Sortowania równoległe i sekwencyjne

Dorota Kapturkiewicz
Wiktor Kuropatwa
Karol Różycki

1 Cel projektu

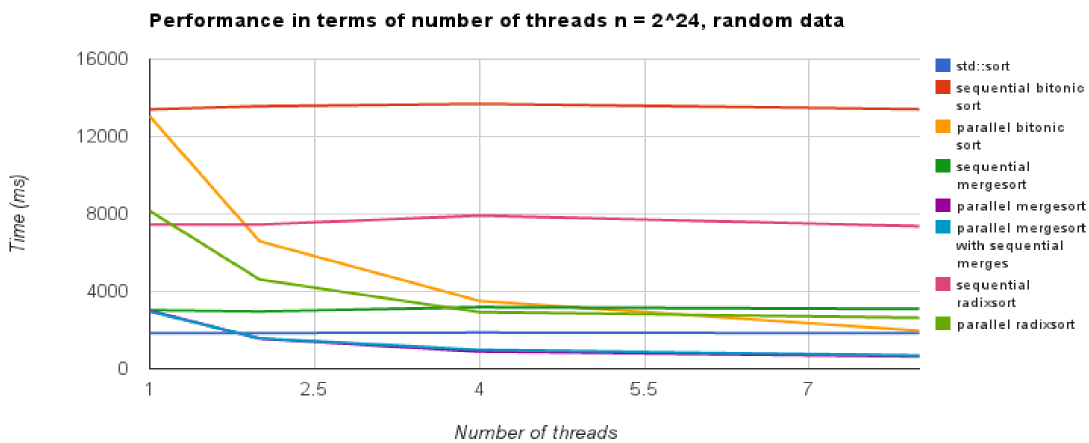
Celem pniższego projektu jest porównanie efektywności i czasu działania kilku algorytmów sortowania z różnym stopniem zrównoleglenia.

2 Zaimplementowane algorytmy

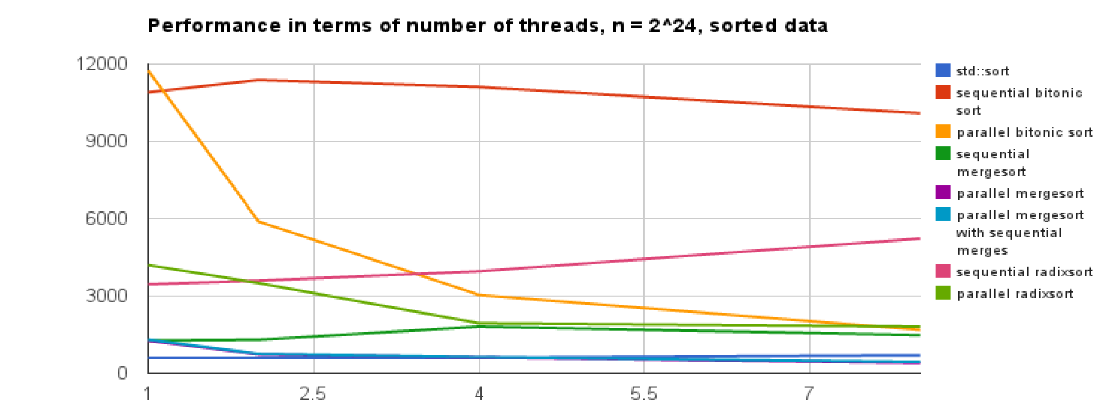
Projekt zaostał zaimplementowany w języku C++ (w standardzie c++ 11).
Do fragmentów wielowątkowych użyliśmy technologii OpenMP.
Dana praca porównuje działania następujących algorytmów:

- 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)

