University at Buffalo The State University of New York

CSE 633 – PARALLEL ALGORITHMS

Parallel implementation of Dijkstra's Single Source Shortest Path Algorithm

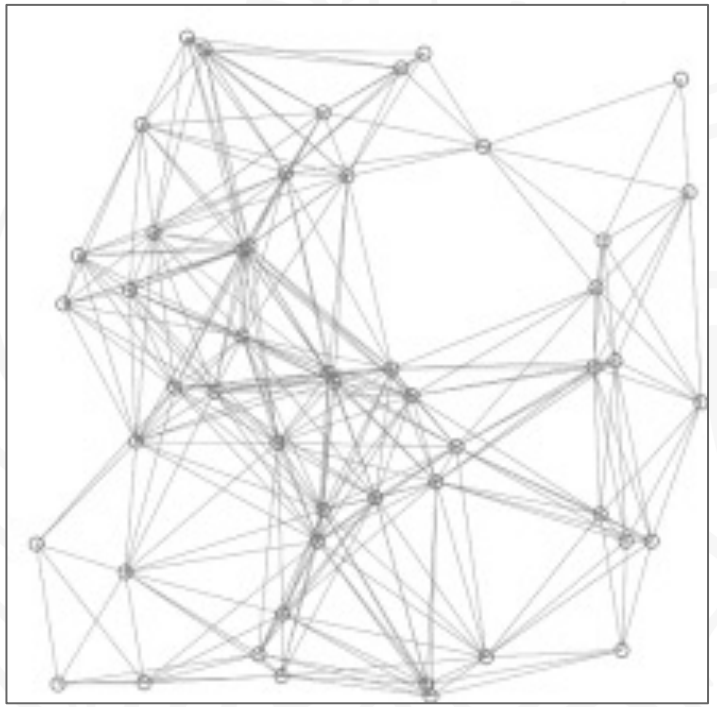
Kartik Sehgal

UB ID - 50466718

Contents

Shortest Path Problem and Dijkstra's algorithm

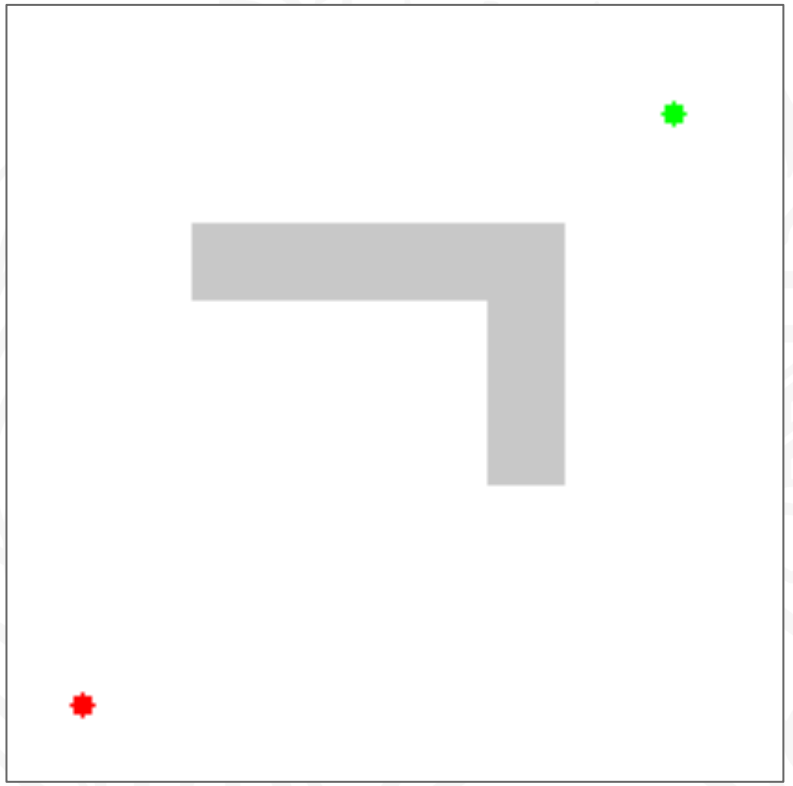
- In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.
- There are many variants of this problem, and in this presentation, we are going to focus on one of them – Dijkstra's algorithm.
- Dijkstra's algorithm finds the shortest path from one vertex, called the source vertex, to every other vertex in the graph.



