PARALLEL IMPLEMENTATION OF FLOYD-WARSHALL ALGORITHM

Guided by: Dr. Russ Miller (UB Distinguished Professor)
CSE 633: Parallel Algorithms
Presented By: Asmita Gautam

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

OVERVIEW:

1. Problem Statement
2. Serial Execution
3. Parallelization and Communication
4. Implementation and Parallel Algorithm
5. Performance and Speedup
6. Challenges
7. References

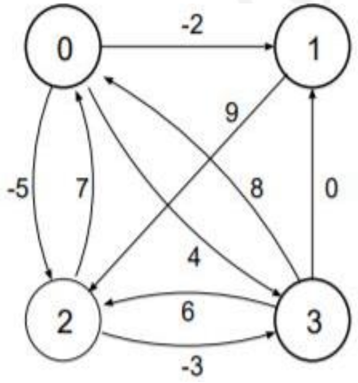
Problem Statement

Perform the parallel implementation of the Floyd-Warshall algorithm.

Floyd-Warshall Algorithm:

- It is an all pair shortest path algorithm for a directed and weighted graph.
- It basically tries to find the minimum distance between any pair of vertices in the graph.
- In this we consider every vertex as an intermediate vertex ‘k’ and find if the distance between i,j through k is smaller than the existing distance.

i.e. dist(i,j) = min(dist(i,j) , dist(i,k) + dist(k,j))



	0	1	2	3
0	0	−2	−5	4
1	∞	0	9	∞
2	7	∞	0	−3
3	8	0	6	0

Adjacency Matrix

Serial Execution

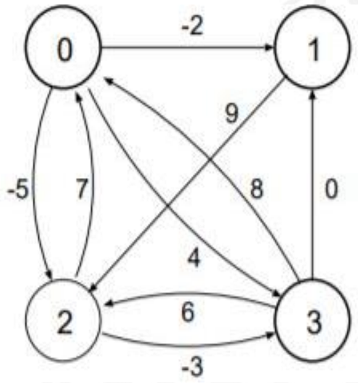
- Since the distance from a vertex to itself is going to be 0 , hence all the diagonals are set to 0 in the matrix.

- **Sequential Algorithm:**

Input: n — number of vertices
a — adjacency matrix
Output: Transformed a that contains the shortest path lengths

for $k \leftarrow$ 0 to $n-1$
  for $i \leftarrow$ 0 to $n-1$
    for $j \leftarrow$ 0 to $n-1$
      $a[i,j] \leftarrow \min(a[i,j],\ a[i,k] + a[k,j])$
    endfor
  endfor
endfor

Easy to see that the algorithm is $\Theta(n^3)$



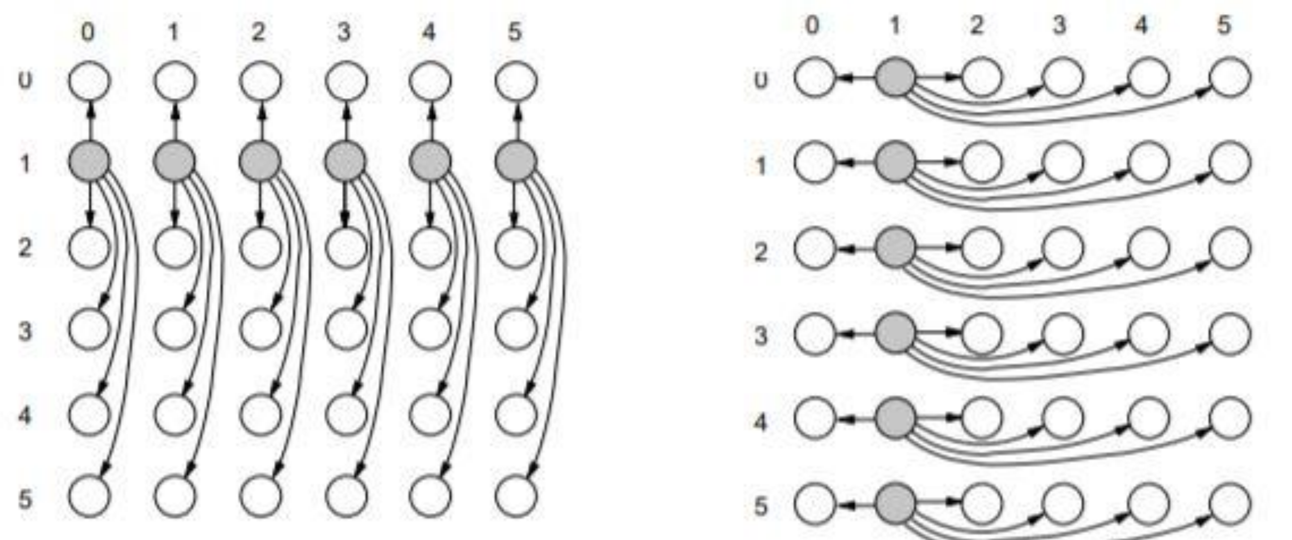
	0	1	2	3
0	0	−2	−5	4
1	∞	0	9	∞
2	7	∞	0	−3
3	8	0	6	0

Adjacency Matrix

Parallelism.. But How?

