PARALLELIZATION OF FLOYD WARSHALL ALGORITHM

Guided by: Dr. Russ Miller (UB Distinguished Professor)

CSE 708: Programming Massively Parallel Systems

Presented By: Yashwanth Krishna Porandla

University at Buffalo The State University of New York

Contents

- Introduction of Algorithm
- Need of Parallelization
- Serial Implementation
- Parallel Implementation
- Results and Visualisations
- References

Importance of Floyd Warshall Algorithm

- **Handles Negative Weights**: Unlike some other shortest-path algorithms (like Dijkstra's), Floyd-Warshall can handle graphs with negative edge weights, as long as there are no negative cycles.

- **All-Pairs Shortest Path**: While many algorithms find the shortest path from a single source to all other vertices, Floyd-Warshall computes the shortest paths between every pair of vertices in a graph.

Floyd Warshall Algorithm

The Floyd-Warshall algorithm finds shortest paths in a weighted graph, even with negative edges (but no negative cycles).

- Initialize solution matrix like the input graph matrix.
- For each vertex, update shortest paths using it as an intermediate vertex.
- After all vertices are processed, the matrix contains shortest path distances for every vertex pair.
- Runs in (O(V^3)) time, where V is the number of vertices.

Need For Parallelization

- **Cubic Complexity**: Given Floyd-Warshall's O(V3) time complexity, parallelization can optimize execution for large graphs.
- **Real-time Needs**: Faster computation through parallelization meets demands of applications like traffic management systems or dynamic network routing needing immediate shortest-path updates.

Serial Implementation

- Code Represents the heart of the Floyd-Warshall algorithm, methodically updating the shortest distances between all pairs of vertices.
- Systematically checks and updates the distance matrix, ensuring that every possible vertex combination is evaluated for optimal path determination.

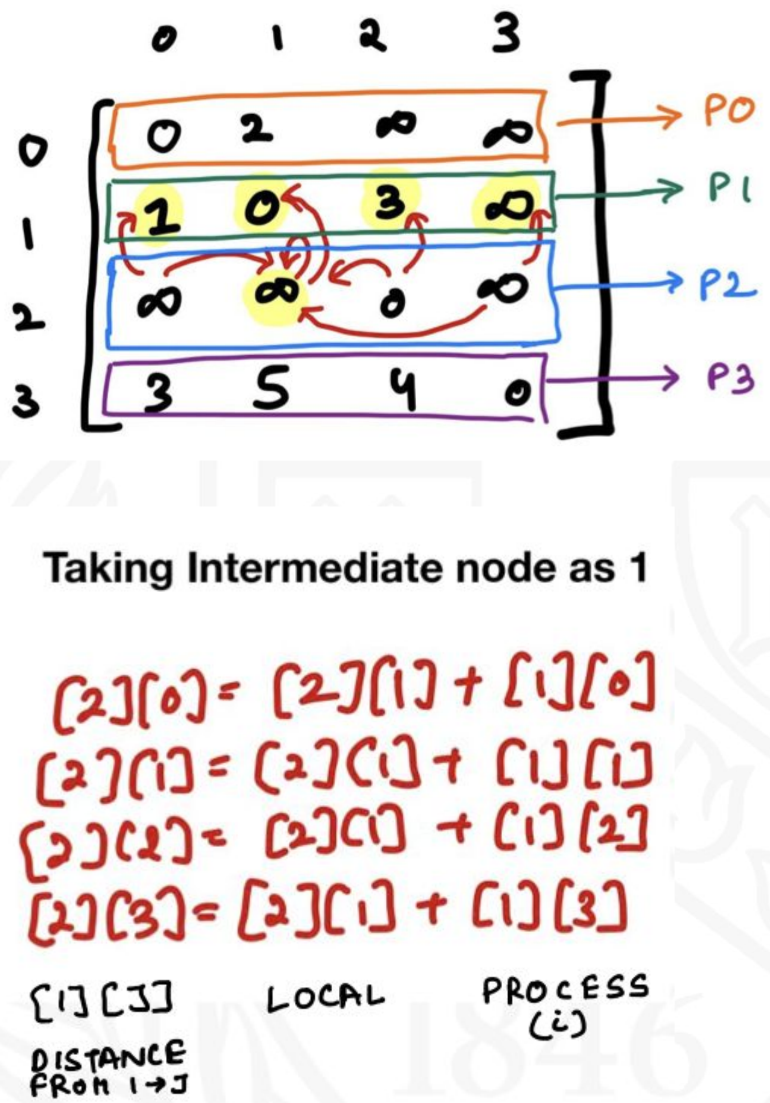
```
n = cardinality(V);
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            if distance[i][j] > distance[i][k] + distance[k][j] then
                distance[i][j] ← distance[i][k] + distance[k][j];
            end
        end
    end
end
```

