Concurrency Assignment by Thabani Dube and Benson Gathee

1. Run the program
    (1) cd Concurrency-of-Dwyer-Triangulation-and-Kruskal-MST
    (2) javac -d bin *.java
    (3) java -classpath bin MST -n num_of_points -t num_of_threads

    NB: for testing, we used num_of_points = 1000000 and num_of_threads = 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64

    hence, java -classpath bin MST -n 1000000 -t 1 (or 2 or …)

2. Implementation Details using examples
    (1) Parallelizing the Dwyer triangulation
        - Assuming num_of_threads = 4
        - Dwyer triangulation has a natural divide and conquer hence easier to parallelize. We added an extra parameter to the triangulate function called '*threadsLeft*' which keeps track of the number of threads that each recursive step can use.
        - For example, in the first recursive call, we have 4 threads. In the divide and conquer stage, this is shared by the two recursive calls of triangulate which start on two separate threads, the main thread and one branching off. In the subsequent recursive calls, this is split into 2 again and so at this point each further recursive call can use only 1 thread. For the rest of the recursive calls deep in the recursion, only 1 thread is left and the rest of the divide and conquer steps run sequentially.
        - The source code is commented out well enough for you to follow this.
        - We also wait for threads to complete before proceeding e.g. on the image below (T2 on same level as T3, awaits T3 before returning to the caller)

    The image below shows the execution of our parallelized triangulation using a different easier example that gets our idea along.

