AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Open MP

Maciej Mucha, Dmytro Zhylko

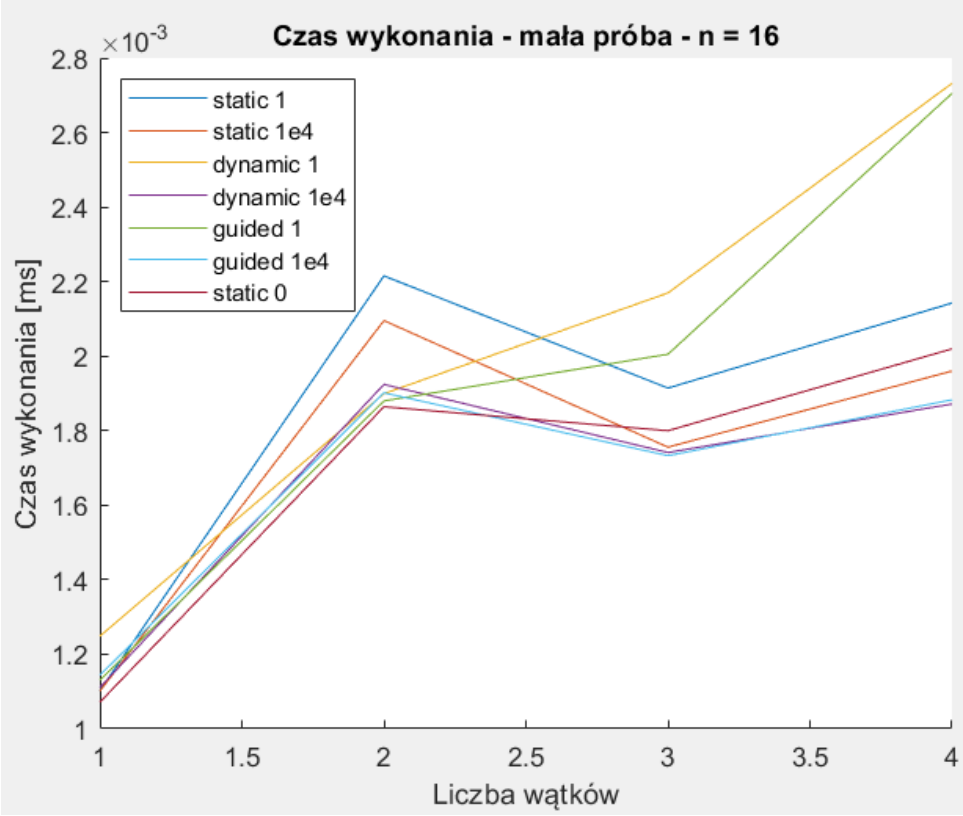
1. Pomiar czasu, przyspieszenia i efektywności

Opis oznaczeń:
n – liczba elementów w tablicy, zmienne miały typ double
Oznaczenia serii składają się z typu harmonogramowania i rozmiaru kawałka (chunk)
Test składał się z 4 różnych rozmiarów danych n = 16, 1e4, 1e6, 1e8 oraz z typu harmonogramowania static 1, static 1e4, dynamic 1, dynamic 1e4, guided 1, guided 1e4, static (bez parametru chunk) przetestowanego na każdym rozmiarze.

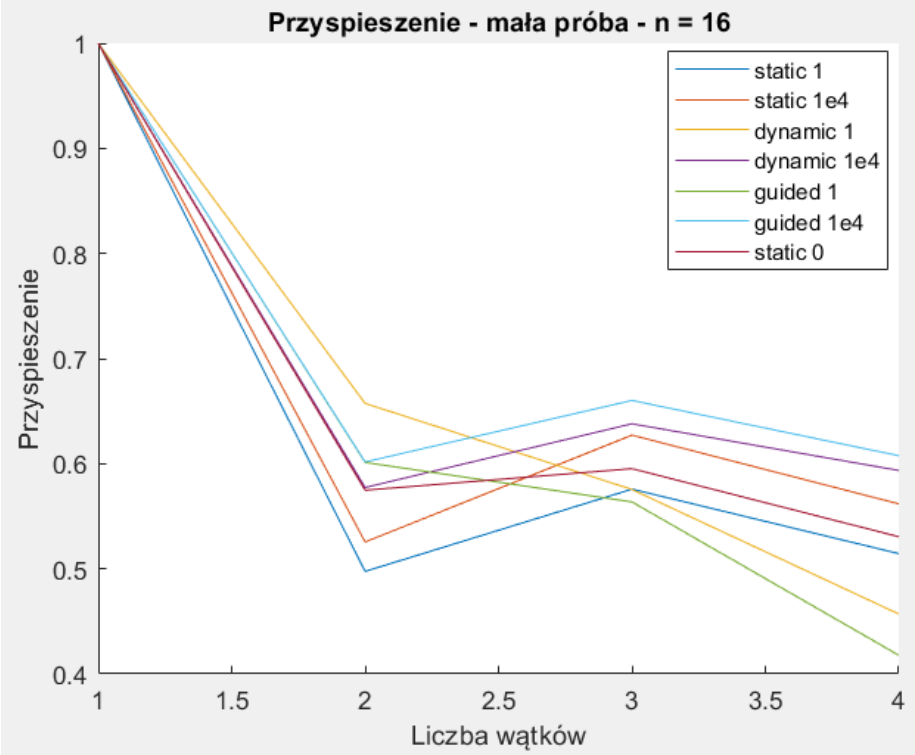
Warto zauważyć, że w przypadku guided rozmiar kawałka to jest rozmiar najmniejszego kawałka na jakie zostaje dzielona tablica, a wątki dostają kawałki większe lub równe tej wielkości. Inne typy harmonogramowania przydzielają wątkom kawałki równe tej wartości.

