```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <unordered_set>
#include <omp.h>
#include <chrono>
using namespace std;
#define int long long

class Graph {
private:
    int vertices;
    vector<vector<int>> adjacencyList;

public:
    Graph(int v) : vertices(v) {
        adjacencyList.resize(v);
    }

    void addEdge(int v, int w) {
        adjacencyList[v].push_back(w);
        adjacencyList[w].push_back(v);
    }

    void sequentialBFS(int startVertex) {
        vector<bool> visited(vertices, false);
        queue<int> queue;

        visited[startVertex] = true;
        queue.push(startVertex);

        while (!queue.empty()) {
            int currentVertex = queue.front();
            queue.pop();

            for (int adjacentVertex : adjacencyList[currentVertex]) {
                if (!visited[adjacentVertex]) {
                    visited[adjacentVertex] = true;
                    queue.push(adjacentVertex);
                }
            }
        }
    }
```

```
    void parallelBFS(int startVertex) {
        vector<bool> visited(vertices, false);
        vector<int> frontier;
        vector<int> next_frontier;

        visited[startVertex] = true;
        frontier.push_back(startVertex);

        while (!frontier.empty()) {
            next_frontier.clear();

            #pragma omp parallel
            {
                vector<int> local_frontier;

                #pragma omp for nowait
                for (size_t i = 0; i < frontier.size(); i++) {
                    int currentVertex = frontier[i];

                    for (int adjacentVertex : adjacencyList[currentVertex]) {
                        bool was_visited = false;

                        #pragma omp critical
                        {
                            if (!visited[adjacentVertex]) {
                                visited[adjacentVertex] = true;
                                local_frontier.push_back(adjacentVertex);
                            }
                        }
                    }
                }

                #pragma omp critical
                {
                    next_frontier.insert(next_frontier.end(),
local_frontier.begin(), local_frontier.end());
                }
            }

            frontier.swap(next_frontier);
        }
    }

    void sequentialDFS(int startVertex) {
        vector<bool> visited(vertices, false);
```

```
    stack<int> stack;

    stack.push(startVertex);

    while (!stack.empty()) {
        int currentVertex = stack.top();
        stack.pop();

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

            for (int adjacentVertex : adjacencyList[currentVertex]) {
                if (!visited[adjacentVertex]) {
                    stack.push(adjacentVertex);
                }
            }
        }
    }
}

void parallelDFS(int startVertex) {
    vector<bool> visited(vertices, false);
    stack<int> stack;

    visited[startVertex] = true;
    stack.push(startVertex);

    while (!stack.empty()) {
        vector<int> current_level;

        while (!stack.empty()) {
            current_level.push_back(stack.top());
            stack.pop();
        }

        #pragma omp parallel
        {
            vector<int> local_stack;

            #pragma omp for nowait
            for (size_t i = 0; i < current_level.size(); i++) {
                int currentVertex = current_level[i];

                for (int adjacentVertex : adjacencyList[currentVertex]) {
                    #pragma omp critical
```

```
                        {
                            if (!visited[adjacentVertex]) {
                                visited[adjacentVertex] = true;
                                local_stack.push_back(adjacentVertex);
                            }
                        }
                    }
                }

                #pragma omp critical
                {
                    for (int vertex : local_stack) {
                        stack.push(vertex);
                    }
                }
            }
        }
    }
};

int32_t main() {
    int numVertices = 2e7;
    int numEdges = 2e7;
    int startVertex = 0;

    Graph g(numVertices);

    for (int i = 0; i < numEdges; i++) {
        int v = rand() % numVertices;
        int w = rand() % numVertices;
        g.addEdge(v, w);
    }

    auto start_time = chrono::high_resolution_clock::now();
    g.sequentialBFS(startVertex);
    auto end_time = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::milliseconds>(end_time -
    start_time);
    cout << "Sequential BFS execution time: " << duration.count() << " ms\n";

    start_time = chrono::high_resolution_clock::now();
    g.parallelBFS(startVertex);
    end_time = chrono::high_resolution_clock::now();
    duration = chrono::duration_cast<chrono::milliseconds>(end_time -
    start_time);
```

```
    cout << "Parallel BFS execution time: " << duration.count() << " ms\n";

    start_time = chrono::high_resolution_clock::now();
    g.sequentialDFS(startVertex);
    end_time = chrono::high_resolution_clock::now();
    duration = chrono::duration_cast<chrono::milliseconds>(end_time -
    start_time);
    cout << "Sequential DFS execution time: " << duration.count() << " ms\n";

    start_time = chrono::high_resolution_clock::now();
    g.parallelDFS(startVertex);
    end_time = chrono::high_resolution_clock::now();
    duration = chrono::duration_cast<chrono::milliseconds>(end_time -
    start_time);
    cout << "Parallel DFS execution time: " << duration.count() << " ms\n";

    return 0;
}
```

```
-----------------------
Output
----------------------
Sequential BFS execution time: 10371 ms
Parallel BFS execution time: 6481 ms
Sequential DFS execution time: 11613 ms
Parallel DFS execution time: 8515 ms
```