# Assignment No: Group A_01

## Aim:

Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. For

1. Use a Tree
2. An undirected graph for BFS and DFS.

**Objective:**

Student will learn:

1. The Basic Concepts of DFS, BFS.
2. Multiple Compiler Directives, library routines, environment variables available for OpenMP

## Theory:

## Introduction:

## Breadth First Search (BFS)

There are many ways to traverse graphs. BFS is the most commonly used approach.

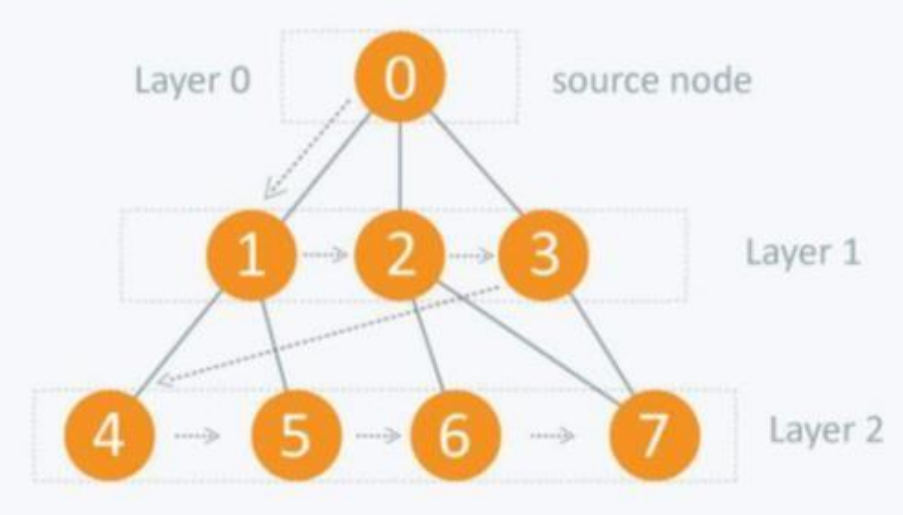
BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbor nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbor nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer

2. Move to the next layer

Consider the following diagram.




## Parallel Breadth First Search

1. To design and implement parallel breadth first search, you will need to divide the graph into smaller sub-graphs and assign each sub-graph to a different processor or thread.
2. Each processor or thread will then perform a breadth first search on its assigned sub-graph concurrently with the other processors or threads.
3. Two methods: Vertex by Vertex OR Level By Level

## Sample Program:

#include <iostream> #include <vector> #include <queue> #include <omp.h>

using namespace std;

void bfs(vector<vector<int>>& graph, int start, vector<bool>& visited) {

#pragma omp task firstprivate(vertex)

{

for (int neighbor : graph[vertex]) {

if (!visited[neighbor]) { q.push(neighbor); visited[neighbor] = true; #pragma omp task bfs(graph, neighbor, visited);

}}}}

}

void parallel_bfs(vector<vector<int>>& graph, int start) {

vector<bool> visited(graph.size(), false);

bfs(graph, start, visited);

}

## Parallel Depth First Search

- Different subtrees can be searched concurrently.
- Subtrees can be very different in size.
- Estimate the size of a subtree rooted at a node.
- Dynamic load balancing is required.

## Parameters in Parallel DFS: Work Splitting

- Work is split by splitting the stack into two.
- Ideally, we do not want either of the split pieces to be small.
- Select nodes near the bottom of the stack (node splitting), or
- Select some nodes from each level (stack splitting)
- The second strategy generally yields a more even split of the space.

## Sample Program

```
#include <iostream>

#include <vector> #include <stack> #include <omp.h>

using namespace std;

void dfs(vector<vector<int>>& graph, int start,

vector<bool>& visited) { stack<int> s; s.push(start); visited[start] = true;

#pragma omp parallel{

#pragma omp single{

while (!s.empty()) { int vertex = s.top(); s.pop();

#pragma omp task firstprivate(vertex){

for (int neighbor : graph[vertex]) {

if (!visited[neighbor]) { s.push(neighbor); visited[neighbor] = true; #pragma omp task dfs(graph, neighbor, visited);

}}}}}

}

void parallel_dfs(vector<vector<int>>& graph, int start) { vector<bool> visited(graph.size(), false);

dfs(graph, start, visited);

}
```

## OpenMP Section Compiler Directive:

A parallel loop is an example of independent work units that are numbered. If you have a pre-determined number of independent work units, the sections is more appropriate. In a sections construct can be any number of section constructs. These need to be independent, and they can be execute by any available thread in the current team, including having multiple sections done by the same thread.

The sections construct is a non-iterative work sharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team. Each structured block is executed once by one of the threads in the team in the context of its implicit task.

## Execute different code blocks in parallel

```
#pragma omp sections
{
    #pragma omp section { ... }
     ...
    #pragma omp section { .. }
}
```

## OpenMP Critical Compiler Directive

The omp critical directive identifies a section of code that must be executed by a single thread at a time..

```
#pragma omp critical
{
  code_block
}
```

**Conclusion:** Thus, we have successfully implemented parallel algorithms for Binary Search andBreadth First Search.