

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
```