Graphs at Pace: Profiling Parallel Dijkstra's Algorithm in HPC

Kiran Radhakrishnan

CSE 633 Spring 2024

Introduction

Dijkstra's algorithm is a method for finding the shortest paths between nodes in a graph, by iteratively selecting the node with the lowest known distance from the start node and updating the distances to its neighbors.

Introduction

Dijkstra's algorithm was conceived by computer scientist Edsger W. Dijkstra in 1956.

This algorithm finds the shortest distance from a source node to all the other nodes in a given weighted graph.

It is ideal for solving the single source shortest path problem.

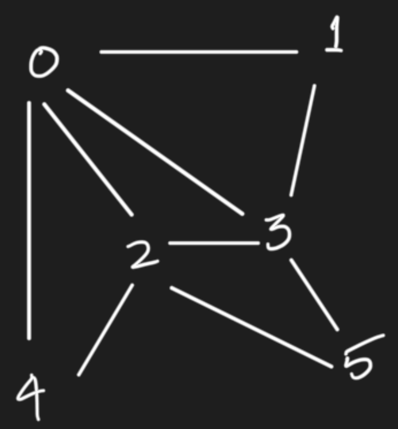
Common uses

- IP routing to find Open shortest Path First.
- Google maps for navigation
- Delivery route optimization
- Warehouse robot pickers
- Modeling biological network pathways

Dijkstra’s Algorithm Sequential

This algorithm finds the shortest path from a source node to other nodes in a graph by sequentially going over temporary distances to each node, picking the nearest node and then updating the distance to its neighbors. This step is repeated till all nodes are visited.

Adjacency Matrix

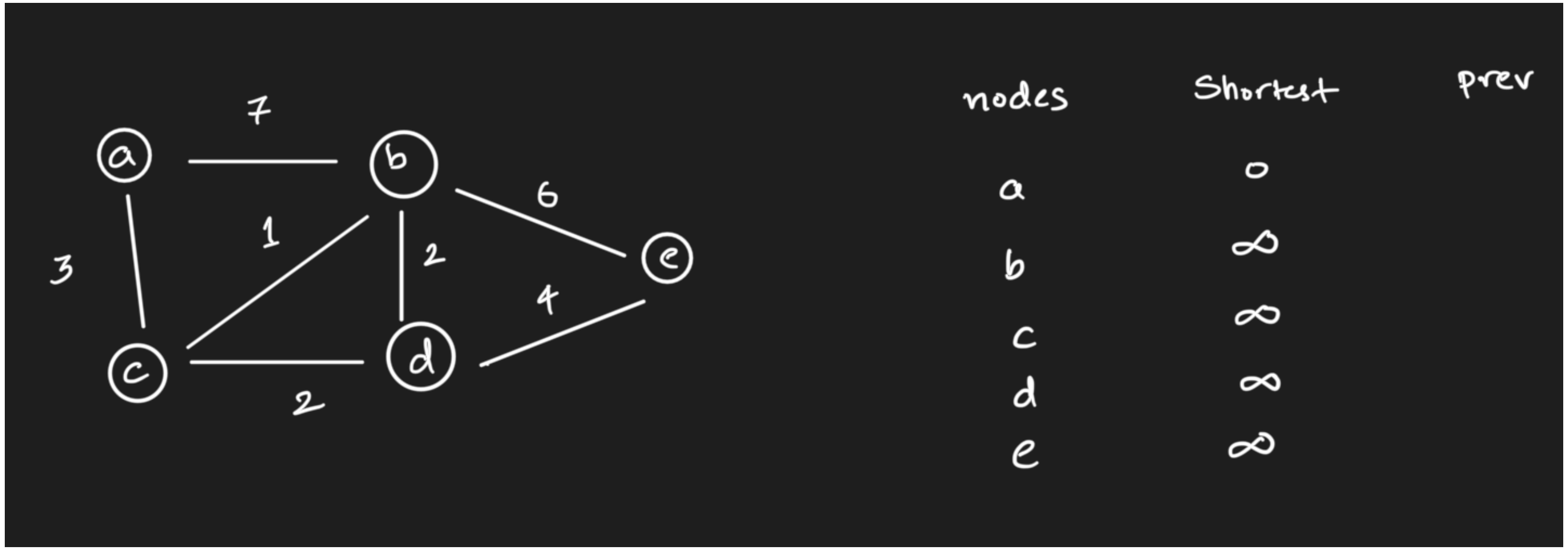


G	0	1	2	3	4	5
0	0	1	1	1	1	0
1	1	0	0	1	0	0
2	1	0	0	1	1	1
3	1	1	1	0	0	1
4	1	0	1	0	0	0
5	0	0	1	1	0	0

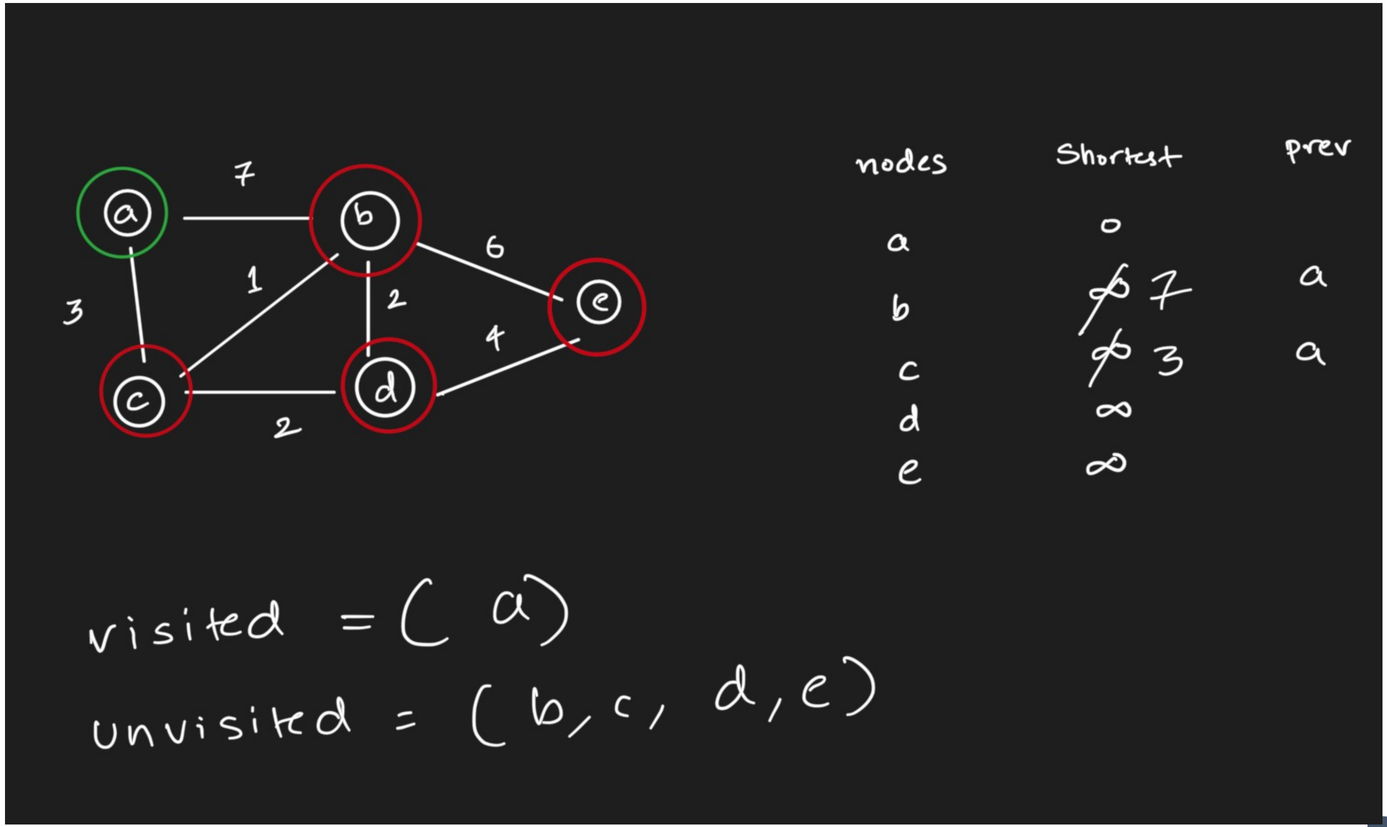
Sequential Pseudocode

```
function Dijkstra(Graph, source):
    dist[] := array of distances initialized to infinity for all nodes
    prev[] := array of predecessors initialized to NULL for all nodes
    visited[] := array initialized to false for all nodes
    dist[source] := 0

    while there are unvisited nodes:
        u := node with the minimum distance in dist[] among unvisited nodes
        mark u as visited

        for each neighbor v of u:
            if v is not visited and dist[u] + weight(u, v) < dist[v]:
                dist[v] := dist[u] + weight(u, v)
                prev[v] := u

    return dist[], prev[]
```

Example:

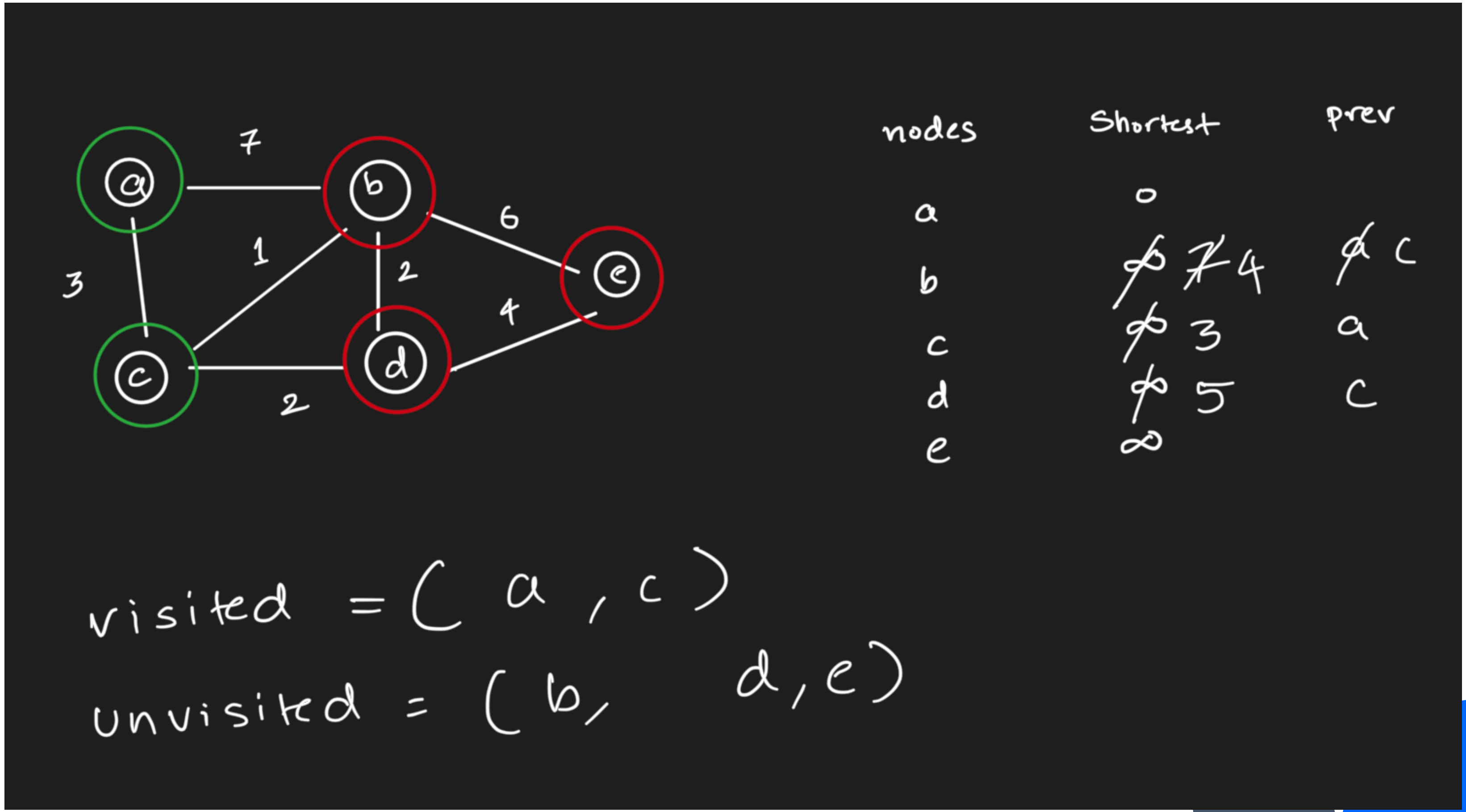


Example:



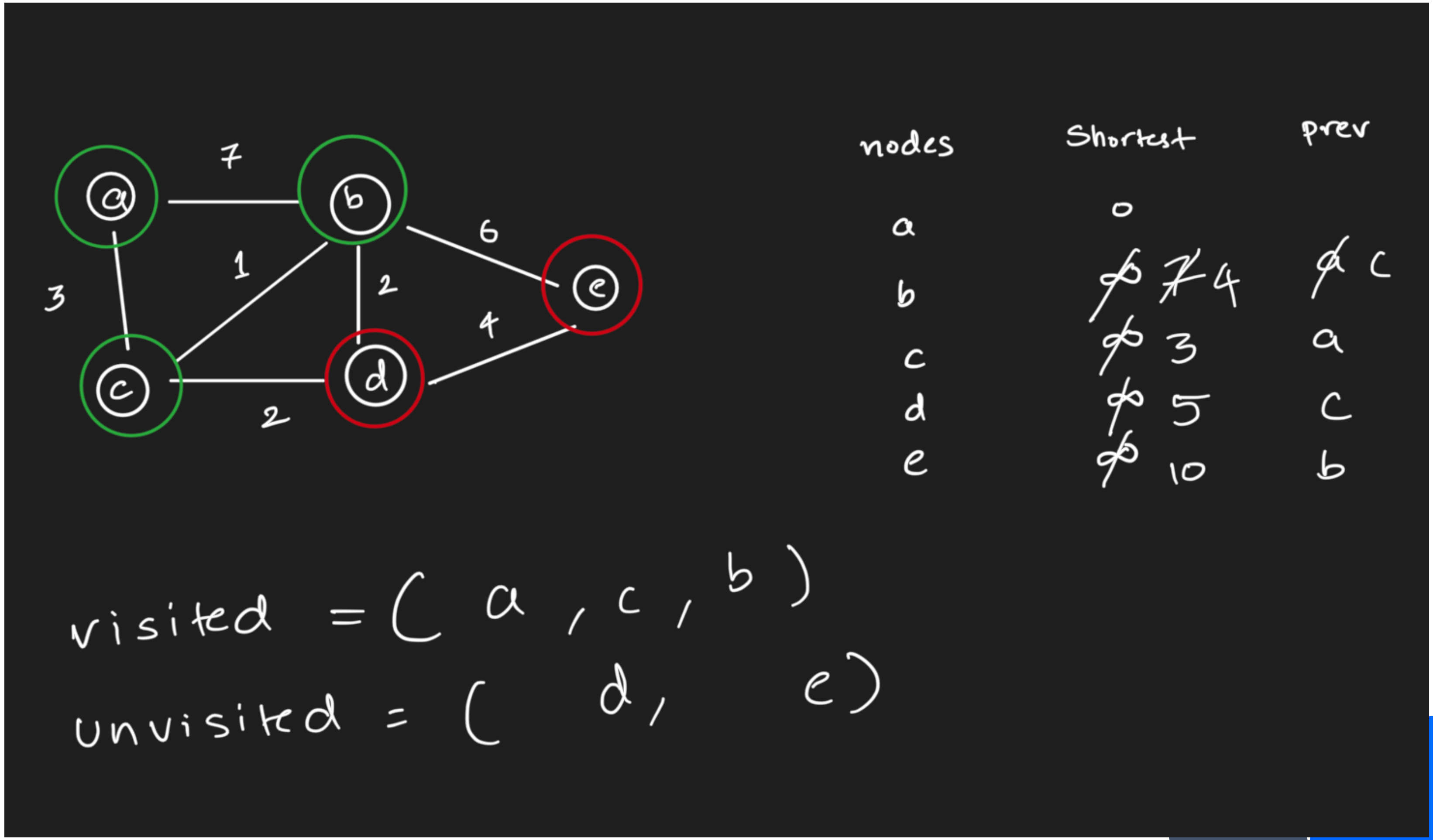
7
6
3
1
2
4
2
a
b
c
d
e
nodes
Shortest
prev
a
0
b
~~0~~ 7
a
c
~~0~~ 3
a
d
∞
e
∞
visited = (a)
unvisited = (b, c, d, e)

Example:

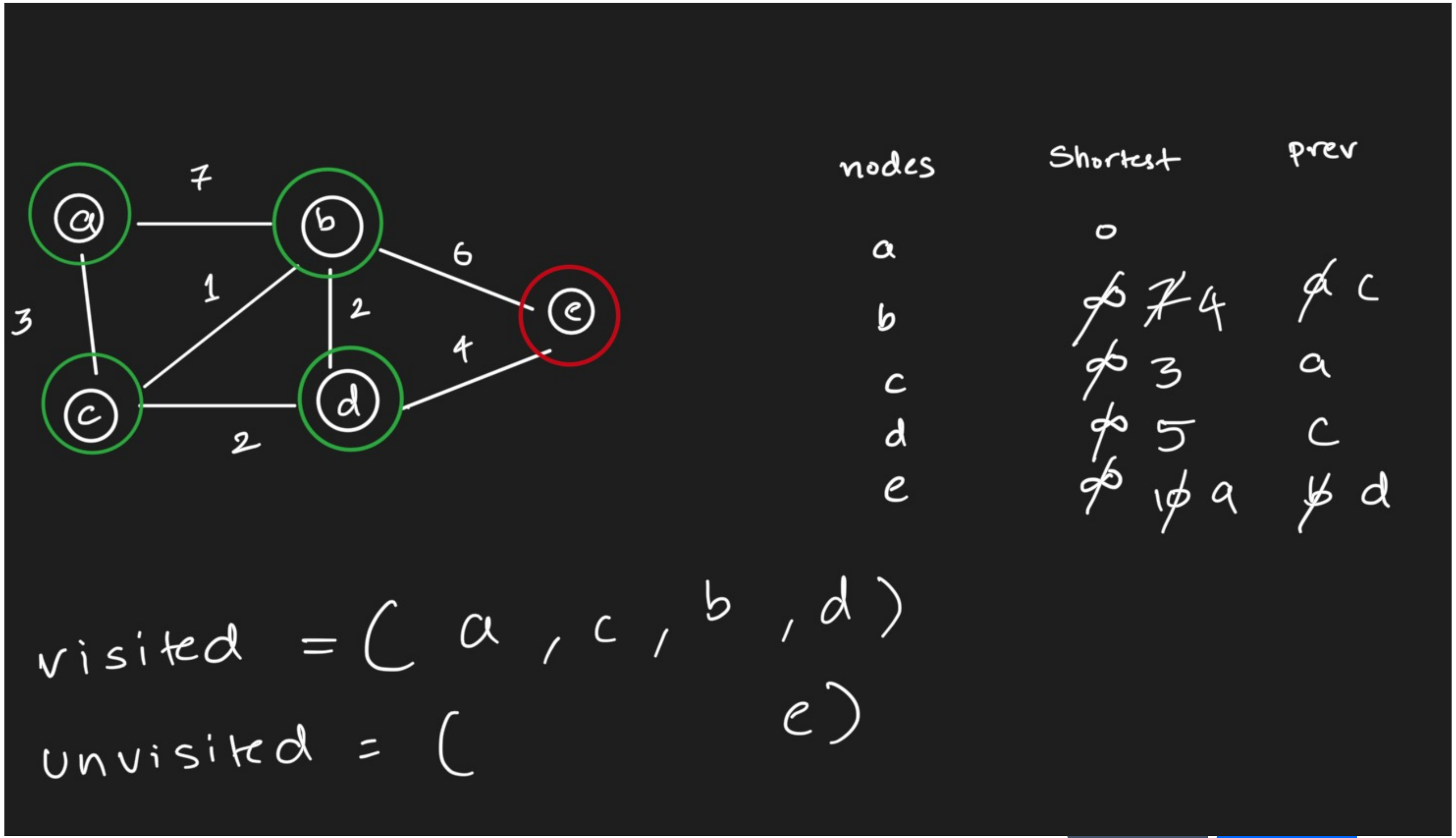


a
b
c
d
e
7
1
6
3
2
4
2
nodes
Shortest
prev
a 0
b 4 c
c 3 a
d 5 c
e ∞
visited = (a , c)
unvisited = (b, d, e)

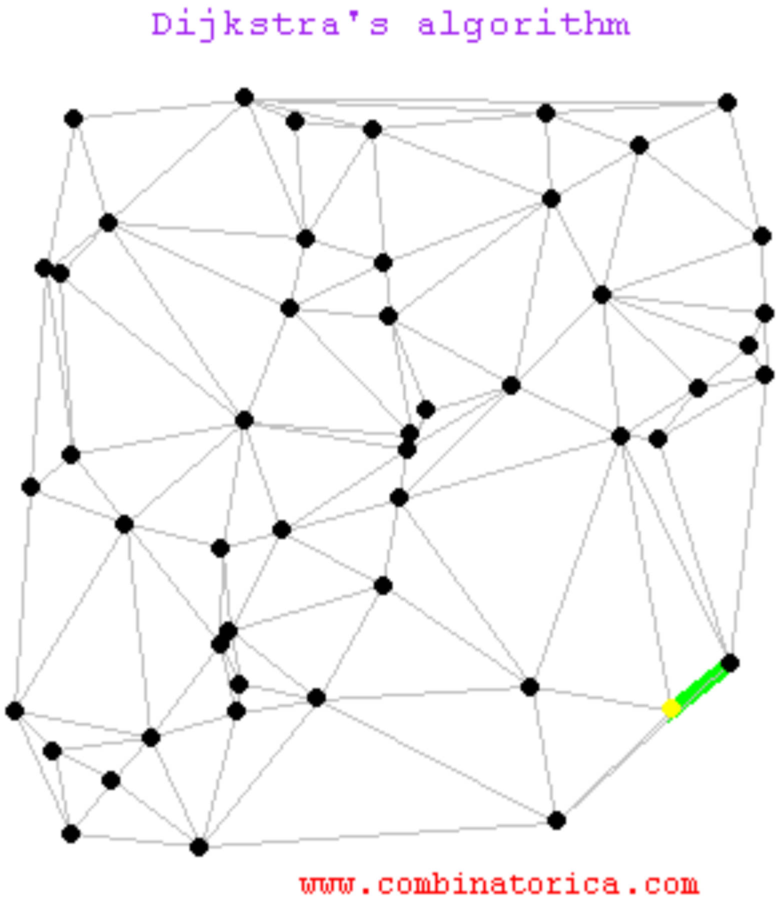
Example:


7
6
3
1
2
4
2
nodes
Shortest
prev
a
b
c
d
e
0
visited = (a , c , b)
unvisited = (d , e)

Example:



a
b
c
d
e
7
1
3
2
2
6
4
nodes
Shortest
prev
a
0
b
~~∅~~ ~~7~~ 4
~~∅~~ c
c
~~∅~~ 3
a
d
~~∅~~ 5
c
e
~~∅~~ ~~10~~ 9
~~∅~~ d
visited = (a , c , b , d)
unvisited = (e)



https://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/dijkstra.html

Parallel Dijkstra's approaches

Two approaches:

- Single source (Vertex Centric)
  - Multiple sources across each processor
  - Less communication required(Not the objective of this course)
  - Better for multiple source path computation
- Shared source (Edge Centric)
  - A single source used by all processors
  - More communication required to find global minimum.

