PARALLEL BREADTH-FIRST SEARCH USING MPI

Author – Aditya Nongmeikapam

Course – 633 Parallel Algorithms

Instructor – Russ Miller

University at Buffalo The State University of New York

CONTENTS

Breadth-First Search

- It is a graph traversing algorithm
- Starts with a given start node and traverse the graph layer wise. We then move towards the next level neighbors.
- Extra memory required, usually a queue.
    - To keep track of unexplored child nodes.

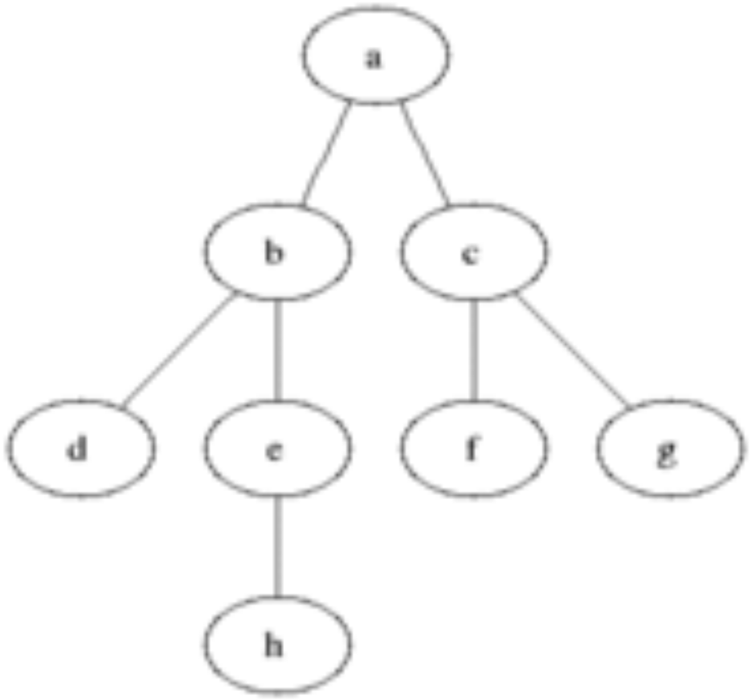


Image Source - Wikipedia

Applications of BFS

- Used to solve many graph theory problems like shortest path between two nodes for an unweighted graph.
- For computing the maximum flow in a flow network.
- In Social networking websites(e.g Linkedin), we can find the ith connection of a source person.
- Detect cycles in an undirected graph

Sequential BFS Algorithm

- Set all the vertices to not visited.
- Create a queue and add the start node or nodes.
- While the queue becomes not empty -
  - Take the first node from queue and remove it
  - If not visited already
    - Make the node visited
    - Add all the neighbors of the node into the queue.

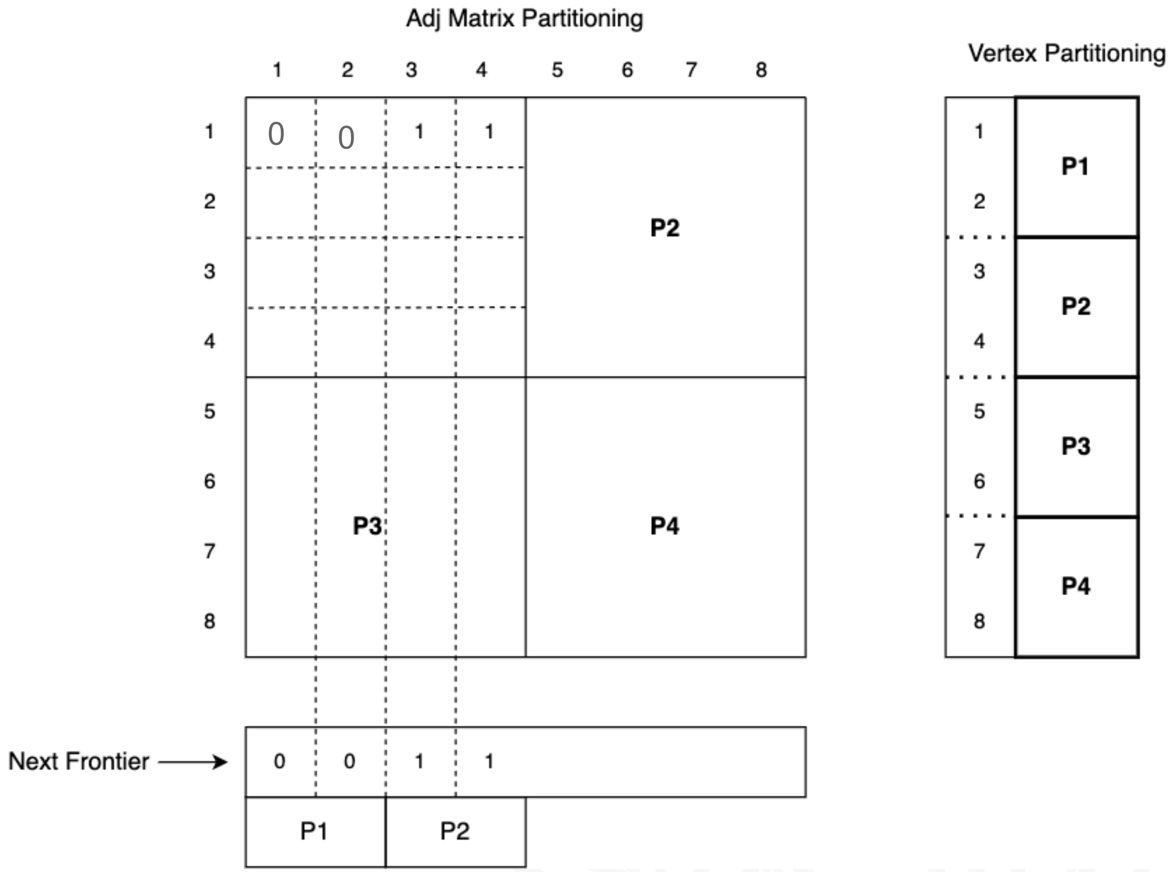
- Time Complexity will be $O(N^2)$
- Space Complexity will be $O(N^2)$
- N is total number of vertices and my implementation is based on adjacency matrix.

