HPC/1/graph.hpp

```cpp
#pragma once

#include <omp.h>

#include <fstream>
#include <functional>
#include <iostream>
#include <queue>
#include <sstream>
#include <string>
#include <tuple>
#include <vector>
#include <algorithm>

// 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
    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) {
```

```cpp
        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];

                    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]));

        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;
}

```

HPC/1/main.cpp

```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 << "Sequential iterative BFS: " << 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: "
```

```cpp
                        << 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 iterative BFS: " << 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: 21ms
Sequential recursive DFS: 13ms
Sequential iterative BFS: 23ms
Using 1 threads...
Parallel iterative DFS: 20ms
Parallel recursive DFS: 20ms
Parallel iterative BFS: 25ms
Using 2 threads...
Parallel iterative DFS: 15ms
Parallel recursive DFS: 12ms
Parallel iterative BFS: 29ms
Using 4 threads...
Parallel iterative DFS: 14ms
Parallel recursive DFS: 8ms
Parallel iterative BFS: 59ms
Using 8 threads...
Parallel iterative DFS: 14ms
Parallel recursive DFS: 6ms
Parallel iterative BFS: 86ms
Using 16 threads...
Parallel iterative DFS: 35ms
Parallel recursive DFS: 9ms
Parallel iterative BFS: 149ms
Using 32 threads...
Parallel iterative DFS: 81ms
Parallel recursive DFS: 11ms
Parallel iterative BFS: 191ms

*/

```

HPC/2/bubble_sort.cpp

```cpp
#include <omp.h>
#include <stdlib.h>

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

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

void s_bubble(int *, int);
void p_bubble(int *, int);
void swap(int &, int &);

void s_bubble(int *a, int n) {
    for (int i = 0; i < n; i++) {
        int first = i % 2;
        for (int j = first; j < n - 1; j += 2) {
            if (a[j] > a[j + 1]) {
                swap(a[j], a[j + 1]);
            }
        }
    }
}

void p_bubble(int *a, int n) {
    for (int i = 0; i < n; i++) {
        int first = i % 2;
#pragma omp parallel for shared(a, first) num_threads(16)
        for (int j = first; j < n - 1; j += 2) {
            if (a[j] > a[j + 1]) {
                swap(a[j], a[j + 1]);
            }
        }
    }
}

void swap(int &a, int &b) {
    int test;
    test = a;
    a = b;
    b = test;
}

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());
}

int main(int argc, const char **argv) {
    if (argc < 3) {
        std::cout << "Specify array length and maximum random value\n";
        return 1;
    }
    int *a, n, rand_max;

    n = stoi(argv[1]);
    rand_max = stoi(argv[2]);
    a = new int[n];

    for (int i = 0; i < n; i++) {
        a[i] = rand() % rand_max;
    }

    int *b = new int[n];
    copy(a, a + n, b);
    cout << "Generated random array of length " << n << " with elements between 0 to " << rand_max
         << "\n\n";

    std::cout << "Sequential Bubble sort: " << bench_traverse([&] { s_bubble(a, n); }) << "ms\n";
    cout << "Sorted array is =>\n";
    for (int i = 0; i < n; i++) {
        cout << a[i] << ", ";
    }
    cout << "\n\n";

    omp_set_num_threads(16);
    std::cout << "Parallel (16) Bubble sort: " << bench_traverse([&] { p_bubble(b, n); }) << "ms\n";
    cout << "Sorted array is =>\n";
    for (int i = 0; i < n; i++) {
        cout << b[i] << ", ";
    }
    return 0;
}

/*

OUTPUT:
Generated random array of length 100 with elements between 0 to 200

Sequential Bubble sort: 0ms
Sorted array is =>
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,
60, 62, 62, 67, 69, 73, 76, 76, 81, 84, 86, 87, 90, 91, 92, 94, 95, 105, 105, 113, 115, 115, 119,
123, 124, 124, 125, 126, 126, 127, 129, 129, 130, 132, 135, 135, 136, 136, 137, 139, 139, 140, 145,
150, 154, 156, 162, 163, 164, 167, 167, 167, 168, 168, 170, 170, 172, 173, 177, 178, 180, 182, 182,
183, 184, 184, 186, 186, 188, 193, 193, 196, 198, 199,

Parallel (16) Bubble sort: 1ms
Sorted array is =>
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,
60, 62, 62, 67, 69, 73, 76, 76, 81, 84, 86, 87, 90, 91, 92, 94, 95, 105, 105, 113, 115, 115, 119,
123, 124, 124, 125, 126, 126, 127, 129, 129, 130, 132, 135, 135, 136, 136, 137, 139, 139, 140, 145,
150, 154, 156, 162, 163, 164, 167, 167, 167, 168, 168, 170, 170, 172, 173, 177, 178, 180, 182, 182,
183, 184, 184, 186, 186, 188, 193, 193, 196, 198, 199,


OUTPUT:

Generated random array of length 100000 with elements between 0 to 100000

Sequential Bubble sort: 16878ms
Parallel (16) Bubble sort: 2914ms

*/

