PARALLEL BREADTH-FIRST SEARCH USING MPI

CSE 633: Parallel Algorithms

Guide: Dr. Russ Miller

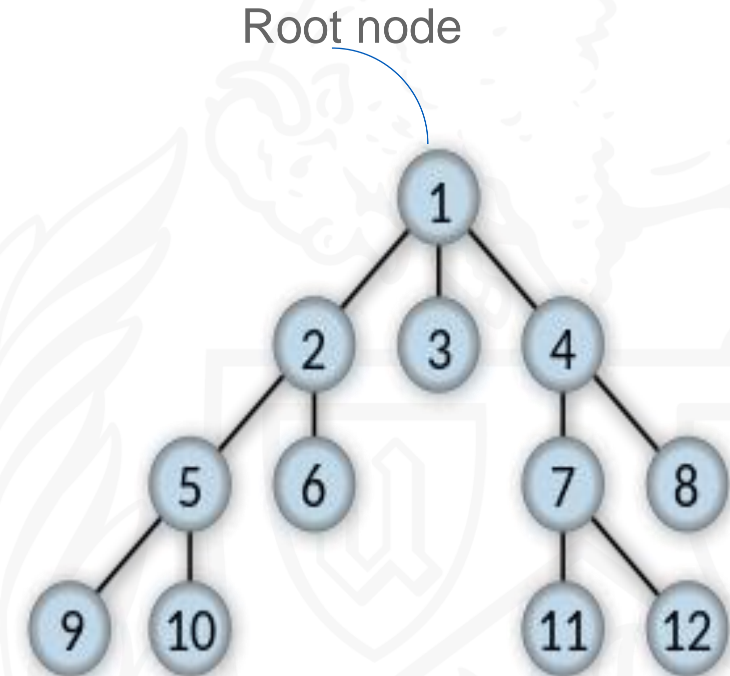
Presenter: Sumanth Thota

University at Buffalo The State University of New York

- Breadth-First Search
- Application of BFS
- Sequential BFS Algorithm
- Communication
- Parallel BFS Algorithm & implementation
- Results
- Conclusion
- References

