

Sortowania równoległe i sekwencyjne

Dorota Kapturkiewicz
Wiktor Kuropatwa
Karol Różycki

1 Cel projektu

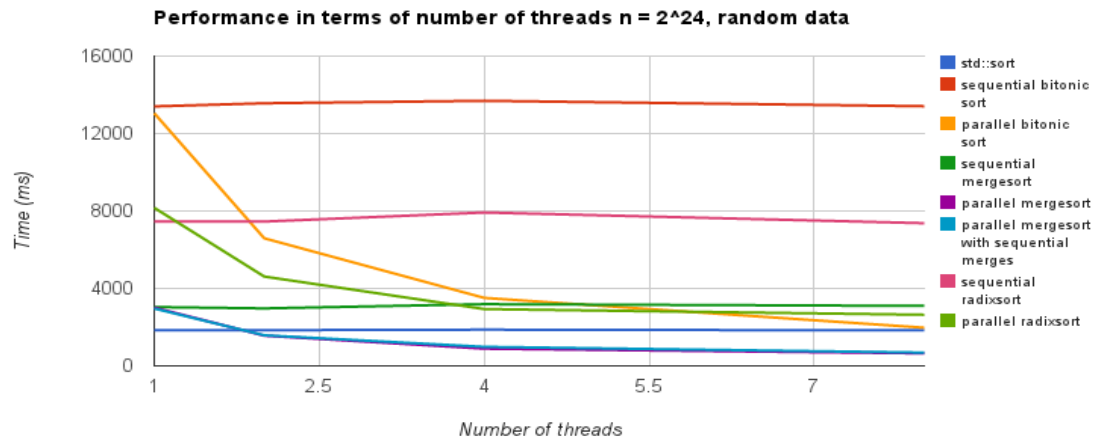
Celem poniższego projektu jest porównanie efektywności i czasu działania kilku algorytmów sortowania z różnym stopniem zrównoleglenia.

2 Zaimplementowane algorytmy

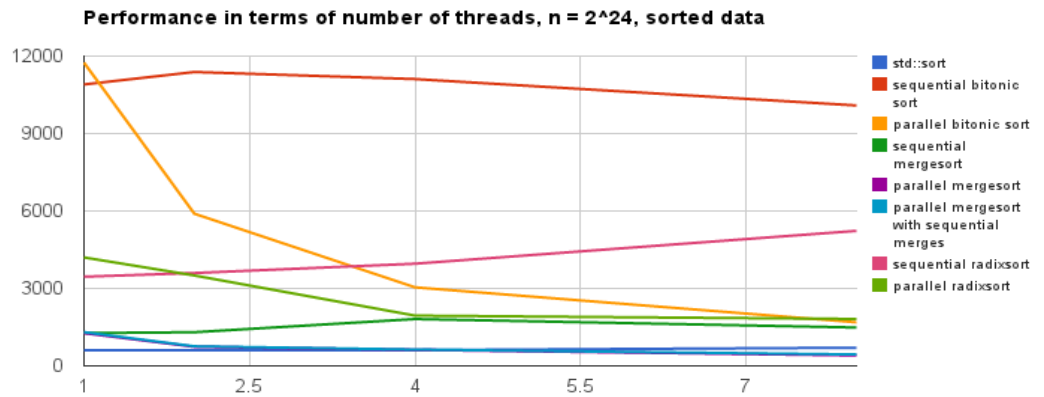
Projekt został zaimplementowany w języku C++ (w standardzie c++ 11). Do fragmentów wielowątkowych użyliśmy biblioteki OpenMP. Dana praca porównuje działania następujących algorytmów:

- merge sort
 - w pełni sekwencyjny
 - równoległe wywołania rekurencyjne sortowania, sekwencyjne złączanie
 - równoległe sortowanie i złączanie
- bitonic sort
 - w pełni sekwencyjny
 - zrównoleglone wywołania rekurencyjne i porównywanie wartości w bitonic merge'u
- radix sort
 - w pełni sekwencyjny
 - zrównoleglony prefix sum, przepisywanie wartości i przygotowywanie danych (xor)

