

```

1 // GRAPH.HPP
2
3 #pragma once
4
5 #include <omp.h>
6 #include <fstream>
7 #include <functional>
8 #include <iostream>
9 #include <queue>
10 #include <sstream>
11 #include <string>
12 #include <tuple>
13 #include <vector>
14
15 // Generic representation of a graph implemented with an adjacency matrix
16 struct Graph {
17     using Node = int;
18
19     int task_threshold = 60;
20     int max_depth_rdfs = 10'000;
21
22     std::vector<std::vector<int>>> adj_matrix;
23
24     // Returns if an edge between two nodes exists
25     bool edge_exists(Node n1, Node n2) { return adj_matrix[n1][n2] > 0; }
26
27     // Returns the number of nodes of the graph
28     int n_nodes() { return adj_matrix.size(); }
29
30     // Returns the number of nodes of the graph
31     int size() { return n_nodes(); }
32
33     // Sequential implementation of the iterative version of depth first search.
34     void dfs(Node src, std::vector<int>& visited) {
35         std::vector<Node> queue{src};
36         while (!queue.empty()) {
37             Node node = queue.back();
38             queue.pop_back();
39
40             if (!visited[node]) {
41                 visited[node] = true;
42                 for (int next_node = 0; next_node < n_nodes(); next_node++)
43                     if (edge_exists(node, next_node) && !visited[next_node])
44                         queue.push_back(next_node);
45             }
46         }
47     }
48
49     // Sequential implementation of the recursive version of depth first search.
50     void rdfs(Node src, std::vector<int>& visited, int depth = 0) {
51         visited[src] = true;
52         for (int node = 0; node < n_nodes(); node++) {
53             if (edge_exists(src, node) && !visited[node]) {
54                 // Limit recursion depth to avoid stack overflow error
55                 if (depth <= max_depth_rdfs)
56                     rdfs(node, visited, depth + 1);
57                 else
58                     dfs(node, visited);
59             }
60         }
61     }
62
63     // Parallel implementation of the iterative version of depth first search.
64     // The general idea is that the main thread extracts the last node from the
65     // queue and check the neighbors of the node in parallel. Each of these threads
66     // have a private queue where neighbors still not visited are added. At the end,
67     // threads concatenate their private queue to the main queue.
68     void p_dfs(Node src, std::vector<int>& visited) {
69         std::vector<Node> queue{src};
70         while (!queue.empty()) {
71             Node node = queue.back();
72             queue.pop_back();
73
74             if (!visited[node]) {

```

```

75         visited[node] = true;
76
77 #pragma omp parallel shared(queue, visited)
78     {
79         // Every thread has a private_queue to avoid continuous lock
80         // checking to update the main one
81         std::vector<Node> private_queue;
82
83 #pragma omp for nowait schedule(static)
84         for (int next_node = 0; next_node < n_nodes(); next_node++)
85             if (edge_exists(node, next_node) && !visited[next_node])
86                 private_queue.push_back(next_node);
87
88 // Append at the end of master queue the private queue of the thread
89 #pragma omp critical(queue_update)
90         queue.insert(queue.end(), private_queue.begin(), private_queue.end());
91     }
92 }
93
94 }
95
96 // Parallel implementation of the iterative version of depth first search.
97 // The general idea is that the main thread extracts the last node from the
98 // queue and check the neighbors of the node in parallel. Each of these
99 // threads have a private queue where neighbors still not visited are added.
100 // At the end, threads concatenate their private queue to the main queue.
101 // **Important**: this version implements node level locks.
102 void p_dfs_with_locks(Node src, std::vector<int>& visited,
103                      std::vector<omp_lock_t>& node_locks) {
104     // Note: Since C++11, different elements in the same container can be
105     // modified concurrently by different threads, except for the elements
106     // of std::vector<bool>
107     //
108     // Possible explanation of why here:
109     // https://stackoverflow.com/a/33617530/2691946
110     // This is why we use a vector of int.
111
112     std::vector<Node> queue{src};
113     while (!queue.empty()) {
114         Node node = queue.back();
115         queue.pop_back();
116
117         bool already_visited = atomic_test_visited(node, visited, &node_locks[node]);
118
119         if (!already_visited) {
120             atomic_set_visited(node, visited, &node_locks[node]);
121
122 #pragma omp parallel shared(queue, visited)
123             {
124                 // Every thread has a private queue to avoid continuous lock
125                 // checking to update the main one
126                 std::vector<Node> private_queue;
127
128 #pragma omp for nowait
129                 for (int next_node = 0; next_node < n_nodes(); next_node++) {
130                     // Check if the edge exists is a non-blocking request,
131                     // so it's better to do it before than checking if the
132                     // node is already visited
133                     if (edge_exists(node, next_node)) {
134                         if (atomic_test_visited(next_node, visited, &node_locks[next_node])) {
135                             private_queue.push_back(next_node);
136                         }
137                     }
138                 }
139
140 // Append at the end of master queue the private queue of the thread
141 #pragma omp critical(queue_update)
142                 queue.insert(queue.end(), private_queue.begin(), private_queue.end());
143             }
144         }
145     }
146 }
147
148 // Parallel implementation of the recursive version of depth first search.
149 // This version automatically initialize locks

