## HPC/1/graph.hpp

```
1
   #pragma once
 2
 3
   #include <omp.h>
 4
 5
   #include <fstream>
   #include <functional>
 6
 7
   #include <iostream>
 8
   #include <queue>
   #include <sstream>
 9
   #include <string>
10
   #include <tuple>
11
12
   #include <vector>
13
   #include <algorithm>
14
   // Generic representation of a graph implemented with an adjacency matrix
15
   struct Graph {
16
        using Node = int;
17
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
        // Sequential implementation of the iterative version of depth first search.
33
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
39
                queue.pop_back();
40
41
                if (!visited[node]) {
42
                    visited[node] = true;
43
44
                    for (int next_node = 0; next_node < n_nodes(); next_node++)</pre>
45
                         if (edge_exists(node, next_node) && !visited[next_node])
                             queue.push_back(next_node);
46
47
                }
48
            3
        }
49
50
51
        // Sequential implementation of the recursive version of depth first search.
52
        void rdfs(Node src, std::vector<int>& visited, int depth = 0) {
            visited[src] = true;
53
54
55
            for (int node = 0; node < n_nodes(); node++) {</pre>
                if (edge_exists(src, node) && !visited[node]) {
56
                    // Limit recursion depth to avoid stack overflow error
57
58
                    if (depth ≤ max_depth_rdfs)
                        rdfs(node, visited, depth + 1);
59
60
61
                        dfs(node, visited);
62
                }
63
            3
        }
64
65
        // Parallel implementation of the iterative version of depth first search.
66
67
68
        // 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
69
        // have a private queue where neighbors still not visited are added. At the end,
70
```

```
71
         // threads concatenate their private queue to the main queue.
 72
         void p_dfs(Node src, std::vector<int>& visited) {
 73
             std::vector<Node> queue{src};
 74
75
             while (!queue.empty()) {
                 Node node = queue.back();
76
 77
                 queue.pop_back();
 78
 79
                 if (!visited[node]) {
 80
                     visited[node] = true;
81
82
    #pragma omp parallel shared(queue, visited)
83
                     {
                         // Every thread has a private_queue to avoid continuous lock
 84
                         // checking to update the main one
 85
 86
                         std::vector<Node> private_queue;
87
    #pragma omp for nowait schedule(static)
 88
89
                         for (int next_node = 0; next_node < n_nodes(); next_node++)</pre>
 90
                             if (edge_exists(node, next_node) & !visited[next_node])
 91
                                 private_queue.push_back(next_node);
 92
 93
    // Append at the end of master queue the private queue of the thread
 94
    #pragma omp critical(queue_update)
 95
                         queue.insert(queue.end(), private_queue.begin(), private_queue.end());
 96
                     }
 97
                 }
 98
             }
99
        }
100
        // Parallel implementation of the iterative version of depth first search.
101
102
        //
        // The general idea is that the main thread extracts the last node from the
103
104
        // 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.
105
106
        // At the end, threads concatenate their private queue to the main queue.
107
         // **Important**: this version implements node level locks.
108
         void p_dfs_with_locks(Node src, std::vector<int>& visited,
109
                               std::vector<omp_lock_t>& node_locks) {
110
             // Note: Since C++11, different elements in the same container can be
111
112
             // modified concurrently by different threads, except for the elements
113
             // of std::vector<bool>
114
             // Possible explanation of why here:
115
             // https://stackoverflow.com/a/33617530/2691946
116
117
             //
118
             // This is why we use a vector of int.
119
120
             std::vector<Node> queue{src};
121
             while (!queue.empty()) {
122
                 Node node = queue.back();
123
124
                 queue.pop_back();
125
                 bool already_visited = atomic_test_visited(node, visited, &node_locks[node]);
126
127
128
                 if (!already_visited) {
                     atomic_set_visited(node, visited, &node_locks[node]);
129
130
    #pragma omp parallel shared(queue, visited)
131
132
                     {
133
                         // Every thread has a private queue to avoid continuos lock
134
                         // checking to update the main one
                         std::vector<Node> private_queue;
135
136
137
    #pragma omp for nowait
138
                         for (int next_node = 0; next_node < n_nodes(); next_node++) {</pre>
139
                              // Check if the edge exists is a non-blocking request,
140
                             // so it's better to do it before than checking if the
141
                             // node is already visited
142
                             if (edge_exists(node, next_node)) {
                                  if (atomic_test_visited(next_node, visited, &node_locks[next_node])) {
143
                                      private_queue.push_back(next_node);
144
```

```
145
                                  }
146
                              }
147
                          }
148
     // Append at the end of master queue the private queue of the thread
149
    #pragma omp critical(queue_update)
150
151
                          queue.insert(queue.end(), private_queue.begin(), private_queue.end());
152
                     }
                 3
153
154
             }
        }
155
156
        // Parallel implementation of the recursive version of depth first search.