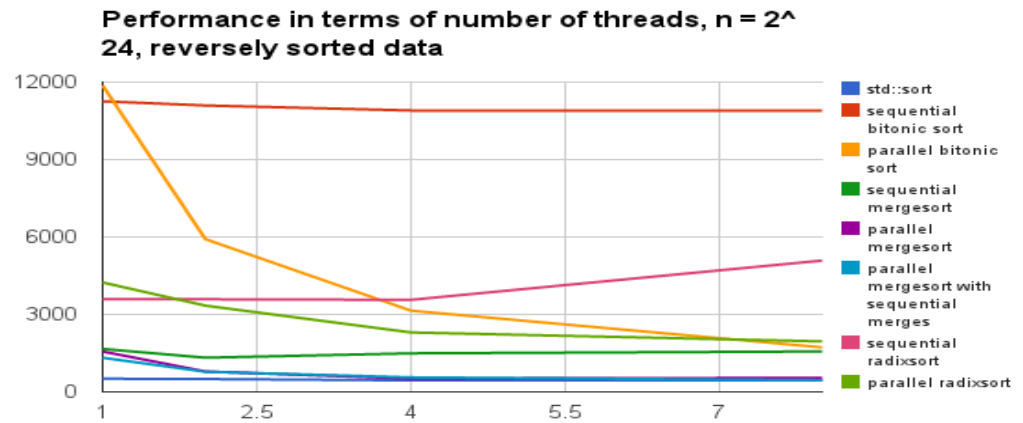
3 Przeprowadzone testy



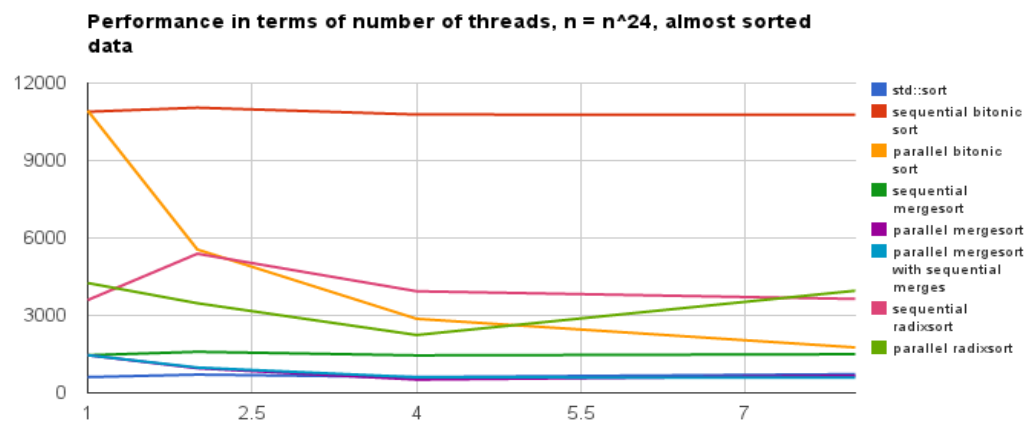
Rysunek 1: Czas działania w zależności od liczby wątków na losowych danych



Rysunek 2: Czas działania w zależności od liczby wątków na posortowanych danych



Rysunek 3: Czas działania w zależności od liczby wątków na odwrotnie posortowanych danych



Rysunek 4: Czas działania w zależności od liczby wątków na prawie posortowanych danych

4 Wnioski

- Algorytmy z elementami równoległymi działają lepiej niż sekwencyjne na większej liczbie wątków.
- Zrównoleglony mergesort na 2, 4 oraz 8 wątkach działa szybciej niż `std::sort`. W przypadku pozostałych algorytmów również możemy odnotować znaczący wzrost wydajności.
- Przy użyciu jednego wątku algorytmy równoległe często wykazują gorszy czas działania niż sekwencyjne.
- Równoległe bitoniczne sortowanie pomimo gorszej złożoności ($O(n \log^2 n)$) jest w stanie wyprzedzić sekwencyjne algorytmy o złożoności $O(n \log n)$.
- Ze względu na małą liczbę wątków prefix sum użyty w równoległym algorytmie radixsort nie był w stanie w pełni wykazać swojej mocy, ale i tak można było zauważyć około czterokrotne przyspieszenie w stosunku do sekwencyjnego.
- Równoległy algorytm merge sort okazał się być najszybszym z całej stawki, wyprzedzając nawet algorytm `std::sort`.

