

## HPC/1/graph.hpp

```
1  #pragma once
2
3  #include <omp.h>
4
5  #include <fstream>
6  #include <functional>
7  #include <iostream>
8  #include <queue>
9  #include <sstream>
10 #include <string>
11 #include <tuple>
12 #include <vector>
13 #include <algorithm>
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
37         while (!queue.empty()) {
38             Node node = queue.back();
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++)
45                     if (edge_exists(node, next_node) && !visited[next_node])
46                         queue.push_back(next_node);
47             }
48         }
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) {
53         visited[src] = true;
54
55         for (int node = 0; node < n_nodes(); node++) {
56             if (edge_exists(src, node) && !visited[node]) {
57                 // Limit recursion depth to avoid stack overflow error
58                 if (depth ≤ max_depth_rdfs)
59                     rdfs(node, visited, depth + 1);
60                 else
61                     dfs(node, visited);
62             }
63         }
64     }
65
66     // Parallel implementation of the iterative version of depth first search.
67     //
68     // The general idea is that the main thread extracts the last node from the
69     // queue and check the neighbors of the node in parallel. Each of these threads
70     // have a private queue where neighbors still not visited are added. At the end,
```

[illegible]

```

145     }
146     }
147 }
148
149 // Append at the end of master queue the private queue of the thread
150 #pragma omp critical(queue_update)
151     queue.insert(queue.end(), private_queue.begin(), private_queue.end());
152 }
153 }
154 }
155 }
156
157 // Parallel implementation of the recursive version of depth first search.
158 //
159 // This version automatically initialize locks
160 void p_rdfs(Node src, std::vector<int>& visited) {
161     // Initialize locks
162     std::vector<omp_lock_t> node_locks;
163     node_locks.reserve(size());
164
165     for (int node = 0; node < n_nodes(); node++) {
166         omp_lock_t lock;
167         node_locks[node] = lock;
168         omp_init_lock(&(node_locks[node]));
169     }
170
171 #pragma omp parallel shared(src, visited, node_locks)
172 #pragma omp single
173     p_rdfs(src, visited, node_locks);
174
175     // Destroy locks
176     for (int node = 0; node < n_nodes(); node++) omp_destroy_lock(&(node_locks[node]));
177 }
178
179 // Parallel implementation of the recursive version of depth first search,
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
185     // Number of tasks in parallel executing at this level of depth
186     int task_count = 0;
187
188     for (int node = 0; node < n_nodes(); node++) {
189         if (edge_exists(src, node) && !atomic_test_visited(node, visited, &node_locks[node])) {
190             // Limit the number of parallel tasks both horizontally (for
191             // checking neighbors) and vertically (between recursive
192             // calls).
193             //
194             // Fallback to iterative version if one of these limits are
195             // reached
196             if (depth ≤ max_depth_rdfs && task_count ≤ task_threshold) {
197                 task_count++;
198
199 #pragma omp task untied default(shared) firstprivate(node)
200                 {
201                     p_rdfs(node, visited, node_locks, depth + 1);
202                     task_count--;
203                 }
204             } else {
205                 // Fallback to parallel iterative version
206                 p_dfs_with_locks(node, visited, node_locks);
207             }
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);
223     std::vector<Node> cost_so_far(size(), -1);
224
225     came_from[src] = src;
226     cost_so_far[src] = 0;
227
228     while (!queue.empty()) {
229         Node current = queue.back();
230         queue.pop_back();
231
232         for (int next = 0; next < n_nodes(); next++) {
233             if (edge_exists(current, next)) {
234                 int new_cost = cost_so_far[current] + adj_matrix[current][next];
235
236                 if (cost_so_far[next] == -1 || new_cost < cost_so_far[next]) {
237                     cost_so_far[next] = new_cost;
238                     queue.push_back(next);
239                     came_from[next] = current;
240                 }
241             }
242         }
243     }
244
245     return std::make_pair(came_from, cost_so_far);
246 }
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++) {
253         omp_lock_t lock;
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.
265 std::pair<std::vector<Node>, std::vector<Node>> p_dijkstra(Node src) {
266     std::vector<Node> queue;
267     queue.push_back(src);
268
269     std::vector<Node> came_from(size(), -1);
270     std::vector<Node> cost_so_far(size(), -1);
271
272     came_from[src] = src;
273     cost_so_far[src] = 0;
274
275     auto node_locks = initialize_locks();
276
277     while (!queue.empty()) {
278         Node current = queue.back();
279         queue.pop_back();
280
281 #pragma omp parallel shared(queue, node_locks)
282 #pragma omp for
283         for (int next = 0; next < n_nodes(); next++) {
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) {
296             omp_set_lock(&node_locks[next]);
297             cost_so_far[next] = new_cost;
298             came_from[next] = current;
299             omp_unset_lock(&node_locks[next]);
300
301 #pragma omp critical(queue_update)
302             queue.push_back(next);
303         }
304     }
305 }
306
307 // Destroy locks
308 for (int node = 0; node < n_nodes(); node++) omp_destroy_lock(&(node_locks[node]));
309
310 return std::make_pair(came_from, cost_so_far);
311 }
312
313 // Reconstruct path from the destination to the source
314 std::vector<Node> reconstruct_path(Node src, Node dst, std::vector<Node> origins) {
315     auto current_node = dst;
316     std::vector<Node> path;
317
318     while (current_node != src) {
319         path.push_back(current_node);
320         current_node = origins.at(current_node);
321     }
322
323     path.push_back(src);
324     reverse(path.begin(), path.end());
325
326     return path;
327 }
328
329 private:
330 // Return true if a node is already visited using a node level lock
331 inline bool atomic_test_visited(Node node, const std::vector<int>& visited, omp_lock_t* lock) {
332     omp_set_lock(lock);
333     bool already_visited = visited.at(node);
334     omp_unset_lock(lock);
335
336     return already_visited;
337 }
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     omp_set_lock(lock);
342     visited[node] = true;
343     omp_unset_lock(lock);
344 }
345 };
346
347 // Import graph from a file
348 Graph import_graph(std::string& path) {
349     Graph graph;
350
351     std::ifstream file(path);
352     if (!file.is_open()) {
353         throw std::invalid_argument("Input file does not exist or is not readable.");
354     }
355
356     std::string line;
357
358     // Read one line at a time into the variable line
359     while (getline(file, line)) {
360         std::vector<int> lineData;
361         std::stringstream lineStream(line);
362
363         // Read an integer at a time from the line
364         int value;
365         while (lineStream >> value) lineData.push_back(value);
366     }

