Design and implement parallel DFS and DFS based on existing algorithm using open Map. Give the tree or an undirected doc for BFS and DFS

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

using namespace std;

class Graph {
    int V;
    vector<vector<int>> adj;

public:
    Graph(int V) {
        this->V = V;
        adj.resize(V);
    }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u); // undirected graph
    }

    void parallelBFS(int start) {
        vector<bool> visited(V, false);
        queue<int> q;

        visited[start] = true;
        q.push(start);

        cout << "Parallel BFS starting from vertex " << start << ": ";

        while (!q.empty()) {
            int level_size = q.size();
            vector<int> current_level;

            // Collect current level nodes
            for (int i = 0; i < level_size; ++i) {
                int curr = q.front();
                q.pop();
                current_level.push_back(curr);
                cout << curr << " ";
            }
```

```
        vector<int> to_add;
        #pragma omp parallel for
        for (int i = 0; i < current_level.size(); ++i) {
            int curr = current_level[i];
            for (int j = 0; j < adj[curr].size(); ++j) {
                int neighbor = adj[curr][j];
                if (!visited[neighbor]) {
                    bool expected = false;
                    #pragma omp critical
                    {
                        if (!visited[neighbor]) {
                            visited[neighbor] = true;
                            to_add.push_back(neighbor);
                        }
                    }
                }
            }
        }

        // Add next level nodes to the queue
        for (int node : to_add) {
            q.push(node);
        }
    }
    cout << endl;
}

void parallelDFSUtil(int start, vector<bool>& visited) {
    stack<int> s;
    s.push(start);

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

        if (!visited[curr]) {
            visited[curr] = true;
            cout << curr << " ";
        }

        vector<int> neighbors;
        #pragma omp parallel for
        for (int i = 0; i < adj[curr].size(); ++i) {
            int neighbor = adj[curr][i];
            if (!visited[neighbor]) {
```

```
                    #pragma omp critical
                    {
                        if (!visited[neighbor]) {
                            neighbors.push_back(neighbor);
                        }
                    }
                }
            }

            // Push neighbors after the parallel region to avoid stack
corruption
            for (int i = neighbors.size() - 1; i >= 0; --i) {
                s.push(neighbors[i]);
            }
        }
    }

    void parallelDFS(int start) {
        vector<bool> visited(V, false);
        cout << "Parallel DFS starting from vertex " << start << ": ";
        parallelDFSUtil(start, visited);
        cout << endl;
    }
};

int main() {
    int V = 6;
    Graph g(V);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 5);

    g.parallelBFS(0);
    g.parallelDFS(0);

    return 0;
}
```