```
#include <iostream> // --> Includes input-output stream for console operations

#include <vector> // --> Includes the vector container from STL

#include <stack> // --> Includes the stack container from STL

#include <omp.h> // --> Includes OpenMP library for parallel processing


using namespace std; // --> Uses the standard namespace to avoid std:: prefix


const int MAX = 100000; // --> Sets the maximum number of nodes in the graph

vector<int> graph[MAX]; // --> Adjacency list to represent the graph

bool visited[MAX]; // --> Array to track visited nodes in DFS

omp_lock_t lock[MAX]; // --> Array of locks for thread-safe node access


// Function to perform DFS

void dfs(int start_node) // --> DFS function starting from a given node

{

  stack<int> s; // --> Stack to manage DFS traversal

  s.push(start_node); // --> Push the start node onto the stack


  while (!s.empty()) // --> Loop until the stack is empty

  {

    int curr_node = s.top(); // --> Get the top node from the stack

    s.pop(); // --> Remove the top node from the stack


    omp_set_lock(&lock[curr_node]); // --> Lock the current node to avoid race condition

    if (!visited[curr_node]) // --> Check if current node is not visited

    {

      visited[curr_node] = true; // --> Mark current node as visited

      cout << curr_node << " "; // --> Print the visited node

    }

    omp_unset_lock(&lock[curr_node]); // --> Unlock the current node
```

```
    #pragma omp parallel for shared(s) // --> Parallel loop to explore neighbors
    for (int i = 0; i < graph[curr_node].size(); i++) // --> Iterate over all adjacent nodes
    {
      int adj_node = graph[curr_node][i]; // --> Get adjacent node

      omp_set_lock(&lock[adj_node]); // --> Lock adjacent node before accessing
      if (!visited[adj_node]) // --> If adjacent node is not visited
      {
        #pragma omp critical // --> Ensure only one thread pushes to stack at a time
        {
          s.push(adj_node); // --> Push adjacent node to the stack
        }
      }
      omp_unset_lock(&lock[adj_node]); // --> Unlock adjacent node
    }
  }
}

int main() // --> Main function
{
  int n, m, start_node; // --> Variables for number of nodes, edges, and start node

  cout << "Enter number of nodes, edges, and the starting node: "; // --> Prompt user for input
  cin >> n >> m >> start_node; // --> Read number of nodes, edges, and starting node

  cout << "Enter pairs of connected edges (u v):\n"; // --> Prompt user for edge inputs
  for (int i = 0; i < m; i++) // --> Loop through each edge
  {
    int u, v; // --> Variables for edge endpoints
    cin >> u >> v; // --> Read an edge between nodes u and v
```

```
        graph[u].push_back(v); // --> Add v to u's adjacency list
        graph[v].push_back(u); // --> Add u to v's adjacency list (undirected graph)
    }

    #pragma omp parallel for // --> Parallel loop to initialize visited and locks
    for (int i = 0; i < n; i++) // --> Loop through all nodes
    {
        visited[i] = false; // --> Mark all nodes as unvisited
        omp_init_lock(&lock[i]); // --> Initialize a lock for each node
    }

    cout << "\nDFS Traversal Order:\n"; // --> Print DFS header
    dfs(start_node); // --> Call DFS function with starting node
    cout << endl; // --> Print newline after traversal

    for (int i = 0; i < n; i++) // --> Loop through all nodes
    {
        omp_destroy_lock(&lock[i]); // --> Destroy all locks after use
    }

    return 0; // --> Exit program
}

// Run Commands:
// g++ -fopenmp -o parallel_bfs 1_Breadth_First_Search.cpp // --> Compile code with OpenMP support
// .\parallel_bfs // --> Run the compiled program

// Output Example:
// Enter number of nodes, edges, and the starting node: 6 5 0 // --> Sample input
// Enter pairs of connected edges (u v): // --> Sample input prompt
```

```
// 0 1
// 0 2
// 1 3
// 1 4
// 2 5

// DFS Traversal Order:
// 0 1 4 3 2 5 // --> Sample DFS output
```