```

```
367
368     lineData.shrink_to_fit(); // Usefull?
369     graph.adj_matrix.push_back(lineData);
370 }
371
372 graph.adj_matrix.shrink_to_fit();
373
374 return graph;
375 }
376
```

## HPC/1/main.cpp

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

```

71         << bench_traverse([&] { graph.p_dfs(src, visited); }) << "ms\n";
72
73     std::fill(visited.begin(), visited.end(), false);
74
75     std::cout << "Parallel recursive DFS: "
76         << bench_traverse([&] { graph.p_rdfs(src, visited); }) << "ms\n";
77
78     std::cout << "Parallel iterative BFS: " << bench_traverse([&] { graph.p_dijkstra(0); })
79         << "ms\n";
80 }
81
82     std::fill(visited.begin(), visited.end(), false);
83
84     std::cout << std::endl;
85 }
86 }
87
88 int main(int argc, const char** argv) {
89     // TODO: Add a CLI? Also, we should accept more input files and process them separately
90     if (argc < 2) {
91         std::cout << "Input file not specified.\n";
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
101     return 0;
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
114 Using 1 threads...
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
137 Parallel iterative BFS: 191ms
138
139 */
140

```



## HPC/2/bubble\_sort.cpp

```
1  #include <omp.h>
2  #include <stdlib.h>
3
4  #include <array>
5  #include <chrono>
6  #include <functional>
7  #include <iostream>
8  #include <string>
9  #include <vector>
10
11 using std::chrono::duration_cast;
12 using std::chrono::high_resolution_clock;
13 using std::chrono::milliseconds;
14 using namespace std;
15
16 void s_bubble(int *, int);
17 void p_bubble(int *, int);
18 void swap(int &, int &);
19
20 void s_bubble(int *a, int n) {
21     for (int i = 0; i < n; i++) {
22         int first = i % 2;
23         for (int j = first; j < n - 1; j += 2) {
24             if (a[j] > a[j + 1]) {
25                 swap(a[j], a[j + 1]);
26             }
27         }
28     }
29 }
30
31 void p_bubble(int *a, int n) {
32     for (int i = 0; i < n; i++) {
33         int first = i % 2;
34 #pragma omp parallel for shared(a, first) num_threads(16)
35         for (int j = first; j < n - 1; j += 2) {
36             if (a[j] > a[j + 1]) {
37                 swap(a[j], a[j + 1]);
38             }
39         }
40     }
41 }
42
43 void swap(int &a, int &b) {
44     int test;
45     test = a;
46     a = b;
47     b = test;
48 }
49
50 std::string bench_traverse(std::function<void()> traverse_fn) {
51     auto start = high_resolution_clock::now();
52     traverse_fn();
53     auto stop = high_resolution_clock::now();
54
55     // Subtract stop and start timepoints and cast it to required unit.
56     // Predefined units are nanoseconds, microseconds, milliseconds, seconds,
57     // minutes, hours. Use duration_cast() function.
58     auto duration = duration_cast<milliseconds>(stop - start);
59
60     // To get the value of duration use the count() member function on the
61     // duration object
62     return std::to_string(duration.count());
63 }
64
65 int main(int argc, const char **argv) {
66     if (argc < 3) {
67         std::cout << "Specify array length and maximum random value\n";
68         return 1;
69     }
70     int *a, n, rand_max;
```

```

71
72     n = stoi(argv[1]);
73     rand_max = stoi(argv[2]);
74     a = new int[n];
75
76     for (int i = 0; i < n; i++) {
77         a[i] = rand() % rand_max;
78     }
79
80     int *b = new int[n];
81     copy(a, a + n, b);
82     cout << "Generated random array of length " << n << " with elements between 0 to " << rand_max
83         << "\n\n";
84
85     std::cout << "Sequential Bubble sort: " << bench_traverse([&] { s_bubble(a, n); }) << "ms\n";
86     cout << "Sorted array is ⇒\n";
87     for (int i = 0; i < n; i++) {
88         cout << a[i] << ", ";
89     }
90     cout << "\n\n";
91
92     omp_set_num_threads(16);
93     std::cout << "Parallel (16) Bubble sort: " << bench_traverse([&] { p_bubble(b, n); }) << "ms\n";
94     cout << "Sorted array is ⇒\n";
95     for (int i = 0; i < n; i++) {
96         cout << b[i] << ", ";
97     }
98     return 0;
99 }
100
101 /*
102
103 OUTPUT:
104 Generated random array of length 100 with elements between 0 to 200
105
106 Sequential Bubble sort: 0ms
107 Sorted array is ⇒
108 2, 3, 8, 11, 11, 12, 13, 14, 21, 21, 22, 26, 26, 27, 29, 29, 34, 42, 43, 46, 49, 51, 56, 57, 58, 59,
109 60, 62, 62, 67, 69, 73, 76, 76, 81, 84, 86, 87, 90, 91, 92, 94, 95, 105, 105, 113, 115, 115, 119,
110 123, 124, 124, 125, 126, 126, 127, 129, 129, 130, 132, 135, 135, 136, 136, 137, 139, 139, 140, 145,
111 150, 154, 156, 162, 163, 164, 167, 167, 168, 168, 170, 170, 172, 173, 177, 178, 180, 182, 182,
112 183, 184, 184, 186, 186, 188, 193, 193, 196, 198, 199,
113
114 Parallel (16) Bubble sort: 1ms
115 Sorted array is ⇒
116 2, 3, 8, 11, 11, 12, 13, 14, 21, 21, 22, 26, 26, 27, 29, 29, 34, 42, 43, 46, 49, 51, 56, 57, 58, 59,
117 60, 62, 62, 67, 69, 73, 76, 76, 81, 84, 86, 87, 90, 91, 92, 94, 95, 105, 105, 113, 115, 115, 119,
118 123, 124, 124, 125, 126, 126, 127, 129, 129, 130, 132, 135, 135, 136, 136, 137, 139, 139, 140, 145,
119 150, 154, 156, 162, 163, 164, 167, 167, 168, 168, 170, 170, 172, 173, 177, 178, 180, 182, 182,
120 183, 184, 184, 186, 186, 188, 193, 193, 196, 198, 199,
121
122
123 OUTPUT:
124
125 Generated random array of length 100000 with elements between 0 to 100000
126
127 Sequential Bubble sort: 16878ms
128 Parallel (16) Bubble sort: 2914ms
129
130 */
131

