Parallel Implementation of Kruskal's and Prim's MST Algorithms with openMP

By:

Imad-Uddin Siddiqui, is293, 158003515
Amod Deo, aad219, 197006010
Shounak Rangwala, snr85, 191004996

Motivation

- The Minimum Spanning Tree (MST) is a subset of the edges of a connected weighted undirected graph that contains all of the vertices without any cycles and the minimum possible total edge weight
- Minimum spanning trees have direct applications in the design of networks, including computer networks, telecommunications networks, transportation networks, water supply networks, and electrical grids (which they were first invented for).
- We wanted to see how parallel computing can improve the performance of algorithms that find the MST of a connected weighted undirected graph
- We settled on Kruskal's Algorithm and Prim's Algorithm

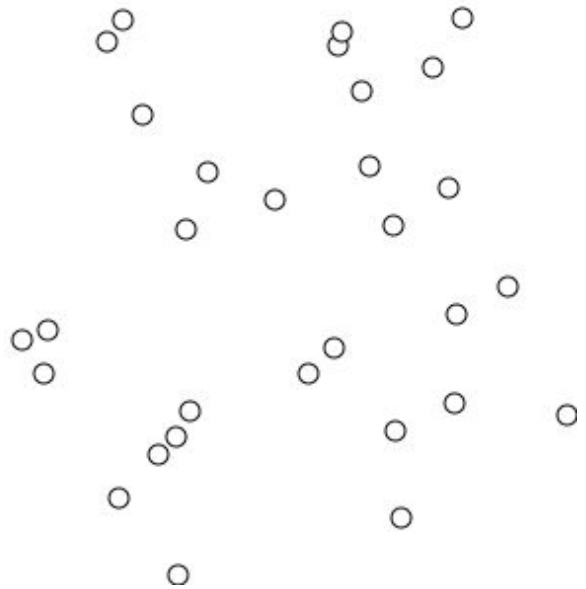
Kruskal’s Algorithm

Concept

*Time Complexity: O(E*log(V))*

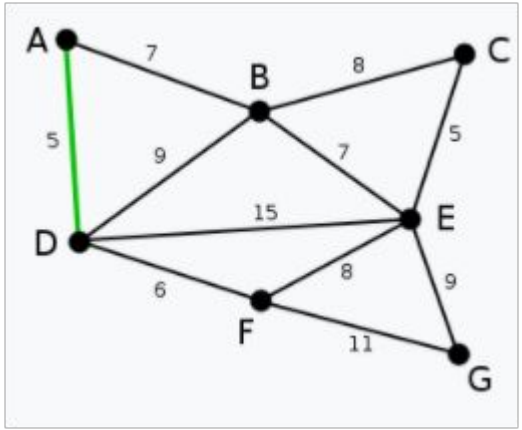
- The algorithm requires a union-find data structure
- You maintain a set for each connected component in the tree
- You proceed in order from the smallest edge to add to the MST, provided that it doesn't create a cycle
- If vertex V and vertex W are in the same set, then adding V-W would create a cycle
- You can check for this by seeing if the parent vertex of both sets are the same or not. If they're not, then it won't result in a cycle

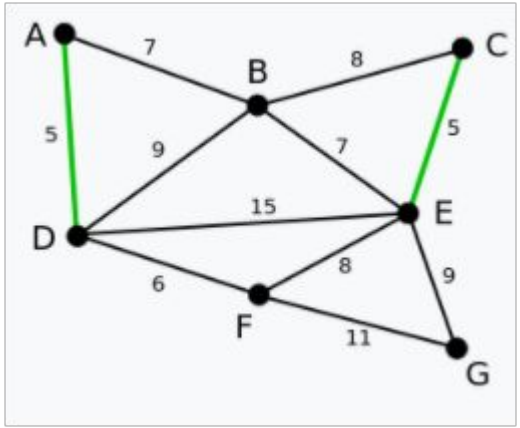
```
algorithm Kruskal(G) is
    F:= ∅
    for each v ∈ G.V do
        MAKE-SET(v)
    for each (u, v) in G.E ordered by weight(u, v), increasing do
        if FIND-SET(u) ≠ FIND-SET(v) then
            F:= F ∪ {(u, v)}
            UNION(FIND-SET(u), FIND-SET(v))
    return F
```