```

HPC/2/merge_sort.cpp

```cpp
#include <omp.h>
#include <stdlib.h>

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

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

void p_mergesort(int *a, int i, int j);
void s_mergesort(int *a, int i, int j);
void merge(int *a, int i1, int j1, int i2, int j2);

void p_mergesort(int *a, int i, int j) {
    int mid;
    if (i < j) {
        if ((j - i) > 1000) {
            mid = (i + j) / 2;

#pragma omp task firstprivate(a, i, mid)
            p_mergesort(a, i, mid);
#pragma omp task firstprivate(a, mid, j)
            p_mergesort(a, mid + 1, j);

#pragma omp taskwait
            merge(a, i, mid, mid + 1, j);
        } else {
            s_mergesort(a, i, j);
        }
    }
}

void parallel_mergesort(int *a, int i, int j) {
#pragma omp parallel num_threads(16)
    {
#pragma omp single
        p_mergesort(a, i, j);
    }
}

void s_mergesort(int *a, int i, int j) {
    int mid;
    if (i < j) {
        mid = (i + j) / 2;
        s_mergesort(a, i, mid);
        s_mergesort(a, mid + 1, j);
        merge(a, i, mid, mid + 1, j);
    }
}

void merge(int *a, int i1, int j1, int i2, int j2) {
    int temp[2000000];
    int i, j, k;
    i = i1;
    j = i2;
    k = 0;
    while (i <= j1 && j <= j2) {
        if (a[i] < a[j]) {
            temp[k++] = a[i++];
        } else {
            temp[k++] = a[j++];
        }
    }
    while (i <= j1) {
        temp[k++] = a[i++];
    }
    while (j <= j2) {
        temp[k++] = a[j++];
    }
    for (i = i1, j = 0; i <= j2; i++, j++) {
        a[i] = temp[j];
    }
}

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());
}

int main(int argc, const char **argv) {
    if (argc < 3) {
        std::cout << "Specify array length and maximum random value\n";
        return 1;
    }
    int *a, n, rand_max;

    n = stoi(argv[1]);
    rand_max = stoi(argv[2]);
    a = new int[n];

    for (int i = 0; i < n; i++) {
        a[i] = rand() % rand_max;
    }

    int *b = new int[n];
    copy(a, a + n, b);
    cout << "Generated random array of length " << n << " with elements between 0 to " << rand_max
         << "\n\n";

    std::cout << "Sequential Merge sort: " << bench_traverse([&] { s_mergesort(a, 0, n - 1); })
              << "ms\n";

    cout << "Sorted array is =>\n";
    for (int i = 0; i < n; i++) {
        cout << a[i] << ", ";
    }
    cout << "\n\n";

    omp_set_num_threads(16);
    std::cout << "Parallel (16) Merge sort: "
              << bench_traverse([&] { parallel_mergesort(b, 0, n - 1); }) << "ms\n";

    cout << "Sorted array is =>\n";
    for (int i = 0; i < n; i++) {
        cout << b[i] << ", ";
    }
    return 0;
}

/*

OUTPUT:

Generated random array of length 100 with elements between 0 to 200

Sequential Merge sort: 0ms
Sorted array is =>
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,
60, 62, 62, 67, 69, 73, 76, 76, 81, 84, 86, 87, 90, 91, 92, 94, 95, 105, 105, 113, 115, 115, 119,
123, 124, 124, 125, 126, 126, 127, 129, 129, 130, 132, 135, 135, 136, 136, 137, 139, 139, 140, 145,
150, 154, 156, 162, 163, 164, 167, 167, 167, 168, 168, 170, 170, 172, 173, 177, 178, 180, 182, 182,
183, 184, 184, 186, 186, 188, 193, 193, 196, 198, 199,

Parallel (16) Merge sort: 1ms
Sorted array is ⇒
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,
60, 62, 62, 67, 69, 73, 76, 76, 81, 84, 86, 87, 90, 91, 92, 94, 95, 105, 105, 113, 115, 115, 119,
123, 124, 124, 125, 126, 126, 127, 129, 129, 130, 132, 135, 135, 136, 136, 137, 139, 139, 140, 145,
150, 154, 156, 162, 163, 164, 167, 167, 167, 168, 168, 170, 170, 172, 173, 177, 178, 180, 182, 182,
183, 184, 184, 186, 186, 188, 193, 193, 196, 198, 199,


OUTPUT:

Generated random array of length 1000000 with elements between 0 to 1000000

Sequential Merge sort: 165ms
Parallel (16) Merge sort: 42ms

*/
```