157
158
         //
         // This version automatically initialize locks
159
         void p_rdfs(Node src, std::vector<int>& visited) {
160
161
             // Initialize locks
             std::vector<omp_lock_t> node_locks;
162
             node_locks.reserve(size());
163
164
165
             for (int node = 0; node < n_nodes(); node++) {</pre>
                 omp_lock_t lock;
166
                 node_locks[node] = lock;
167
168
                 omp_init_lock(&(node_locks[node]));
169
             }
170
    #pragma omp parallel shared(src, visited, node_locks)
171
172
    #pragma omp single
             p_rdfs(src, visited, node_locks);
173
174
             // Destory locks
175
             for (int node = 0; node < n_nodes(); node++) omp_destroy_lock(&(node_locks[node]));</pre>
176
         }
177
178
        // Parallel implementation of the recursive version of depth first search,
179
180
        // full version with locks
181
         void p_rdfs(Node src, std::vector<int>& visited, std::vector<omp_lock_t>& node_locks,
182
                     int depth = 0) {
183
             atomic_set_visited(src, visited, &node_locks[src]);
184
             // Number of tasks in parallel executing at this level of depth
185
             int task_count = 0;
186
187
188
             for (int node = 0; node < n_nodes(); node++) {</pre>
                 if (edge_exists(src, node) && !atomic_test_visited(node, visited, &node_locks[node])) {
189
190
                     // Limit the number of parallel tasks both horizontally (for
                     // checking neighbors) and vertically (between recursive
191
                     // calls).
192
193
                     //
194
                     // Fallback to iterative version if one of these limits are
195
                     // reached
                     if (depth 

max_depth_rdfs & task_count 

task_threshold) {
196
197
                          task_count++;
198
    #pragma omp task untied default(shared) firstprivate(node)
199
200
                          {
                              p_rdfs(node, visited, node_locks, depth + 1);
201
202
                              task_count--;
203
                          3
204
205
                     } else {
206
                          // Fallback to parallel iterative version
207
                          p_dfs_with_locks(node, visited, node_locks);
208
                     }
                 }
209
             }
210
211
212
    #pragma omp taskwait
213
214
215
        // Serial implementation of the Dijkstra algorithm without early exit condition.
216
         //
217
         // Note: It does not use a priority queue.
218
         std::pair<std::vector<Node>, std::vector<Node>> dijkstra(Node src) {
```

```
219
             std::vector<Node> queue;
220
             queue.push_back(src);
221
222
             std::vector<Node> came_from(size(), -1);
             std::vector<Node> cost_so_far(size(), -1);
223
224
225
             came_from[src] = src;
226
             cost_so_far[src] = 0;
227
228
             while (!queue.empty()) {
229
                 Node current = queue.back();
                 queue.pop_back();
230
231
232
                 for (int next = 0; next < n_nodes(); next++) {</pre>
233
                      if (edge_exists(current, next)) {
234
                          int new_cost = cost_so_far[current] + adj_matrix[current][next];
235
                          if (cost_so_far[next] = -1 || new_cost < cost_so_far[next]) {</pre>
236
                              cost_so_far[next] = new_cost;
237
238
                              queue.push_back(next);
                              came_from[next] = current;
239
                          }
240
241
                     }
242
                 3
             }
243
244
245
             return std::make_pair(came_from, cost_so_far);
246
         3
247
248
         inline std::vector<omp_lock_t> initialize_locks() {
249
             std::vector<omp_lock_t> node_locks;
250
             node_locks.reserve(n_nodes());
251
252
             for (int node = 0; node < n_nodes(); node++) {</pre>
                 omp_lock_t lock;
253
254
                 node_locks[node] = lock;
255
                 omp_init_lock(&(node_locks[node]));
256
             }
257
258
             return node_locks;
259
         }
260
261
         // Parallel implementation of the Dijkstra algorithm without early exit
262
         // condition using node level locks. As expected, it performs very poorly
263
         //
264
         // Note: It does not use a priority queue.
