PARALLEL BREADTH FIRST SEARCH USING MPI

Course: CSE 708

Presenter: Venkata Bala Vamsi

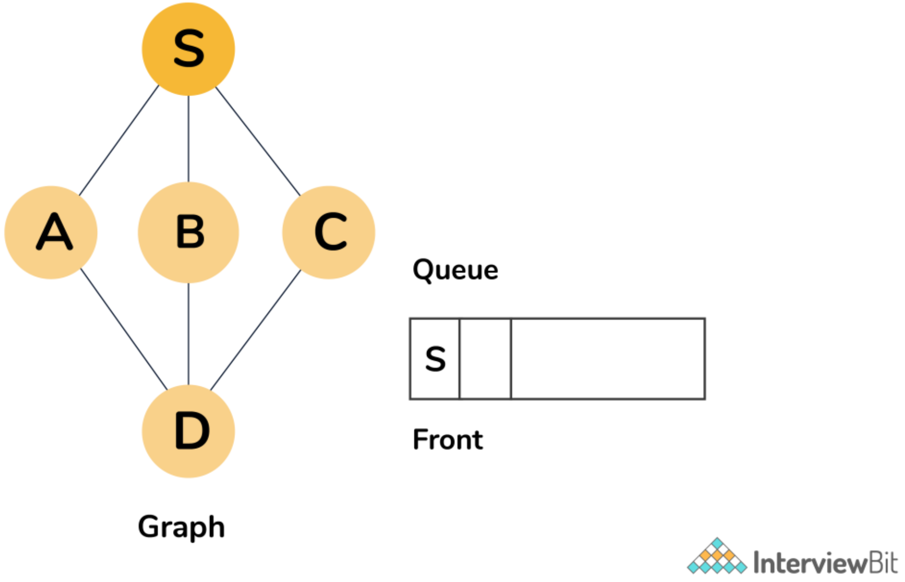
Instructor: Dr. Russ Miller

University at Buffalo The State University of New York

CONTENTS:

Overview of BFS

- Given a source node, BFS performs a Level Order traversal of the graph with respect to the source node.
- Explores all the vertices in the current level before moving on to exploring vertices in the next level.



BFS Traversal: S A B C D

Applications of BFS

1. Shortest Path
2. Cycle Detection
3. Finding connected components
4. Network Broadcast

Sequential approach to BFS

```
int visited[all_nodes] = {0};
queue<int>Q;

Q.push(source_node);
visited[source_node] = 1;

while(Q is not empty){
    int current_node = Q.front();
Q.pop();
    for(each adj_node of current_node){
        if(visited[adj_node] == 0){
            Q.push(adj_node);
            visited[adj_node] = 1;
        }
    }
}
```

The Necessity of Parallelization

- **Memory Constraints:** Large Graphs can exceed memory capacity of a single processor.
- **High Computational Demand:** BFS is computationally intensive because it explores each vertex and edge of the graph.

Sequential Implementation:

```
int visited[all_nodes] = {0};
queue<int>Q;

Q.push(source_node);
visited[source_node] = 1;

while(Q is not empty){
    int current_node = Q.front();
Q.pop();
    for(each adj_node of current_node){
        if(visited[adj_node] == 0){
            Q.push(adj_node);
            visited[adj_node] = 1;
        }
    }
}
```

→

High level Parallel Implementation:

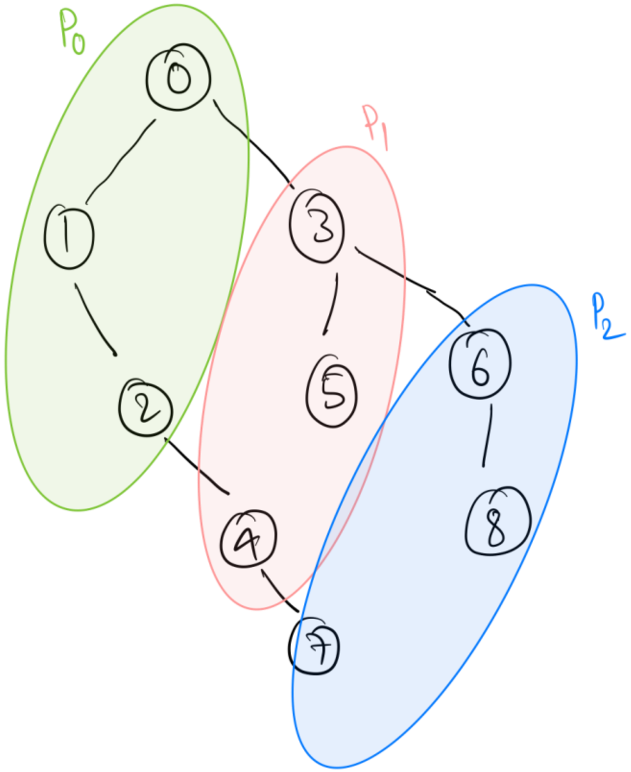
```
vector<bool> visited(n, false);
vector<int> frontier; // nodes in current level
vector<int> next_frontier; // nodes in next level

int curr_level = 0;
frontier.push_back(source_node);
visited[source_node] = true;

while(!frontier.empty()){
    for(int node : frontier){
        for(int adj_node : adj[node]){
            if(!visited[adj_node]){
                visited[adj_node] = true;
                next_frontier.push_back(adj_node);

            }
        }
    }

    frontier = next_frontier;
    next_frontier.clear();
    curr_level++;
}
```

```cpp
void distributed_bfs_1d(const Graph& local_graph, int source, int num_vertices, int rank, int ranks)
{   int vertices_per_rank = num_vertices / ranks;

    vector<int> levels(vertices_per_rank, -1); // Levels of vertices in the BFS tree
    vector<int> frontier;
    vector<int> next_frontier;

    if (owner(source, vertices_per_rank) == rank) {
        frontier.push_back(source % vertices_per_rank);
        set_level(levels, source % vertices_per_rank, 0);
    }

    int curr_level = 0;
    while (true) {
        vector<vector<int>> send_buffer(ranks);
        vector<int> local_new_frontier;

        for (int v : frontier) {
            int global_v = rank * vertices_per_rank + v;

            for (int neighbor : local_graph[v]) {
                int neighbor_owner = owner(neighbor, vertices_per_rank);
                if (neighbor_owner != rank) {
                    send_buffer[neighbor_owner].push_back(neighbor);
                } else {
                    int local_neighbor = neighbor % vertices_per_rank;
                    if (levels[local_neighbor] == -1) {
                        levels[local_neighbor] = curr_level + 1;
                        local_new_frontier.push_back(local_neighbor);
                    }
                }
            }
        }

        all_to_all_communication(send_buffer, ranks, next_frontier);

        frontier.clear();

        for (int v : next_frontier) {
            int local_v = v % vertices_per_rank;
            if (levels[local_v] == -1) {
                levels[local_v] = curr_level + 1;
                local_new_frontier.push_back(local_v);
            }
        }

        frontier.swap(local_new_frontier);
        next_frontier.clear();

        // Check for global termination condition
        int local_size = frontier.size();
        int global_size;
        MPI_Allreduce(&local_size, &global_size, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
        if (global_size == 0) break;  // Termination condition

        ++curr_level;
    }
}
```

Graph nodes are distributed evenly across the available processors.

Each Processor maintains its own:

- **Local adjacency list**, which corresponds to the subset of vertices it is responsible for.
- **Vertex levels**, specifically for the vertices it owns, denoting their distances from the source vertex.
- **Current frontier**, which is a list of vertices it owns that are to be explored at the current level of the algorithm.
- **Next frontier**, which comprises the vertices it owns that will be explored in the subsequent level of the algorithm.



