```cpp
// GRAPH.HPP

#pragma once

#include <omp.h>
#include <fstream>
#include <functional>
#include <iostream>
#include <queue>
#include <sstream>
#include <string>
#include <tuple>
#include <vector>

// Generic representation of a graph implemented with an adjacency matrix
struct Graph {
    using Node = int;

    int task_threshold = 60;
    int max_depth_rdfs = 10'000;

    std::vector<std::vector<int>> adj_matrix;

    // Returns if an edge between two nodes exists
    bool edge_exists(Node n1, Node n2) { return adj_matrix[n1][n2] > 0; }

    // Returns the number of nodes of the graph
    int n_nodes() { return adj_matrix.size(); }

    // Returns the number of nodes of the graph
    int size() { return n_nodes(); }

    // Sequential implementation of the iterative version of depth first search.
    void dfs(Node src, std::vector<int>& visited) {
        std::vector<Node> queue{src};
        while (!queue.empty()) {
            Node node = queue.back();
            queue.pop_back();

            if (!visited[node]) {
                visited[node] = true;
                for (int next_node = 0; next_node < n_nodes(); next_node++)
                    if (edge_exists(node, next_node) && !visited[next_node])
                        queue.push_back(next_node);
            }
        }
    }

    // Sequential implementation of the recursive version of depth first search.
    void rdfs(Node src, std::vector<int>& visited, int depth = 0) {
        visited[src] = true;
        for (int node = 0; node < n_nodes(); node++) {
            if (edge_exists(src, node) && !visited[node]) {
                // Limit recursion depth to avoid stack overflow error
                if (depth <= max_depth_rdfs)
                    rdfs(node, visited, depth + 1);
                else
                    dfs(node, visited);
            }
        }
    }

    // Parallel implementation of the iterative version of depth first search.
    // The general idea is that the main thread extracts the last node from the
    // queue and check the neighbors of the node in parallel. Each of these threads
    // have a private queue where neighbors still not visited are added. At the end,
    // threads concatenate their private queue to the main queue.
    void p_dfs(Node src, std::vector<int>& visited) {
        std::vector<Node> queue{src};
        while (!queue.empty()) {
            Node node = queue.back();
            queue.pop_back();

            if (!visited[node]) {
```

```
                visited[node] = true;

#pragma omp parallel shared(queue, visited)
                {
                    // Every thread has a private_queue to avoid continuous lock
                    // checking to update the main one
                    std::vector<Node> private_queue;

#pragma omp for nowait schedule(static)
                    for (int next_node = 0; next_node < n_nodes(); next_node++)
                        if (edge_exists(node, next_node) && !visited[next_node])
                            private_queue.push_back(next_node);

// Append at the end of master queue the private queue of the thread
#pragma omp critical(queue_update)
                    queue.insert(queue.end(), private_queue.begin(), private_queue.end());
                }
            }
        }
    }

    // Parallel implementation of the iterative version of depth first search.
    // The general idea is that the main thread extracts the last node from the
    // queue and check the neighbors of the node in parallel. Each of these
    // threads have a private queue where neighbors still not visited are added.
    // At the end, threads concatenate their private queue to the main queue.
    // **Important**: this version implements node level locks.
    void p_dfs_with_locks(Node src, std::vector<int>& visited,
                          std::vector<omp_lock_t>& node_locks) {
        // Note: Since C++11, different elements in the same container can be
        // modified concurrently by different threads, except for the elements
        // of std::vector<bool>
        //
        // Possible explanation of why here:
        // https://stackoverflow.com/a/33617530/2691946
        // This is why we use a vector of int.

        std::vector<Node> queue{src};
        while (!queue.empty()) {
            Node node = queue.back();
            queue.pop_back();

            bool already_visited = atomic_test_visited(node, visited, &node_locks[node]);

            if (!already_visited) {
                atomic_set_visited(node, visited, &node_locks[node]);

#pragma omp parallel shared(queue, visited)
                {
                    // Every thread has a private queue to avoid continuos lock
                    // checking to update the main one
                    std::vector<Node> private_queue;

#pragma omp for nowait
                    for (int next_node = 0; next_node < n_nodes(); next_node++) {
                        // Check if the edge exists is a non-blocking request,
                        // so it's better to do it before than checking if the
                        // node is already visited
                        if (edge_exists(node, next_node)) {
                            if (atomic_test_visited(next_node, visited, &node_locks[next_node])) {
                                private_queue.push_back(next_node);
                            }
                        }
                    }

// Append at the end of master queue the private queue of the thread
#pragma omp critical(queue_update)
                    queue.insert(queue.end(), private_queue.begin(), private_queue.end());
                }
            }
        }
    }

    // Parallel implementation of the recursive version of depth first search.