Mała próba, n = 16

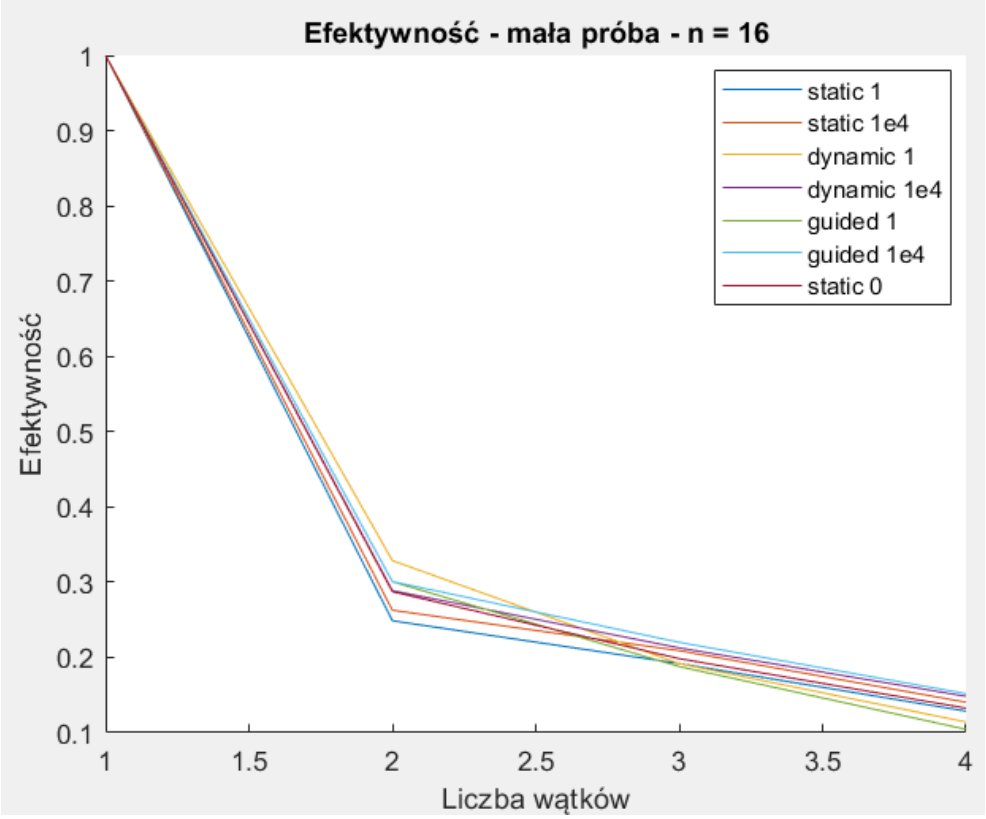
W tak małej próbie podział na wątki jest zbyteczny i spodziewamy się spowolnienia przy uruchamianiu równoległym.
Z uwagi na bardzo małe czasy wykonania, mierzony był czas wykonywania tysięcy przejść po tablicy tak aby mierzony czas wynosił około 1s.