5 Źródła

Bitonic sort:

```
1 #include <omp.h>
2 #include "bitonic_sort.h"
3
4 namespace bitonic_sort_seq_internal {
5
6 void bitonic_sort_seq(int *T, unsigned n, bool ascending);
7
8 void bitonic_merge_seq(int *T, unsigned n, bool ascending) {
9     if (n == 1)
10         return;
11
12     unsigned m = 1;
13     while ((m << 1) < n) {
14         m <<= 1;
15     }
16
17     if (ascending) {
18         for (unsigned i = 0; i < n - m; i++) {
```

```

19         if (T[i] > T[i + m]) {
20             int tmp = T[i];
21             T[i] = T[i + m];
22             T[i + m] = tmp;
23         }
24     }
25 } else {
26     for (unsigned i = 0; i < n - m; i++) {
27         if (T[i] < T[i + m]) {
28             int tmp = T[i];
29             T[i] = T[i + m];
30             T[i + m] = tmp;
31         }
32     }
33 }
34 // Recursive calls
35 bitonic_merge_seq(T, m, ascending);
36 bitonic_merge_seq(T + m, n - m, ascending);
37 }
38
39 void bitonic_sort_seq(int *T, unsigned n, bool ascending) {
40     if (n == 1)
41         return;
42
43     unsigned half_n = n >> 1;
44     // Recursive calls
45     bitonic_sort_seq(T, half_n, !ascending);
46     bitonic_sort_seq(T + half_n, n - half_n, ascending);
47
48     // Bitonic merge
49     bitonic_merge_seq(T, n, ascending);
50 }
51 }
52
53 /* Sequential bitonic sort */
54 void bitonic_sort_seq(int *T, unsigned n) {
55     bitonic_sort_seq_internal::bitonic_sort_seq(T, n, true);
56 }
57
58
59 namespace bitonic_sort_par_internal {
60
61 void bitonic_sort_seq(int *T, unsigned n, bool ascending);
62 void bitonic_sort(int *T, unsigned n, bool ascending, unsigned
    num_threads);
63
64 void bitonic_merge_seq(int *T, unsigned n, bool ascending) {
65     if (n == 1)
66         return;

```

```

67
68 unsigned m = 1;
69 while((m << 1) < n) {
70     m <<= 1;
71 }
72
73 if (ascending) {
74     for (unsigned i = 0; i < n - m; i++) {
75         if (T[i] > T[i + m]) {
76             int tmp = T[i];
77             T[i] = T[i + m];
78             T[i + m] = tmp;
79         }
80     }
81 } else {
82     for (unsigned i = 0; i < n - m; i++) {
83         if (T[i] < T[i + m]) {
84             int tmp = T[i];
85             T[i] = T[i + m];
86             T[i + m] = tmp;
87         }
88     }
89 }
90 // Recursive calls
91 bitonic_merge_seq(T, m, ascending);
92 bitonic_merge_seq(T + m, n - m, ascending);
93 }
94
95 void bitonic_merge(int *T, unsigned n, bool ascending, unsigned
num_threads) {
96     if (num_threads == 1) {
97         bitonic_merge_seq(T, n, ascending);
98         return;
99     }
100
101     if (n == 1)
102         return;
103
104     unsigned m = 1;
105     while((m << 1) < n) {
106         m <<= 1;
107     }
108
109     if (ascending) {
110         #pragma omp parallel for num_threads(num_threads)
111         for (unsigned i = 0; i < n - m; i++) {
112             if (T[i] > T[i + m]) {
113                 int tmp = T[i];
114                 T[i] = T[i + m];

