PARALLELIZING THE FLOYD-WARSHALL ALGORITHM

Guided by : Prof. Russ Miller

CSE 633 : Parallel Algorithms

Presented by : Prashant Godhwani

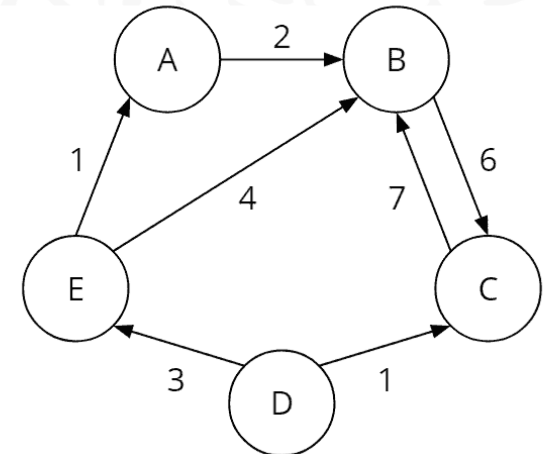
University at Buffalo The State University of New York

Contents

- Introducing Floyd-Warshall Algorithm
- Why parallelizing makes sense?
- Serial Implementation
- Parallel Implementation
- Unequal Row Distribution
- Negative Cycle detection
- Results and Visualizations
- Planning for Block Based Floyd-Warshall
- Progress Report
- What’s next?
- References

What is Floyd-Warshall Algorithm?

An algorithm for finding the shortest path between all pair of nodes in a directed graph.



Some points to keep in mind –

1) While algorithms like Dijkstra's and Bellman Ford also find shortest path between source and destination nodes, Floyd-Warshall takes it a step further and calculates shortest distance between every pair of node unlike the former two.

2) The algorithm has many applications ranging from Network Routing, Transportation Planning to Social Networks and Robotics.

Applications

Why do we care about parallelizing it?

- Time Complexity of Floyd-Warshall is $O(n^3)$ where n is the number of nodes in the graph.
- This is because for each pair of vertices, the algorithm considers all possible intermediate vertices and computes the shortest path between them. Therefore, the running time of the algorithm is proportional to the cube of the number of vertices in the graph.

$$d_{ij}^{(k)} \leftarrow min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

- This means that –

1000
COMPUTATIONS FOR 10 NODES

1,000,000
COMPUTATIONS FOR 100 NODES

1,000,000,000
COMPUTATIONS FOR 1,000 NODES

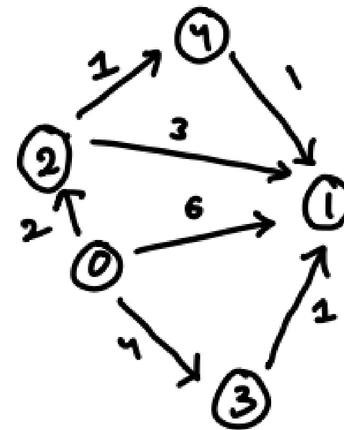
Okay, but how does it work, Serially?

The way Floyd-Warshall algorithm works is by traversing over every intermediate node and finding a shortest path from Source S, to destination D, going through Intermediate node I.

Here pointer k is traversing over the intermediate nodes, i is traversing over all the sources and j over all the destinations possible,

```
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
```

Example Graph –



Paths from 0→1

0→1 = 6

0→2→1 = 5

0→3→1 = 5

0→3→4→1 = 4

Shortest Path

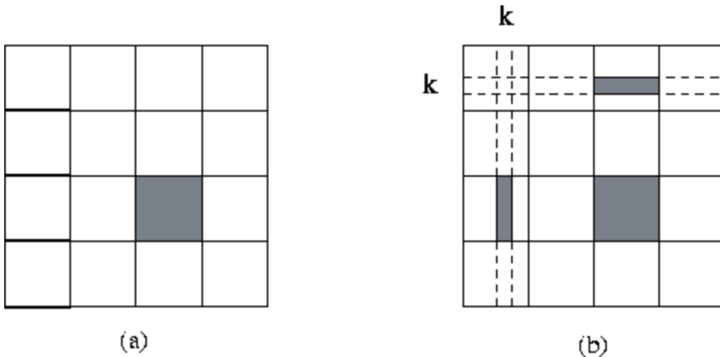
Now, how do you parallelize it?

Can be parallelized using different ways, but two approaches that I considered were -

BLOCK BASED PARALLELIZATION

In this approach, the adjacency matrix is divided into blocks with each block being assigned to one of the processes. Each process is responsible for calculating the shortest paths within its block.

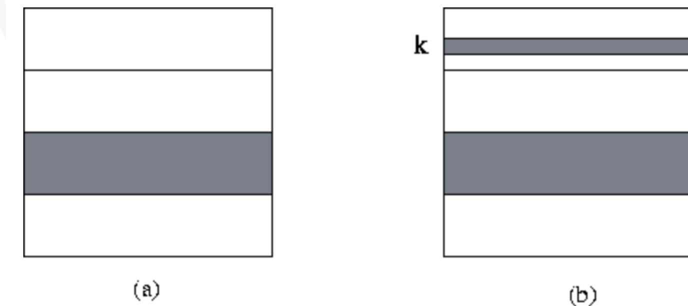
WHEN – More efficient when matrix is large, and processes are less.



ROW BASED PARALLELIZATION

In this approach, 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.