Parallel BFS Algorithm

- Similar algorithm as the sequential BFS.
- Instead of popping out one vertex at a time, pop out all the nodes in the same level. (These nodes are known as **frontier nodes**)
- **Level synchronous** traversal. Each the processor will take a set of frontier vertices and calculate their next frontier vertices in parallel.
- For the above step we will need to partition the adjacency matrix and the vertices and allocate them to the processors.

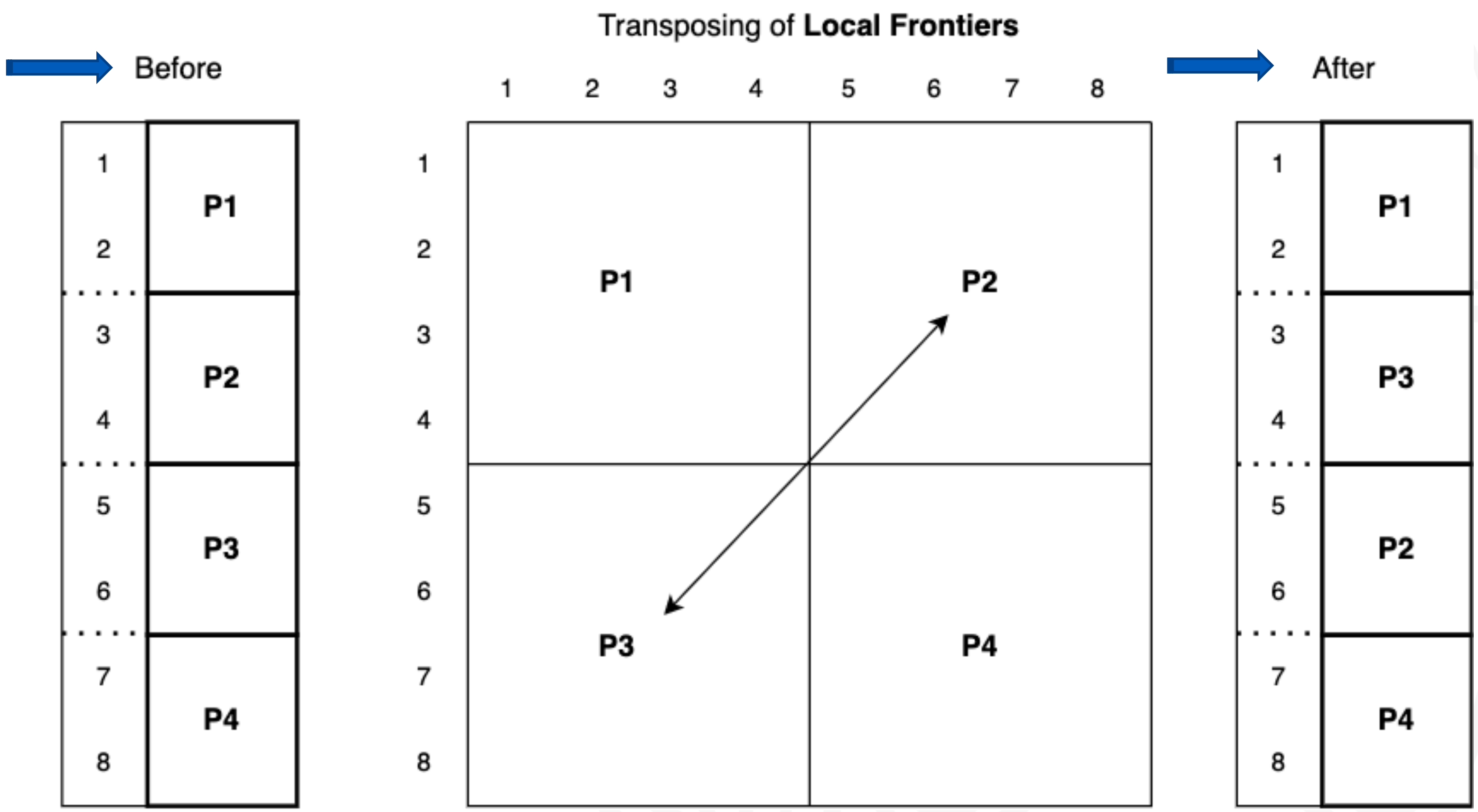
2-D Partition of Adjacency Matrix

- The adj matrix is divided into P blocks of size $\frac{N}{\sqrt{\{P\}}}$ X $\frac{N}{\sqrt{\{P\}}}$
- Vertex are partitioned into N/P size groups.
- N – Number of Vertices
- P – Number of Processors (In my case it is always a perfect square)