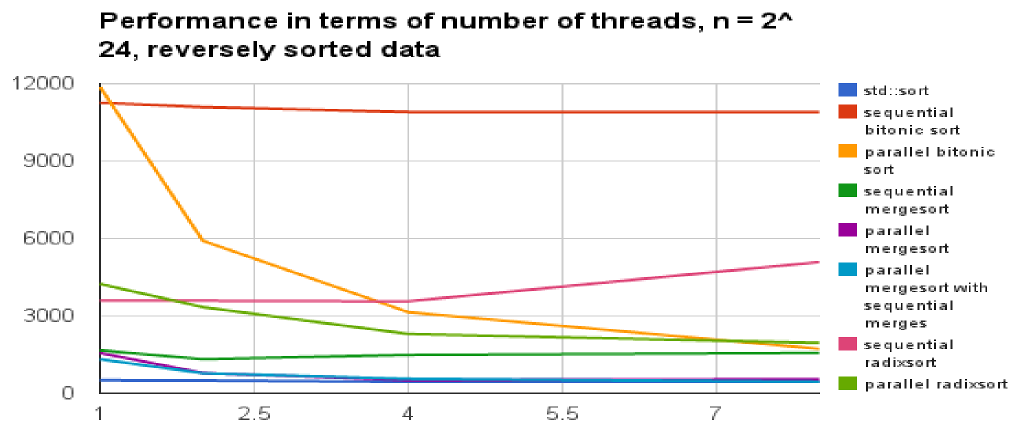
3 Przeprowadzone testy



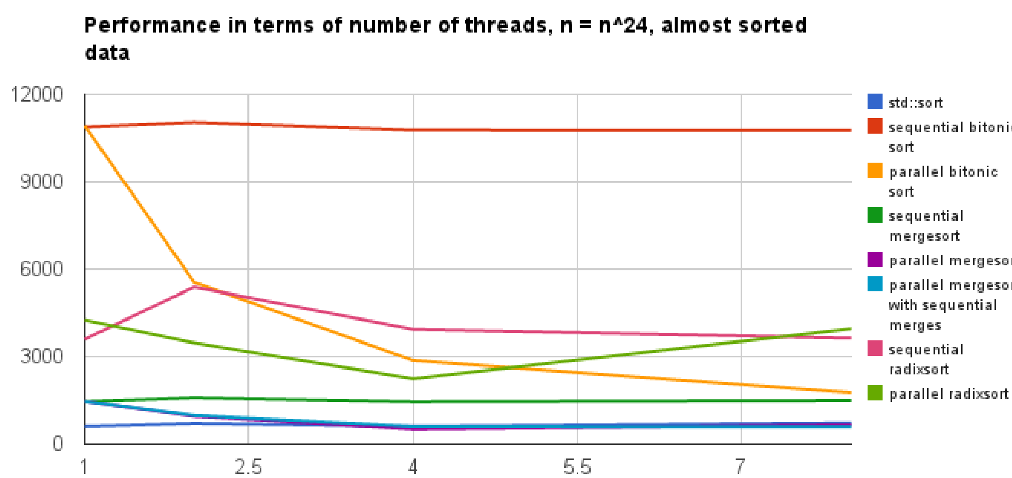
Rysunek 1: Czas działania w zależności od liczby wątków na losowych danych



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

5 Źródła

Bitonic sort:

```
#include <omp.h>
#include "bitonic_sort.h"

namespace bitonic_sort_seq_internal {

void bitonic_sort_seq(int *T, unsigned n, bool ascending);

void bitonic_merge_seq(int *T, unsigned n, bool ascending) {
  if (n == 1)
    return;

  unsigned m = 1;
  while((m << 1) < n) {
    m <<= 1;
  }

  if (ascending) {
    for (unsigned i = 0; i < n - m; i++) {
      if (T[i] > T[i + m]) {
        int tmp = T[i];
        T[i] = T[i + m];
        T[i + m] = tmp;
      }
    }
  } else {
    for (unsigned i = 0; i < n - m; i++) {
      if (T[i] < T[i + m]) {
        int tmp = T[i];
        T[i] = T[i + m];
        T[i + m] = tmp;
      }
    }
  }
  // Recursive calls
  bitonic_merge_seq(T, m, ascending);
  bitonic_merge_seq(T + m, n - m, ascending);
}

void bitonic_sort_seq(int *T, unsigned n, bool ascending) {
  if (n == 1)
    return;