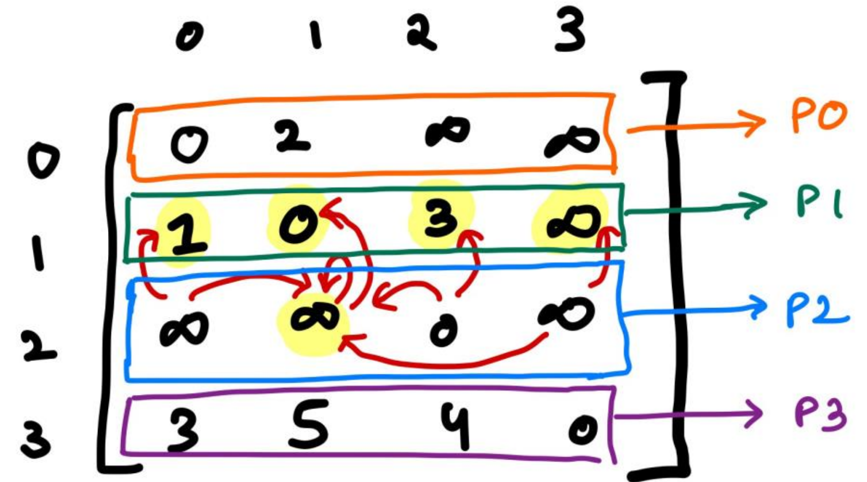
WHEN – More efficient when matrix is small, and processes are more.



What and how ?

To get things up and moving, I decided to proceed with *Row based parallelization*.

- For each matrix of size n x n, and p processes, each process receives a matrix of size [n/p][n].
- As we can see on the right, to fill in the value of arr[i][j], we need arr[i][k] and arr[k][j].
- Thanks to the row-based approach, process P can locally access arr[i][k], but for arr[k][j], the process that was assigned kth row, has to somehow broadcast all the elements of the kth row to all the processes.



Taking Intermediate node as 1

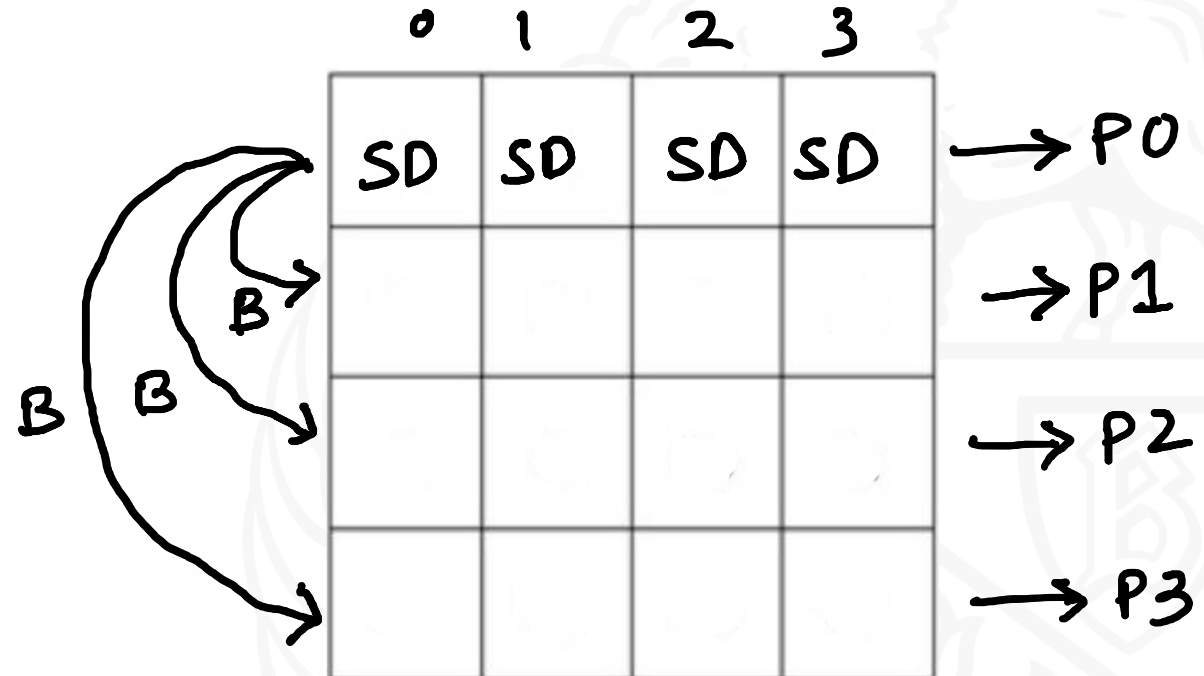
[2][0] = [2][1] + [1][0]
[2][1] = [2][1] + [1][1]
[2][2] = [2][1] + [1][2]
[2][3] = [2][1] + [1][3]

[I][J] — DISTANCE FROM I→J
LOCAL
PROCESS (i)

Okay, what about Other Processes?

