SPRAWOZDANIE METODY PROGRAMOWANIA RÓWNOLEGŁEGO

OpenMP

Autorzy
Bartosz Szlachta
Robert Kubok

Spis treści

1 Opis implementowanego algorytmu i analiza rzedu złożoności.

1.1 Opis algorytmu.

Pierwszy algorytm implementacji równoległego sortowania kubełkowego składał sie z 4 warunków:

1. Każdy watek czyta cała tablice poczatkowa (różne indexy poczatkowe).
2. Każdy watek posiada własne kubełki z ustalonym zakresem liczb.
3. Każdy watek zapisuje swoje kubełki i sortuje.
4. Każdy watek wpisuje posortowane kubełki do odpowiedniego miejsca w tablicy.

Podczas implementacji tego rodzaju sortowania kubelkowego należało odpowiedzieć sobie na pytanie czy potrzebna jest ochrona danych wspólnych:

- Tablica poczatkowa przy odczycie: nie, wystarczy, że w tym samym czasie dokładnie jeden watek bedzie odczytywał z danego indeksu tablicy.
- Tablica poczatkowa przy zapisie: nie, jeśli każdy watek wpisuje do innego indeksu.
- Kubełki: w tym przypadku nie, każdy watek ma swoje osobne kubełki.

1.2 Analiza rzedu złożoności.

Dane:

n - wielkość tablicy wejściowej

k - ilość kubełków

Złożoność składa sie z:

1. Przejścia po tablicy poczatkowej i włożenie elementów do kubełków = $O(n)$.
2. Posortowania każdego kubełka = $k * \frac{n}{k} * log(\frac{n}{k})$.
3. Przepisania kubełków do tablicy wynikowej = $O(n)$.

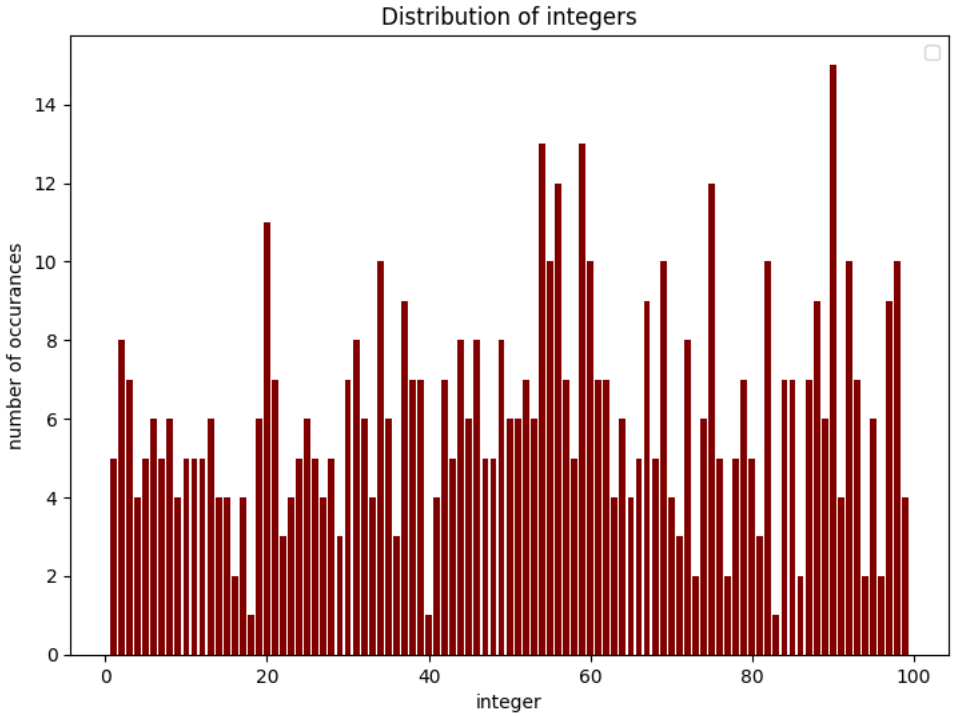
Ostatecznie uzyskujemy złożoność oblczeniowa = $O(n + k * \frac{n}{k} * log(\frac{n}{k})$.

W ramach algorytmu równoległego przyspieszyć powinna cześć sortowania każdego kubełka, oraz przepisywania kubełków do tablicy wynikowej.