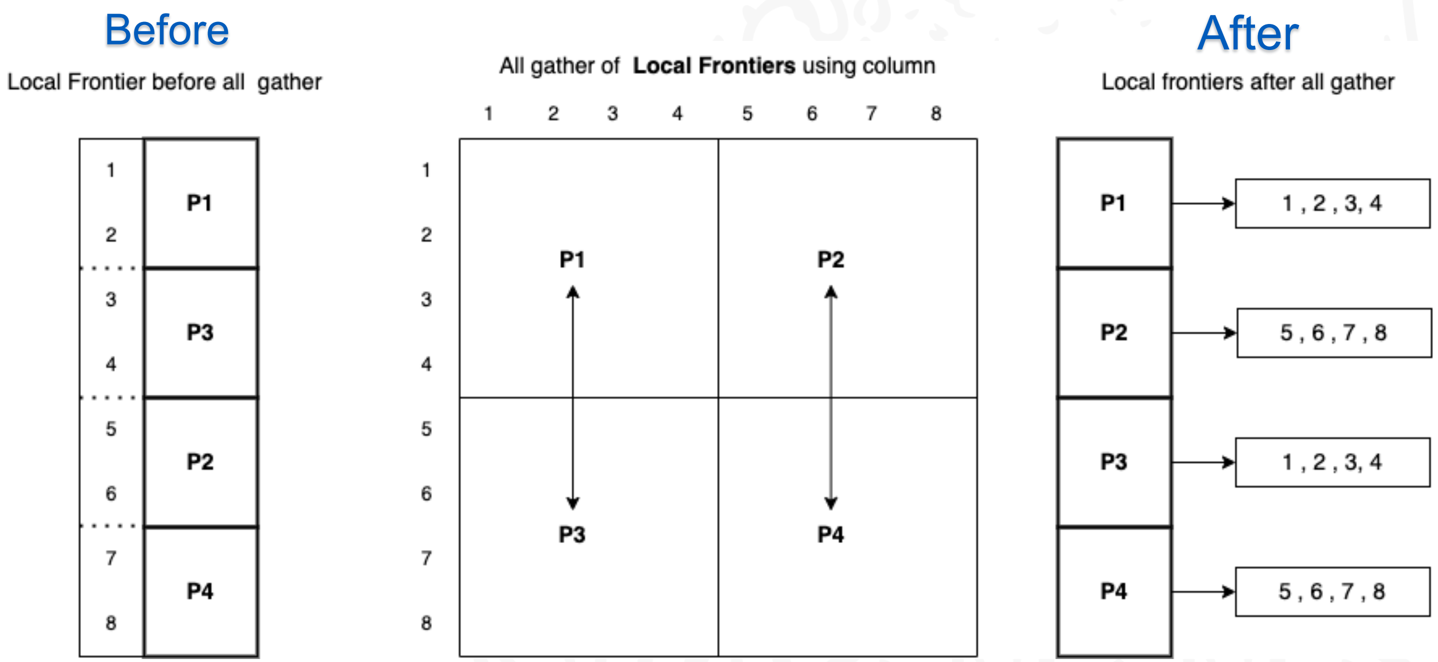
Steps of Parallel BFS Algorithm

- Do a transpose of the frontier vector between the processors.
- After this all the columns processors will have matching frontier with their local adjacency matrix.



Steps of Parallel BFS Algorithm

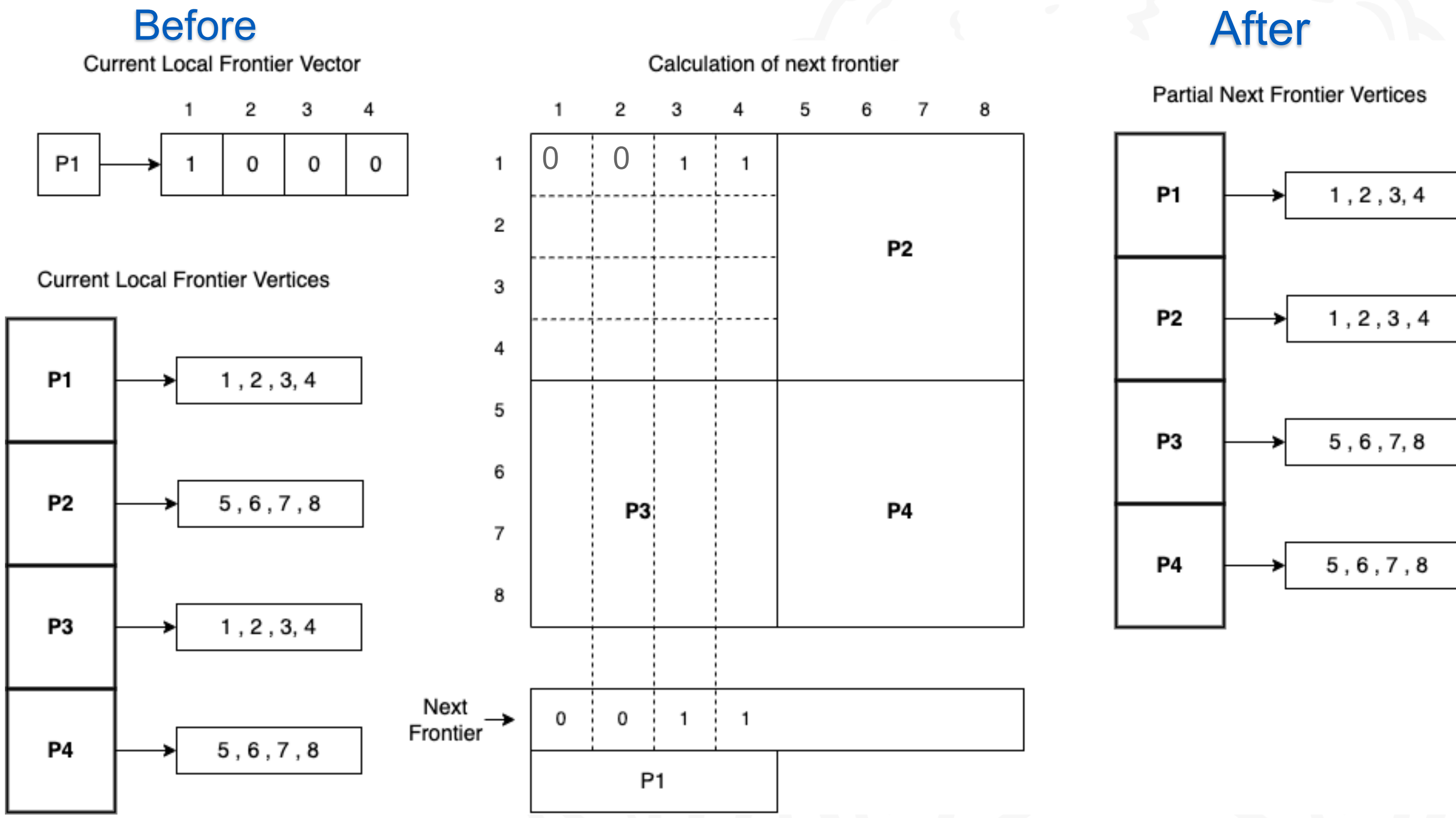
- We then do a column wise all gather for the frontier vertices.
- This will broadcast the required frontier vertices for each column.



