

Sortowanie bitoniczne

Opis algorytmu

Sortowanie bitoniczne nie było żadnym z zadań na kursach ASD. Urzekło mnie ono swoim wzorem wywołań, naprzemiennie obserwujemy wywołania `bitSortUp` i `bitSortDown`. Dodatkowo należy do klasy algorytmów *niepomnych* - każde wywołanie dotyka tych samych komórek pamięci. Łącząc *niepomność* z brakiem konfliktów odczytu i zapisu otrzymujemy, wydaje się, prawie idealny algorytm sortujący do zrównoleglenia.

Spójrzmy na sam algorytm:

```
void bit_merge(
    long long* ar,
    size_t left,
    size_t right,
    size_t n,
    int dir)
{
    if (n == 1) return;
    size_t mid = (left + right) / 2;
    size_t half = n / 2;
    for (size_t i = 0; i < half; ++i)
        bitonic_swap(
            ar,
            left + i,
            mid + i,
            dir);
    bit_merge(ar, left, mid, n/2, dir);
    bit_merge(ar, mid, right, n/2, dir);
}

void bit_sort(
    long long* ar,
    size_t left,
    size_t right,
    size_t n,
    int dir)
{
    if (n == 1) return;
    size_t mid = (left + right) / 2;
    bit_sort(ar, left, mid, n/2, ASC);
    bit_sort(ar, mid, right, n/2, DESC);
    bit_merge(ar, left, right, n, dir);
}
```

Sortowanie bitoniczne ma jedną wadę, długość tablicy wejściowej musi być postaci 2^k . Łatwo można sobie z tym poradzić dopełniając tablicę do najbliższej potęgi 2 wartością ∞ (na przykład `LONG LONG MAX`), a potem ucinając nadmiarowy koniec posortowanej zmodyfikowanej tablicy.

Złożoności

Klasyczny algorytm w wersji 1 wątkowej ma złożoność czasową: $\mathcal{O}(n \ln^2(n))$, co otrzymujemy wprost z *Twierdzenia o rekursji uniwersalnej*.

Złożoność pamięciowa: $\mathcal{O}(\ln^2(n))$ przez rekursję, samo sortowanie następuje *w miejscu*.

Model PRAM

Jeśli zrównoleglimy pętlę `for` i wszystkie podwywołania w rekursji to otrzymamy:

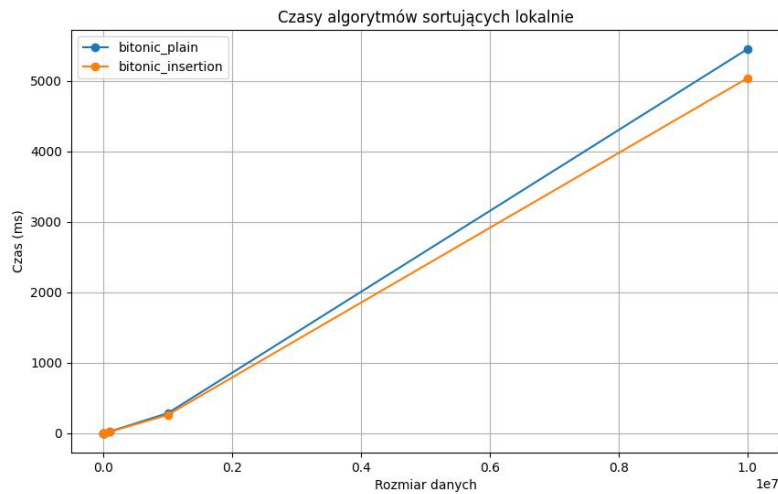
$$T(n) = \mathcal{O}(\ln^2(n))$$

$$W(n) = \mathcal{O}(n \ln^2(n))$$

Brak konfliktów odczytu i zapisu, więc jesteśmy w klasie **EREW**.

Optymalizacje

Zawsze dobrym pomysłem jest uciąć rekursję w porę, więc od pewnego rozmiaru podzadania zamiast zagłębiać się rekurencyjnie wywołuję `insertionSort`.



