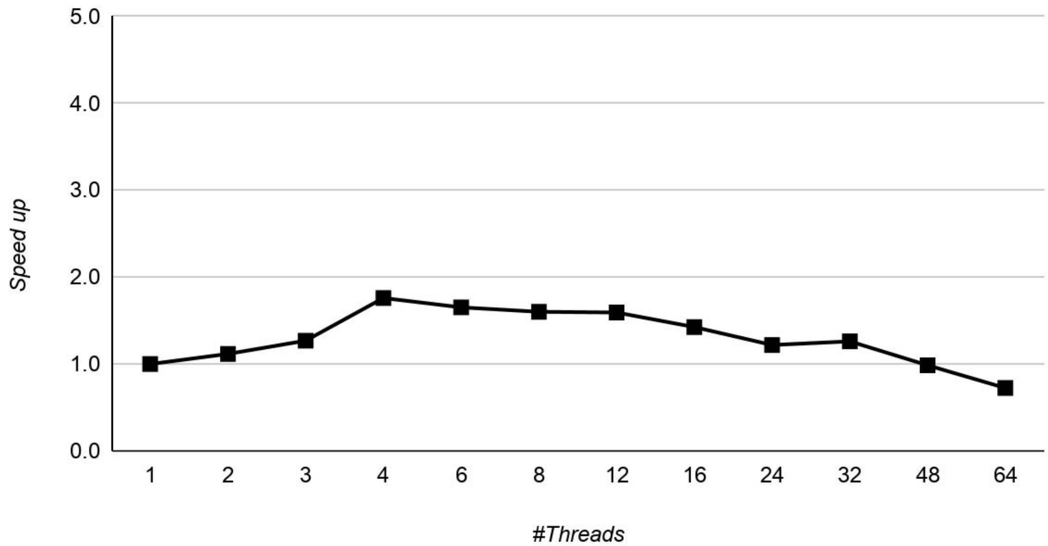
1. 6.854s 2. 5.047s 3. 3.704s 4. 3.287s 6. 2.920s 8. 2.931s
12. 2.898s 16. 2.662s 24. 3.164s 32. 2.183s 48. 4.076s 64. 3.750s

100,000 Points



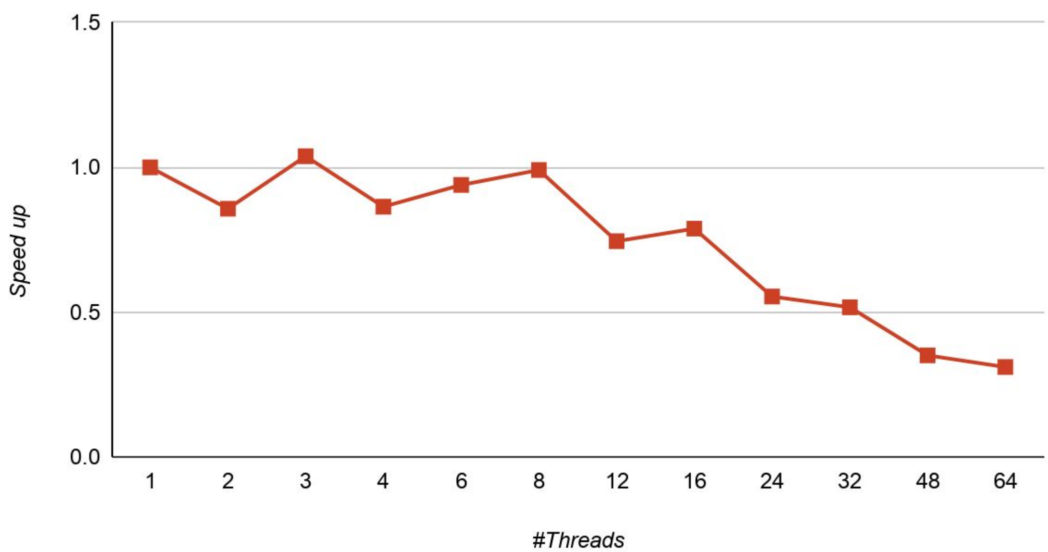
1. 1.132s 2. 1.118s 3. 0.640s 4. 0.765s 6. 0.823s 8. 0.697s
12. 0.698s 16. 0.792s 24. 0.755s 32. 0.687s 48. 0.906s 64. 1.124s