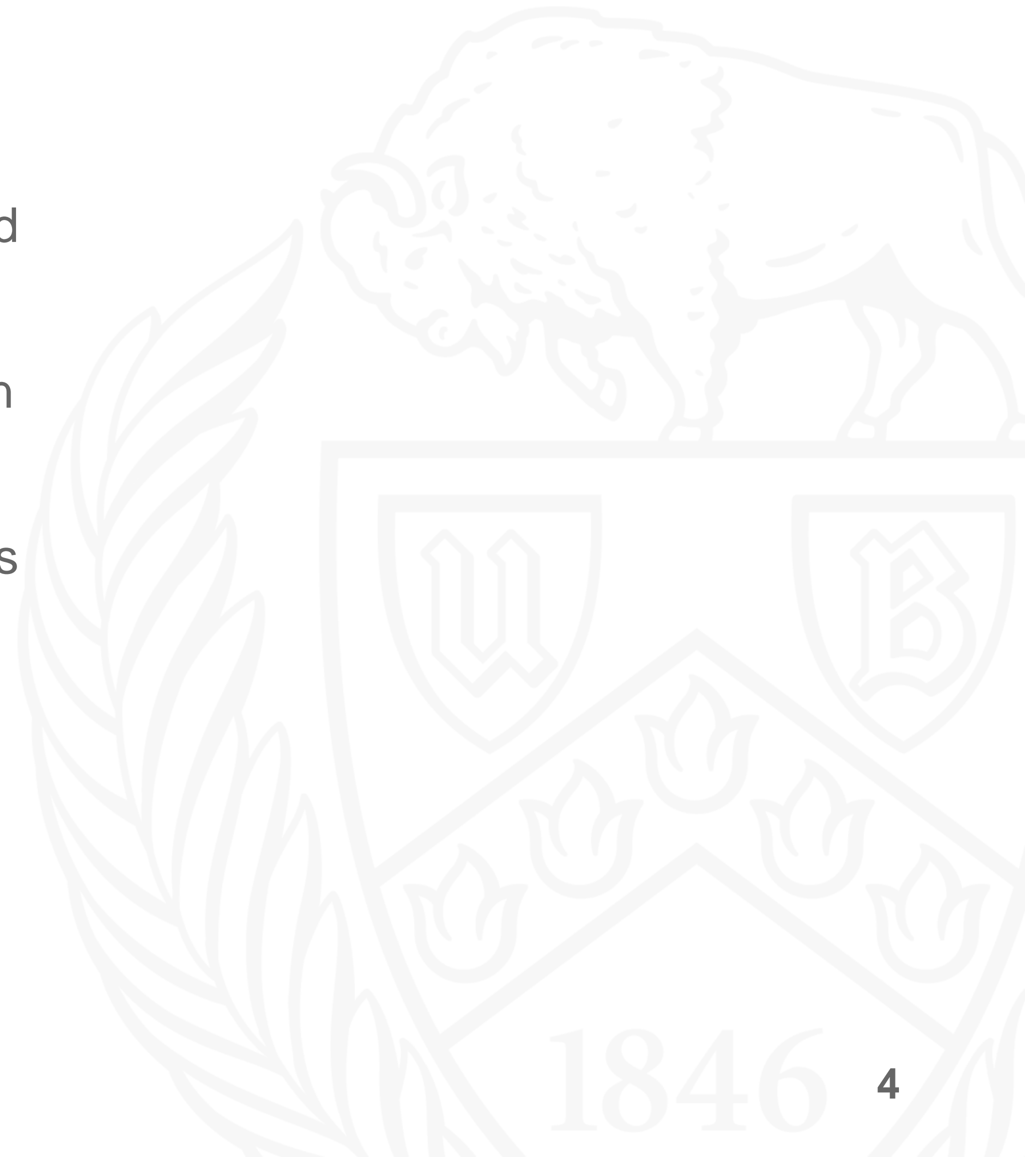
Breadth-First Search

- BFS is a graph traversal algorithm that visits all the vertices of a graph in breadth-first order.
- It starts at the root node and visits all the nodes at the same level before moving on to the next level.
- BFS is typically used to find the shortest path between two nodes.
- One drawback of BFS is that it requires more memory as it needs to keep track of all the nodes in the queue.