Source: https://en.wikipedia.org/wiki/Dijkstra's_algorithm

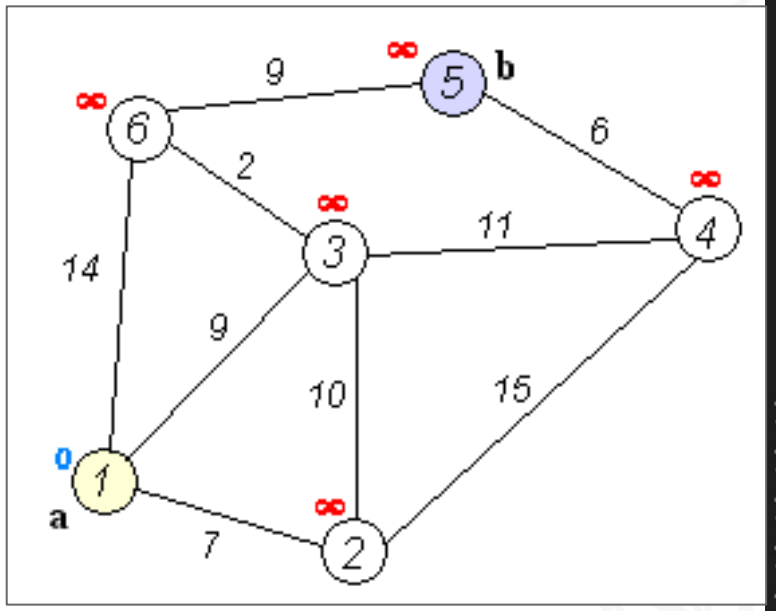
Applications of Dijkstra's algorithm

- Used to find the minimum distance between two destinations in Map applications
- Least cost paths are calculated to establish tracks of electricity lines or oil pipelines.
- Used to calculate optimal long-distance footpaths in Ethiopia and contrast them with the situation on the ground.
- Used in game theory, for example in Rubik's cube where each directed edge corresponds to a single move or turn, to solve the game in minimum possible turns.
- Used to find the minimum delay path in networking and telecommunication.



Sequential algorithm

- Initialize the distance all vertices to infinity.
- Initialize the distance of source vertex to 0.
- Initialize a visitedQueue Q and add source vertex to it.
- While Q is not empty, dequeue the first element u.
- For all neighbors of v of u, check if the distance from u to v is greater than the currently stored distance.
- If it is, update with a shorter distance and add it it to Q to process its neighbors.



Source: https://en.wikipedia.org/wiki/Dijkstra's_algorithm

```
1   for each vertex u in graph
2     dist[u] = INF
3   dist[source] = 0
4
5   initialize visitedQueue Q
6   add source vertex to Q
7   while Q is not empty
8     remove vertex u from Q
9
10    for each neighbor v of u
11      if d(v) > d(u) + l(u, v)
12        d(v) = d(u) + l(u, v)
13        insert v to Q
14  END
```

The need for parallelizing the algorithm

- The original sequential algorithm is slow, with a running time of $O(V^2)$, where V is the number of vertices.
- There are some optimizations available
  - Binary heap - O((E + V) log V)
  - Fibonacci heap - O(E + V log V)
- However, for a massive graph with millions of vertices, these approaches still take a long time.
- A parallel approach could further reduce the overall running time, improving the performance of the algorithm.
- A massive graph is hard to fit in memory of a single node. With a parallel approach, we can scale our algorithm easily.
- All computers these days are multicore systems. With a parallel implementation, we make sure that we are utilizing the resources of our system efficiently.

