

## Assignment 1(b)

**Title: Parallel Depth First Search based on existing algorithms using OpenMP**

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

using namespace std;

const int MAX = 100000;
vector<int> graph[MAX];
bool visited[MAX];

void dfs(int node) {
    stack<int> s;
    s.push(node);

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

        if (!visited[curr_node]) {
            visited[curr_node] = true;

            if (visited[curr_node]) {
                cout << curr_node << " ";
            }

            #pragma omp parallel for
            for (int i = 0; i < graph[curr_node].size(); i++) {
                int adj_node = graph[curr_node][i];
                if (!visited[adj_node]) {
                    s.push(adj_node);
                }
            }
        }
    }
}

int main() {
    int n, m, start_node;
```

```

        cout << "Enter No of Node,Edges,and start node:" ;
        cin >> n >> m >> start_node;
//n: node,m:edges

cout << "Enter Pair of edges:" ;
    for (int i = 0; i < m; i++) {
        int u, v;

        cin >> u >> v;
//u and v: Pair of edges
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

#pragma omp parallel for
    for (int i = 0; i < n; i++) {
        visited[i] = false;
    }

    dfs(start_node);

/*    for (int i = 0; i < n; i++) {
        if (visited[i]) {
            cout << i << " ";
        }
    }*/

    return 0;
}

```

Explanation:

Let's go through the code step by step:

1. We start by including the necessary headers and declaring some global variables, such as the graph adjacency list, an array to keep track of visited nodes, and a maximum limit for the number of nodes in the graph.
2. Next, we define a function called `dfs()` which takes a starting node as input and performs the depth-first search algorithm. We use a stack to keep track of the nodes to be visited. The algorithm works as follows:
  - We push the starting node onto the stack.
  - While the stack is not empty, we pop the top node from the stack.

- If the current node has not been visited, we mark it as visited and explore all its neighbors that have not been visited yet by adding them to the stack.
3. In the main() function, we read the input values: the number of nodes in the graph, the number of edges, and the starting node.
  4. We then read the edge information and build the adjacency list of the graph.
  5. We initialize the visited array to false for all nodes in the graph using OpenMP's parallel for construct to allow multiple threads to set the array values in parallel.
  6. We call the dfs() function with the starting node, which performs the depth-first search algorithm in parallel.
  7. Finally, we print out the list of visited nodes in the order they were visited.

The parallelization of the DFS algorithm is achieved by using the OpenMP parallel for construct inside the dfs() function to explore the neighbours of each node in parallel. This allows multiple threads to work on different parts of the graph simultaneously, which can lead to significant performance gains on multi-core CPUs

### step-by-step explanation of how this program works:

#### dfs function

This function performs a parallel depth-first search starting from a given node. It takes the node number as a parameter and modifies the global visited array to keep track of which nodes have been visited.

1. It starts by creating an empty stack and pushing the starting node onto it.

c++

Copy code

```
void dfs(int node) {  
    stack<int> s;  
    s.push(node);
```

2. It enters a loop that continues until the stack is empty.

c++

Copy code

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

3. For each node that is popped off the stack, it checks if that node has already been visited. If not, it sets the visited flag for that node to true and adds its unvisited neighbors to the stack in parallel.

```
c++ Copy code

    if (!visited[curr_node]) {
        visited[curr_node] = true;

        #pragma omp parallel for
        for (int i = 0; i < graph[curr_node].size(); i++) {
            int adj_node = graph[curr_node][i];
            if (!visited[adj_node]) {
                s.push(adj_node);
            }
        }
    }
}
}
```

Note that the `#pragma omp parallel for` directive is used to parallelize the loop that adds neighboring nodes to the stack. This can speed up the execution of the program on multi-core machines.

## main function

This function reads in the input, initializes the visited array, calls the dfs function, and prints out the list of visited nodes.

1. It reads in the number of nodes, the number of edges, and the starting node from standard input.

```
c++ Copy code

int main() {
    int n, m, start_node;
    cin >> n >> m >> start_node;
```

2. It reads in the edge list and builds the adjacency list representation of the graph. Note that each edge is added to both the u node's and v node's adjacency list to represent an undirected graph.

```
c++ Copy code

    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }
```

3. It initializes the visited array to false in parallel.

```
c++ Copy code  
  
#pragma omp parallel for  
for (int i = 0; i < n; i++) {  
    visited[i] = false;  
}
```

4. It calls the dfs function with the starting node as the argument.

```
c++ Copy code  
  
dfs(start_node);
```

5. It prints out the list of visited nodes in ascending order.

```
c++ Copy code  
  
for (int i = 0; i < n; i++) {  
    if (visited[i]) {  
        cout << i << " ";  
    }  
}  
  
return 0;  
}
```


Output:

here's an example input and output for a small graph with 6 nodes and 5 edges:

Input:

```
Copy code  
  
6 7 0  
0 1  
0 2  
1 3  
2 4  
2 5  
4 5  
5 3
```

Output:

 Copy code

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