Wykres 1. Czas wykonania małej próby n=16



Wykres 2. Przyspieszenie małej próby n=16

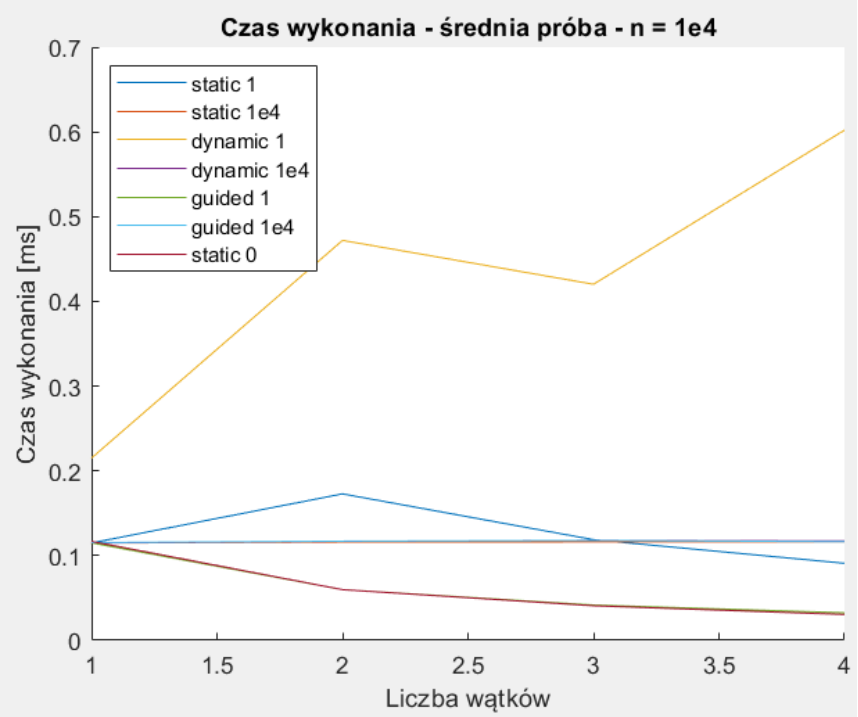


Wykres 3. Efektywność wykonania małej próby n=16

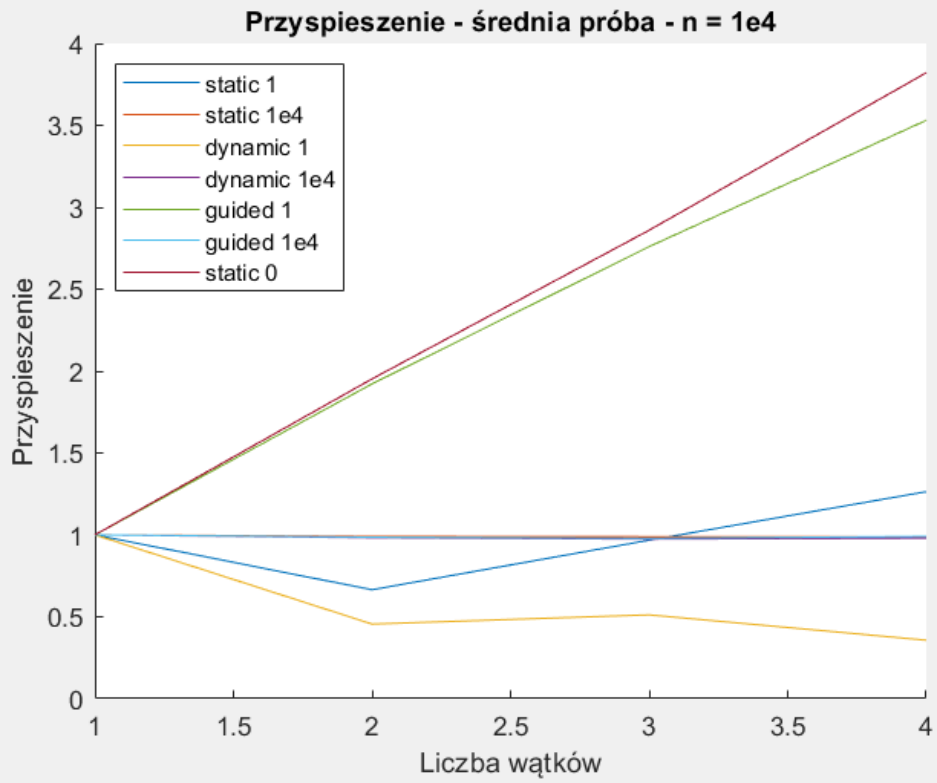
Zgodnie z oczekiwaniami czas wzrósł i przyspieszenie było poniżej 1. Dla tak małego problemu nie należy stosować wielu wątków. Ciężko jest coś powiedzieć o jakości harmonogramowań dla tego przykładu.

Średnia próba n = 1e4

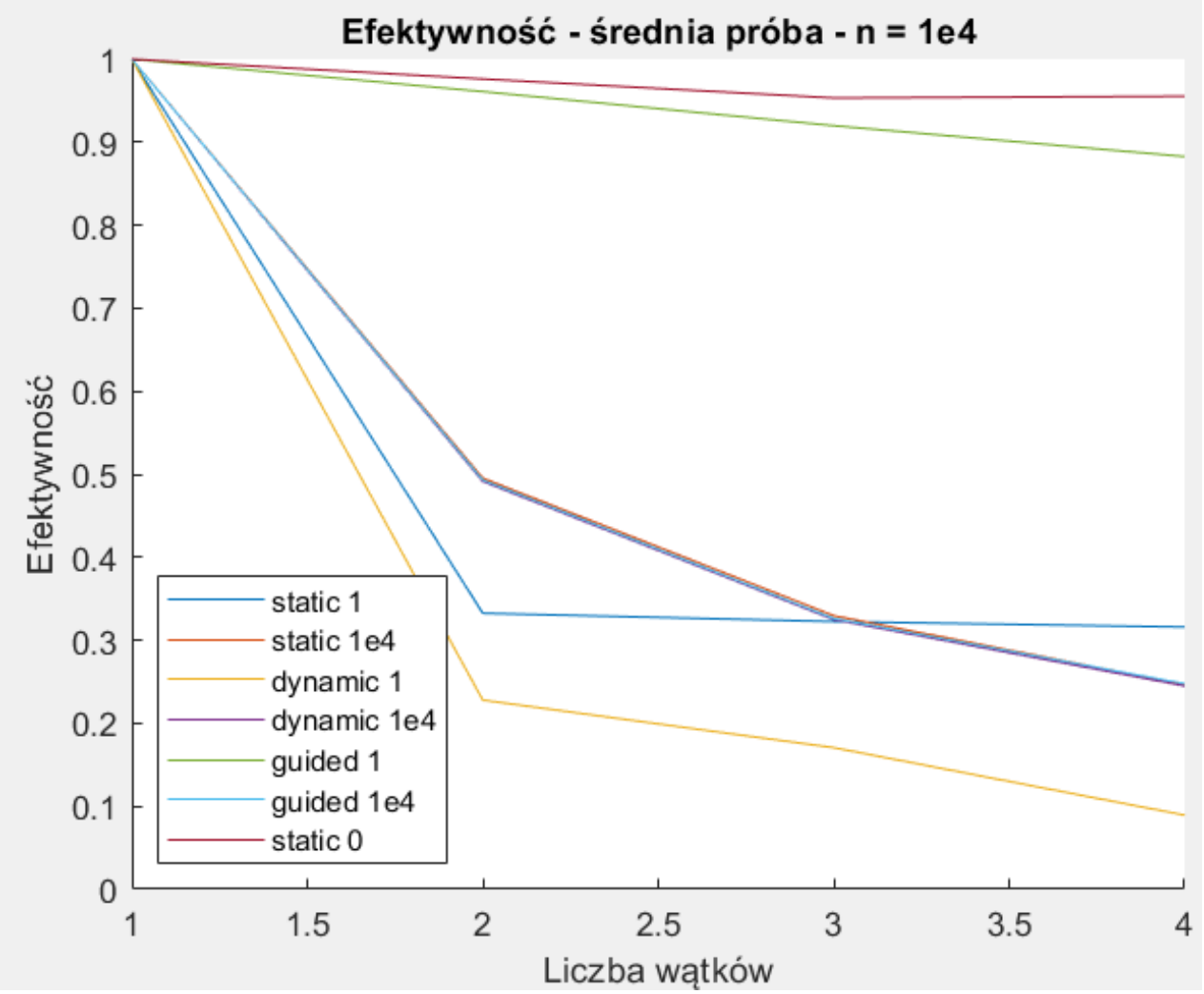
Ta próba została dobrana tak aby rozmiar problemu wynosił rozmiarowi kawałka w niektórych testach.