```

## HPC/2/merge\_sort.cpp

```
1  #include <omp.h>
2  #include <stdlib.h>
3
4  #include <array>
5  #include <chrono>
6  #include <functional>
7  #include <iostream>
8  #include <string>
9  #include <vector>
10
11 using std::chrono::duration_cast;
12 using std::chrono::high_resolution_clock;
13 using std::chrono::milliseconds;
14 using namespace std;
15
16 void p_mergesort(int *a, int i, int j);
17 void s_mergesort(int *a, int i, int j);
18 void merge(int *a, int i1, int j1, int i2, int j2);
19
20 void p_mergesort(int *a, int i, int j) {
21     int mid;
22     if (i < j) {
23         if ((j - i) > 1000) {
24             mid = (i + j) / 2;
25
26 #pragma omp task firstprivate(a, i, mid)
27             p_mergesort(a, i, mid);
28 #pragma omp task firstprivate(a, mid, j)
29             p_mergesort(a, mid + 1, j);
30
31 #pragma omp taskwait
32             merge(a, i, mid, mid + 1, j);
33         } else {
34             s_mergesort(a, i, j);
35         }
36     }
37 }
38
39 void parallel_mergesort(int *a, int i, int j) {
40 #pragma omp parallel num_threads(16)
41 {
42 #pragma omp single
43     p_mergesort(a, i, j);
44 }
45 }
46
47 void s_mergesort(int *a, int i, int j) {
48     int mid;
49     if (i < j) {
50         mid = (i + j) / 2;
51         s_mergesort(a, i, mid);
52         s_mergesort(a, mid + 1, j);
53         merge(a, i, mid, mid + 1, j);
54     }
55 }
56
57 void merge(int *a, int i1, int j1, int i2, int j2) {
58     int temp[2000000];
59     int i, j, k;
60     i = i1;
61     j = i2;
62     k = 0;
63     while (i ≤ j1 && j ≤ j2) {
64         if (a[i] < a[j]) {
65             temp[k++] = a[i++];
66         } else {
67             temp[k++] = a[j++];
68         }
69     }
70     while (i ≤ j1) {
```

```

71     temp[k++] = a[i++];
72 }
73 while (j ≤ j2) {
74     temp[k++] = a[j++];
75 }
76 for (i = i1, j = 0; i ≤ j2; i++, j++) {
77     a[i] = temp[j];
78 }
79 }
80
81 std::string bench_traverse(std::function<void()> traverse_fn) {
82     auto start = high_resolution_clock::now();
83     traverse_fn();
84     auto stop = high_resolution_clock::now();
85
86     // Subtract stop and start timepoints and cast it to required unit.
87     // Predefined units are nanoseconds, microseconds, milliseconds, seconds,
88     // minutes, hours. Use duration_cast() function.
89     auto duration = duration_cast<milliseconds>(stop - start);
90
91     // To get the value of duration use the count() member function on the
92     // duration object
93     return std::to_string(duration.count());
94 }
95
96 int main(int argc, const char **argv) {
97     if (argc < 3) {
98         std::cout << "Specify array length and maximum random value\n";
99         return 1;
100     }
101     int *a, n, rand_max;
102
103     n = stoi(argv[1]);
104     rand_max = stoi(argv[2]);
105     a = new int[n];
106
107     for (int i = 0; i < n; i++) {
108         a[i] = rand() % rand_max;
109     }
110
111     int *b = new int[n];
112     copy(a, a + n, b);
113     cout << "Generated random array of length " << n << " with elements between 0 to " << rand_max
114         << "\n\n";
115
116     std::cout << "Sequential Merge sort: " << bench_traverse([&] { s_mergesort(a, 0, n - 1); })
117         << "ms\n";
118
119     cout << "Sorted array is ⇒\n";
120     for (int i = 0; i < n; i++) {
121         cout << a[i] << ", ";
122     }
123     cout << "\n\n";
124
125     omp_set_num_threads(16);
126     std::cout << "Parallel (16) Merge sort: "
127         << bench_traverse([&] { parallel_mergesort(b, 0, n - 1); }) << "ms\n";
128
129     cout << "Sorted array is ⇒\n";
130     for (int i = 0; i < n; i++) {
131         cout << b[i] << ", ";
132     }
133     return 0;
134 }
135
136 /*
137 OUTPUT:
138
139 Generated random array of length 100 with elements between 0 to 200
140
141 Sequential Merge sort: 0ms
142 Sorted array is ⇒
143 2, 3, 8, 11, 11, 12, 13, 14, 21, 21, 22, 26, 26, 27, 29, 29, 34, 42, 43, 46, 49, 51, 56, 57, 58, 59,