Rysunek 1: klasyczny bitonicSort vs bitonicSortWithInsertion

Nie poprawiło to wydajności, tak jakbym się spodziewał. Lepsze wyniki osiągnelibyśmy zastępując `bitMerge` skalaniem podobnym jak w `mergeSort`, ale wtedy algorytm wykorzystywałby dodatkową pamięć, czego nie chciałem.

OpenMP

Bibliotekę **OpenMP** wykorzystałem do zrównoleglenia rekurencyjnych podzadań **bitSort**. Utworzyłem pojedynczą grupę wątków nad korzeniem rekursji i pozwoliłem bibliotece zarządzać wątkami w podzadaniach. Przed **bitMerge** zsynchronizowałem wątki. Próba zrównoleglenia również pętli **for** w **bitMerge** zakończyła się niepowodzeniem, wątki zaczynały walczyć o zasoby i czas wykonania się diametralnie wydłużał.

```
...                               ...
#pragma omp parallel              #pragma omp task
{                                  bit_sort_omp(ar, left, mid, n/2, ASC);
    #pragma omp single            #pragma omp task
    {                             bit_sort_omp(ar, mid, right, n/2, DESC);
        bit_sort_omp(...);        #pragma omp taskwait
    }                             bit_merge_omp(ar, left, right, n, dir);
}                                  ...
...
```

Wątki systemowe

Tutaj zasada zrównoleglenia jest bardzo podobna. Ustaliłem granicę, powyżej której funkcja będzie wielowątkowa. Poniżej tego poziomu nakład zarządzania wątkami może przekroczyć czas działania wersji 1 wątkowej. Załóżmy, że dane są odpowiednio duże.

1. Najpierw ustalona przeze mnie liczba wątków podzieli się tablicą z danymi. Każdy wątek będzie odpowiedzialny za swój blok.
2. Wątki sortują bloki wywołując **bitSort** na bloku.
3. Synchronizacja.
4. Dwukrotnie maleje liczba wątków, dwukrotnie zwiększa się rozmiar bloku. Jesteśmy teraz w momencie, kiedy klasyczny **bitSort** wchodzi do góry od liści wykonując tylko **bitMerge**.
5. Wątki scalają bloki wywołując **bitMerge**.
6. Synchronizacja.
7. Zmniejszenie liczby wątków, zwiększenie rozmiarów bloków.
8. ... aż rozmiar bloku, będzie równy rozmiarowi tablicy wejściowej.

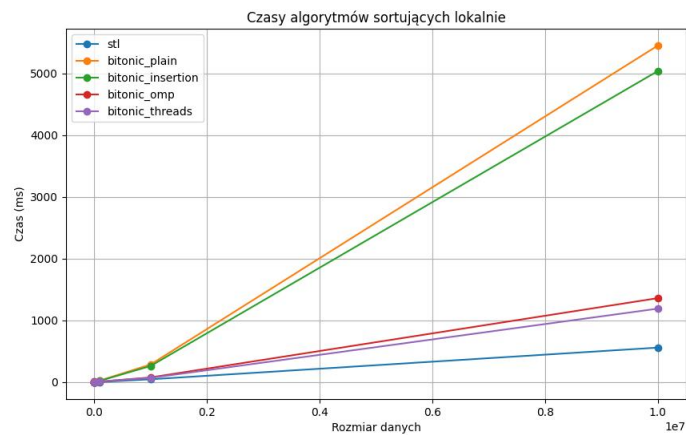
Z moich obliczeń wynika, że:

$$T(n) = \mathcal{O}(n \ln(n))$$

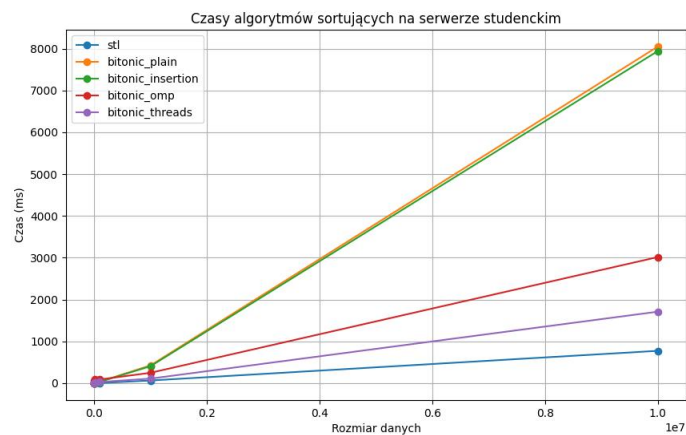
$$W(n) = \mathcal{O}(n \ln^2(n)) \text{ bez zmian}$$