Parallel Implementation

- The adjacency matrix is divided into rows with each row being assigned to one of the processes. Each process is responsible for calculating the shortest paths within the rows assigned to it.
- Row-wise parallel implementation divides the distance matrix into rows, assigning each row to a separate thread or processor.

Steps to parallelize?

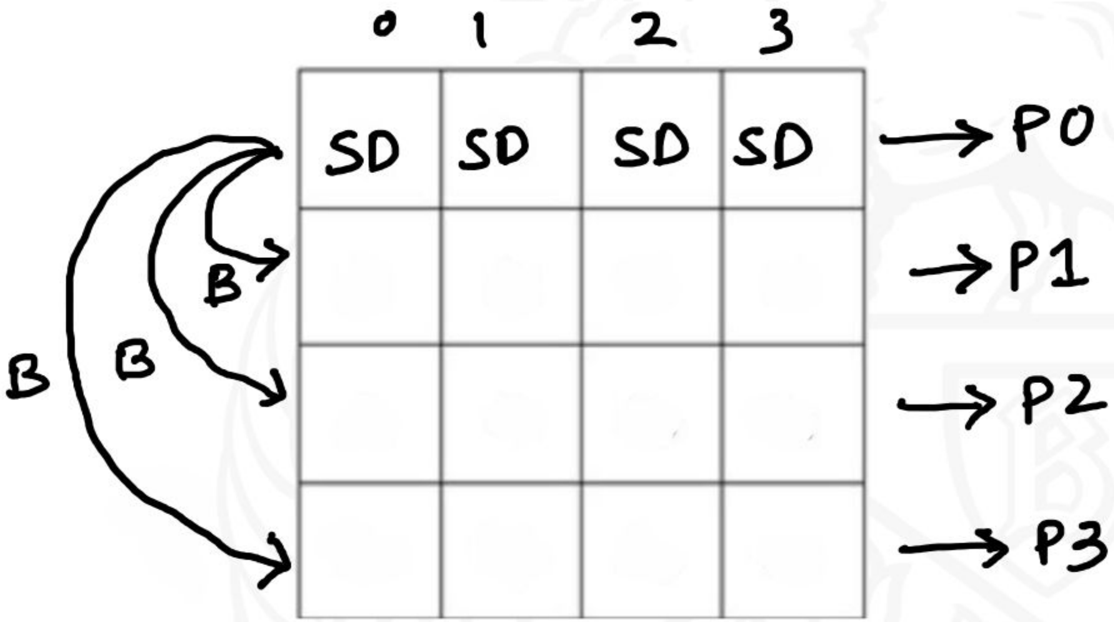
- For each matrix of n*n and p processes, each process is given matrix of size [n]*[n/p]
- From the example in the right, you can clearly see to find the value of arr[i][j], we need arr[i][k] and arr[k][j]
- With the help of row based approach, Process can locally access arr[i][k], but for arr[k][j] you can see that that process assigned with kth row, needs to broadcast all the elements of the kth row to all the processes.



University at Buffalo The State University of New York

Steps to parallelize?

- Once, we update distance of one processes , but othe processes also need these updated distances to calculate distances for their submatrix.
- Before updating kth row, we send it to all the other processes.This enables them to proceed immediately to do their work.