Processor 0:

Levels = { 0: -1, 1: -1, 2: -1}
FS = {0}

ALL to ALL communication

Levels = { 0: 0, 1: -1, 2: -1}
FS = {1}

ALL to ALL communication

Levels = { 0: 0, 1: 1, 2: -1}
FS = {2}

ALL to ALL communication

Levels = { 0: 0, 1: 1, 2: 2}
FS = {}

ALL to ALL communication

Levels = { 0: 0, 1: 1, 2: 2}
FS = {}

ALL to ALL communication

Levels = { 0: 0, 1: 1, 2: 2}
FS = {}

Processor 1:

Levels = {3: -1, 4: -1, 5: -1}
FS = {}

ALL to ALL communication

Levels = {3: -1, 4: -1, 5: -1}
FS = {3}

ALL to ALL communication

Levels = {3: 1, 4: -1, 5: -1}
FS = {5}

ALL to ALL communication

Levels = {3: 1, 4: -1, 5: 2}
FS = {4}

ALL to ALL communication

Levels = {3: 1, 4: 3, 5: 2}
FS = {}

ALL to ALL communication

Levels = {3: 1, 4: 3, 5: 2}
FS = {}

Processor 2:

Levels = {6: -1, 7: -1, 8: -1}
FS = {}

ALL to ALL communication

Levels = {6: -1, 7: -1, 8: -1}
FS = {}