Source Vertex = 1

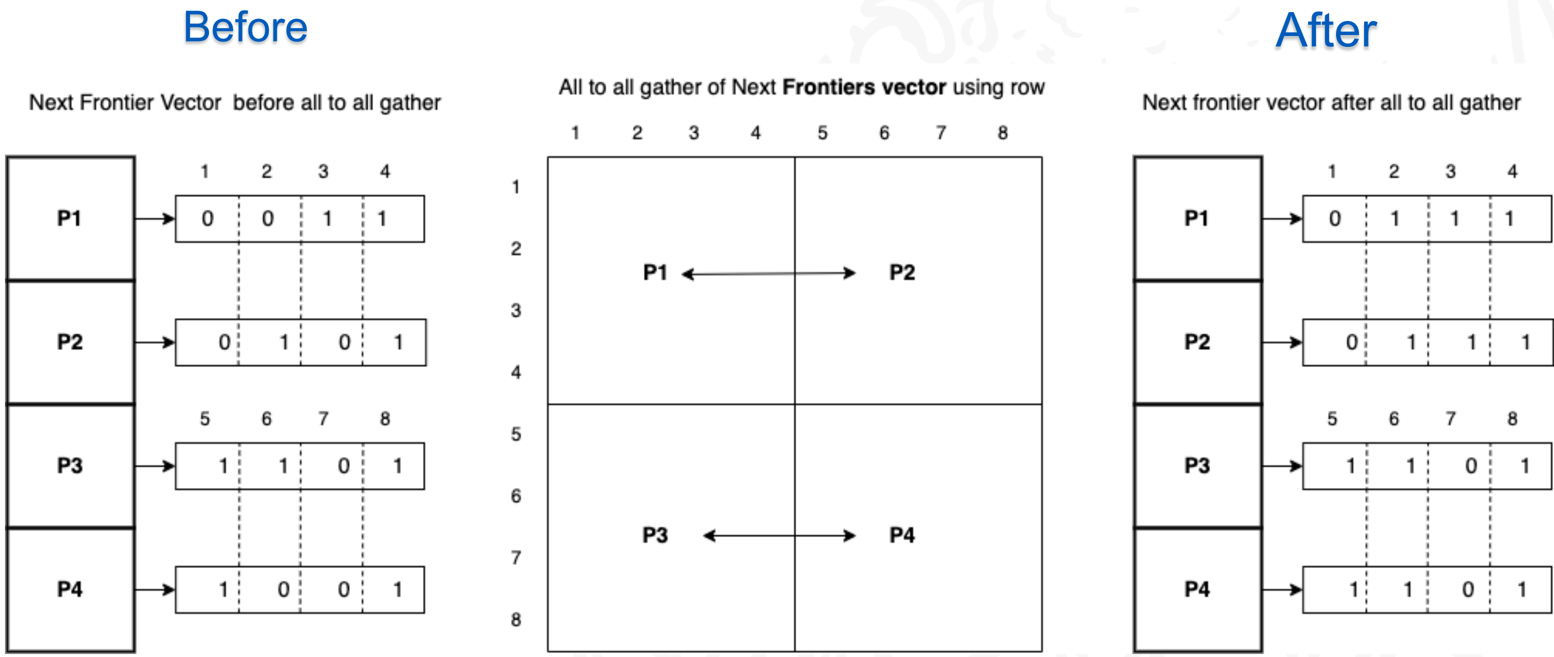
Steps of Parallel BFS Algorithm

- Calculation of next frontier vertices is based on the current frontier vector that the processor has.
- Using the local adj matrix the next frontier vector is calculated
- Note that each processor row now has the full information of the next frontier vertices.



Steps of Parallel BFS Algorithm

- Now we do a all to all gather row wise so that all the next frontier vectors are merged. (**union**)
- All the processors now know if they have any frontier element (Next frontier now becomes current local frontier) that they own.
- We mark the node as visited and store its parent node.