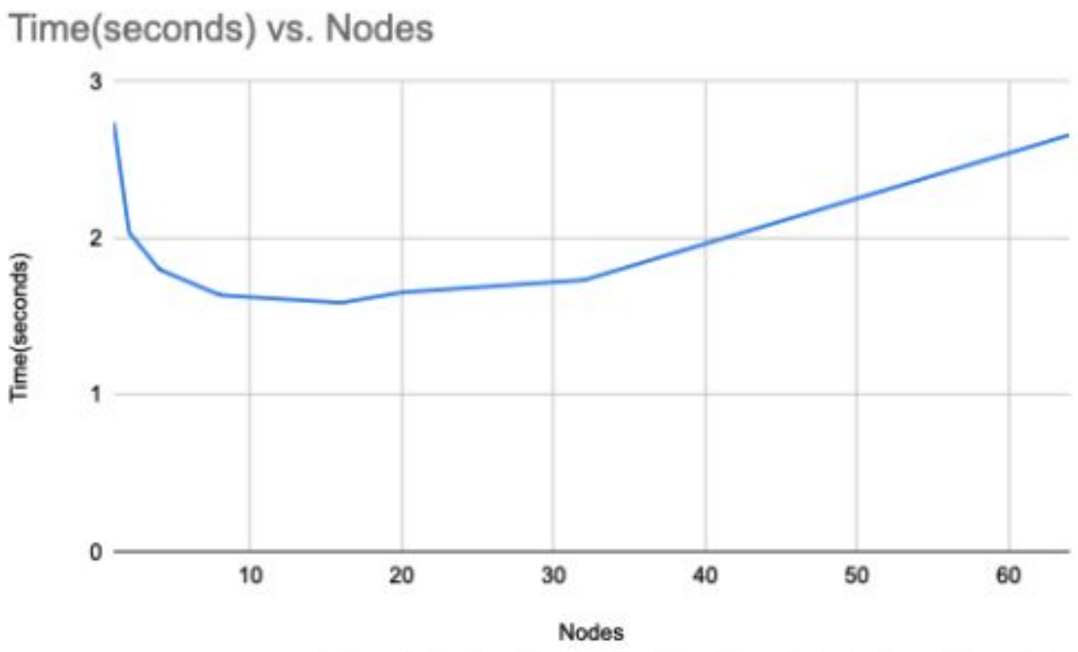
Pseudo code for Parallel Approach

MPI Init
$n \leftarrow$ size of rows
pid $\leftarrow$ id of process
pN $\leftarrow$ number of processes
$D^{(0)} \leftarrow$ input distance matrix
for $k \leftarrow 1$ to n
  do for $i \leftarrow \frac{\text{pid} * n}{\text{pN}}$ to n
    do for $j \leftarrow 1$ to n
      do $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
    send i'st row to another processes
  receive updated rows from another processes
return $D^{(n)}$
MPI Finalize