ALL to ALL communication

Levels = {6: -1, 7: -1, 8: -1}
FS = {6}

ALL to ALL communication

Levels = {6: 2, 7: -1, 8: -1}
FS = {8}

ALL to ALL communication

Levels = {6: 2, 7: -1, 8: 3}
FS = {7}

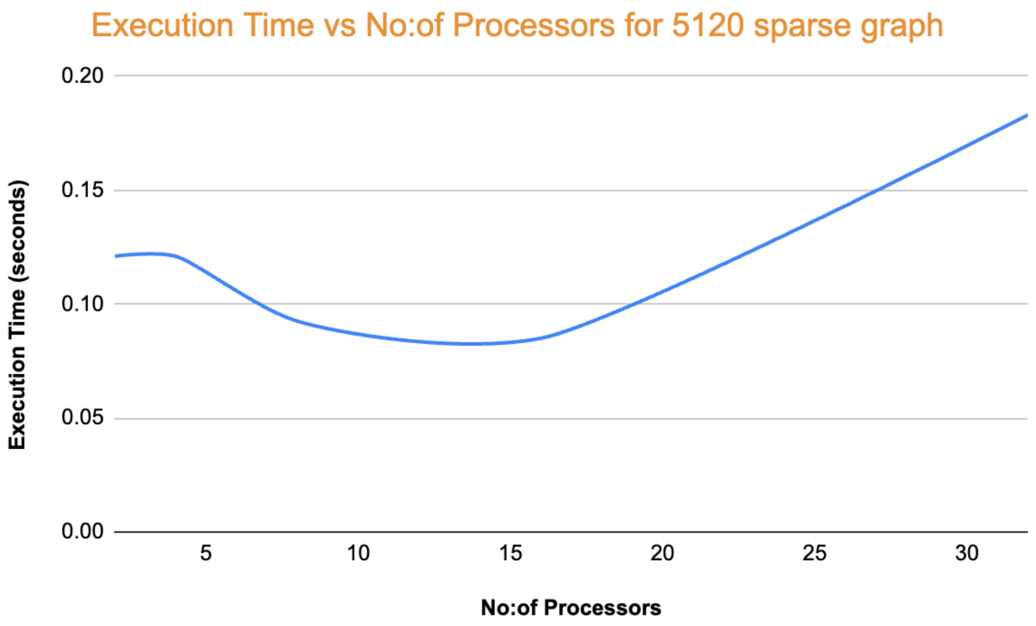
ALL to ALL communication

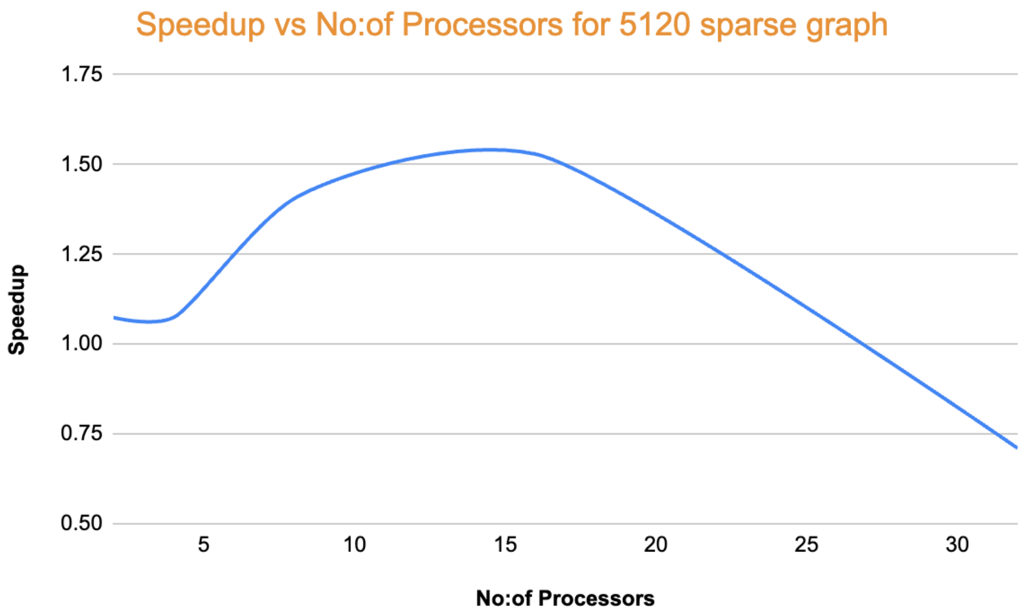
Levels = {6: 2, 7: 4, 8: 3}
FS = {}

Slurm.sh:

```
#!/bin/sh
#SBATCH --ntasks-per-node=1
#SBATCH --nodes=16
#SBATCH --time=00:03:00
#SBATCH --job-name=bfs_sparse_5000_vertices_16:1_nodes
#SBATCH --output=bfs_sparse_5000_vertices_16:1_nodes.out
#SBATCH --mem=5000M
#SBATCH --partition=general-compute
#SBATCH --qos=general-compute
#SBATCH --mail-type=END
#SBATCH --mail-user=vatukuri@buffalo.edu
#SBATCH --cluster=ub-hpc
#SBATCH --exclusive

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
mpicxx -o parallel_bfs_mpi parallel_bfs_mpi.cpp
srun -n 16 ./parallel_bfs_mpi
```