```

```

150 void p_rdfs(Node src, std::vector<int>& visited) {
151     // Initialize locks
152     std::vector<omp_lock_t> node_locks;
153     node_locks.reserve(size());
154
155     for (int node = 0; node < n_nodes(); node++) {
156         omp_lock_t lock;
157         node_locks[node] = lock;
158         omp_init_lock(&(node_locks[node]));
159     }
160
161 #pragma omp parallel shared(src, visited, node_locks)
162 #pragma omp single
163     p_rdfs(src, visited, node_locks);
164
165     // Destroy locks
166     for (int node = 0; node < n_nodes(); node++) omp_destroy_lock(&(node_locks[node]));
167 }
168
169 // Parallel implementation of the recursive version of depth first search,
170 // full version with locks
171 void p_rdfs(Node src, std::vector<int>& visited, std::vector<omp_lock_t>& node_locks,
172             int depth = 0) {
173     atomic_set_visited(src, visited, &node_locks[src]);
174
175     // Number of tasks in parallel executing at this level of depth
176     int task_count = 0;
177
178     for (int node = 0; node < n_nodes(); node++) {
179         if (edge_exists(src, node) && !atomic_test_visited(node, visited, &node_locks[node])) {
180             // Limit the number of parallel tasks both horizontally (for
181             // checking neighbors) and vertically (between recursive
182             // calls).
183             //
184             // Fallback to iterative version if one of these limits are
185             // reached
186             if (depth <= max_depth_rdfs && task_count <= task_threshold) {
187                 task_count++;
188
189 #pragma omp task untied default(shared) firstprivate(node)
190                 {
191                     p_rdfs(node, visited, node_locks, depth + 1);
192                     task_count--;
193                 }
194
195             } else {
196                 // Fallback to parallel iterative version
197                 p_dfs_with_locks(node, visited, node_locks);
198             }
199         }
200     }
201
202 #pragma omp taskwait
203 }
204
205 // Serial implementation of the Dijkstra algorithm without early exit condition.
206 //
207 // Note: It does not use a priority queue.
208 std::pair<std::vector<Node>, std::vector<Node>>> dijkstra(Node src) {
209     std::vector<Node> queue;
210     queue.push_back(src);
211
212     std::vector<Node> came_from(size(), -1);
213     std::vector<Node> cost_so_far(size(), -1);
214
215     came_from[src] = src;
216     cost_so_far[src] = 0;
217
218     while (!queue.empty()) {
219         Node current = queue.back();
220         queue.pop_back();
221
222         for (int next = 0; next < n_nodes(); next++) {
223             if (edge_exists(current, next)) {
224                 int new_cost = cost_so_far[current] + adj_matrix[current][next];

