PARALLEL BREADTH-FIRST SEARCH USING MPI

CSE 708 Programming Massively Parallel Systems

Presenter: Sandeep Kunusoth (50465621)

Instructor: Dr. Russ Miller

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

Contents

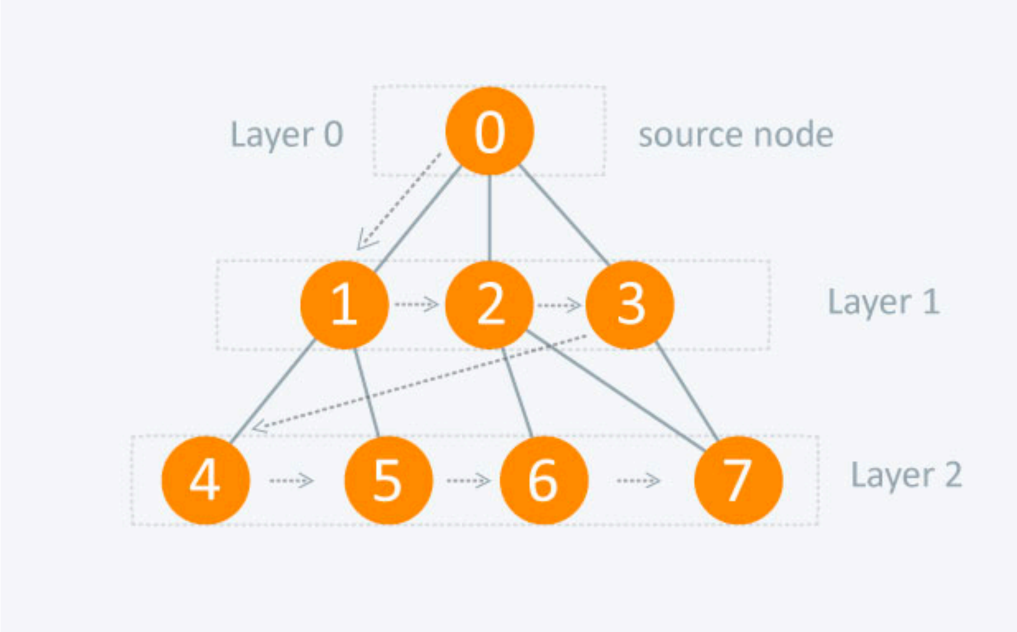
- Breadth First Search
- Applications of BFS
- Serial Implementation of BFS - Dry Run
- Issues with serial implementation and Need for Parallelization
- Parallel Implementation of BFS - Dry Run
- Advantages of Parallel over Serial Implementation
- Results
- References

Breadth-First Search

- It is a graph traversal algorithm.

- Starts with a given start node and traverse the graph layer wise. We then move towards the next level neighbors.

- Drawback: Extra memory required. Generally Queue, to keep track of unexplored nodes.



source: hackerearth

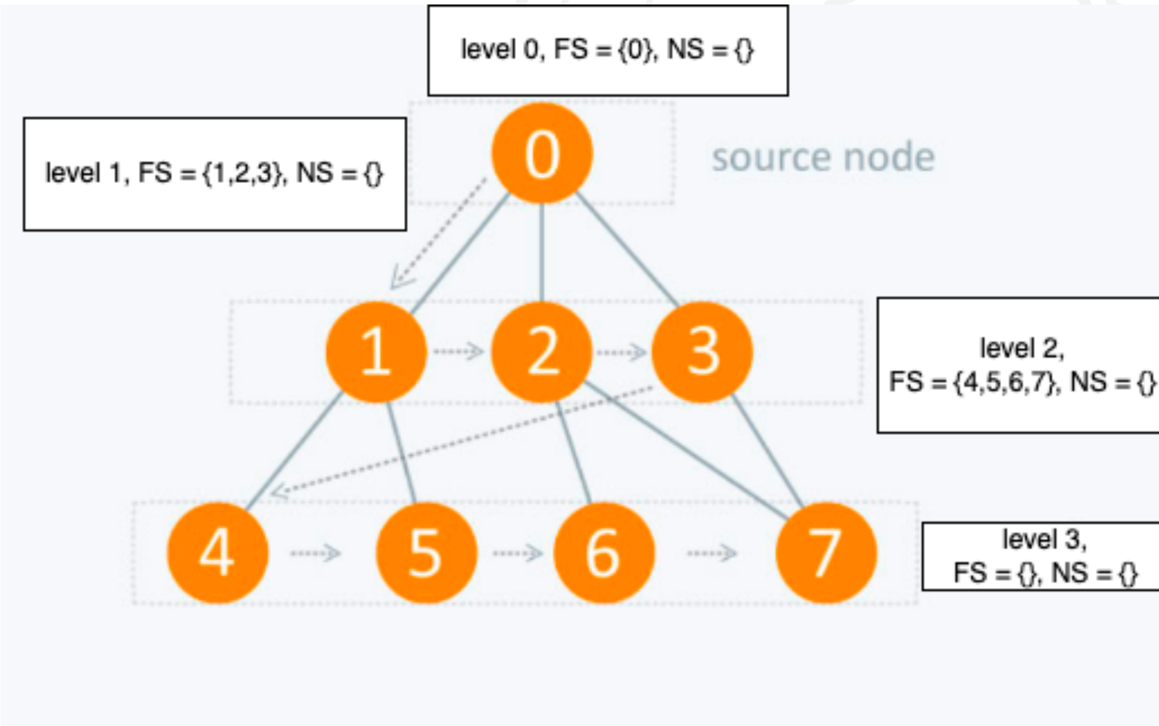
Applications of BFS:

- BFS can be used to find shortest path between 2 geographical locations on map as routing algorithms for navigation systems.
- BFS is used by search engines to index and crawl the web pages.
- Peer to Peer Networks like BitTorrent.
- BFS can be used in AI applications such as path finding, recommender systems.
- BFS can be used in game theory to find next best move in games like Chess etc.

https://www.ijcsma.com/articles/graph-traversals-and-its-applications-in-graph-theory.pdf

Serial BFS implementation

```
define serial_bfs(graph (V,E), source s):
    for all v in V do
        distance[v] = -1;
    distance[s] = 0; level = 0; FS = {s}; NS = {};
    while FS is not empty do
        level = level + 1;
        for u in FS do
            for each neighbour v of u do
                if distance[v] = -1 then
                    push(v, NS);
                    distance[v] = level;
        FS = NS, NS = {};
```



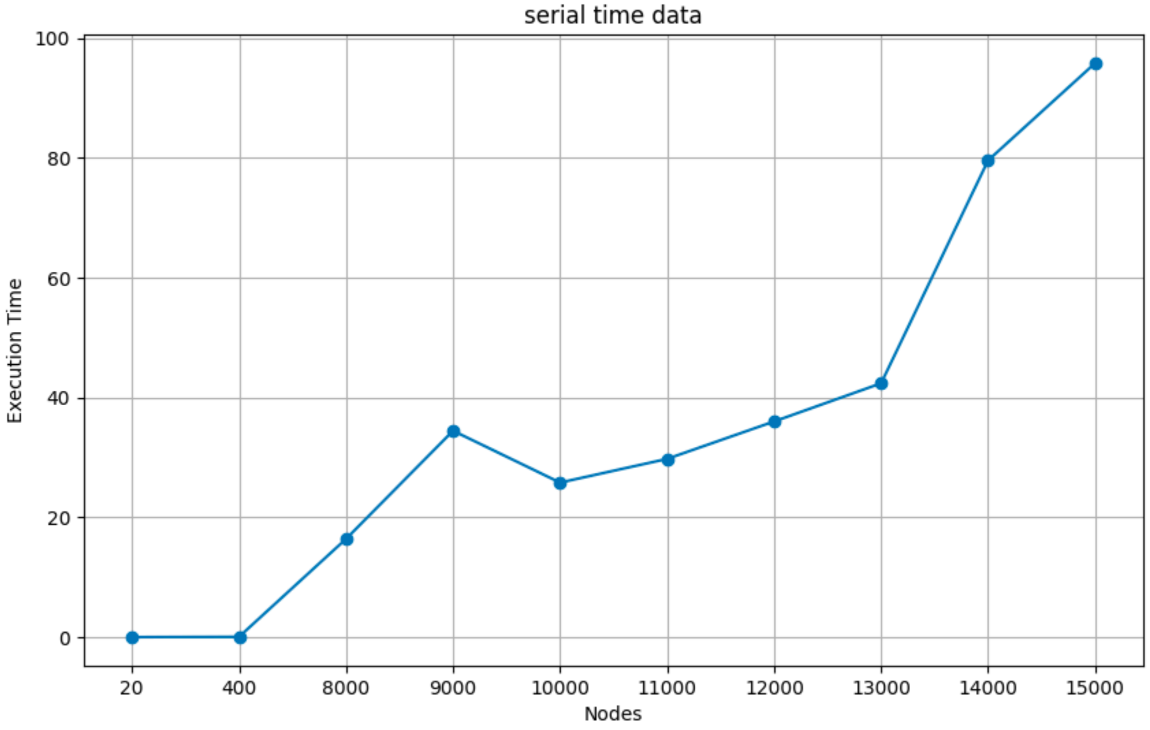
https://en.wikipedia.org/wiki/Parallel_breadth-first_search

order of traversal:
0 -> 1 -> 2 -> 3 -> 4-> 5 -> 6 -> 7

Issues with serial implementation and Need for Parallelization

```
define serial_bfs(graph (V,E), source s):
    for all v in V do
        distance[v] = -1;
    distance[s] = 0; level = 0; FS = {s}; NS = {
    while FS is not empty do
        level = level + 1;
        for u in FS do
            for each neighbour v of u do
                if distance[v] = -1 then
                    push(v, NS);
                    distance[v] = level;
        FS = NS, NS = {};
```