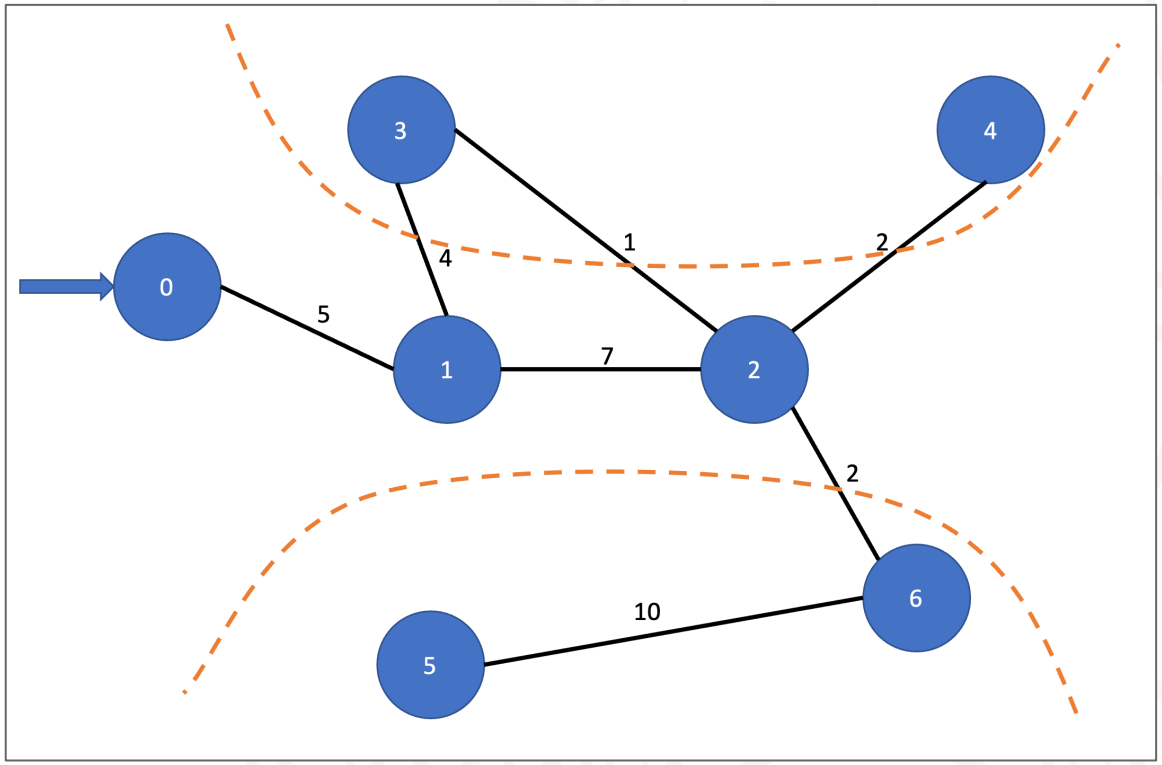
Parallel Approach

- The first step is to divide the graph into subgraphs
  - We try to divide the number of vertices evenly in all processors, i.e., n/p vertices per processor.
    - If there is an uneven distribution, we divide the remaining ‘k’ vertices evenly amongst the first ‘k’ processors.
- Each processor is responsible for computing the shortest path for its assigned vertices only.
- Each processor stores 3 data structures
  1. The adjacency columns for each of its vertices with size = N/P * N
  2. A distances array storing the current distances for its vertices with size = N/P * 1
  3. (P-1) number of MessageArrays to send information to the other processors & 1 MessageArray to receive information from the other processors with size = N/P * P

```
1   for each process parallel do
2     if processrank == 0
3       partition the graph
4       allocate respective memory for the data structures
5       Send graph partition information to all processes
6
7     Initialize visitedQueue Q
8     insert source node to Q
9     while true do
10      while Q is not empty
11        remove first node from Q
12        for each adjacent node v of u in current subgraph
13          if d(v) > d(u) + l(u, v)
14            d(v) = d(u) + l(u, v)
15            insert v to Q
16
17        send = false
18        for each adjacent subgraph g
19          for each adjacent node n in g of node m
20            if d(n) > d(m) + l(m, n)
21              add information of node n to MessageArray[g]
22              send = true
23
24      if send is not false
25        for each adjacent subgraph g
26          send MessageArray[g] to every adjacent subgraph g
27          receive MessageArray from subgraph g
28
29        for each received node y with adjacent node in current process x
30          if d(y) > d(x) + l(x, y)
31            d(y) = d(x) + l(x, y)
32            insert y to Q
33      else
34        break from while(true)
35
36    if processrank == 0
37      get results from all other processes
38  END
```

Parallel Approach

- Computation step
  - Each processor first finds local optimal shortest paths in each sub-graph using the sequential Dijkstra algorithm.
  - If the currently calculated distance of the successor of the boundary node in the adjacent subgraph is greater than the distance from the boundary node to the successor node, we need to update its distance.
  - This information, called boundary information, is collected for exchanging in the Communication step. If there is some exchange required, we store it in the MessageArray for that processor.
- Communication Step
  - If there is some exchange required, we send and receive the boundary information with the adjacent subgraphs.
  - We update the shortest path of the above nodes.



Parallel Approach

- The Computation step and the Communication step are repeated until we establish that the boundary nodes don't need to update the distances in the adjacent subgraphs.
- The algorithm terminates and the source node gets the results from all other processes.
- MPI functions used –
  - MPI_Send
  - MPI_Receive
  - MPI_Scatter
  - MPI_Gather
  - MPI_Bcast

	0	1	2	3	4	5	6
0	0	5	0	4	0	0	0
1	5	0	7	0	0	0	2
2	0	7	0	1	2	0	0
3	4	0	1	0	0	0	0
4	0	0	2	0	0	0	0
5	0	0	0	0	0	0	10
6	0	2	0	0	0	10	0