```

```

115         T[i + m] = tmp;
116     }
117 }
118 } else {
119     #pragma omp parallel for num_threads(num_threads)
120     for (unsigned i = 0; i < n - m; i++) {
121         if (T[i] < T[i + m]) {
122             int tmp = T[i];
123             T[i] = T[i + m];
124             T[i + m] = tmp;
125         }
126     }
127 }
128 // Recursive calls
129 unsigned num_threads_first_half = (num_threads >> 1);
130 unsigned num_threads_second_half = num_threads -
    num_threads_first_half;
131 #pragma omp parallel sections
132 {
133     #pragma omp section
134     { bitonic_merge(T, m, ascending, num_threads_first_half); }
135     #pragma omp section
136     { bitonic_merge(T + m, n - m, ascending,
    num_threads_second_half); }
137 }
138 }
139
140 void bitonic_sort_seq(int *T, unsigned n, bool ascending) {
141     if (n == 1)
142         return;
143
144     unsigned half_n = n >> 1;
145     // Recursive calls
146     bitonic_sort_seq(T, half_n, !ascending);
147     bitonic_sort_seq(T + half_n, n - half_n, ascending);
148
149     // Bitonic merge
150     bitonic_merge_seq(T, n, ascending);
151 }
152
153 void bitonic_sort(int *T, unsigned n, bool ascending, unsigned
    num_threads) {
154     if (num_threads == 1) {
155         bitonic_sort_seq(T, n, ascending);
156         return;
157     }
158
159     if (n == 1)
160         return;

```

```

161
162 unsigned half_n = n >> 1;
163 // Recursive calls
164 unsigned num_threads_first_half = (num_threads >> 1);
165 unsigned num_threads_second_half = num_threads -
    num_threads_first_half;
166 #pragma omp parallel sections
167 {
168     #pragma omp section
169     { bitonic_sort(T, half_n, !ascending, num_threads_first_half
170 ); }
171     #pragma omp section
172     { bitonic_sort(T + half_n, n - half_n, ascending,
173 num_threads_second_half); }
174 }
175 // Bitonic merge
176 bitonic_merge(T, n, ascending, num_threads);
177 }
178
179 /* Parallel bitonic sort using OpenMP */
180 void bitonic_sort_par(int *T, unsigned n) {
181     if(n > 0) {
182         unsigned num_threads = omp_get_max_threads();
183         bitonic_sort_par_internal::bitonic_sort(T, n, true,
184 num_threads);
185     }
186 }

```

bitonic_sort.cpp

Merge sort:

```

1 #include "merge_sort.h"
2
3 #include <cstring>
4 #include <algorithm>
5 #include <omp.h>
6
7 namespace merge_sort_internal {
8
9 void merge(int *first, unsigned first_size, int *second,
10 unsigned second_size, int *dest) {
11     while (first_size > 0 && second_size > 0) {
12         if (*first < *second) {
13             *dest = *first;
14             first++;
15             first_size--;
16         } else {
17

```



```

16         *dest = *second;
17         second++;
18         second_size--;
19     }
20     dest++;
21 }
22 std::memcpy(dest, first, first_size*sizeof(int));
23 std::memcpy(dest, second, second_size*sizeof(int));
24 }
25
26 int find_element(int *T, unsigned n, int element) {
27     int low = 0, high = n;
28     while (low < high) {
29         int mid = (low + high) >> 1;
30         if (T[mid] < element)
31             low = mid + 1;
32         else
33             high = mid;
34     }
35     return low;
36 }
37
38 void parallel_merge(int *first, unsigned first_size, int *second,
39     , unsigned second_size, int *dest, unsigned threads) {
40     if (threads <= 1) {
41         merge(first, first_size, second, second_size, dest);
42         return;
43     }
44     // Make sure first is bigger
45     if (first_size < second_size) {
46         std::swap(first, second);
47         std::swap(first_size, second_size);
48     }
49     // Nothing left to be merged
50     if (first_size <= 0)
51         return;
52
53     unsigned half_first_size = first_size >> 1;
54     int mid = first[half_first_size];
55     unsigned half_second_size = find_element(second, second_size, mid);
56     dest[half_first_size + half_second_size] = mid;
57     unsigned half_threads = threads >> 1;
58
59     // Recursive calls
60     #pragma omp parallel sections
61     {
62

