```
#include<iostream>
#include<omp.h>
#include<bits/stdc++.h>
using namespace std;
class Graph{
   public:
      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;
      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(auto j = 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;
}

Adjacency List:
0 -> 1
1 -> 0 2 3
```

2 -> 1 4 5
3 -> 1 4
4 -> 2 3
5 -> 2
Depth First Search:
0 1 3 4 2 5
Time taken: 11 microseconds
Parallel Depth First Search:
0 1 3 4 2 5
Time taken: 8 microseconds
Breadth First Search:
0 1 2 3 4 5
Time taken: 13 microseconds
Parallel Breadth First Search:
0 1 2 3 4 5
Time taken: 12 microseconds