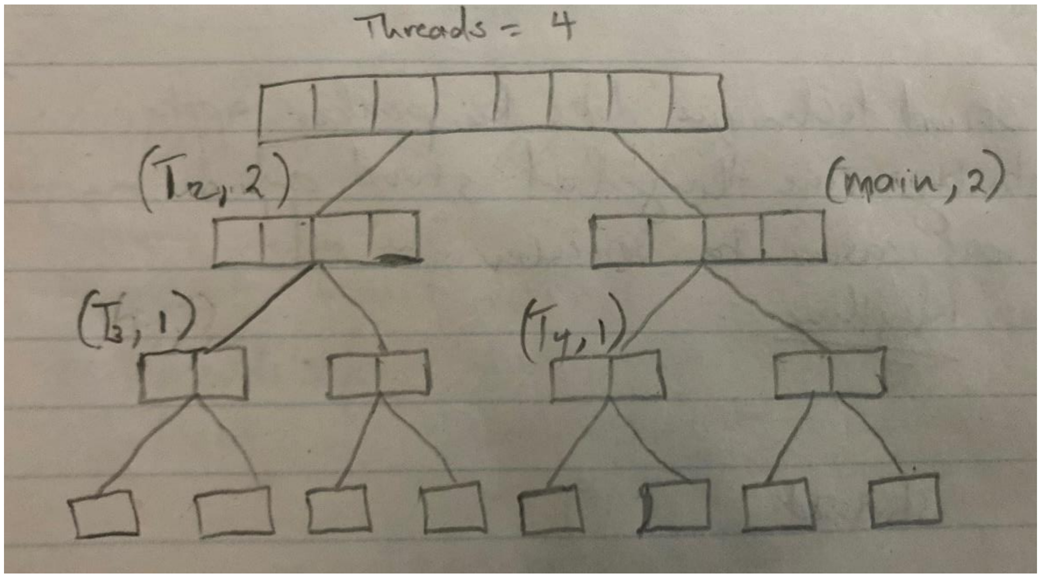


Image showing the conceptual execution of our parallelized triangulation

(2) Parallelizing the Kruskal's algorithm

- Assuming num_of_edges (produced by the triangulation) = 100 and num_of_threads = 5, we divided the number of edges equally among the threads, hence in this case, each thread works on 100/5 = 20 edges.
- At the Kruskal's stage, it is worth noting that the edges are sorted in nondecreasing order in an ArrayList data structure. Thread 1 (main thread) starts working on the edge 0 (min weight edge) and continues through edge 100 (max weight edge) from the ArrayList. Thread 2,3,4,5 (helper threads) works on edges 21-40, 41-60, 61-80, 81-100 respectively from the Arraylist. The function of these helper threads is to evaluate whether their respective edges form a cycle based on the currect MST formed by the main thread. If an edge forms a cycle, it is marked as CYCLE_EDGE thus eliminating and leaving less edges for the main thread to check once it arrives in that thread's section. This saves a lot of time for the sequential main thread. Once the main thread arrives at another thread's section (eg. main thread arrives at edge 21 which is thread 2's section), the thread is effectively stopped (thread 2 is stopped) to avoid synchronization issues, and the main thread continues to evaluate the non-eliminated edges.
- The helper threads loop continuously checking its edge set with the MST that has been constructed so far until the main thread arrives in its section at which point it stops after already cutting down on the work the main thread has to do.

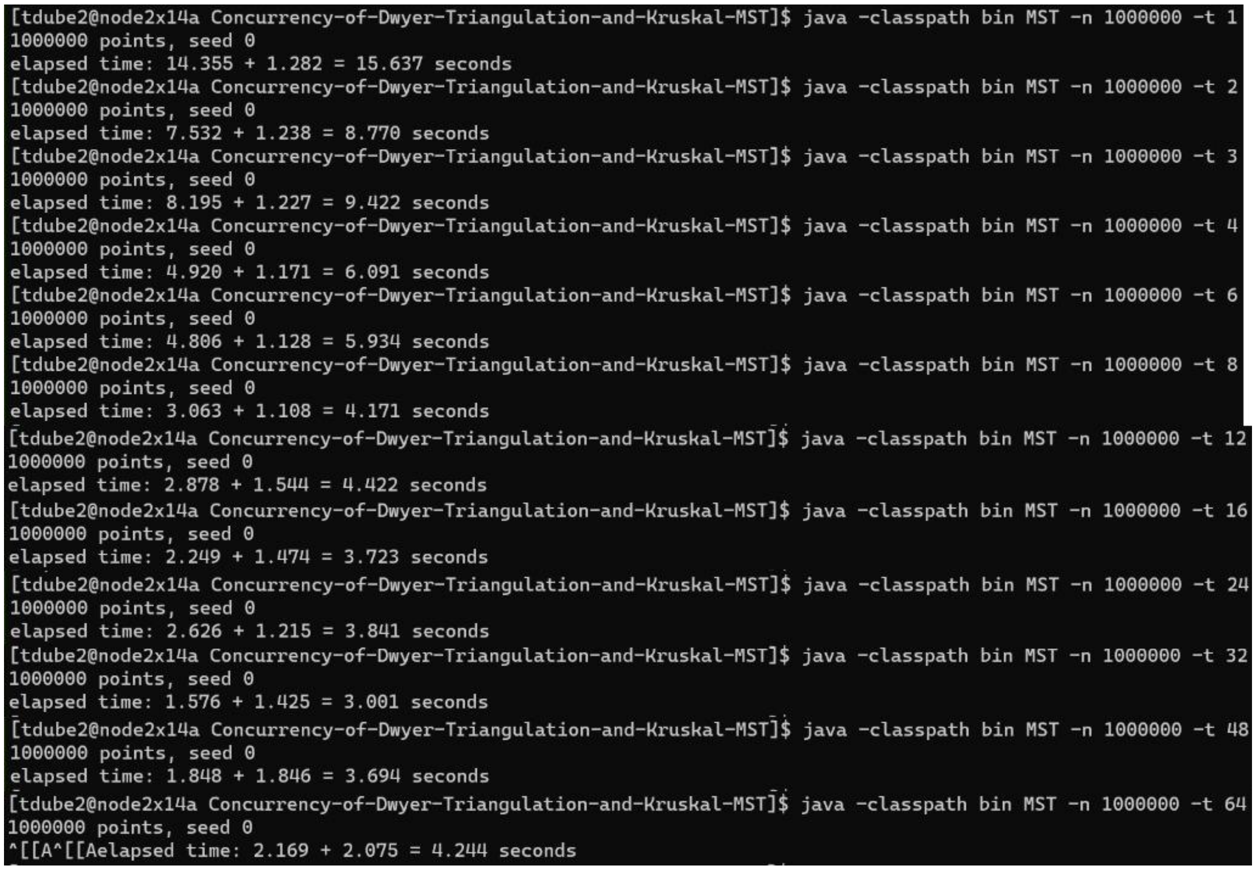
For more implementation details on our parallelization strategy of Kruskal's algorithm, check the '*Implementation Details*' from the paper referenced below.