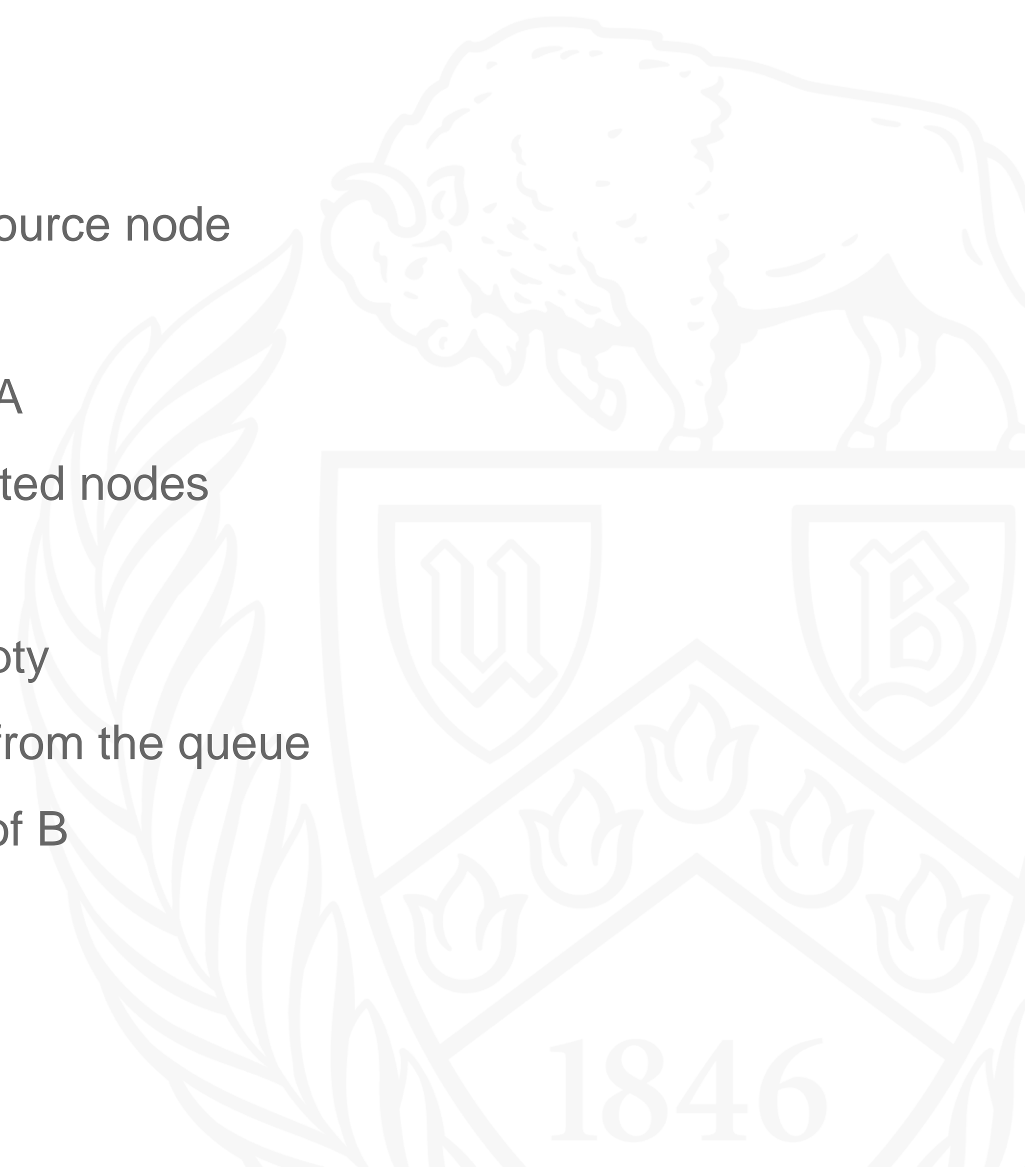
Applications of BFS

- BFS is used by search engines like Google to crawl the web and index web pages.
- BFS can be used to find the shortest path between two users on a social networking site like Facebook or LinkedIn.
- BFS can be used to find the shortest path between two locations on a map as routing algorithms for navigation systems.
- BFS can be used in AI applications, such as pathfinding and decision-making.



Sequential BFS Algorithm

```
BreadthFirstSearch(G, A):            // G is a graph and A is the source node
    Q = Queue()                      // Create an empty queue
    Q.enqueue(A)                     // Enqueue the source node A
    visited = set()                  // Create an empty set of visited nodes
    visited.add(A)                   // Mark A as visited
    while not Q.isEmpty():           // While the queue is not empty
        B = Q.dequeue()              // Dequeue the next node B from the queue
        for C in G.neighbors(B):     // Process all the neighbors of B
            if C not in visited:     // If C is not visited
                Q.enqueue(C)         // Enqueue C
                visited.add(C)       // Mark C as visited
```



Parallel BFS Algorithm

```
1    define 1_D_distributed_memory_BFS( graph(V,E), source s):
2        //normal initialization
3        for all v in V do
4            d[v] = -1;
5        d[s] = 0; level = 0; FS = {}; NS = {};
6        //begin BFS traversal
7        while True do:
8            FS = {the set of local vertices with level}
9            //all vertices traversed
10           if FS = {} for all processors then:
11               terminate the while loop
12           //construct the NS based on local vertices in current frontier
13           NS = {neighbors of vertices in FS, both local and not local vertices}
14           //synchronization: all-to-all communication
15           for 0 <= j < p do:
16               N_j = {vertices in NS owned by processor j}
17               send N_j to processor j
18               receive N_j_rcv from processor j
19           //combine the received message to form local next vertex frontier then update the level for them
20           NS_rcv = Union(N_j_rcv)
21           for v in NS_rcv and d[v] == -1 do
22               d[v] = level + 1
```

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

Communication

A B C D

Offset

0	0	0	1
0	0	0	1
0	0	0	1
1	1	1	0

Rank 0

Rank 1

Rank 2

Rank 3

Barrier

Parallel BFS Implementation

- Creating the data offset and sending it to all processors.

```
// parallel processing
void parallel(int n, int* adjacency_matrix, int rank, int size, int save)
{
    int level = 0;
    bool alive = true;
    std::queue<int> fs, ns; // "frontier" queue and queue for the next level
    std::vector<bool> used(n);
    std::vector<int> d(n); // distance to vertices
    int* sendcounts = (int*)malloc(sizeof(int) * size);
    int* displs = (int*)malloc(sizeof(int) * size);
    // calculate the number of vertices and offset in the adjacency matrix
    int count = n;
    for (int i = 0; i < size - 1; i++)
    {
        sendcounts[i] = (n / size) * n;
        displs[i] = (n - count) * n;
        count -= (n / size);
    }
    sendcounts[size - 1] = count * n;
    displs[size - 1] = (n - count) * n;

    // adjacency matrix distribution for each rank
    int* adjacency_thread = (int*)malloc(sizeof(int) * n * n);
    MPI_Scatterv(adjacency_matrix, sendcounts, displs, MPI_INT, adjacency_thread, n * n, MPI_INT, 0, MPI_COMM_WORLD);
```