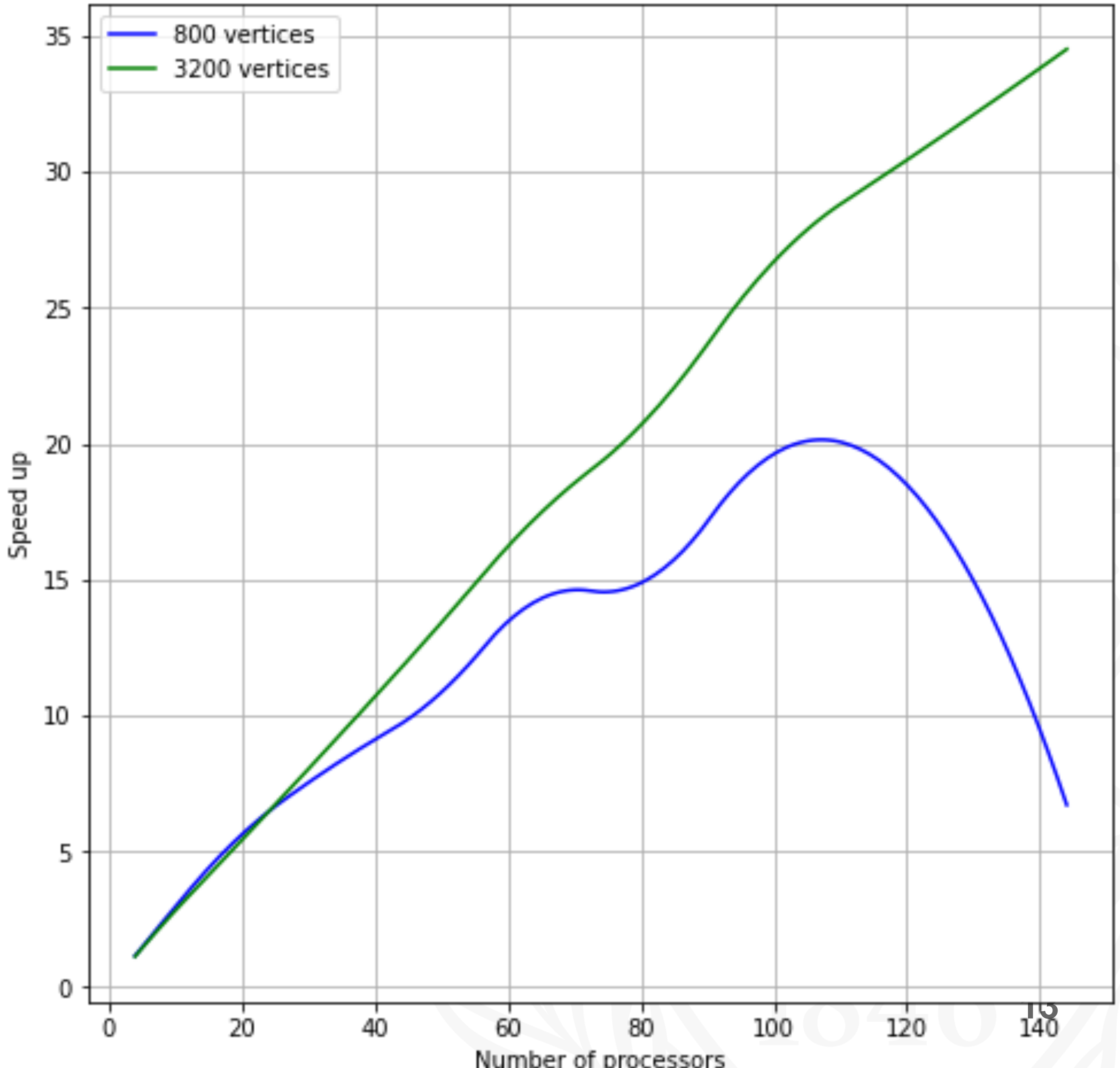


Steps of Parallel BFS Algorithm

- We do a row wise all gather and then column wise all gather to broadcast the local frontiers globally.
- We continue the process till there is no vertices left in the global frontier vertices.
- Note- The communication cost here is O($\sqrt{\{P\}}$)