Wykres 4. Czas wykonania średniej próby n=1e4



Wykres 5. Przyspieszenie średniej próby n=1e4

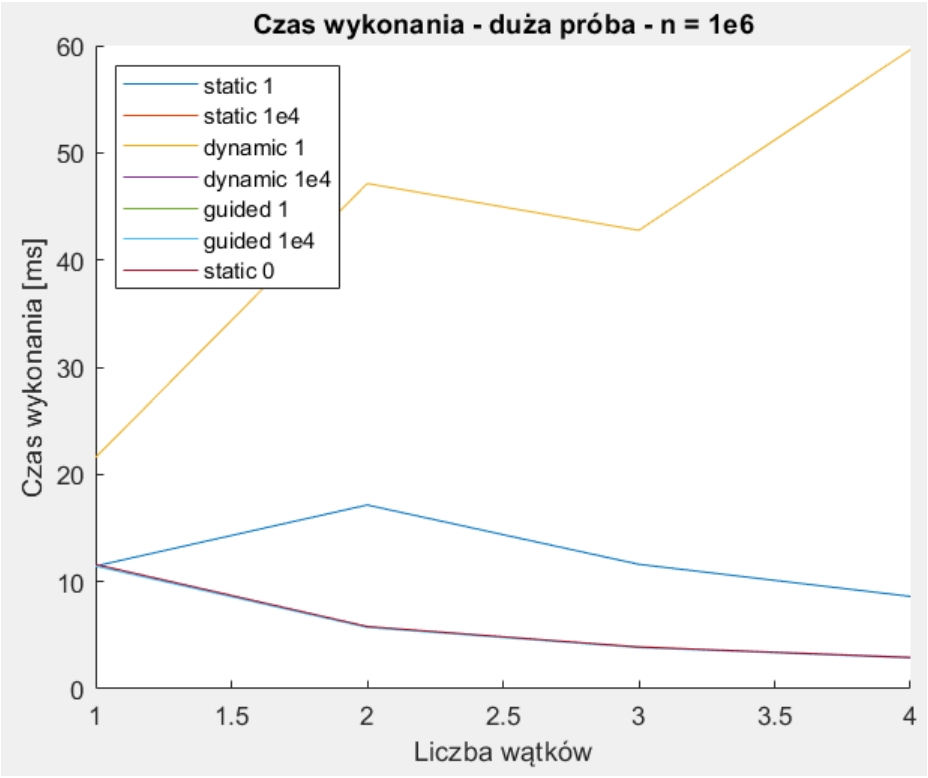


Wykres 6. Efektywność średniej próby n=1e4

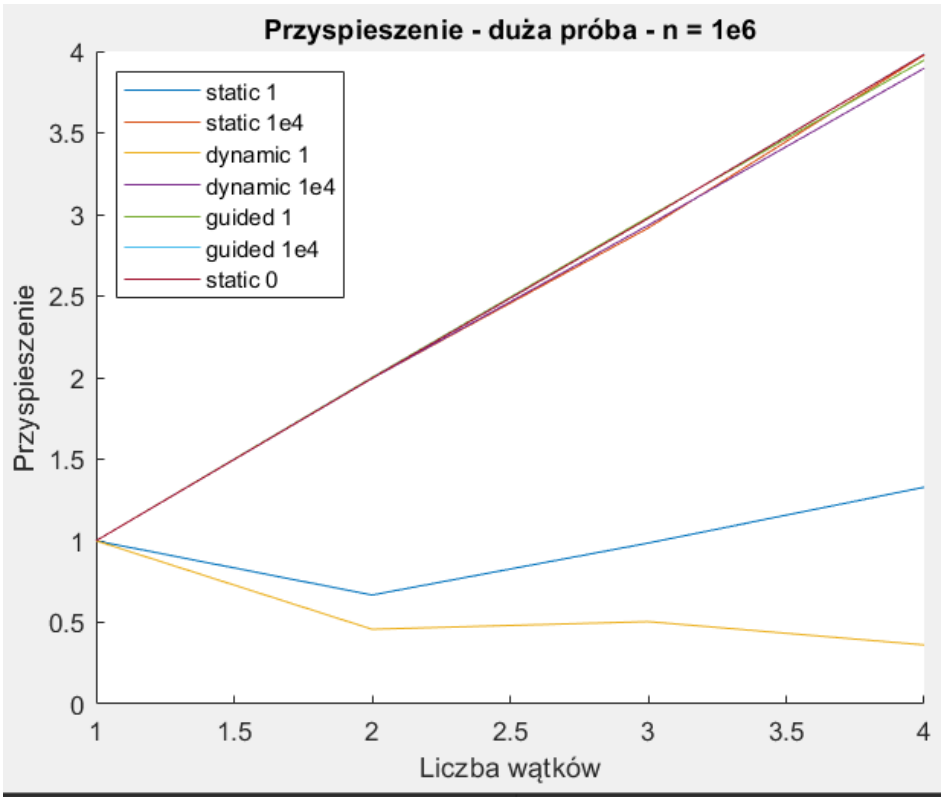
Zgodnie z oczekiwaniami testy w rozmiarem kawałka równym rozmiarowi tablicy osiągnęły przyspieszenie ~1. Całą praca spadła na 1 wątek, a inne były bezczynne.
Pozostałe testy wykazały, że jedynie guided 1 i static 0 osiągnął przyspieszenie. Wynik opcji guided 1 jest skutkiem faktu opisanego na początku rozdziału, tablica nie jest dzielona na kawałki o rozmiarze 1, tylko na kawałki proporcjonalne ilości iteracji i odwrotnie proporcjonalne ilości wątków, co zapewnia odpowiednią granulacje, z najmniejszym kawałkiem o rozmiarze równym 1. Opcja static 0 dzieli tablice na kawałki o równej wielkości, co daje najlepsze efekty. Static 1 oraz Dynamic 1 osiągnęły spowolnienie, co było gorsze niż oddanie całej pracy jednego wątkowy (testy z kawałkiem 1e4). Jest to spowodowane narzutem związanym z organizacją wątków.