Reference

http://tarjomefa.com/wp-content/uploads/2017/10/7793-English-TarjomeFa.pdf

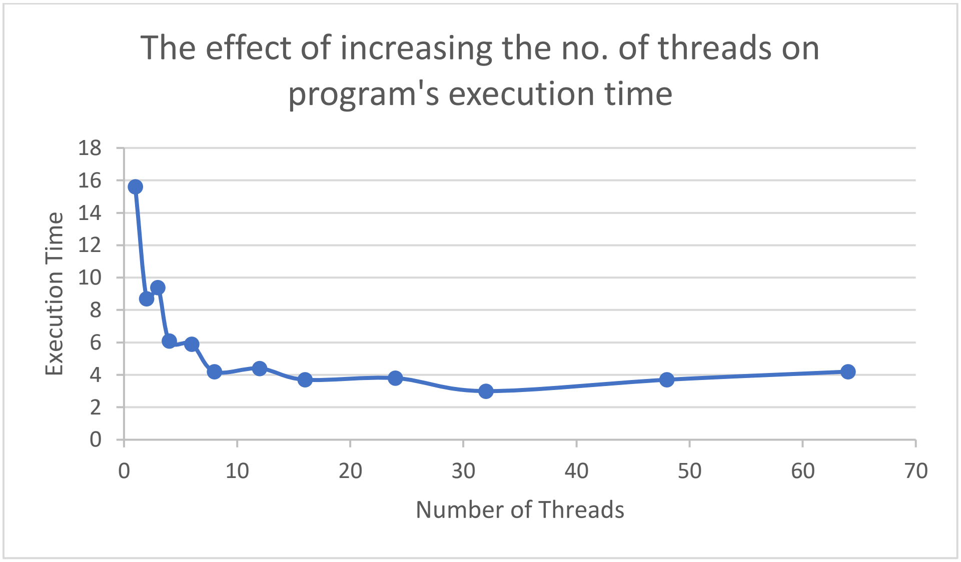
3. Evaluation

The image below shows the execution of our parallelization on both the Dwyer triangulation and Kruskal's algorithm. As the number of threads increases, the time decreases but later increases due to thread creation overhead.