wiki/Parallel_breadth-first_search



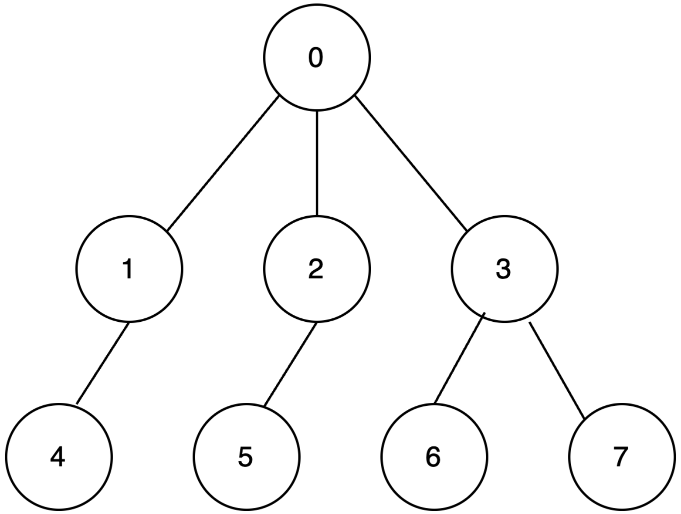
Parallel BFS implementation

```
define 1_D_distributed_BFS(graph(V,E), source s, rank):
    for all v in V do
        distance[v] = -1;
    level = 0; FS = {}; NS = {};
    if find_owner(s) = rank then
        FS = {s};  distance[s] = 0;
    global_FS_is_not_empty = true
    while global_FS_is_not_empty do
        level = level + 1;
        FS = {set of local vertices}
        for each u in FS do
            for each neighbor v of u do
                j = find_owner(v)
                push(v, send_buffer[j])
        // all-to-all communication
        for 0 <= j < p do
            if j != rank then
                send send_buffer[j] to j
                recv recv_buffer[j] from j
        NS = {neighbors of vertices in FS including non local}
        for each u in NS and distance[u] == -1 do
            distance[u] = level
            push(u, FS)
        NS = {};
        global_FS_is_not_empty = AllReduce(FS.size(), SUM) == 0
```

Modifications:

- Similar to serial BFS implementation, but instead of checking the queue of vertices sequentially, we implement this in parallel across all the vertices at the same level.
- A neighbor vertex from one processor may belong to other processor. Hence each processor needs to communicate with all others.
- The algorithm ends when global size of frontier across all processors is zero.

Dry Run



8 Vertices divided between 4 processors
Processor 0: {0, 1}
Processor 1: {2, 3}
Processor 2: {4, 5}
Processor 3: {6, 7}

Processor 0:
Iteration1:
FS = {0}
NS = {1}
All visited = {1,2,3}
Iteration 2:
FS = {1}
NS = {}
All visited={4}

Processor 1:
Iteration1:
FS = {}
NS = {}
Iteration 2:
FS = {2,3}
NS = {}
All visited={5,6,7}

Processor 2:
Iteration1:
FS = {}
NS = {}
Iteration 2:
FS = {}
NS = {}
Iteration 3:
FS = {4,5}
NS = {}