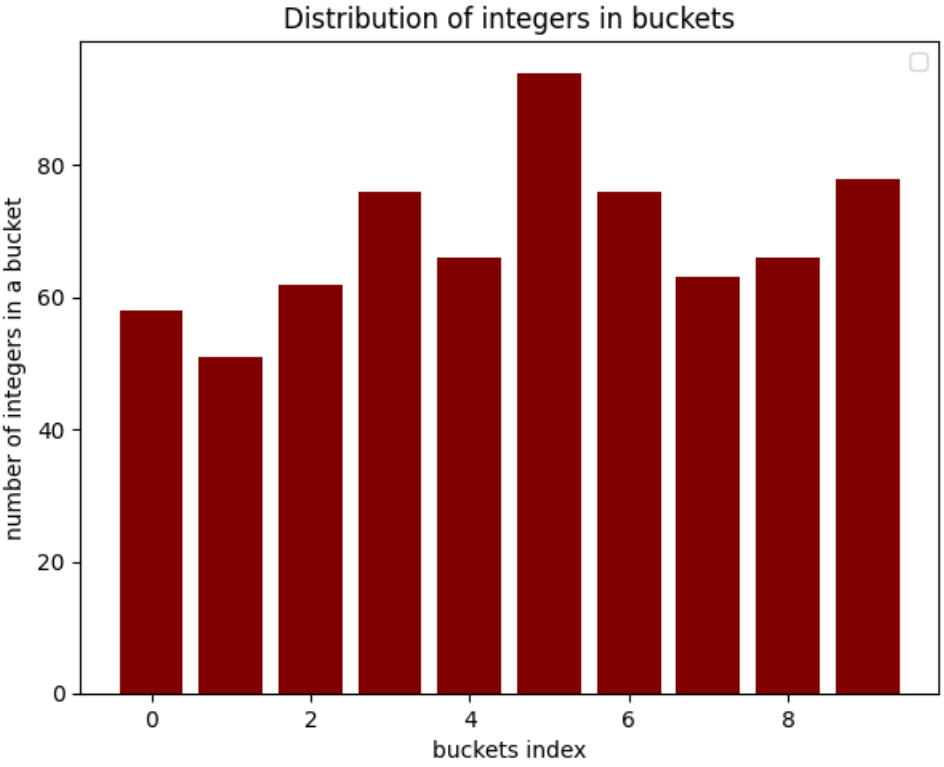
2 Losowanie liczb do tablicy.

2.1 Generator liczb.

Na poczatek sprawdziłem jak zachowuje sie użyty generator liczb losowych poprzez wygenerowanie 700 liczb losowych w zakresie od 1 do 99 włacznie. Dzieki temu mogłem sprawdzić jak wyglada rozkład generowanych liczb oraz jak wyglada rozkład liczb do kubełków. Eksperyment został przeprowadzony na lokalnym komputerze. Wyniki przedstawione na Rysunku 1 oraz Rysunku 2.



Rysunek 1: Wykres rozładu liczb w tablicy.



Rysunek 2: Wykres rozładu kubełków.

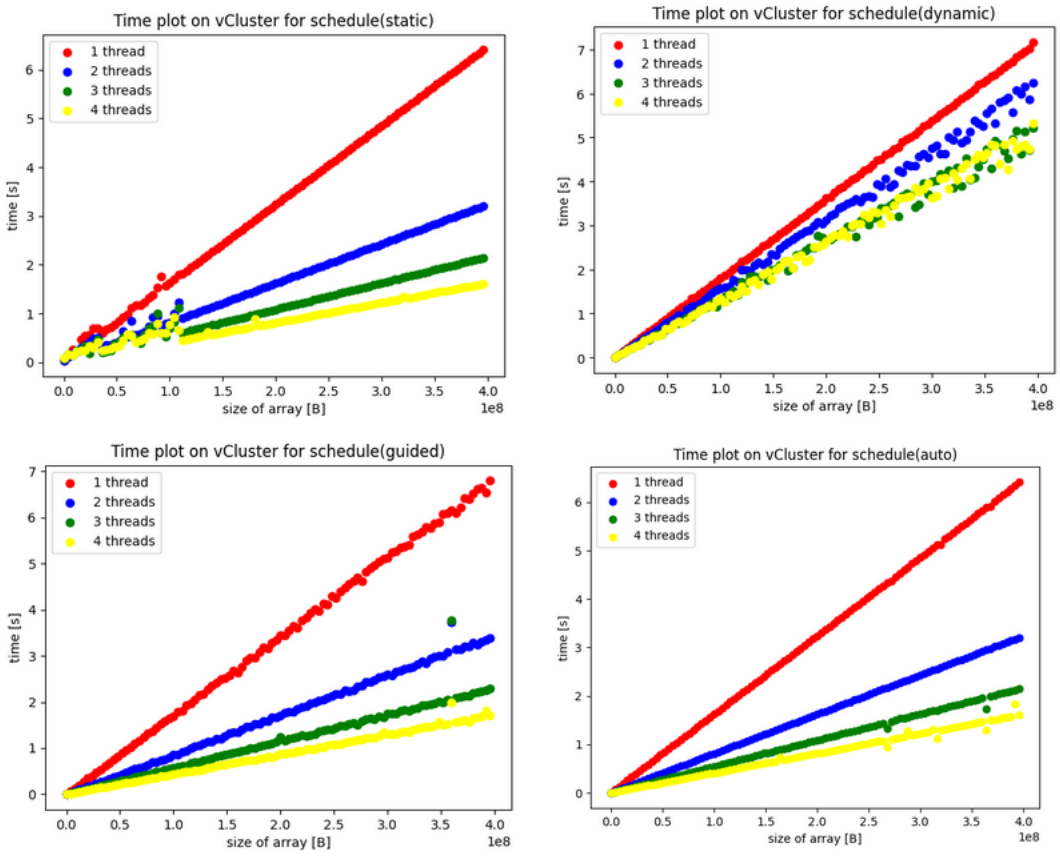
2.2 Wpisywanie wygenerowanych liczb do tablicy