Parallel Dijkstra’s

1. Initialize MPI environment.

2. Read the number of vertices 'n' from command line argument.

3. Broadcast 'n' to all processes.

4. Calculate the local size of matrix.

5. Allocate memory for local matrices, distances, and predecessors.

7. Initialize random weight matrix on process 0 and broadcast it.

8. Perform Dijkstra's initialization locally:

   - Set the known status of the source node to 1.

   - For all other nodes, set known status to 0.

   - Initialize distances from source node and predecessors.

9. Execute Dijkstra's algorithm :

   - Find the node with the minimum tentative distance among unvisited nodes.

   - Share the minimum distance globally.

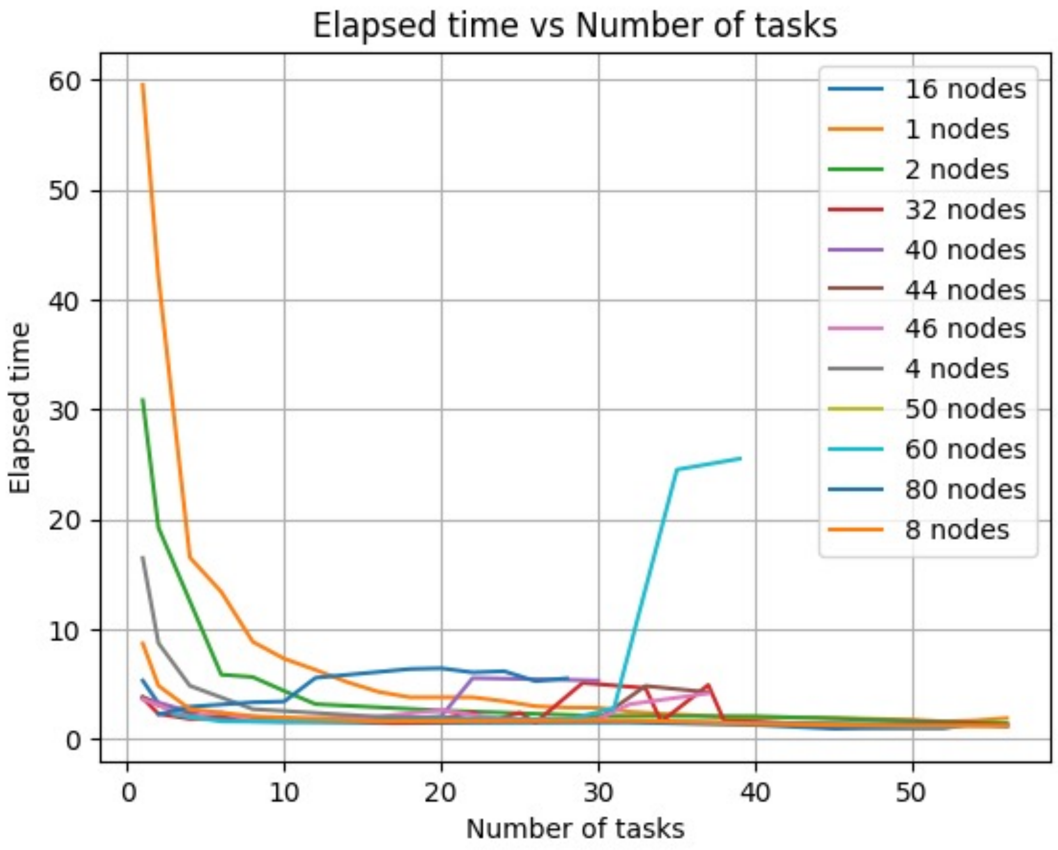
   - Update local distances and predecessors based on global minimum distances.

10. Gather local distances and predecessors to process 0.

11. Print global distances and paths.

12. Finalize MPI environment.

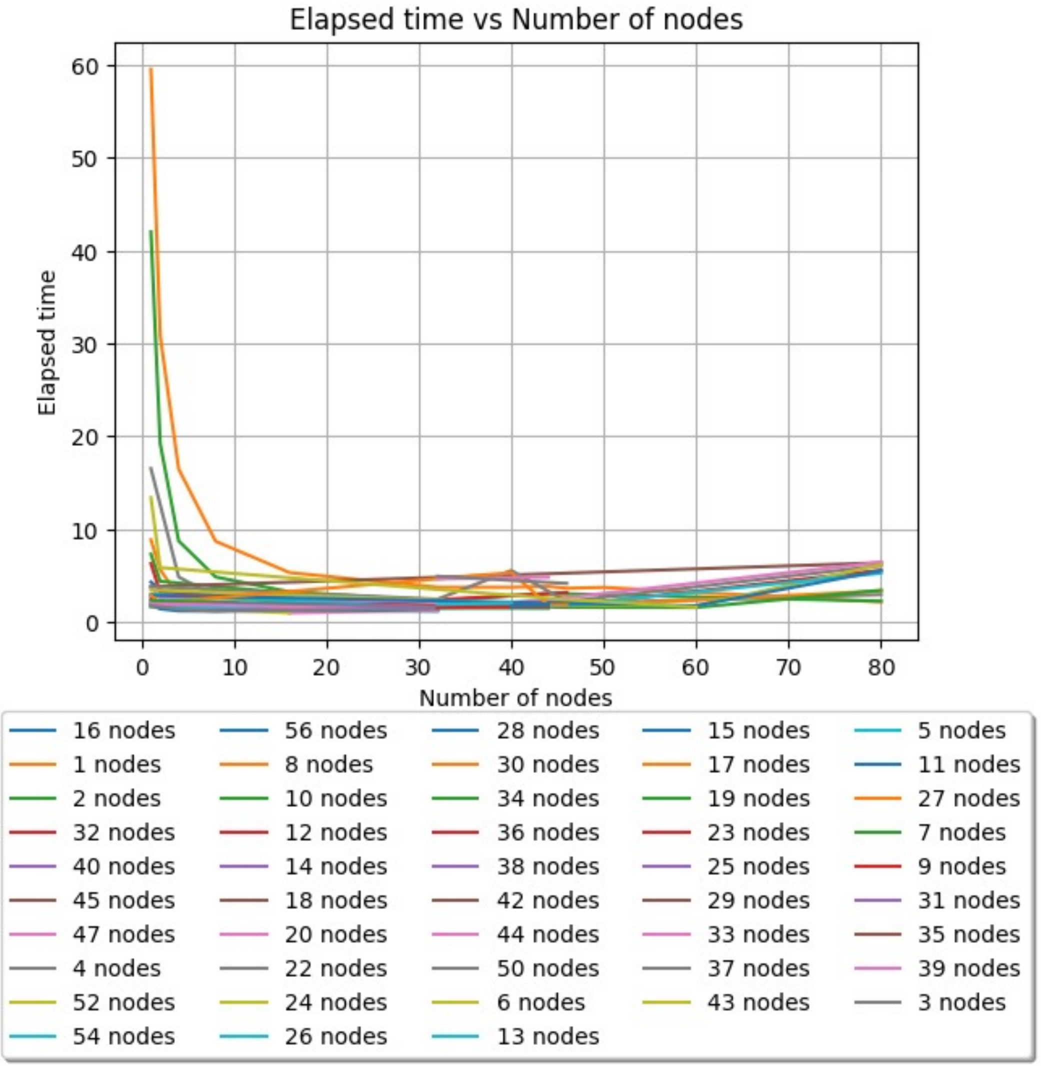
Overall performance with 40000 vertices



For Reference:
# of tasks = # of Cores per node

Total # of cores =
# of nodes * # of tasks per node

Overall performance with 40000 vertices

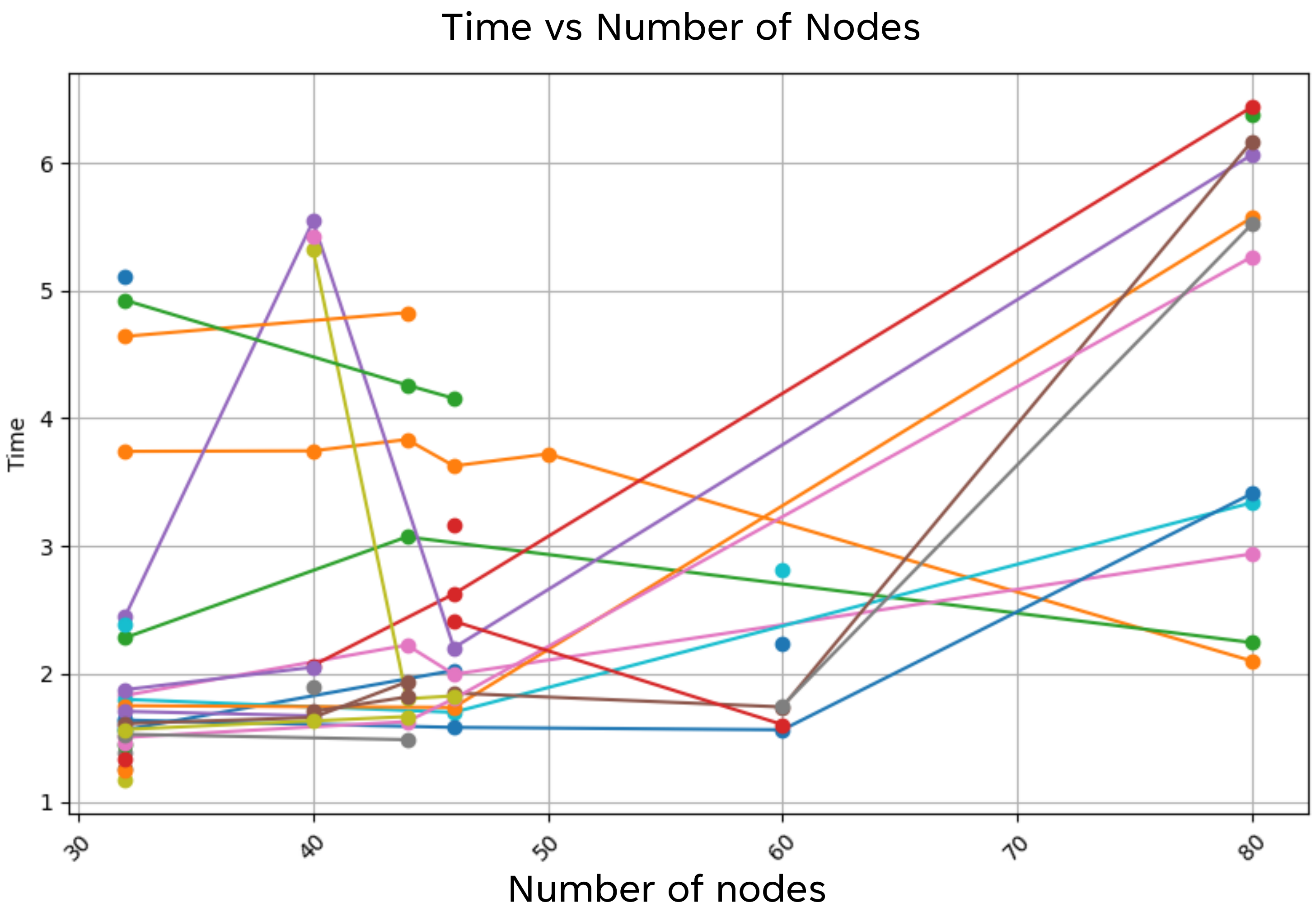


For Reference:
\# of tasks = # of Cores per node

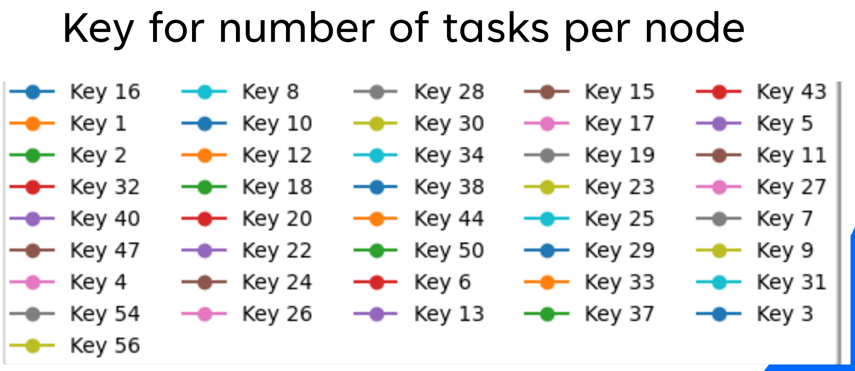
Total # of cores =
\# of nodes * # of tasks per node

Each line represents a constant number of nodes

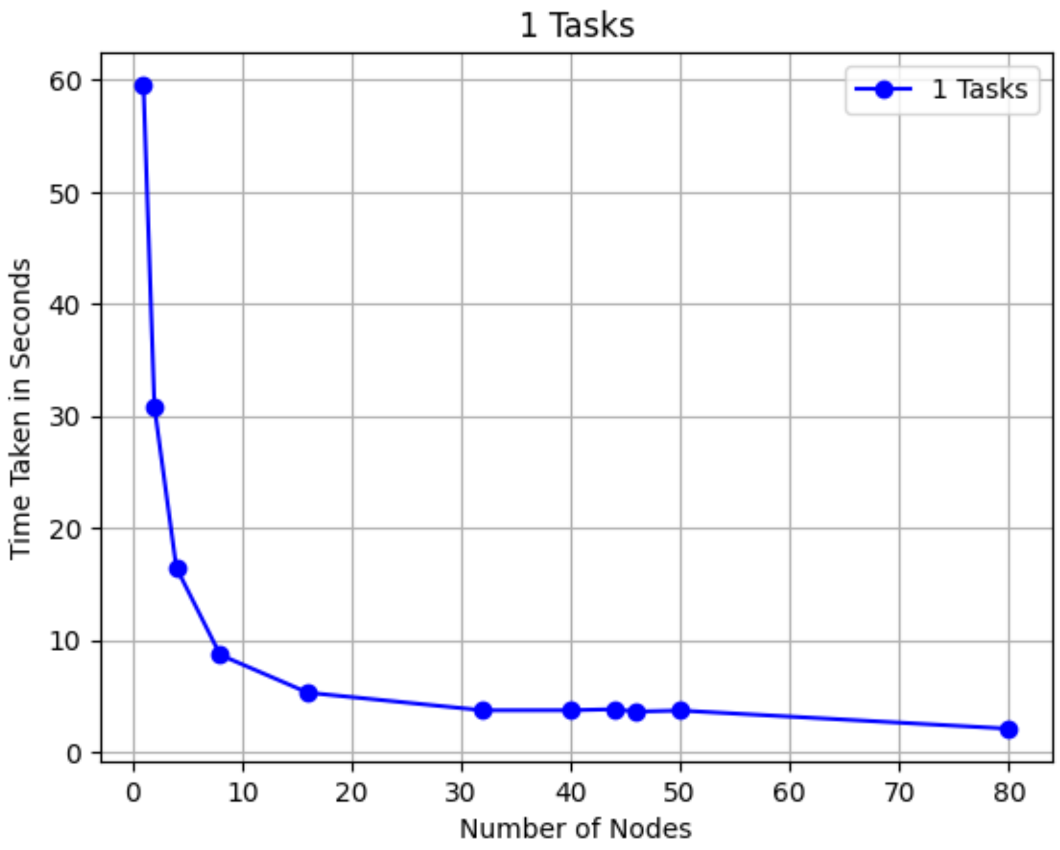
Overall performance with 40000 vertices but for x>20



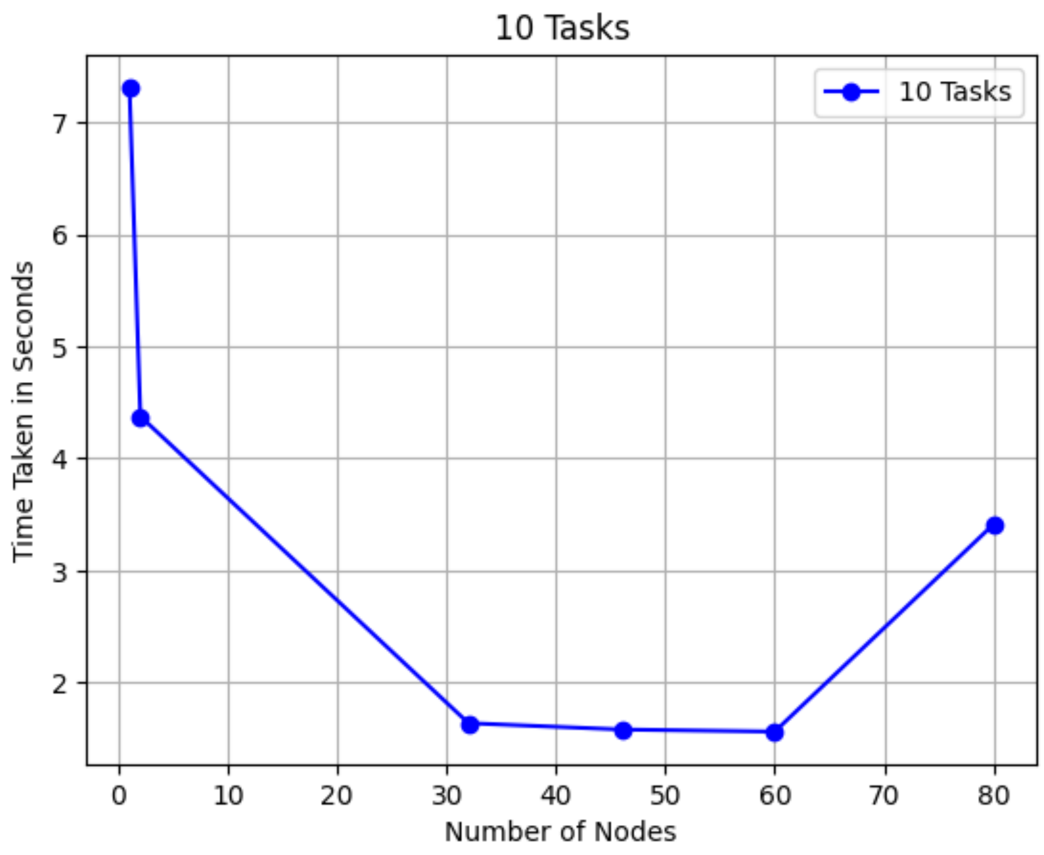
Cropped graph, with a zoomed in perspective for a better understanding.



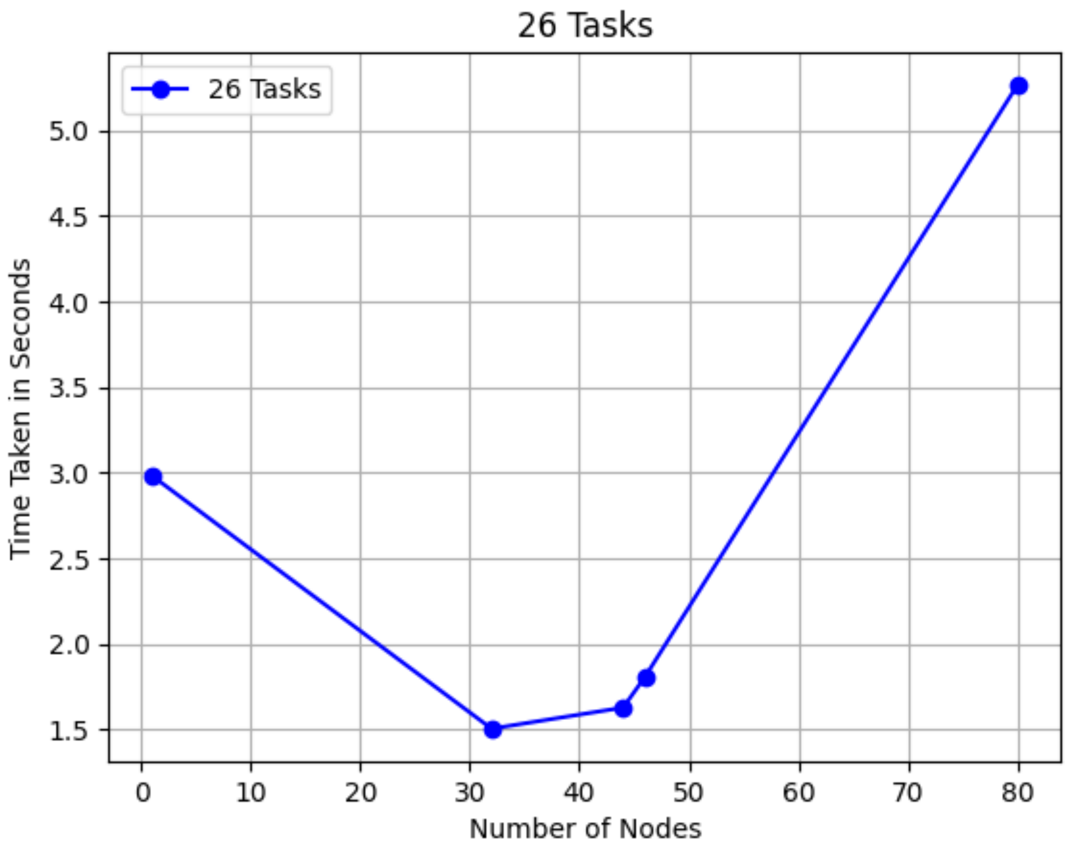
Time taken for 1 Task per node with 40000 vertices