Duża próba n = 1e6

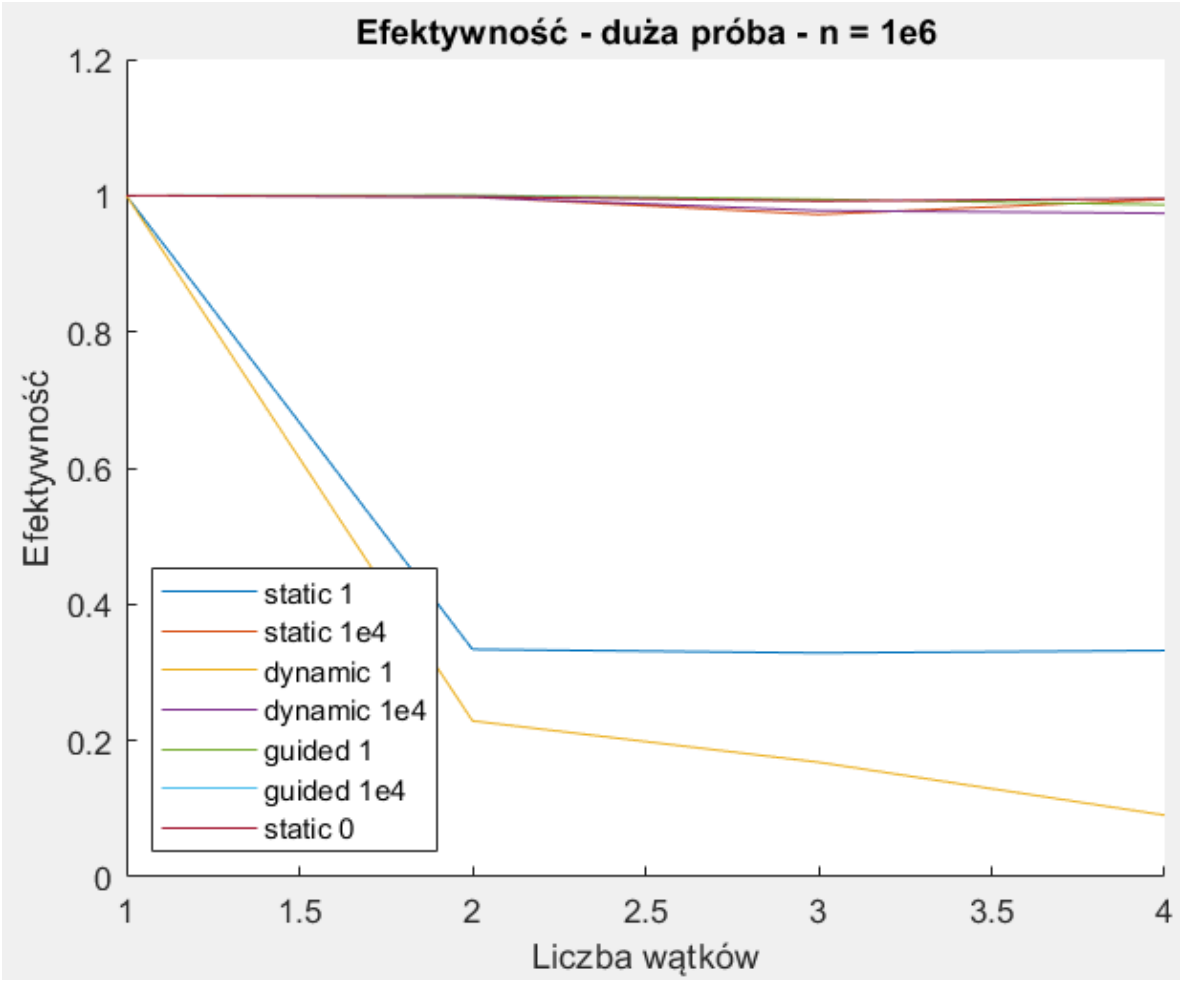
Próba pozwalająca na porównanie wszystkich 6 rodzajów harmonogramowań. Podział na kawałki o rozmiarze 1e4 da 100 kawałków.