$$T_{\text{Floyd}} = \frac{N^3}{P} + \text{TIME TO COMMUNICATE}$$

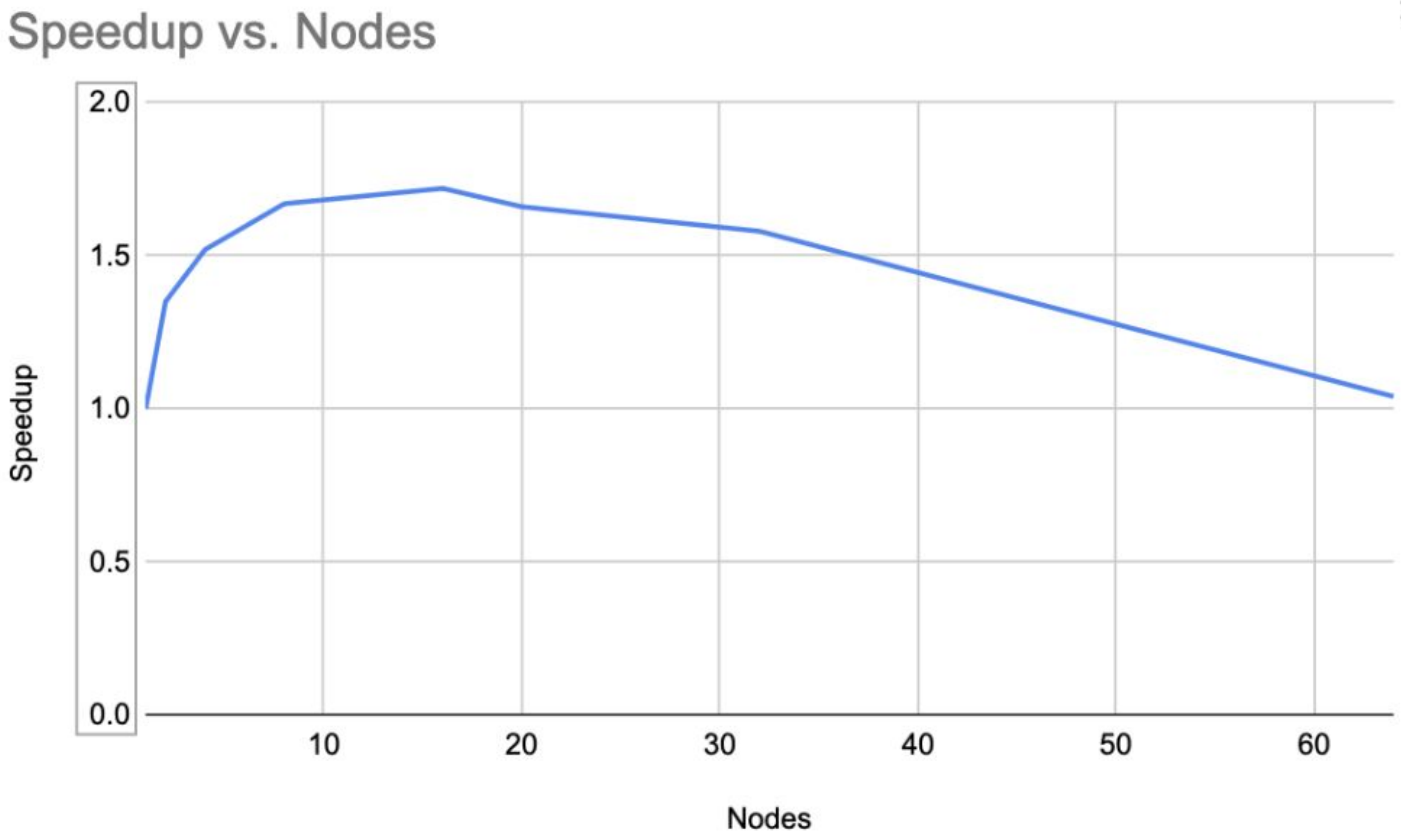
Performance of Row based parallel algorithm on 1000 * 1000 matrix

Nodes	Time(seconds)
1	2.739342
2	2.036491
4	1.801417
8	1.636419
16	1.588156
20	1.654381
32	1.733691
64	2.659988



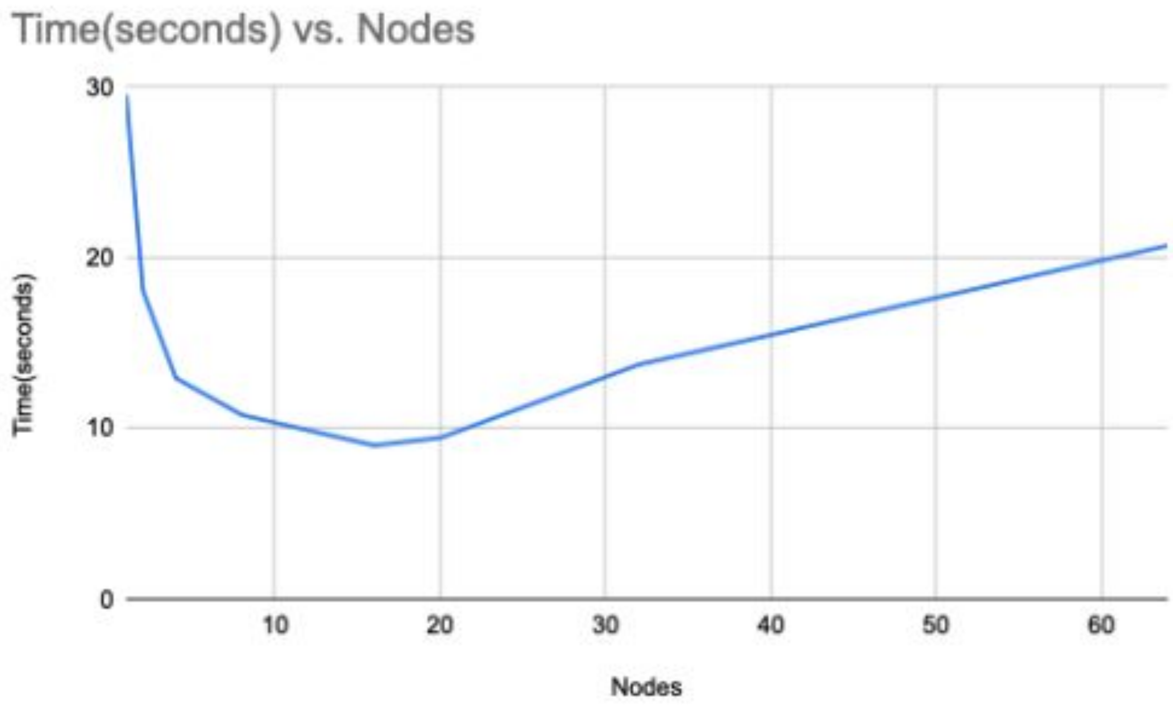
Performance of Row based parallel algorithm on 1000 * 1000 matrix