Pomiary czasowe

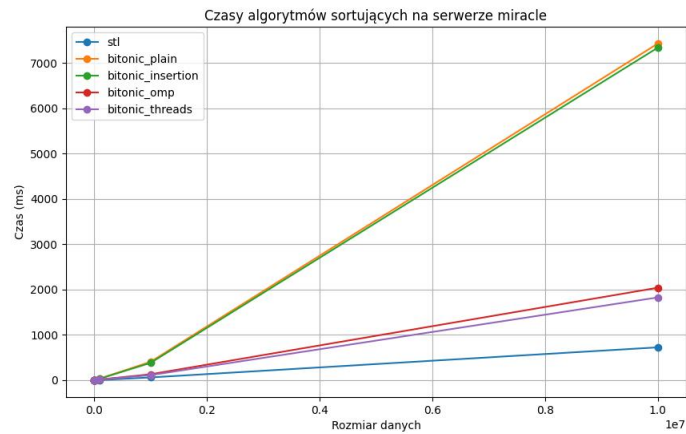
Lokalne pomiary przeprowadziłem na procesorze z 16 rdzeniami. Raz **scheduler** tak ułożył wątki, że moje sortowanie bitoniczne na wątkach systemowych wyprzedziło **sort** z STL o *300ms*. To się już nigdy nie powtórzyło.



Rysunek 2: Pomiary lokalne



Rysunek 3: Pomiary na serwerze student



Rysunek 4: Pomiary na serwerze miracle

Jak widać na powyższych wykresach:

1. Nie udało mi się pokonać **sorta** z **STL**.
2. **OpenMP** działa zauważalnie gorzej na studencie.
3. Sukcesem jest, że lepiej zarządzam wątkami niż robi to biblioteka **OpenMP**

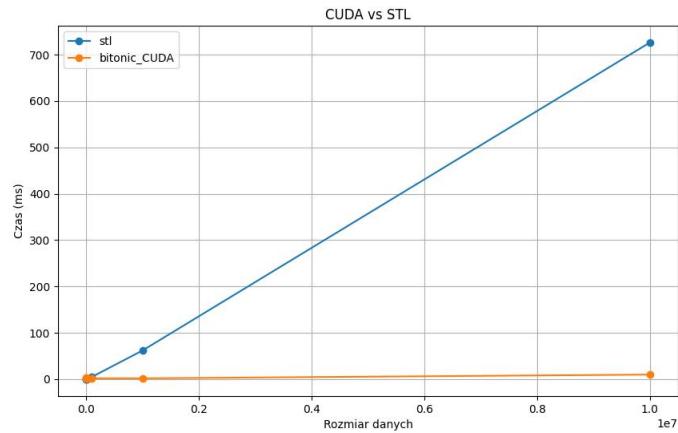
Podsumowanie

Mojej implementacji jest daleko od idealnej sieci sortującej. Wyniki pokazują również, jak bardzo dopracowany jest **sort** z **STL**. Być może na **CUDA** udałoby się pokonać **sort** z **STL**.

CUDA (aktualizacja po czasie)

W końcu udało mi się zebrać do implementacji sortowania bitonicznego na CUDA i wyniki są bardzo zadowalające. `sort` z STL został pokonany ;).

Testowane na karcie NVIDIA RTX A4000.



Rysunek 5: STL vs CUDA

Kilka słów o implementacji:

1. Całość opiera się o 2 pętle `for`. Jedna przebiega po rozmiarze podzadania w sortowaniu bitonicznym, a druga udaje mergowanie.
2. Równoległość osiągnięta z pomocą biblioteki `moderngpu` (powodem jest tylko i wyłącznie moja wygoda, żeby samemu nie pisać kerneli).