```

```
  unsigned half_n = n >> 1;
  // Recursive calls
  bitonic_sort_seq(T, half_n, !ascending);
  bitonic_sort_seq(T + half_n, n - half_n, ascending);

  // Bitonic merge
  bitonic_merge_seq(T, n, ascending);
}
}

/* Sequential bitonic sort */
void bitonic_sort_seq(int *T, unsigned n) {
  bitonic_sort_seq_internal::bitonic_sort_seq(T, n, true);
}


namespace bitonic_sort_par_internal {

void bitonic_sort_seq(int *T, unsigned n, bool ascending);
void bitonic_sort(int *T, unsigned n, bool ascending, unsigned
    num_threads);

void bitonic_merge_seq(int *T, unsigned n, bool ascending) {
  if (n == 1)
    return;

  unsigned m = 1;
  while((m << 1) < n) {
    m <<= 1;
  }

  if (ascending) {
    for (unsigned i = 0; i < n - m; i++) {
      if (T[i] > T[i + m]) {
        int tmp = T[i];
        T[i] = T[i + m];
        T[i + m] = tmp;
      }
    }
  } else {
    for (unsigned i = 0; i < n - m; i++) {
      if (T[i] < T[i + m]) {
        int tmp = T[i];
        T[i] = T[i + m];
        T[i + m] = tmp;
      }
    }
  }
  // Recursive calls
```

```
  bitonic_merge_seq(T, m, ascending);
  bitonic_merge_seq(T + m, n - m, ascending);
}

void bitonic_merge(int *T, unsigned n, bool ascending, unsigned num_threads) {
  if (num_threads == 1) {
    bitonic_merge_seq(T, n, ascending);
    return;
  }

  if (n == 1)
    return;

  unsigned m = 1;
  while((m << 1) < n) {
    m <<= 1;
  }

  if (ascending) {
    #pragma omp parallel for num_threads(num_threads)
    for (unsigned i = 0; i < n - m; i++) {
      if (T[i] > T[i + m]) {
        int tmp = T[i];
        T[i] = T[i + m];
        T[i + m] = tmp;
      }
    }
  } else {
    #pragma omp parallel for num_threads(num_threads)
    for (unsigned i = 0; i < n - m; i++) {
      if (T[i] < T[i + m]) {
        int tmp = T[i];
        T[i] = T[i + m];
        T[i + m] = tmp;
      }
    }
  }
  // Recursive calls
  unsigned num_threads_first_half = (num_threads >> 1);
  unsigned num_threads_second_half = num_threads - num_threads_first_half;
  #pragma omp parallel sections
  {
    #pragma omp section
    { bitonic_merge(T, m, ascending, num_threads_first_half); }
    #pragma omp section
    { bitonic_merge(T + m, n - m, ascending, num_threads_second_half); }
```

```
  }
}

void bitonic_sort_seq(int *T, unsigned n, bool ascending) {
  if (n == 1)
    return;

  unsigned half_n = n >> 1;
  // Recursive calls
  bitonic_sort_seq(T, half_n, !ascending);
  bitonic_sort_seq(T + half_n, n - half_n, ascending);

  // Bitonic merge
  bitonic_merge_seq(T, n, ascending);
}

void bitonic_sort(int *T, unsigned n, bool ascending, unsigned num_threads) {
  if (num_threads == 1) {
    bitonic_sort_seq(T, n, ascending);
    return;
  }

  if (n == 1)
    return;

  unsigned half_n = n >> 1;
  // Recursive calls
  unsigned num_threads_first_half = (num_threads >> 1);
  unsigned num_threads_second_half = num_threads - num_threads_first_half;
  #pragma omp parallel sections
  {
    #pragma omp section
    { bitonic_sort(T, half_n, !ascending, num_threads_first_half); }
    #pragma omp section
    { bitonic_sort(T + half_n, n - half_n, ascending, num_threads_second_half); }
  }

  // Bitonic merge
  bitonic_merge(T, n, ascending, num_threads);
}
}

/* Parallel bitonic sort using OpenMP */
void bitonic_sort_par(int *T, unsigned n) {
  if(n > 0) {
```

```
    unsigned num_threads = omp_get_max_threads();
    bitonic_sort_par_internal::bitonic_sort(T, n, true,
   num_threads);
  }
}
```