Nodes	SpeedUp
2	1.345
4	1.521
8	1.674
16	1.725
20	1.656
32	1.580
64	1.117



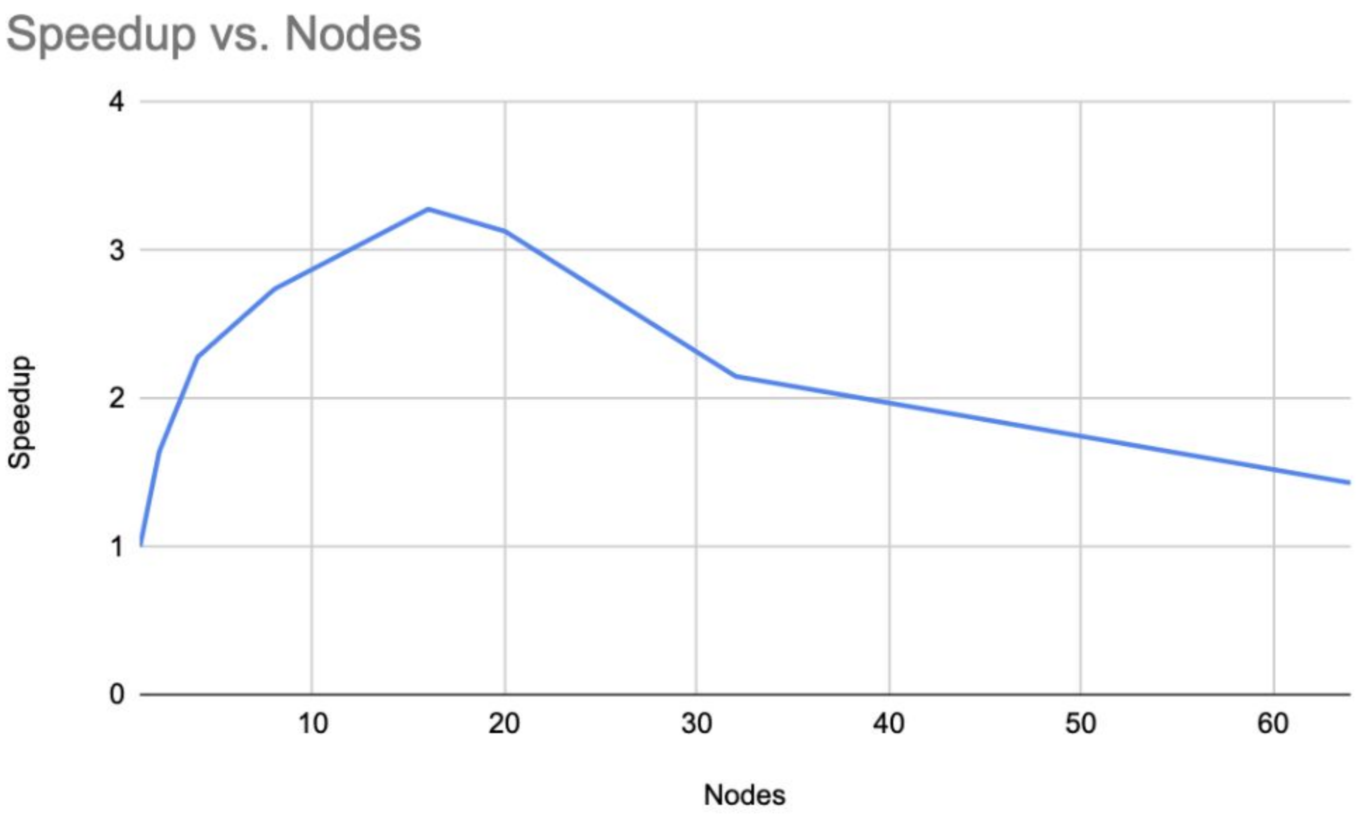
Performance of Row based parallel algorithm on 2500 * 2500 matrix