Parallel BFS Implementation

- Pop and push operations on FS and NS for all processors.

```
if (rank == 0)
{
    fs.push(0);
    used[0] = true;
    d[0] = level;
}

// processing is in progress while the "frontier" queue of at least one of the ranks is not empty
while (alive)
{
    level++;
    // while the queue at the current level is not empty, we look through the vertices
    while (!fs.empty())
    {
        int v = fs.front();
        fs.pop();
        for (int i = 0; i < n; i++)
        {
            //not previously visited edge enqueued for the next level and mark visited
            int to = adjacency_thread[adjust_vertex(n, size, v) * n + i];
            if (to == 1 && !used[i])
            {
                used[i] = true;
                ns.push(i);
            }
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
```

Parallel BFS Implementation

- Merging the received data and sending the updated data to all processors

```
if (rank == 0)
{
    // if the rank is root, then we accept queues from other ranks, otherwise we send
    bool* recv_q = (bool*)calloc(n, sizeof(bool));
    memcpy(recv_q, send_q, sizeof(bool) * n);

    for (int i = 1; i < size; i++)
    {
        MPI_Recv(send_q, n, MPI_C_BOOL, i, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        for (int j = 0; j < n; j++)
        {
            if (send_q[j] == true)
            {
                recv_q[j] = true;
                used[j] = true;
                if (d[j] == 0 && j != 0) d[j] = level;
            }
        }
    }


    for (int i = 0; i < n / size; i++) if (recv_q[i]) fs.push(i);
    for (int i = 1; i < size; i++) MPI_Send(recv_q, n, MPI_C_BOOL, i, 0, MPI_COMM_WORLD);
    free(recv_q);

}
else
{
    MPI_Send(send_q, n, MPI_C_BOOL, 0, rank, MPI_COMM_WORLD); // send next level queue to rank 0
    MPI_Recv(send_q, n, MPI_C_BOOL, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);// receive the next level common queue
    if (rank != size - 1)
    {
        for (int i = (n / size) * rank; i < (n / size) * (rank + 1); i++) if (send_q[i]) fs.push(i);
    }
    else for (int i = (n / size) * rank; i < n; i++) if (send_q[i]) fs.push(i);
}
MPI_Barrier(MPI_COMM_WORLD);
```

Sample Output

```
Generating adjacency matrix...
Serial processing, please wait...
TIME: 2.629992 seconds
 Parallel processing, please wait...
TIME: 1.182144 seconds
```