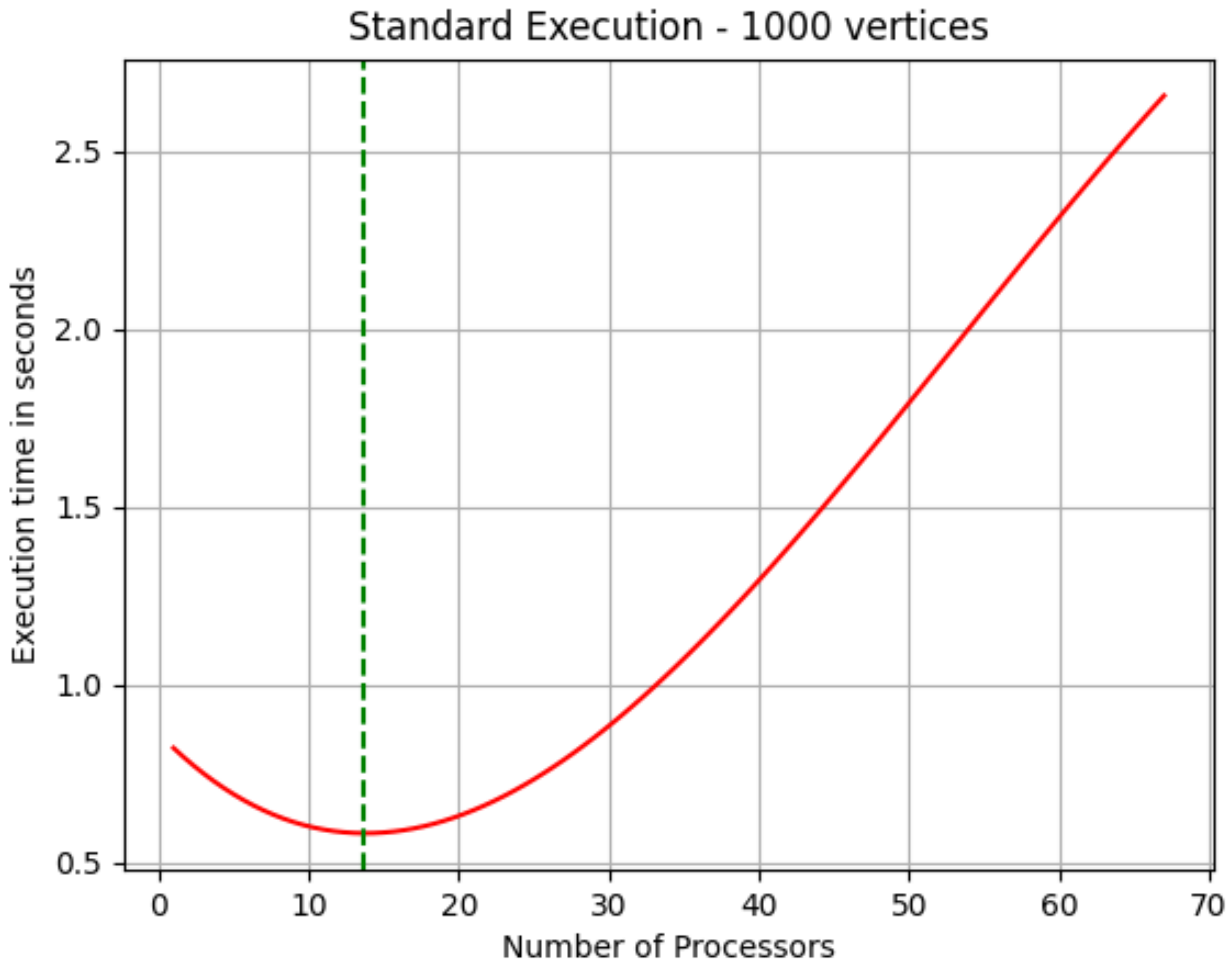
Processor 3:
Iteration1:
FS = {}
NS = {}
Iteration 2:
FS = {}
NS = {}
Iteration 3:
FS = {6,7}
NS = {}

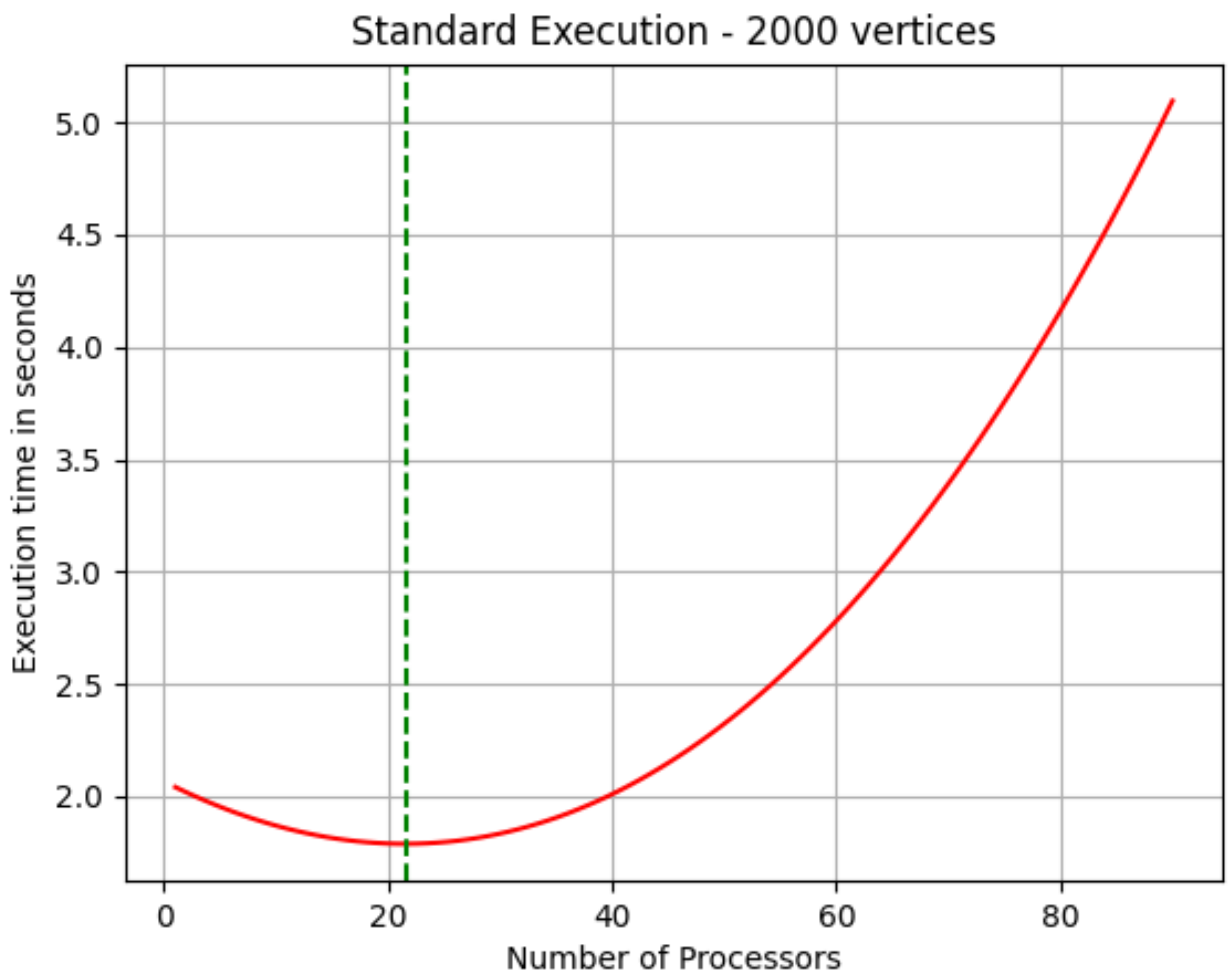
Advantages of Parallel over Serial Implementation