- We see that the task responsible for updating A[i, j] needs the values of A[i, k] and A[k, j]
- **For k=1**
- the task responsible for A[0, 2] needs access to A[0, 1] and A[1, 2]
- the task responsible for A[1, 2] needs access to A[1, 1] and A[1, 2]
- the task responsible for A[2, 2] needs access to A[2, 1] and A[1, 2]
- the task responsible for A[3, 2] needs access to A[3, 1] and A[1, 2]
- That means for a particular k, j, A[k][j] is needed by all the column elements of j.
- Similarly,
- the task responsible for A[0, 0] needs access to A[0, 1] and A[1, 0]
- the task responsible for A[0, 1] needs access to A[0, 1] and A[1, 1]
- the task responsible for A[0, 2] needs access to A[0, 1] and A[1, 2]
- the task responsible for A[0, 3] needs access to A[0, 1] and A[1, 3];

Communication:



(a) Each task in row 1 broadcasts to tasks in same column.

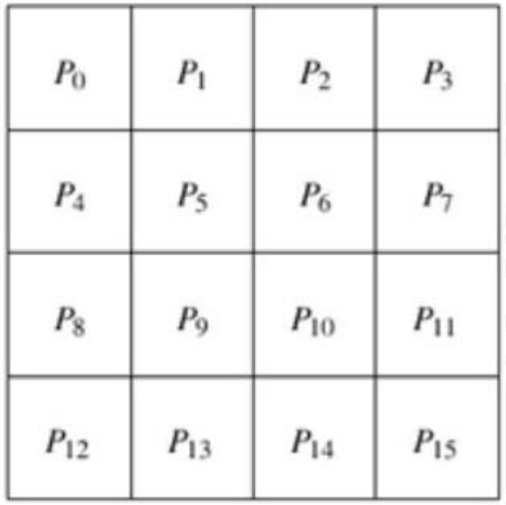
(b) Each task in column 1 broadcasts to tasks in same row.

Here for K = 1
1st column elements would do a respective row broadcast
And 1st row elements would do a respective column broadcast.

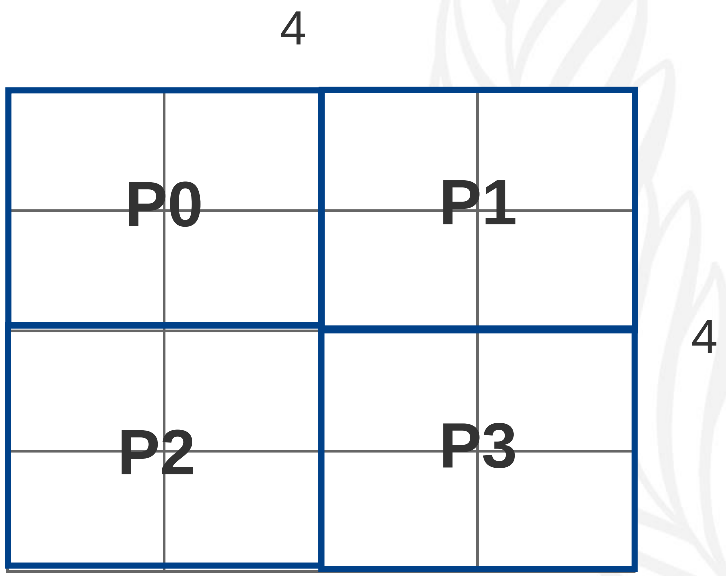
- During iteration k of the outer loop, each element of row k of A must be broadcast to every task in the same column as that element.
- Every element of column k of A must be broadcast to every task in the same row as that element.
- A broadcast is a global communication operation in which a single task sends a message to all processes in its communication group.