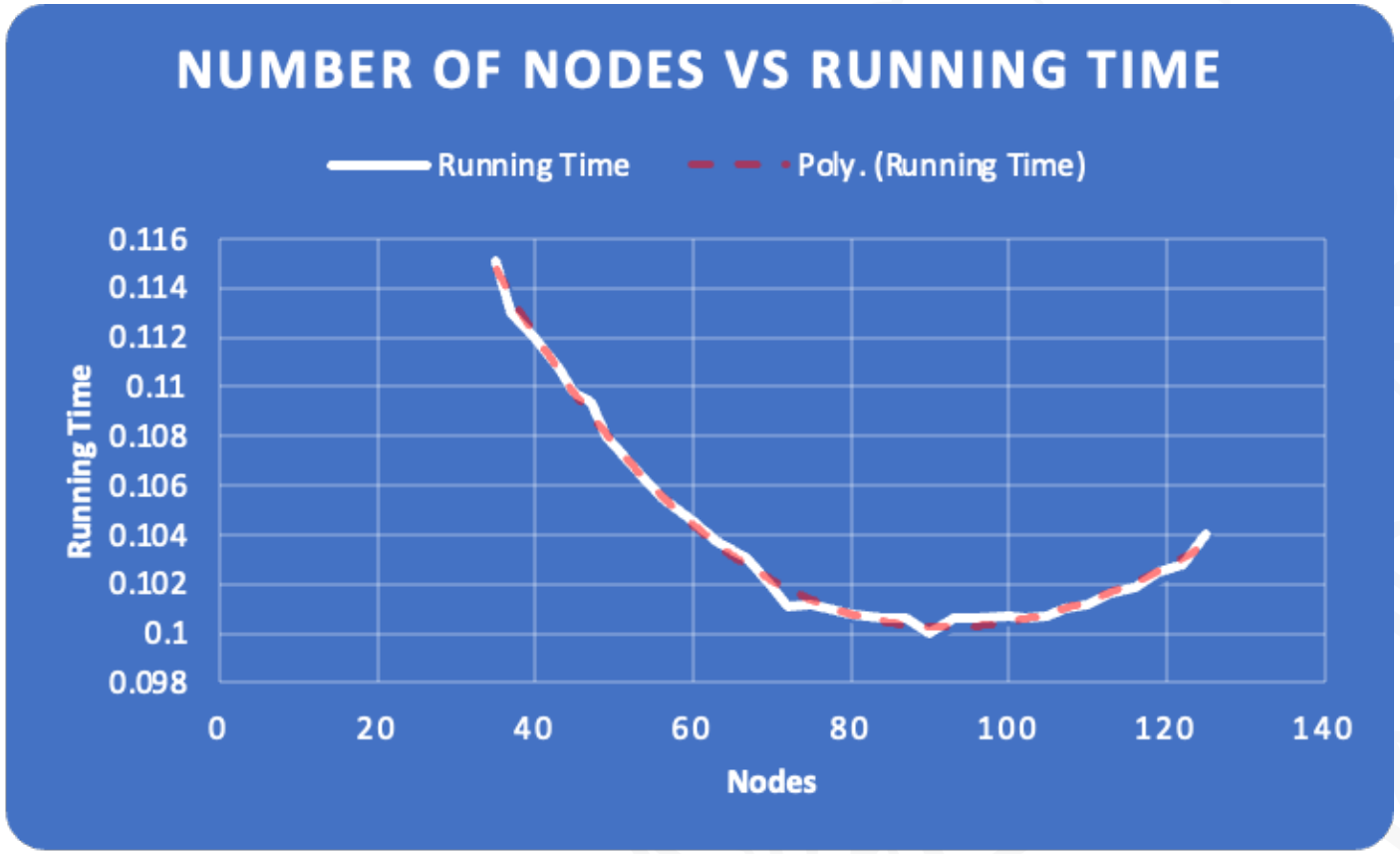
Tasks achieved since midterm

- Work on fixing the Slurm script. ✓
- Increase the number of processors. ✓
- Increase the size of the graph. ✓
- Obtain results for other parameters like speed up factor, sequential v/s parallel execution. ✓
- Try to remove the outliers in the execution result. ✓

Execution results (Running time)