Wykres 7. Czas wykonania dużej próby n=1e6



Wykres 8. Przyspieszenie dużej próby n=1e6

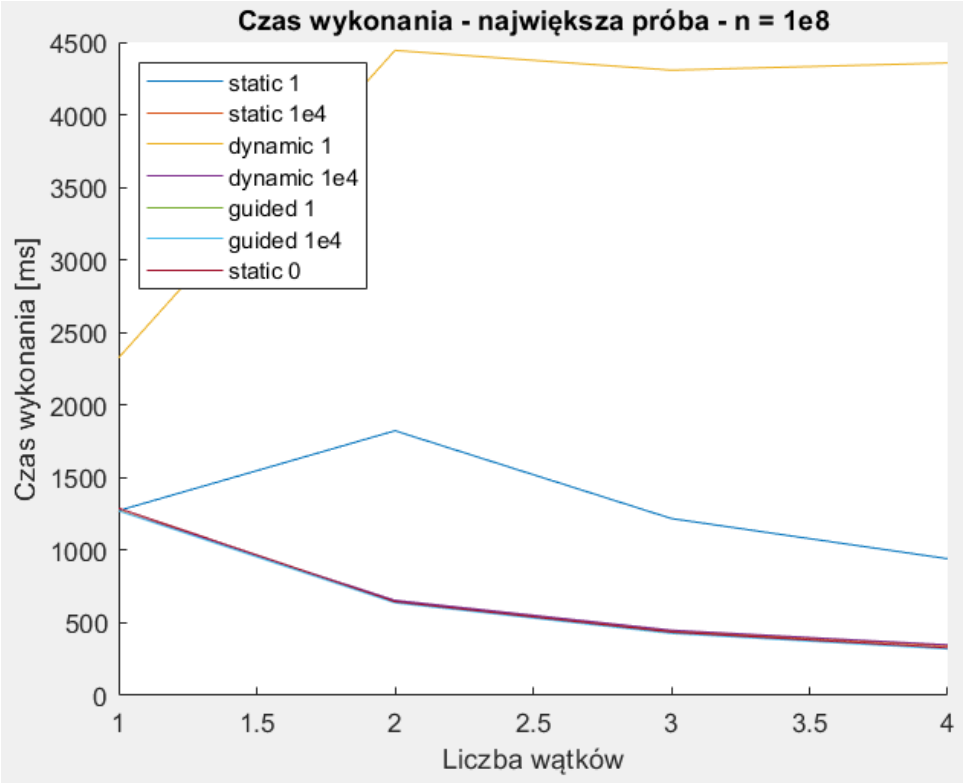


Wykres 9. Przyspieszenie dużej próby n=1e6

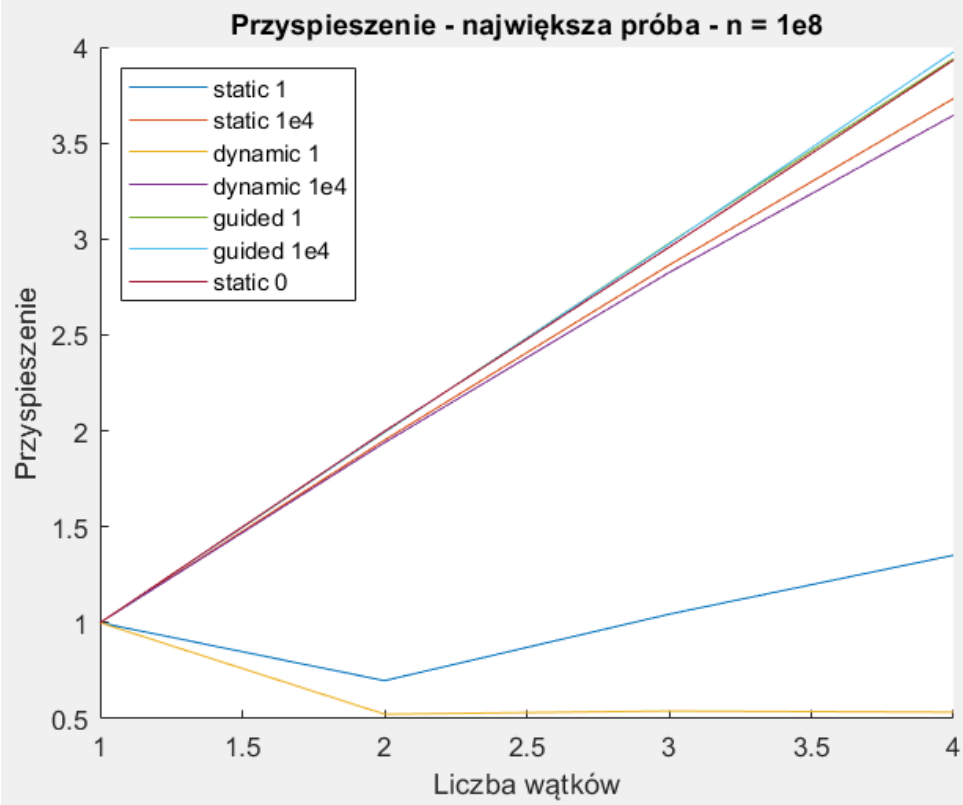
Efektywność bliską 1 osiągnęły static 1e4, dynamic 1e4, guided 1, guided 1e4 oraz static 0 Utrzymujący się na poziomie ponad 0.97 dla 3 i więcej wątków oraz 0.998 na 2 wątkach. W tej sytuacji podział na 100 kawałków zapewnił odpowiednią granulacje. Zarówno static 1 jak i dynamic 1 spowodowały spowolnienie, co jest zgodne ze średnią próbą.