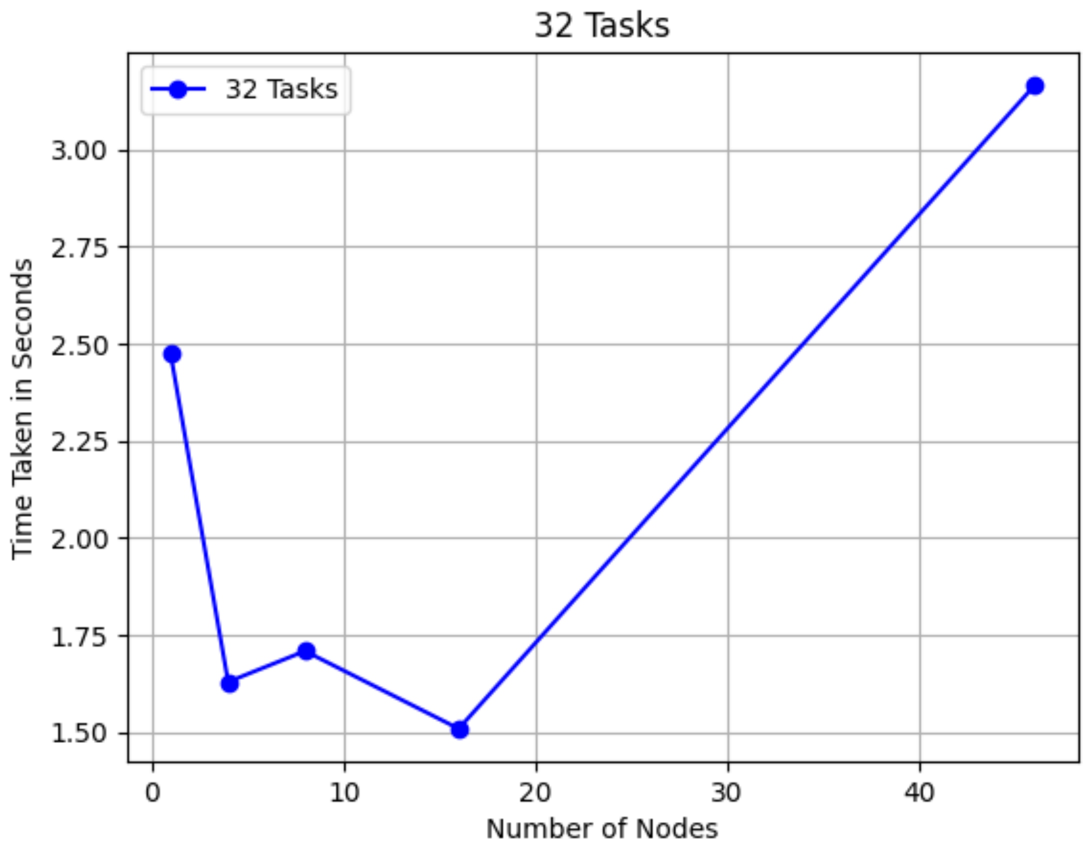
Time taken for 10 Task per node with 40000 vertices



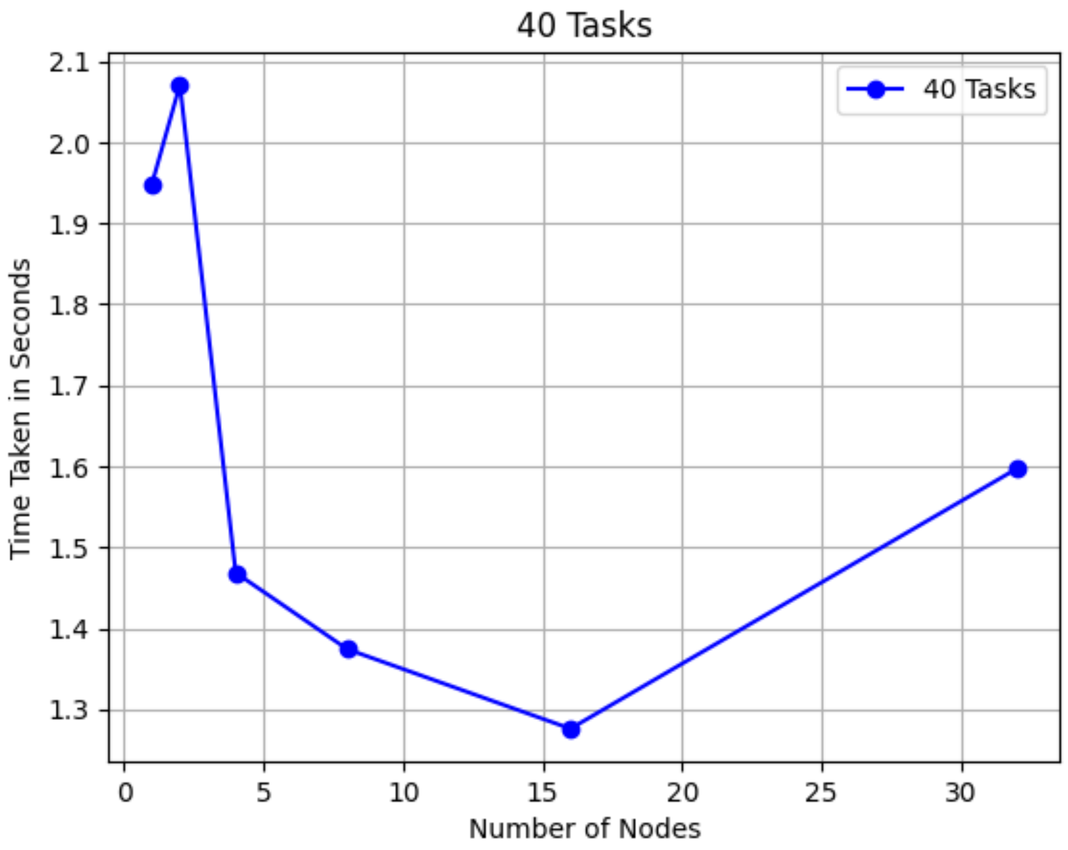
Time taken for 26 Task per node with 40000 vertices



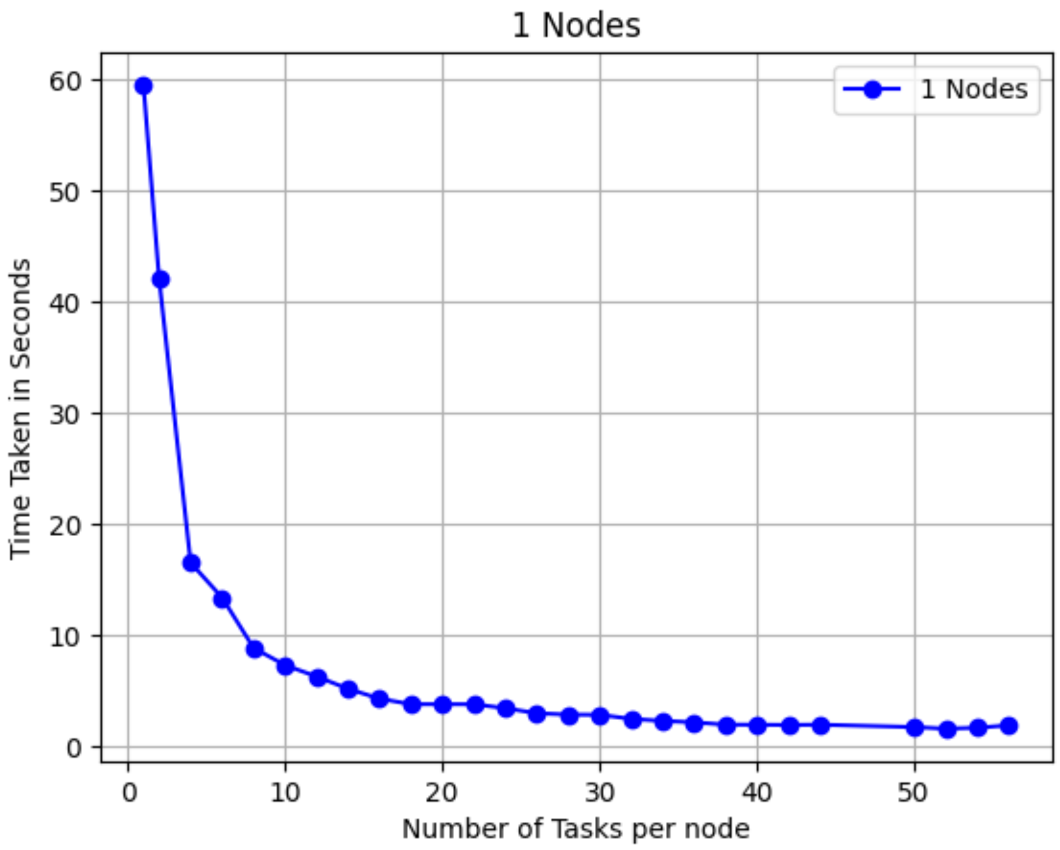
Time taken for 32 Task per node with 40000 vertices



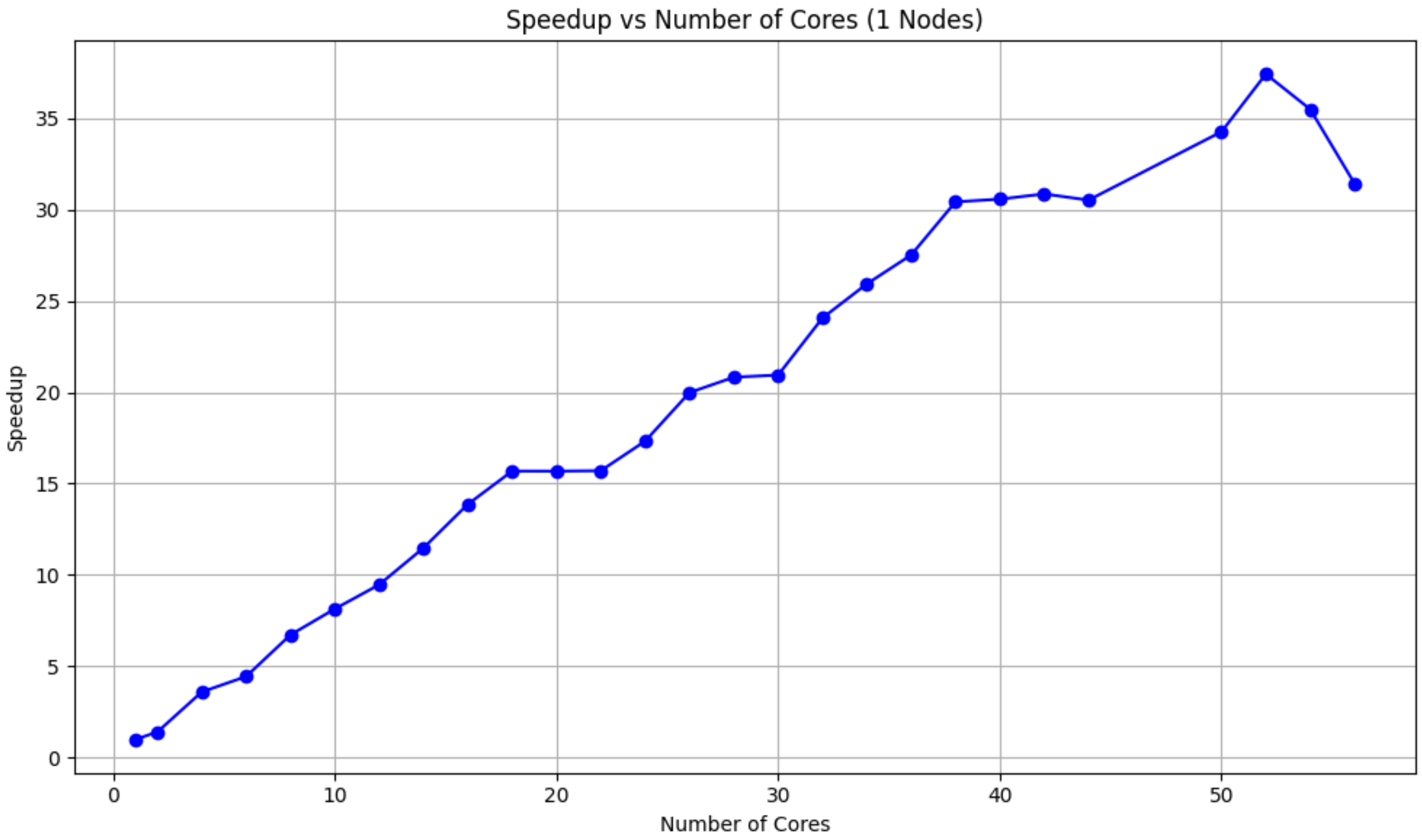
Time taken for 40 Task per node with 40000 vertices



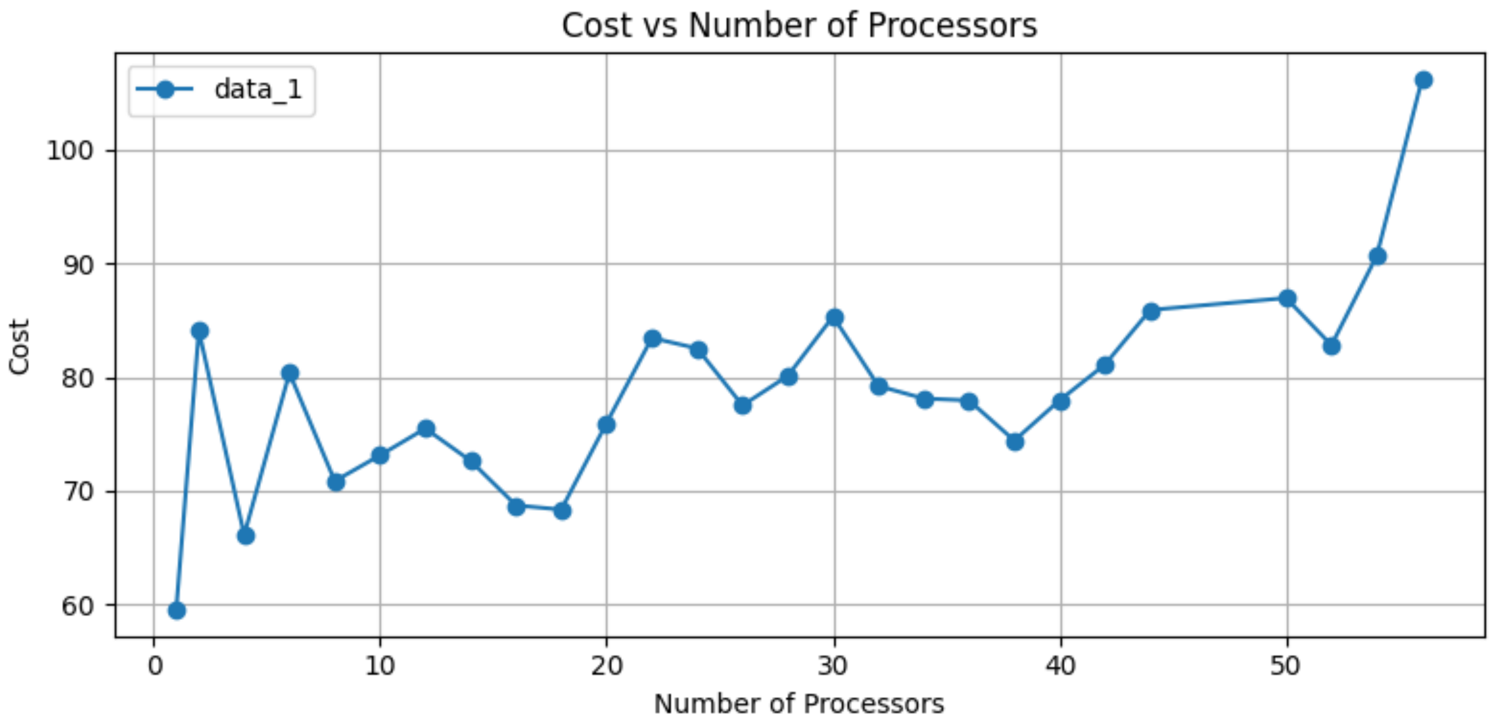
Time taken for 1 Node with 40000 vertices



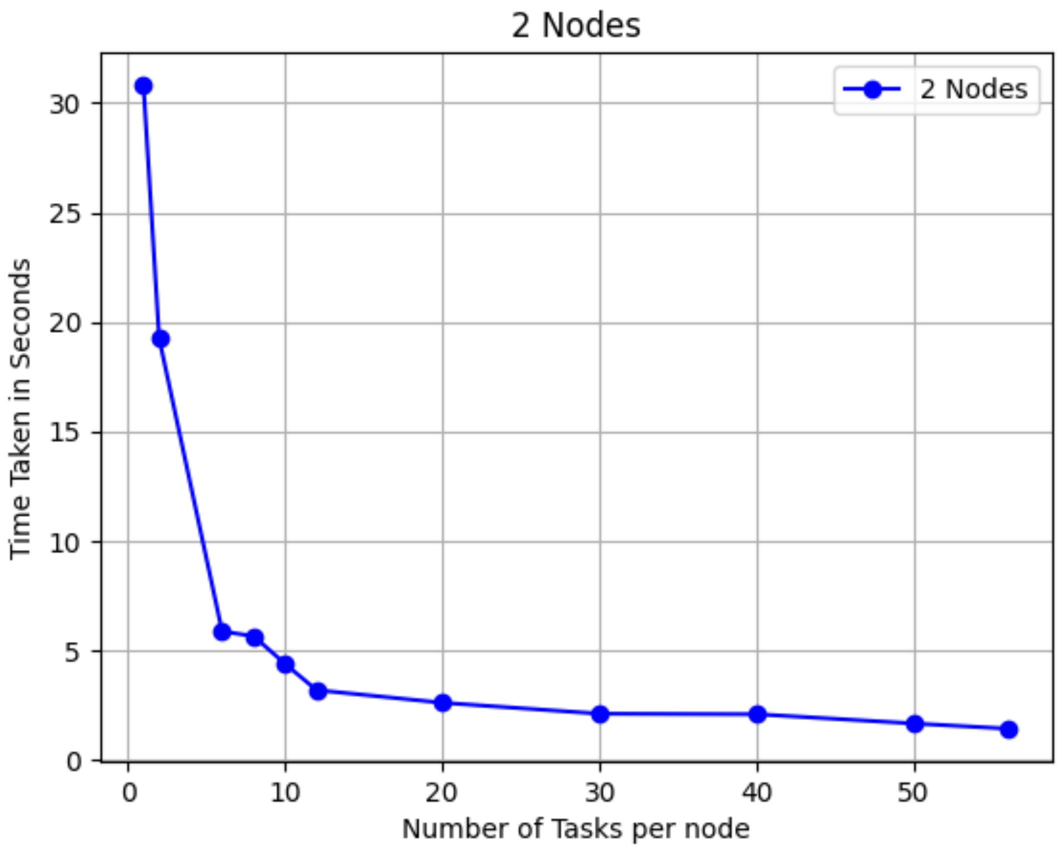
Speedup for 1 Node with 40000 vertices



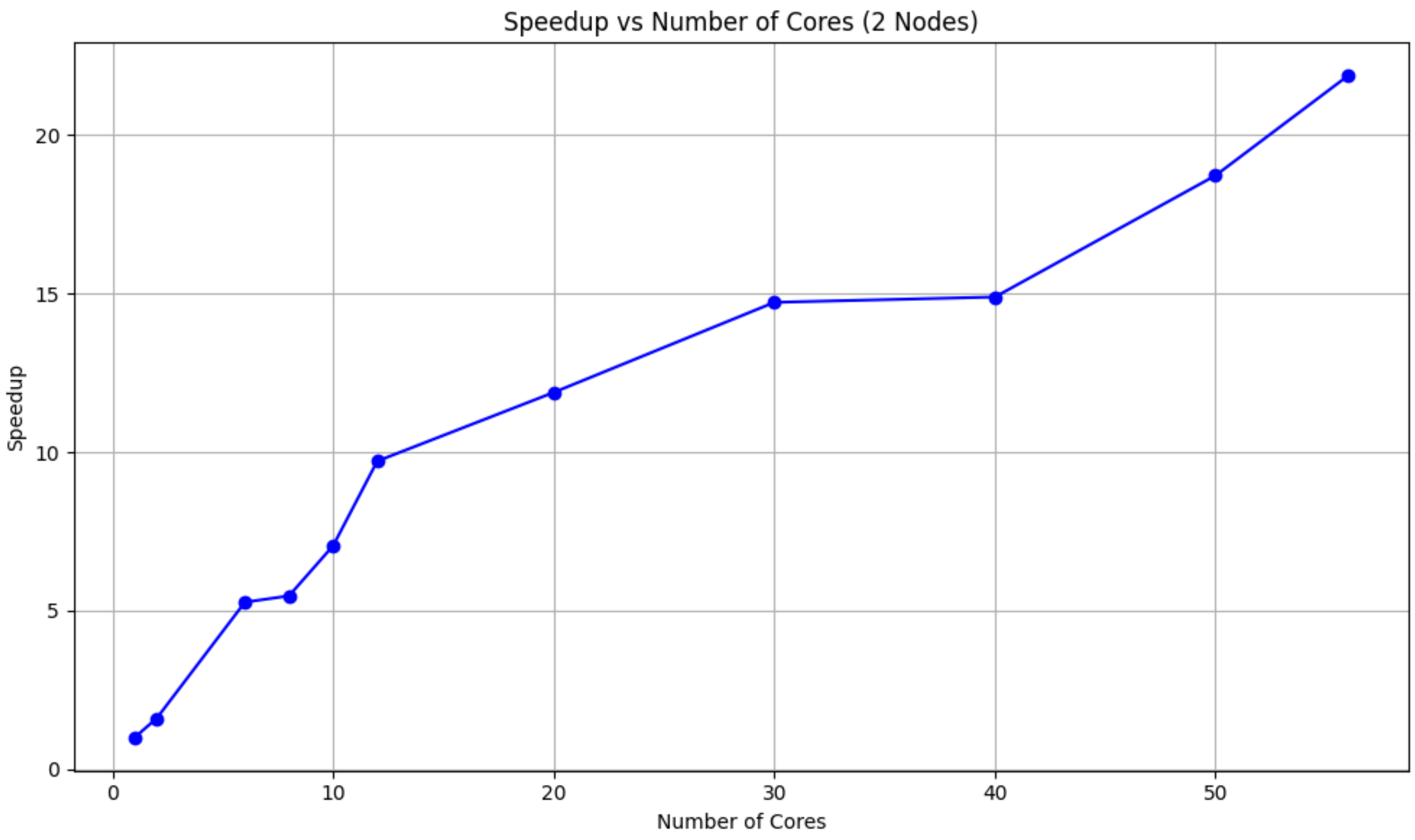
Cost for 1 Node with 40000 vertices



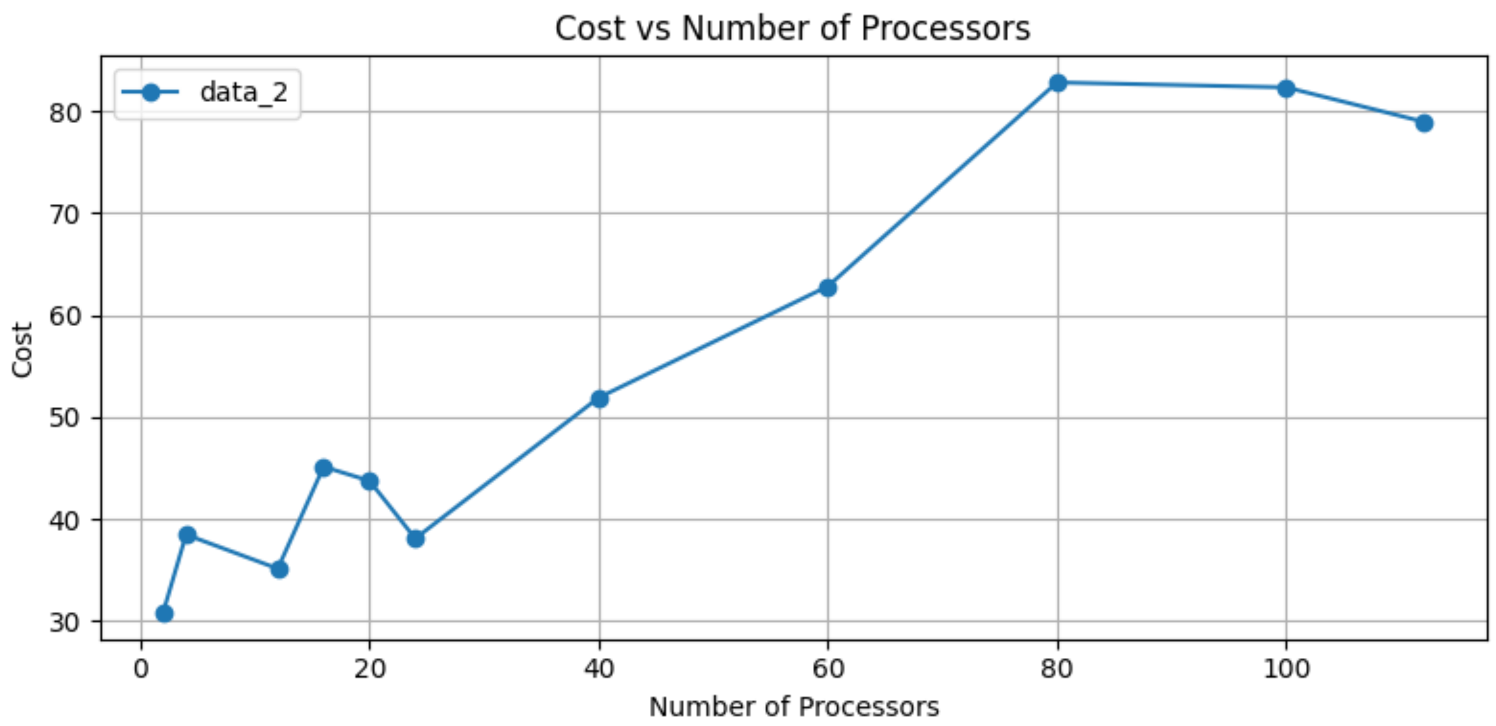
Time taken for 2 Nodes with 40000 vertices