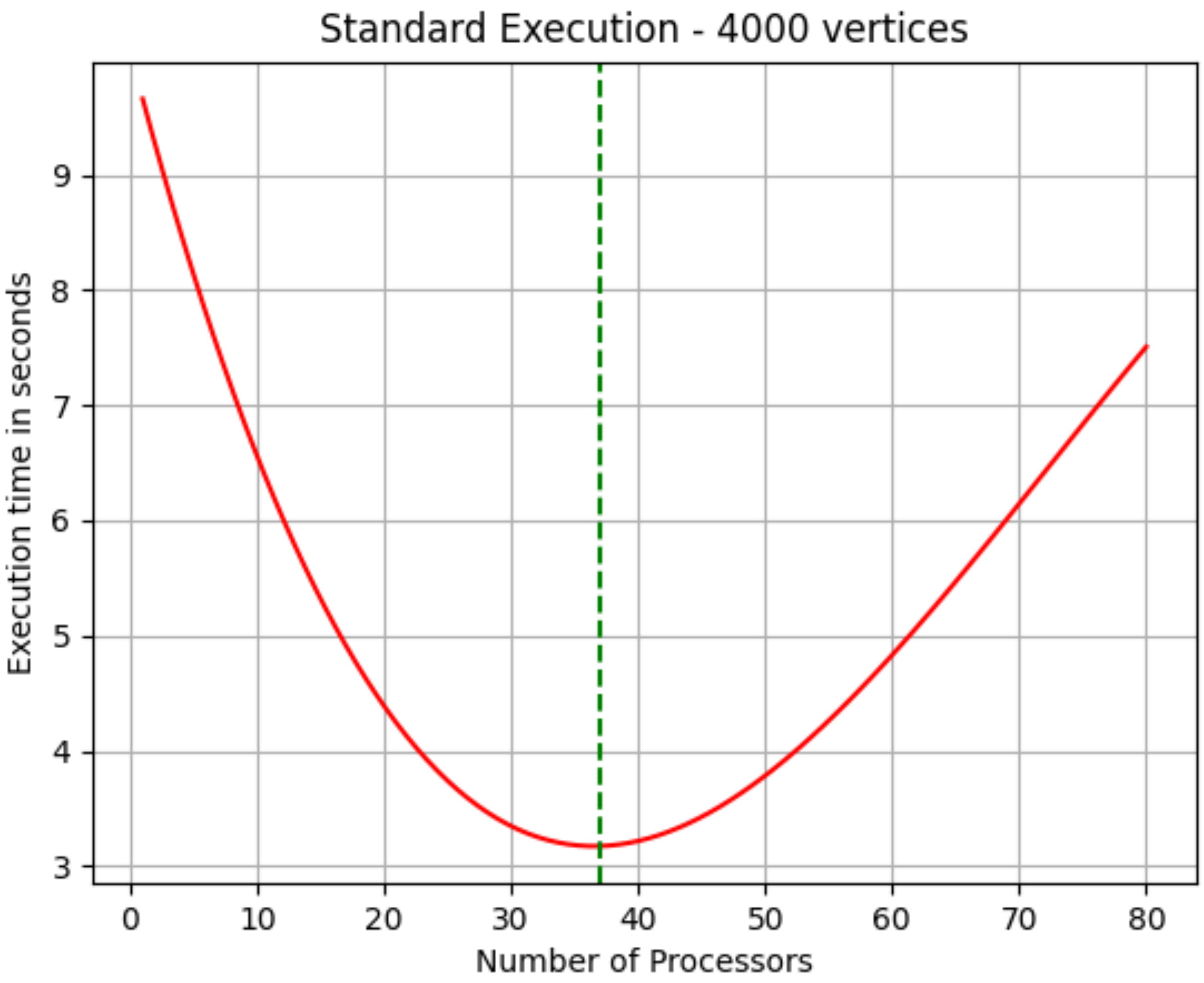
- Efficiency: Parallel BFS improves performance by processing multiple vertices in parallel, significantly enhancing overall efficiency.
- Scalability: It is highly scalable and can handle large scale graphs.
- Concurrency: Parallel BFS allows for concurrent exploration, minimizing idle time and maximizing resource utilization.
- Load balancing: This ensures efficient utilization of computational resources.