```

```
145 60, 62, 62, 67, 69, 73, 76, 76, 81, 84, 86, 87, 90, 91, 92, 94, 95, 105, 105, 113, 115, 115, 119,
146 123, 124, 124, 125, 126, 126, 127, 129, 129, 130, 132, 135, 135, 136, 136, 137, 139, 139, 140, 145,
147 150, 154, 156, 162, 163, 164, 167, 167, 167, 168, 168, 170, 170, 172, 173, 177, 178, 180, 182, 182,
148 183, 184, 184, 186, 186, 188, 193, 193, 196, 198, 199,
149
150 Parallel (16) Merge sort: 1ms
151 Sorted array is ⇒
152 2, 3, 8, 11, 11, 12, 13, 14, 21, 21, 22, 26, 26, 27, 29, 29, 34, 42, 43, 46, 49, 51, 56, 57, 58, 59,
153 60, 62, 62, 67, 69, 73, 76, 76, 81, 84, 86, 87, 90, 91, 92, 94, 95, 105, 105, 113, 115, 115, 119,
154 123, 124, 124, 125, 126, 126, 127, 129, 129, 130, 132, 135, 135, 136, 136, 137, 139, 139, 140, 145,
155 150, 154, 156, 162, 163, 164, 167, 167, 167, 168, 168, 170, 170, 172, 173, 177, 178, 180, 182, 182,
156 183, 184, 184, 186, 186, 188, 193, 193, 196, 198, 199,
157
158
159 OUTPUT:
160
161 Generated random array of length 1000000 with elements between 0 to 1000000
162
163 Sequential Merge sort: 165ms
164 Parallel (16) Merge sort: 42ms
165
166 */
167
```

## HPC/3/statistics.cpp

```
1  #include <limits.h>
2  #include <omp.h>
3  #include <stdlib.h>
4
5  #include <array>
6  #include <chrono>
7  #include <functional>
8  #include <iostream>
9  #include <string>
10 #include <vector>
11
12 using std::chrono::duration_cast;
13 using std::chrono::high_resolution_clock;
14 using std::chrono::milliseconds;
15 using namespace std;
16
17 void s_avg(int arr[], int n) {
18     long sum = 0L;
19     int i;
20     for (i = 0; i < n; i++) {
21         sum = sum + arr[i];
22     }
23     cout << sum / long(n);
24 }
25
26 void p_avg(int arr[], int n) {
27     long sum = 0L;
28     int i;
29 #pragma omp parallel for reduction(+ : sum) num_threads(16)
30     for (i = 0; i < n; i++) {
31         sum = sum + arr[i];
32     }
33     cout << sum / long(n);
34 }
35
36 void s_sum(int arr[], int n) {
37     long sum = 0L;
38     int i;
39     for (i = 0; i < n; i++) {
40         sum = sum + arr[i];
41     }
42     cout << sum;
43 }
44
45 void p_sum(int arr[], int n) {
46     long sum = 0L;
47     int i;
48 #pragma omp parallel for reduction(+ : sum) num_threads(16)
49     for (i = 0; i < n; i++) {
50         sum = sum + arr[i];
51     }
52     cout << sum;
53 }
54
55 void s_max(int arr[], int n) {
56     int max_val = INT_MIN;
57     int i;
58     for (i = 0; i < n; i++) {
59         if (arr[i] > max_val) {
60             max_val = arr[i];
61         }
62     }
63     cout << max_val;
64 }
65
66 void p_max(int arr[], int n) {
67     int max_val = INT_MIN;
68     int i;
69 #pragma omp parallel for reduction(max : max_val) num_threads(16)
70     for (i = 0; i < n; i++) {
```

```

71         if (arr[i] > max_val) {
72             max_val = arr[i];
73         }
74     }
75     cout << max_val;
76 }
77
78 void s_min(int arr[], int n) {
79     int min_val = INT_MAX;
80     int i;
81     for (i = 0; i < n; i++) {
82         if (arr[i] < min_val) {
83             min_val = arr[i];
84         }
85     }
86     cout << min_val;
87 }
88
89 void p_min(int arr[], int n) {
90     int min_val = INT_MAX;
91     int i;
92     #pragma omp parallel for reduction(min : min_val) num_threads(16)
93     for (i = 0; i < n; i++) {
94         if (arr[i] < min_val) {
95             min_val = arr[i];
96         }
97     }
98     cout << min_val;
99 }
100
101 std::string bench_traverse(std::function<void()> traverse_fn) {
102     auto start = high_resolution_clock::now();
103     traverse_fn();
104     cout << " (";
105     auto stop = high_resolution_clock::now();
106
107     // Subtract stop and start timepoints and cast it to required unit.
108     // Predefined units are nanoseconds, microseconds, milliseconds, seconds,
109     // minutes, hours. Use duration_cast() function.
110     auto duration = duration_cast<milliseconds>(stop - start);
111
112     // To get the value of duration use the count() member function on the
113     // duration object
114     return std::to_string(duration.count());
115 }
116
117 int main(int argc, const char **argv) {
118     if (argc < 3) {
119         std::cout << "Specify array length and maximum random value\n";
120         return 1;
121     }
122     int *a, n, rand_max;
123
124     n = stoi(argv[1]);
125     rand_max = stoi(argv[2]);
126     a = new int[n];
127
128     for (int i = 0; i < n; i++) {
129         a[i] = rand() % rand_max;
130     }
131
132     cout << "Generated random array of length " << n << " with elements between 0 to " << rand_max
133         << "\n\n";
134     cout << "Given array is =>\n";
135     for (int i = 0; i < n; i++) {
136         cout << a[i] << ", ";
137     }
138     cout << "\n\n";
139
140     omp_set_num_threads(16);
141
142     std::cout << "Sequential Min: " << bench_traverse([&] { s_min(a, n); }) << "ms)\n";
143
144     std::cout << "Parallel (16) Min: " << bench_traverse([&] { p_min(a, n); }) << "ms)\n\n";