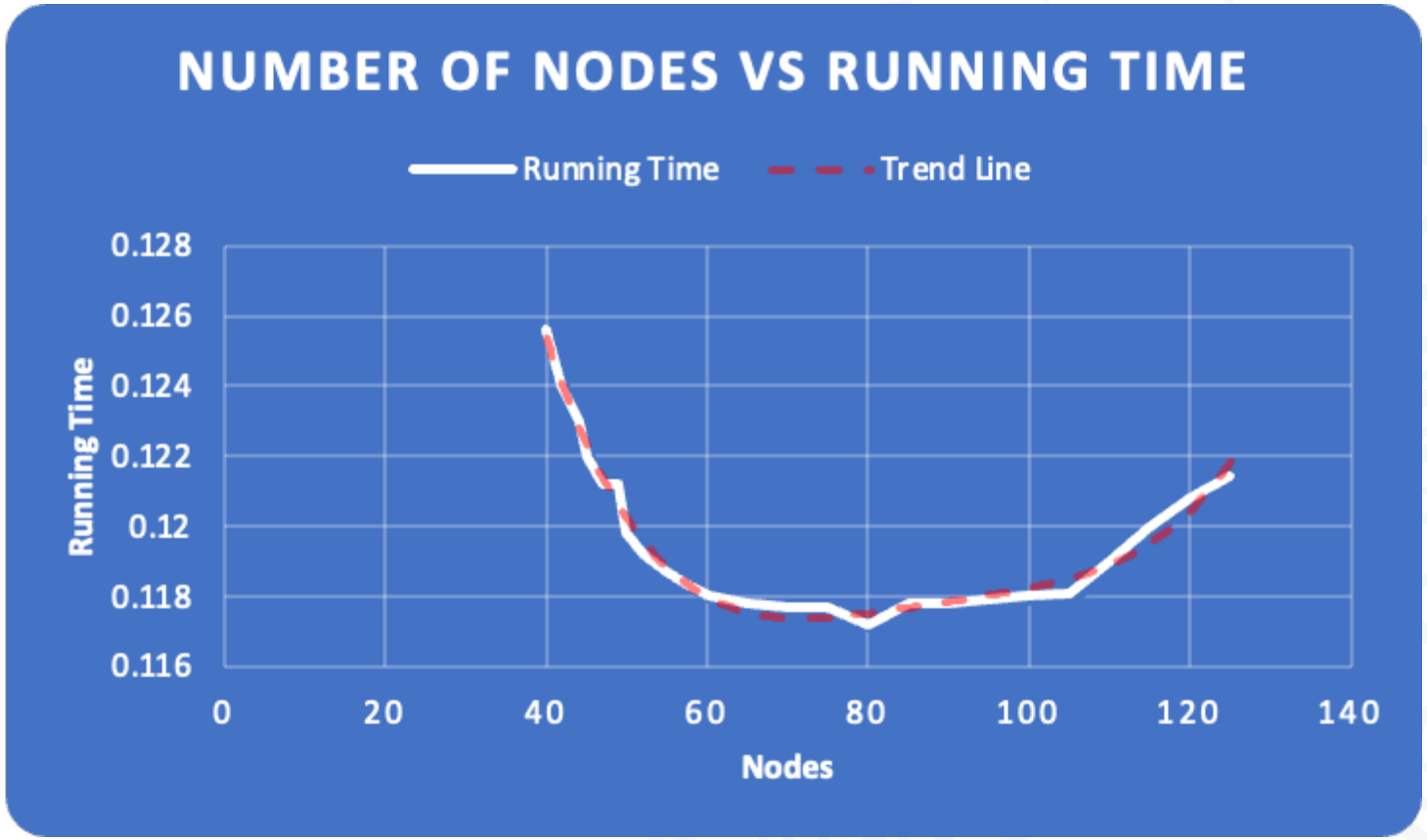
- Number of nodes up to 125.
- Number of vertices up to 1400.
- Results of
  - Running time v/s no of nodes.
  - Sequential vs parallel running time.
  - Speed up.
  - Cost.

1000 vertices



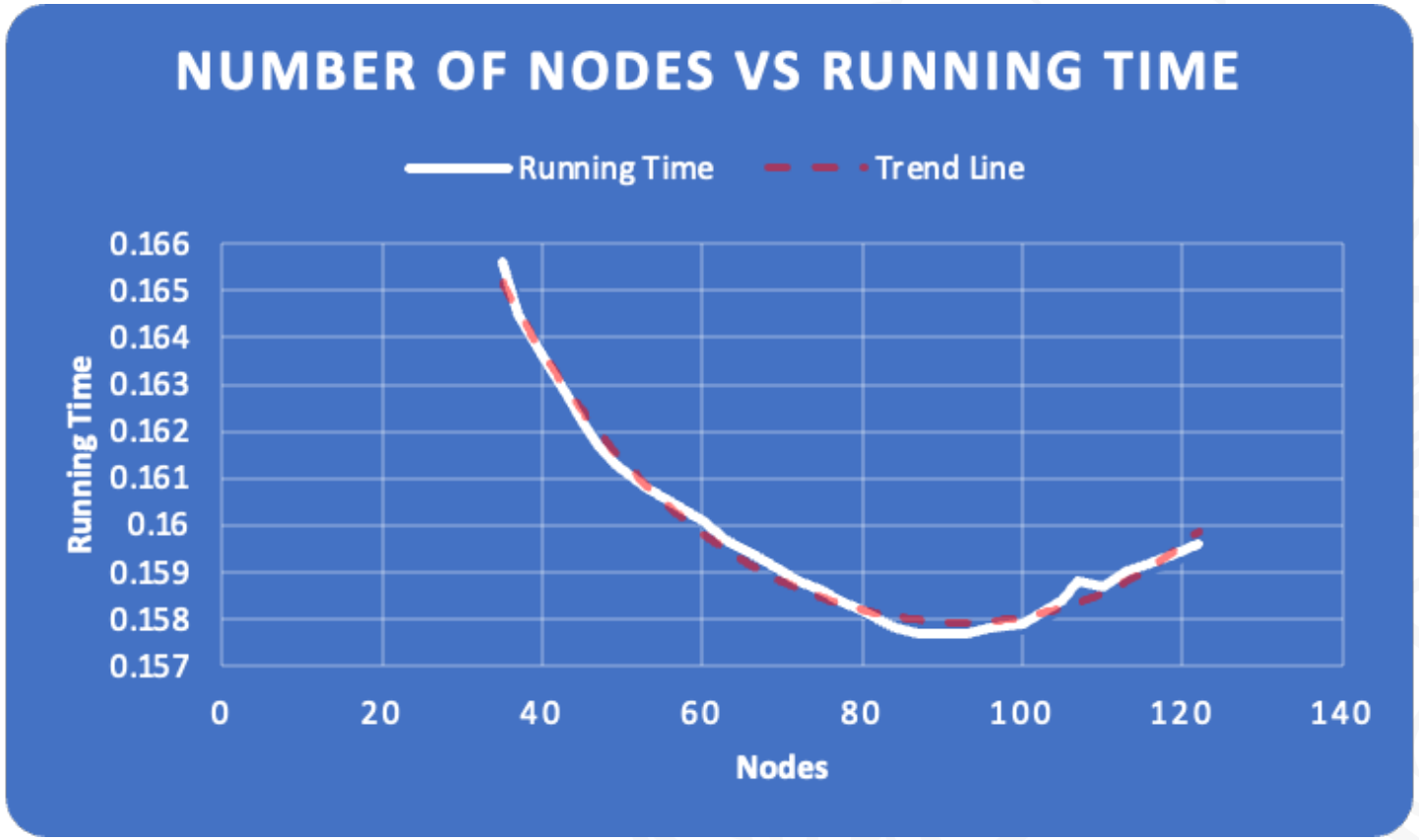
Execution results (Running time)