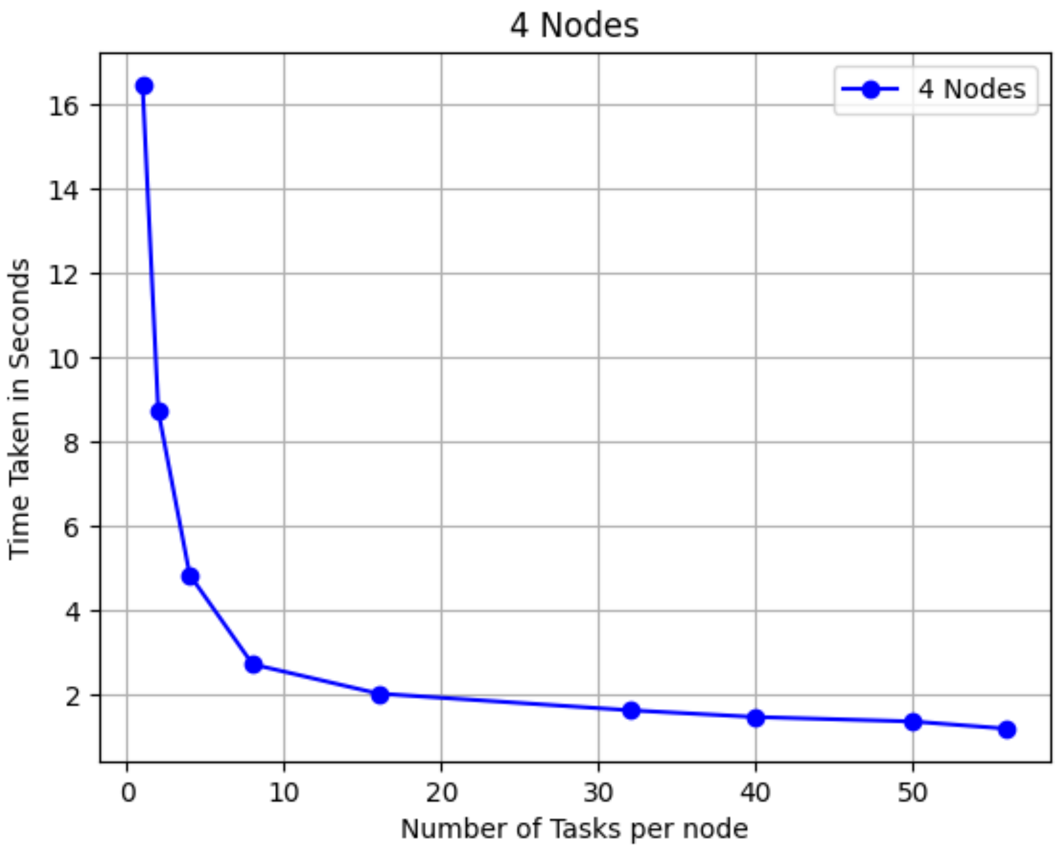
Speedup for 2 Nodes with 40000 vertices



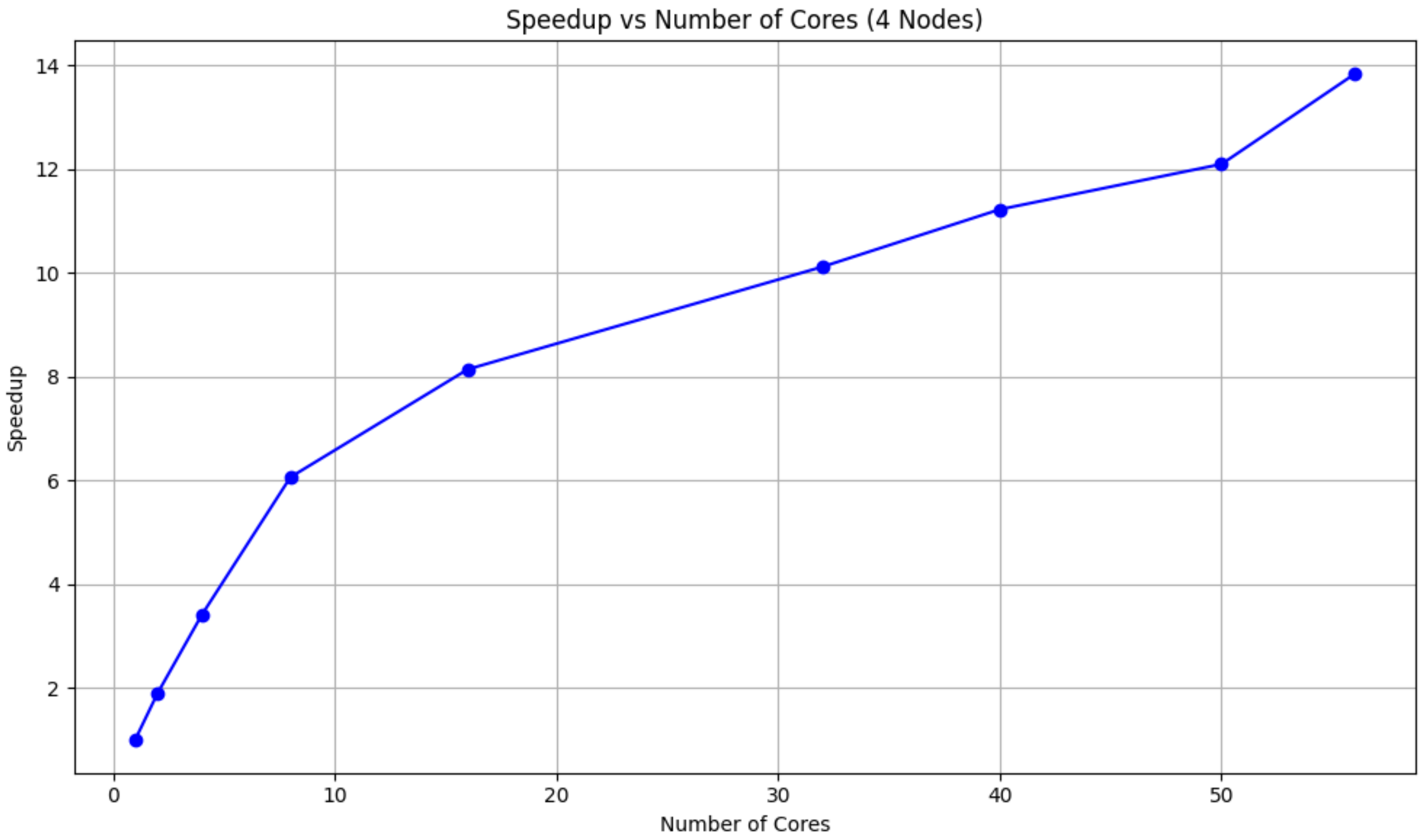
Cost for 2 Nodes with 40000 vertices



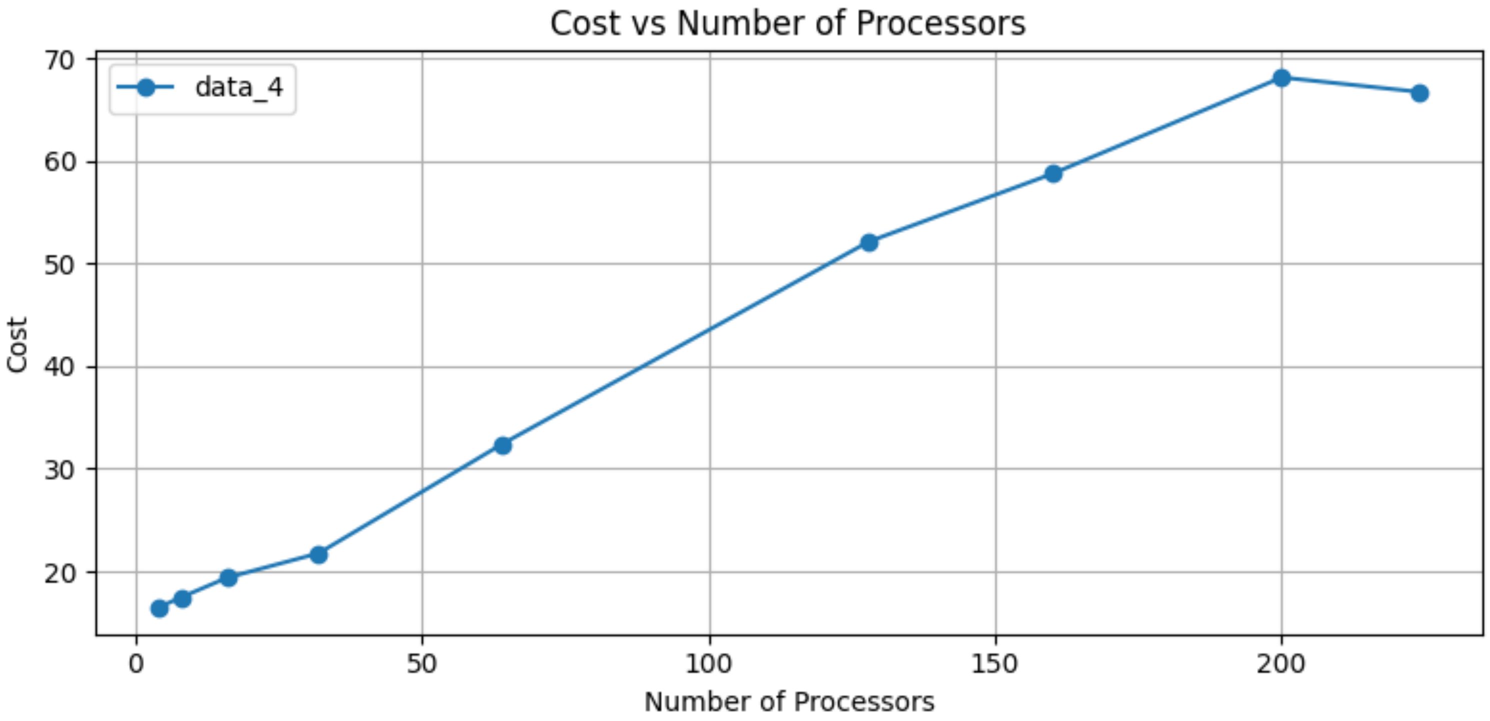
Time taken for 4 Nodes with 40000 vertices



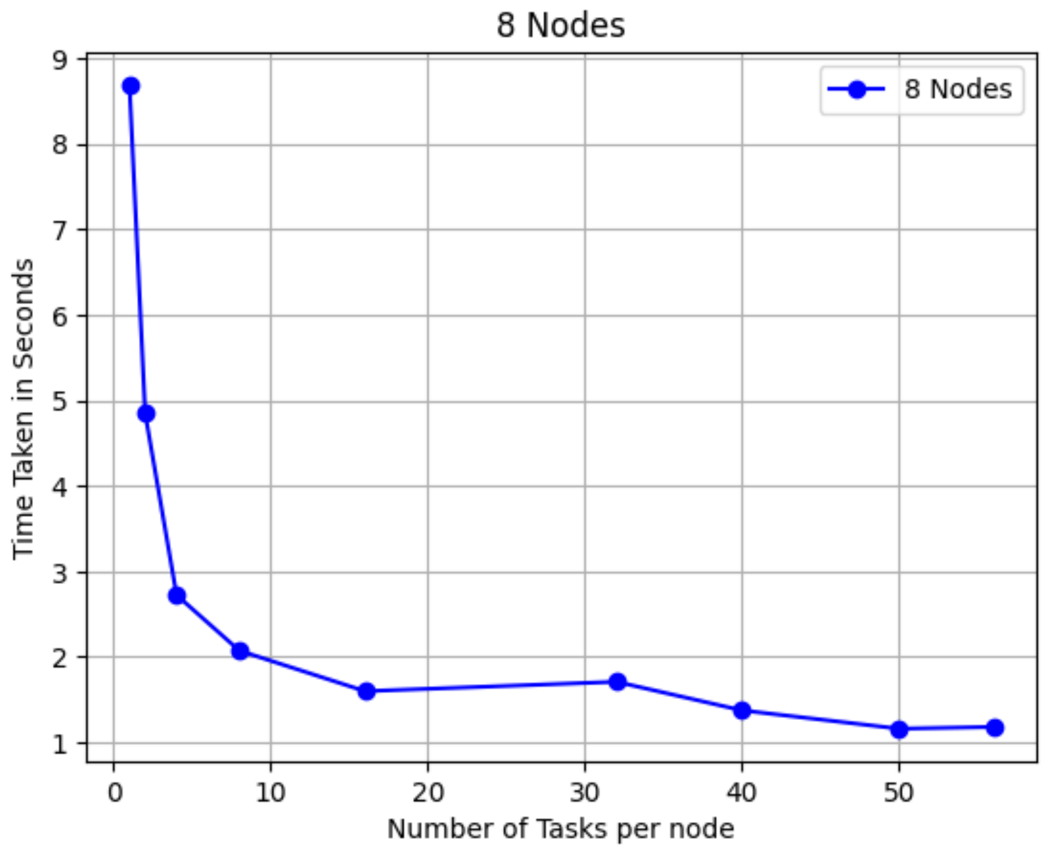
Speedup for 4 Nodes with 40000 vertices



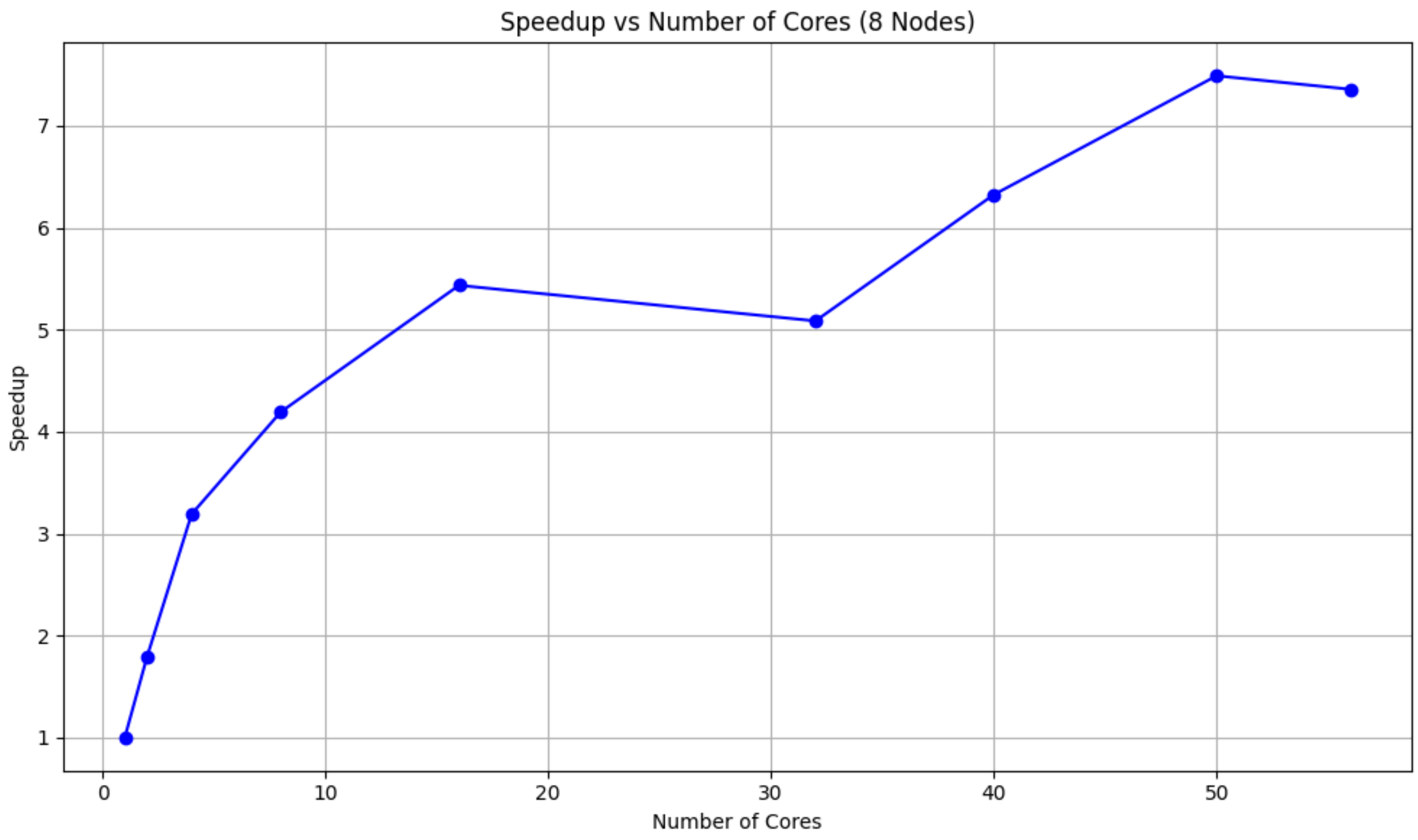
Cost for 4 Nodes with 40000 vertices



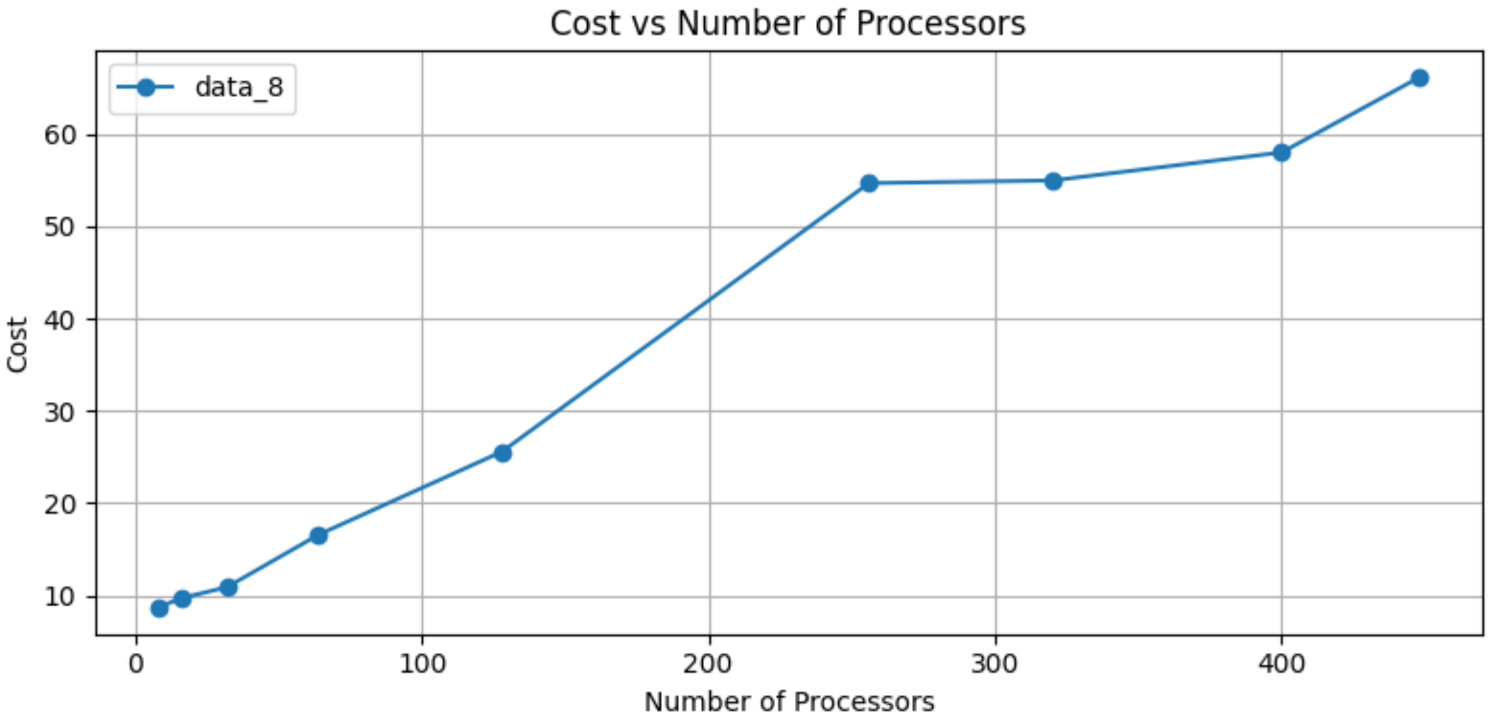
Time taken for 8 Nodes with 40000 vertices