         std::pair<std::vector<Node>, std::vector<Node>> p_dijkstra(Node src) {
265
266
             std::vector<Node> queue;
267
             queue.push_back(src);
268
             std::vector<Node> came_from(size(), -1);
269
             std::vector<Node> cost_so_far(size(), -1);
270
271
             came_from[src] = src;
272
273
             cost_so_far[src] = 0;
274
275
             auto node_locks = initialize_locks();
276
277
             while (!queue.empty()) {
                 Node current = queue.back();
278
279
                 queue.pop_back();
280
281
    #pragma omp parallel shared(queue, node_locks)
282
    #pragma omp for
                 for (int next = 0; next < n_nodes(); next++) {</pre>
283
284
                      if (edge_exists(current, next)) {
285
                          omp_set_lock(&node_locks[current]);
286
                          auto cost_so_far_current = cost_so_far[current];
287
                          omp_unset_lock(&node_locks[current]);
288
289
                          int new_cost = cost_so_far_current + adj_matrix[current][next];
290
291
                          omp_set_lock(&node_locks[next]);
292
                          auto cost_so_far_next = cost_so_far[next];
```

```
293
                          omp_unset_lock(&node_locks[next]);
294
295
                          if (cost_so_far_next = -1 || new_cost < cost_so_far_next) {</pre>
296
                              omp_set_lock(&node_locks[next]);
297
                              cost_so_far[next] = new_cost;
                              came_from[next] = current;
798
299
                              omp_unset_lock(&node_locks[next]);
300
301
    #pragma omp critical(queue_update)
302
                              queue.push_back(next);
303
                          }
304
                     }
305
                 }
             }
306
307
308
             // Destory locks
             for (int node = 0; node < n_nodes(); node++) omp_destroy_lock(&(node_locks[node]));</pre>
309
310
311
             return std::make_pair(came_from, cost_so_far);
312
        }
313
         // Reconstruct path from the destination to the source
314
315
         std::vector<Node> reconstruct_path(Node src, Node dst, std::vector<Node> origins) {
316
             auto current_node = dst;
             std::vector<Node> path;
317
318
319
             while (current_node \neq src) {
320
                 path.push_back(current_node);
321
                 current_node = origins.at(current_node);
322
             3
323
324
             path.push_back(src);
             reverse(path.begin(), path.end());
325
326
327
             return path;
328
        }
329
330
        private:
         // Return true if a node is already visited using a node level lock
331
332
         inline bool atomic_test_visited(Node node, const std::vector<int>& visited, omp_lock_t* lock) {
333
             omp_set_lock(lock);
334
             bool already_visited = visited.at(node);
335
             omp_unset_lock(lock);
336
337
             return already_visited;
338
         }
339
         // Set that a node is already visited using a node level lock
340
         inline void atomic_set_visited(Node node, std::vector<int>& visited, omp_lock_t* lock) {
341
342
             omp_set_lock(lock);
343
             visited[node] = true;
344
             omp_unset_lock(lock);
345
         3
346
    };
347
348
    // Import graph from a file
349
    Graph import_graph(std::string& path) {
350
         Graph graph;
351
352
         std::ifstream file(path);
353
         if (!file.is_open()) {
             throw std::invalid_argument("Input file does not exist or is not readable.");
354
355
356
         std::string line;
357
358
359
         // Read one line at a time into the variable line
360
         while (getline(file, line)) {
             std::vector<int> lineData;
361
362
             std::stringstream lineStream(line);
363
364
             // Read an integer at a time from the line
365
             int value:
             while (lineStream >> value) lineData.push_back(value);
366
```

## HPC/1/main.cpp

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

```
<< bench_traverse([&] { graph.p_dfs(src, visited); }) << "ms\n";</pre>
 71
 72
 73
                 std::fill(visited.begin(), visited.end(), false);
 74
                 std::cout << "Parallel recursive DFS: "</pre>
 75
76
                           << bench_traverse([&] { graph.p_rdfs(src, visited); }) << "ms\n";</pre>
 77
 78
                 std::cout << "Parallel iterative BFS: " << bench_traverse([&] { graph.p_dijkstra(0); })</pre>
 79
                           << "ms\n";
             }
 80
 81
 82
             std::fill(visited.begin(), visited.end(), false);
 83
             std::cout << std::endl;</pre>
 84
 85
        }
 86
    }
 87
 88
    int main(int argc, const char** argv) {
         // TODO: Add a CLI? Also, we should accept more input files and process them separately
 89
 90
         if (argc < 2) {
 91
             std::cout << "Input file not specified.\n";</pre>
 92
             return 1:
 93
         }
 94
 95
        std::string file_path = argv[1];
 96
97
         auto graph = import_graph(file_path);
98
99
         full_bench(graph);
100
        return 0;
101
    }
102
103
104
105
106
    OUTPUT:
107
108 Number of nodes: 1000
109
110
             Execution 1
111
    Sequential iterative DFS: 21ms
112
    Sequential recursive DFS: 13ms
113
    Sequential iterative BFS: 23ms
    Using 1 threads...
114
115 | Parallel iterative DFS: 20ms
116 | Parallel recursive DFS: 20ms
117 | Parallel iterative BFS: 25ms
118 Using 2 threads...
119 Parallel iterative DFS: 15ms
120 Parallel recursive DFS: 12ms
121 Parallel iterative BFS: 29ms
122 Using 4 threads...
123 Parallel iterative DFS: 14ms
124 Parallel recursive DFS: 8ms
125 Parallel iterative BFS: 59ms
126 Using 8 threads...
127 Parallel iterative DFS: 14ms
128 Parallel recursive DFS: 6ms
129 Parallel iterative BFS: 86ms
130 Using 16 threads...
131 Parallel iterative DFS: 35ms
132 Parallel recursive DFS: 9ms
133 Parallel iterative BFS: 149ms
134 Using 32 threads...
135 Parallel iterative DFS: 81ms
136 Parallel recursive DFS: 11ms
    Parallel iterative BFS: 191ms
137
138
139
140
```