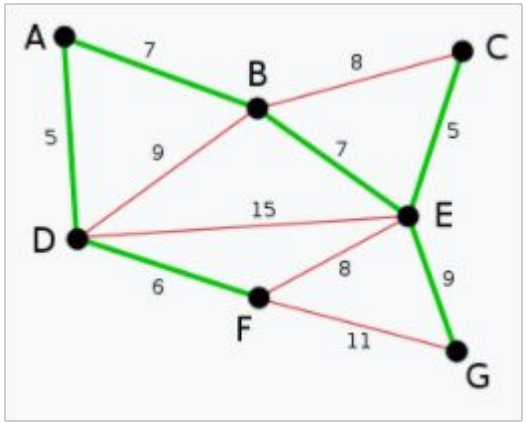


Approach

- Kruskal's algorithm is inherently sequential and hard to parallelize.
  It is, however, possible to perform the initial sorting of the edges in parallel
- There's a sequential and parallel implementation of Kruskal's Algorithm, the latter using OpenMP, which is an API to support shared-memory multiprocessing
- For pre-processing, we use multiple threads to handle quick sorting of the data in parallel before using merge-sort to bring the sorted parts together for the MST.





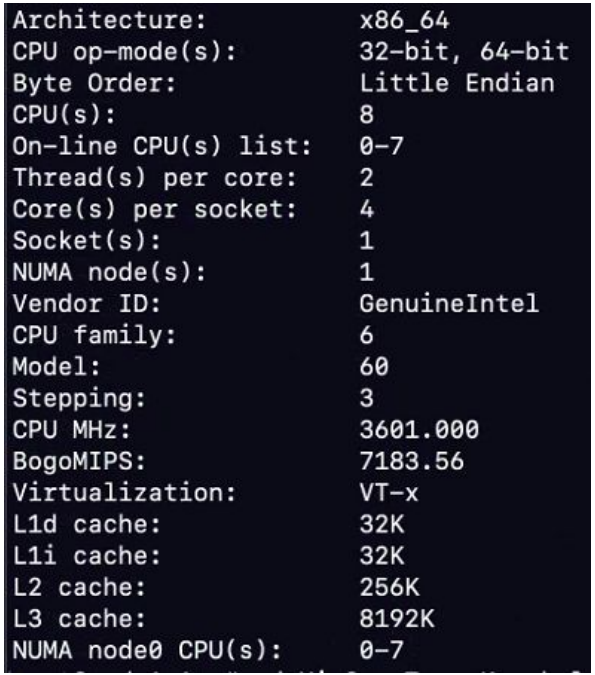


Approach

- For testing purposes we have multiple .bin files for the graph as it is faster to access binary data
- Different bin files are of varying graph density, ranging from (1%, 2%, 5%, 10%)
- Each bin file has 10000 vertices
- A baseline processing time is found by executing the serial algorithm
- The parallel speedup is calculated considering both the processing as well as the communication (between threads) time.
- A speedup without considering I/O (parsing of the edges from the file) is also calculated to visualize the difference

Hardware Specifications

Architecture	x86_64 (Intel)
# of Cores	4
# of Threads per Core	2
Clockspeed	3600 MHz



Results

- We cannot give actual demo because this was done in orbit
- The tests are run on incrementing number of threads
- Speedup is calculated as compared to the processing time of the sequential program
- Speedup without I/O is the speedup after not considering the data parsing time from the files
- We observe that the speedup in most of the tests doesn't improve or change much as the number of threads increase

	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
Test 0 - Speedup	1.89	1.16	1.24	1.00	0.65
Test 0 - Speedup no I/O	2.14	1.18	1.50	1.15	0.66
Test 1 - Speedup	1.99	1.97	1.54	1.04	2.20
Test 1 - Speedup no I/O	2.28	2.32	2.06	2.10	2.66
Test 2 - Speedup	2.86	2.12	1.65	1.60	0.86
Test 2 - Speedup no I/O	2.34	2.47	2.23	2.10	1.66
Test 3 - Speedup	1.99	1.96	1.50	1.11	0.88
Test 3 - Speedup no I/O	2.29	2.32	2.06	2.02	3.01