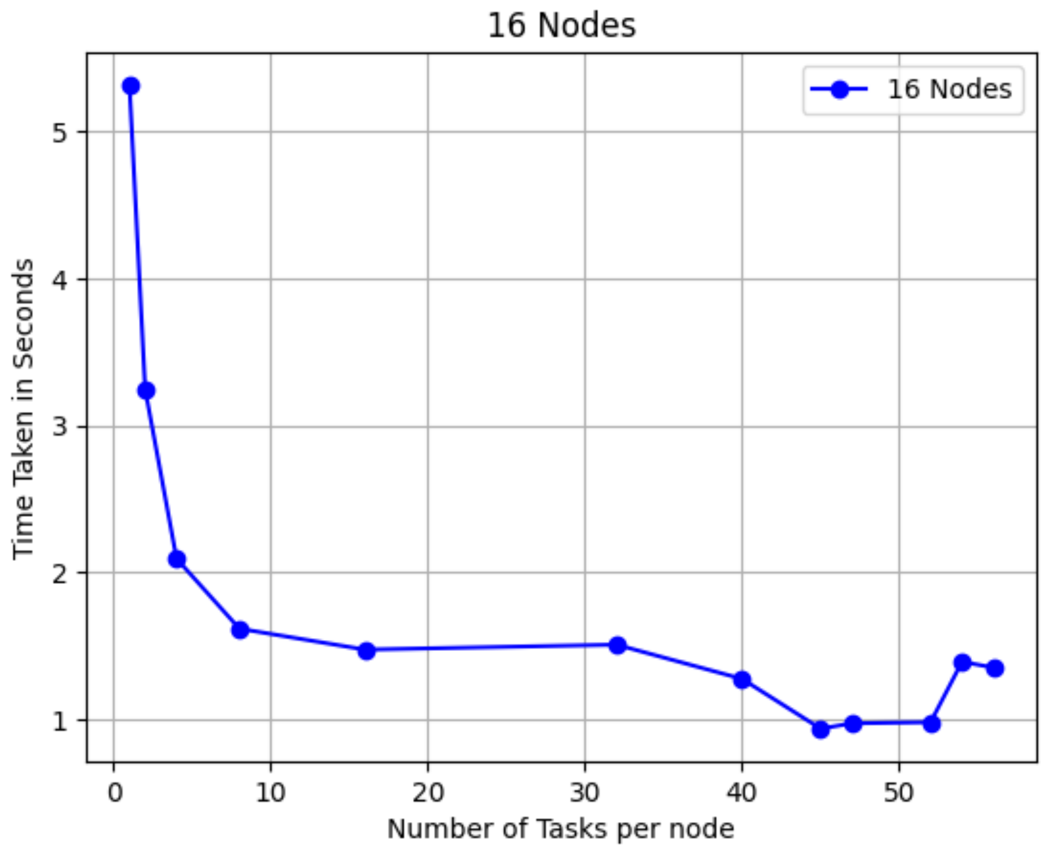
Speedup for 8 Nodes with 40000 vertices



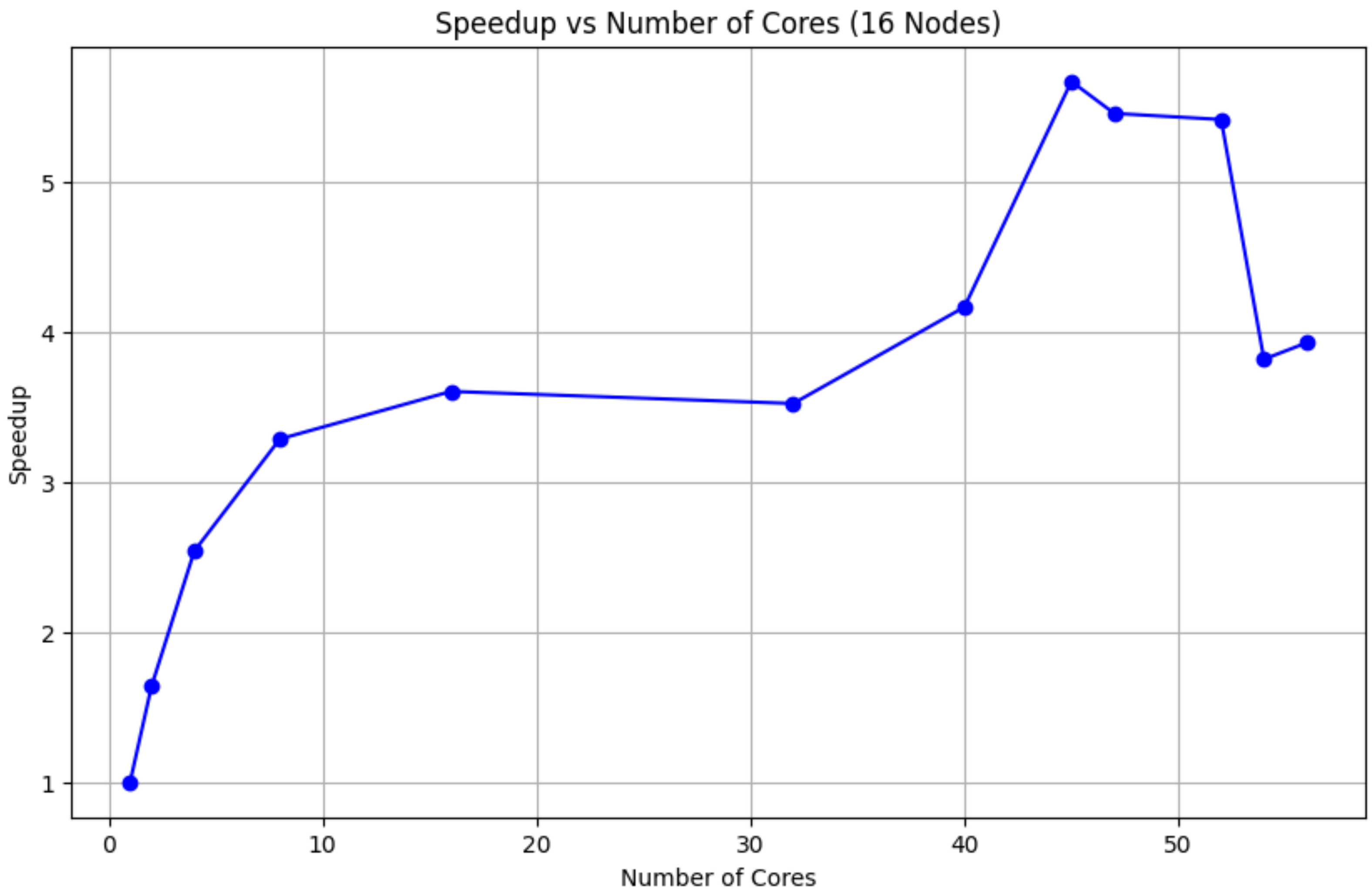
Cost for 8 Nodes with 40000 vertices



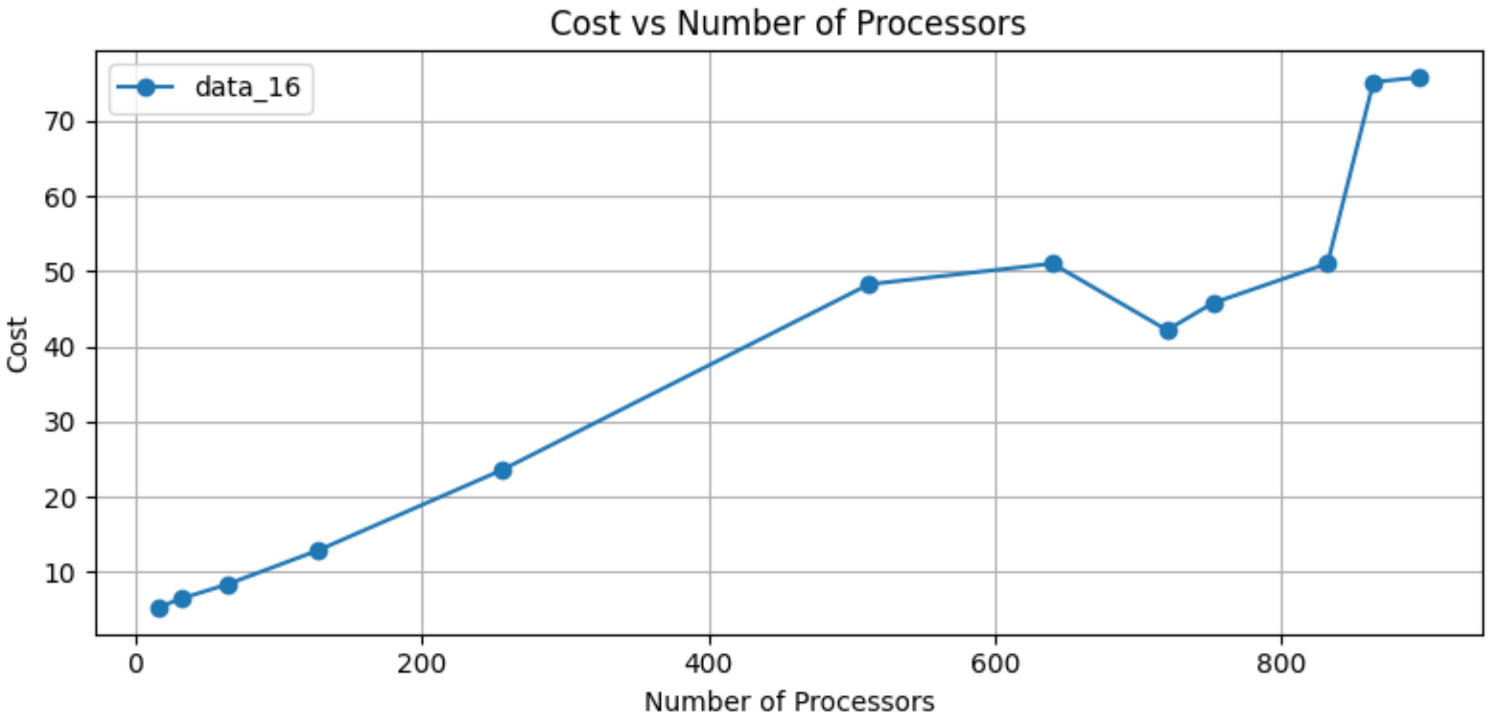
Time taken for 16 Nodes with 40000 vertices



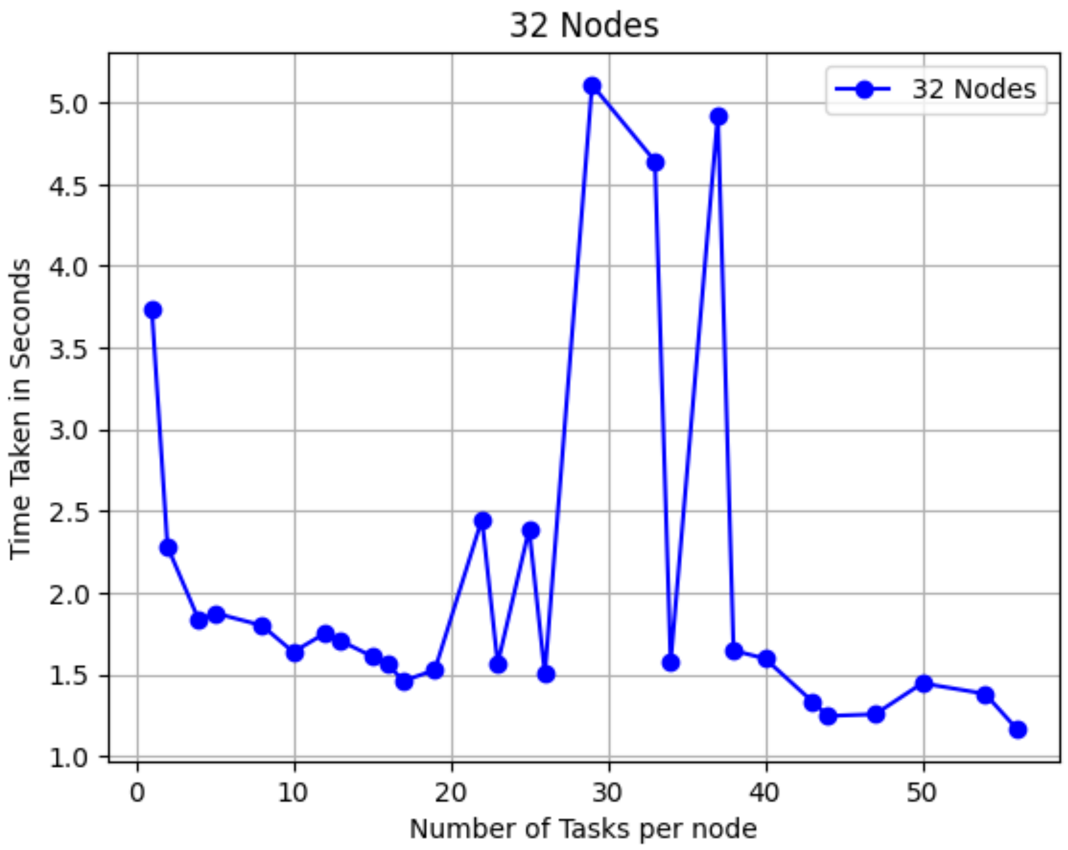
Speedup for 16 Nodes with 40000 vertices



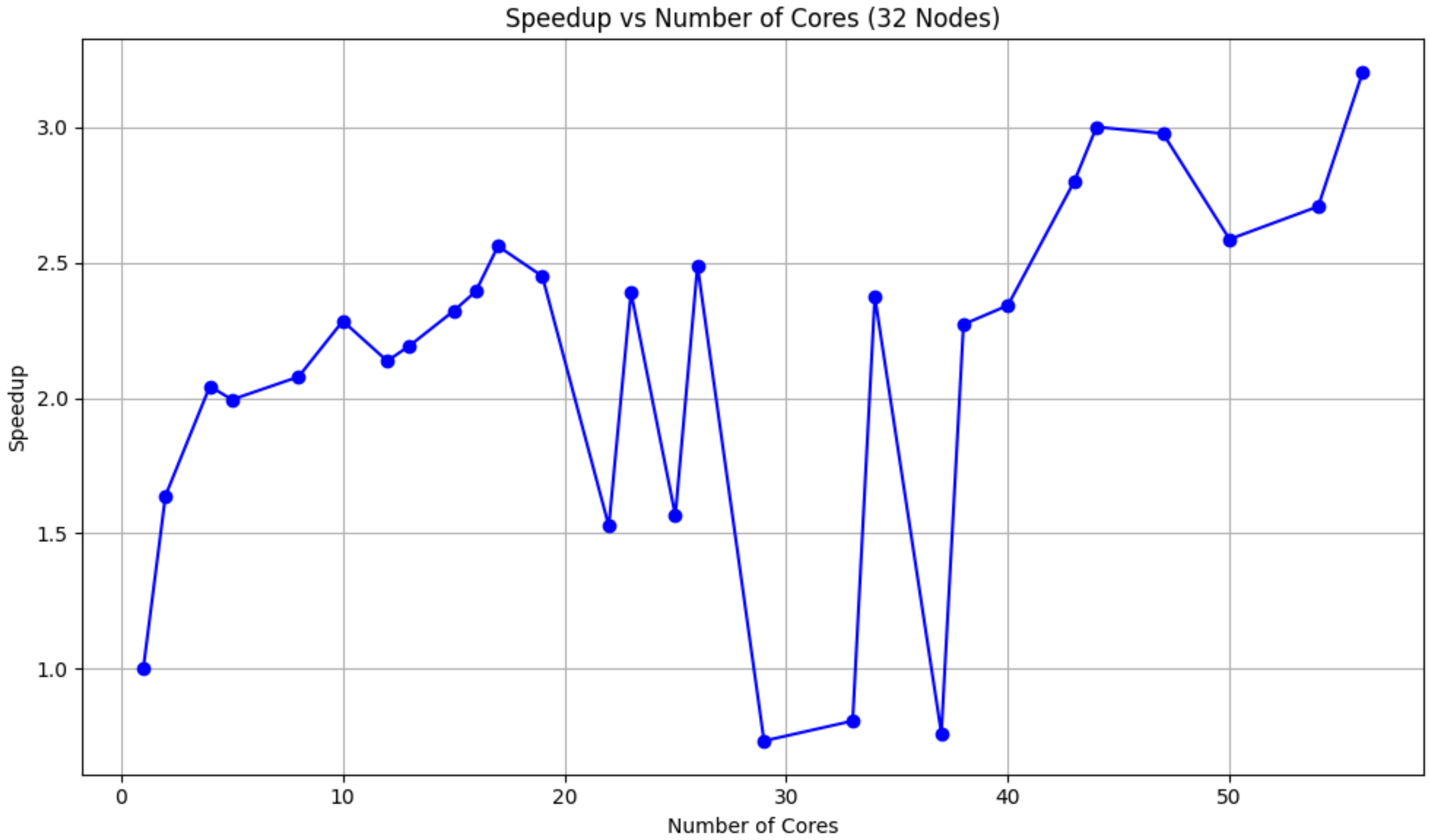
Cost for 16 Nodes with 40000 vertices



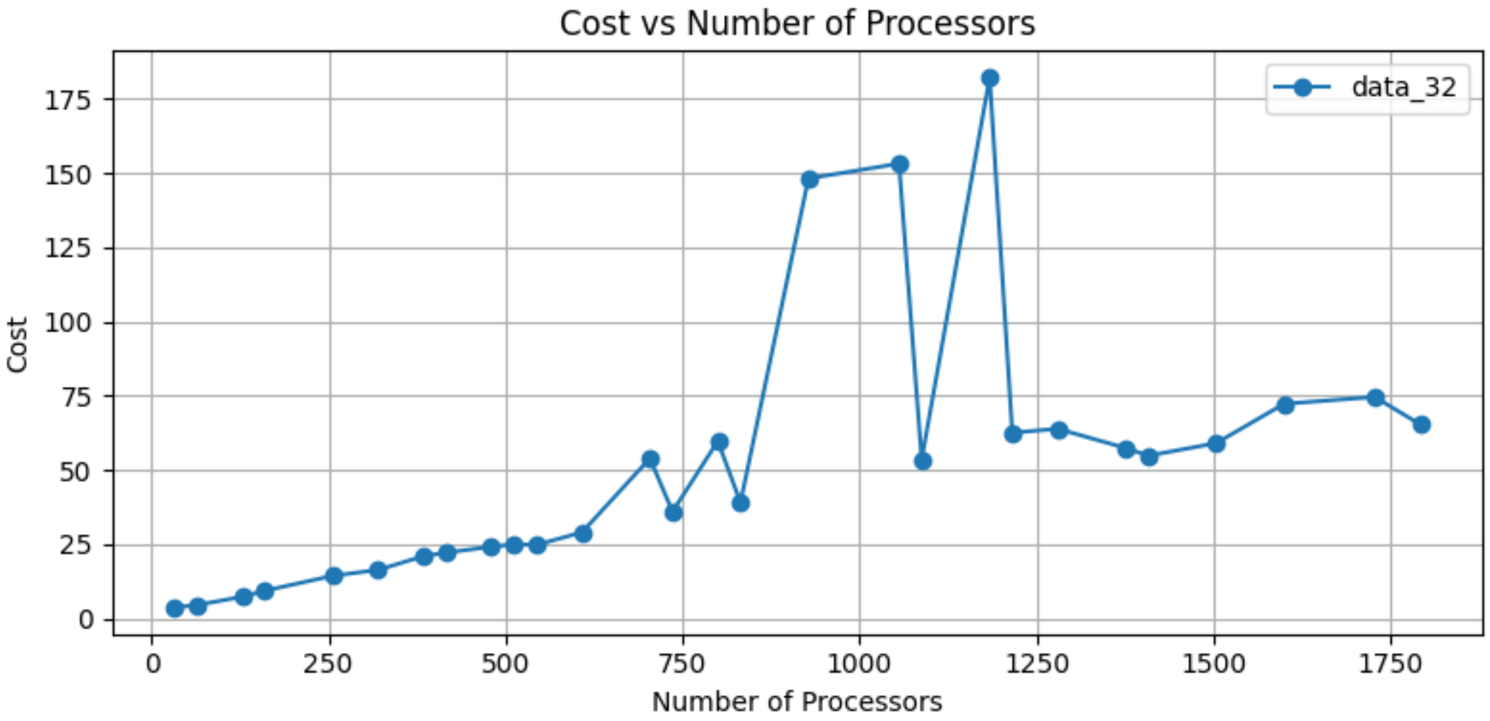
Time taken for 32 Nodes with 40000 vertices



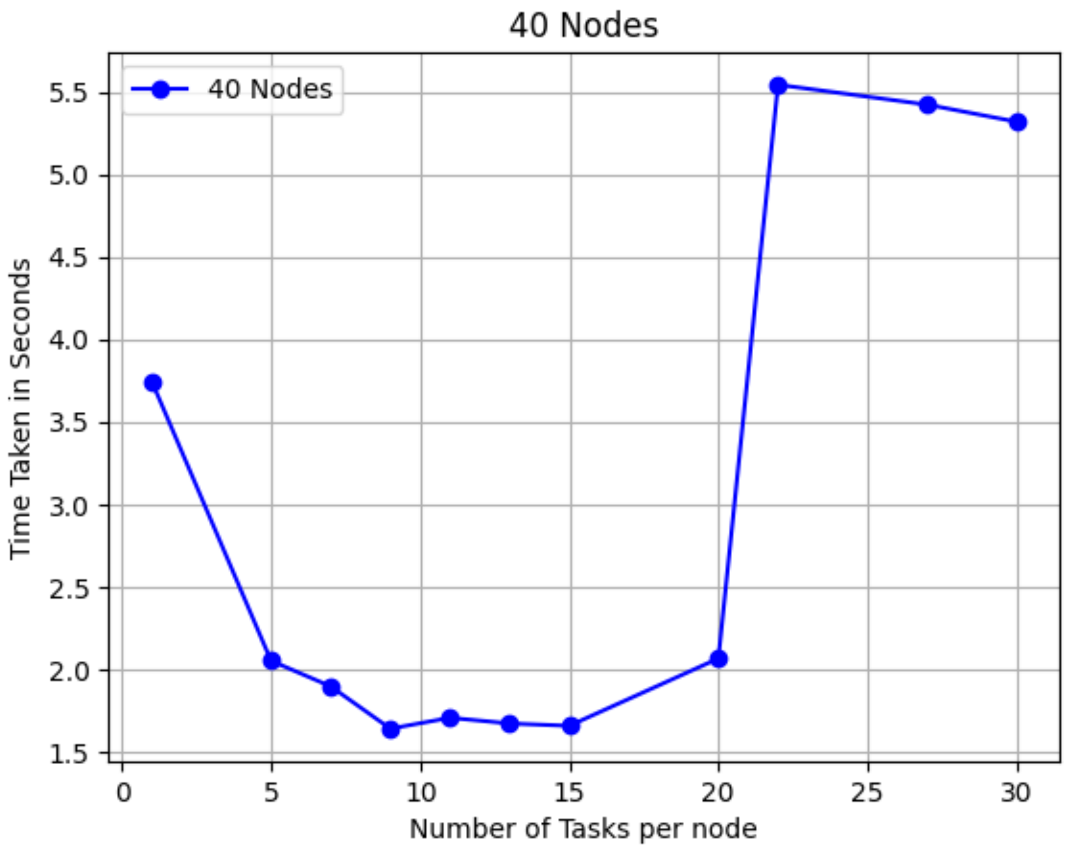
Speedup for 32 Nodes with 40000 vertices