Liczb zostawały wpisane równolegle za pomoca **omp for schedule**. Eksperyment został przeprowadzony na środowisku VCluster. OpenMP pozwala użyć różnego rodzaju planowania:

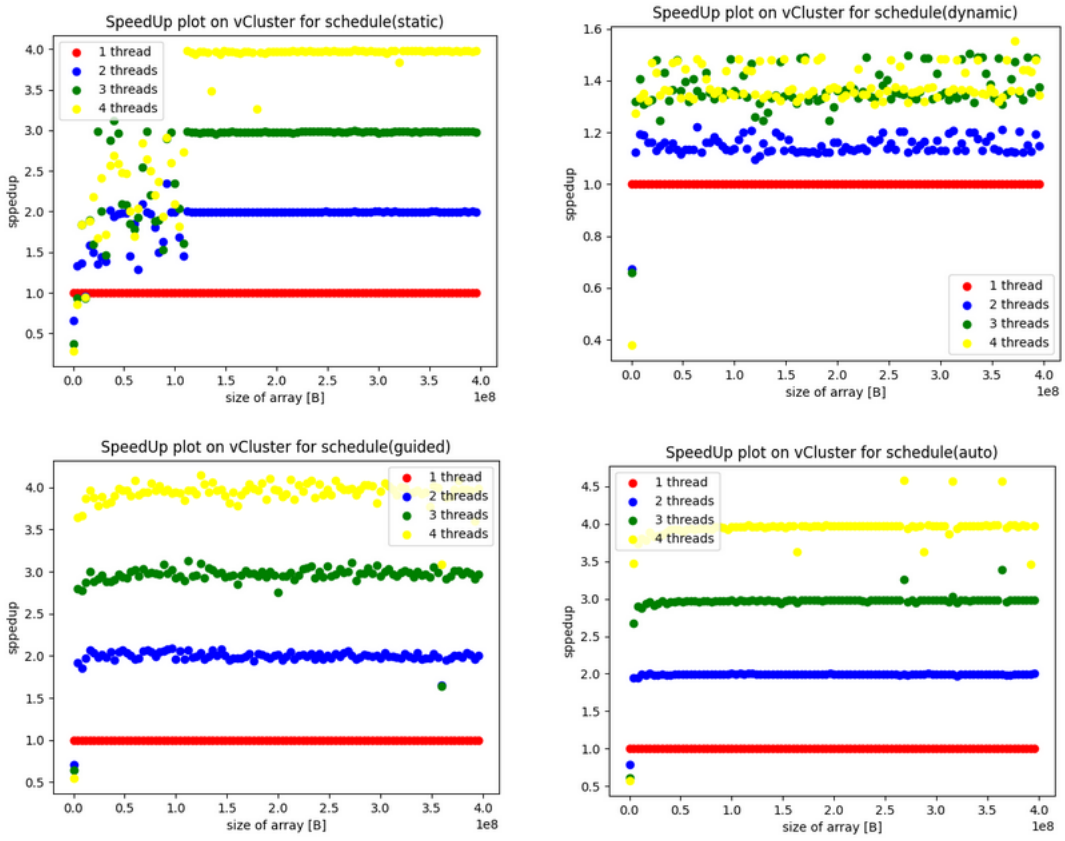
- Static schedule - OpenMP dzieli iteracje na rozmiary równe chunk-size i rozdaje je do watków w okrężnym porzadku. Jeśli chunk-size nie jest podany, OpenMP dzieli iteracje w około równe porcje i dzieli conajwyżej jedna porcje na jeden watek.
- Dynamic schedule - tym razem nie ma tu kolejności, watki biora porcje o podanym rozmiarze i przetwarzaja je do momentu aż nie ma już nic do przetwarzania, a nastepnie requestuja nastepny chunk. Jeśli chunk-size nie jest podany, OpenMP przypisuje mu wartość = 1. Dobre podejście gdy iteracje wymagaja różnego nakładu pracy (sa źle zrównoważone). Mniejsza szansa na to, że jakiś watek bedzie ciagle do tyłu w porównaniu z reszta.
- Guided schedule - podobny do dynamic, jednak każda nastepna iteracja jest mniejsza, aż do momentu chunk-size (ostatnia iteracja może być mniejsza). Rozmiar aktualnego chunka jest proporcjonalny do ilości nieprzypisasnych iteracji podzielonych przez ilość watków. Jeśli chunk-size nie jest podany, OpenMP przypisuje mu wartość = 1. Dobre, gdy zle zrównoważenie wystepuje na końcu.
- Auto schedule - deleguje prace do kompilatora i/lub systemu.