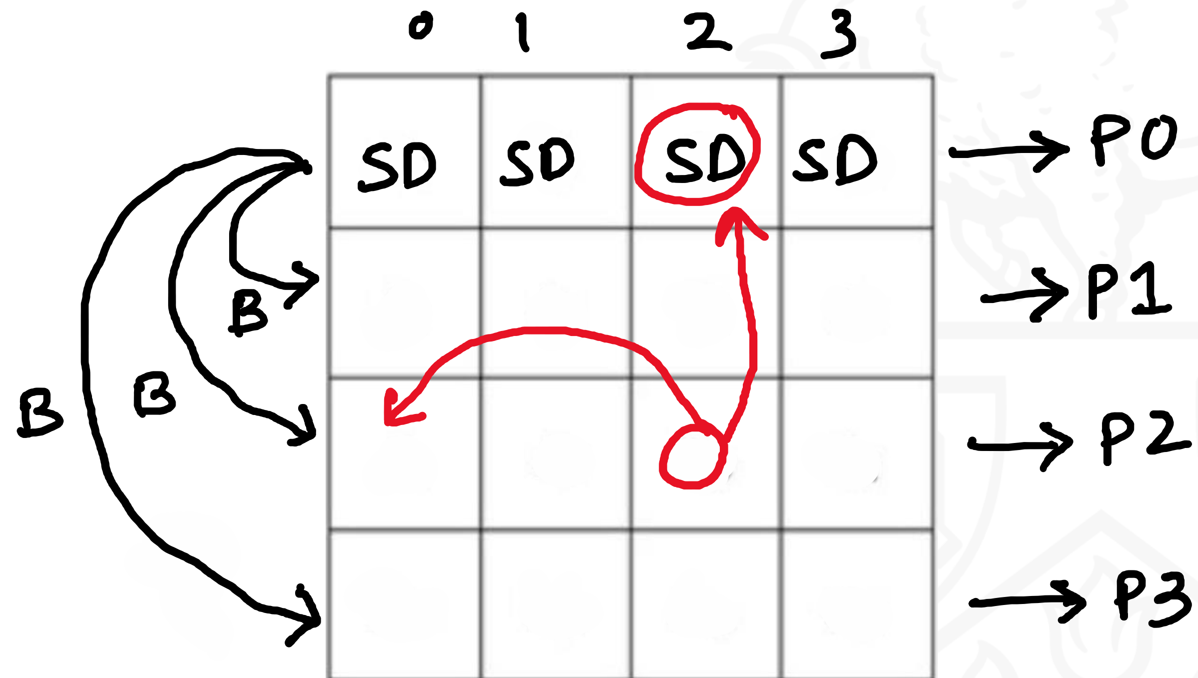
So, we updated the distances for one processes, but other processes also need these updated distances to calculate distances for their sub-matrix.

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



For i=2, j=2 and k=0

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

Unequal Row Distribution

- Worked on unequal row distribution.
- Earlier, the program where number of rows where not divisible by the number of processes would give incorrect results.
- Calculated sendCounts and displacements to be used for Scatterv and Gatherv.
- Scatterv and Gatherv unequally distributed the rows to different processes.

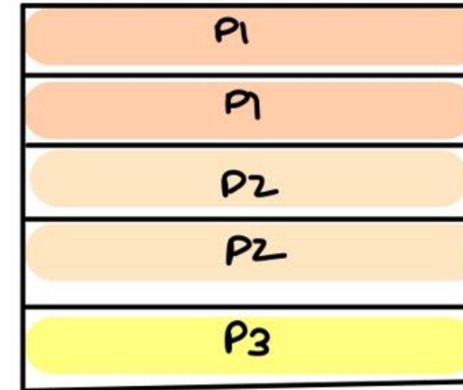
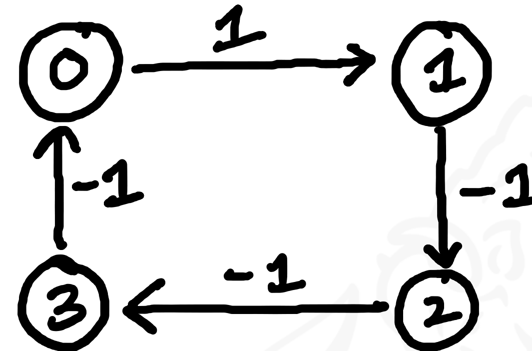


Fig: Row distribution with 5 Rows and 3 Processes

```
sendcounts = new int[noOfProcesses];
displs = new int[noOfProcesses];

for (int i = 0; i < noOfProcesses; i++)
{
    sendcounts[i] = (i < rem) ? (rowsPerProcess + 1) * v : rowsPerProcess * v;
    displs[i] = i > 0 ? displs[i - 1] + sendcounts[i - 1] : 0;
}
```