Implementation:

- Partitioned the matrix data using 2-D block mapping.
- The entire n x n matrix data is divided into squares of the same size and each square is assigned to a processor.
- For n x n matrix and p processors each process calculates a $n/\sqrt{p} \times n/\sqrt{p}$ part of the distance matrix.



n = 4
n x n = 16 data elements
No of processors = 4
n^2 elements are distributed amongst p processors = $n^2/p = n/\sqrt{p} \times n/\sqrt{p}$

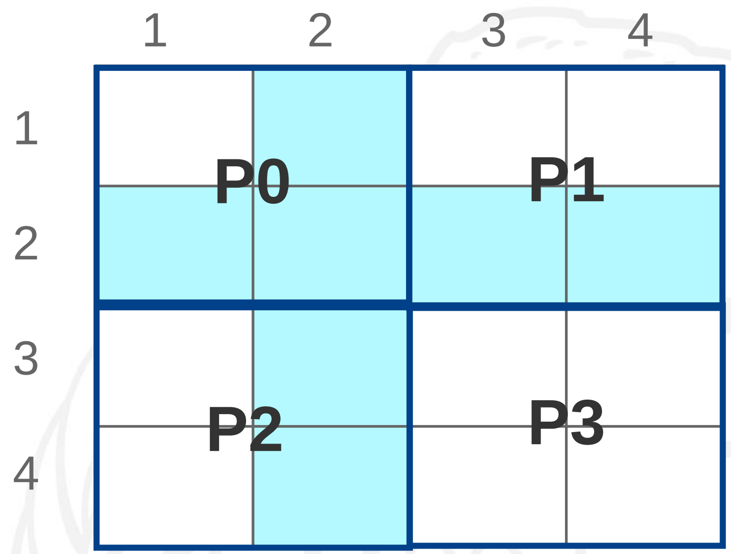


Critical condition for equal distribution of data:

$n \% \sqrt{p} = 0$

Parallel Pseudocode:

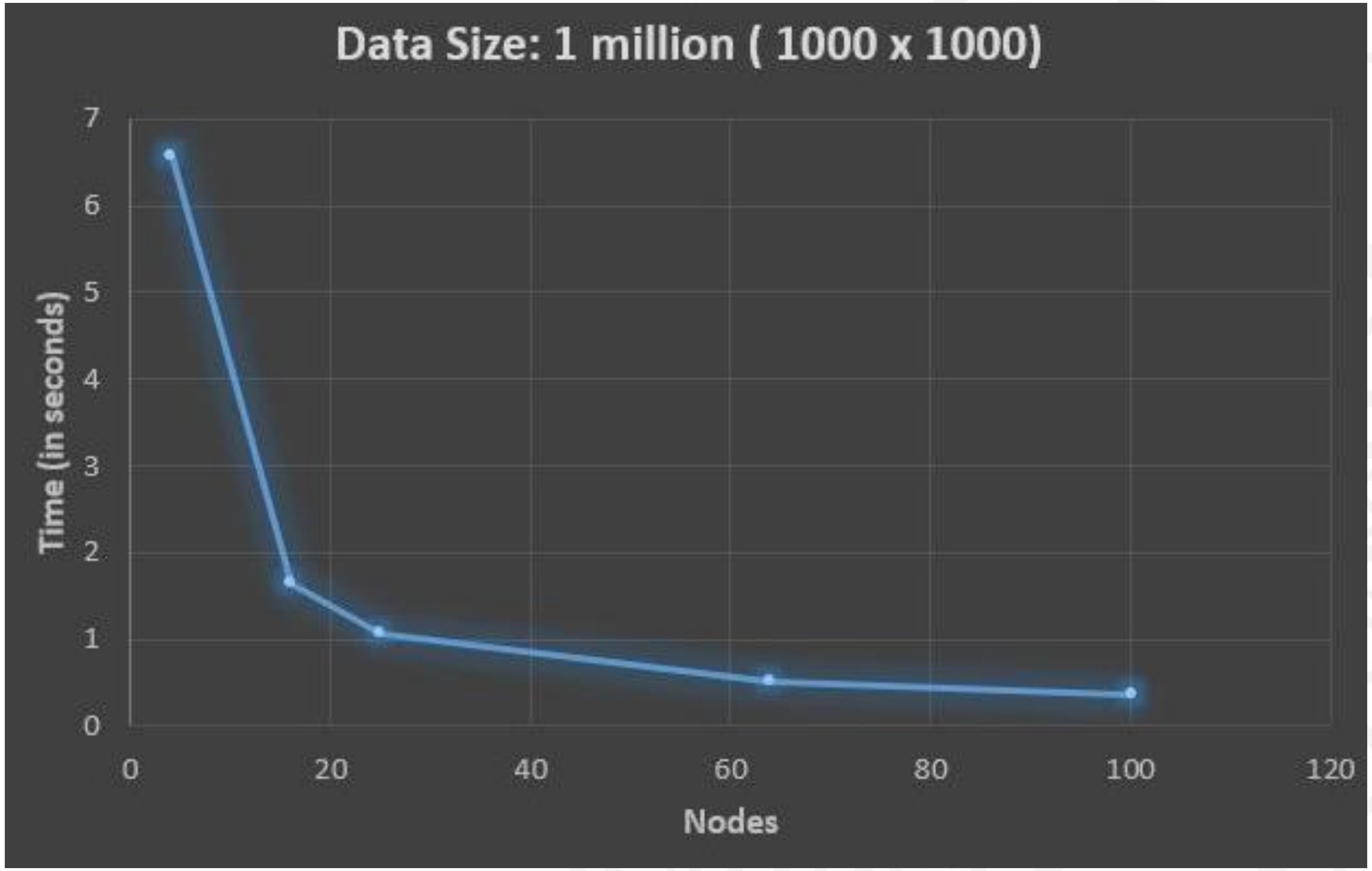
func Floyd_All_Pairs_Parallel($D^{(0)}$) {
  for k := 1 **to** n **do**{
    Each process $p_{i,j}$ that has a segment of the k-th row of $D^{(k-1)}$,
    broadcasts it to the $p_{*,j}$ processes;
    Each process $p_{i,j}$ that has a segment of the k-th column of $D^{(k-1)}$,
    broadcasts it to the $p_{i,*}$ processes;
    Each process waits to receive the needed segments;
    Each process computes its part of the $D^{(k)}$ matrix;
    }
}