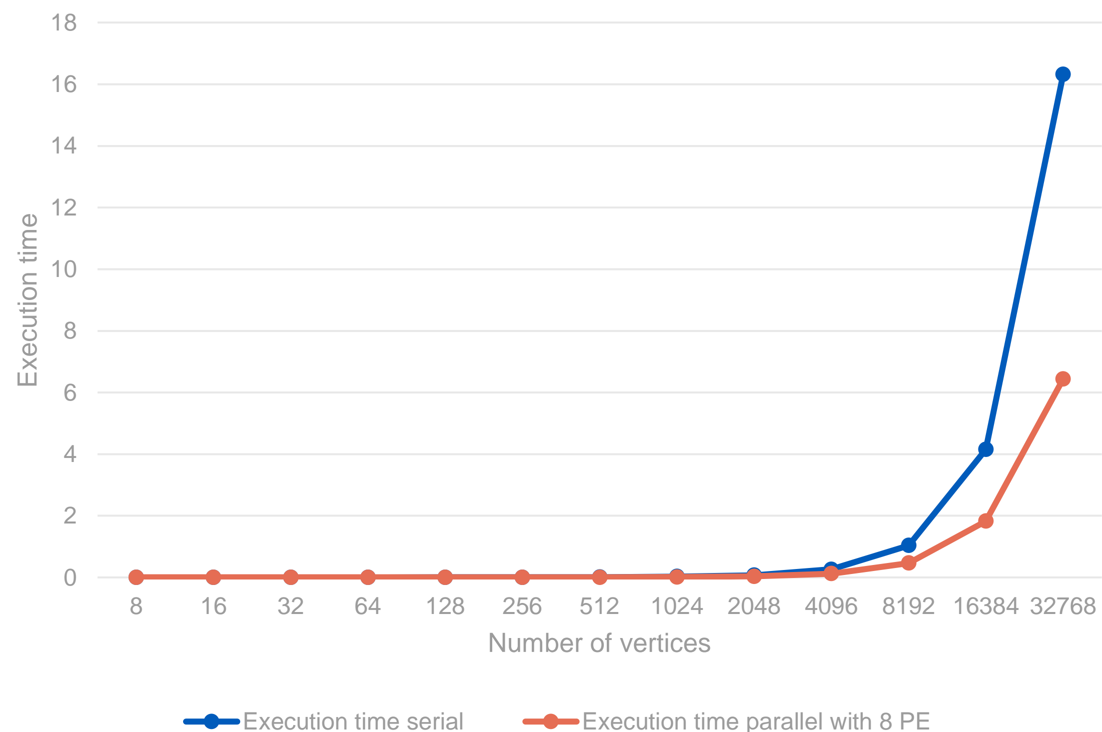
Slurm Script

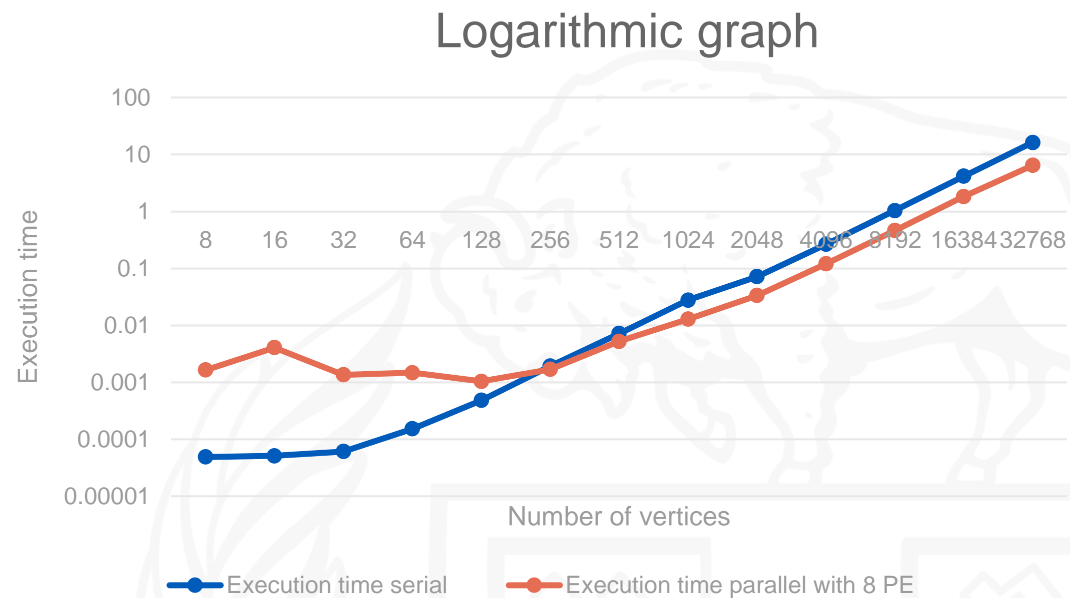
slurm.sh

```
#!/bin/sh
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=1
#SBATCH --constraint=IB|OPA
#SBATCH --time=00:10:00
#SBATCH --partition=general-compute
#SBATCH --qos=general-compute
#SBATCH --mail-type=END
#SBATCH --mail-user=sthota5@buffalo.edu
#SBATCH --job-name="test"
#SBATCH --output=pbfs.out
#SBATCH --exclusive
#SBATCH --mem=200G

module load intel
module list
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
source /util/academic/intel/20.2/compilers_and_libraries_2020.2.254/linux/mpi/intel64/bin/mpivars.sh
unset I_MPI_PMI_LIBRARY
mpicxx -o pbfs pbfs.cpp
srun -n 8 ./pbfs
```