HPC/3/statistics.cpp

```cpp
#include <limits.h>
#include <omp.h>
#include <stdlib.h>

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

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

void s_avg(int arr[], int n) {
    long sum = 0L;
    int i;
    for (i = 0; i < n; i++) {
        sum = sum + arr[i];
    }
    cout << sum / long(n);
}

void p_avg(int arr[], int n) {
    long sum = 0L;
    int i;
#pragma omp parallel for reduction(+ : sum) num_threads(16)
    for (i = 0; i < n; i++) {
        sum = sum + arr[i];
    }
    cout << sum / long(n);
}

void s_sum(int arr[], int n) {
    long sum = 0L;
    int i;
    for (i = 0; i < n; i++) {
        sum = sum + arr[i];
    }
    cout << sum;
}

void p_sum(int arr[], int n) {
    long sum = 0L;
    int i;
#pragma omp parallel for reduction(+ : sum) num_threads(16)
    for (i = 0; i < n; i++) {
        sum = sum + arr[i];
    }
    cout << sum;
}

void s_max(int arr[], int n) {
    int max_val = INT_MIN;
    int i;
    for (i = 0; i < n; i++) {
        if (arr[i] > max_val) {
            max_val = arr[i];
        }
    }
    cout << max_val;
}

void p_max(int arr[], int n) {
    int max_val = INT_MIN;
    int i;
#pragma omp parallel for reduction(max : max_val) num_threads(16)
    for (i = 0; i < n; i++) {
        if (arr[i] > max_val) {
            max_val = arr[i];
        }
    }
    cout << max_val;
}

void s_min(int arr[], int n) {
    int min_val = INT_MAX;
    int i;
    for (i = 0; i < n; i++) {
        if (arr[i] < min_val) {
            min_val = arr[i];
        }
    }
    cout << min_val;
}

void p_min(int arr[], int n) {
    int min_val = INT_MAX;
    int i;
#pragma omp parallel for reduction(min : min_val) num_threads(16)
    for (i = 0; i < n; i++) {
        if (arr[i] < min_val) {
            min_val = arr[i];
        }
    }
    cout << min_val;
}

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

int main(int argc, const char **argv) {
    if (argc < 3) {
        std::cout << "Specify array length and maximum random value\n";
        return 1;
    }
    int *a, n, rand_max;

    n = stoi(argv[1]);
    rand_max = stoi(argv[2]);
    a = new int[n];

    for (int i = 0; i < n; i++) {
        a[i] = rand() % rand_max;
    }

    cout << "Generated random array of length " << n << " with elements between 0 to " << rand_max
         << "\n\n";
    cout << "Given array is =>\n";
    for (int i = 0; i < n; i++) {
        cout << a[i] << ", ";
    }
    cout << "\n\n";

    omp_set_num_threads(16);

    std::cout << "Sequential Min: " << bench_traverse([&] { s_min(a, n); }) << "ms)\n";

    std::cout << "Parallel (16) Min: " << bench_traverse([&] { p_min(a, n); }) << "ms)\n\n";

    std::cout << "Sequential Max: " << bench_traverse([&] { s_max(a, n); }) << "ms)\n";

    std::cout << "Parallel (16) Max: " << bench_traverse([&] { p_max(a, n); }) << "ms)\n\n";

    std::cout << "Sequential Sum: " << bench_traverse([&] { s_sum(a, n); }) << "ms)\n";

    std::cout << "Parallel (16) Sum: " << bench_traverse([&] { p_sum(a, n); }) << "ms)\n\n";

    std::cout << "Sequential Average: " << bench_traverse([&] { s_avg(a, n); }) << "ms)\n";

    std::cout << "Parallel (16) Average: " << bench_traverse([&] { p_avg(a, n); }) << "ms)\n";
    return 0;
}

/*

OUTPUT:

Generated random array of length 100 with elements between 0 to 200

Given array is =>
183, 86, 177, 115, 193, 135, 186, 92, 49, 21, 162, 27, 90, 59, 163, 126, 140, 26, 172, 136, 11, 168,
167, 29, 182, 130, 62, 123, 67, 135, 129, 2, 22, 58, 69, 167, 193, 56, 11, 42, 29, 173, 21, 119,
184, 137, 198, 124, 115, 170, 13, 126, 91, 180, 156, 73, 62, 170, 196, 81, 105, 125, 84, 127, 136,
105, 46, 129, 113, 57, 124, 95, 182, 145, 14, 167, 34, 164, 43, 150, 87, 8, 76, 178, 188, 184, 3,
51, 154, 199, 132, 60, 76, 168, 139, 12, 26, 186, 94, 139,

Sequential Min: 2 (0ms)
Parallel (16) Min: 2 (0ms)

Sequential Max: 199 (0ms)
Parallel (16) Max: 199 (0ms)

Sequential Sum: 10884 (0ms)
Parallel (16) Sum: 10884 (1ms)

Sequential Average: 108 (0ms)
Parallel (16) Average: 108 (0ms)


OUTPUT:

Generated random array of length 100000000 with elements between 0 to 100000000

Sequential Min: 0 (185ms)
Parallel (16) Min: 0 (19ms)

Sequential Max: 99999999 (187ms)
Parallel (16) Max: 99999999 (18ms)

Sequential Sum: 4942469835882961 (191ms)
Parallel (16) Sum: 4942469835882961 (14ms)

Sequential Average: 49424698 (190ms)
Parallel (16) Average: 49424698 (14ms)

*/