Performance:

Data Size : 1 million (1000 x 1000)

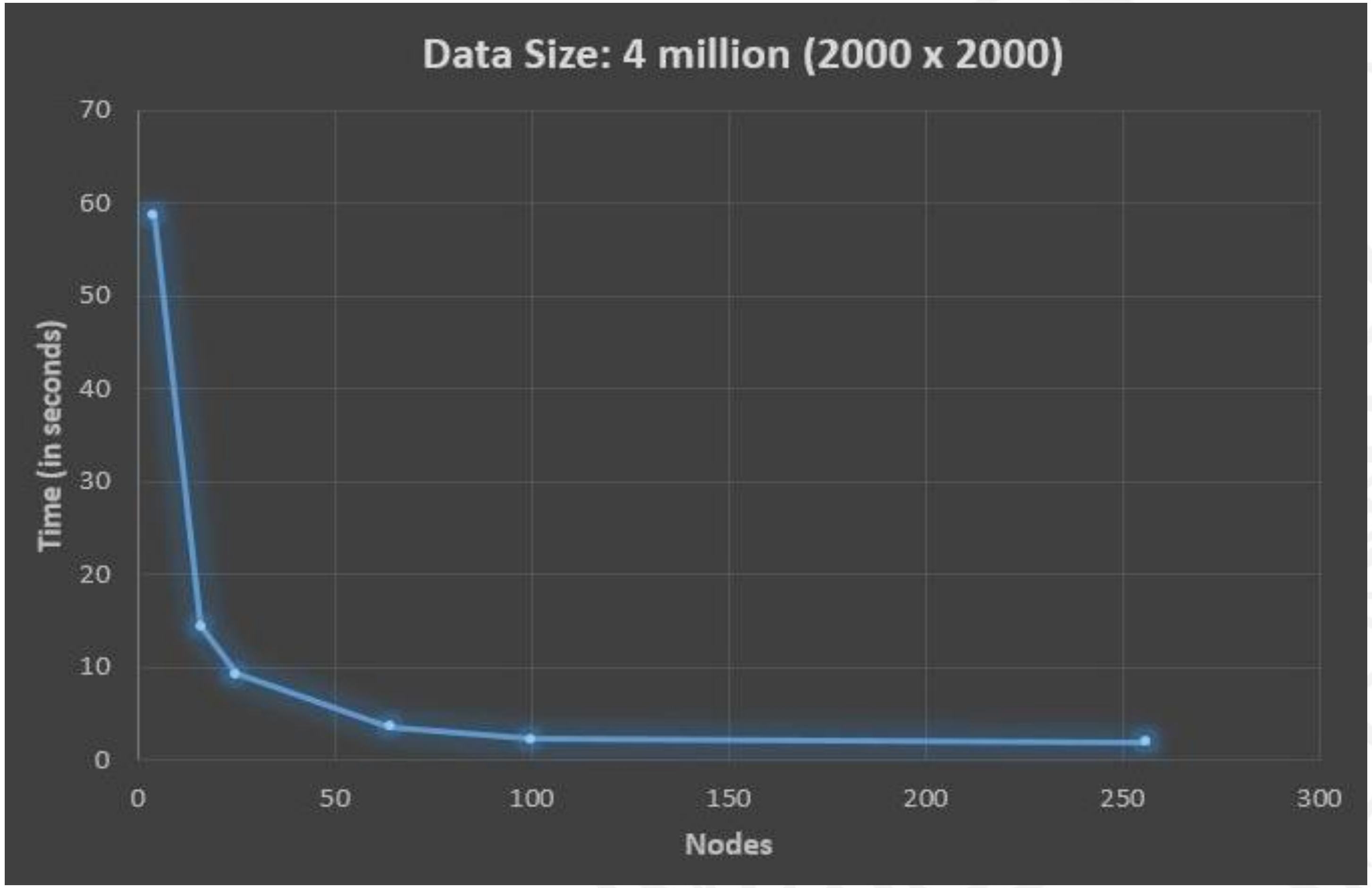
Nodes	Time (in secs)
4	6.587
16	1.646
25	1.075
64	0.519
100	0.3753