2.3 Test rodzajów schedule.

Testy zostały przeprowadzone dla ilości watków od 1 do 4, oraz dla domyślnej wielkości chunków. Wyniki zostały przedstawione na Rysunku 3 i Rysunku 4.



Rysunek 3: Wykresy czasu różnego rodzaju szeregowania watków.



Rysunek 4: Wykresy przyspieszenia różnego rodzaju szeregowania watków.

2.4 Wnioski z eksperymentu z domyślnym rodzajem schedule.

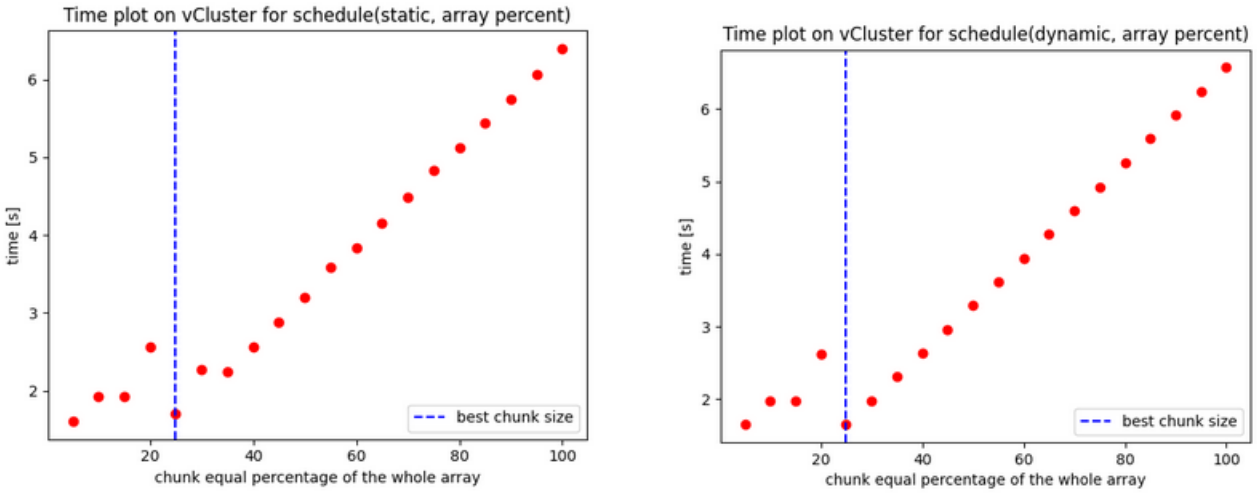
Wykresy dla static, guided i auto sa zbliżone, dzieje sie to dlatego, że:

- Auto deleguje prace do kompilatora i/lub systemu i najprawdopodobniej zostaje tam ustawione static.
- Guided na poczatku działania ma duży rozmiar chunków, wiec problem z ciagłbym proszeniem o nowe porcje jest mały.

Widać jednak, że schedule(dynamic) zachowuje sie inaczej, jest tak dlatego, że mamy ustawiony chunk = 1. Każdy watek dostaje pojedynczo element do wylosowania i prosi o nastepny. powoduje to opóźnienia które widać na wykresie. Warte uwagi sa również anomalia czasowe na poczatkach wykresów. Dzieje sie tak dlatego, że dla małego problemu zrównoleglenie może powodować zakłócenia zwiazane z samym procesem zrównoleglenia tzn. komunikacja miedzy procesami i delegowaniem kolejnych batchy do pracy.

2.5 Static i Dynamic schedule dla różnych rozmiarów batch.

Widzac jak zachowuja sie poszczególne rodzaje schedulowania, sprawdziłem jak wygladać bedzie wykres dla różnych rozmiarów chunka. Wyrażam go tutaj jako procent całego problemu. Najlepsze rozstawienie bedzie dla 25 procent, ponieważ wtedy każdy watek dostanie cały problem do wykonania na raz i nie bedzie utraty czasu na wysyłanie kolejnego batcha pracy. Wyniki przedstawione na Rysunek 5.



Rysunek 5: Wykres czasu schedulowania dla różnych wielkości chunków.