50,000 Points



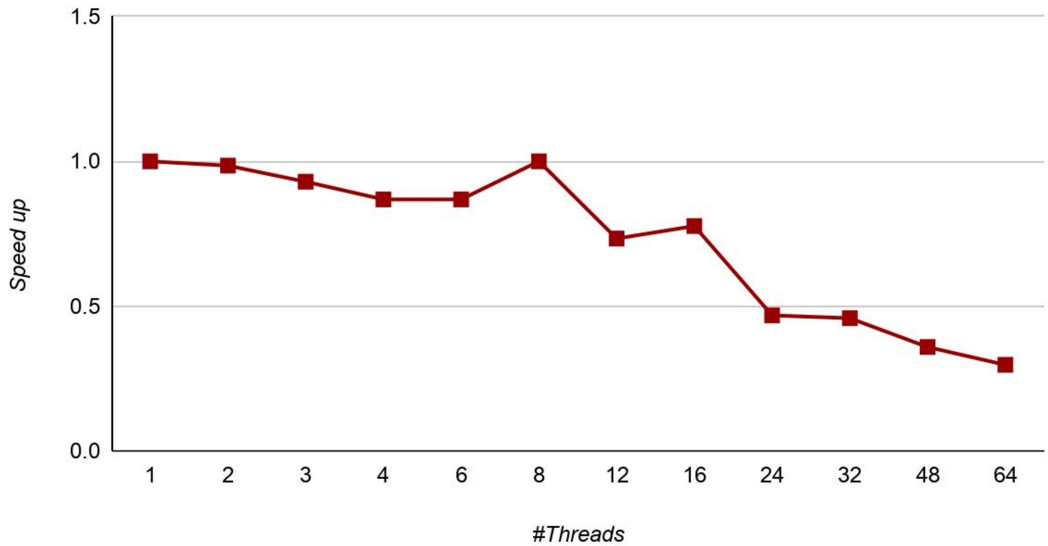
1. 0.623s	2. 0.558s	3. 0.491s	4. 0.354s	6. 0.377s	8. 0.389s
12. 0.391s	16. 0.437s	24. 0.511s	32. 0.494s	48. 0.632s	64. 0.860s

10,000 Points



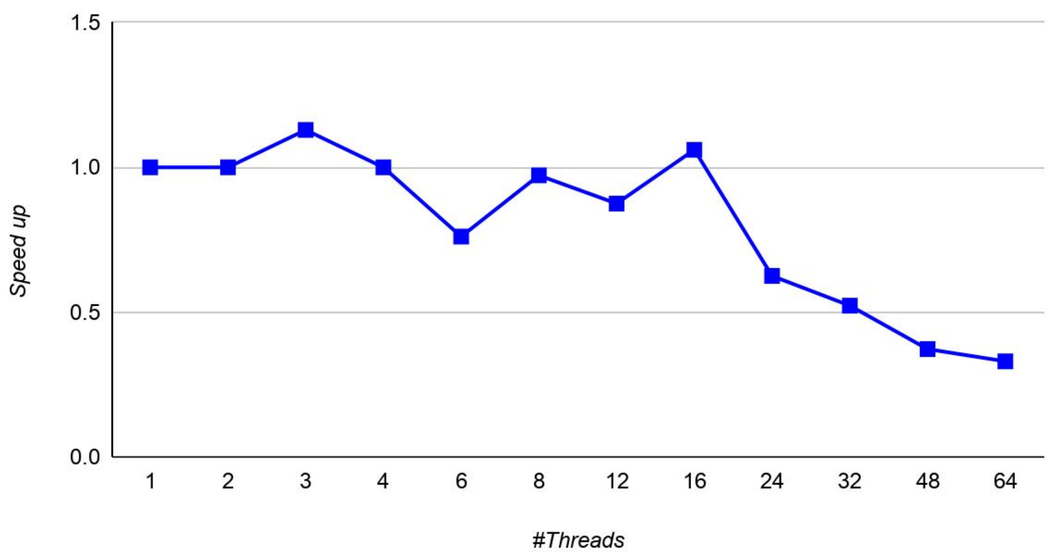
1. 0.108s	2. 0.126s	3. 0.104s	4. 0125s	6. 0.115s	8. 0.109s
12. 0.145s	16. 0.137s	24. 0.195s	32. 0.209s	48. 0.308s	64. 0.348s