Performance:

Data Size : 4 million (2000 x 2000)

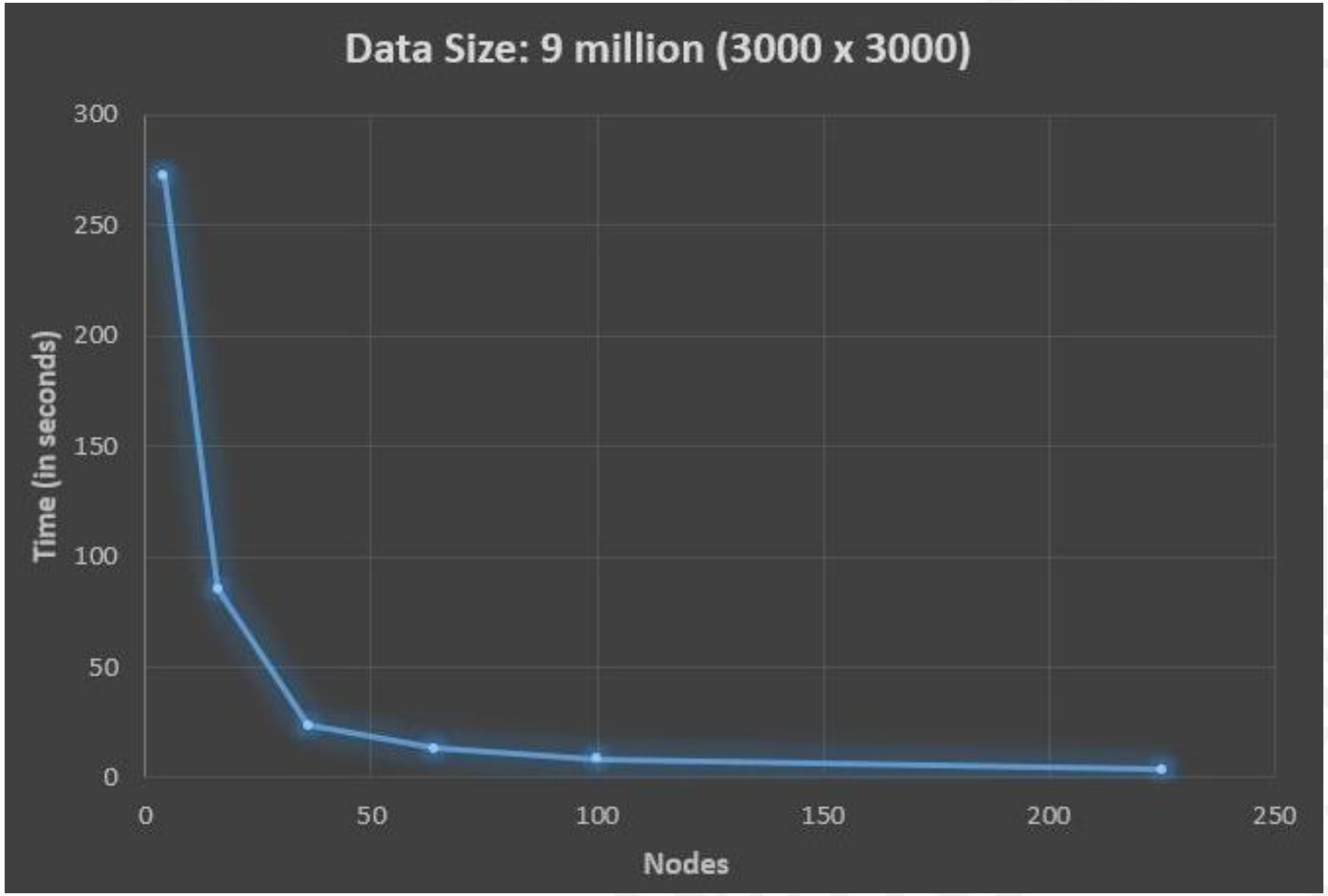
Nodes	Time (in secs)
4	58.654
16	14.452
25	9.332
64	3.627
100	2.35
256	2.01