bitonic_sort.cpp

Merge sort:

```
#include "merge_sort.h"

#include <cstring>
#include <algorithm>
#include <omp.h>

namespace merge_sort_internal {

void merge(int *first, unsigned first_size, int *second,
    unsigned second_size, int *dest) {
    while (first_size > 0 && second_size > 0) {
        if (*first < *second) {
            *dest = *first;
            first++;
            first_size--;
        } else {
            *dest = *second;
            second++;
            second_size--;
        }
        dest++;
    }
    std::memcpy(dest, first, first_size*sizeof(int));
    std::memcpy(dest, second, second_size*sizeof(int));
}

int find_element(int *T, unsigned n, int element) {
    int low = 0, high = n;
    while (low < high) {
        int mid = (low + high) >> 1;
        if (T[mid] < element)
            low = mid + 1;
        else
            high = mid;
    }
    return low;
}

void parallel_merge(int *first, unsigned first_size, int *second
    , unsigned second_size, int *dest, unsigned threads) {
```

```
    if (threads <= 1) {
        merge(first, first_size, second, second_size, dest);
        return;
    }
    // Make sure first is bigger
    if (first_size < second_size) {
        std::swap(first, second);
        std::swap(first_size, second_size);
    }

    // Nothing left to be merged
    if (first_size <= 0)
        return;

    unsigned half_first_size = first_size >> 1;
    int mid = first[half_first_size];
    unsigned half_second_size = find_element(second, second_size, mid);
    dest[half_first_size + half_second_size] = mid;
    unsigned half_threads = threads >> 1;


    // Recursive calls
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            parallel_merge(first, half_first_size, second, half_second_size, dest, half_threads);
        }
        #pragma omp section
        {
            parallel_merge(first + half_first_size + 1, first_size - half_first_size - 1,
                           second + half_second_size, second_size - half_second_size,
                           dest + half_first_size + half_second_size + 1, threads - half_threads);
        }
    }
}

void merge_sort_seq(int *T, unsigned n, int *buffer) {
    if (n <= 1)
        return;
    unsigned half_n = n >> 1;

    merge_sort_seq(T, half_n, buffer);
    merge_sort_seq(T + half_n, n - half_n, buffer + half_n);
```

```c

    merge(T, half_n, T + half_n, n - half_n, buffer);
    memcpy(T, buffer, n * sizeof(int));
}

void merge_sort_par(int *T, unsigned n, int *buffer, unsigned
    threads) {
    if (n <= 1)
        return;
    if (threads <= 1) {
        merge_sort_seq(T, n, buffer);
        return;
    }
    unsigned half_n = n >> 1;
    unsigned half_threads = threads >> 1;
    // Recursive calls
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            merge_sort_par(T, half_n, buffer, half_threads);
        }
        #pragma omp section
        {
            merge_sort_par(T + half_n, n - half_n, buffer +
    half_n, threads - half_threads);
        }
    }

    parallel_merge(T, half_n, T + half_n, n - half_n, buffer,
    threads);
    memcpy(T, buffer, n * sizeof(int));
}

void merge_sort_par_merge_seq(int *T, unsigned n, int *buffer,
    unsigned threads) {
    if (n <= 1)
        return;
    if (threads <= 1) {
        merge_sort_seq(T, n, buffer);
        return;
    }

    unsigned half_n = n >> 1;
    unsigned half_threads = threads >> 1;

    // Recursive calls
    #pragma omp parallel sections
    {
```

```
        #pragma omp section
        {
            merge_sort_par_merge_seq(T, half_n, buffer,
    half_threads);
        }
        #pragma omp section
        {
            merge_sort_par_merge_seq(T + half_n, n - half_n,
    buffer + half_n, threads - half_threads);
        }
    }

    merge(T, half_n, T + half_n, n - half_n, buffer);
    memcpy(T, buffer, n * sizeof(int));
}

} // namespace merge_sort_par::internal

void merge_sort_par(int *T, unsigned n) {
    int *buffer = new int[n];
    merge_sort_internal::merge_sort_par(T, n, buffer,
    omp_get_max_threads());
    delete[] buffer;
}

void merge_sort_seq(int *T, unsigned n) {
    int *buffer = new int[n];
    merge_sort_internal::merge_sort_seq(T, n, buffer);
    delete[] buffer;
}

void merge_sort_par_merge_seq(int *T, unsigned n) {
    int *buffer = new int[n];
    merge_sort_internal::merge_sort_par_merge_seq(T, n, buffer,
    omp_get_max_threads());
    delete[] buffer;
}
```

merge_sort.cpp

Radix sort:

```
#include <omp.h>
#include <algorithm>
#include <cmath>
#include <cstring>
#include "radix_sort.h"

void radix_sort_par(int* T, unsigned n) {