```

```

145     std::cout << "Sequential Max: " << bench_traverse([&] { s_max(a, n); }) << "ms)\n";
146
147     std::cout << "Parallel (16) Max: " << bench_traverse([&] { p_max(a, n); }) << "ms)\n\n";
148
149     std::cout << "Sequential Sum: " << bench_traverse([&] { s_sum(a, n); }) << "ms)\n";
150
151     std::cout << "Parallel (16) Sum: " << bench_traverse([&] { p_sum(a, n); }) << "ms)\n\n";
152
153     std::cout << "Sequential Average: " << bench_traverse([&] { s_avg(a, n); }) << "ms)\n";
154
155     std::cout << "Parallel (16) Average: " << bench_traverse([&] { p_avg(a, n); }) << "ms)\n";
156     return 0;
157 }
158
159 /*
160
161 OUTPUT:
162
163 Generated random array of length 100 with elements between 0 to 200
164
165 Given array is =>
166 183, 86, 177, 115, 193, 135, 186, 92, 49, 21, 162, 27, 90, 59, 163, 126, 140, 26, 172, 136, 11, 168,
167 167, 29, 182, 130, 62, 123, 67, 135, 129, 2, 22, 58, 69, 167, 193, 56, 11, 42, 29, 173, 21, 119,
168 184, 137, 198, 124, 115, 170, 13, 126, 91, 180, 156, 73, 62, 170, 196, 81, 105, 125, 84, 127, 136,
169 105, 46, 129, 113, 57, 124, 95, 182, 145, 14, 167, 34, 164, 43, 150, 87, 8, 76, 178, 188, 184, 3,
170 51, 154, 199, 132, 60, 76, 168, 139, 12, 26, 186, 94, 139,
171
172 Sequential Min: 2 (0ms)
173 Parallel (16) Min: 2 (0ms)
174
175 Sequential Max: 199 (0ms)
176 Parallel (16) Max: 199 (0ms)
177
178 Sequential Sum: 10884 (0ms)
179 Parallel (16) Sum: 10884 (1ms)
180
181 Sequential Average: 108 (0ms)
182 Parallel (16) Average: 108 (0ms)
183
184
185 OUTPUT:
186
187 Generated random array of length 100000000 with elements between 0 to 100000000
188
189 Sequential Min: 0 (185ms)
190 Parallel (16) Min: 0 (19ms)
191
192 Sequential Max: 99999999 (187ms)
193 Parallel (16) Max: 99999999 (18ms)
194
195 Sequential Sum: 4942469835882961 (191ms)
196 Parallel (16) Sum: 4942469835882961 (14ms)
197
198 Sequential Average: 49424698 (190ms)
199 Parallel (16) Average: 49424698 (14ms)
200
201
202 */
203