Results

```
SERIAL (1%):
SERIAL (5%):
SERIAL (10%):
SERIAL (20%):
MPI (1%): 2 procs
MPI (5%): 2 procs
MPI (10%): 2 procs
MPI (20%): 2 procs

Test 0: Correct length (96785169)
Speedup       : 1.89
Speedup no I/O: 2.14

Test 1: Correct length (96750818)
Speedup       : 1.99
Speedup no I/O: 2.28

Test 2: Correct length (95223283)
Speedup       : 2.06
Speedup no I/O: 2.34

Test 3: Correct length (89211066)
Speedup       : 1.99
Speedup no I/O: 2.29
root@node1-1:~/MinSpanTree_Kruskal
```

```
SERIAL (1%):
SERIAL (5%):
SERIAL (10%):
SERIAL (20%):
MPI (1%): 4 procs
MPI (5%): 4 procs
MPI (10%): 4 procs
MPI (20%): 4 procs

Test 0: Correct length (96785169)
Speedup       : 1.16
Speedup no I/O: 1.18

Test 1: Correct length (96750818)
Speedup       : 1.97
Speedup no I/O: 2.32

Test 2: Correct length (95223283)
Speedup       : 2.12
Speedup no I/O: 2.47

Test 3: Correct length (89211066)
Speedup       : 1.96
Speedup no I/O: 2.32
root@node1-1:~/MinSpanTree_Kruskal
```

```
SERIAL (1%):
SERIAL (5%):
SERIAL (10%):
SERIAL (20%):
MPI (1%): 8 procs
MPI (5%): 8 procs
MPI (10%): 8 procs
MPI (20%): 8 procs

Test 0: Correct length (96785203)
Speedup       : 1.24
Speedup no I/O: 1.50

Test 1: Correct length (96750818)
Speedup       : 1.54
Speedup no I/O: 2.06

Test 2: Correct length (95223283)
Speedup       : 1.65
Speedup no I/O: 2.23

Test 3: Correct length (89211066)
Speedup       : 1.50
Speedup no I/O: 2.06
root@node1-1:~/MinSpanTree_Kruskal
```

```
SERIAL (1%):
SERIAL (5%):
SERIAL (10%):
SERIAL (20%):
MPI (1%): 16 procs
MPI (5%): 16 procs
MPI (10%): 16 procs
MPI (20%): 16 procs

Test 0: Correct length (96785171)
Speedup       : 1.00
Speedup no I/O: 1.15

Test 1: Correct length (96750818)
Speedup       : 1.04
Speedup no I/O: 2.10

Test 2: Correct length (95223283)
Speedup       : 1.60
Speedup no I/O: 2.10

Test 3: Correct length (89211066)
Speedup       : 1.11
Speedup no I/O: 2.02
root@node1-1:~/MinSpanTree_Kruska
```

```
SERIAL (1%):
SERIAL (5%):
SERIAL (10%):
SERIAL (20%):
MPI (1%): 32 procs
MPI (5%): 32 procs
MPI (10%): 32 procs
MPI (20%): 32 procs

Test 0: Correct length (96785252)
Speedup       : 0.65
Speedup no I/O: 0.66

Test 1: Correct length (96750818)
Speedup       : 2.20
Speedup no I/O: 2.66

Test 2: Correct length (95223283)
Speedup       : 0.86
Speedup no I/O: 1.66

Test 3: Correct length (89211066)
Speedup       : 0.88
Speedup no I/O: 3.01
root@node1-1:~/MinSpanTree_Kruskal
```

Summary

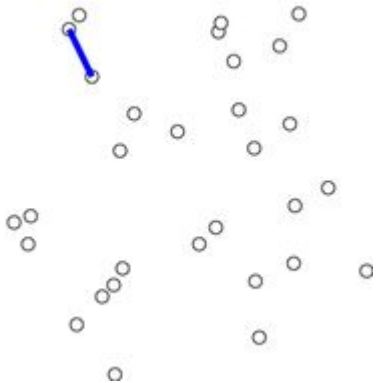
- As we faced several issues with openMPI integration we lost lot of time and could not run many tests.
- From the results we gathered, we conclude that as the Krushkal's algorithm is not inherently parallel, only the sorting of edges can be parallelized.
- Therefore, multiple threads might not always result in linear speedup as time would wasted in synchronizing threads.
- Also, the number of threads incorporated should be evaluated first based on the number of vertices and the graph density otherwise it could be waste of resources

Prim’s Algorithm

Concept

Overview of the algorithm:

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.



Project : Parallel implementation of Prims algorithm and comparison with the sequential implementation

Approach:

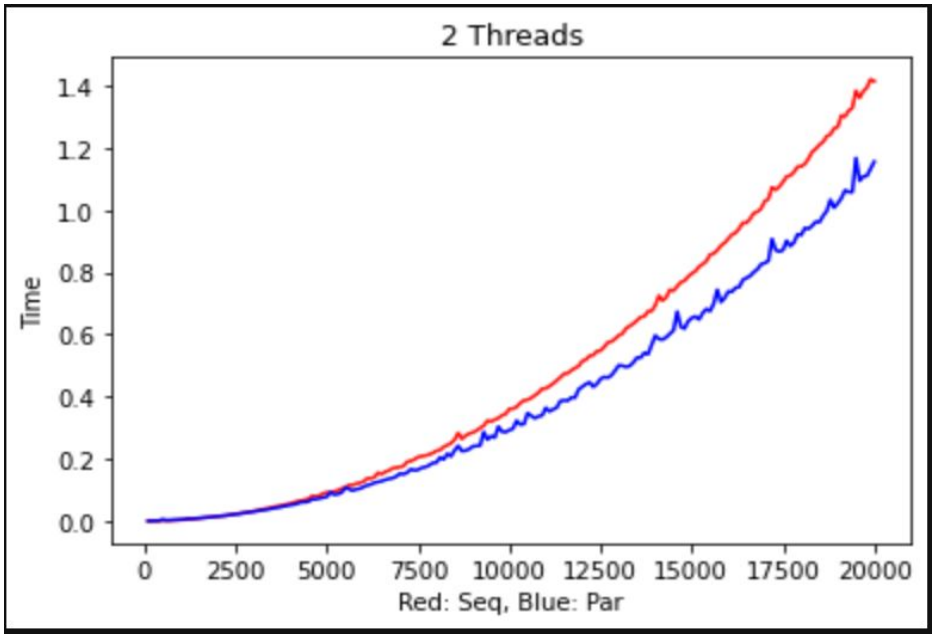
1. We have a sequential implementation and a parallel implementation of Prims MST made using OpenMP.
2. We have a benchmark file coded that takes in a random sized input and processes the points through both the parallel and sequential implementations and gets the time difference of the implementation. We run this for 2 threads and 4 threads.
3. The results are collected in a .csv file and are then processed through graphs so that we can visualize the output (time taken for sequential, time taken for parallel and time difference).

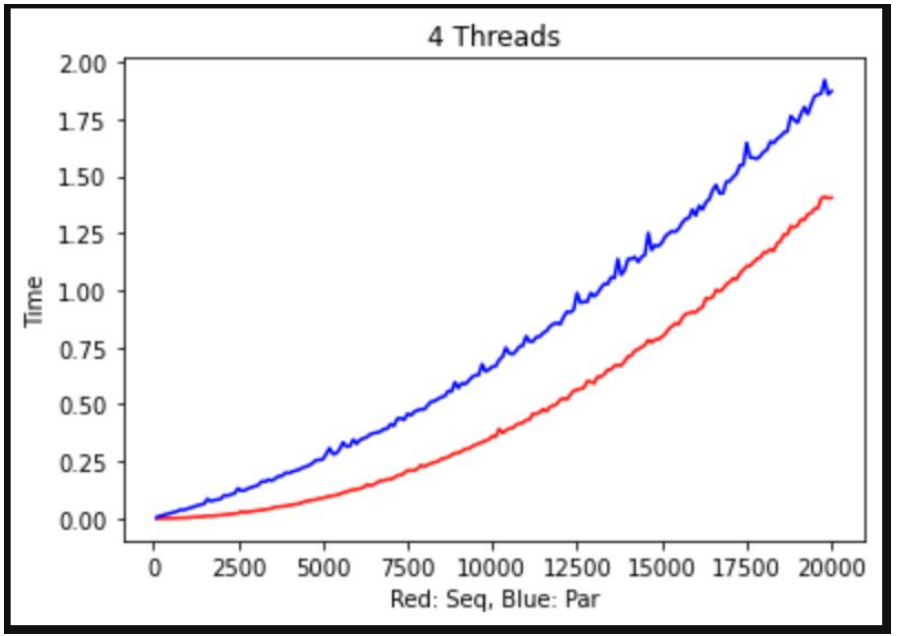
Continued...

Implementation:

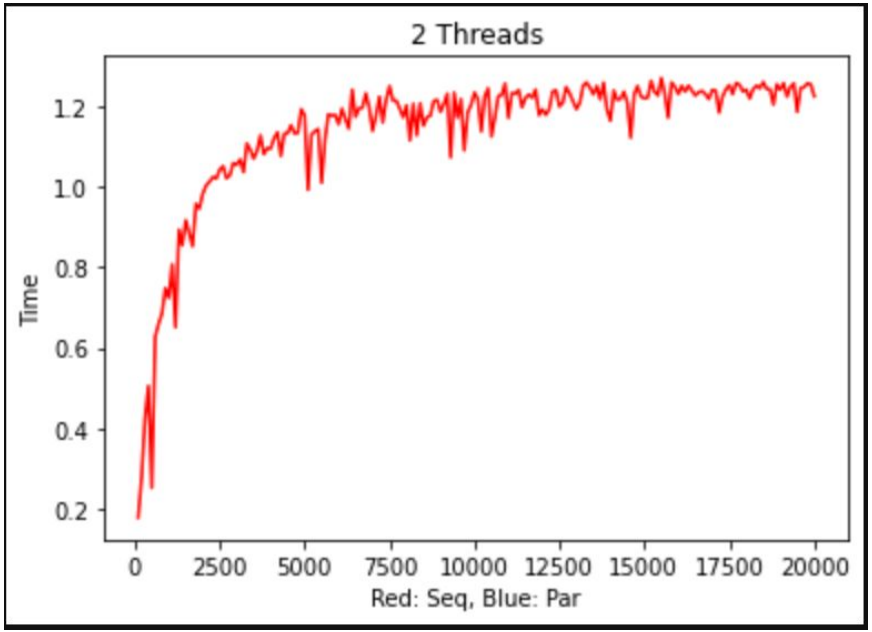
1. System specs: Intel(R) Xeon(R) CPU @ 2.30GHz No. of cores:1 No. of threads=2
2. Makefile with 2 compilation codes: 1 to compile the bench.c file and other to remove the resources (compiled objects) from the CPU core after finishing implementation
3. Bench.py python file that will call “make” to compile the bench.c with input size varying from 100 to 20000. This will loop through the iterations and store result in “*threads.txt”. Change is made in the `omp_set_num_threads(2)` parameter so that the number of threads used can be changed.
4. Stored .csv files as processed to pandas and the respective graphs can be plaotted to visualize the variation using the matplotlib library

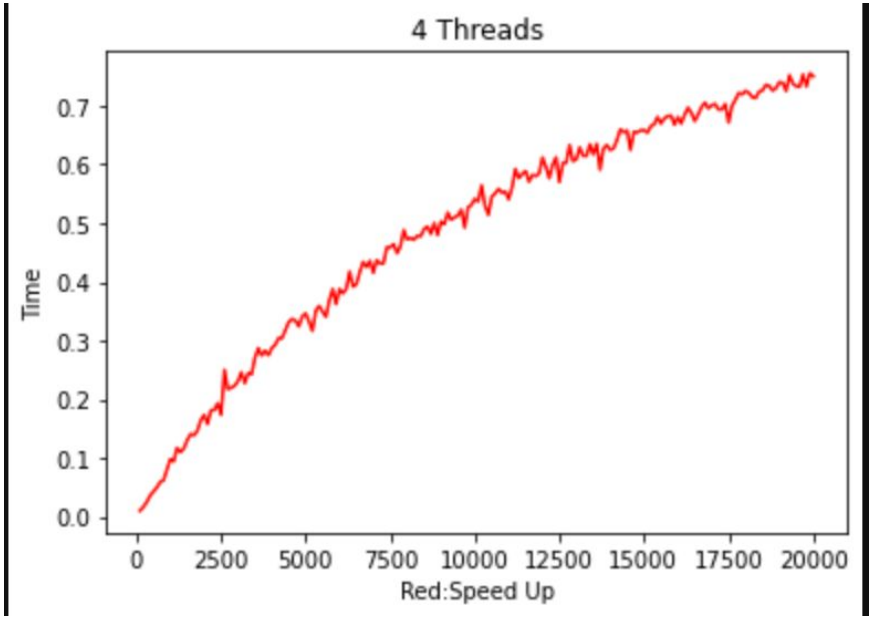
Results.....


2 Threads
Time
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
2500
5000
7500
10000
12500
15000
17500
20000
Red: Seq, Blue: Par


4 Threads
Time
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0
2500
5000
7500
10000
12500
15000
17500
20000
Red: Seq, Blue: Par

Results continued….





References

[1] R. C. Prim, “Shortest connection networks and some generalizations,”The Bell System Technical Journal, vol. 36,no. 6, pp. 1389–1401, 1957.

[2] J. B. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem,”Proceedings of the American Mathematical society, vol. 7, no. 1, pp. 48–50, 1956.

[3] B. A. Galler and M. J. Fisher, “An improved equivalence algorithm,”Communications of the ACM, vol. 7, no. 5,pp. 301–303, 1964.9

[4] R. E. Tarjan and J. Van Leeuwen, “Worst-case analysis of set union algorithms,”Journal of the ACM (JACM),vol. 31, no. 2, pp. 245–281, 1984.

[5] W. D. Frazer and A. McKellar, “Samplesort: A sampling approach to minimal storage tree sorting,”Journal of the ACM (JACM), vol. 17, no. 3, pp. 496–507, 1970.