**Source**

#include<iostream>

#include<omp.h> #include<bits/stdc++.h> using namespace std; class Graph{ public:

// vector<vector<int>> graph;

// vector<bool> visited;

// int vertices = 0; // int edges = 0; int vertices = 6; int edges = 5; vector<vector<int>>graph={{1},{0,2,3},{1,4,5},{1,4},{2,3},{2}}; vector<bool> visited;

// Graph(){

// cout << "Enter number of nodes: ";

// cin >> vertices;

// cout << "Enter number of edges: ";

// cin >> edges;

// graph.assign(vertices,vector<int>());

// for(int i = 0 ; i < edges;i++){

// int a,b;

// cout << "Enter adjacent nodes: ";

// cin >> a >> b;

// addEdge(a,b);

// }

// }

void addEdge(int a, int b){ graph[a].push\_back(b); graph[b].push\_back(a);

} void printGraph(){ for(int i = 0; i < vertices; i++){ cout << i << " -> "; for(auto j = graph[i].begin(); j != graph[i].end();j++){ cout << \*j << " ";

} cout << endl;

} } void initialize\_visited(){ visited.assign(vertices,false);

} void dfs(int i){ stack<int> s; s.push(i); visited[i] = true; while(s.empty() != true){ int current = s.top(); cout << current << " "; s.pop(); for(auto j = graph[current].begin(); j != graph[current].end();j++){ if(visited[\*j] == false){ s.push(\*j); visited[\*j] = true;

}

}

}

}

void parallel\_dfs(int i){ stack<int> s; s.push(i); visited[i] = true; while(s.empty() != true){ int current = s.top(); cout << current << " "; #pragma omp critical s.pop();

#pragma omp parallel for for(autoj=graph[current].begin();j!=graph[current].end();j++){ if(visited[\*j] == false){

#pragma omp critical

{

s.push(\*j); visited[\*j] = true;

}

}

}

} } void bfs(int i){ queue<int> q; q.push(i); visited[i] = true; while(q.empty() != true){ int current = q.front(); q.pop(); cout << current << " "; for(auto j = graph[current].begin(); j != graph[current].end();j++){ if(visited[\*j] == false){ q.push(\*j); visited[\*j] = true;

}

}

} } void parallel\_bfs(int i){ queue<int> q; q.push(i); visited[i] = true; while(q.empty() != true){ int current = q.front(); cout << current << " "; #pragma omp critical q.pop();

#pragma omp parallel for for(auto j = graph[current].begin(); j != graph[current].end();j++){ if(visited[\*j] == false){ #pragma omp critical q.push(\*j); visited[\*j] = true;

}

}

}

} }; int main(int argc, char const \*argv[])

{

Graph g;

cout << "Adjacency List:\n"; g.printGraph();

g.initialize\_visited(); cout << "Depth First Search: \n"; auto start = chrono::high\_resolution\_clock::now(); g.dfs(0); cout << endl; auto end = chrono::high\_resolution\_clock::now(); cout << "Time taken: " << chrono::duration\_cast<chrono::microseconds>(end

* start).count() << " microseconds" << endl; cout << "Parallel Depth First Search: \n"; g.initialize\_visited(); start = chrono::high\_resolution\_clock::now(); g.parallel\_dfs(0); cout << endl; end = chrono::high\_resolution\_clock::now(); cout << "Time taken: "<< chrono::duration\_cast<chrono::microseconds>(end
* start).count() << " microseconds" << endl; start = chrono::high\_resolution\_clock::now(); cout << "Breadth First Search: \n"; g.initialize\_visited();

g.bfs(0); cout << endl; end = chrono::high\_resolution\_clock::now(); cout << "Time taken: "<< chrono::duration\_cast<chrono::microseconds>(end

* start).count() << " microseconds" << endl; start = chrono::high\_resolution\_clock::now(); cout << "Parallel Breadth First Search: \n"; g.initialize\_visited();

g.parallel\_bfs(0);

cout << endl; end = chrono::high\_resolution\_clock::now(); cout << "Time taken: " << chrono::duration\_cast<chrono::microseconds>(end

* start).count() << " microseconds" << endl; return 0;

}