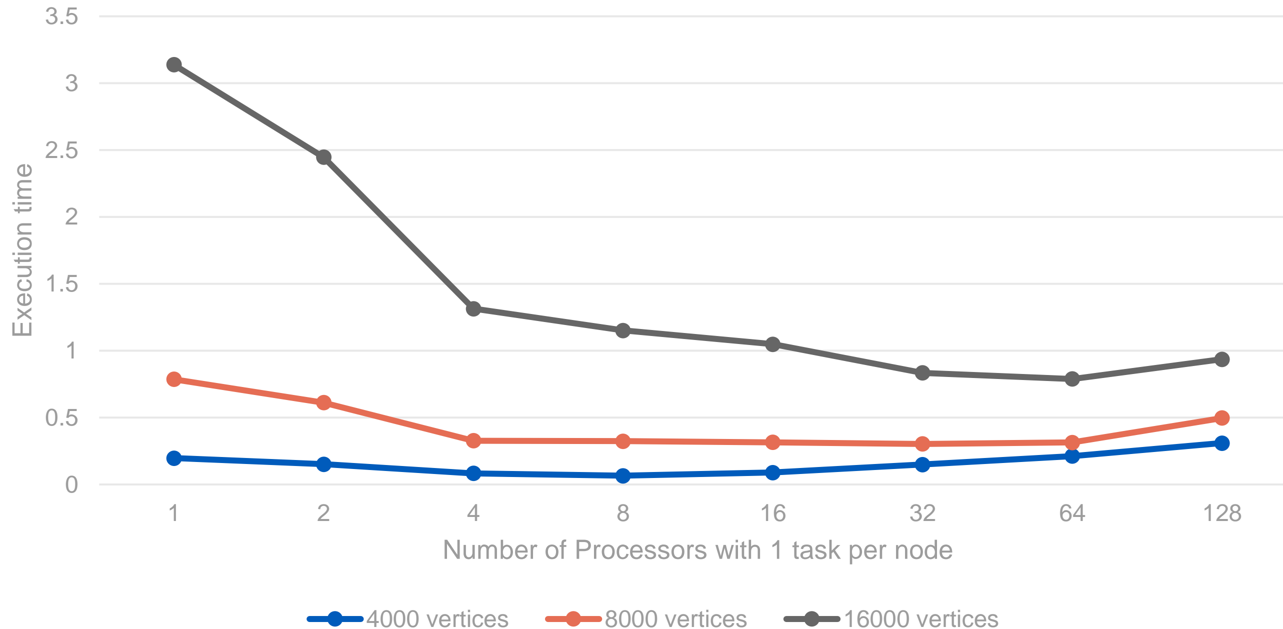
Execution time VS No. of vertices with constant 8 PE





No of vertices	Execution time serial	Execution time parallel w 8 PE
8	0.000049	0.001647
16	0.000051	0.004066
32	0.000061	0.001355
64	0.000152	0.001477
128	0.000483	0.001041
256	0.001925	0.001688
512	0.007164	0.005234
1024	0.027783	0.012878
2048	0.071938	0.03344
4096	0.26456	0.120046
8192	1.038307	0.462638
16384	4.149131	1.825428
32768	16.324268	6.438635

Execution time VS No. of processors

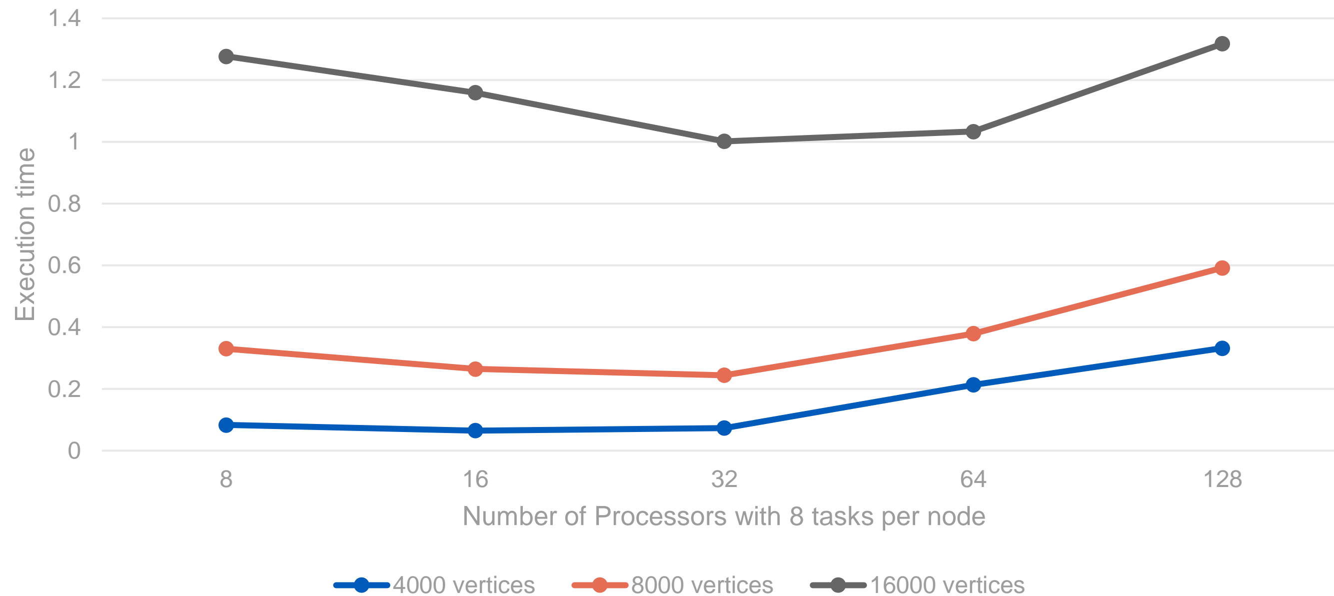


1 task per node

- The Execution time reduces linearly with increase in the number of processors
- The decrease in execution time gets more signification in more vertices