Detect Negative Cycles

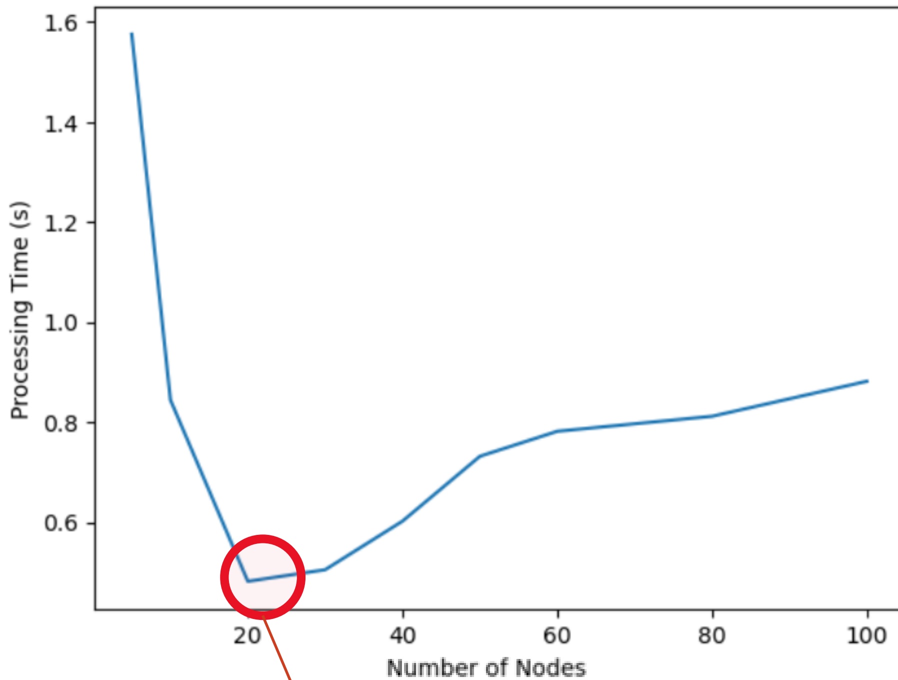


- We know that minimum distance from one vertex to itself is always zero.
- However, if we see that the distance is negative, we can imply that there must be a negative cycle as the distance after traversing got negative.

```
1  for (int i = 0; i < rowsPerProcess; i++)
2  {
3      if (localArr[i * n + i] < 0)
4      {
5          cout << "Error: Negative Cycle Found\n";
6          MPI_Abort(MPI_COMM_WORLD, 1000);
7      }
8  }
```