2.6 Wnioski z eksperymentu z różnymi wielkościami chunków.

Dla naszego problemu najwieszke znaczenie miała wartość chunka. Ustawiony za duży powodował, że cały problem mógł wykonać sie sekwencyjnie, za mały powodował niepotrzebne prośby o nastepne batche. Na wykresach przedstawiajacych czas wykonania programu od wielkości chunka widać, że dla obu rodzajów schedulowania najlepszy wynik osiagamy przy 25 procent. Dzieje sie tak ponieważ cały problem udaje sie perfekcyjnie zrównoleglić i każdy watek wykonuje prace na raz i ma taki sam rozmiar problemu.

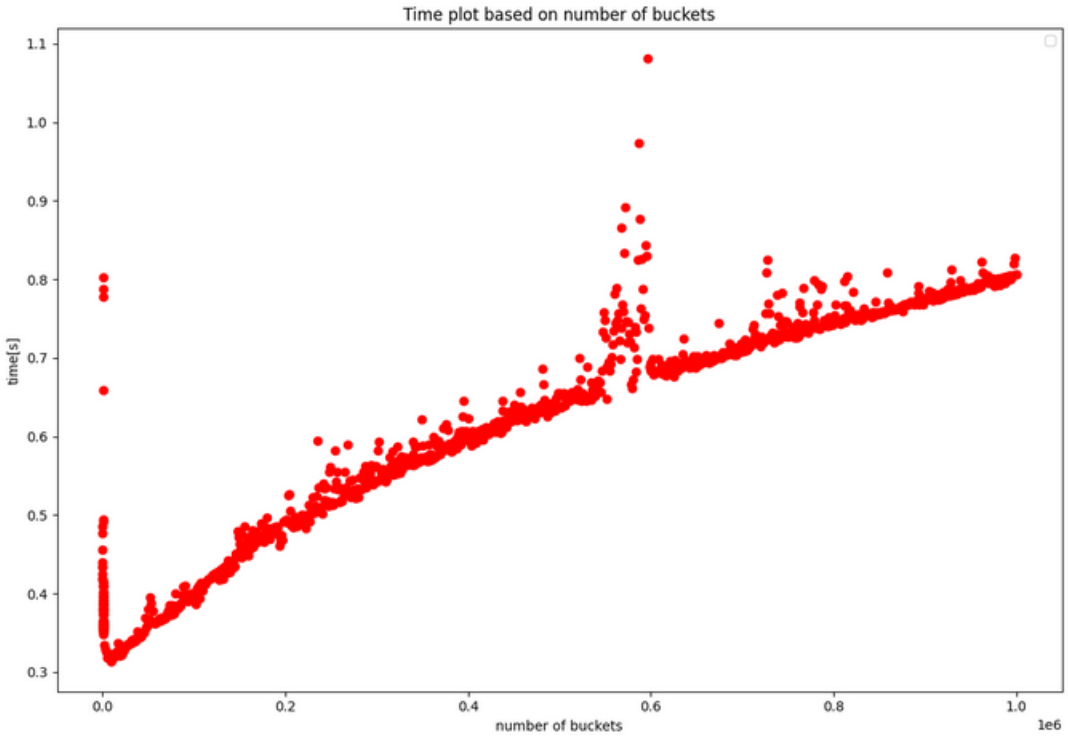
3 Sortowanie kubełkowe.

3.1 Opis środowiska.

Badanie algorytmu kubełkowego przeprowadzono na superkomputerze Ares. Według specyfikacji, ma on 532 wezły, CPU 48 cores, Intel(R) Xeon(R) Platinum 8268 CPU @ 2.90GHz i 192GB pamieci RAM.

3.2 Wybór rozmiaru problemu oraz ilości kubełków.

Rozmiar tablicy wynosi 10^6 B. Dla takiego rozmiaru sprawdziłem jak zachowuje sie algorytm sekwencyjnego sortowania kubełkowego w zależności od ilości kubełków. Wyniki przedstawione na Rysunku 6.



Rysunek 6: Wykres czasu sortowania sekwencyjnego dla różnych wielkości kubełków.

