

```

1 // BUBBLE_SORT.CPP
2
3 #include <omp.h>
4 #include <stdlib.h>
5
6 #include <array>
7 #include <chrono>
8 #include <functional>
9 #include <iostream>
10 #include <string>
11 #include <vector>
12
13 using std::chrono::duration_cast;
14 using std::chrono::high_resolution_clock;
15 using std::chrono::milliseconds;
16 using namespace std;
17
18 void s_bubble(int *, int);
19 void p_bubble(int *, int);
20 void swap(int &, int &);
21
22 void s_bubble(int *a, int n) {
23     for (int i = 0; i < n; i++) {
24         int first = i % 2;
25         for (int j = first; j < n - 1; j += 2) {
26             if (a[j] > a[j + 1]) {
27                 swap(a[j], a[j + 1]);
28             }
29         }
30     }
31 }
32
33 void p_bubble(int *a, int n) {
34     for (int i = 0; i < n; i++) {
35         int first = i % 2;
36 #pragma omp parallel for shared(a, first) num_threads(16)
37         for (int j = first; j < n - 1; j += 2) {
38             if (a[j] > a[j + 1]) {
39                 swap(a[j], a[j + 1]);
40             }
41         }
42     }
43 }
44
45 void swap(int &a, int &b) {
46     int test;
47     test = a;
48     a = b;
49     b = test;
50 }
51
52 std::string bench_traverse(std::function<void()> traverse_fn) {
53     auto start = high_resolution_clock::now();
54     traverse_fn();
55     auto stop = high_resolution_clock::now();
56
57     // Subtract stop and start timepoints and cast it to required unit.
58     // Predefined units are nanoseconds, microseconds, milliseconds, seconds,
59     // minutes, hours. Use duration_cast() function.
60     auto duration = duration_cast<milliseconds>(stop - start);
61
62     // To get the value of duration use the count() member function on the
63     // duration object
64     return std::to_string(duration.count());
65 }
66
67 int main(int argc, const char **argv) {
68     if (argc < 2) {
69         std::cout << "Specify array length.\n";
70         return 1;
71     }
72     int *a, n;
73     n = stoi(argv[1]);
74     a = new int[n];

```

```

75
76     for (int i = 0; i < n; i++) {
77         a[i] = rand() % n;
78     }
79
80     int *b = new int[n];
81     copy(a, a + n, b);
82     cout << "Generated random array of length " << n << "\n\n";
83
84     std::cout << "Sequential Bubble sort: " << bench_traverse([&] { s_bubble(a, n); }) << "ms\n";
85
86     omp_set_num_threads(16);
87     std::cout << "Parallel (16) Bubble sort: " << bench_traverse([&] { p_bubble(b, n); }) << "ms\n";
88
89     // cout << "Sorted array is =>\n";
90     // for (int i = 0; i < n; i++) {
91     //     cout << a[i] << endl;
92     // }
93     return 0;
94 }
95
96 /*
97
98 OUTPUT:
99
100 Generated random array of length 100000
101
102 Sequential Bubble sort: 50038ms
103 Parallel (16) Bubble sort: 10608ms
104
105 */

```

```

1 //MERGE_SORT.CPP
2
3 #include <omp.h>
4 #include <stdlib.h>
5
6 #include <array>
7 #include <chrono>
8 #include <functional>
9 #include <iostream>
10 #include <string>
11 #include <vector>
12
13 using std::chrono::duration_cast;
14 using std::chrono::high_resolution_clock;
15 using std::chrono::milliseconds;
16 using namespace std;
17
18 void p_mergesort(int a[], int i, int j);
19 void s_mergesort(int a[], int i, int j);
20 void merge(int a[], int i1, int j1, int i2, int j2);
21
22 void p_mergesort(int a[], int i, int j) {
23     int mid;
24     if (i < j) {
25         if ((j - i) > 1000) {
26             mid = (i + j) / 2;
27
28 #pragma omp task firstprivate(a, i, mid)
29             p_mergesort(a, i, mid);
30 #pragma omp task firstprivate(a, mid, j)
31             p_mergesort(a, mid + 1, j);
32
33 #pragma omp taskwait
34             merge(a, i, mid, mid + 1, j);
35         }
36     } else {
37         s_mergesort(a, i, j);
38     }
39 }
40
41 void parallel_mergesort(int a[], int i, int j) {
42 #pragma omp parallel num_threads(16)
43 {
44 #pragma omp single
45     { p_mergesort(a, i, j); }
46 }
47 }
48
49 void s_mergesort(int a[], int i, int j) {
50     int mid;
51     if (i < j) {
52         mid = (i + j) / 2;
53         s_mergesort(a, i, mid);
54         s_mergesort(a, mid + 1, j);
55         merge(a, i, mid, mid + 1, j);
56     }
57 }
58
59 void merge(int a[], int i1, int j1, int i2, int j2) {
60     int temp[1000000];
61     int i, j, k;
62     i = i1;
63     j = i2;
64     k = 0;
65     while (i <= j1 && j <= j2) {
66         if (a[i] < a[j]) {
67             temp[k++] = a[i++];
68         } else {
69             temp[k++] = a[j++];
70         }
71     }
72     while (i <= j1) {
73         temp[k++] = a[i++];
74     }

```

```

75     while (j <= j2) {
76         temp[k++] = a[j++];
77     }
78     for (i = i1, j = 0; i <= j2; i++, j++) {
79         a[i] = temp[j];
80     }
81 }
82
83 std::string bench_traverse(std::function<void()> traverse_fn) {
84     auto start = high_resolution_clock::now();
85     traverse_fn();
86     auto stop = high_resolution_clock::now();
87
88     // Subtract stop and start timepoints and cast it to required unit.
89     // Predefined units are nanoseconds, microseconds, milliseconds, seconds,
90     // minutes, hours. Use duration_cast() function.
91     auto duration = duration_cast<milliseconds>(stop - start);
92
93     // To get the value of duration use the count() member function on the
94     // duration object
95     return std::to_string(duration.count());
96 }
97
98 int main(int argc, const char **argv) {
99     if (argc < 2) {
100         std::cout << "Specify array length.\n";
101         return 1;
102     }
103     int *a, n, i;
104
105     n = stoi(argv[1]);
106     a = new int[n];
107
108     for (int i = 0; i < n; i++) {
109         a[i] = rand() % n;
110     }
111
112     int *b = new int[n];
113     copy(a, a + n, b);
114     cout << "Generated random array of length " << n << "\n\n";
115
116     std::cout << "Sequential Merge sort: " << bench_traverse([&] { s_mergesort(a, 0, n - 1); })
117         << "ms\n";
118
119     omp_set_num_threads(16);
120     std::cout << "Parallel (16) Merge sort: "
121         << bench_traverse([&] { parallel_mergesort(b, 0, n - 1); }) << "ms\n";
122
123     // cout << "\nSorted array is =>";
124     // for (i = 0; i < n; i++) {
125     //     cout << "\n" << a[i];
126     // }
127     return 0;
128 }
129
130 /*
131
132 OUTPUT:
133
134 Generated random array of length 1000000
135
136 Sequential Merge sort: 532ms
137 Parallel (16) Merge sort: 62ms
138
139 */

```