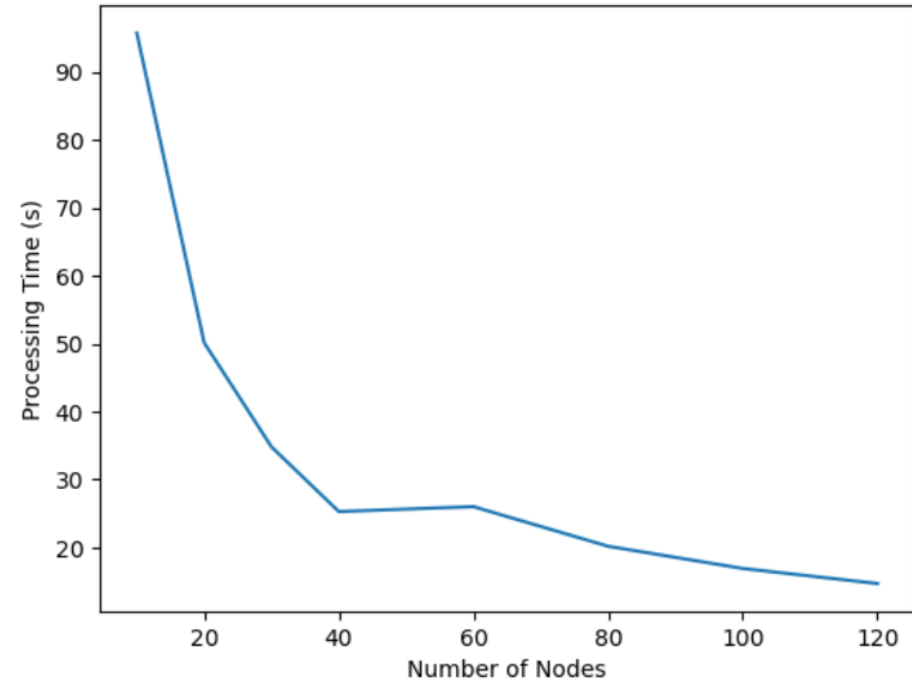
Performance

With [2,000 x 2,000] matrix
4 Million elements



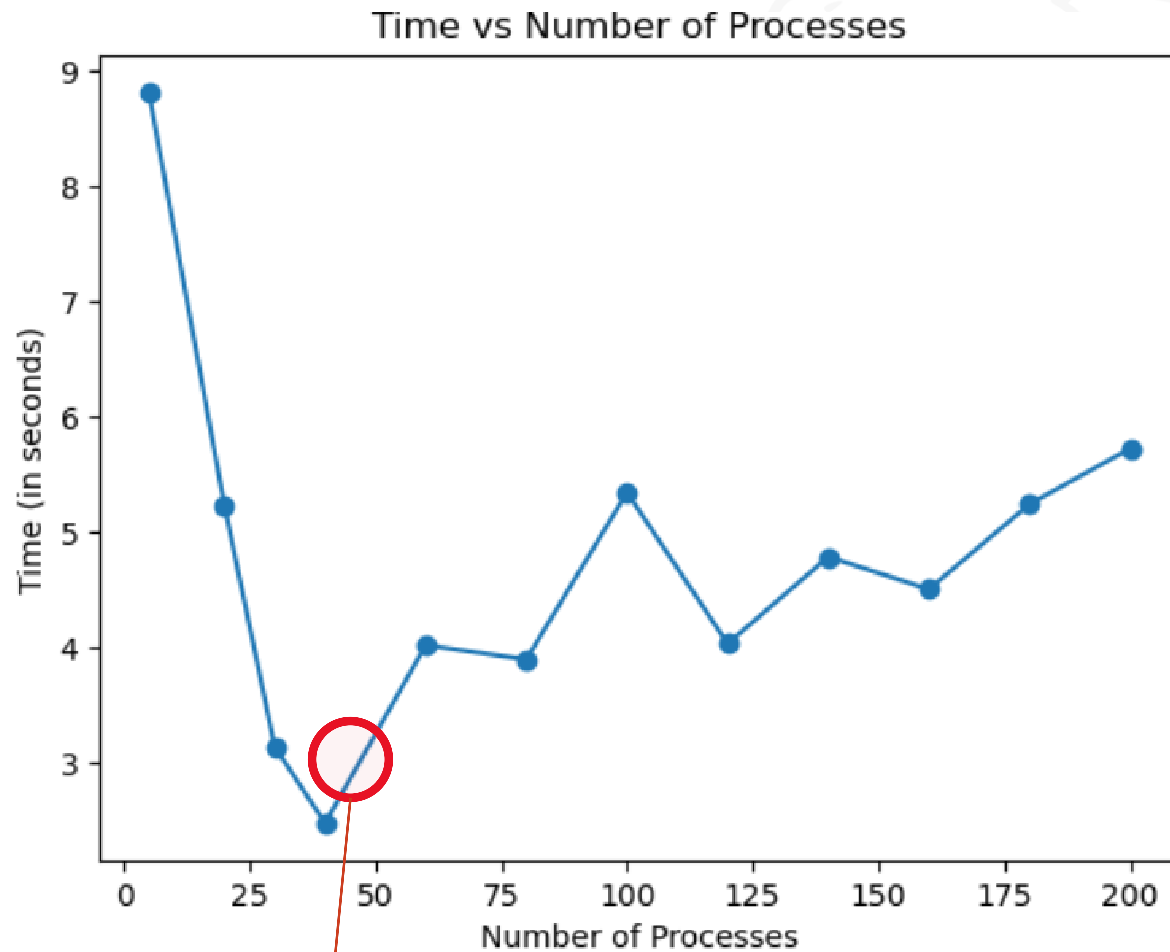
Communication overtakes Computation

With [10,000 x 10,000] matrix
100 Million elements



Scaling Processes with Minimal Nodes
With [4,000 x 4,000] matrix

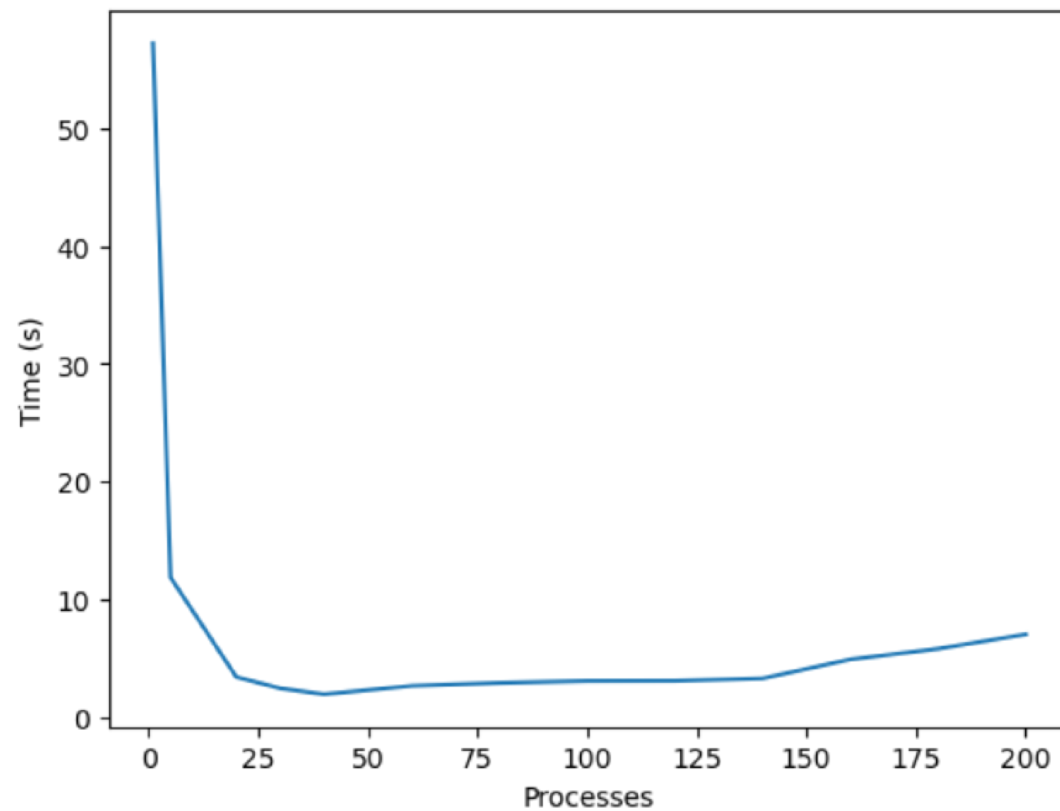
Processes	Nodes	Processes /Node	Time (in seconds)
1	1	1	57.203
5	1	5	8.81601
20	1	20	5.23303
30	1	30	3.1345
40	1	40	2.47199
60	2	30	4.01625
80	2	40	3.89197
100	2	50	5.34201
120	3	40	4.02888
140	3	47	4.78264
160	3	40	4.50202
180	4	45	5.24053
200	4	50	5.72356



Scaling Nodes with Minimal Processes

With [4,000 x 4,000] matrix

Processes	Nodes	Processes /Node	Time (in seconds)
1	1	1	57.203
5	5	1	11.8571
20	20	1	3.39511
30	30	1	2.43993
40	40	1	1.93742
60	60	1	2.64698
80	80	1	2.898319
100	50	2	3.06283
120	60	2	3.07363
140	70	2	3.2733
160	40	4	4.89029
180	60	3	5.79321
200	70	4	7.01954