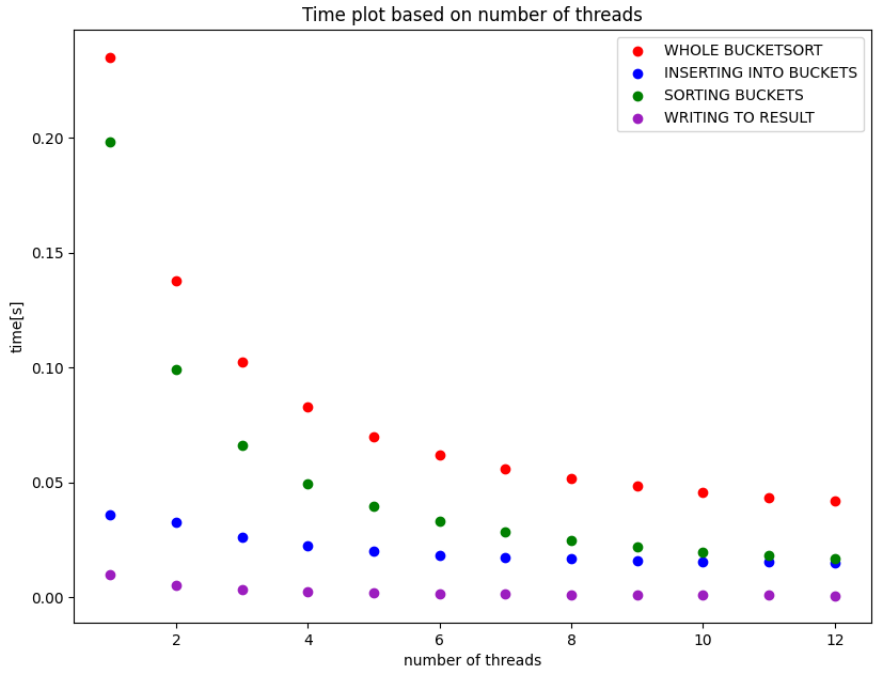
Z wykresu można wywnioskować, że najlepszy rozmiar znajduje sie blisko 1000. Według teorii dla sortowania kubełkowego powinno wybrać sie pierwiastek wielkości tablicy jako ilość kubełków, co zgadza sie z wynikami na wykresie. Z tego wzgledu dalej bede testował algorytm dla 1000 kubełków.

3.3 Przeprowadzenie badań.

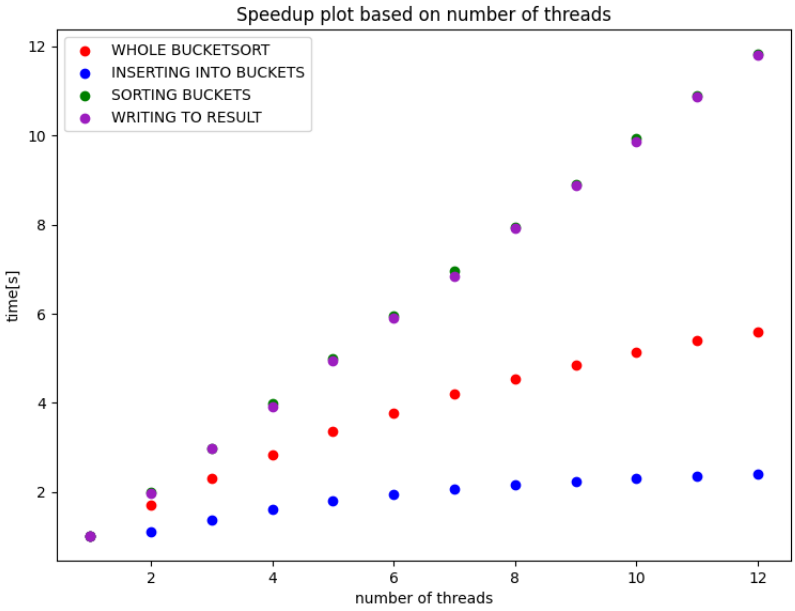
Algorytm sortowania został uruchomiony 20 razy dla ilości watków od 1 do 12 na superkomputerze Ares. Wyniki z każdej próby zostały uśrednione a nastepnie przeanalizowane.

3.4 Wyniki testowania równoległego algorytmu sortowania kubełkowego.

Na Rysunku 7 i Rysunku 8 przedstawiłem czas oraz przyspieszenie policzone na podstawie prawa Amdhala:



Rysunek 7: Wykres czasu sortowania równoległego dla różnej ilości procesorów.



Rysunek 8: Wykres przyspieszenia sortowania równoległego dla różnej ilości procesorów.

3.5 Wnioski z przeprowadzonych badań sortowania kubełkowego równoległego.

- Czas wykonania całego algorytmu skaluje sie wraz z dodaniem kolejnych procesorów.
- Najmniej czasu zajmuje wpisywanie posortowanych kubełków do wynikowej tablicy.
- Najdłużej trwa sortowanie kubełków.
- Wpisywanie do kubełków nie skaluje sie dobrze, ponieważ każdy watek wykonuje taka sama prace.
- Wpisywanie do wynikowej tablicy nie skaluje sie dobrze. Moim zdaniem spowodowane jest to, że każdy watek musi policzyć przesuniecie, od którego musi zaczyać wpisywanie tablicy dodatkowo samo wpisywanie zajmuje bardzo mało czasu nawet dla sekwencyjnej wersji problemu.
- Każdy element algorytmu w jakimś stopniu skaluje sie wraz z dodaniem kolejnych procesorów.

4 Kod, opis użytych struktur danych oraz sekcja parallel.

Kod programu przedstawiony na Listingu 1. Co ważne cały algorytm został zamkniety w jednej sekcji parallel. Do przechowywania wartości został użyty vector, a pomiar czasu liczony jest za pomoca sekcji master.