```

```

63         #pragma omp section
64         {
65             parallel_merge(first, half_first_size, second,
66                             half_second_size, dest, half_threads);
67         }
68         #pragma omp section
69         {
70             parallel_merge(first + half_first_size + 1,
71                             first_size - half_first_size - 1,
72                             second + half_second_size,
73                             second_size - half_second_size,
74                             dest + half_first_size +
75                             half_second_size + 1, threads - half_threads);
76         }
77     }
78 }
79
80 void merge_sort_seq(int *T, unsigned n, int *buffer) {
81     if (n <= 1)
82         return;
83     unsigned half_n = n >> 1;
84
85     merge_sort_seq(T, half_n, buffer);
86     merge_sort_seq(T + half_n, n - half_n, buffer + half_n);
87
88     merge(T, half_n, T + half_n, n - half_n, buffer);
89     memcpy(T, buffer, n * sizeof(int));
90 }
91
92 void merge_sort_par(int *T, unsigned n, int *buffer, unsigned
93 threads) {
94     if (n <= 1)
95         return;
96     if (threads <= 1) {
97         merge_sort_seq(T, n, buffer);
98         return;
99     }
100     unsigned half_n = n >> 1;
101     unsigned half_threads = threads >> 1;
102     // Recursive calls
103     #pragma omp parallel sections
104     {
105         #pragma omp section
106         {
107             merge_sort_par(T, half_n, buffer, half_threads);
108         }
109         #pragma omp section
110         {

```

```

106         merge_sort_par(T + half_n, n - half_n, buffer +
107         half_n, threads - half_threads);
108     }
109 }
110
111     parallel_merge(T, half_n, T + half_n, n - half_n, buffer,
112     threads);
113     memcpy(T, buffer, n * sizeof(int));
114 }
115
116 void merge_sort_par_merge_seq(int *T, unsigned n, int *buffer,
117 unsigned threads) {
118     if (n <= 1)
119         return;
120     if (threads <= 1) {
121         merge_sort_seq(T, n, buffer);
122         return;
123     }
124
125     unsigned half_n = n >> 1;
126     unsigned half_threads = threads >> 1;
127
128     // Recursive calls
129     #pragma omp parallel sections
130     {
131         #pragma omp section
132         {
133             merge_sort_par_merge_seq(T, half_n, buffer,
134             half_threads);
135         }
136         #pragma omp section
137         {
138             merge_sort_par_merge_seq(T + half_n, n - half_n,
139             buffer + half_n, threads - half_threads);
140         }
141     }
142
143     merge(T, half_n, T + half_n, n - half_n, buffer);
144     memcpy(T, buffer, n * sizeof(int));
145 }
146
147 } // namespace merge_sort_par::internal
148
149 void merge_sort_par(int *T, unsigned n) {
150     int *buffer = new int[n];
151     merge_sort_internal::merge_sort_par(T, n, buffer,
152     omp_get_max_threads());
153     delete [] buffer;
154 }

```

```

149
150 void merge_sort_seq(int *T, unsigned n) {
151     int *buffer = new int[n];
152     merge_sort_internal::merge_sort_seq(T, n, buffer);
153     delete [] buffer;
154 }
155
156 void merge_sort_par_merge_seq(int *T, unsigned n) {
157     int *buffer = new int[n];
158     merge_sort_internal::merge_sort_par_merge_seq(T, n, buffer,
159     omp_get_max_threads());
160     delete [] buffer;
161 }