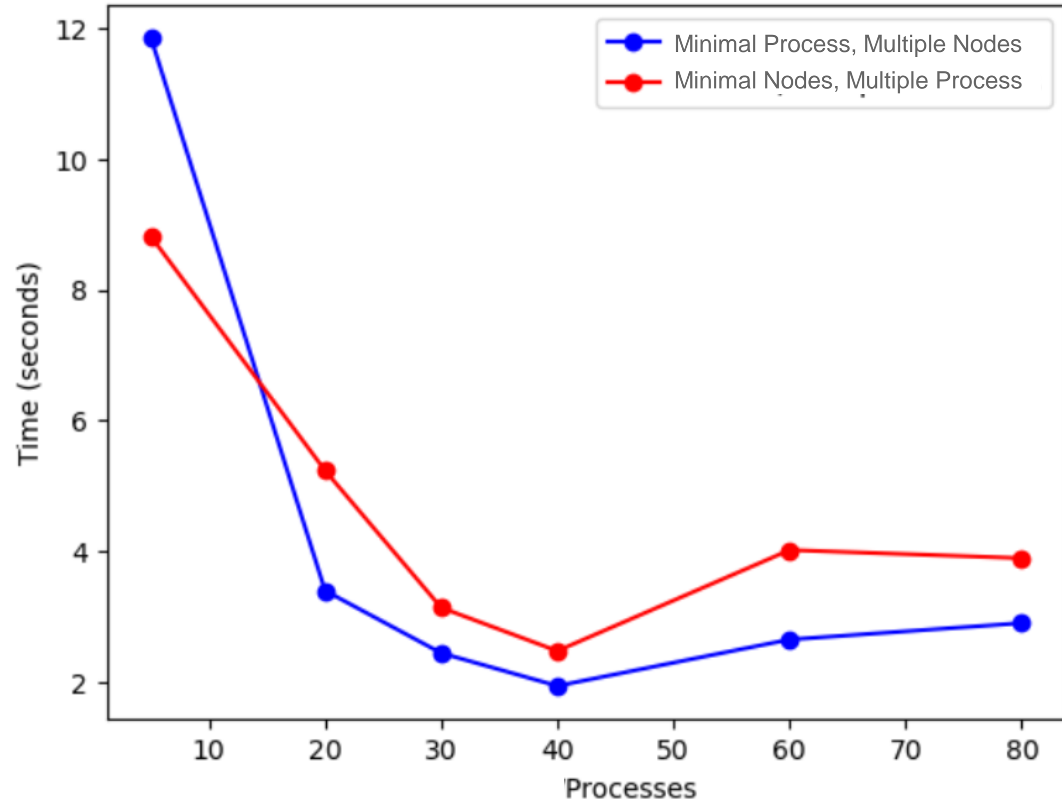
Processes	Nodes	Processes/Node	Time (in seconds)
1	1	1	57.203
5	5	1	11.8571
20	20	1	3.39511
30	30	1	2.43993
40	40	1	1.93742
60	60	1	2.64698
80	80	1	2.898319

Scaling Nodes with Minimal Processes

Processes	Nodes	Processes/ Node	Time (in seconds)
1	1	1	57.203
5	1	5	8.81601
20	1	20	5.23303
30	1	30	3.1345
40	1	40	2.47199
60	1	60	4.01625
80	2	40	3.89197

Scaling Processes with Minimal Nodes

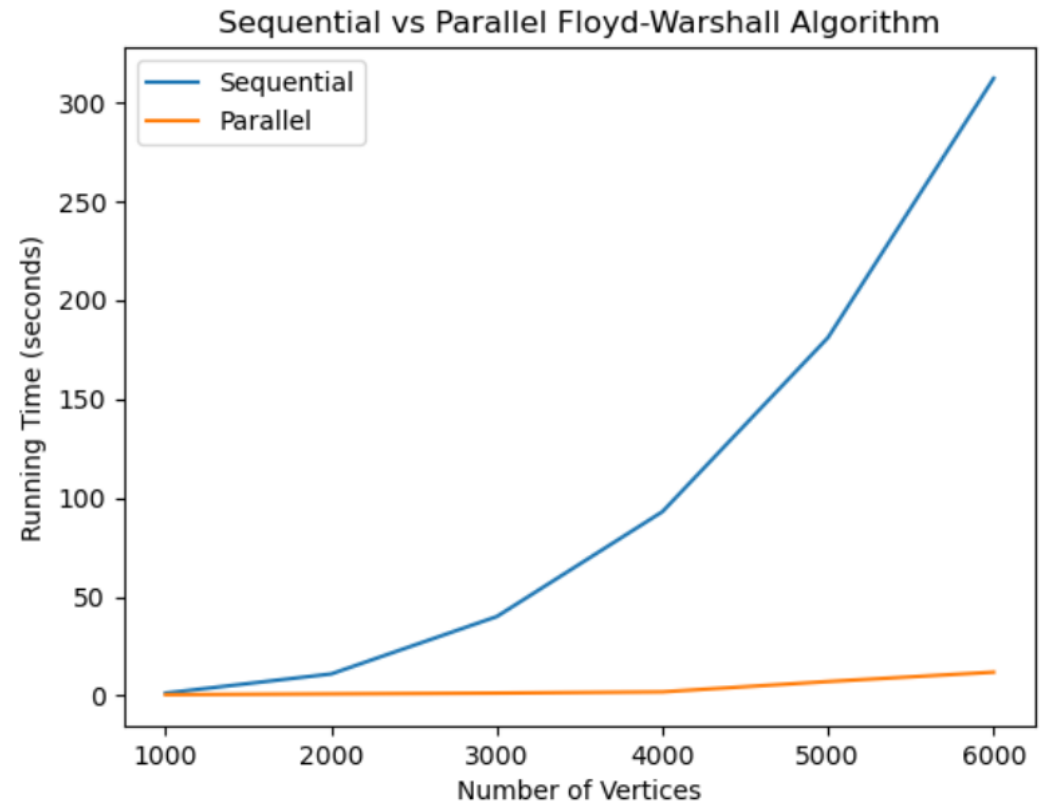
With [4,000 x 4,000] matrix – Comparing Multiple Nodes vs Multiple Processes



Note: More time taken by **"Minimal Nodes, Multiple Processes"** owing to Context Switching by CPU and/or Resource Contention at Nodes.

