=== CSC 254 A6: Concurrency Timing Analysis of MST Algorithms ===

Chester Holtz - 28264729 - choltz2@u.rochester.edu
Lee Murphy - 28250920 - lmurp14@u.rochester.edu

---

=== Instructions ===

To compile: “javac MST.java”
To run: “java MST [-a (0|1|2|3)] [-n *num*] [-s *num*] [-t *num*]”

---

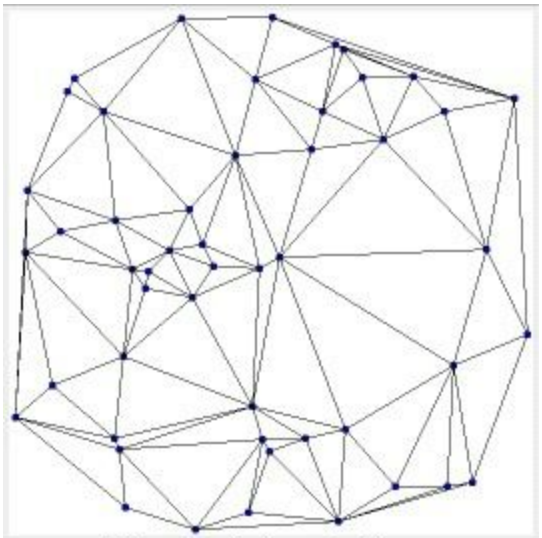
=== Bugs ===

No known limitations

---

=== Introduction ===

This assignment tasked us with converting two sequential algorithms into their parallelized counterparts in Java. We were provided code to compute the Delaunay Triangulation of a set of points on the 2d plane. A *triangulation* of points in the plane is a maximal set of line segments whose endpoints are among the given points and which do not otherwise intersect. If one were to stretch a rubber band around the given points, a triangulation divides the interior space into triangles; hence the name. The *Delaunay triangulation* (example given below.) has the property that the *circumcircle* of the vertices of a triangle (the unique circle on which all three vertices lie) contains no other point in the set. One can prove that the Delaunay triangulation of a set is unique if no four points lie on the same circle and no three points lie on the same line. One can also prove that the Delaunay triangulation maximizes the minimum corner angle across all triangles. Delaunay triangulations tend to be pleasing to the eye. They are used in graphics rendering and mechanical simulation.



Delaunay triangulation
of a random set of points

Once the triangulation has been computed, the program then computes the Minimum Spanning Tree (MST) from the triangulation mesh using Kruskal’s O(nlogn) algorithm. The specific

purpose of this project was to parallelize the triangulation algorithm (Dwyer's Algorithm) and Kruskal's MST algorithm and to do timing analysis on the resulting performance.

---

=== Approach ===

--- Dwyer's Algorithm ---

Dwyer's Algorithm is used for computing the Delaunay triangulation of the given points. The algorithm, in its sequential form, creates the triangulation by recursively calling the triangulation function on smaller and smaller sets of points until it can directly determine which edges are in the triangulation. It then returns back through the recursive layers, calling a stitching function each time it goes up a level to back sure the two triangulations it is combining results in a full triangulation of the union of the points. The stitching step may add or remove edges in order to join the two triangulations. Thus, with an understanding of how Dwyer's algorithm runs sequentially, it is easier to understand how to get it to run concurrently. As mentioned before, the algorithm operates by constantly splitting the set of points into two smaller sets of points. Thus, with this natural divide-and-conquer feel, there is nothing stopping us from running the triangulation on the two sets concurrently. Thus, we spawn a thread to handle the "left" half of the triangulation and run the "right" half in our own thread context. Then, we wait for the thread that we spawned to finish using the join() function from the Java Thread class. Waiting for the spawned thread to finish is critical because we cannot move on to the stitching step until both halves are created. This is because our stitching algorithm is sequential and making it concurrent was part of the extra credit that we did not pursue. In order to use the [-t *num*] argument option where a max number of threads is provided, we increment a global counter called *threads_running* when we spawn a new thread and decrement it after the join() step.

