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

*/
```