Performance:

Data Size : 9 million (3000 x 3000)

Nodes	Time (in secs)
4	272.234
16	85.372
36	23.724
64	13.202
100	8.483
225	3.902

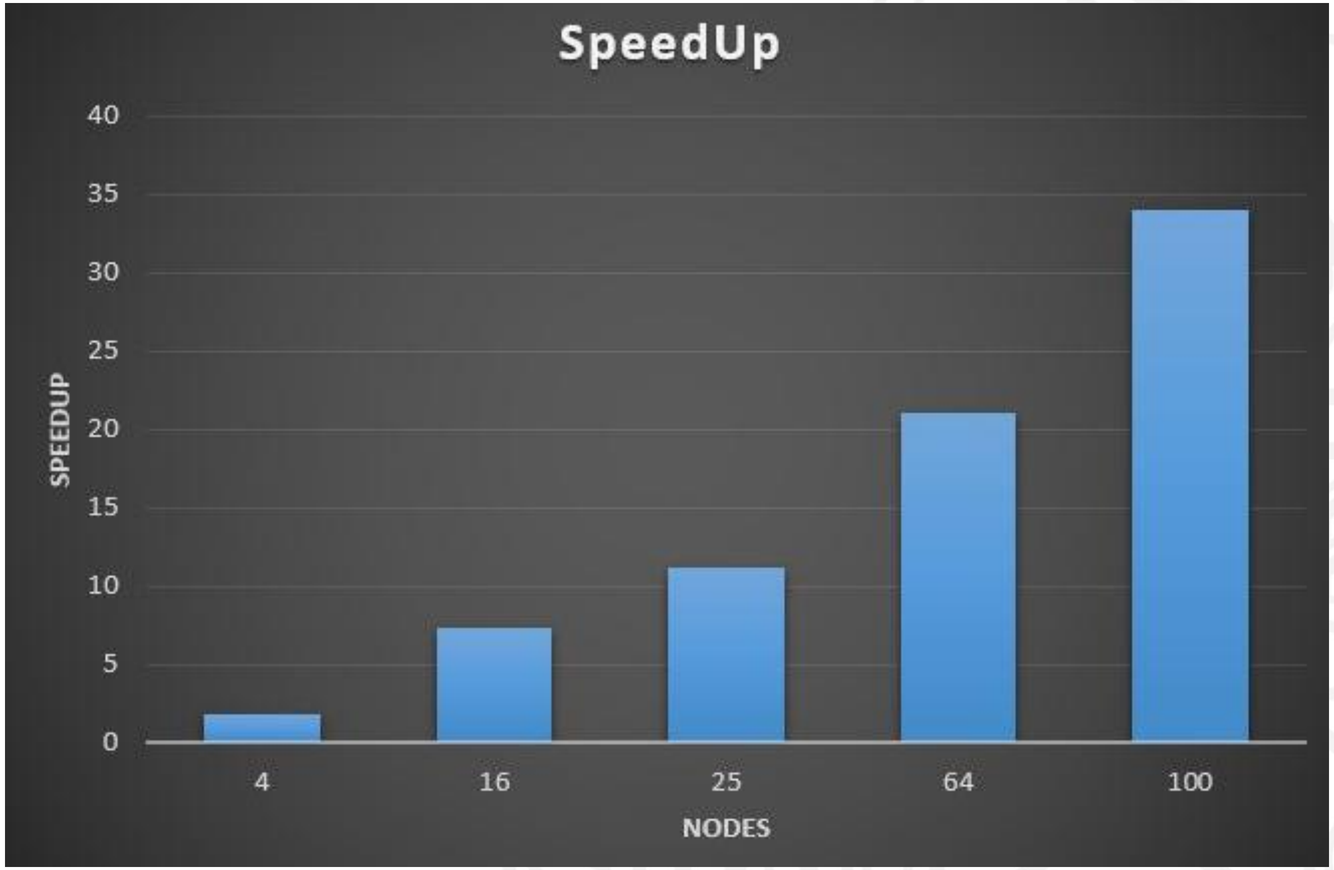


SpeedUp:

Data Size : 1 million (1000 x 1000)

Serial Execution Time: 12 seconds

Nodes	Speed-Up
4	1.821
16	7.29
25	11.162
64	21.089
100	33.99

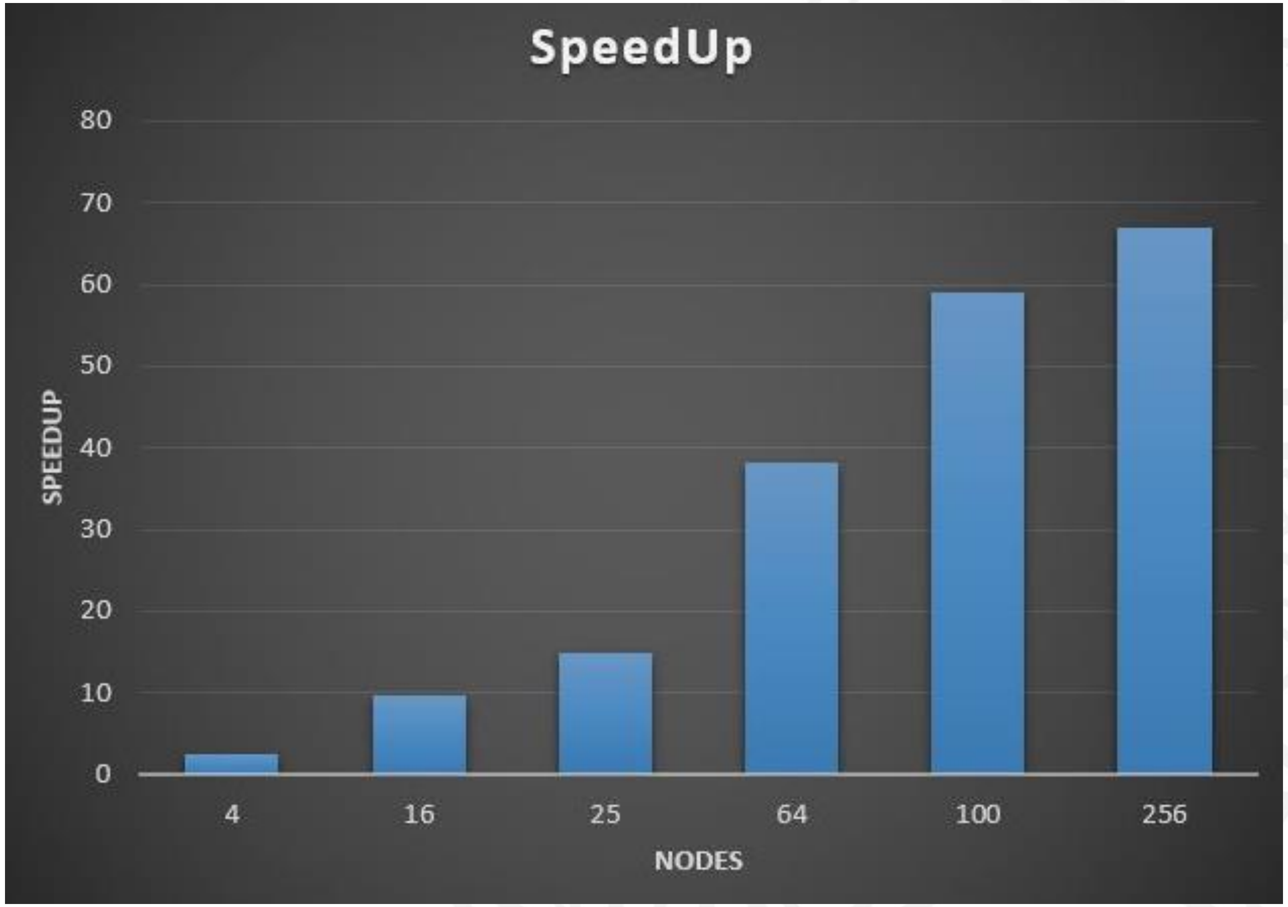


SpeedUp:

Data Size : 4 million (2000 x 2000)