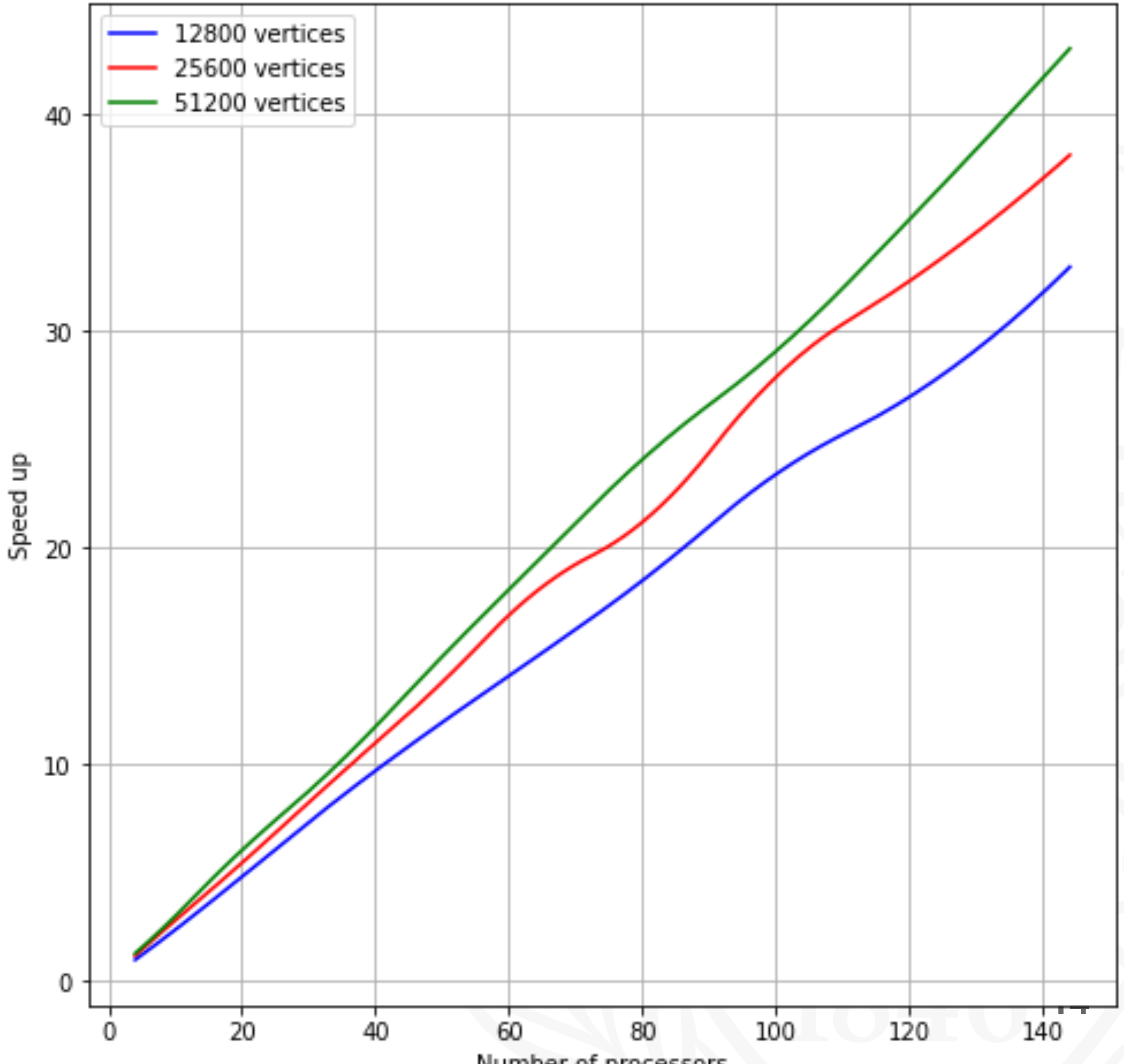
Results

- For small number of processors the graph is linear but as the number of processor increases the speed up goes down.
- But due to parallel communication overhead, we get a point ("Sweet spot") from where the speed up starts decreasing with increasing processors.