--- Kruskal's Algorithm ---

Kruskal's Algorithm is an inherently sequential algorithm used to compute the Minimum Spanning Tree of a graph. The pseudocode for the provided implementation is given below.

```
KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3    MAKE-SET(v)
4 foreach (u, v) ordered by weight(u, v), increasing:
5    if FIND-SET(u) ≠ FIND-SET(v):
6       A = A ∪ {(u, v)}
7       UNION(u, v)
8 return A
```

By definition, Kruskal's Algorithm operates on a sorted set of edges - sorted by weights. For this reason, it is difficult to imagine a concurrent version of Kruskal's Algorithm. Due to time constraints, we were unable to come up with our own scheme, and instead implemented the approach outlined in Katsigiannis et al's. paper *An approach to parallelize Kruskal's algorithm using Helper Threads*. In this study, Katsigiannis et al proposed a scheme to parallelize Kruskal's algorithm by utilizing helper threads to offload various tasks. He also proves through definition that the Minimum Spanning Forest is composed only of sorted threads that do not form cycles. Essentially, the algorithm he outlines computes the MST by having one main

thread compute the MST as normal. Concurrently, helper threads examine edges of larger weight and add them to the graph as temporary edges. If a cycle is formed by adding these edges to the graph, they are flagged as cycle edges and discarded. It follows that the more cycle edges found by the helper threads, the more work is offloaded from the main thread.