```

HPC/4/matrixMul.cpp

```cpp
#include <cmath>
#include <cstdlib>
#include <iostream>

#define checkCudaErrors(call)                                                                \
    do {                                                                                     \
        cudaError_t err = call;                                                              \
        if (err != cudaSuccess) {                                                            \
            printf("CUDA error at %s %d: %s\n", __FILE__, __LINE__, cudaGetErrorString(err)); \
            exit(EXIT_FAILURE);                                                              \
        }                                                                                    \
    } while (0)

using namespace std;

// Matrix multiplication Cuda
__global__ void matrixMultiplication(int *a, int *b, int *c, int n) {
    int row = threadIdx.y + blockDim.y * blockIdx.y;
    int col = threadIdx.x + blockDim.x * blockIdx.x;
    int sum = 0;

    if (row < n && col < n)
        for (int j = 0; j < n; j++) {
            sum = sum + a[row * n + j] * b[j * n + col];
        }

    c[n * row + col] = sum;
}

int main() {
    int *a, *b, *c;
    int *a_dev, *b_dev, *c_dev;
    int n = 10;

    a = new int[n * n];
    b = new int[n * n];
    c = new int[n * n];
    int *d = new int[n * n];
    int size = n * n * sizeof(int);
    checkCudaErrors(cudaMalloc(&a_dev, size));
    checkCudaErrors(cudaMalloc(&b_dev, size));
    checkCudaErrors(cudaMalloc(&c_dev, size));

    // Array initialization
    for (int i = 0; i < n * n; i++) {
        a[i] = rand() % 10;
        b[i] = rand() % 10;
    }

    cout << "Given matrix A is =>\n";
    for (int row = 0; row < n; row++) {
        for (int col = 0; col < n; col++) {
            cout << a[row * n + col] << " ";
        }
        cout << "\n";
    }
    cout << "\n";

    cout << "Given matrix B is =>\n";
    for (int row = 0; row < n; row++) {
        for (int col = 0; col < n; col++) {
            cout << b[row * n + col] << " ";
        }
        cout << "\n";
    }
    cout << "\n";

    cudaEvent_t start, end;

    checkCudaErrors(cudaEventCreate(&start));
    checkCudaErrors(cudaEventCreate(&end));

    checkCudaErrors(cudaMemcpy(a_dev, a, size, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice));

    dim3 threadsPerBlock(n, n);
    dim3 blocksPerGrid(1, 1);

    // GPU Multiplication
    checkCudaErrors(cudaEventRecord(start));
    matrixMultiplication<<<blocksPerGrid, threadsPerBlock>>>(a_dev, b_dev, c_dev, n);

    checkCudaErrors(cudaEventRecord(end));
    checkCudaErrors(cudaEventSynchronize(end));

    float time = 0.0;