```



## HPC/4/matrixMul.cpp

```
1  #include <cmath>
2  #include <cstdlib>
3  #include <iostream>
4
5  #define checkCudaErrors(call) \
6      do { \
7          cudaError_t err = call; \
8          if (err != cudaSuccess) { \
9              printf("CUDA error at %s %d: %s\n", __FILE__, __LINE__, cudaGetErrorString(err)); \
10             exit(EXIT_FAILURE); \
11         } \
12     } while (0) \
13
14  using namespace std;
15
16  // Matrix multiplication Cuda
17  __global__ void matrixMultiplication(int *a, int *b, int *c, int n) {
18      int row = threadIdx.y + blockDim.y * blockIdx.y;
19      int col = threadIdx.x + blockDim.x * blockIdx.x;
20      int sum = 0;
21
22      if (row < n && col < n)
23          for (int j = 0; j < n; j++) {
24              sum = sum + a[row * n + j] * b[j * n + col];
25          }
26
27      c[n * row + col] = sum;
28  }
29
30  int main() {
31      int *a, *b, *c;
32      int *a_dev, *b_dev, *c_dev;
33      int n = 10;
34
35      a = new int[n * n];
36      b = new int[n * n];
37      c = new int[n * n];
38      int *d = new int[n * n];
39      int size = n * n * sizeof(int);
40      checkCudaErrors(cudaMalloc(&a_dev, size));
41      checkCudaErrors(cudaMalloc(&b_dev, size));
42      checkCudaErrors(cudaMalloc(&c_dev, size));
43
44      // Array initialization
45      for (int i = 0; i < n * n; i++) {
46          a[i] = rand() % 10;
47          b[i] = rand() % 10;
48      }
49
50      cout << "Given matrix A is =>\n";
51      for (int row = 0; row < n; row++) {
52          for (int col = 0; col < n; col++) {
53              cout << a[row * n + col] << " ";
54          }
55          cout << "\n";
56      }
57      cout << "\n";
58
59      cout << "Given matrix B is =>\n";
60      for (int row = 0; row < n; row++) {
61          for (int col = 0; col < n; col++) {
62              cout << b[row * n + col] << " ";
63          }
64          cout << "\n";
65      }
66      cout << "\n";
67
68      cudaEvent_t start, end;
69
70      checkCudaErrors(cudaEventCreate(&start));
```

```

71     checkCudaErrors(cudaEventCreate(&end));
72
73     checkCudaErrors(cudaMemcpy(a_dev, a, size, cudaMemcpyHostToDevice));
74     checkCudaErrors(cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice));
75
76     dim3 threadsPerBlock(n, n);
77     dim3 blocksPerGrid(1, 1);
78
79     // GPU Multiplication
80     checkCudaErrors(cudaEventRecord(start));
81     matrixMultiplication<<<blocksPerGrid, threadsPerBlock>>>(a_dev, b_dev, c_dev, n);
82
83     checkCudaErrors(cudaEventRecord(end));
84     checkCudaErrors(cudaEventSynchronize(end));
85
86     float time = 0.0;
87     checkCudaErrors(cudaEventElapsedTime(&time, start, end));
88
89     checkCudaErrors(cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost));
90
91     // CPU matrix multiplication
92     int sum = 0;
93     for (int row = 0; row < n; row++) {
94         for (int col = 0; col < n; col++) {
95             sum = 0;
96             for (int k = 0; k < n; k++) sum = sum + a[row * n + k] * b[k * n + col];
97             d[row * n + col] = sum;
98         }
99     }
100
101     cout << "CPU product is ⇒\n";
102     for (int row = 0; row < n; row++) {
103         for (int col = 0; col < n; col++) {
104             cout << d[row * n + col] << " ";
105         }
106         cout << "\n";
107     }
108     cout << "\n";
109
110     cout << "GPU product is ⇒\n";
111     for (int row = 0; row < n; row++) {
112         for (int col = 0; col < n; col++) {
113             cout << c[row * n + col] << " ";
114         }
115         cout << "\n";
116     }
117     cout << "\n";
118
119     int error = 0;
120     int _c, _d;
121     for (int row = 0; row < n; row++) {
122         for (int col = 0; col < n; col++) {
123             _c = c[row * n + col];
124             _d = d[row * n + col];
125             error += _c - _d;
126             if (0 ≠ (_c - _d)) {
127                 cout << "Error at (" << row << ", " << col << ") ⇒ GPU: " << _c << ", CPU: " << _d
128                     << "\n";
129             }
130         }
131     }
132     cout << "\n";
133
134     cout << "Error : " << error;
135     cout << "\nTime Elapsed: " << time;
136
137     return 0;
138 }
139
140 /*
141
142 OUTPUT:
143
144 Given matrix A is ⇒