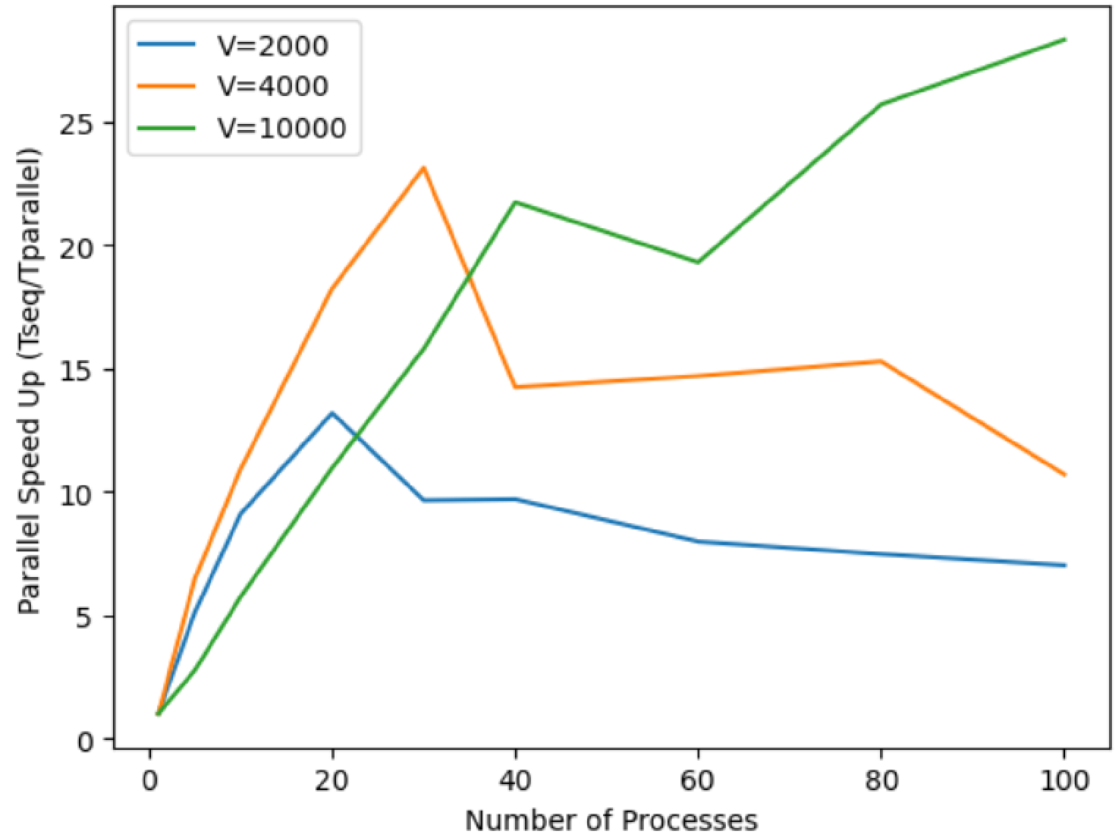
Contrasting Serial vs Parallel Running Times

- Varied the vertices of the Graph keeping the number of nodes constant *(40)*.
- With 6000 vertices, Sequential process takes around 312s whereas Parallel takes around 11.90s.