Serial Execution Time: 139 seconds

Nodes	Speed-Up
4	2.369
16	9.61
25	14.89
64	38.3
100	59.14
256	67.05

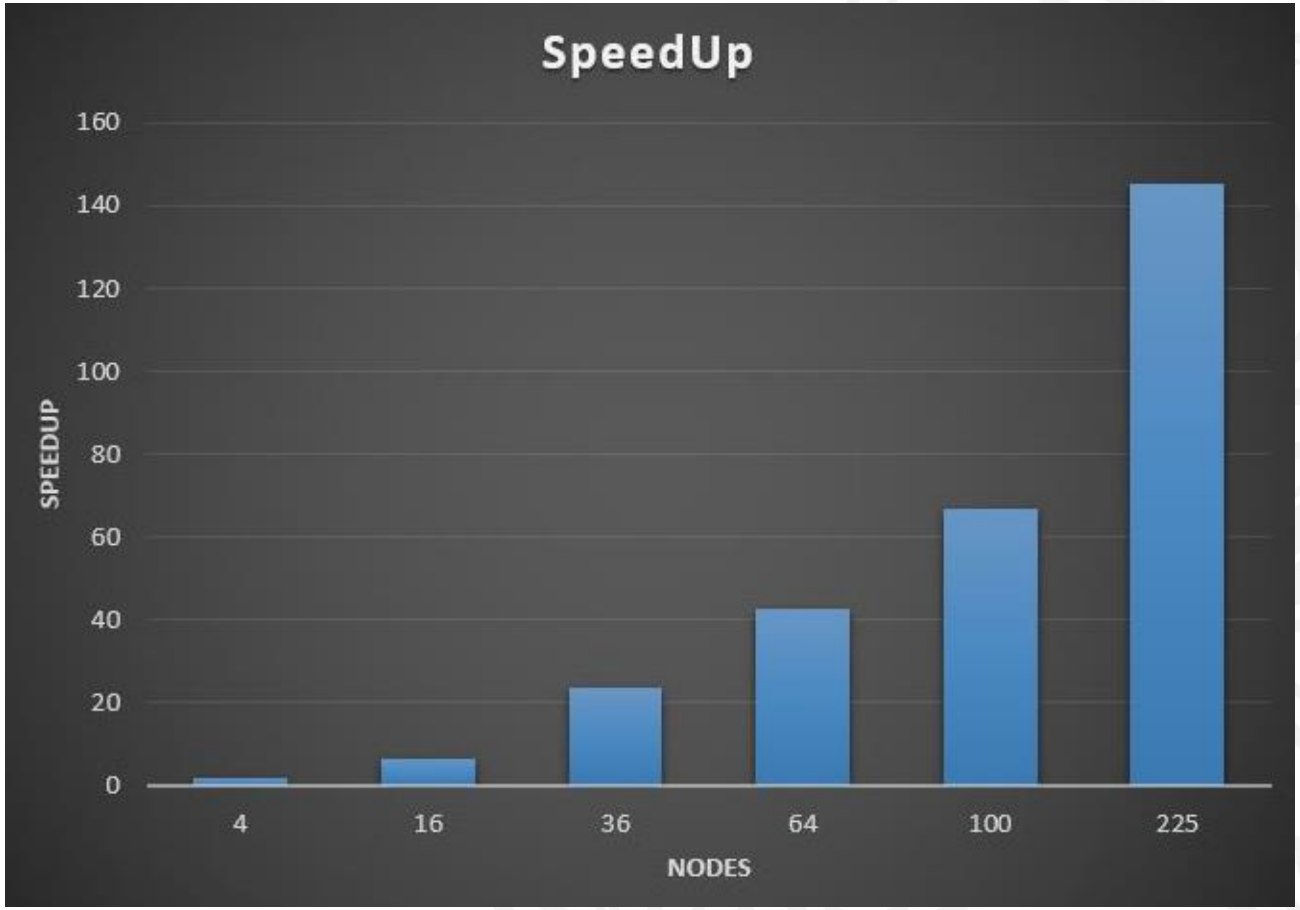


Speed-Up:

Data Size : 9 million (3000 x 3000)

Serial Execution Time: 567 seconds

Nodes	Speed-Up
4	2.082
16	6.641
36	23.899
64	42.94
100	66.839
225	145.3



Challenges:

- Distributing data amongst processors in a 2-D block fashion.
- Communication between respective row and column processors.
- Waiting time for running on 256 (or ~256) nodes.

References:

- https://en.wikipedia.org/wiki/Parallel_all-pairs_shortest_path_algorithm#Parallelization
- http://parallelcomp.uw.hu/ch10lev1sec4.html
- CCR Tutorials and handouts https://ubccr.freshdesk.com/support/solutions/articles/13000026245-tutorials-workshops-and-training-documents

MPI_Bcast("ANY QUESTIONS ???")