1200 vertices



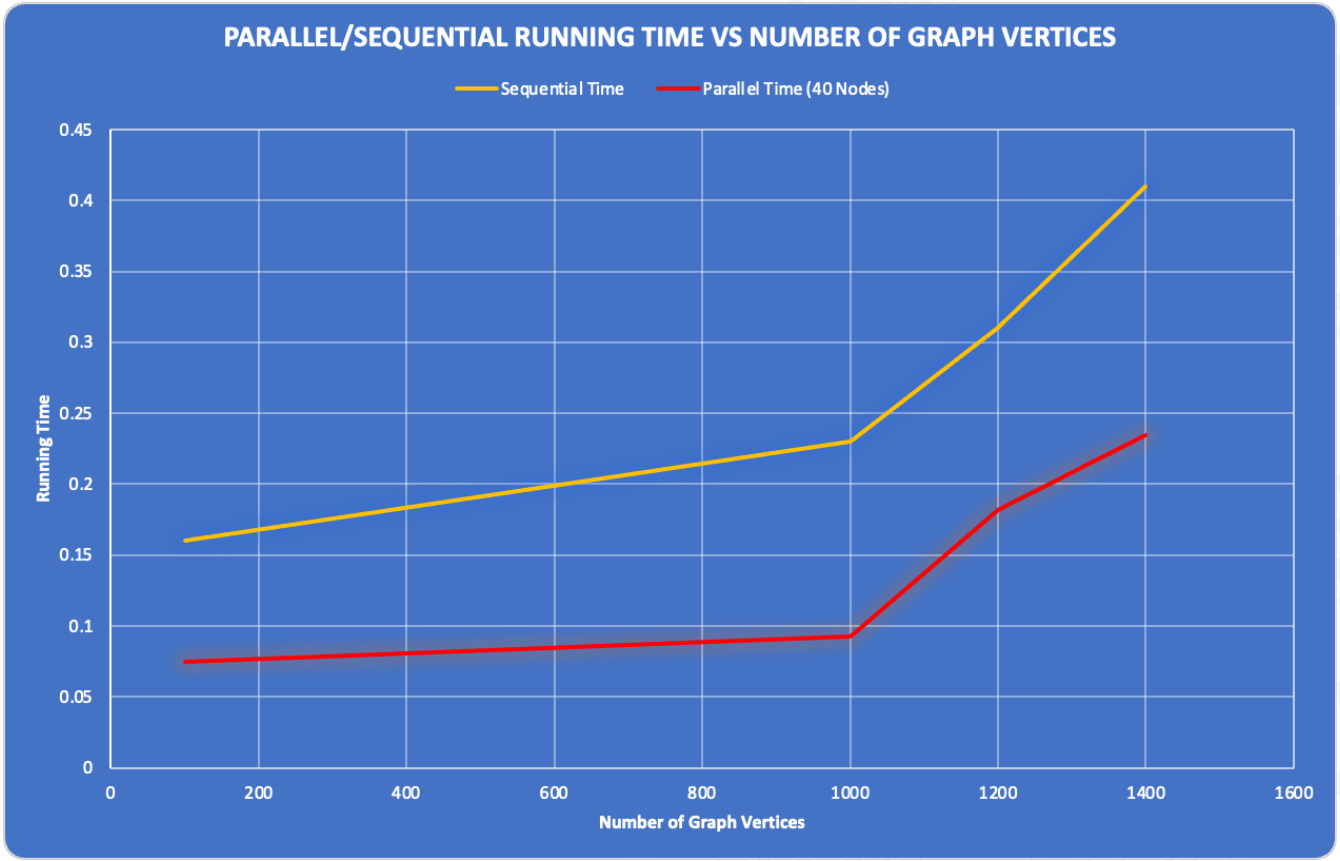
Execution results (Running time)