    checkCudaErrors(cudaEventElapsedTime(&time, start, end));

    checkCudaErrors(cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost));

    // CPU matrix multiplication
    int sum = 0;
    for (int row = 0; row < n; row++) {
        for (int col = 0; col < n; col++) {
            sum = 0;
            for (int k = 0; k < n; k++) sum = sum + a[row * n + k] * b[k * n + col];
            d[row * n + col] = sum;
        }
    }

    cout << "CPU product is ⇒\n";
    for (int row = 0; row < n; row++) {
        for (int col = 0; col < n; col++) {
            cout << d[row * n + col] << " ";
        }
        cout << "\n";
    }
    cout << "\n";

    cout << "GPU product is ⇒\n";
    for (int row = 0; row < n; row++) {
        for (int col = 0; col < n; col++) {
            cout << c[row * n + col] << " ";
        }
        cout << "\n";
    }
    cout << "\n";

    int error = 0;
    int _c, _d;
    for (int row = 0; row < n; row++) {
        for (int col = 0; col < n; col++) {
            _c = c[row * n + col];
            _d = d[row * n + col];
            error += _c - _d;
            if (0 ≠ (_c - _d)) {
                cout << "Error at (" << row << ", " << col << ") ⇒ GPU: " << _c << ", CPU: " << _d
                     << "\n";
            }
        }
    }
    cout << "\n";

    cout << "Error : " << error;
    cout << "\nTime Elapsed: " << time;

    return 0;
}

/*

OUTPUT:

Given matrix A is ⇒
```

```
3 7 3 6 9 2 0 3 0 2
1 7 2 2 7 9 2 9 3 1
9 1 4 8 5 3 1 6 2 6
5 4 6 6 3 4 2 4 4 3
7 6 8 3 4 2 6 9 6 4
5 4 7 7 7 2 1 6 5 4
0 1 7 1 9 7 7 6 6 9
8 2 3 0 8 0 6 8 6 1
9 4 1 3 4 4 7 3 7 9
2 7 5 4 8 9 5 8 3 8

Given matrix B is ⇒
6 5 5 2 1 7 9 6 6 6
8 9 0 3 5 2 8 7 6 2
3 9 7 4 0 6 0 3 0 1
5 7 5 9 7 5 5 7 4 0
8 8 4 1 9 0 8 2 6 9
0 8 1 2 2 6 0 1 9 9
9 7 1 5 7 6 3 5 3 4
1 9 9 8 5 9 3 5 1 5
8 8 0 0 4 4 6 1 5 6
1 8 7 1 5 7 3 8 1 9

CPU product is ⇒
190 278 145 132 190 136 200 169 161 167
186 355 156 157 207 209 185 164 210 246
191 335 233 179 196 257 220 227 174 232
191 319 172 156 167 218 182 186 165 186
276 433 239 205 229 305 251 252 193 257
233 378 222 181 218 240 231 216 180 226
232 430 221 155 255 274 187 203 193 328
248 319 178 137 201 217 233 171 165 236
267 379 184 141 231 276 259 247 218 301
252 477 239 204 282 302 239 261 245 334

GPU product is ⇒
190 278 145 132 190 136 200 169 161 167
186 355 156 157 207 209 185 164 210 246
191 335 233 179 196 257 220 227 174 232
191 319 172 156 167 218 182 186 165 186
276 433 239 205 229 305 251 252 193 257
233 378 222 181 218 240 231 216 180 226
232 430 221 155 255 274 187 203 193 328
248 319 178 137 201 217 233 171 165 236
267 379 184 141 231 276 259 247 218 301
252 477 239 204 282 302 239 261 245 334


Error : 0
Time Elapsed: 0.018144

*/
```