Results for 5120 sparse graph vertices

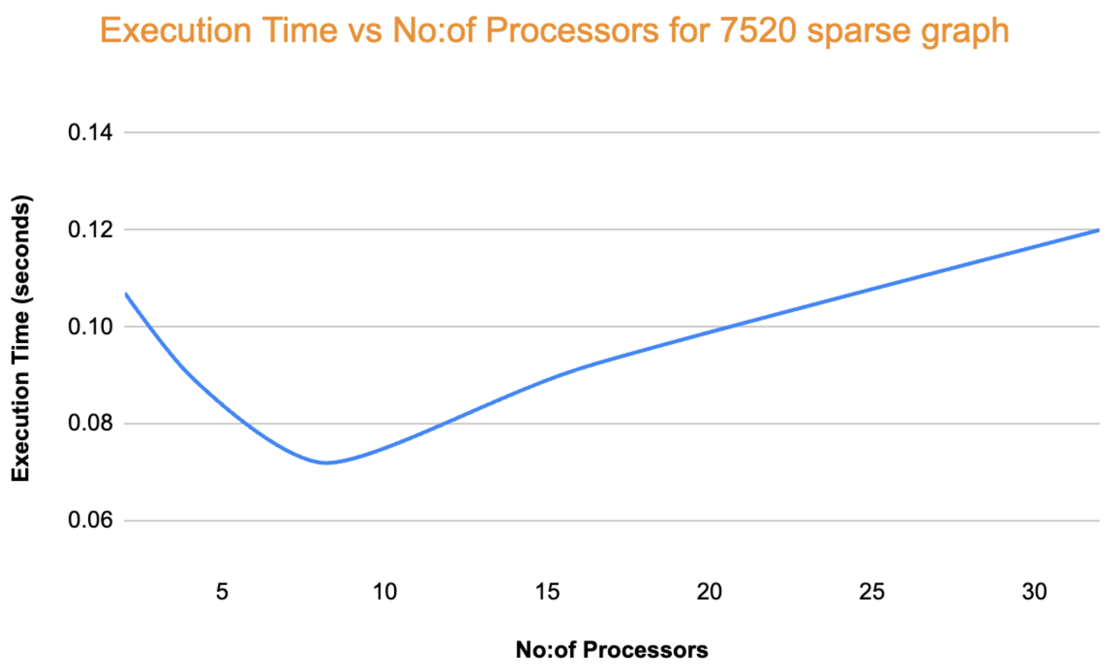


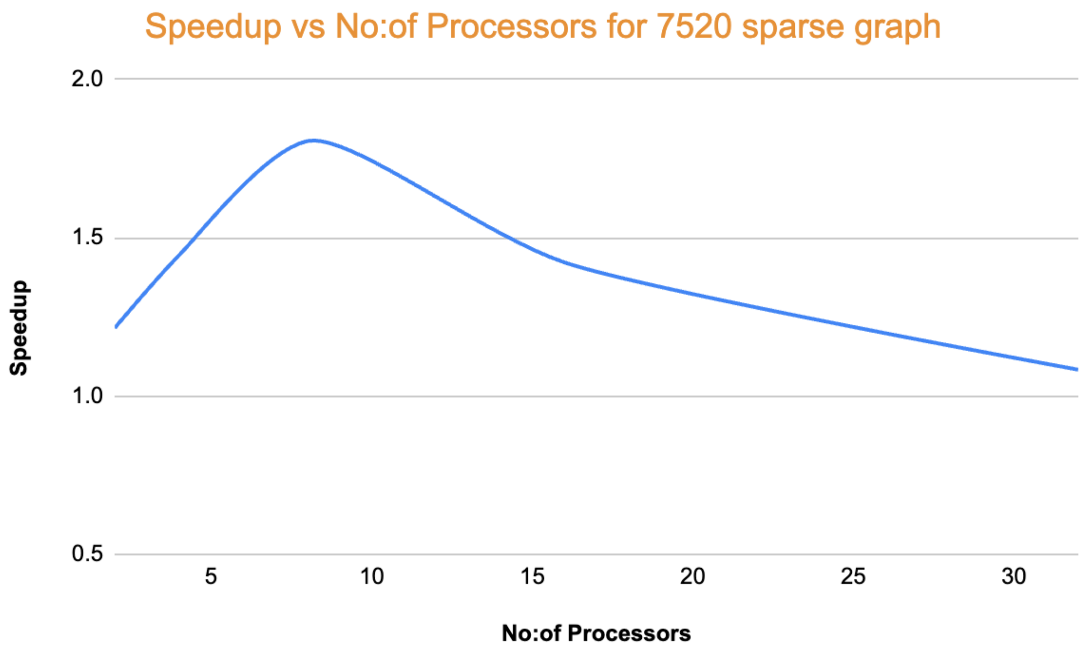


1 task per node

Speedup = Sequential Time / Parallel Execution Time

Results for 7520 sparse graph vertices





1 task per node

Speedup = Sequential Time / Parallel Execution Time

References:

- https://people.eecs.berkeley.edu/~aydin/sc11_bfs.pdf [Parallel Breadth-First Search on Distributed Memory Systems]

- https://www.youtube.com/watch?v=wpWvCabHqQU [Distributed BFS Algorithm, IIT Delhi July 2018]

- https://docs.ccr.buffalo.edu/en/latest/

- https://ubccr.freshdesk.com/support/solutions/articles/13000026245-tutorials-and-training-documents

Thank You. Questions?