1400 vertices



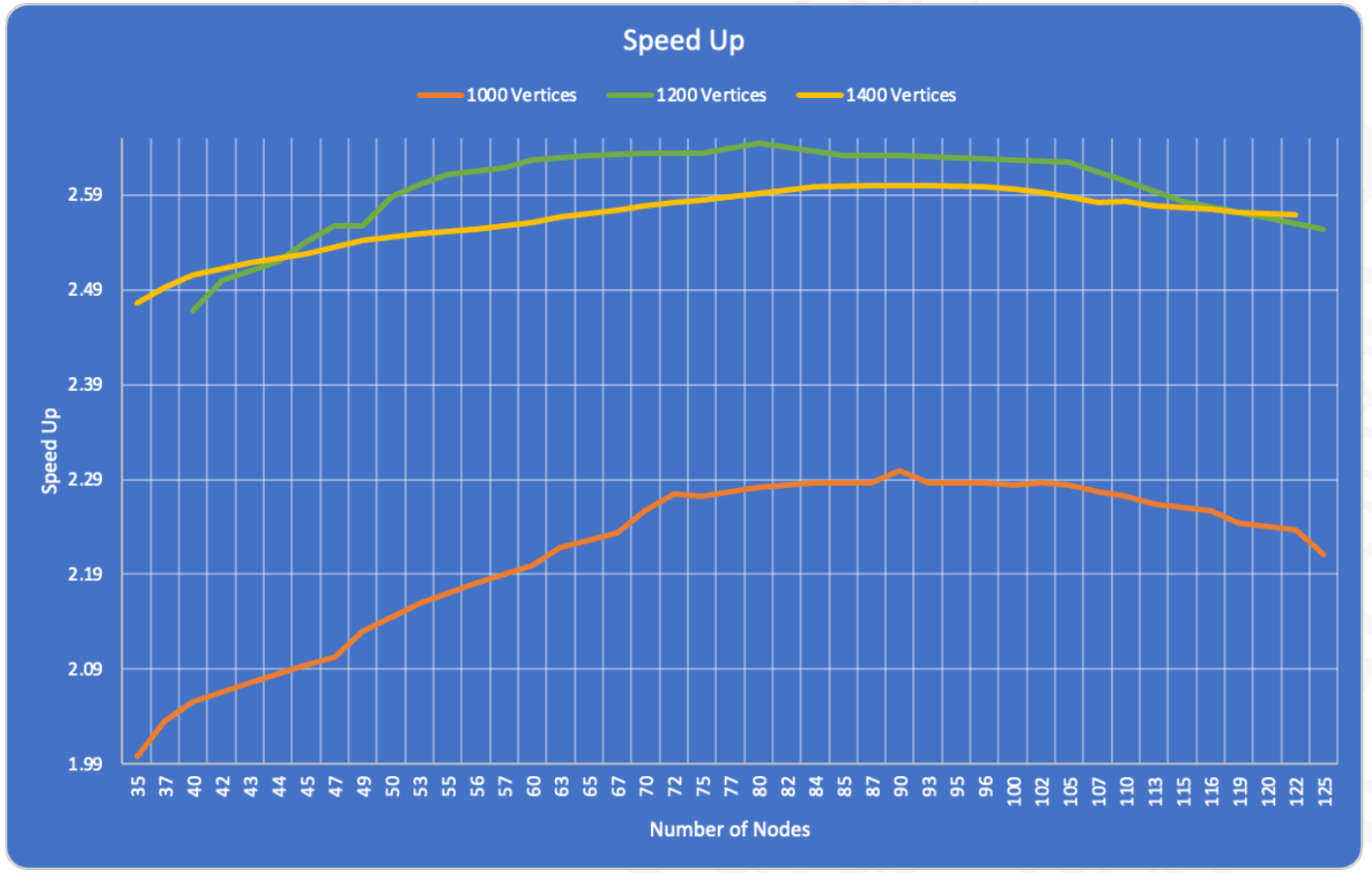
Execution results (Sequential vs Parallel Running Times)

- Varied the number of vertices in the graph, keeping the number of nodes constant.