    // This version automatically initialize locks
```

```cpp
    void p_rdfs(Node src, std::vector<int>& visited) {
        // Initialize locks
        std::vector<omp_lock_t> node_locks;
        node_locks.reserve(size());

        for (int node = 0; node < n_nodes(); node++) {
            omp_lock_t lock;
            node_locks[node] = lock;
            omp_init_lock(&(node_locks[node]));
        }

#pragma omp parallel shared(src, visited, node_locks)
#pragma omp single
        p_rdfs(src, visited, node_locks);

        // Destory locks
        for (int node = 0; node < n_nodes(); node++) omp_destroy_lock(&(node_locks[node]));
    }

    // Parallel implementation of the recursive version of depth first search,
    // full version with locks
    void p_rdfs(Node src, std::vector<int>& visited, std::vector<omp_lock_t>& node_locks,
                int depth = 0) {
        atomic_set_visited(src, visited, &node_locks[src]);

        // Number of tasks in parallel executing at this level of depth
        int task_count = 0;

        for (int node = 0; node < n_nodes(); node++) {
            if (edge_exists(src, node) && !atomic_test_visited(node, visited, &node_locks[node])) {
                // Limit the number of parallel tasks both horizontally (for
                // checking neighbors) and vertically (between recursive
                // calls).
                //
                // Fallback to iterative version if one of these limits are
                // reached
                if (depth <= max_depth_rdfs && task_count <= task_threshold) {
                    task_count++;

#pragma omp task untied default(shared) firstprivate(node)
                    {
                        p_rdfs(node, visited, node_locks, depth + 1);
                        task_count--;
                    }

                } else {
                    // Fallback to parallel iterative version
                    p_dfs_with_locks(node, visited, node_locks);
                }
            }
        }

#pragma omp taskwait
    }

    // Serial implementation of the Dijkstra algorithm without early exit condition.
    //
    // Note: It does not use a priority queue.
    std::pair<std::vector<Node>, std::vector<Node>> dijkstra(Node src) {
        std::vector<Node> queue;
        queue.push_back(src);

        std::vector<Node> came_from(size(), -1);
        std::vector<Node> cost_so_far(size(), -1);

        came_from[src] = src;
        cost_so_far[src] = 0;

        while (!queue.empty()) {
            Node current = queue.back();
            queue.pop_back();

            for (int next = 0; next < n_nodes(); next++) {
                if (edge_exists(current, next)) {
                    int new_cost = cost_so_far[current] + adj_matrix[current][next];
```

```cpp

                    if (cost_so_far[next] == -1 || new_cost < cost_so_far[next]) {
                        cost_so_far[next] = new_cost;
                        queue.push_back(next);
                        came_from[next] = current;
                    }
                }
            }
        }

        return std::make_pair(came_from, cost_so_far);
    }

    inline std::vector<omp_lock_t> initialize_locks() {
        std::vector<omp_lock_t> node_locks;
        node_locks.reserve(n_nodes());

        for (int node = 0; node < n_nodes(); node++) {
            omp_lock_t lock;
            node_locks[node] = lock;
            omp_init_lock(&(node_locks[node]));
        }

        return node_locks;
    }

    // Parallel implementation of the Dijkstra algorithm without early exit
    // condition using node level locks. As expected, it performs very poorly
    //
    // Note: It does not use a priority queue.
    std::pair<std::vector<Node>, std::vector<Node>> p_dijkstra(Node src) {
        std::vector<Node> queue;
        queue.push_back(src);

        std::vector<Node> came_from(size(), -1);
        std::vector<Node> cost_so_far(size(), -1);

        came_from[src] = src;
        cost_so_far[src] = 0;

        auto node_locks = initialize_locks();

        while (!queue.empty()) {
            Node current = queue.back();
            queue.pop_back();

#pragma omp parallel shared(queue, node_locks)
#pragma omp for
            for (int next = 0; next < n_nodes(); next++) {
                if (edge_exists(current, next)) {
                    omp_set_lock(&node_locks[current]);
                    auto cost_so_far_current = cost_so_far[current];
                    omp_unset_lock(&node_locks[current]);

                    int new_cost = cost_so_far_current + adj_matrix[current][next];

                    omp_set_lock(&node_locks[next]);
                    auto cost_so_far_next = cost_so_far[next];
                    omp_unset_lock(&node_locks[next]);

                    if (cost_so_far_next == -1 || new_cost < cost_so_far_next) {
                        omp_set_lock(&node_locks[next]);
                        cost_so_far[next] = new_cost;
                        came_from[next] = current;
                        omp_unset_lock(&node_locks[next]);

#pragma omp critical(queue_update)
                        queue.push_back(next);
                    }
                }
            }
        }

        // Destory locks
        for (int node = 0; node < n_nodes(); node++) omp_destroy_lock(&(node_locks[node]));
```

```cpp

        return std::make_pair(came_from, cost_so_far);
    }

    // Reconstruct path from the destination to the source
    std::vector<Node> reconstruct_path(Node src, Node dst, std::vector<Node> origins) {
        auto current_node = dst;
        std::vector<Node> path;

        while (current_node != src) {
            path.push_back(current_node);
            current_node = origins.at(current_node);
        }

        path.push_back(src);
        reverse(path.begin(), path.end());

        return path;
    }

   private:
    // Return true if a node is already visited using a node level lock
    inline bool atomic_test_visited(Node node, const std::vector<int>& visited, omp_lock_t* lock) {
        omp_set_lock(lock);
        bool already_visited = visited.at(node);
        omp_unset_lock(lock);

        return already_visited;
    }

    // Set that a node is already visited using a node level lock
    inline void atomic_set_visited(Node node, std::vector<int>& visited, omp_lock_t* lock) {
        omp_set_lock(lock);
        visited[node] = true;
        omp_unset_lock(lock);
    }
};

// Import graph from a file
Graph import_graph(std::string& path) {
    Graph graph;

    std::ifstream file(path);
    if (!file.is_open()) {
        throw std::invalid_argument("Input file does not exist or is not readable.");
    }

    std::string line;

    // Read one line at a time into the variable line
    while (getline(file, line)) {
        std::vector<int> lineData;
        std::stringstream lineStream(line);

        // Read an integer at a time from the line
        int value;
        while (lineStream >> value) lineData.push_back(value);

        lineData.shrink_to_fit();  // Usefull?