5,000 Points

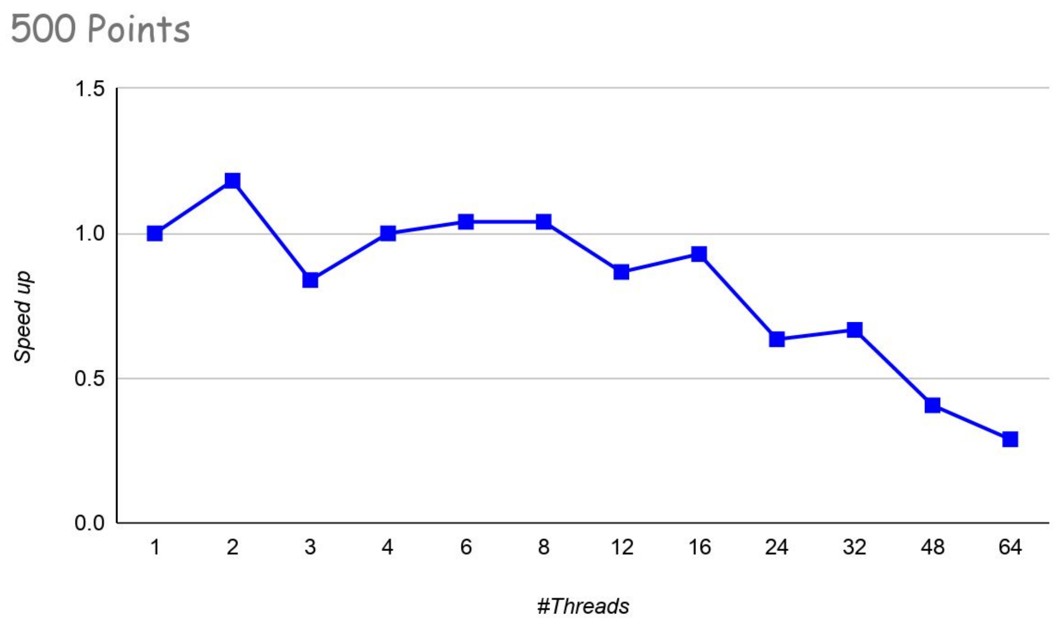


1. 0.066s	2. 0.067s	3. 0.071s	4. 0.076s	6. 0.076s	8. 0.066s
12. 0.09s	16. 0.085s	24. 0.141s	32. 0.144s	48. 0.184s	64. 0.222s

1,000 Points



1. 0.035s	2. 0.035s	3. 0.031s	4. 0.035s	6. 0.046s	8. 0.036s
12. 0.04s	16. 0.033s	24. 0.056s	32. 0.067s	48. 0.094s	64. 0.106s



1. 0.026s	2. 0.022s	3. 0.031s	4. 0.026s	6. 0.025s	8. 0.025s
12. 0.03s	16. 0.028s	24. 0.041s	32. 0.039s	48. 0.064s	64. 0.09s

Overall, the results have shown to be effective for large sets of points (>50,000) with the number of threads between 4 and 48. Most significantly, the program has helped to increase the speedup by 4.56 times when running in 32 threads at 2,000,000 points.

However, when the set of points is smaller, the speedup is not very meaningful. At large numbers of threads, it might even make the runtime much worse. A conviction that we make is that the creation of thread takes more time than the time spent on normal programs itself.