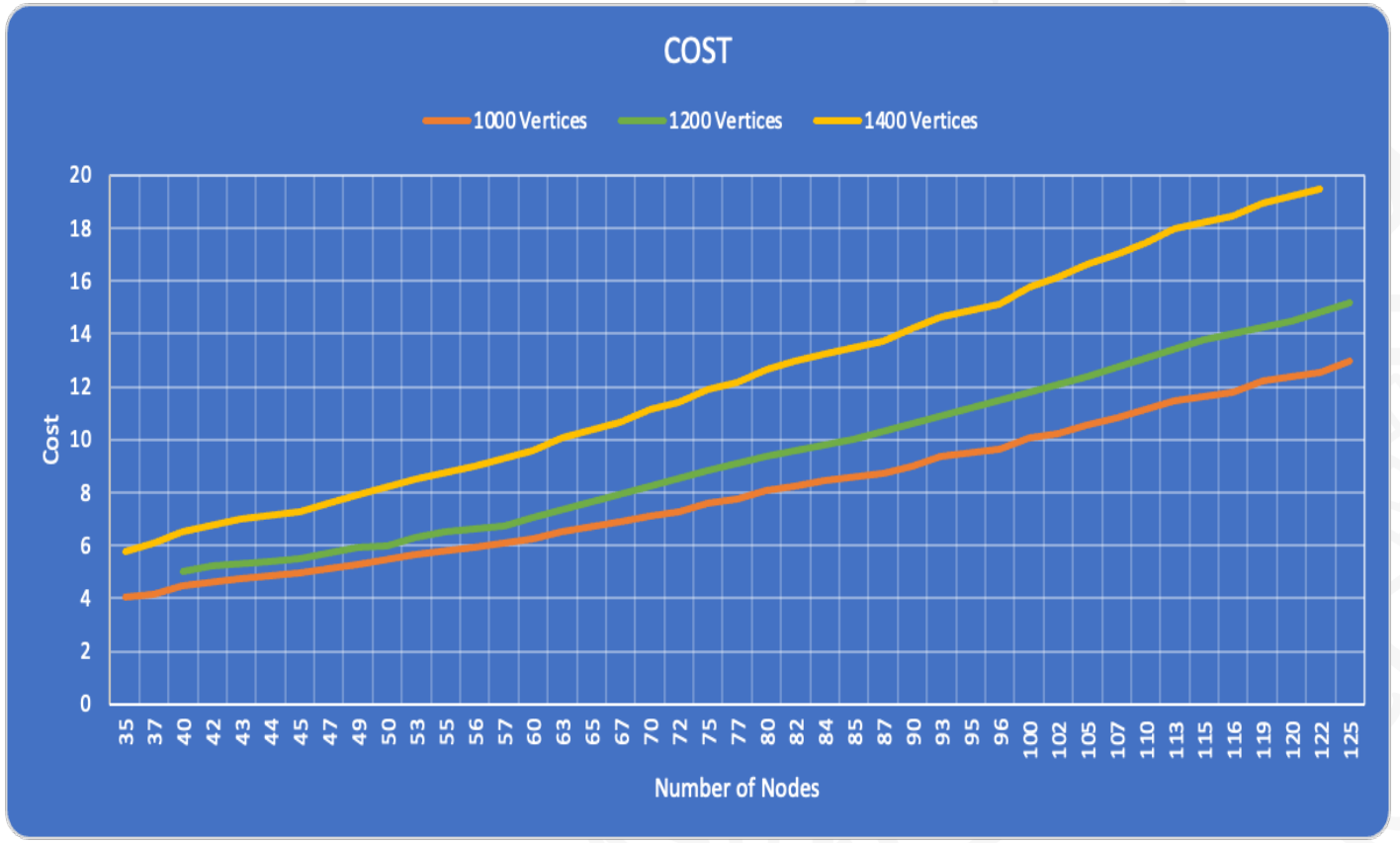
Execution results (Speed-up)

- Speed-up is defined as
  - $S = T_{seq} / T_{par}$



Execution results (Cost)

- Cost is defined as
  - $C = P * T_{par}$



References

- Dijkstra, E. W. (1959). "A note on two problems in connection with graphs,". Numerische Mathematik 1: 269–271. doi:10.1007/BF01386390.
- Y. Tang, Y. Zhang, H. Chen, "A Parallel Shortest Path Algorithm Based on Graph- Partitioning and Iterative Correcting", in Proc. of IEEE HPCC'08, pp. 155-161, 2008.
- G.Stefano,A.Petricola,C.Zaroliagis,"On the implementation of parallel shortest path algorithms on a supercomputer", in Proc. of ISPA'06, pp. 406-417, 2006.
- A.Crauser, K.Mehlhorn, U.Meyer, P.Sanders, "A parallelization of Dijikstra's shortest path algorithm", in Proc. of MFCS'98, pp. 722-731, 1998.
- https://en.wikipedia.org/wiki/Shortest_path_problem#CITEREFThorup1999
- https://en.wikipedia.org/wiki/Dijkstra's_algorithm
- https://en.wikipedia.org/wiki/Parallel_single-source_shortest_path_algorithm