HPC/4/matrixVectorMul.cpp

```cpp
#include <time.h>

#include <cmath>
#include <cstdlib>
#include <iostream>

#define checkCudaErrors(call)                                                                \
    do {                                                                                      \
        cudaError_t err = call;                                                               \
        if (err != cudaSuccess) {                                                             \
            printf("CUDA error at %s %d: %s\n", __FILE__, __LINE__, cudaGetErrorString(err)); \
            exit(EXIT_FAILURE);                                                               \
        }                                                                                     \
    } while (0)

using namespace std;

__global__ void matrixVectorMultiplication(int *a, int *b, int *c, int n) {
    int row = threadIdx.x + blockDim.x * blockIdx.x;
    int sum = 0;

    if (row < n)
        for (int j = 0; j < n; j++) {
            sum = sum + a[row * n + j] * b[j];
        }

    c[row] = sum;
}

int main() {
    int *a, *b, *c;
    int *a_dev, *b_dev, *c_dev;
    int n = 10;

    a = new int[n * n];
    b = new int[n];
    c = new int[n];
    int *d = new int[n];
    int size = n * sizeof(int);
    checkCudaErrors(cudaMalloc(&a_dev, size * size));
    checkCudaErrors(cudaMalloc(&b_dev, size));
    checkCudaErrors(cudaMalloc(&c_dev, size));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            a[i * n + j] = rand() % 10;
        }
        b[i] = rand() % 10;
    }

    cout << "Given matrix is =>\n";
    for (int row = 0; row < n; row++) {
        for (int col = 0; col < n; col++) {
            cout << a[row * n + col] << " ";
        }
        cout << "\n";
    }
    cout << "\n";

    cout << "Given vector is =>\n";
    for (int i = 0; i < n; i++) {
        cout << b[i] << ", ";
    }
    cout << "\n\n";

    cudaEvent_t start, end;

    checkCudaErrors(cudaEventCreate(&start));
    checkCudaErrors(cudaEventCreate(&end));

    checkCudaErrors(cudaMemcpy(a_dev, a, size * size, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice));

    dim3 threadsPerBlock(n, n);
    dim3 blocksPerGrid(1, 1);

    checkCudaErrors(cudaEventRecord(start));
    matrixVectorMultiplication<<<blocksPerGrid, threadsPerBlock>>>(a_dev, b_dev, c_dev, n);

    checkCudaErrors(cudaEventRecord(end));
    checkCudaErrors(cudaEventSynchronize(end));

    float time = 0.0;
    checkCudaErrors(cudaEventElapsedTime(&time, start, end));

    checkCudaErrors(cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost));

    // CPU matrixVector multiplication
    int sum = 0;
    for (int row = 0; row < n; row++) {
        sum = 0;
        for (int col = 0; col < n; col++) {
            sum = sum + a[row * n + col] * b[col];
        }
        d[row] = sum;
    }

    cout << "CPU product is =>\n";
    for (int i = 0; i < n; i++) {
        cout << d[i] << ", ";
    }
    cout << "\n\n";

    cout << "GPU product is =>\n";
    for (int i = 0; i < n; i++) {
        cout << c[i] << ", ";
    }
    cout << "\n\n";

    int error = 0;
    for (int i = 0; i < n; i++) {
        error += d[i] - c[i];
        if (0 != (d[i] - c[i])) {
            cout << "Error at (" << i << ") => GPU: " << c[i] << ", CPU: " << d[i] << "\n";
        }
    }

    cout << "Error: " << error;
    cout << "\nTime Elapsed: " << time;

    return 0;
}

/*

OUTPUT:

Given matrix is =>
3 6 7 5 3 5 6 2 9 1
7 0 9 3 6 0 6 2 6 1
7 9 2 0 2 3 7 5 9 2
8 9 7 3 6 1 2 9 3 1
4 7 8 4 5 0 3 6 1 0
3 2 0 6 1 5 5 4 7 6
6 9 3 7 4 5 2 5 4 7
4 3 0 7 8 6 8 8 4 3
4 9 2 0 6 8 9 2 6 6
9 5 0 4 8 7 1 7 2 7

Given vector is =>
2, 8, 2, 9, 6, 5, 4, 1, 4, 2,

CPU product is =>
220, 147, 190, 201, 168, 171, 245, 235, 234, 210,

GPU product is ⇒
220, 147, 190, 201, 168, 171, 245, 235, 234, 210,

Error: 0
Time Elapsed: 0.014336

*/