Nodes	Time(seconds)
1	29.547671
2	18.055083
4	12.945697
8	10.780331
16	9.003291
20	9.450963
32	13.736447
64	20.719030



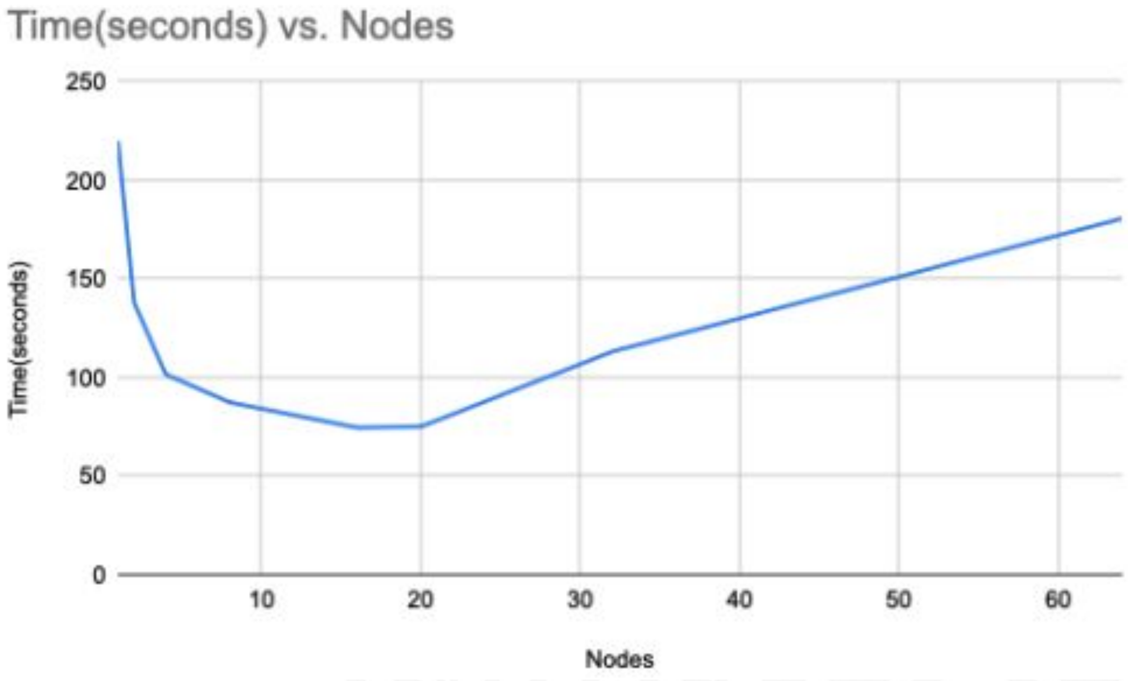
Performance of Row based parallel algorithm on 2500 * 2500 matrix