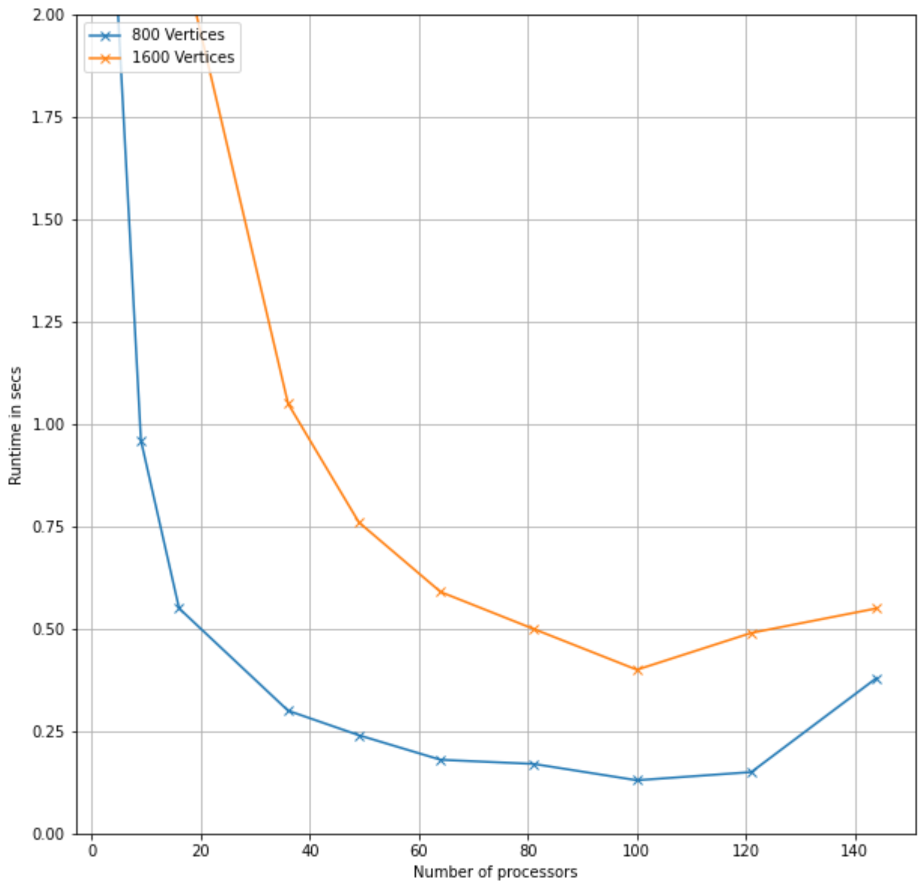


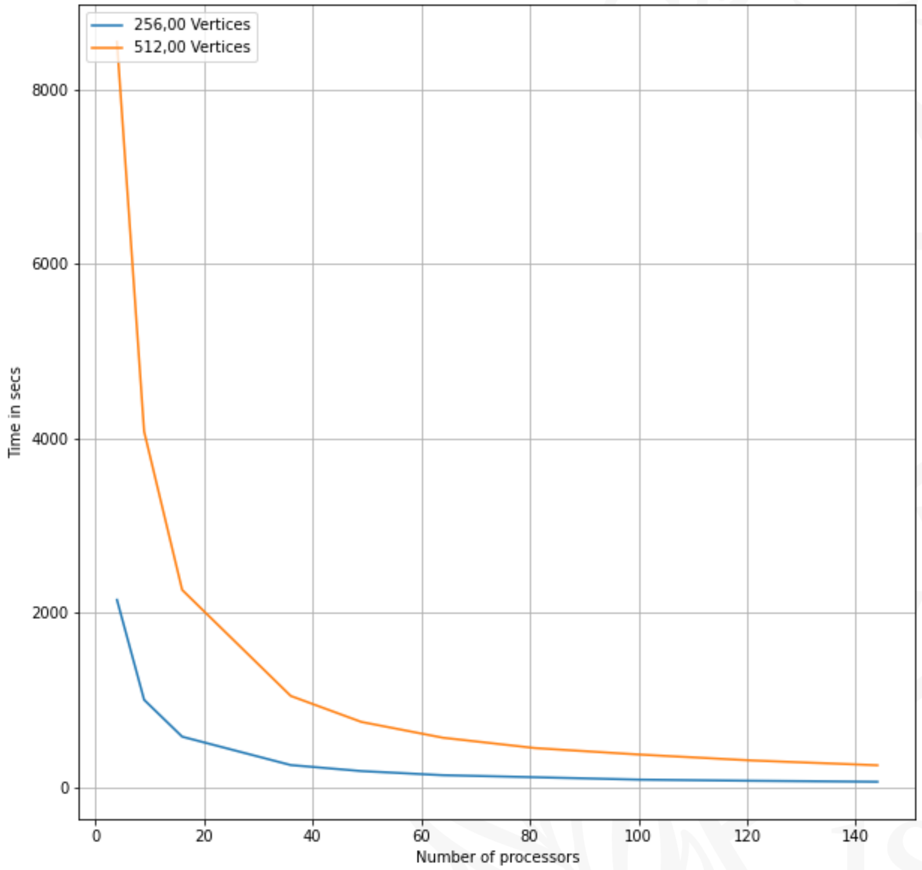
Results

- For large graphs (i.e size of adjacency matrix > 10^8). The speed up remains keeps increasing linearly with increasing processors.
- As we increase the size of our problem input size, putting more processors makes more sense as it leads to more speed up.