```

```
145 3 7 3 6 9 2 0 3 0 2
146 1 7 2 2 7 9 2 9 3 1
147 9 1 4 8 5 3 1 6 2 6
148 5 4 6 6 3 4 2 4 4 3
149 7 6 8 3 4 2 6 9 6 4
150 5 4 7 7 7 2 1 6 5 4
151 0 1 7 1 9 7 7 6 6 9
152 8 2 3 0 8 0 6 8 6 1
153 9 4 1 3 4 4 7 3 7 9
154 2 7 5 4 8 9 5 8 3 8
155
156 Given matrix B is =>
157 6 5 5 2 1 7 9 6 6 6
158 8 9 0 3 5 2 8 7 6 2
159 3 9 7 4 0 6 0 3 0 1
160 5 7 5 9 7 5 5 7 4 0
161 8 8 4 1 9 0 8 2 6 9
162 0 8 1 2 2 6 0 1 9 9
163 9 7 1 5 7 6 3 5 3 4
164 1 9 9 8 5 9 3 5 1 5
165 8 8 0 0 4 4 6 1 5 6
166 1 8 7 1 5 7 3 8 1 9
167
168 CPU product is =>
169 190 278 145 132 190 136 200 169 161 167
170 186 355 156 157 207 209 185 164 210 246
171 191 335 233 179 196 257 220 227 174 232
172 191 319 172 156 167 218 182 186 165 186
173 276 433 239 205 229 305 251 252 193 257
174 233 378 222 181 218 240 231 216 180 226
175 232 430 221 155 255 274 187 203 193 328
176 248 319 178 137 201 217 233 171 165 236
177 267 379 184 141 231 276 259 247 218 301
178 252 477 239 204 282 302 239 261 245 334
179
180 GPU product is =>
181 190 278 145 132 190 136 200 169 161 167
182 186 355 156 157 207 209 185 164 210 246
183 191 335 233 179 196 257 220 227 174 232
184 191 319 172 156 167 218 182 186 165 186
185 276 433 239 205 229 305 251 252 193 257
186 233 378 222 181 218 240 231 216 180 226
187 232 430 221 155 255 274 187 203 193 328
188 248 319 178 137 201 217 233 171 165 236
189 267 379 184 141 231 276 259 247 218 301
190 252 477 239 204 282 302 239 261 245 334
191
192
193 Error : 0
194 Time Elapsed: 0.018144
195
196 */
197
```

## HPC/4/matrixVectorMul.cpp

```
1  #include <time.h>
2
3  #include <cmath>
4  #include <cstdlib>
5  #include <iostream>
6
7  #define checkCudaErrors(call) \
8      do { \
9          cudaError_t err = call; \
10         if (err != cudaSuccess) { \
11             printf("CUDA error at %s %d: %s\n", __FILE__, __LINE__, cudaGetErrorString(err)); \
12             exit(EXIT_FAILURE); \
13         } \
14     } while (0)
15
16 using namespace std;
17
18 __global__ void matrixVectorMultiplication(int *a, int *b, int *c, int n) {
19     int row = threadIdx.x + blockDim.x * blockIdx.x;
20     int sum = 0;
21
22     if (row < n)
23         for (int j = 0; j < n; j++) {
24             sum = sum + a[row * n + j] * b[j];
25         }
26
27     c[row] = sum;
28 }
29
30 int main() {
31     int *a, *b, *c;
32     int *a_dev, *b_dev, *c_dev;
33     int n = 10;
34
35     a = new int[n * n];
36     b = new int[n];
37     c = new int[n];
38     int *d = new int[n];
39     int size = n * sizeof(int);
40     checkCudaErrors(cudaMalloc(&a_dev, size * size));
41     checkCudaErrors(cudaMalloc(&b_dev, size));
42     checkCudaErrors(cudaMalloc(&c_dev, size));
43
44     for (int i = 0; i < n; i++) {
45         for (int j = 0; j < n; j++) {
46             a[i * n + j] = rand() % 10;
47         }
48         b[i] = rand() % 10;
49     }
50
51     cout << "Given matrix is =>\n";
52     for (int row = 0; row < n; row++) {
53         for (int col = 0; col < n; col++) {
54             cout << a[row * n + col] << " ";
55         }
56         cout << "\n";
57     }
58     cout << "\n";
59
60     cout << "Given vector is =>\n";
61     for (int i = 0; i < n; i++) {
62         cout << b[i] << ", ";
63     }
64     cout << "\n\n";
65
66     cudaEvent_t start, end;
67
68     checkCudaErrors(cudaEventCreate(&start));
69     checkCudaErrors(cudaEventCreate(&end));
```

```

71     checkCudaErrors(cudaMemcpy(a_dev, a, size * size, cudaMemcpyHostToDevice));
72     checkCudaErrors(cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice));
73
74     dim3 threadsPerBlock(n, n);
75     dim3 blocksPerGrid(1, 1);
76
77     checkCudaErrors(cudaEventRecord(start));
78     matrixVectorMultiplication<<<blocksPerGrid, threadsPerBlock>>>(a_dev, b_dev, c_dev, n);
79
80     checkCudaErrors(cudaEventRecord(end));
81     checkCudaErrors(cudaEventSynchronize(end));
82
83     float time = 0.0;
84     checkCudaErrors(cudaEventElapsedTime(&time, start, end));
85
86     checkCudaErrors(cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost));
87
88     // CPU matrixVector multiplication
89     int sum = 0;
90     for (int row = 0; row < n; row++) {
91         sum = 0;
92         for (int col = 0; col < n; col++) {
93             sum = sum + a[row * n + col] * b[col];
94         }
95         d[row] = sum;
96     }
97
98     cout << "CPU product is =>\n";
99     for (int i = 0; i < n; i++) {
100         cout << d[i] << ", ";
101     }
102     cout << "\n\n";
103
104     cout << "GPU product is =>\n";
105     for (int i = 0; i < n; i++) {
106         cout << c[i] << ", ";
107     }
108     cout << "\n\n";
109
110     int error = 0;
111     for (int i = 0; i < n; i++) {
112         error += d[i] - c[i];
113         if (0 != (d[i] - c[i])) {
114             cout << "Error at (" << i << ") => GPU: " << c[i] << ", CPU: " << d[i] << "\n";
115         }
116     }
117
118     cout << "Error: " << error;
119     cout << "\nTime Elapsed: " << time;
120
121     return 0;
122 }
123
124 /*
125
126 OUTPUT:
127
128 Given matrix is =>
129 3 6 7 5 3 5 6 2 9 1
130 7 0 9 3 6 0 6 2 6 1
131 7 9 2 0 2 3 7 5 9 2
132 8 9 7 3 6 1 2 9 3 1
133 4 7 8 4 5 0 3 6 1 0
134 3 2 0 6 1 5 5 4 7 6
135 6 9 3 7 4 5 2 5 4 7
136 4 3 0 7 8 6 8 8 4 3
137 4 9 2 0 6 8 9 2 6 6
138 9 5 0 4 8 7 1 7 2 7
139
140 Given vector is =>
141 2, 8, 2, 9, 6, 5, 4, 1, 4, 2,
142
143 CPU product is =>
144 220, 147, 190, 201, 168, 171, 245, 235, 234, 210,