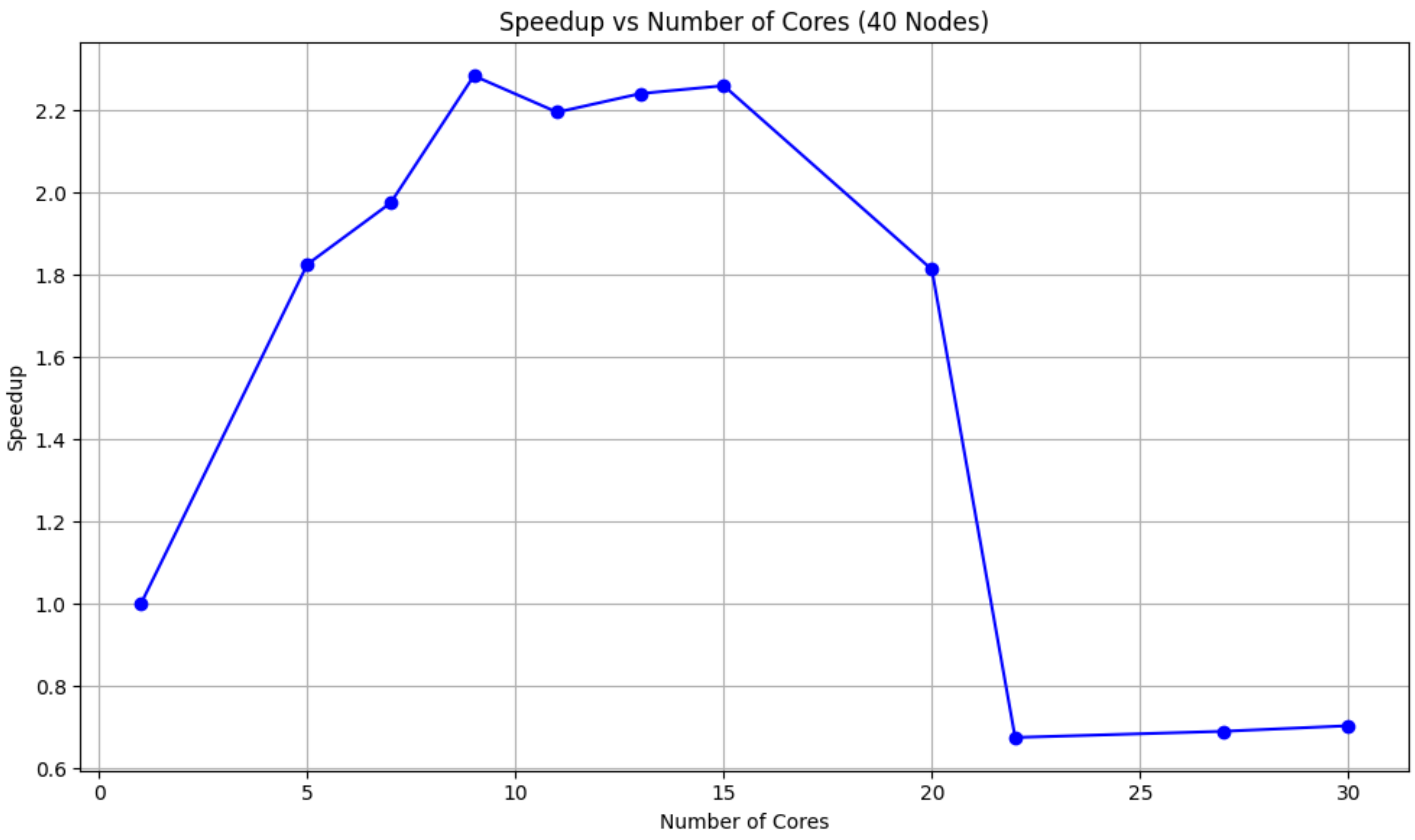
Cost for 32 Nodes with 40000 vertices



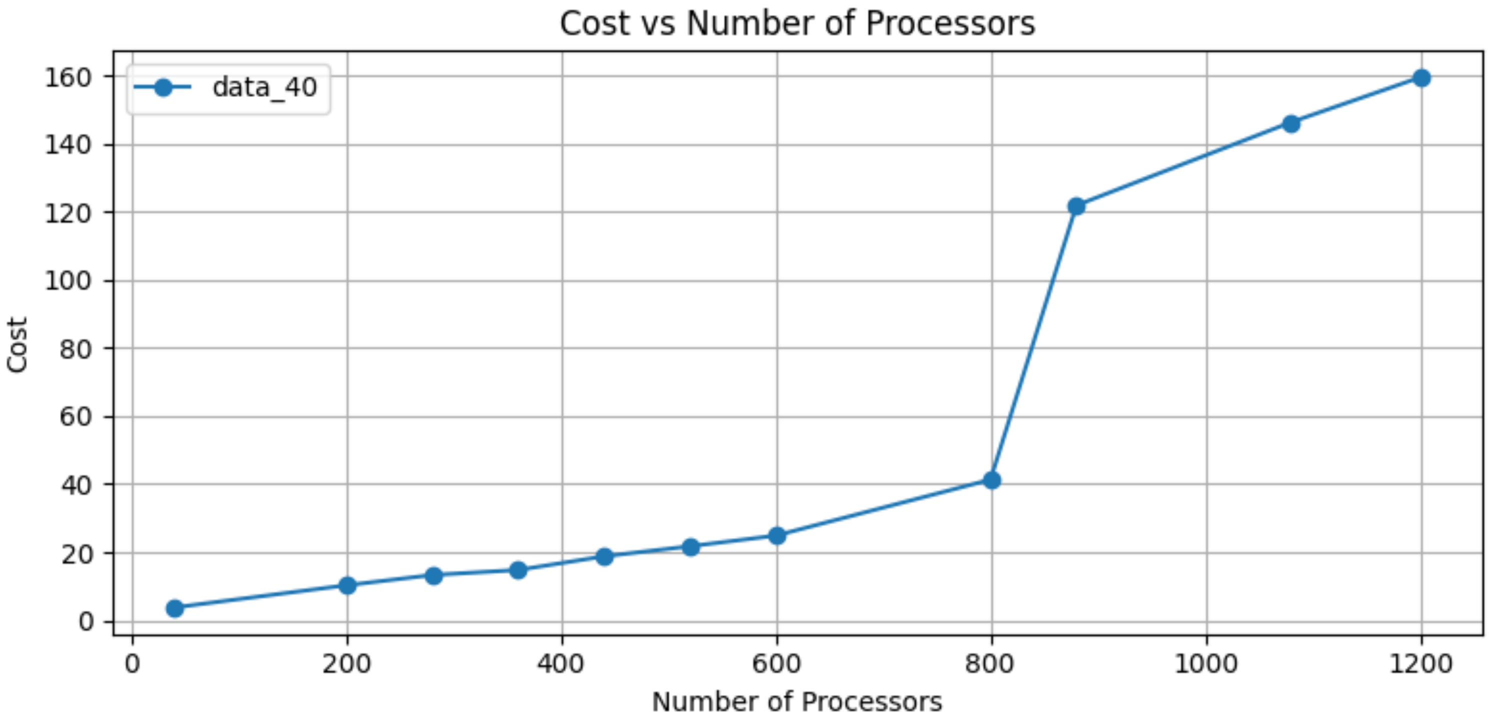
Time taken for 40 Nodes with 40000 vertices



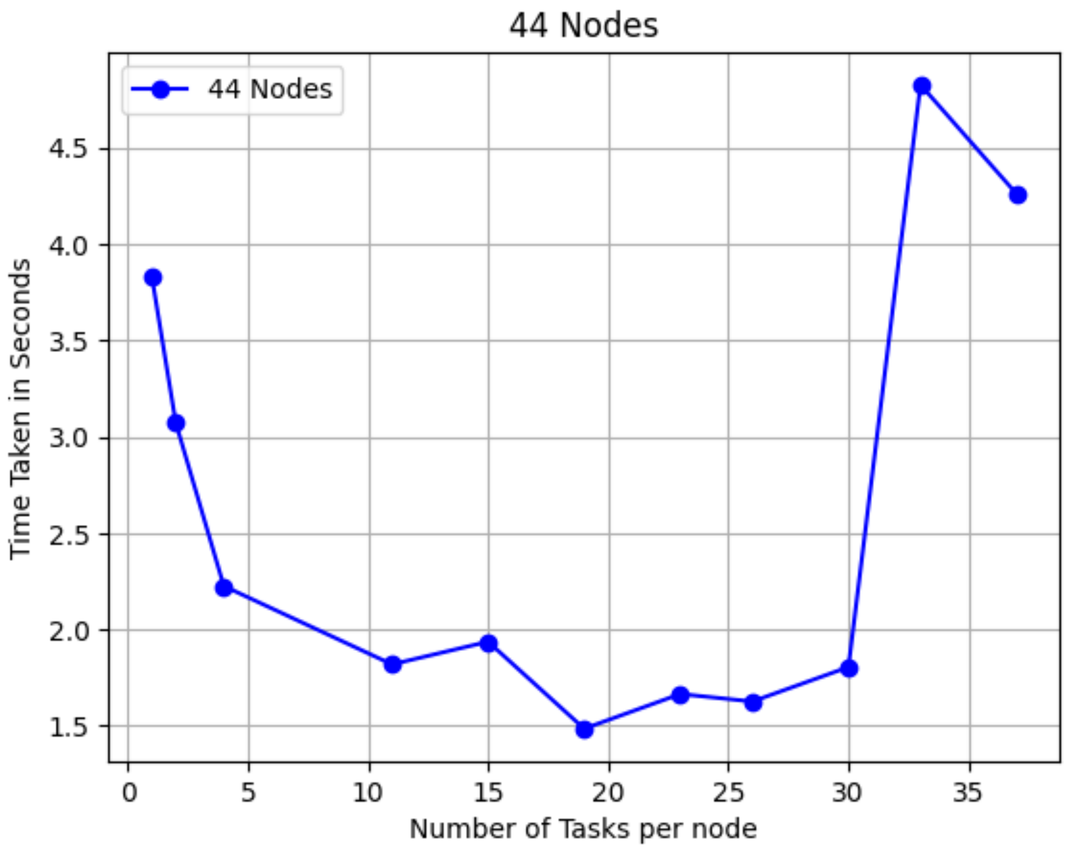
Speedup for 40 Nodes with 40000 vertices



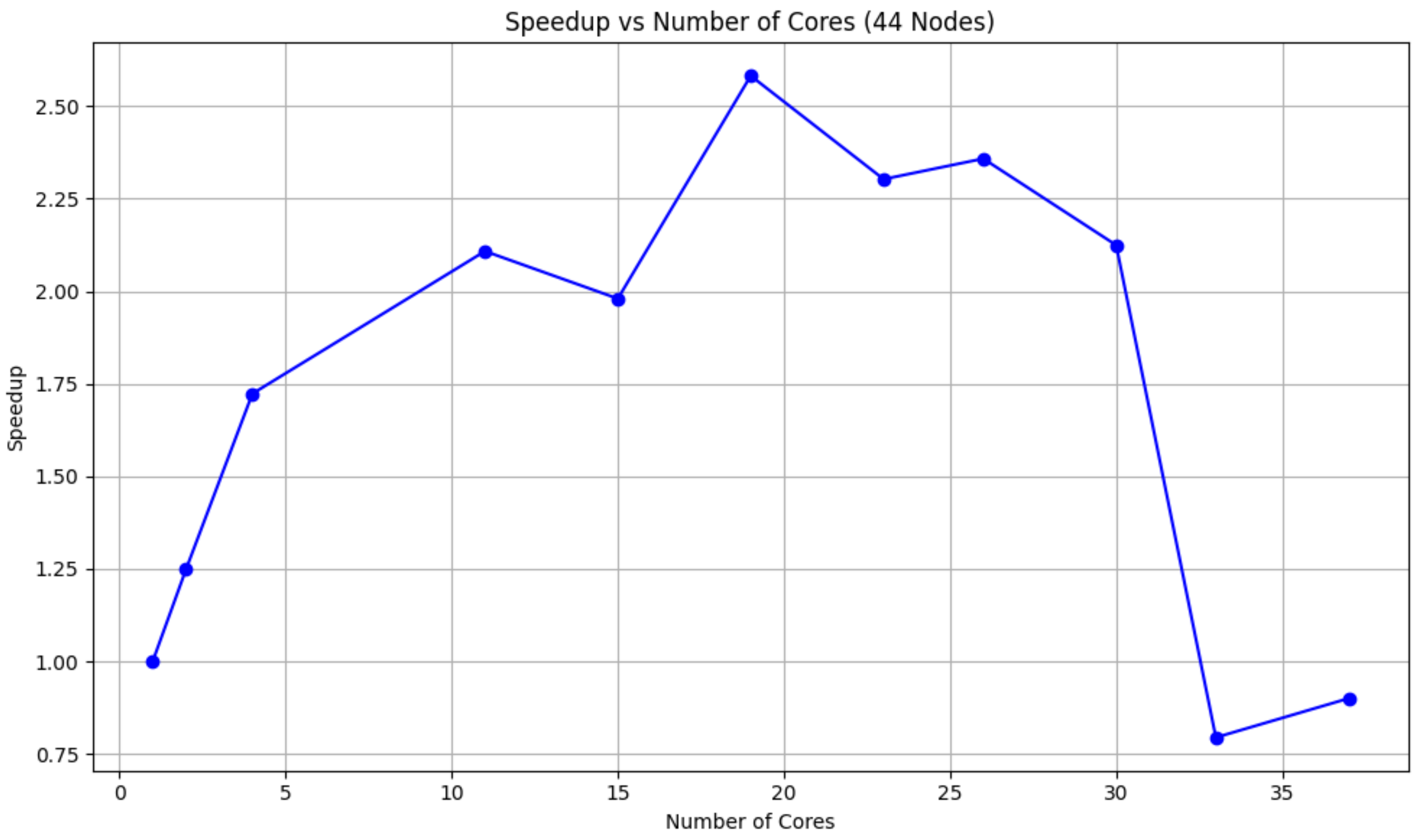
Cost for 40 Nodes with 40000 vertices



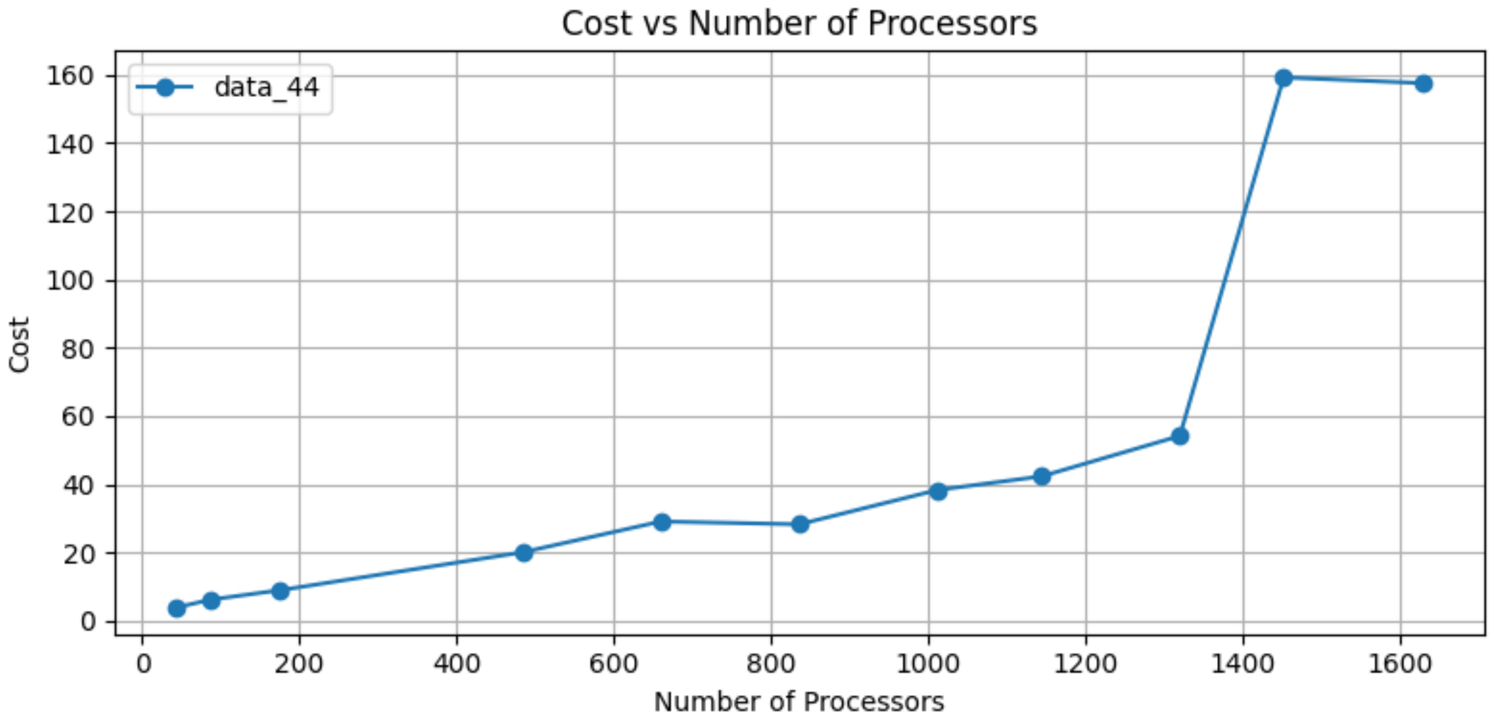
Time taken for 44 Nodes with 40000 vertices



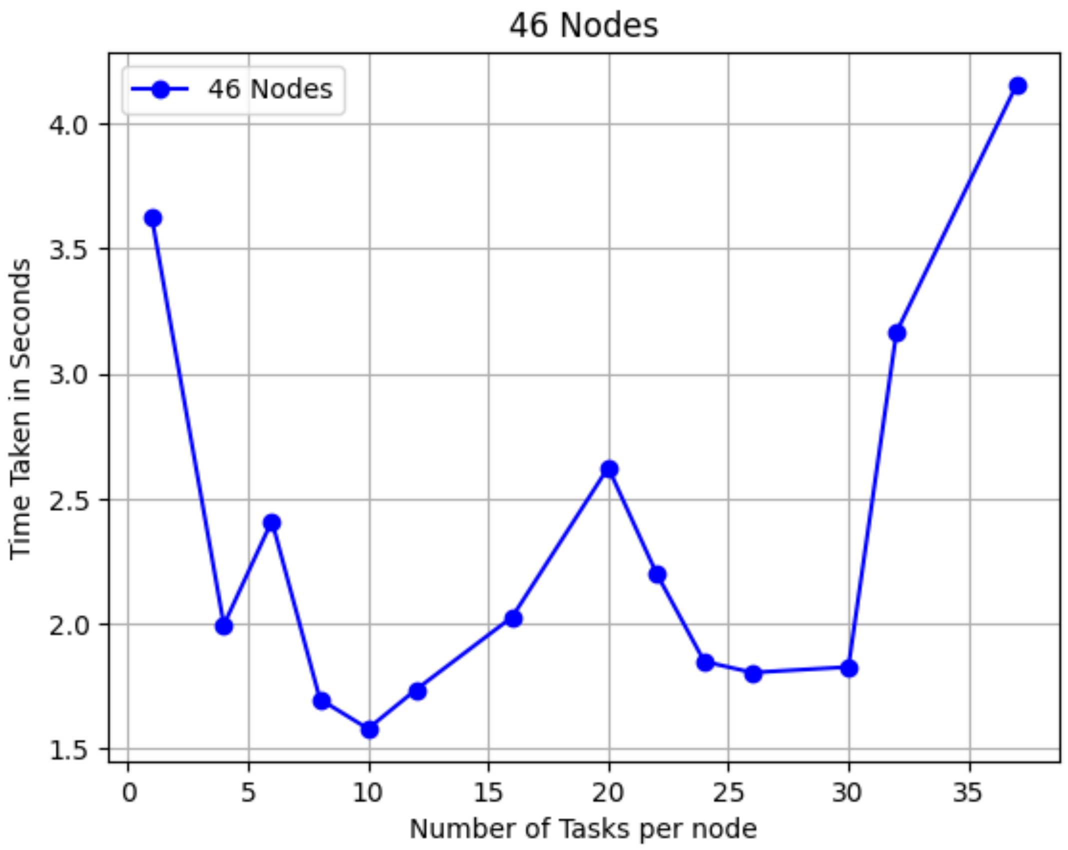
Speedup for 44 Nodes with 40000 vertices