```

```
    // xor with sign bit
    int sign_bit = 1 << 31;

    #pragma omp parallel for firstprivate(sign_bit, T)
    for (int i=0; i<n; i++) {
        T[i] ^= sign_bit;
    }

    int nthreads = omp_get_max_threads();
    int th_size = n/nthreads;
    unsigned* last = new unsigned[nthreads];

    unsigned* U = (unsigned*) T;
    unsigned* U_res = new unsigned[n];
    unsigned* pref_sum = new unsigned[n];
    memset(pref_sum, 0, n*sizeof(unsigned));

    for (int i=0; i<32; i++) {
        unsigned ith_bit = 1 << i;
        // prefix_sum for 0s

        #pragma omp parallel num_threads(nthreads) firstprivate(pref_sum, U, last, nthreads, ith_bit, th_size)
        {
            int thid = omp_get_thread_num();
            int beg = thid * th_size;
            int end = thid == nthreads-1 ? n: (thid + 1) * th_size;
            pref_sum[beg] = U[beg] & ith_bit ? 0 : 1;
            for (int j=beg+1; j<end; j++) {
                pref_sum[j] = pref_sum[j-1];
                if (!(U[j] & ith_bit)) {
                    ++pref_sum[j];
                }
            }
            last[thid] = pref_sum[end-1];
        }
        for (int j=1; j<nthreads; j++) {
            last[j] += last[j-1];
        }
        #pragma omp parallel num_threads(nthreads) firstprivate(pref_sum, last, th_size)
        {
            int thid = omp_get_thread_num();
            int beg = thid * th_size;
            int end = thid == nthreads-1 ? n: (thid + 1) * th_size;
            int add = thid == 0 ? 0 : last[thid-1];
            for (int j=beg; j<end; j++) {
```

```
                pref_sum[j] += add;
            }
        }

        int zeros = pref_sum[n-1];
        // rewrite to U_res

        #pragma omp parallel for firstprivate(ith_bit, zeros, U, U_res, pref_sum)
        for (int j=0; j<n; j++) {
            if (U[j] & ith_bit) {
                U_res[zeros + j - pref_sum[j]] = U[j];
            } else {
                U_res[pref_sum[j]-1] = U[j];
            }
        }

        // swap U with U_res;
        std::swap(U, U_res);
    }

    // xor back
    #pragma omp parallel for firstprivate(sign_bit, T)
    for (int i=0; i<n; i++) {
        T[i] ^= sign_bit;
    }

    delete[] last;
    delete[] U_res;
    delete[] pref_sum;
}



void radix_sort_seq(int* T, unsigned n) {

    // xor with sign bit
    int sign_bit = 1 << 31;
    for (int i=0; i<n; i++) {
        T[i] ^= sign_bit;
    }
    unsigned* U = (unsigned*) T;
    unsigned* U_res = new unsigned[n];
    unsigned* pref_sum = new unsigned[n];
    memset(pref_sum, 0, n*sizeof(unsigned));

    for (int i=0; i<32; i++) {
        unsigned ith_bit = 1 << i;
        // prefix_sum for 0s
```

```cpp
        pref_sum[0] = U[0] & ith_bit ? 0 : 1;
        for (int j=1; j<n; j++) {
            pref_sum[j] = pref_sum[j-1];
            if (!(U[j] & ith_bit)) {
                ++pref_sum[j];
            }
        }
        int zeros = pref_sum[n-1];
        // rewrite to U_res
        for (int j=0; j<n; j++) {
            if (U[j] & ith_bit) {
                U_res[zeros + j - pref_sum[j]] = U[j];
            } else {
                U_res[pref_sum[j]-1] = U[j];
            }
        }
        // swap U with U_res;
        std::swap(U, U_res);
    }
    // xor back
    for (int i=0; i<n; i++) {
        T[i] ^= sign_bit;
    }
    delete[] U_res;
    delete[] pref_sum;
}
```

radix_sort.cpp