No of Processors	4000 vertices	8000 vertices	16000 vertices
1	0.1963	0.7855	3.1379
2	0.1513	0.6113	2.4469
4	0.0830	0.3272	1.3138
8	0.0650	0.3235	1.1504
16	0.0883	0.3146	1.0477
32	0.1482	0.3023	0.8340
64	0.2119	0.3136	0.7888
128	0.3081	0.4961	0.9360

Execution time VS No. of processors

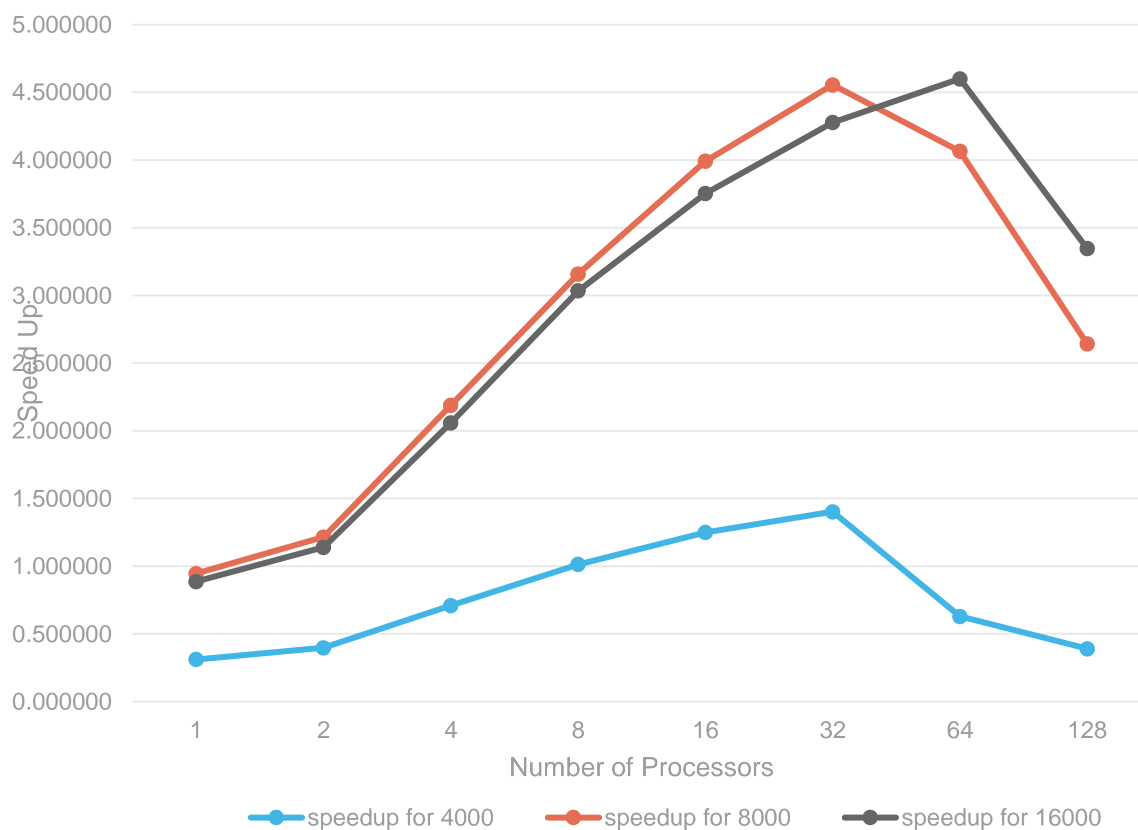


8 task per node

- The experiment shows a distinct increase in the execution time as the number of processors exceeds 32.

No of Processors	4000 vertices	8000 vertices	16000 vertices
8	0.0831	0.3301	1.2770
16	0.0647	0.2643	1.1593
32	0.0733	0.2440	1.0018
64	0.2130	0.3789	1.0333
128	0.3319	0.5918	1.3178