```

merge_sort.cpp

Radix sort:

```

1 #include <omp.h>
2 #include <algorithm>
3 #include <cmath>
4 #include <cstring>
5 #include "radix_sort.h"
6
7 void radix_sort_par(int* T, unsigned n) {
8
9     // xor with sign bit
10    int sign_bit = 1 << 31;
11
12    #pragma omp parallel for firstprivate(sign_bit, T)
13    for (int i=0; i<n; i++) {
14        T[i] ^= sign_bit;
15    }
16
17    int nthreads = omp_get_max_threads();
18    int th_size = n/nthreads;
19    unsigned* last = new unsigned[nthreads];
20
21    unsigned* U = (unsigned*) T;
22    unsigned* U_res = new unsigned[n];
23    unsigned* pref_sum = new unsigned[n];
24    memset(pref_sum, 0, n*sizeof(unsigned));
25
26    for (int i=0; i<32; i++) {
27        unsigned ith_bit = 1 << i;
28        // prefix-sum for 0s
29
30        #pragma omp parallel num_threads(nthreads) firstprivate(
31        pref_sum, U, last, nthreads, ith_bit, th_size)
32        {

```

```

32         int thid = omp_get_thread_num();
33         int beg = thid * th_size;
34         int end = thid == nthreads-1 ? n: (thid + 1) *
th_size;
35         pref_sum[beg] = U[beg] & ith_bit ? 0 : 1;
36         for (int j=beg+1; j<end; j++) {
37             pref_sum[j] = pref_sum[j-1];
38             if (!(U[j] & ith_bit)) {
39                 ++pref_sum[j];
40             }
41         }
42         last[thid] = pref_sum[end-1];
43     }
44     for (int j=1; j<nthreads; j++) {
45         last[j] += last[j-1];
46     }
47     #pragma omp parallel num_threads(nthreads) firstprivate(
pref_sum, last, th_size)
48     {
49         int thid = omp_get_thread_num();
50         int beg = thid * th_size;
51         int end = thid == nthreads-1 ? n: (thid + 1) *
th_size;
52         int add = thid == 0 ? 0 : last[thid-1];
53         for (int j=beg; j<end; j++) {
54             pref_sum[j] += add;
55         }
56     }
57
58     int zeros = pref_sum[n-1];
59     // rewrite to U_res
60
61     #pragma omp parallel for firstprivate(ith_bit, zeros, U,
U_res, pref_sum)
62     for (int j=0; j<n; j++) {
63         if (U[j] & ith_bit) {
64             U_res[zeros + j - pref_sum[j]] = U[j];
65         } else {
66             U_res[pref_sum[j]-1] = U[j];
67         }
68     }
69
70     // swap U with U_res;
71     std::swap(U, U_res);
72 }
73
74 // xor back
75 #pragma omp parallel for firstprivate(sign_bit, T)
76 for (int i=0; i<n; i++) {

```

```

77         T[i] ^= sign_bit;
78     }
79
80     delete [] last;
81     delete [] U_res;
82     delete [] pref_sum;
83 }
84
85
86
87 void radix_sort_seq(int* T, unsigned n) {
88
89     // xor with sign bit
90     int sign_bit = 1 << 31;
91     for (int i=0; i<n; i++) {
92         T[i] ^= sign_bit;
93     }
94     unsigned* U = (unsigned*) T;
95     unsigned* U_res = new unsigned[n];
96     unsigned* pref_sum = new unsigned[n];
97     memset(pref_sum, 0, n*sizeof(unsigned));
98
99     for (int i=0; i<32; i++) {
100         unsigned ith_bit = 1 << i;
101         // prefix_sum for 0s
102         pref_sum[0] = U[0] & ith_bit ? 0 : 1;
103         for (int j=1; j<n; j++) {
104             pref_sum[j] = pref_sum[j-1];
105             if (!(U[j] & ith_bit)) {
106                 ++pref_sum[j];
107             }
108         }
109         int zeros = pref_sum[n-1];
110         // rewrite to U_res
111         for (int j=0; j<n; j++) {
112             if (U[j] & ith_bit) {
113                 U_res[zeros + j - pref_sum[j]] = U[j];
114             } else {
115                 U_res[pref_sum[j]-1] = U[j];
116             }
117         }
118         // swap U with U_res;
119         std::swap(U, U_res);
120     }
121     // xor back
122     for (int i=0; i<n; i++) {
123         T[i] ^= sign_bit;
124     }
125     delete [] U_res;

```

```
126 |     delete [] pref_sum ;  
127 | }
```

radix_sort.cpp