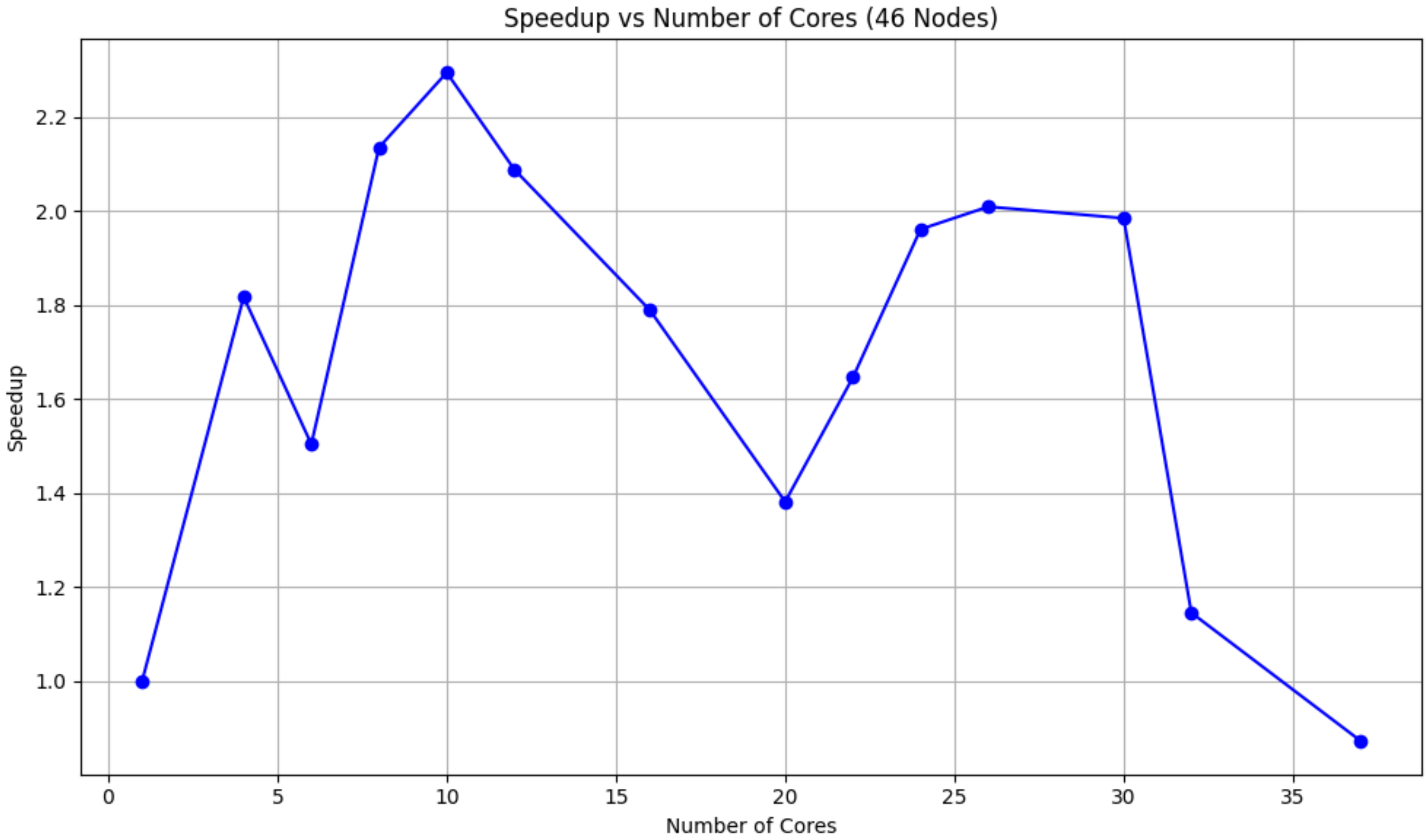
Cost for 44 Nodes with 40000 vertices



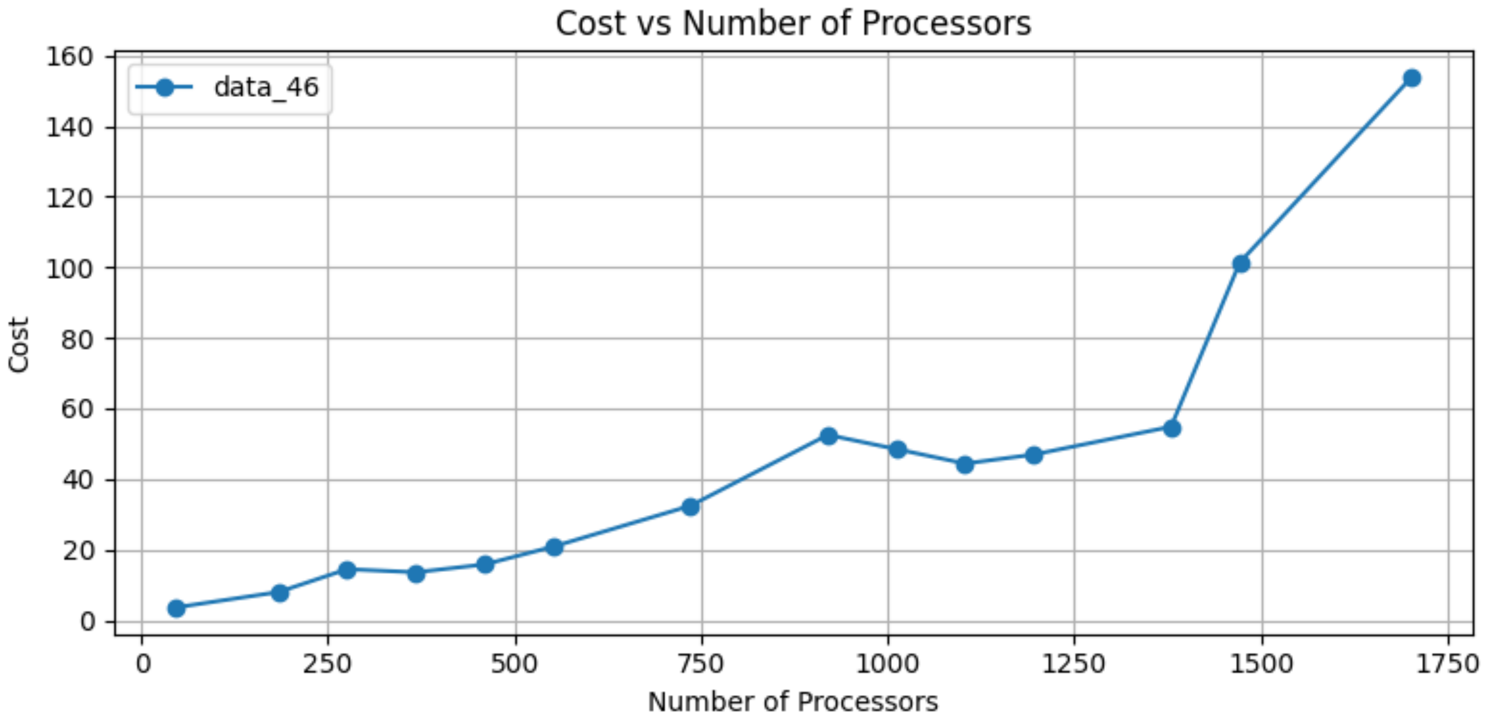
Time taken for 46 Nodes with 40000 vertices



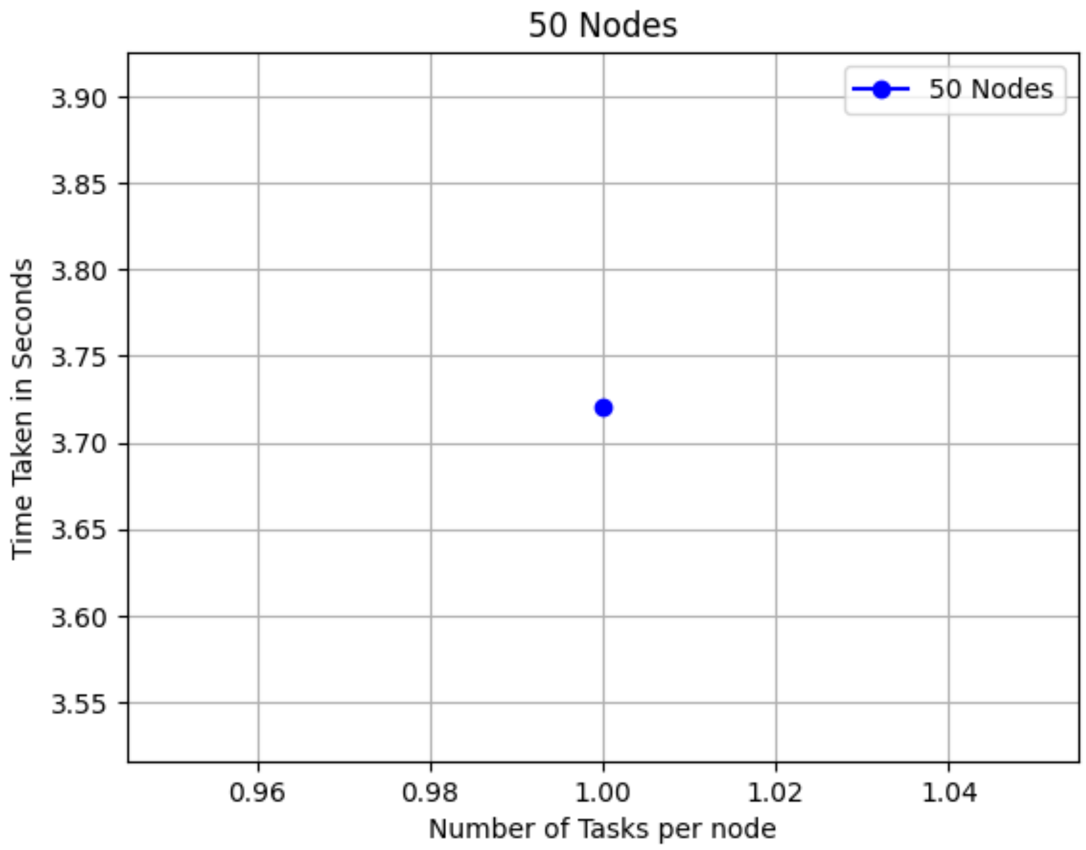
Speedup for 46 Nodes with 40000 vertices



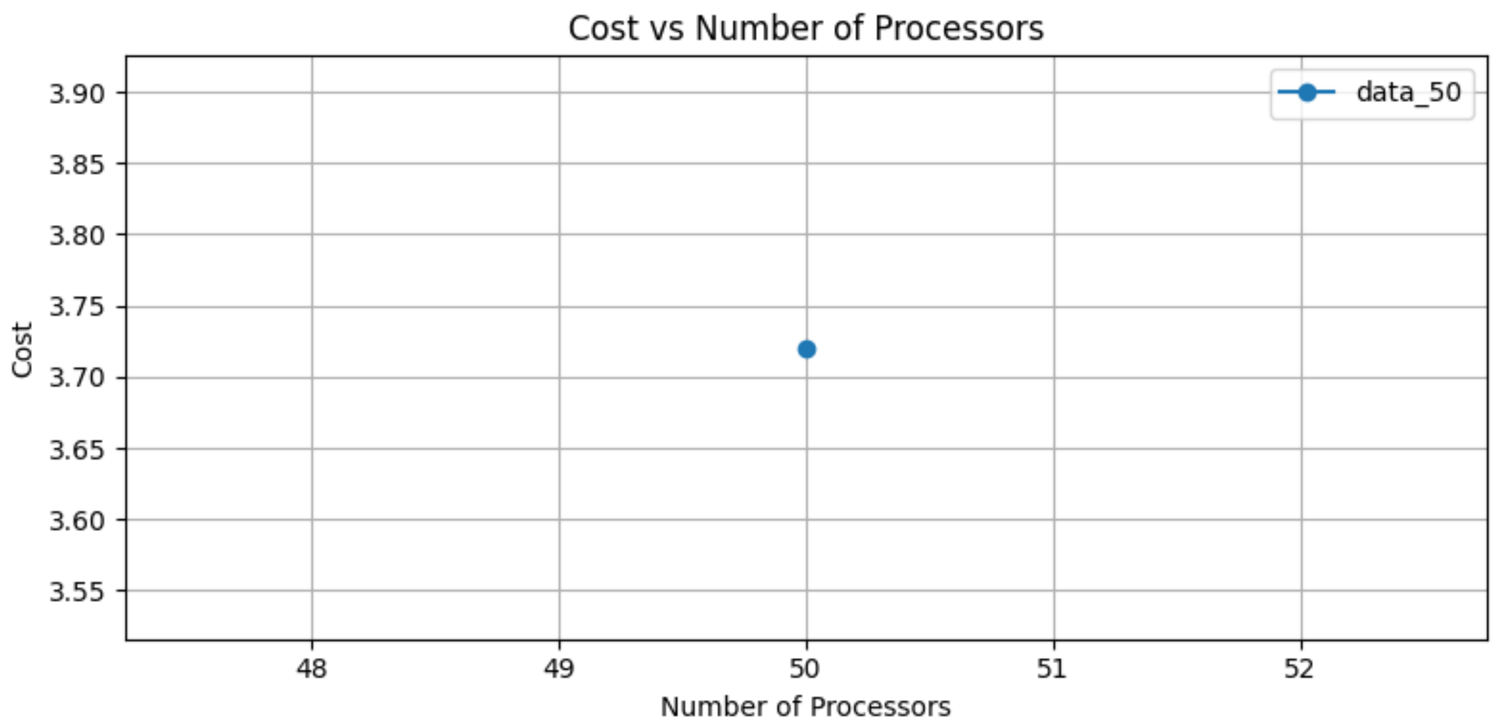
Cost for 46 Nodes with 40000 vertices



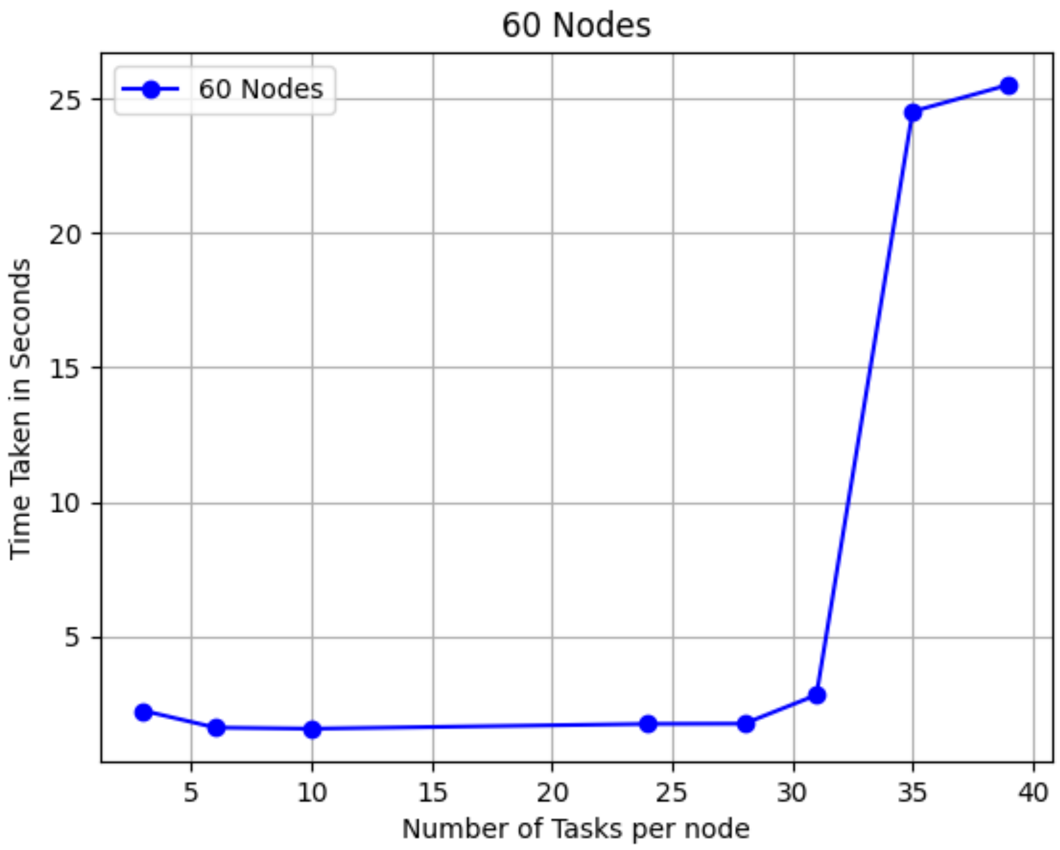
Time taken for 50 Nodes with 40000 vertices



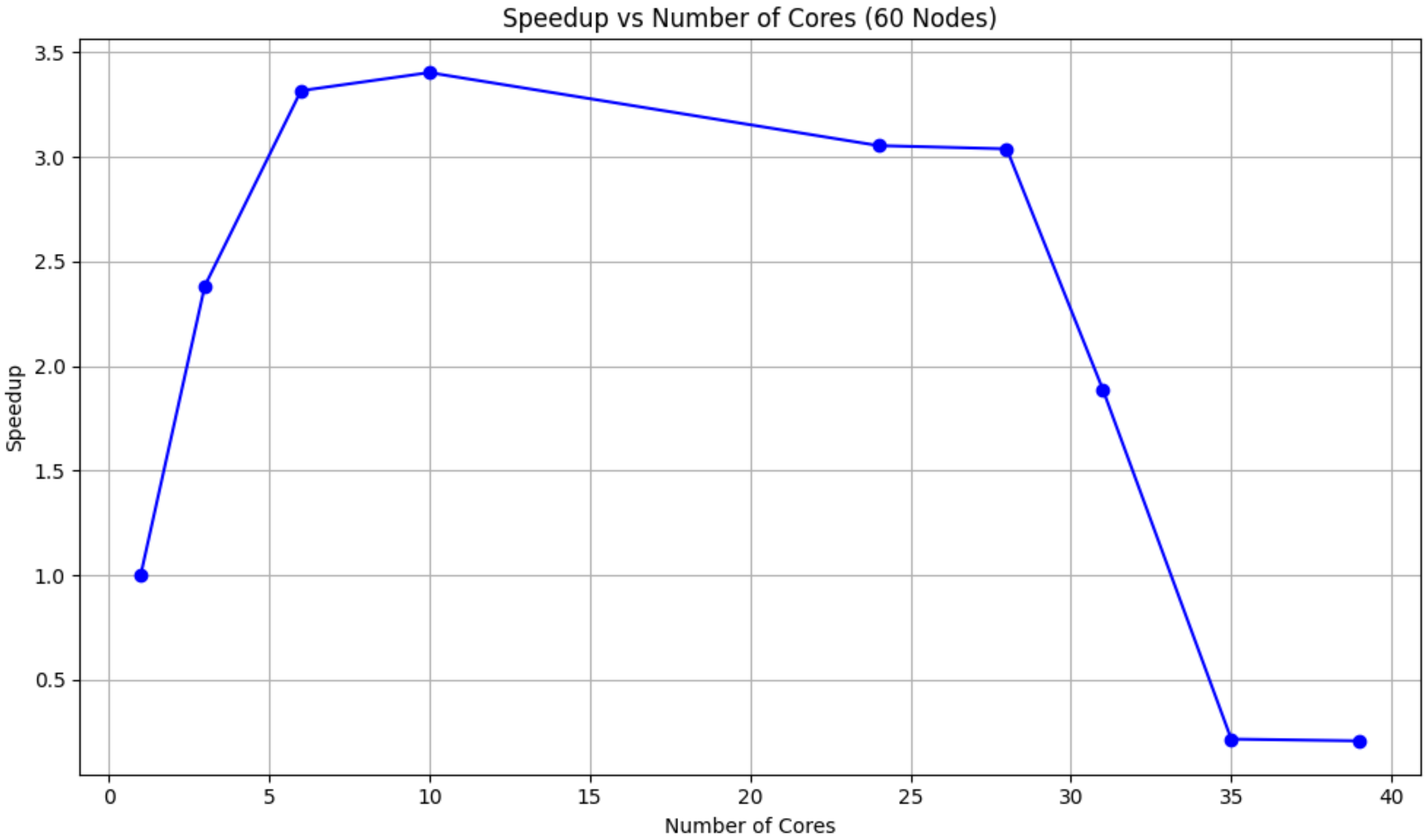
Cost for 50 Nodes with 40000 vertices



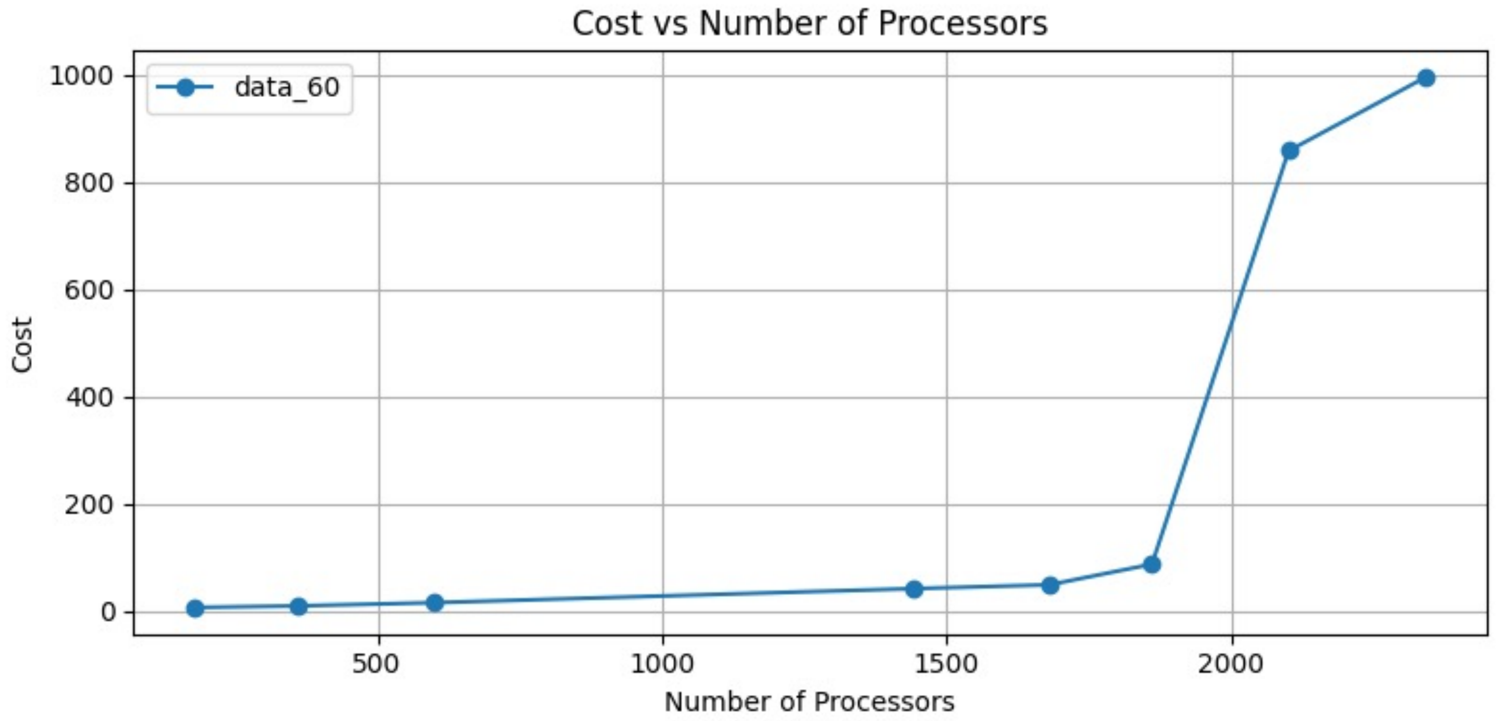
Time taken for 60 Nodes with 40000 vertices