```

```

225
226         if (cost_so_far[next] == -1 || new_cost < cost_so_far[next]) {
227             cost_so_far[next] = new_cost;
228             queue.push_back(next);
229             came_from[next] = current;
230         }
231     }
232 }
233
234
235     return std::make_pair(came_from, cost_so_far);
236 }
237
238 inline std::vector<omp_lock_t> initialize_locks() {
239     std::vector<omp_lock_t> node_locks;
240     node_locks.reserve(n_nodes());
241
242     for (int node = 0; node < n_nodes(); node++) {
243         omp_lock_t lock;
244         node_locks[node] = lock;
245         omp_init_lock(&(node_locks[node]));
246     }
247
248     return node_locks;
249 }
250
251 // Parallel implementation of the Dijkstra algorithm without early exit
252 // condition using node level locks. As expected, it performs very poorly
253 //
254 // Note: It does not use a priority queue.
255 std::pair<std::vector<Node>, std::vector<Node>> p_dijkstra(Node src) {
256     std::vector<Node> queue;
257     queue.push_back(src);
258
259     std::vector<Node> came_from(size(), -1);
260     std::vector<Node> cost_so_far(size(), -1);
261
262     came_from[src] = src;
263     cost_so_far[src] = 0;
264
265     auto node_locks = initialize_locks();
266
267     while (!queue.empty()) {
268         Node current = queue.back();
269         queue.pop_back();
270
271 #pragma omp parallel shared(queue, node_locks)
272 #pragma omp for
273         for (int next = 0; next < n_nodes(); next++) {
274             if (edge_exists(current, next)) {
275                 omp_set_lock(&node_locks[current]);
276                 auto cost_so_far_current = cost_so_far[current];
277                 omp_unset_lock(&node_locks[current]);
278
279                 int new_cost = cost_so_far_current + adj_matrix[current][next];
280
281                 omp_set_lock(&node_locks[next]);
282                 auto cost_so_far_next = cost_so_far[next];
283                 omp_unset_lock(&node_locks[next]);
284
285                 if (cost_so_far_next == -1 || new_cost < cost_so_far_next) {
286                     omp_set_lock(&node_locks[next]);
287                     cost_so_far[next] = new_cost;
288                     came_from[next] = current;
289                     omp_unset_lock(&node_locks[next]);
290
291 #pragma omp critical(queue_update)
292                     queue.push_back(next);
293                 }
294             }
295         }
296     }
297
298 // Destroy locks
299     for (int node = 0; node < n_nodes(); node++) omp_destroy_lock(&(node_locks[node]));

```

```

300         return std::make_pair(came_from, cost_so_far);
301     }
302 }
303
304 // Reconstruct path from the destination to the source
305 std::vector<Node> reconstruct_path(Node src, Node dst, std::vector<Node> origins) {
306     auto current_node = dst;
307     std::vector<Node> path;
308
309     while (current_node != src) {
310         path.push_back(current_node);
311         current_node = origins.at(current_node);
312     }
313
314     path.push_back(src);
315     reverse(path.begin(), path.end());
316
317     return path;
318 }
319
320 private:
321 // Return true if a node is already visited using a node level lock
322 inline bool atomic_test_visited(Node node, const std::vector<int>& visited, omp_lock_t* lock) {
323     omp_set_lock(lock);
324     bool already_visited = visited.at(node);
325     omp_unset_lock(lock);
326
327     return already_visited;
328 }
329
330 // Set that a node is already visited using a node level lock
331 inline void atomic_set_visited(Node node, std::vector<int>& visited, omp_lock_t* lock) {
332     omp_set_lock(lock);
333     visited[node] = true;
334     omp_unset_lock(lock);
335 }
336 };
337
338 // Import graph from a file
339 Graph import_graph(std::string& path) {
340     Graph graph;
341
342     std::ifstream file(path);
343     if (!file.is_open()) {
344         throw std::invalid_argument("Input file does not exist or is not readable.");
345     }
346
347     std::string line;
348
349     // Read one line at a time into the variable line
350     while (getline(file, line)) {
351         std::vector<int> lineData;
352         std::stringstream lineStream(line);
353
354         // Read an integer at a time from the line
355         int value;
356         while (lineStream >> value) lineData.push_back(value);
357
358         lineData.shrink_to_fit(); // Usefull?
359         graph.adj_matrix.push_back(lineData);
360     }
361
362     graph.adj_matrix.shrink_to_fit();
363
364     return graph;
365 }