Największa próba n = 1e8

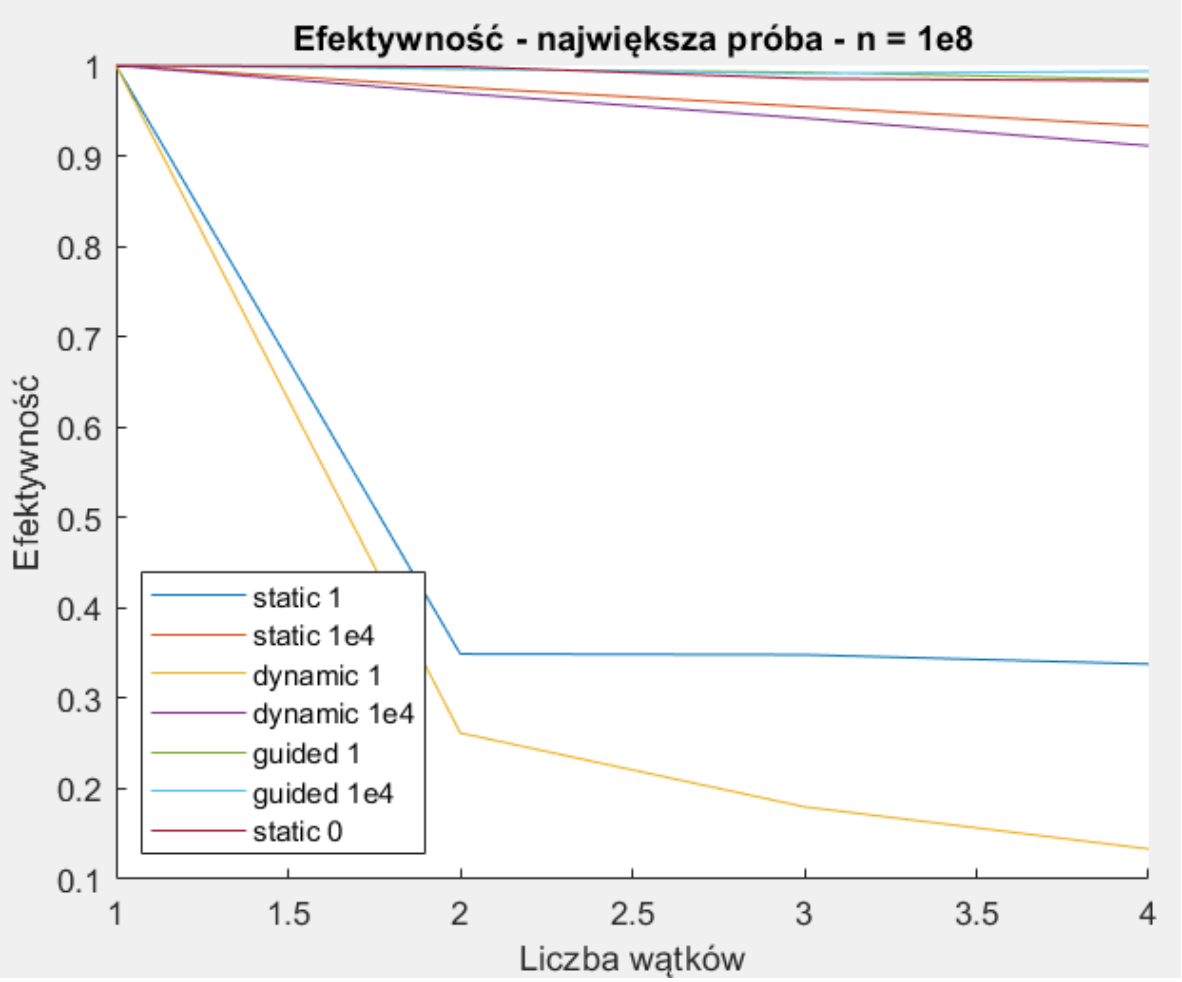
Testy dla największej możliwej tablicy.



Wykres 10. Czas wykonania największej próby n=1e8



Wykres 11. Przyspieszenie największej próby n=1e8



Wykres 12. Efektywność największej próby n=1e8

Największa wykonywana próba. Static 1 oraz dynamic 1 spowolniły program co jest spójne z poprzednimi testami.
Dla 4 wątków efektywności zostały przedstawione w poniższej tabeli.

Typ	Guided 1e4	Guided 1	Static 0	Static 1e4	Dynamic 1e4
Efektywność	0.994	0.985	0.983	0.933	0.911

Wszystkie opcje osiągnęły efektywność ponad 91%. Najszybsze okazały się w kolejności guided 1e4, guided 1, static 0. Złożone dzielenie tablicy na wątki, które jest oferowane przez harmonogrowanie guided dało lepsze efekty niż prosty podział tablicy na 4 części (4 wątki).

Wnioski

Dla małych tablic nie opłaca się używać wielu wątków.
Dzielenie na kawałki o rozmiarze 1 spowalnia program pomimo użycia wielu wątków.
Najprostsze dzielenie na bloki o równym rozmiarze (static 0) daje bardzo dobre efekty, zbliżone do najlepszych.
Złożone dzielenie (guided) daje bardzo dobre efekty, zbliżone do najlepszych, i nie jest czułe na podawany rozmiar kawałka. W przypadku złożonych problemów może pozwolić na uzyskanie bardzo

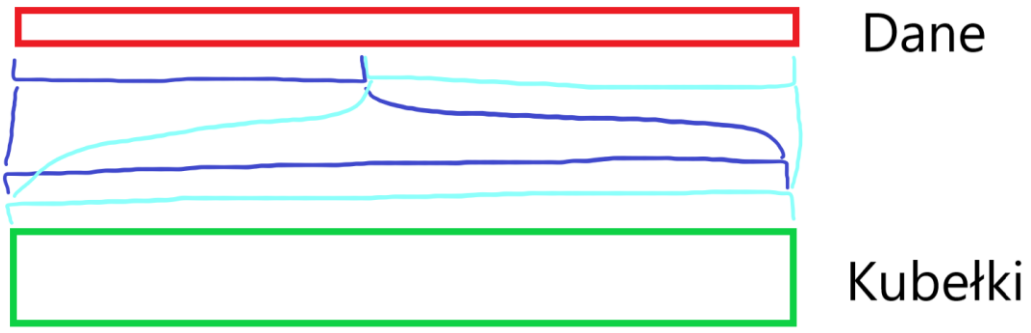
dobrego wyniku bez konieczności poszukiwania najlepszego parametru.
Static 1e4 oraz Dynamic 1e4 dawały podobne rezultaty. Wynika to z natury problemu - dużej liczby takich samych operacji z których wykonanie każdej zajmuje tyle samo czasu. Podawany parametr rozmiaru kawałka jest istotny i decyduje o jakości rozwiązania. Porównanie dużej i największej próby pokazuje, że istnieje optymalny parametr, który należy wyznaczyć dla danego problemu. W przeciwnym razie stracimy na wydajności. Dla dużej próby różnica pomiędzy static 1e4 i dynamic 1e4, a static 0 była znacznie mniejsza niż w największej próbie, co oznacza, że dla problemu o n = 1e6 wielkość kawałka 1e4 była lepiej dobrana niż w problemie o n = 1e8.