Speed Up

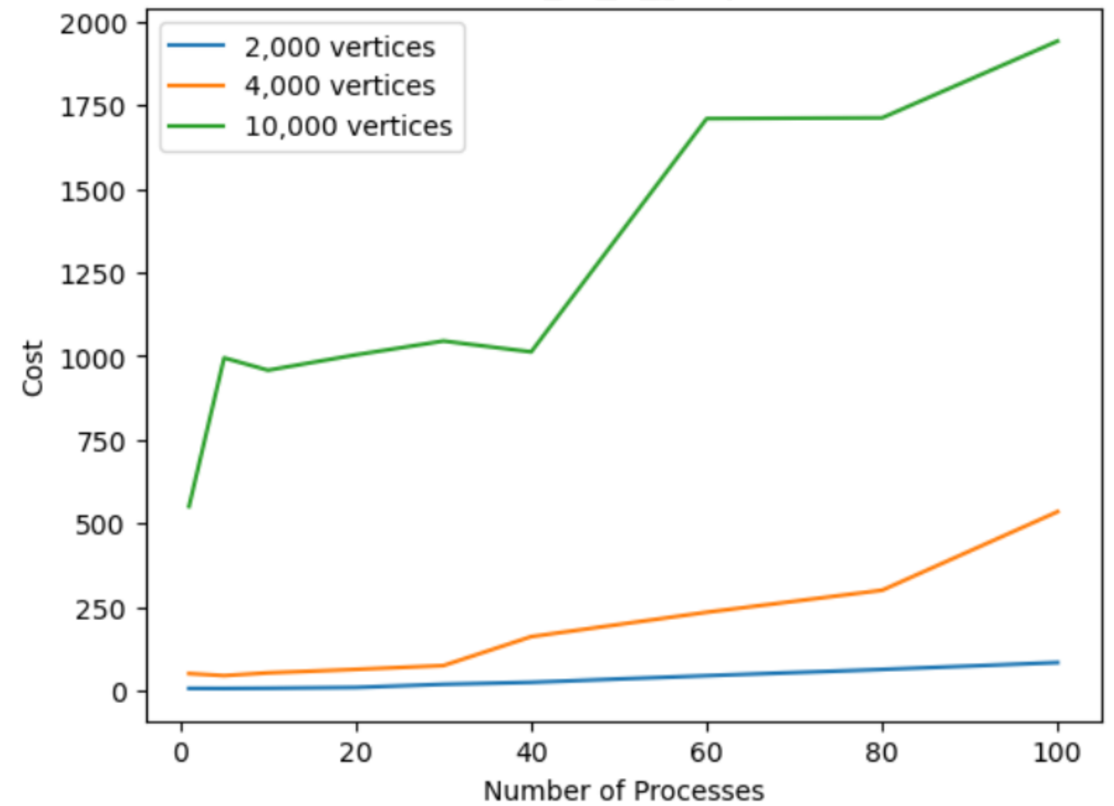
- Calculated using Tseq / Tparallel.
- For 2000 vertices, the max speedup of 13 comes around when 20 processes work in parallel.
- For 4000 vertices, speedup of 23 comes around with 40 processes. After which communication overtakes computation and the speedup decreases.
- For 10,000 vertices, max speedup might come later (couldn't compute due to limited computational resources).



Cost

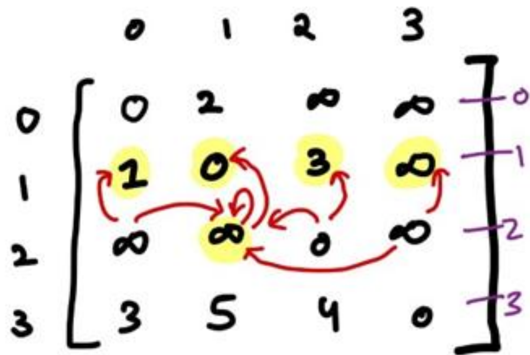
- Shows effect of cubic-algorithm complexity.
- Calculated using

Number of Processes * $T_{parallel}$



Planning for Block Based Floyd-Warshall

ROW BASED



Taking Intermediate node as 1

P2 P2 $P_{K=1}$

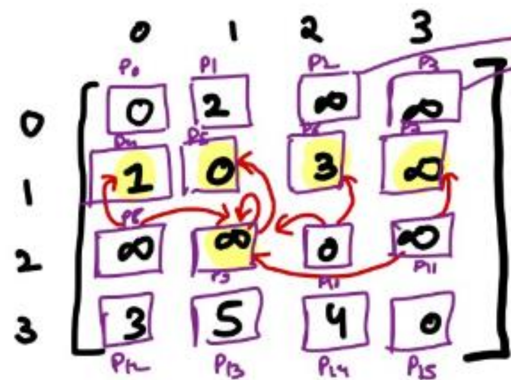
[2][0] = [2][1] + [1][0]

[2][1] = [2][1] + [1][1]

[2][2] = [2][1] + [1][2]

[2][3] = [2][1] + [1][3]

BLOCK BASED



different Nodes

P8 P9 P4

[2][0] = [2][1] + [1][0]

[2][1] = [2][1] + [1][1]

[2][2] = [2][1] + [1][2]

[2][3] = [2][1] + [1][3]

Progress Report

1. Looking into - sbatch: error: Batch job submission failed: Requested node configuration is not available for requesting more than 140 nodes – **Infiniband has max spread of 140 nodes. Increased Processes thereafter.** ✔
2. Draw Inferences and Calculate Speed Up – **Done** ✔
3. Trying to make the algorithm work for cases where number of rows is not completely divisible by number of processes. Exploring MPI_Scatterv and MPI_Gatherv for that – **Done.** ✔
4. Next, I intend to work upon negative cycle detection – **Done.** ✔
5. Block-based Floyd-Warshall to find out if it performs better than the row-based parallelization? – ⌛

What’s next ?

1. Implement Block-based Floyd-Warshall
2. Find out if Block Based parallelization performs better than the row-based parallelization?
3. Have an Open MP implementation
4. Use Open MP and MPI collectively to implement Floyd-Warshall and make observations

References

1. Kang, S. J., Lee, S. Y., & Lee, K. M. (2015, August 4). *Performance comparison of openmp, MPI, and mapreduce in practical problems*. Advances in Multimedia. Retrieved March 20, 2023, from https://www.hindawi.com/journals/am/2015/575687/
2. Case Study on Shortest-Path Algorithms. (n.d.). Retrieved March 20, 2023, from https://www.mcs.anl.gov/~itf/dbpp/text/node35.html
3. MPI Tutorials. Tutorials · MPI Tutorial. (n.d.). Retrieved March 24, 2023, from https://mpitutorial.com/tutorials/
4. Dr. Matthew D. Jones. *Lectures on MPI & CCR*
5. Striver, *https://takeuforward.org*
6. GeeksForGeeks, https://www.geeksforgeeks.org/detecting-negative-cycle-using-floyd-warshall/

Thank You