```

HPC/4/vectorAdd.cpp

```cpp
#include <cstdlib>
#include <iostream>

#define checkCudaErrors(call)                                                                \
    do {                                                                                     \
        cudaError_t err = call;                                                              \
        if (err != cudaSuccess) {                                                            \
            printf("CUDA error at %s %d: %s\n", __FILE__, __LINE__, cudaGetErrorString(err)); \
            exit(EXIT_FAILURE);                                                              \
        }                                                                                    \
    } while (0)

using namespace std;

// VectorAdd parallel function
__global__ void vectorAdd(int *a, int *b, int *result, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n) {
        result[tid] = a[tid] + b[tid];
    }
}

int main() {
    int *a, *b, *c;
    int *a_dev, *b_dev, *c_dev;
    int n = 1 << 4;

    a = new int[n];
    b = new int[n];
    c = new int[n];
    int *d = new int[n];
    int size = n * sizeof(int);
    checkCudaErrors(cudaMalloc(&a_dev, size));
    checkCudaErrors(cudaMalloc(&b_dev, size));
    checkCudaErrors(cudaMalloc(&c_dev, size));

    // Array initialization..You can use Randon function to assign values
    for (int i = 0; i < n; i++) {
        a[i] = rand() % 1000;
        b[i] = rand() % 1000;
        d[i] = a[i] + b[i];  // calculating serial addition
    }
    cout << "Given array A is =>\n";
    for (int i = 0; i < n; i++) {
        cout << a[i] << ", ";
    }
    cout << "\n\n";

    cout << "Given array B is =>\n";
    for (int i = 0; i < n; i++) {
        cout << b[i] << ", ";
    }
    cout << "\n\n";

    cudaEvent_t start, end;

    checkCudaErrors(cudaEventCreate(&start));
    checkCudaErrors(cudaEventCreate(&end));

    checkCudaErrors(cudaMemcpy(a_dev, a, size, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice));
    int threads = 1024;
    int blocks = (n + threads - 1) / threads;
    checkCudaErrors(cudaEventRecord(start));

    // Parallel addition program
    vectorAdd<<<blocks, threads>>>(a_dev, b_dev, c_dev, n);

    checkCudaErrors(cudaEventRecord(end));
    checkCudaErrors(cudaEventSynchronize(end));

    float time = 0.0;
    checkCudaErrors(cudaEventElapsedTime(&time, start, end));

    checkCudaErrors(cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost));

    // Calculate the error term.

    cout << "CPU sum is =>\n";
    for (int i = 0; i < n; i++) {
        cout << d[i] << ", ";
    }
    cout << "\n\n";

    cout << "GPU sum is =>\n";
    for (int i = 0; i < n; i++) {
        cout << c[i] << ", ";
    }
    cout << "\n\n";

    int error = 0;
    for (int i = 0; i < n; i++) {
        error += d[i] - c[i];
        if (0 != (d[i] - c[i])) {
            cout << "Error at (" << i << ") => GPU: " << c[i] << ", CPU: " << d[i] << "\n";
        }
    }

    cout << "\nError : " << error;
    cout << "\nTime Elapsed: " << time;

    return 0;
}

/*

OUTPUT:

Given array A is =>
383, 777, 793, 386, 649, 362, 690, 763, 540, 172, 211, 567, 782, 862, 67, 929,

Given array B is =>
886, 915, 335, 492, 421, 27, 59, 926, 426, 736, 368, 429, 530, 123, 135, 802,

CPU sum is =>
1269, 1692, 1128, 878, 1070, 389, 749, 1689, 966, 908, 579, 996, 1312, 985, 202, 1731,

GPU sum is =>
1269, 1692, 1128, 878, 1070, 389, 749, 1689, 966, 908, 579, 996, 1312, 985, 202, 1731,


Error : 0
Time Elapsed:  0.017408

*/

```