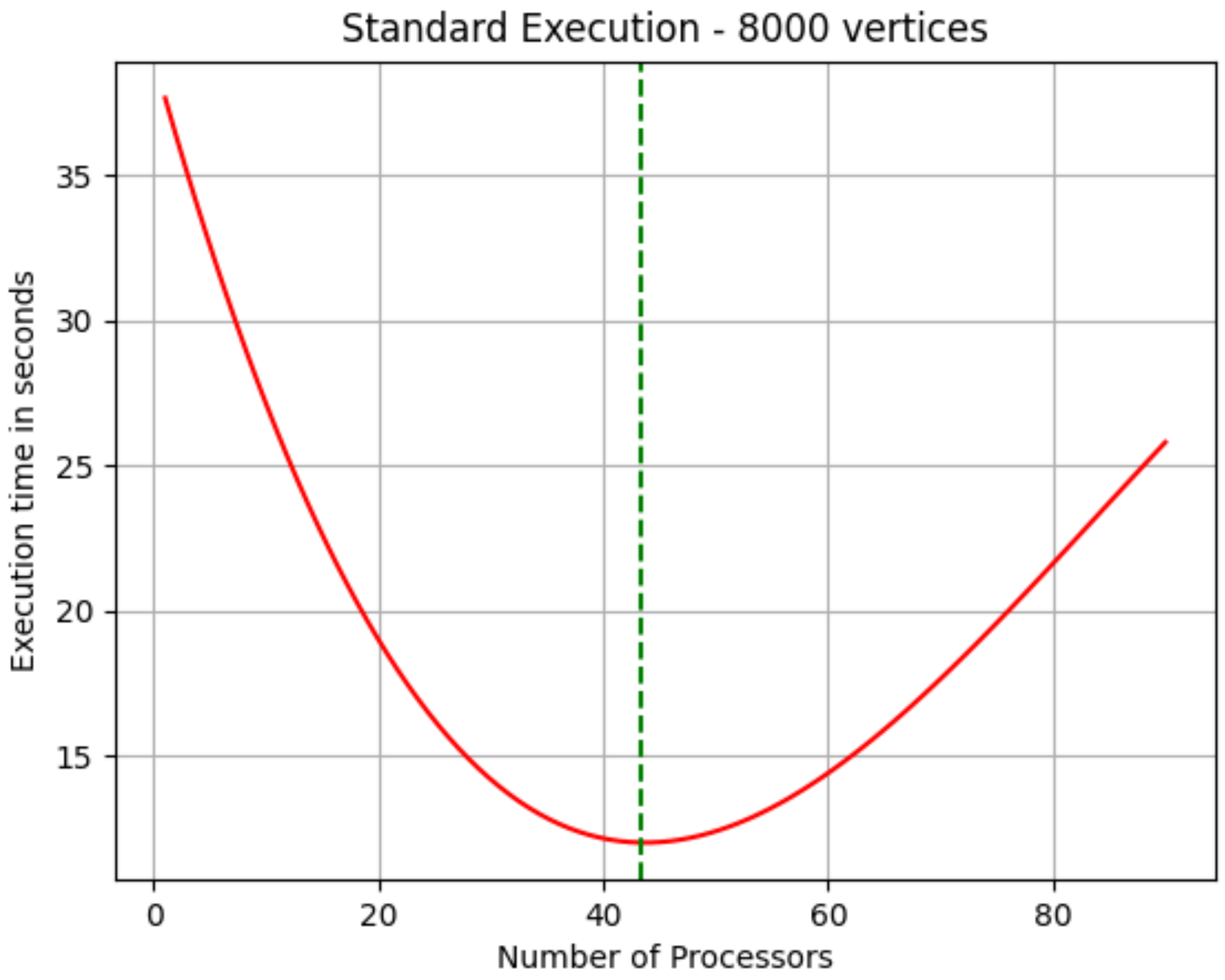
$ slurm.sh

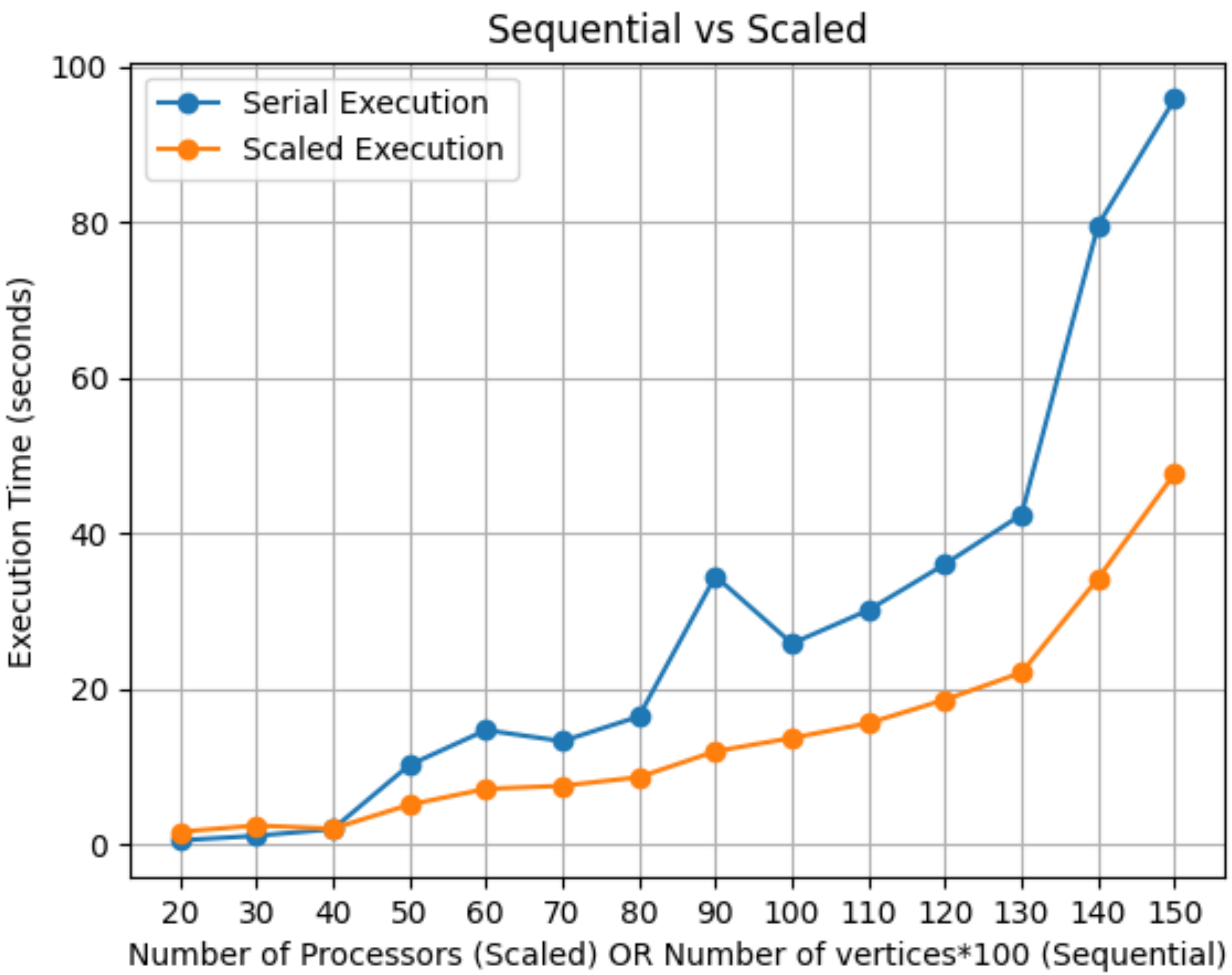
```bash
#!/bin/bash

#SBATCH --nodes=40
#SBATCH --ntasks-per-node=1
#SBATCH --constraint=IB|OPA
#SBATCH --time=00:10:00
#SBATCH --partition=general-compute
#SBATCH --qos=general-compute
#SBATCH --job-name="bfs-4000-vertices-40-nodecore"
#SBATCH --output=output-4000-vertices-40-nodecore.txt
#SBATCH --error=output-4000-vertices-40-error.txt
#SBATCH --exclusive

module load ccrsoft/2023.01
module load gcccore/11.2.0
module load intel
module load python/3.9.6

export I_MPI_PMI_LIBRARY=/opt/software/slurm/lib64/libpmi.so
srun pip install mpi4py numpy > /dev/null 2>&1

srun -n 40 python parallel-bfs.py 4000 60
```


Standard Execution - 1000 vertices
Execution time in seconds
0.5
1.0
1.5
2.0
2.5
0
10
20
30
40
50
60
70
Number of Processors


Standard Execution - 2000 vertices
Execution time in seconds
5.0
4.5
4.0
3.5
3.0
2.5
2.0
0
20
40
60
80
Number of Processors