Listing 1: Równoległe sortowanie kubełkowe.

```
#include <iostream>
#include <cstdlib>
#include <random>
#include <string>
#include <omp.h>
#include <typeinfo>
#include <algorithm>

using namespace std;

double times[4];

void bucket_sort(vector<long long> &v, long long number_of_buckets, int threads)
{
    //   cigamy   rozmiar tablicy i tworzymy kube ki
    const long long n = v.size();
    vector<vector<long long>> buckets(number_of_buckets);

    double start_time_putting_in_buckets;
    double start_time_sorting_elelements;
    double start_time_writing;

#pragma omp parallel
    {
        // id i ilo    w tk w
        int thread_id = omp_get_thread_num();
        long long thread_count = omp_get_num_threads();

// *** WRITING INTO BUCKETS ***
#pragma omp master
        {
            start_time_putting_in_buckets = omp_get_wtime();
        }

        // definiujemy zakresy kube ka
        long long thread_offset = n / thread_count;
        long long bucket_lower = thread_id * (number_of_buckets / thread_count);
        long long bucket_upper = (thread_id + 1) * (number_of_buckets / thread_count);

        if (thread_id == thread_count - 1)
            bucket_upper = number_of_buckets;

        // Umieszczamy elementy we w a ciwych kube kach iteruj c przez ca    tablic  zaczyna
        for (long long i = thread_offset; i < n; ++i)
        {
            long long bucket_index = (number_of_buckets * v[i]) / n;
            if (bucket_index >= bucket_lower && bucket_index < bucket_upper)
            {
                buckets[bucket_index].push_back(v[i]);
            }
        }
        for (long long i = 0; i < thread_offset; ++i)
        {
```

```
            long long bucket_index = (number_of_buckets * v[i]) / n;
            if (bucket_index >= bucket_lower && bucket_index < bucket_upper)
            {
                buckets[bucket_index].push_back(v[i]);
            }
        }

#pragma omp master
        {
            times[1] = omp_get_wtime() - start_time_putting_in_buckets;
        }

        // bariera, poniewa  mo e by  tak,  e  jaki  w tke sko czy prac  szybciej a teraz b
#pragma omp barrier

// *** SORTING BUCKETS ***
#pragma omp master
        {
            start_time_sorting_elelements = omp_get_wtime();
        }
#pragma omp for schedule(static)
        for (int i = 0; i < number_of_buckets; i++)
        {
            sort(buckets[i].begin(), buckets[i].end());
        }

#pragma omp master
        {
            times[2] = omp_get_wtime() - start_time_sorting_elelements;
        }

// *** WRITING INTO RESULT LIST ***
#pragma omp master
        {
            start_time_writing = omp_get_wtime();
        }
        // Wpisywanie posortowanych bucket w do tablicy
        // Dla ka dego  w tku  b dzie  r   ny start = i
        // ka dy policzy sobie prefix sum i na jego podstawie powpisuje odpowiednie buckety na odp
        int last_bucket = 0;
        int prev_buckets_sizes = 0;
#pragma omp for schedule(static)
        for (int i = 0; i < number_of_buckets; i++)
        {
            for (int j = last_bucket; j < i; j++)
            {
                prev_buckets_sizes += buckets[j].size();
            }
            last_bucket = i;
            int e = prev_buckets_sizes;
            for (int k = 0; k < buckets[i].size(); k++)
            {
                v[e] = buckets[i][k];
                e++;
```

```
            }
        }
#pragma omp master
        {
            times[3] = omp_get_wtime() - start_time_writing;
        }
    }
}

int main(int argc, char **argv)
{

    if (argc != 4)
    {
        printf("invalid number of arguments");
        return 1;
    }

    //  Pobranie parametr w programu
    unsigned long long int arr_size = stoull(argv[1]);
    int threads = atoi(argv[2]);
    int number_of_buckets = atoi(argv[3]);

    // Stworzenie tablicy do posortowania
    vector<long long> data = vector<long long>(arr_size);

    // Ustalenie ilo ci dzia aj cych w tk w
    omp_set_dynamic(0);
    omp_set_num_threads(threads);

    // Generator seedujemy osobno dla ka dego w tku
#pragma omp parallel default(none) shared(data, arr_size)
    {
        mt19937_64 rng(random_device{}());
        uniform_int_distribution<long long> distribution(1, arr_size - 1);

        // Losowanie liczb do tablicy
#pragma omp for schedule(static)
        for (int I = 0; I < arr_size; I++)
            data[I] = distribution(rng);
    }

    // Wywo anie wsp  bie nego sortowania
    double start, end;
    start = omp_get_wtime();
    bucket_sort(data, number_of_buckets, threads);
    end = omp_get_wtime();
    times[0] = end - start;

    // Sprawdzanie czy posortowana
    for (int i = 0; i < arr_size - 1; i++)
    {
        if (data[i + 1] < data[i])
        {
```

```
            cout << "ERROR" << endl;
        }
    }

    // Koniec programu
    return 0;
}
```