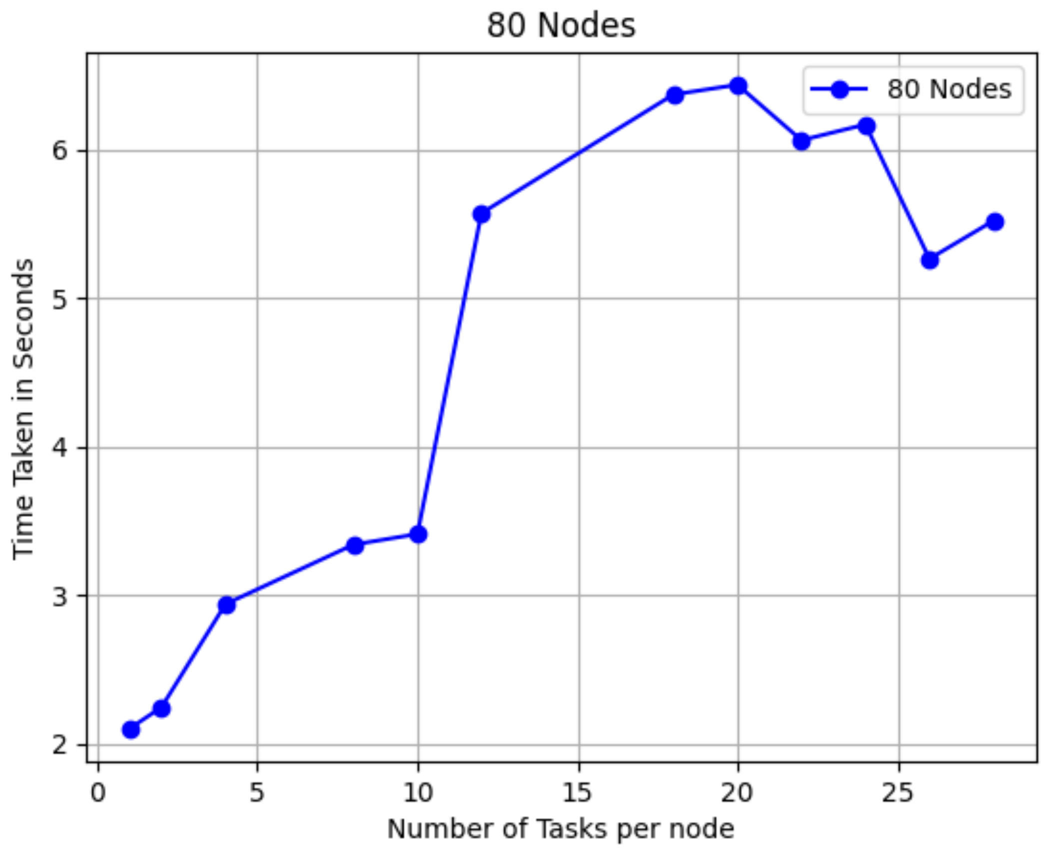
Speedup for 60 Nodes with 40000 vertices



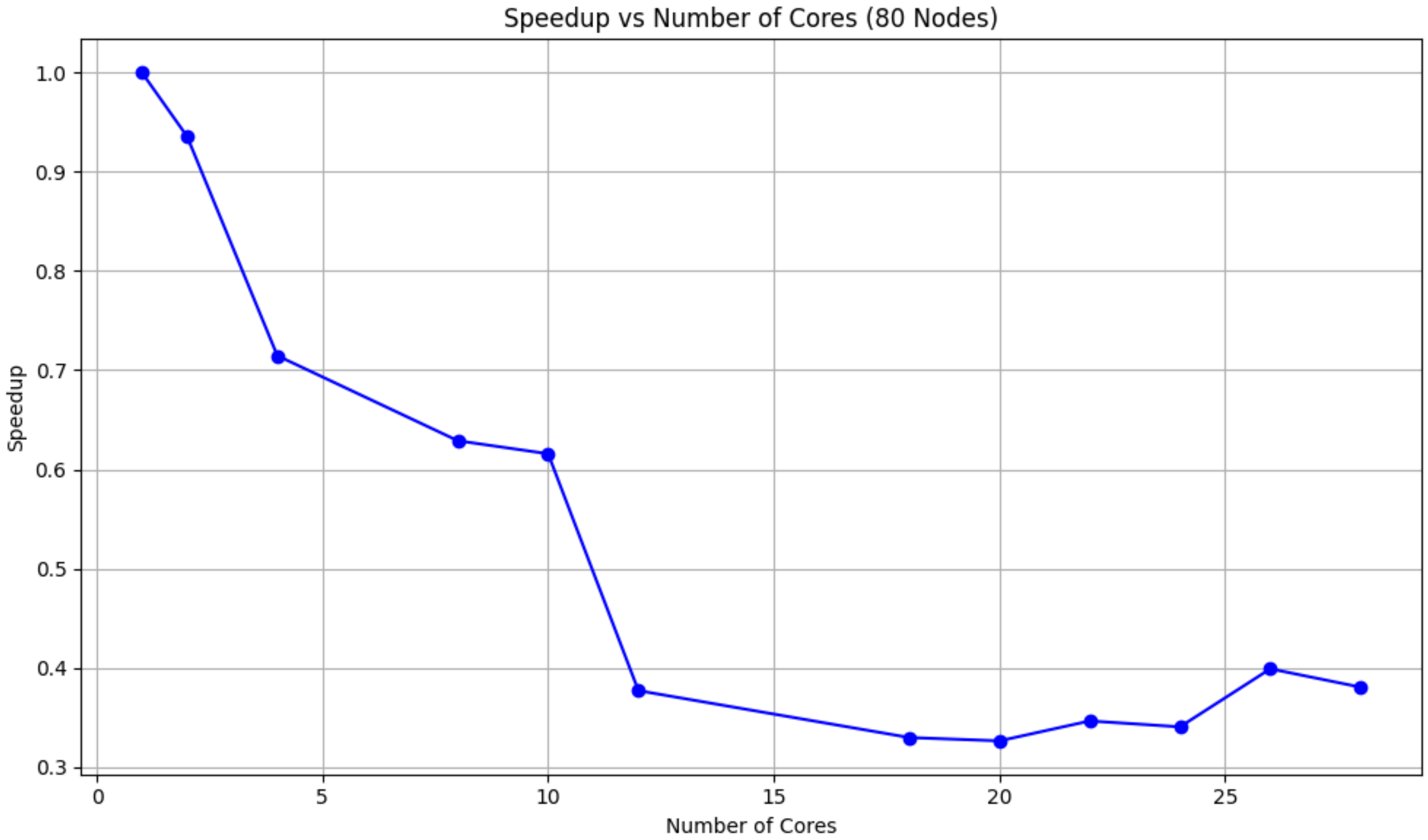
Cost for 60 Nodes with 40000 vertices



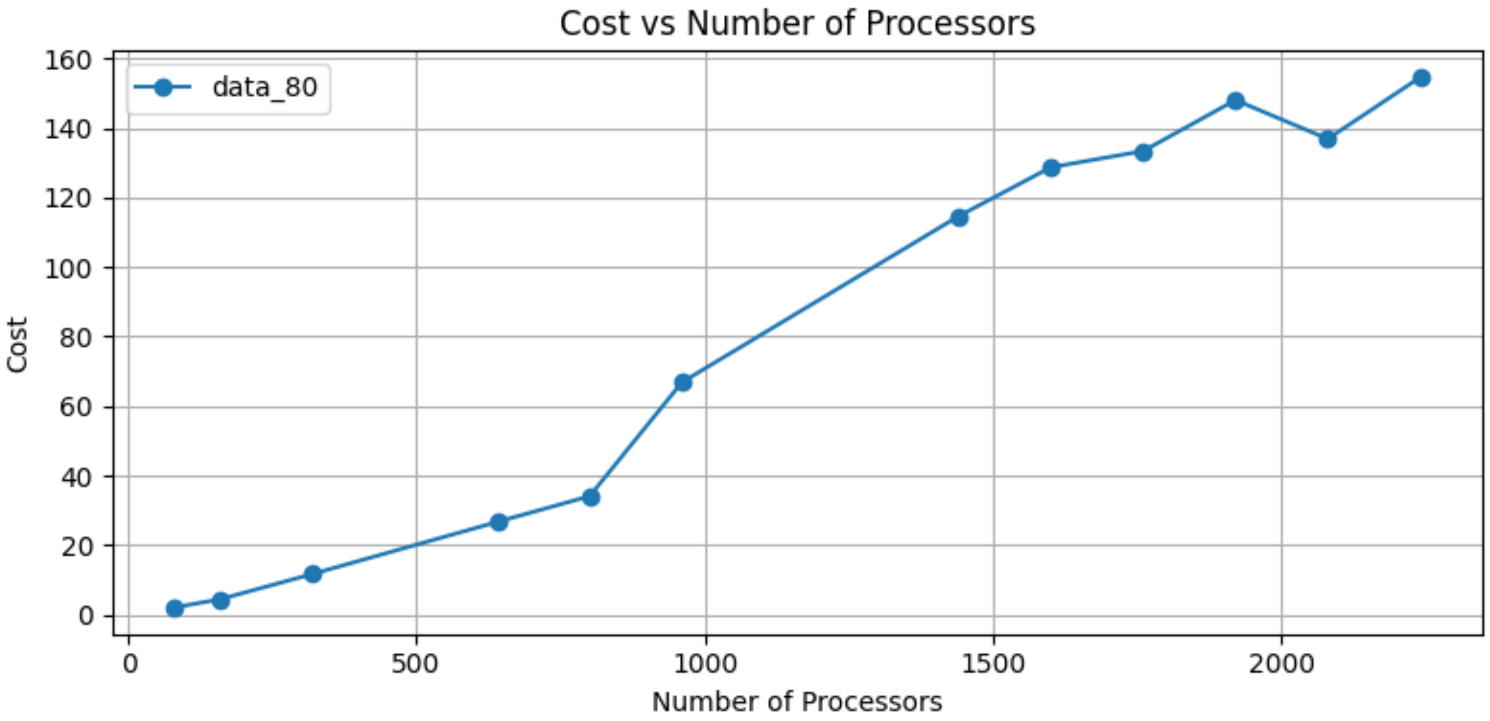
Time taken for 80 Nodes with 40000 vertices



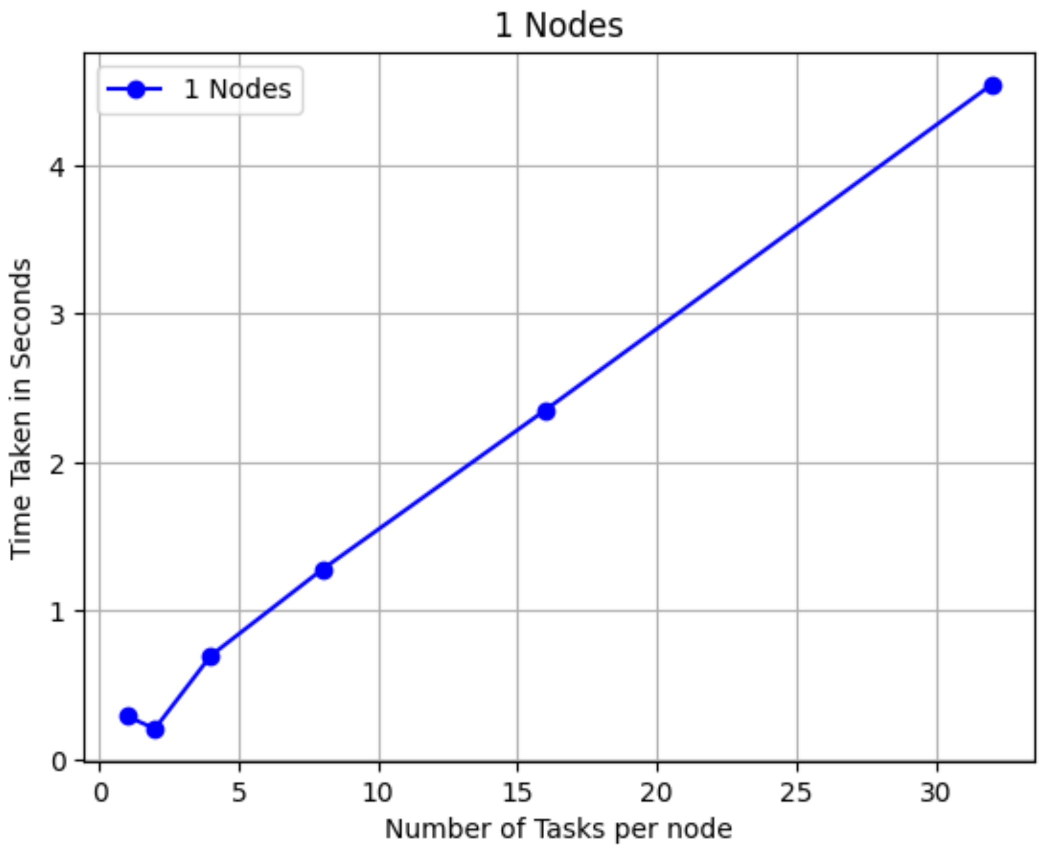
Speedup for 80 Nodes with 40000 vertices



Cost for 80 Nodes with 40000 vertices

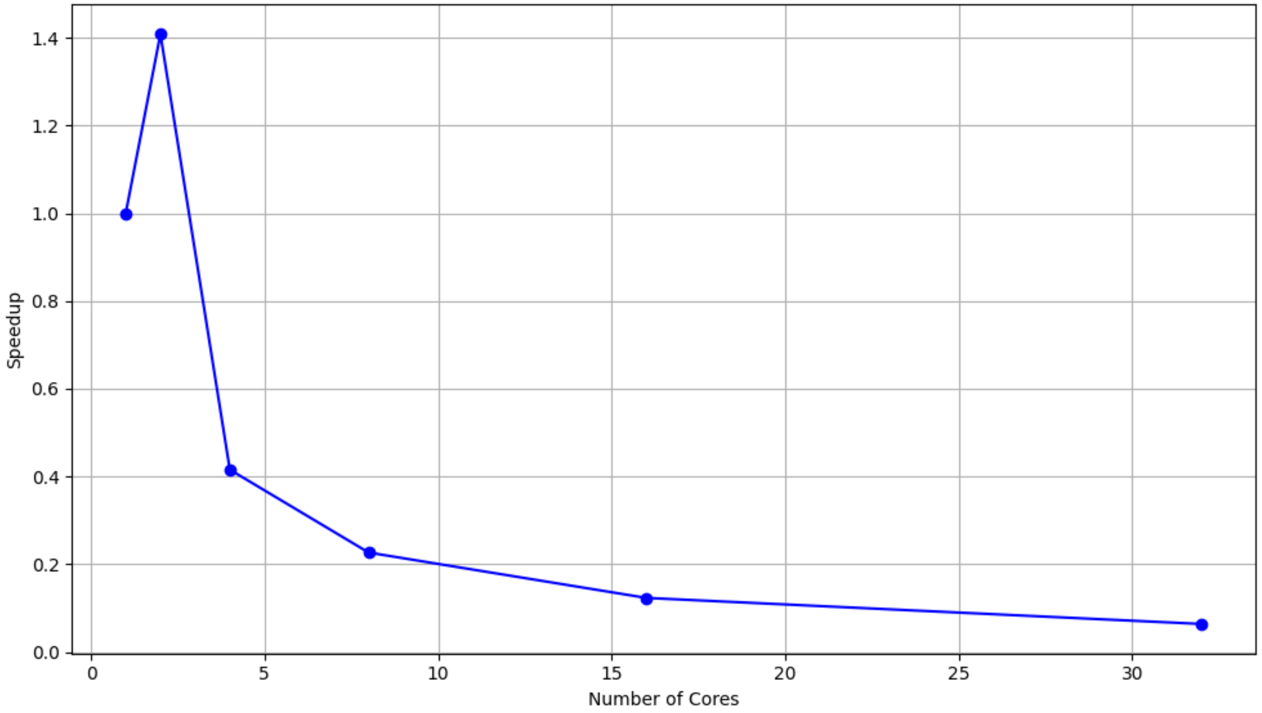


Time Taken for 1 nodes with increasing vertices



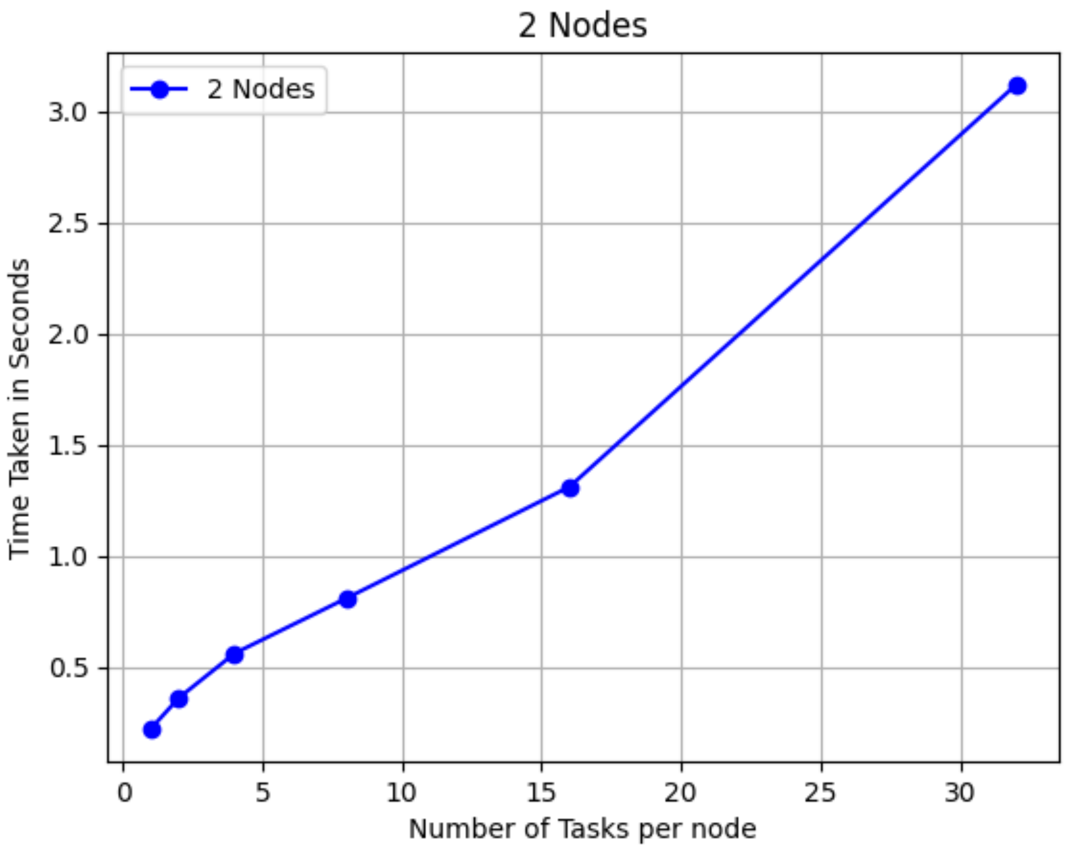
1 core => 625 vertices
2 cores => 1250 vertices
4 cores => 2500 vertices
8 cores => 5000 vertices
16 cores => 10000 vertices
32 cores => 20000 vertices

Speedup for 1 nodes with increasing vertices



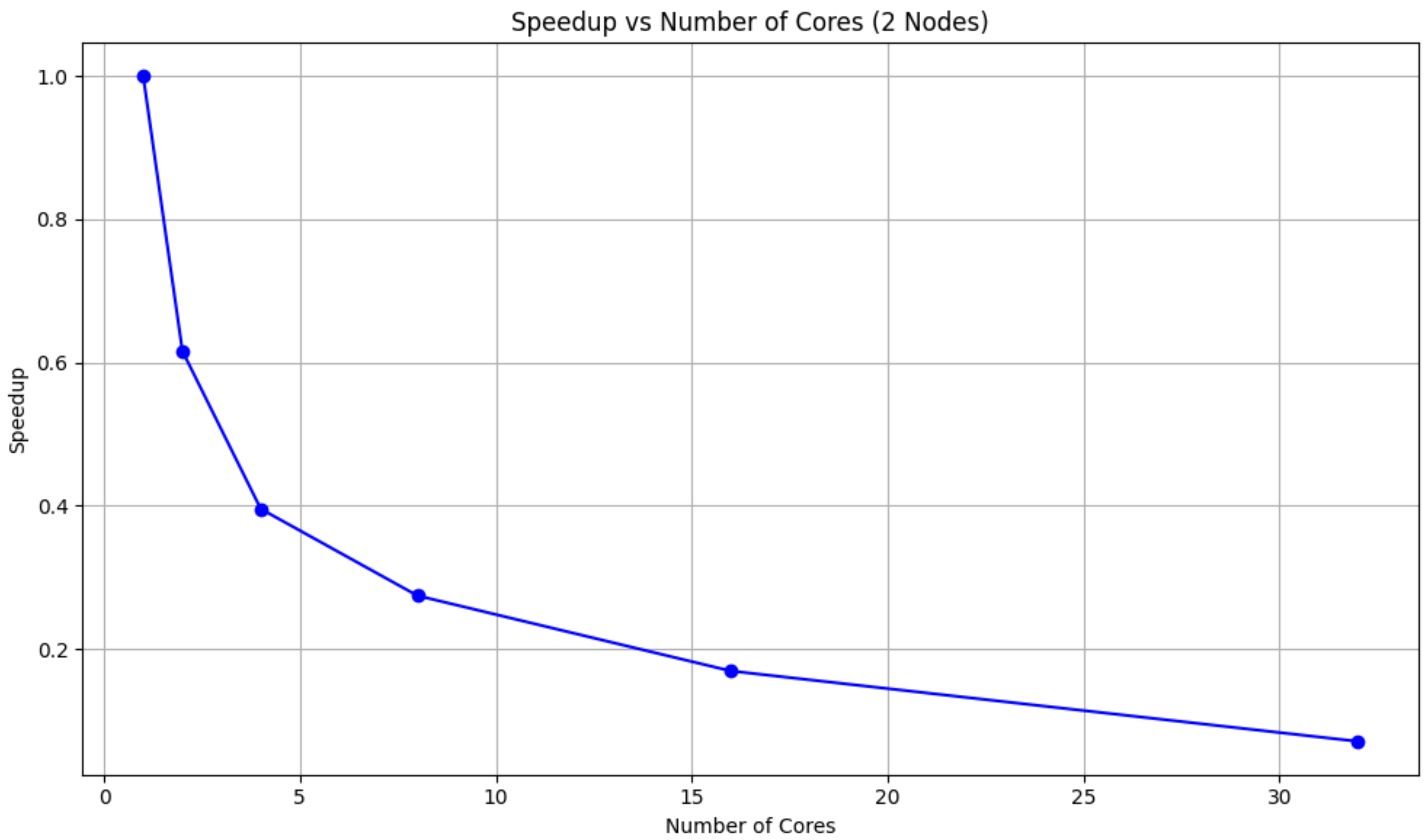
1 core => 625 vertices
2 cores => 1250 vertices
4 cores => 2500 vertices
8 cores => 5000 vertices
16 cores => 10000 vertices
32 cores => 20000 vertices

Time Taken for 2 nodes with increasing vertices



1 core => 625 vertices
2 cores => 1250 vertices
4 cores => 2500 vertices
8 cores => 5000 vertices
16 cores => 10000 vertices
32 cores => 20000 vertices

Speedup for 2 nodes with increasing vertices



Conclusions

- We could see that when increasing the scaling, initially we notice a constant drop in the time taken from parallellization.
- After a certain point, especially with over 20 nodes, and around 40 tasks per node, we see a gradual increase in time taken for most scenarios.
- This tells us that Parallelization is beneficial to us, to a certain extent, but when the overhead of inter process communication, and extent of parallellization is really high, the cons outweigh the pros of parallellization.

References

https://www.researchgate.net/publication/273264449_Understanding_Dijkstra_Algorithm

http://www.algolist.com/Dijkstra's_algorithm

https://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/dijkstra.html

https://resources.saylor.org/wwwresources/archived/site/wp-content/uploads/2012/06/CS408-2.3.2-Dijkstras-algorithm.pdf

Thank you

Kiran Radhakrishnan

kiranrad@buffalo.edu