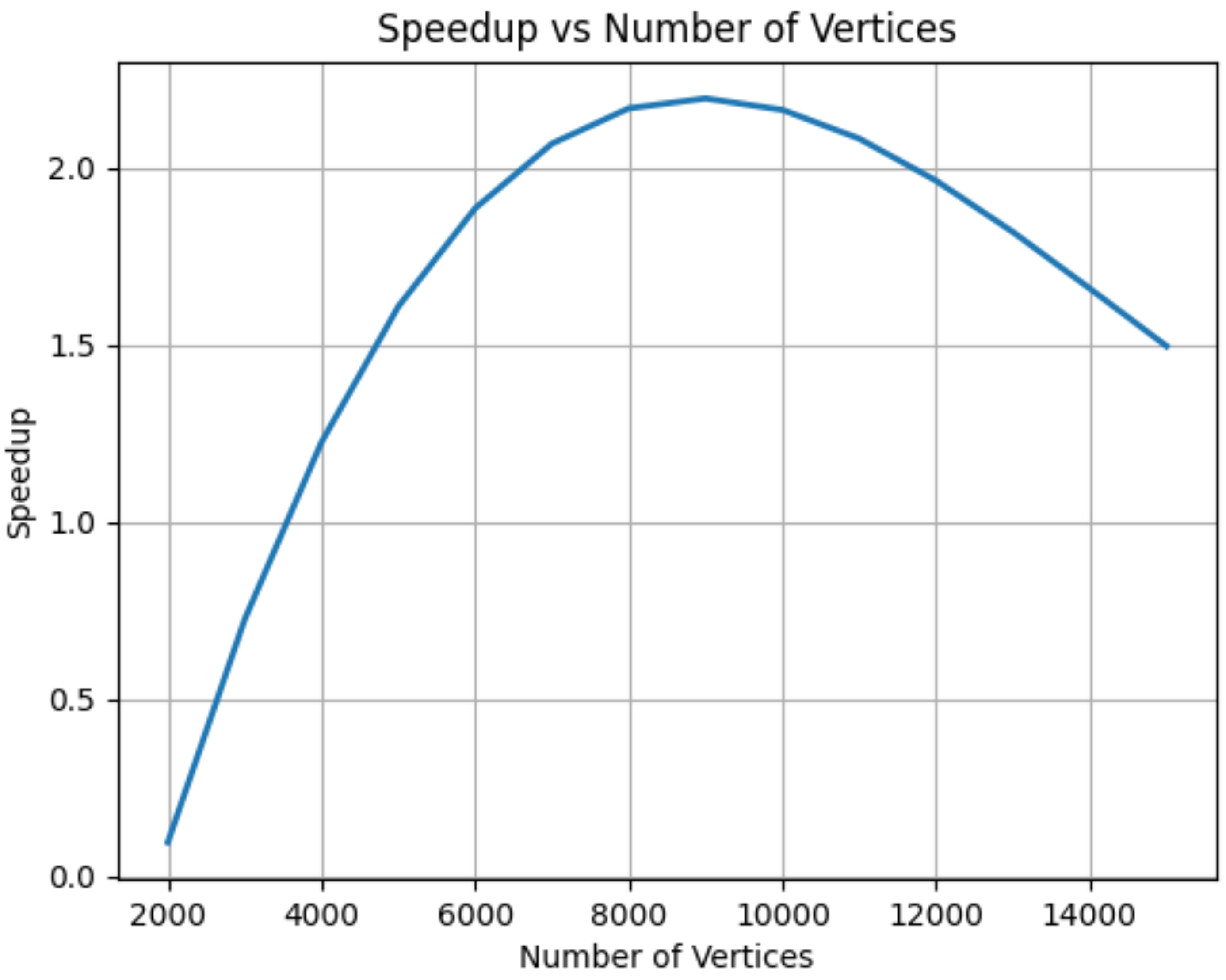

Standard Execution - 4000 vertices
Execution time in seconds
Number of Processors
3
4
5
6
7
8
9
0
10
20
30
40
50
60
70
80


Standard Execution - 8000 vertices
Execution time in seconds
35
30
25
20
15
0
20
40
60
80
Number of Processors


Sequential vs Scaled
Serial Execution
Scaled Execution
Execution Time (seconds)
0
20
40
60
80
100
20 30 40 50 60 70 80 90 100 110 120 130 140 150
Number of Processors (Scaled) OR Number of vertices*100 (Sequential)

Speed up = T_{seq} / T_p

T_{seq} is the execution time of sequential algorithm.
T_p is the execution time of the parallel algorithm with p Processors



Future Work

- Access nodes greater than 143 nodes with 1 core per node.
- Test performance by changing density of edges in the graphs.
- Implement my parallel approach using OpenMPI or Hybrid of MPI and OpenMPI.

References

- Wikipedia https://en.wikipedia.org/wiki/Parallel_breadth-first_search
- BFS https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/
- Parallel BFS on Distributed Memory Systems https://people.eecs.berkeley.edu/~aydin/sc11_bfs.pdf
- Distributed BFS Algorithm, IIT Delhi https://www.youtube.com/watch?v=wpWvCabHqQU
- Applications https://www.ijcsma.com/articles/graph-traversals-and-its-applications-in-graph-theory.pdf
- CCR Docs https://docs.ccr.buffalo.edu/en/latest/
- MPI for Python https://mpi4py.readthedocs.io/en/stable/
- MPI python https://www.youtube.com/watch?v=36nCgG40DJo&ab_channel=SharcnetHPC