        graph.adj_matrix.push_back(lineData);
    }

    graph.adj_matrix.shrink_to_fit();

    return graph;
}
```

```
// MAIN.CPP

#include <array>
#include <chrono>
#include <functional>
#include <string>
#include <vector>

#include "graph.hpp"

using std::chrono::duration_cast;
using std::chrono::high_resolution_clock;
using std::chrono::milliseconds;

std::string bench_traverse(std::function<void()> traverse_fn) {
    auto start = high_resolution_clock::now();
    traverse_fn();
    auto stop = high_resolution_clock::now();

    // Subtract stop and start timepoints and cast it to required unit.
    // Predefined units are nanoseconds, microseconds, milliseconds, seconds,
    // minutes, hours. Use duration_cast() function.
    auto duration = duration_cast<milliseconds>(stop - start);

    // To get the value of duration use the count() member function on the
    // duration object
    return std::to_string(duration.count());
}

void full_bench(Graph& graph) {
    int num_test = 1;
    std::array<int, 6> num_threads{{1, 2, 4, 8, 16, 32}};

    std::vector<Graph::Node> visited(graph.size(), false);
    Graph::Node src = 0;

    // Explicitly disable dynamic teams as we are going to set a fixed number of
    // threads
    omp_set_dynamic(0);

    // TODO: find a better way to avoid code repetition

    std::cout << "Number of nodes: " << graph.size() << "\n\n";

    for (int i = 0; i < num_test; i++) {
        std::cout << "\t"
                  << "Execution " << i + 1 << std::endl;

        std::cout << "Sequential iterative DFS: "
                  << bench_traverse([&] { graph.dfs(src, visited); }) << "ms\n";

        // We cannot pass a copy of the vector, so we "reset" it every time
        std::fill(visited.begin(), visited.end(), false);

        std::cout << "Sequential recursive DFS: "
                  << bench_traverse([&]() { graph.rdfs(src, visited); }) << "ms\n";

        std::cout << "Dijkstra: " << bench_traverse([&] { graph.dijkstra(0); }) << "ms\n";

        for (const auto n : num_threads) {
            std::fill(visited.begin(), visited.end(), false);

            std::cout << "Using " << n << " threads..." << std::endl;

            // Set to use N threads
            omp_set_num_threads(n);

            // Should we change also this?
            // graph.task_threshold = n;

            std::cout << "Parallel iterative DFS: "
                      << bench_traverse([&] { graph.p_dfs(src, visited); }) << "ms\n";

            std::fill(visited.begin(), visited.end(), false);
```

```
            std::cout << "Parallel recursive DFS: "
                      << bench_traverse([&] { graph.p_rdfs(src, visited); }) << "ms\n";

            std::cout << "Parallel Dijkstra: " << bench_traverse([&] { graph.p_dijkstra(0); })
                      << "ms\n";
        }

        std::fill(visited.begin(), visited.end(), false);

        std::cout << std::endl;
    }
}

int main(int argc, const char** argv) {
    // TODO: Add a CLI? Also, we should accept more input files and process them separately
    if (argc < 2) {
        std::cout << "Input file not specified.\n";
        return 1;
    }

    std::string file_path = argv[1];

    auto graph = import_graph(file_path);

    full_bench(graph);

    return 0;
}

/*

OUTPUT:

Number of nodes: 1000

        Execution 1
Sequential iterative DFS: 64ms
Sequential recursive DFS: 39ms
Dijkstra: 72ms
Using 1 threads...
Parallel iterative DFS: 61ms
Parallel recursive DFS: 63ms
Parallel Dijkstra: 79ms
Using 2 threads...
Parallel iterative DFS: 44ms
Parallel recursive DFS: 37ms
Parallel Dijkstra: 89ms
Using 4 threads...
Parallel iterative DFS: 37ms
Parallel recursive DFS: 25ms
Parallel Dijkstra: 256ms
Using 8 threads...
Parallel iterative DFS: 36ms
Parallel recursive DFS: 15ms
Parallel Dijkstra: 257ms
Using 16 threads...
Parallel iterative DFS: 80ms
Parallel recursive DFS: 15ms
Parallel Dijkstra: 532ms
Using 32 threads...
Parallel iterative DFS: 186ms
Parallel recursive DFS: 21ms
Parallel Dijkstra: 481ms
*/
```