Speedup VS No. of processors

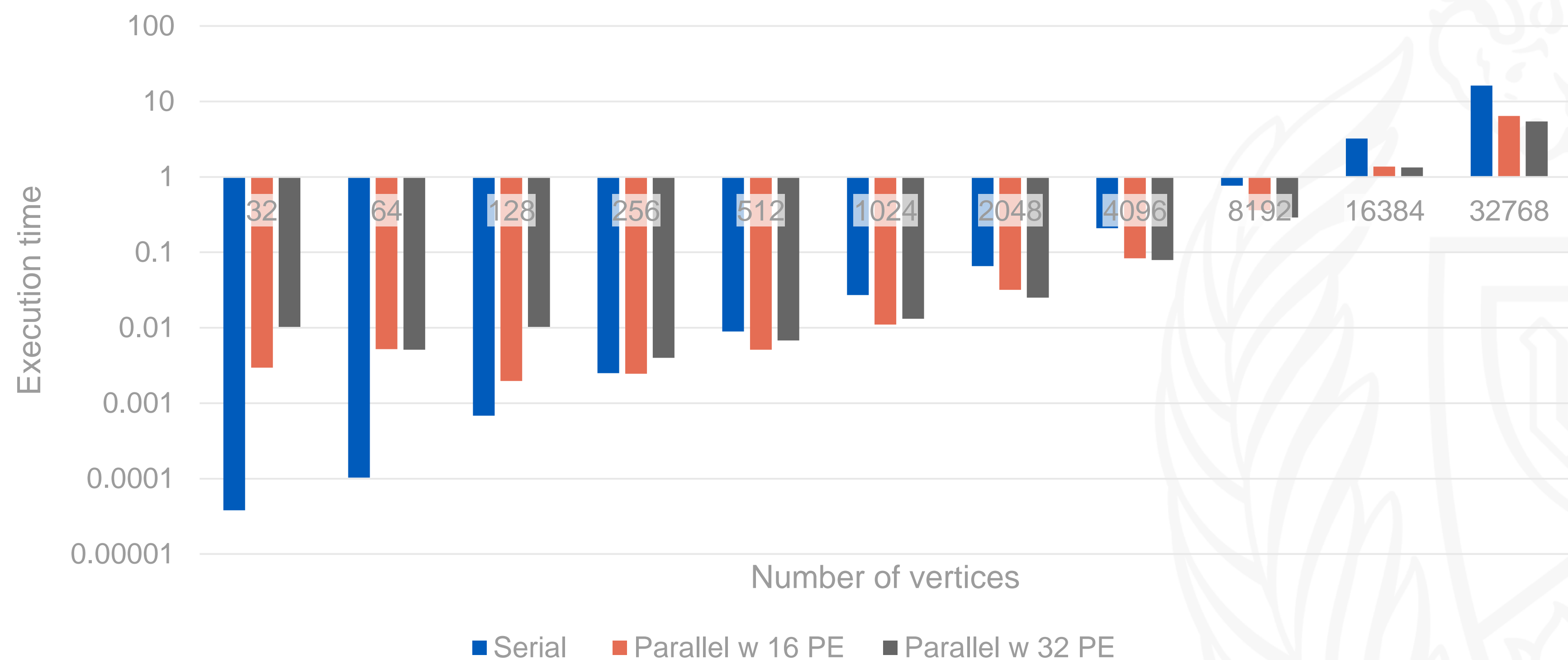


Speed up = $T_{serial} / T_{Parallel}$

- For smaller number of graph there isn't much improvement in the speedup
- As we increase the number of vertices the speedup increases with processors
- After one point the trend starts going down even for large number of vertices.
- Likely because of the parallel communication overhead

No of Processors	speedup for 4000	speedup for 8000	speedup for 16000
1	0.3114	0.9461	0.8867
2	0.3976	1.2157	1.1397
4	0.7099	2.1888	2.0592
8	1.0136	3.1585	3.0337
16	1.2496	3.9910	3.7524
32	1.4026	4.5546	4.2779
64	0.6290	4.0648	4.5996
128	0.3900	2.6419	3.3467

Bar graph comparison for all three types of execution



Logarithmic Bar Graph

Conclusion

- As can be interpreted form the graphs that the algorithm for parallel BFS is working effectively.
- The trends in the graphs reflect that parallelizing the process make it more efficient in terms of execution time but only up to a certain number of processors.
- For the input size of around 30K vertices, making the adjacency matrix size (30K * 30k), we can see the algorithm works effectively until 32 processors. From there we can observe the increase in the execution time with increase in processors due to communication over head.

References

- https://en.wikipedia.org/wiki/Parallel_breadth-first_search
- https://people.eecs.berkeley.edu/~aydin/sc11_bfs.pdf
- https://medium.com/geekculture/configuring-mpi-on-windows-10-and-executing-the-hello-world-program-in-visual-studio-code-2019-879776f6493f
- https://docs.ccr.buffalo.edu/en/latest/hpc/data-transfer/#data-transfer
- https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather

Thank You