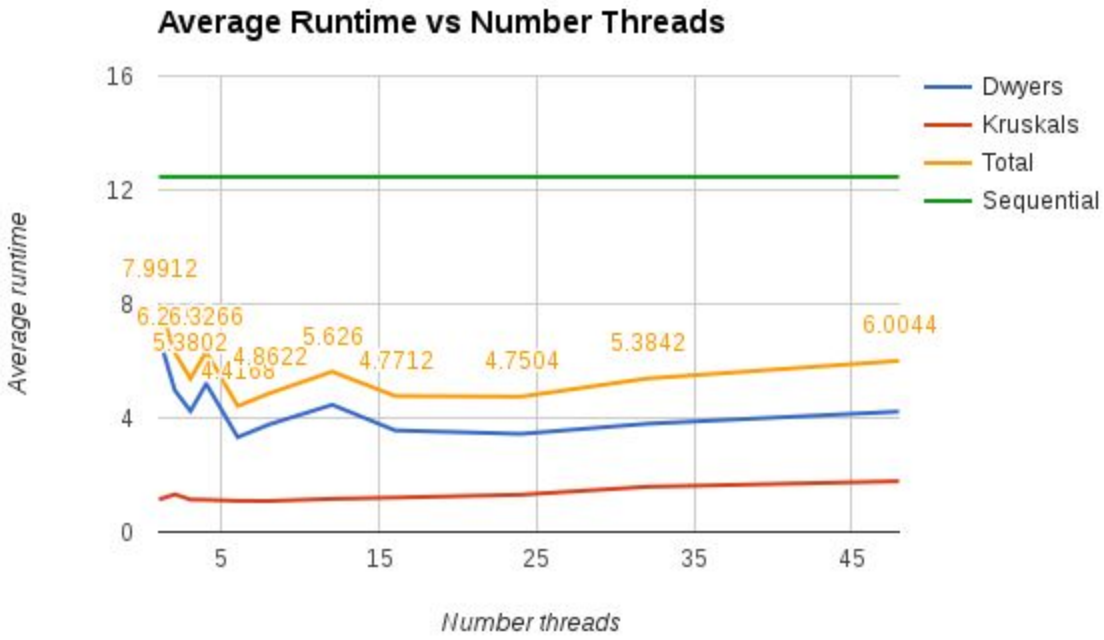
---

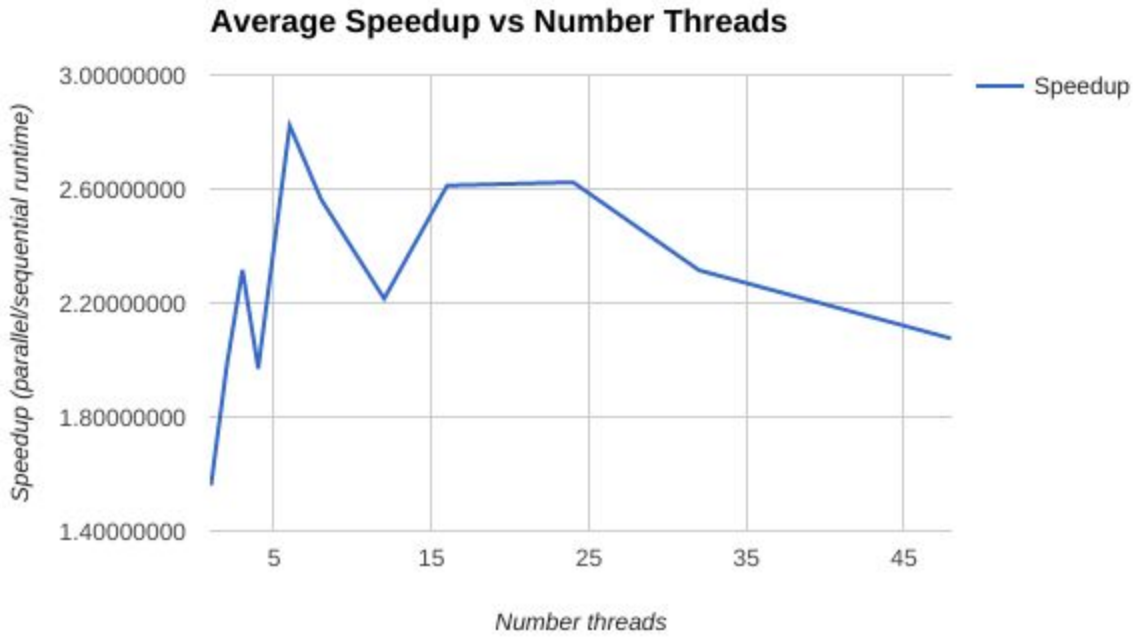
=== Data Collection ===

We performed timing analysis on node2x14a.csug.rochester.edu. This machine has two processor chips, each containing 14 cores, each of which has 2 hardware contexts (hyperthreads). This means the machine can execute up to 56 threads in parallel. We ran our program against 1,000,000 points using 1, 2, 3, 6, 8, 12, 16, 24, 32, 48 cores for five times each and plotted the average results against the average runtime of the sequential algorithm. We used the Java Datetime library's getTime() method which is precise to an order of milliseconds. The graphs below provide help to visually represent our analysis. The first graph plots where the program spent the most time (Dwyer's Algorithm, Kruskal's Algorithm, total time), and the second graph is a representation of the average speedup vs the number of threads.

---

=== Results ===






---

=== Discussion ===

It is interesting to observe the above graphs. We see that for low numbers of threads, the difference in threads drastically alters the runtime, but as we increase the number of threads, the runtime of our algorithm stays about the same or increases slightly. For an optimal implementation, we say that for k threads, a speedup proportional to k should be achieved. We attribute this to the overhead involved with thread creation. Additionally the parallelization scheme we implemented for Kruskal's Algorithm results in a large increase in time. Although Kruskal's algorithm only accounts for approximately 5 ~ 10% of the total runtime, we create threads prior to generating MST edges, and the associated overhead results in a large time cost.

---

=== Conclusion ===

In conclusion, we successfully implemented two schemes for parallelizing sequential algorithms - Dwyer's Algorithm and Kruskal's Algorithm. We base most of our work on Kruskal's Algorithm off of the paper written by Katsigiannis et al., and performed in depth time analysis on the result. We posit that the overhead involved in thread creation is responsible for the lack of serious improvement. The algorithms themselves are efficient, but the non-optimal speedup can be attributed to Java's implementation of multithreading.

---

=== Future Work ===

Additional work can be done to further parallelize our program. Concurrent algorithms for operations on edge sets: set and union subroutines exist and are well known. Unfortunately, we

did not have enough time to implement them for this project, but will consider examining how these schemes could be included later.

---

=== Extra Credit & misc. ===

For this project we successfully implemented the algorithm outlined by Katsigiannis et al. Since this algorithm does not require edges be selected in order, we effectively solved the problem outlined in extra credit 6 [1]. In addition, we provide a python script to automate much of the testing. Prior to our date, we load the script into the crontab file so it executes automatically when our schedule begins - and removes itself after its done.

---

=== References ===

[1] http://www.cs.rochester.edu/courses/254/fall2015/assignments/java.shtml
[2] Katsigiannis, Anastopoulos, Nikas, Koziris, An approach to parallelize Kruskal's algorithm using Helper Threads, University of Athens.
[3] http://www.cs.rochester.edu/u/scott/254/ur_only/1986_Dwyer_Delaunay.pdf