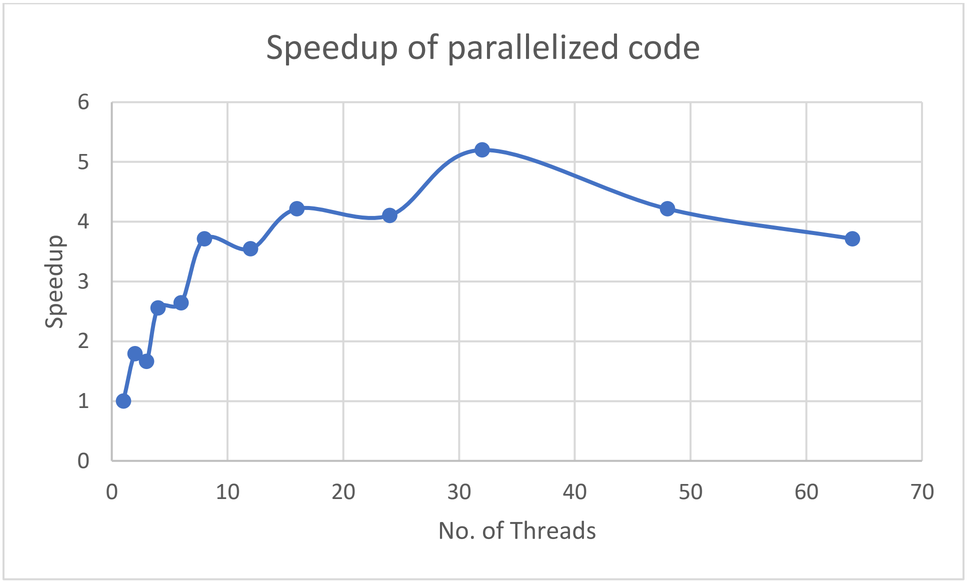


```
[tdube2@node2x14a Concurrency-of-Dwyer-Triangulation-and-Kruskal-MST]$ java -classpath bin MST -n 1000000 -t 1
1000000 points, seed 0
elapsed time: 14.355 + 1.282 = 15.637 seconds
[tdube2@node2x14a Concurrency-of-Dwyer-Triangulation-and-Kruskal-MST]$ java -classpath bin MST -n 1000000 -t 2
1000000 points, seed 0
elapsed time: 7.532 + 1.238 = 8.770 seconds
[tdube2@node2x14a Concurrency-of-Dwyer-Triangulation-and-Kruskal-MST]$ java -classpath bin MST -n 1000000 -t 3
1000000 points, seed 0
elapsed time: 8.195 + 1.227 = 9.422 seconds
[tdube2@node2x14a Concurrency-of-Dwyer-Triangulation-and-Kruskal-MST]$ java -classpath bin MST -n 1000000 -t 4
1000000 points, seed 0
elapsed time: 4.920 + 1.171 = 6.091 seconds
[tdube2@node2x14a Concurrency-of-Dwyer-Triangulation-and-Kruskal-MST]$ java -classpath bin MST -n 1000000 -t 6
1000000 points, seed 0
elapsed time: 4.806 + 1.128 = 5.934 seconds
[tdube2@node2x14a Concurrency-of-Dwyer-Triangulation-and-Kruskal-MST]$ java -classpath bin MST -n 1000000 -t 8
1000000 points, seed 0
elapsed time: 3.063 + 1.108 = 4.171 seconds
[tdube2@node2x14a Concurrency-of-Dwyer-Triangulation-and-Kruskal-MST]$ java -classpath bin MST -n 1000000 -t 12
1000000 points, seed 0
elapsed time: 2.878 + 1.544 = 4.422 seconds
[tdube2@node2x14a Concurrency-of-Dwyer-Triangulation-and-Kruskal-MST]$ java -classpath bin MST -n 1000000 -t 16
1000000 points, seed 0
elapsed time: 2.249 + 1.474 = 3.723 seconds
[tdube2@node2x14a Concurrency-of-Dwyer-Triangulation-and-Kruskal-MST]$ java -classpath bin MST -n 1000000 -t 24
1000000 points, seed 0
elapsed time: 2.626 + 1.215 = 3.841 seconds
[tdube2@node2x14a Concurrency-of-Dwyer-Triangulation-and-Kruskal-MST]$ java -classpath bin MST -n 1000000 -t 32
1000000 points, seed 0
elapsed time: 1.576 + 1.425 = 3.001 seconds
[tdube2@node2x14a Concurrency-of-Dwyer-Triangulation-and-Kruskal-MST]$ java -classpath bin MST -n 1000000 -t 48
1000000 points, seed 0
elapsed time: 1.848 + 1.846 = 3.694 seconds
[tdube2@node2x14a Concurrency-of-Dwyer-Triangulation-and-Kruskal-MST]$ java -classpath bin MST -n 1000000 -t 64
1000000 points, seed 0
^[[A^[[Aelapsed time: 2.169 + 2.075 = 4.244 seconds
```

The graph below shows the effects of increasing the number of threads on the program's execution time.



The graph below shows the speedup of our parallelized code (the runtime of the original unmodified sequential code divided by the runtime of our parallelized code)



Though we were able to parallelize both programs correctly using the strategies explained above, we have thought of other strategies that would have led to greater performances.

For the Kruskal’s algorithm, a better approach would be to come up with an equation that adjusts the threads to work on a different set of edges other than stopping a thread when the main thread gets to the helper threads part.

For the Dwyer triangulation algorithm, a better approach would be to reuse threads as soon as they are done with their section.

Thank you for looking at our last project for the class. Happy Holiday!