```

```
145
146 GPU product is  $\Rightarrow$ 
147 220, 147, 190, 201, 168, 171, 245, 235, 234, 210,
148
149 Error: 0
150 Time Elapsed: 0.014336
151
152 */
153
```

## HPC/4/vectorAdd.cpp

```
1  #include <cstdlib>
2  #include <iostream>
3
4  #define checkCudaErrors(call) \
5      do { \
6          cudaError_t err = call; \
7          if (err != cudaSuccess) { \
8              printf("CUDA error at %s %d: %s\n", __FILE__, __LINE__, cudaGetErrorString(err)); \
9              exit(EXIT_FAILURE); \
10         } \
11     } while (0)
12
13 using namespace std;
14
15 // VectorAdd parallel function
16 __global__ void vectorAdd(int *a, int *b, int *result, int n) {
17     int tid = threadIdx.x + blockIdx.x * blockDim.x;
18     if (tid < n) {
19         result[tid] = a[tid] + b[tid];
20     }
21 }
22
23 int main() {
24     int *a, *b, *c;
25     int *a_dev, *b_dev, *c_dev;
26     int n = 1 << 4;
27
28     a = new int[n];
29     b = new int[n];
30     c = new int[n];
31     int *d = new int[n];
32     int size = n * sizeof(int);
33     checkCudaErrors(cudaMalloc(&a_dev, size));
34     checkCudaErrors(cudaMalloc(&b_dev, size));
35     checkCudaErrors(cudaMalloc(&c_dev, size));
36
37     // Array initialization..You can use Randon function to assign values
38     for (int i = 0; i < n; i++) {
39         a[i] = rand() % 1000;
40         b[i] = rand() % 1000;
41         d[i] = a[i] + b[i]; // calculating serial addition
42     }
43     cout << "Given array A is ⇒\n";
44     for (int i = 0; i < n; i++) {
45         cout << a[i] << ", ";
46     }
47     cout << "\n\n";
48
49     cout << "Given array B is ⇒\n";
50     for (int i = 0; i < n; i++) {
51         cout << b[i] << ", ";
52     }
53     cout << "\n\n";
54
55     cudaEvent_t start, end;
56
57     checkCudaErrors(cudaEventCreate(&start));
58     checkCudaErrors(cudaEventCreate(&end));
59
60     checkCudaErrors(cudaMemcpy(a_dev, a, size, cudaMemcpyHostToDevice));
61     checkCudaErrors(cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice));
62     int threads = 1024;
63     int blocks = (n + threads - 1) / threads;
64     checkCudaErrors(cudaEventRecord(start));
65
66     // Parallel addition program
67     vectorAdd<<<blocks, threads>>>(a_dev, b_dev, c_dev, n);
68
69     checkCudaErrors(cudaEventRecord(end));
70     checkCudaErrors(cudaEventSynchronize(end));
```

```

71
72     float time = 0.0;
73     checkCudaErrors(cudaEventElapsedTime(&time, start, end));
74
75     checkCudaErrors(cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost));
76
77     // Calculate the error term.
78
79     cout << "CPU sum is ⇒\n";
80     for (int i = 0; i < n; i++) {
81         cout << d[i] << ", ";
82     }
83     cout << "\n\n";
84
85     cout << "GPU sum is ⇒\n";
86     for (int i = 0; i < n; i++) {
87         cout << c[i] << ", ";
88     }
89     cout << "\n\n";
90
91     int error = 0;
92     for (int i = 0; i < n; i++) {
93         error += d[i] - c[i];
94         if (0 ≠ (d[i] - c[i])) {
95             cout << "Error at (" << i << ") ⇒ GPU: " << c[i] << ", CPU: " << d[i] << "\n";
96         }
97     }
98
99     cout << "\nError : " << error;
100    cout << "\nTime Elapsed: " << time;
101
102    return 0;
103 }
104
105 /*
106
107 OUTPUT:
108
109 Given array A is ⇒
110 383, 777, 793, 386, 649, 362, 690, 763, 540, 172, 211, 567, 782, 862, 67, 929,
111
112 Given array B is ⇒
113 886, 915, 335, 492, 421, 27, 59, 926, 426, 736, 368, 429, 530, 123, 135, 802,
114
115 CPU sum is ⇒
116 1269, 1692, 1128, 878, 1070, 389, 749, 1689, 966, 908, 579, 996, 1312, 985, 202, 1731,
117
118 GPU sum is ⇒
119 1269, 1692, 1128, 878, 1070, 389, 749, 1689, 966, 908, 579, 996, 1312, 985, 202, 1731,
120
121
122 Error : 0
123 Time Elapsed:  0.017408
124
125 */
126

```