```

```

1 // MAIN.CPP
2
3 #include <array>
4 #include <chrono>
5 #include <functional>
6 #include <string>
7 #include <vector>
8
9 #include "graph.hpp"
10
11 using std::chrono::duration_cast;
12 using std::chrono::high_resolution_clock;
13 using std::chrono::milliseconds;
14
15 std::string bench_traverse(std::function<void()> traverse_fn) {
16     auto start = high_resolution_clock::now();
17     traverse_fn();
18     auto stop = high_resolution_clock::now();
19
20     // Subtract stop and start timepoints and cast it to required unit.
21     // Predefined units are nanoseconds, microseconds, milliseconds, seconds,
22     // minutes, hours. Use duration_cast() function.
23     auto duration = duration_cast<milliseconds>(stop - start);
24
25     // To get the value of duration use the count() member function on the
26     // duration object
27     return std::to_string(duration.count());
28 }
29
30 void full_bench(Graph& graph) {
31     int num_test = 1;
32     std::array<int, 6> num_threads{{1, 2, 4, 8, 16, 32}};
33
34     std::vector<Graph::Node> visited(graph.size(), false);
35     Graph::Node src = 0;
36
37     // Explicitly disable dynamic teams as we are going to set a fixed number of
38     // threads
39     omp_set_dynamic(0);
40
41     // TODO: find a better way to avoid code repetition
42
43     std::cout << "Number of nodes: " << graph.size() << "\n\n";
44
45     for (int i = 0; i < num_test; i++) {
46         std::cout << "\t"
47             << "Execution " << i + 1 << std::endl;
48
49         std::cout << "Sequential iterative DFS: "
50             << bench_traverse([&] { graph.dfs(src, visited); }) << "ms\n";
51
52         // We cannot pass a copy of the vector, so we "reset" it every time
53         std::fill(visited.begin(), visited.end(), false);
54
55         std::cout << "Sequential recursive DFS: "
56             << bench_traverse([&]() { graph.rdfs(src, visited); }) << "ms\n";
57
58         std::cout << "Dijkstra: " << bench_traverse([&] { graph.dijkstra(0); }) << "ms\n";
59
60         for (const auto n : num_threads) {
61             std::fill(visited.begin(), visited.end(), false);
62
63             std::cout << "Using " << n << " threads..." << std::endl;
64
65             // Set to use N threads
66             omp_set_num_threads(n);
67
68             // Should we change also this?
69             // graph.task_threshold = n;
70
71             std::cout << "Parallel iterative DFS: "
72                 << bench_traverse([&] { graph.p_dfs(src, visited); }) << "ms\n";
73
74             std::fill(visited.begin(), visited.end(), false);

```

```

75
76         std::cout << "Parallel recursive DFS: "
77             << bench_traverse([&] { graph.p_rdfs(src, visited); }) << "ms\n";
78
79         std::cout << "Parallel Dijkstra: " << bench_traverse([&] { graph.p_dijkstra(0); })
80             << "ms\n";
81     }
82
83     std::fill(visited.begin(), visited.end(), false);
84
85     std::cout << std::endl;
86 }
87 }
88
89 int main(int argc, const char** argv) {
90     // TODO: Add a CLI? Also, we should accept more input files and process them separately
91     if (argc < 2) {
92         std::cout << "Input file not specified.\n";
93         return 1;
94     }
95
96     std::string file_path = argv[1];
97
98     auto graph = import_graph(file_path);
99
100    full_bench(graph);
101
102    return 0;
103 }
104
105 /*
106
107 OUTPUT:
108
109 Number of nodes: 1000
110
111     Execution 1
112 Sequential iterative DFS: 64ms
113 Sequential recursive DFS: 39ms
114 Dijkstra: 72ms
115 Using 1 threads...
116 Parallel iterative DFS: 61ms
117 Parallel recursive DFS: 63ms
118 Parallel Dijkstra: 79ms
119 Using 2 threads...
120 Parallel iterative DFS: 44ms
121 Parallel recursive DFS: 37ms
122 Parallel Dijkstra: 89ms
123 Using 4 threads...
124 Parallel iterative DFS: 37ms
125 Parallel recursive DFS: 25ms
126 Parallel Dijkstra: 256ms
127 Using 8 threads...
128 Parallel iterative DFS: 36ms
129 Parallel recursive DFS: 15ms
130 Parallel Dijkstra: 257ms
131 Using 16 threads...
132 Parallel iterative DFS: 80ms
133 Parallel recursive DFS: 15ms
134 Parallel Dijkstra: 532ms
135 Using 32 threads...
136 Parallel iterative DFS: 186ms
137 Parallel recursive DFS: 21ms
138 Parallel Dijkstra: 481ms
139 */

```