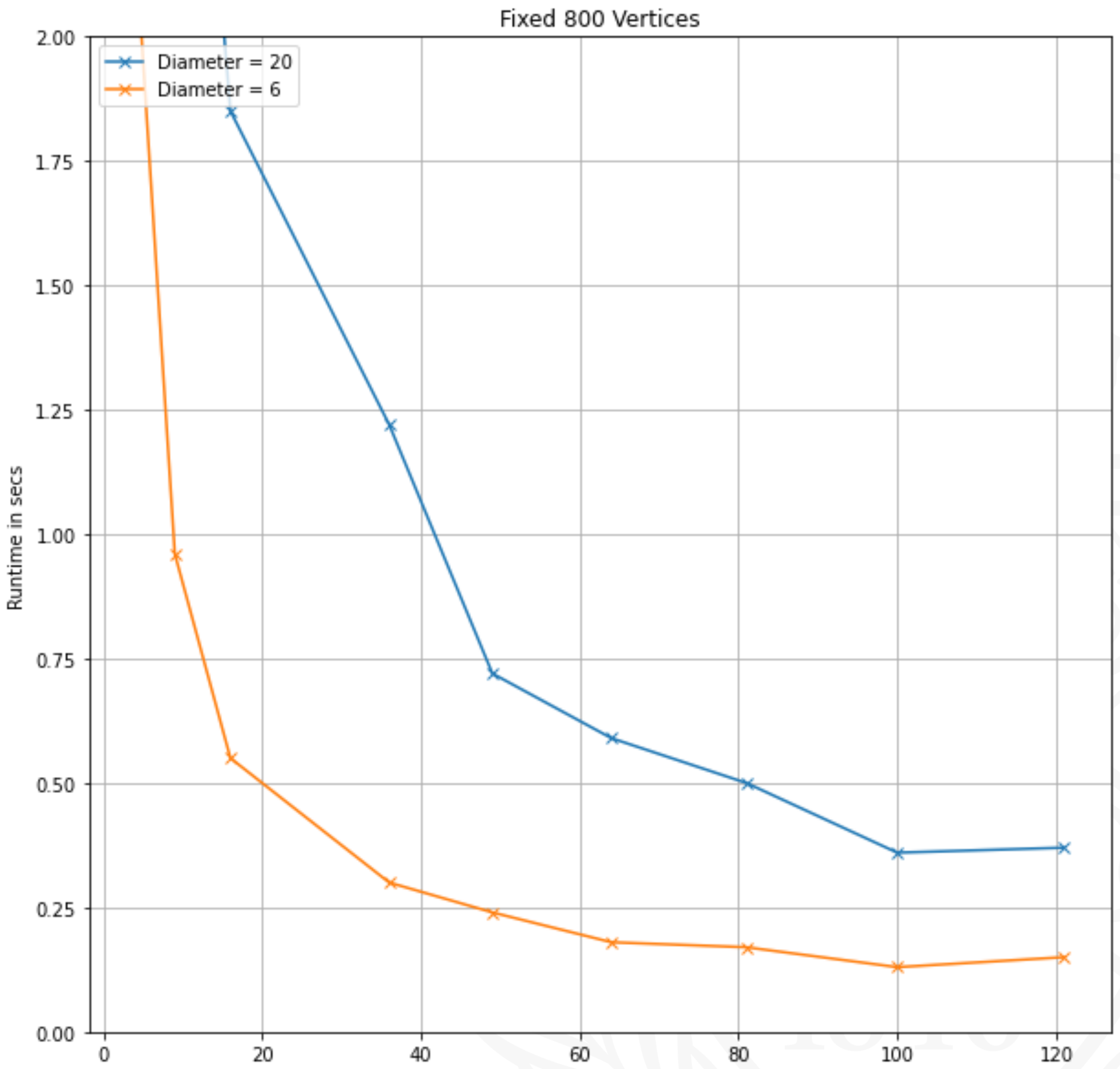
Execution Time Vs Processor





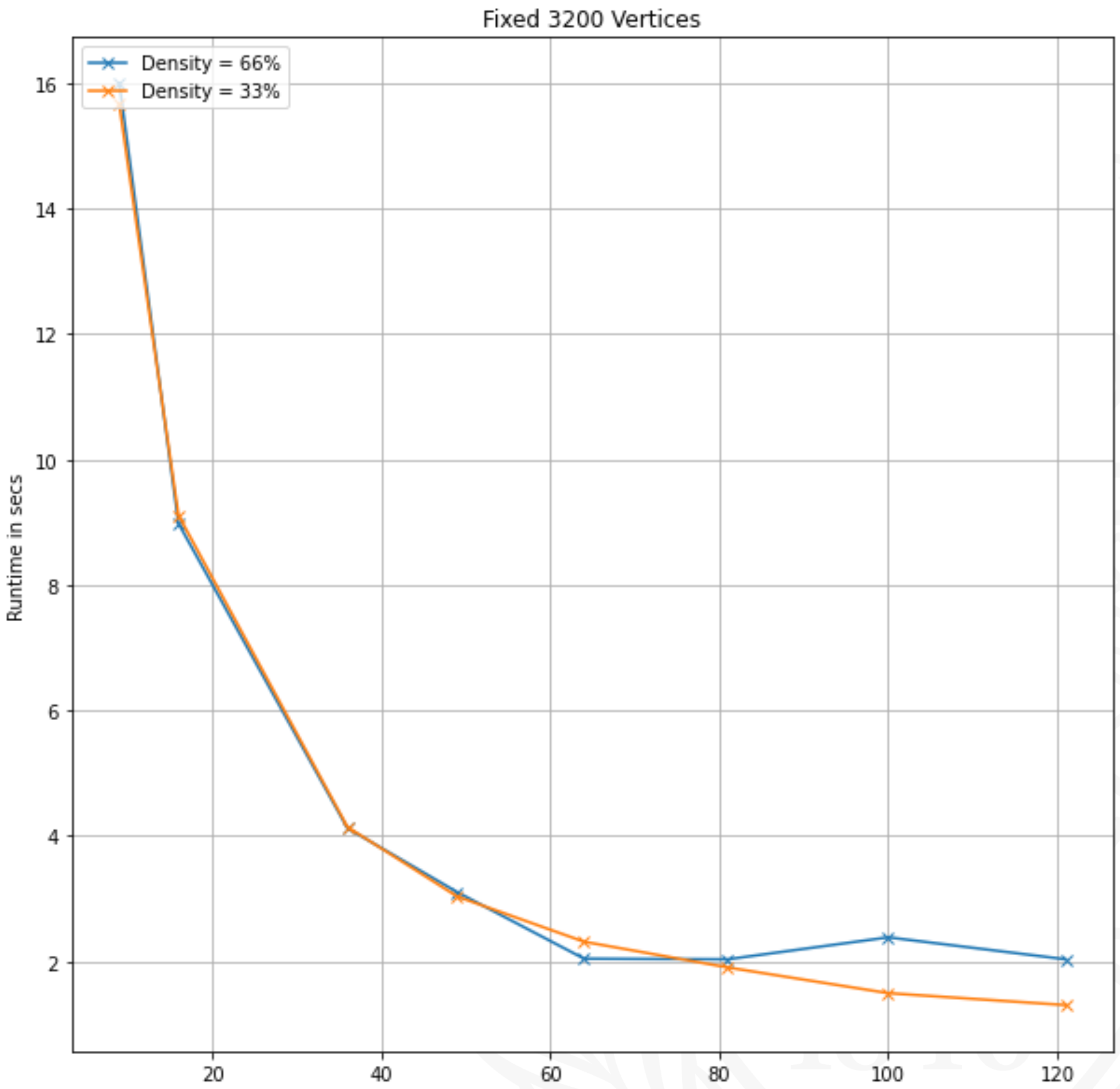
How Diameter affects the runtime

- The runtime of the algorithm depends on the diameter of the graph.
- As we increase the diameter of 6 to 20. The runtime is also increased by a factor of around 3.3x.
- This is in fact expected as the number of rounds of the of the algorithm is also increased by the same factor
- PRAM asymptotic time complexity for a level-synchronous parallel BFS is O(D) where D is the diameter of the graph.



How Density affects runtime

- The runtime of the algorithm is not depending on the density of the graph.
- As we double the density of the graph from 33% to 66%, there is no significant change in the runtime.
- This is because we use a adjacency matrix based approach and do not take advantage of the sparseness of the matrix or the frontier vectors.



Average Execution times

Processors →

Vertices ↓

	1 (Seq)	4	9	16	36	49	64	81	100	121	144*
800	2.55	2.2	0.96	0.55	0.3	0.24	0.18	0.17	0.13	0.15	0.38
1600	10.05	8.69	3.91	2.18	1.05	0.76	0.59	0.5	0.4	0.49	0.55
3200	39.77	35	15.66	9.11	4.14	3.03	2.31	1.9	1.49	1.3	1.15
6400	132	142	64.19	36.88	16.39	11.75	9.03	7.41	5.82	5.03	4.16
12800	545.54	547	254.53	142.17	62.02	46.46	36.47	29.1	23.3	20.07	16.54
25600	2566.99	2152.41	1006.26	584.11	258.76	189.89	142.79	119.56	92	78.87	67.29
51200	11052.59	8545.06	4080.1	2265.75	1050.67	753	572.11	453.41	379.88	311.34	256.66

* - Executed in 142 nodes with 1 processor and 1 node with 2 processor

Conclusion & Challenges

- We see that for smaller input sizes after 100 processors the speedup is decreasing.
- We have access to only 143 nodes in the HPC cluster
- Was able to run with 10^5 Vertices (6 billion edges) (320 GB of memory used) but had problems running 10^6 vertices.

Benefits of using HPC

Processor	Vertices	Execution Time
128x2 = 256	10^5	0.16 hrs
1 –> sequential	10^5	12.3 hrs (estimated)

Future Work

- Optimisation of the algorithm using sparse representation of the matrix .
- Use space efficient bitmaps for storing the data/vector.
- Inter-processor collective communication optimisation.

References

- https://people.eecs.berkeley.edu/~aydin/sc11_bfs.pdf [Parallel Breadth-First Search on Distributed Memory Systems]
- https://ieeexplore.ieee.org/document/1559977 [A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L]
- https://en.wikipedia.org/wiki/Parallel_breadth-first_search
- https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/
- https://en.wikipedia.org/wiki/Collective_operation
- https://ubccr.freshdesk.com/support/home
- https://slurm.schedmd.com/

Thank You