Student1: Jinhao Zhang Email: jzh160@ur.rochester.edu
Student2: Mohammed Ibrahim Khan Email: mkhan26@ur.rochester.edu

WRITEUP

The Coordinator.java and MST.java is source code for Assignment 5: Concurrency of CSC 454.

The goal of this project is to parallelize an existing sequential program, written in Java, that constructs a (Euclidean) minimum spanning tree (MST) for a collection of points in the plane.
The code separates into two main parts: Delaunay Triangulation of the given points, and then MST of the created plane.

For our project, we mainly optimized two functions: *DwyerSolve* and *KruskalSolve* corresponding to the triangulation part and MST part.

In *Dwyersolve*, our method is to build a fork-join thread pool, which has semaphore in its underlying code. We change the *triangulate* function into a class, which is an extension of the thread. In original code, it uses divide-and-conquer method, which break the points slot into two parts, and call function itself. In our new thread class, this part became to create new threads. After creation, it forks the new threads apart from the parent thread. These threads will join back when they terminated. We also made the pool adjust to a fixed number of threads from input.

In *Kruskalsolve,* we define a helper thread class which contains the operation to be performed by a thread. The parallelization strategy involves having a main thread performing the standard MST and helper threads taking bigger edges and adding them to the solution tree temporarily. If these edges form a cycle they are marked as such and discarded. The efficiency takes effect when as more of these helper threads find cycle edges and those are discarded thus reducing load on the main MST thread. This approach of using helper threads is taken from Katsigiannis et al [1].

DIVISION of LABOR

Jinhao Zhang implemented the triangulation part and Ibrahim Khan implemented the MST part.

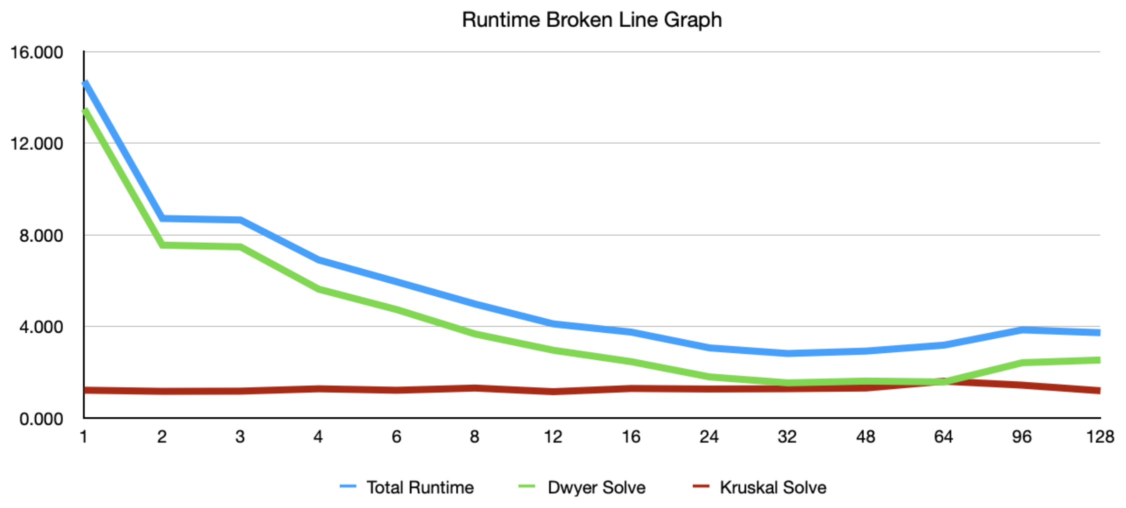
CODE EVALUATION

Our result of code evaluation on node2x18a is shown below:

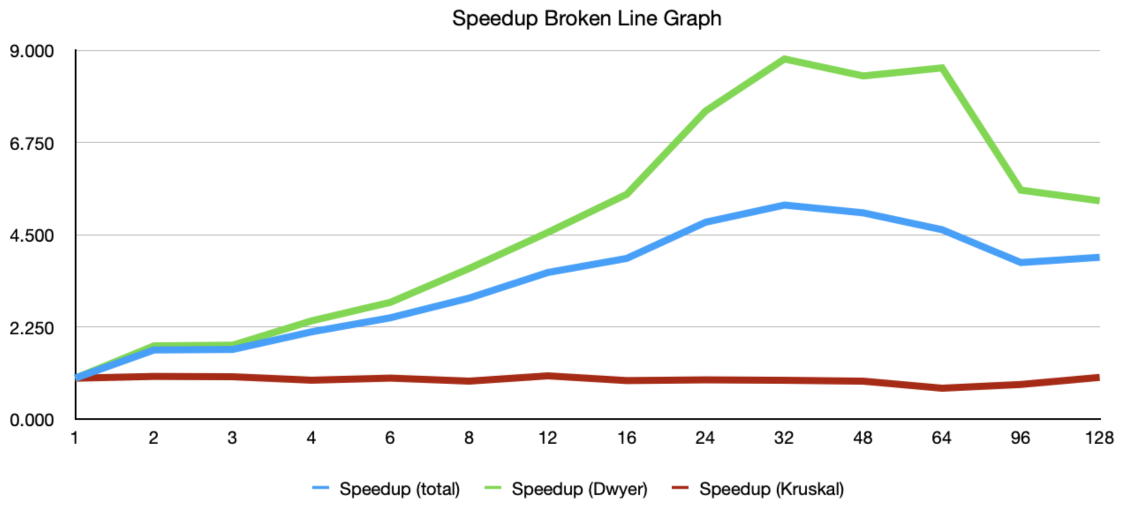
Number of Threads	**Total Runtime**	**Dwyer Solve**	**Kruskal Solve**	**Speedup (total)**	**Speedup (Dwyer)**	**Speedup (Kruskal)**	**Similarity to Sequential Version**
1	14.704	13.489	1.215	1.000	1.000	1.000	100.000%
2	8.709	7.545	1.164	1.688	1.788	1.044	84.418%
3	8.642	7.469	1.173	1.701	1.806	1.036	56.715%
4	6.900	5.623	1.277	2.131	2.399	0.951	53.275%

6	5.943	4.731	1.212	2.474	2.851	1.002	41.236%
8	4.976	3.668	1.308	2.955	3.677	0.929	36.937%
12	4.108	2.959	1.149	3.579	4.559	1.057	29.828%
16	3.748	2.458	1.290	3.923	5.488	0.942	24.520%
24	3.060	1.794	1.266	4.805	7.519	0.960	20.022%
32	2.815	1.535	1.280	5.223	8.788	0.949	16.323%
48	2.921	1.611	1.310	5.034	8.373	0.927	10.487%
64	3.181	1.574	1.607	4.622	8.570	0.756	7.223%
96	3.848	2.414	1.434	3.821	5.588	0.847	3.980%
128	3.723	2.532	1.191	3.950	5.327	1.020	3.086%

The graph of execution time:



The graph of speedup



We can see that execution time line of MST is nearly flat. It's because there's a significant overhead in thread creation and MST sequential being very fast, this overhead of multithreading greatly increases the runtime.

Runtime of triangulation part significantly decreases when the number of threads goes up, which also leads to reduction of the total runtime.

For the speedup, it grows steadily with the number of threads increases before 32. After 32 threads, the speedup slows down and eventually falls. It probably due to thread creation overhead or lack of available concurrency.

Comparing to the sequential version of speedup (k with k threads), the percentage of similarity (our result divided by ideal result) keeps going down when threads increase. Except for the lack of available concurrency, it still has other bottlenecks.

BOTTLENECKS

For triangulation, every time when the code tries to stitch the small recursive parts together, it involves creating new edges and breaking existing edges. Bring them together only happens when small recursive triangulation executed. These stitch-up steps are sequential and there is large amount of these steps have to be done when the input points number is large.

The regular sequential MST algorithm is really fast especially when compared against triangulation. That coupled with the fact that Kruskal is inherently sequential makes good parallelization hard to achieve and overhead of multithreading is very much visible. Creating threads itself is a big bottleneck in this case.

EXTRA CREDIT

The parallelization strategy that we use for Kruskal involves using edges out of order by a lot of threads. But since one thread makes MST in a sequential way, we’re not sure if it qualifies.

CITATION

[1] A. Katsigiannis, N. Anastopoulos, K. Nikas and N. Koziris, "An Approach to Parallelize Kruskal's Algorithm Using Helper Threads," 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, Shanghai, 2012, pp. 1601-1610, doi: 10.1109/IPDPSW.2012.201.