5 Porównanie obu rodzajów implementacji.

5.1 Przedstawienie drugiego algorytmu.

Drugi algorytm opiera sie o nastepujace zasady:

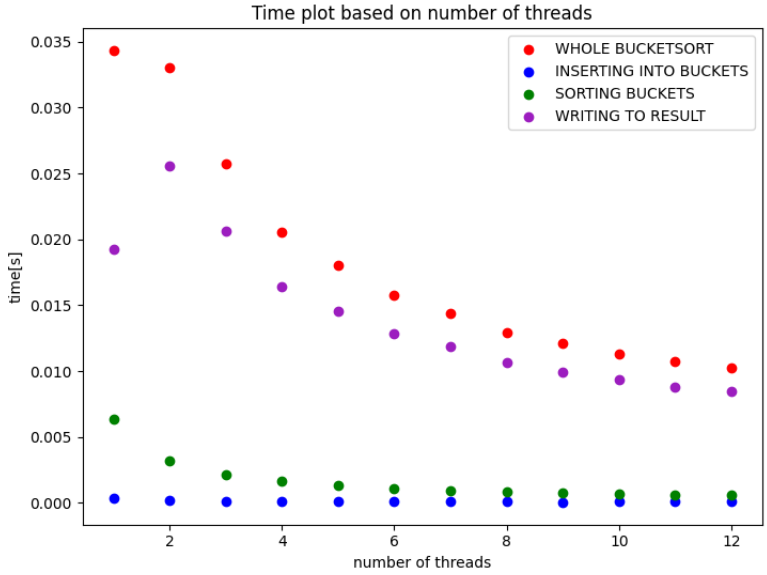
1. Każdy watek czyta przydzielony do niego fragment tablicy.
2. Wszystkie watki współdziela kubełki przy czytaniu poczatkowej tablicy.
3. Nastepnie każdy watek dostaje odpowiednie kubełki do posrotowania.
4. Każdy watek wpisuje posortowane kubełki do odpowiedniego miejsca w tablicy.

Różnica od pierwszego rodzaju algorytmu jest taka, że przy zapisie do kubełków może wystapić konflikt. Wymagane jest w takim wypadku użycia jakiegoś mechanizmu ochrony danych wspólnych.

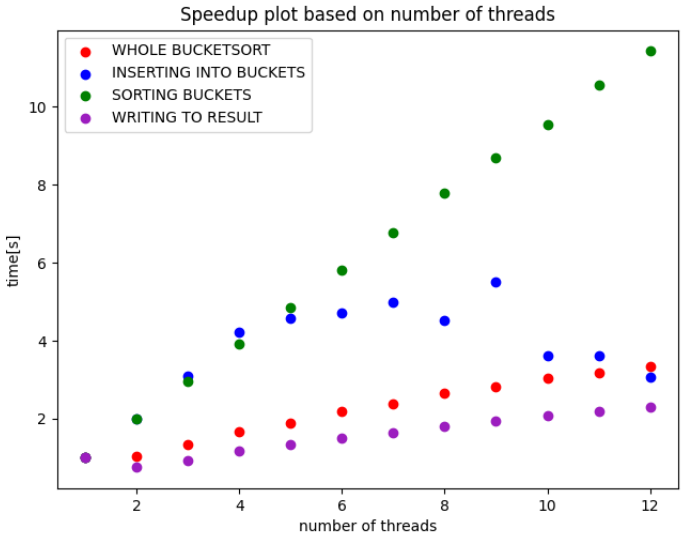
Algorytm został przetestowany dla wielkości tablicy 2^20 B i 10^3 kubełków.

5.2 Porównanie wyników algorytmów.

Na Rysunek 9 i Rysunek 10 przedstawione badania algorytmu drugiego implementowanego przez druga osobe z zespołu.



Rysunek 9: Wykres przyspieszenia sortowania równoległego dla różnej ilości procesorów (algorytm drugi).



Rysunek 10: Wykres przyspieszenia sortowania równoległego dla różnej ilości procesorów (algorytm drugi).

Porównujac z wynikami algorytmu pierwszego:

1. Oba algorytmy skaluja sie liniowo wraz z dodawaniem kolejnych watków.
2. W pierwszym algorytmie widać, że najdłużej trwa sortowanie kubełków, a w drugim wpisywanie do tablicy wynikowej.
3. Problem z wykresami dla algorytmu drugiego jest bład w pierwszych pomiarach. Dla wpisywania do tablicy wynikowej widać wyłamanie z trendu. Spowodowane może być to mała ilościa powtórzeń doświadczenia lub dziwnym zachowaniem algorytmu równoległego dla jednego procesora.