Nodes	SpeedUp
1	1.0
2	1.64
4	2.28
8	2.74
16	3.28
20	3.13
32	2.15
64	1.43



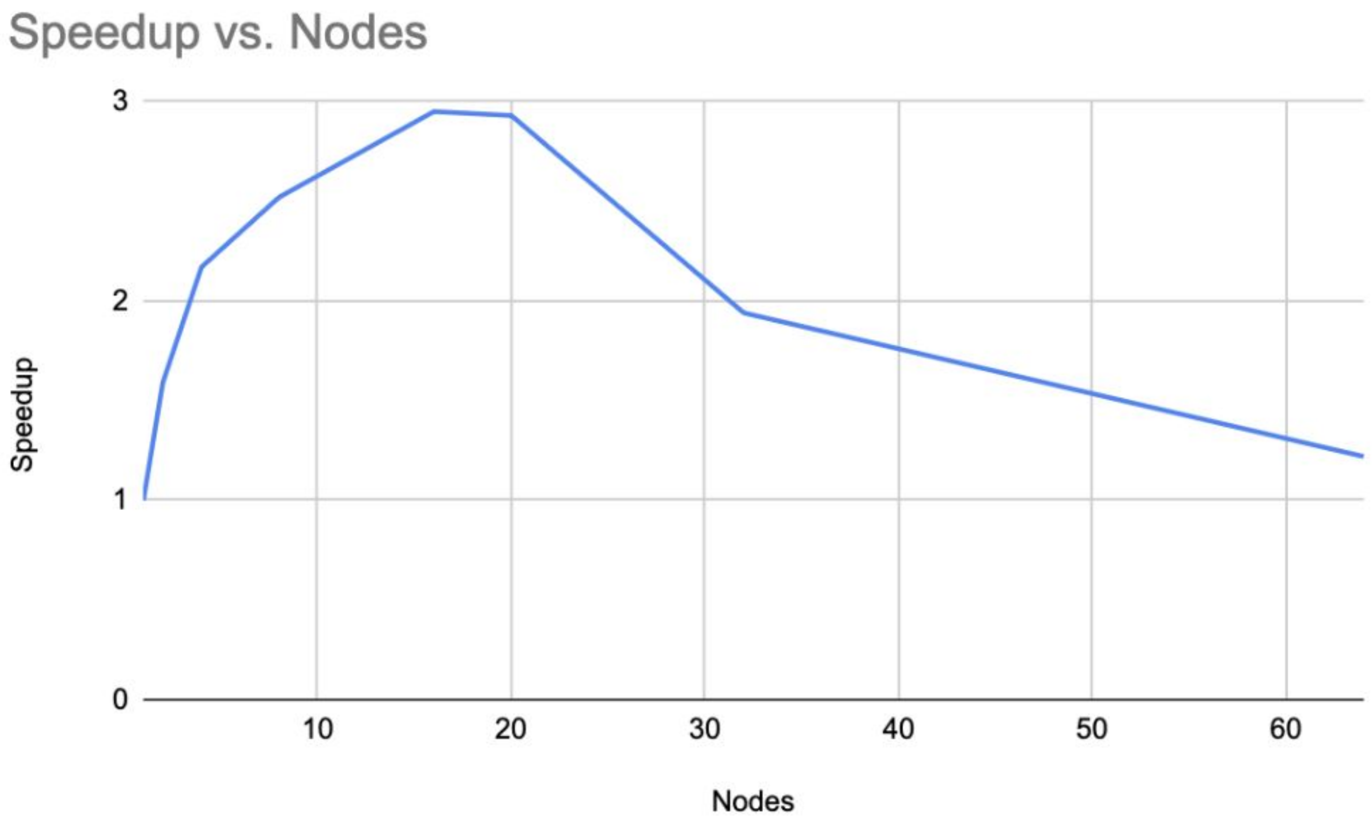
Performance of Row based parallel algorithm on 5000 * 5000 matrix

Nodes	Time(seconds)
1	219.923315
2	137.994994
4	101.533101
8	87.358489
16	74.548490
20	75.028922
32	113.078432
64	180.565270



Performance of Row based parallel algorithm on 5000 * 5000 matrix

Nodes	SpeedUp
1	1.0
2	1.59
4	2.17
8	2.52
16	2.95
20	2.93
32	1.94
64	1.22



References

1. MPI Tutorials. Tutorials · MPI Tutorial. (n.d.). Retrieved March 24, 2023, from https://mpitutorial.com/tutorials/
2. Case Study on Shortest-Path Algorithms. (n.d.). Retrieved March 20, 2023, from https://www.mcs.anl.gov/~itf/dbpp/text/node35.html