Analizowany problem zawierał przeprowadzenie dużej liczby takich samych operacji, z których każda zajmuje bardzo zbliżony czas. Dla innych typów problemów, takich w których czas pojedynczej operacji jest różny, użycie harmonogramowania statycznego może nie dawać najlepszych rezultatów, ponieważ czas wykonania programu to czas w jakim najwolniejszy wątek wykona swoją pracę i czas ten jest najmniejszy, jeśli wszystkie wątki pracują cały czas.

2. Propozycje sortowania kubełkowego

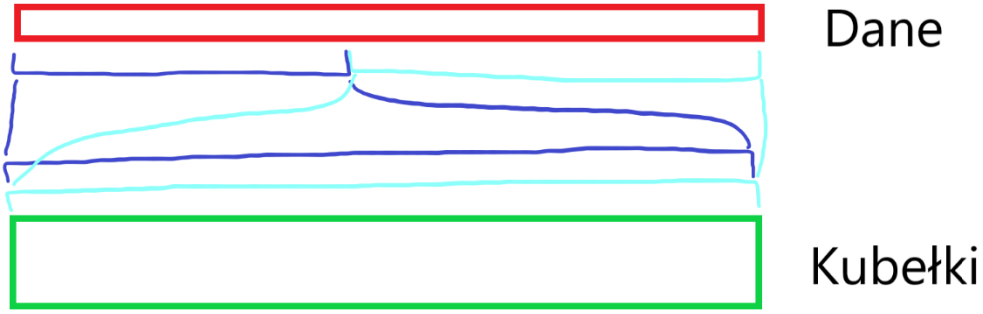
Przykłady skupiają się na 2 wątkach, ale możne je w trywialny sposób uogólnić na większą liczbę wątków. Na szkicu kolorek czerwonym oznaczono tablicę z danymi, zielonym tablice z kubełkami, różnymi odcieniami niebieskiego 2 wątki i jakie obszary tablic dotykają.

1. Podział danych na części i przydzielenie jednej do każdego wątku



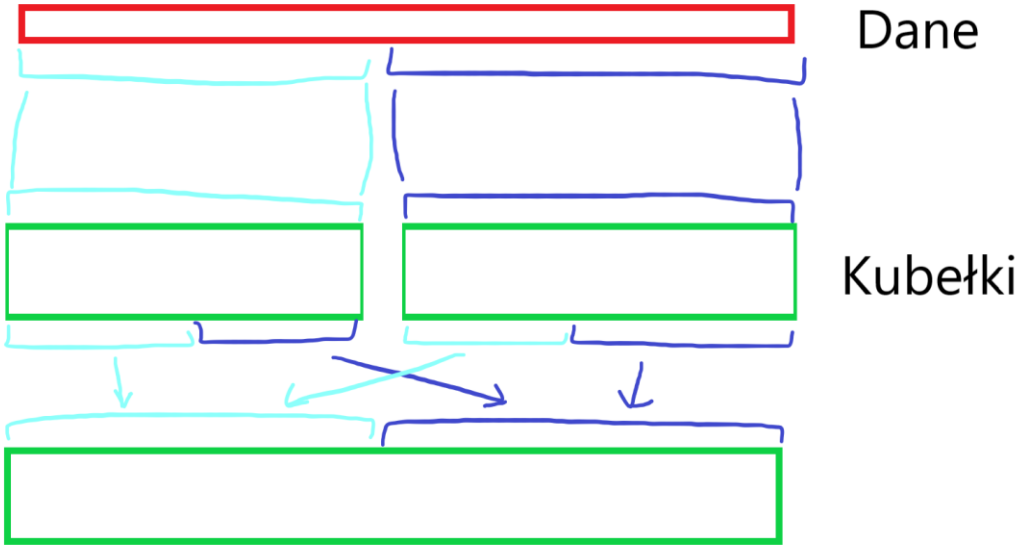
Każdy element tablicy z danymi jest przeglądany tylko przez 1 wątek co jest najszybsze.
Wiele wątków może pisać do tego samego kubełka co wymaga monitora. Każdy wątek przegląda jedynie część tablicy z danymi

2. Podział kubełków na części i przypisanie do każdego wątku



Każdy wątek przegląda całą tablicę z danymi, ale ignoruje wartości, które miałyby trafić do kubełków, które nie są mu przypisane. Konflikt na czytaniu elementów, ale brak konfliktu na zapisywaniu do kubełków.

3. Każdy wątek dostaje swój kubełek



Wymaga 2 etapów, w pierwszym każdy wątek wypełnia swój własny kubełek analizując jedynie część danych wejściowych, brak konfliktów na czytaniu ani na zapisie. W drugim etapie wszystkie kubełki są łączone w jeden wyjściowy, co też da się przeprowadzić równolegle, bez konfliktów na czytaniu ani na zapisie. Wymaga więcej pamięci więcej operacji, ale bez konfliktów.

Część 2

Z powodów technicznych wielkość tablicy została ograniczona do 5.000.000.

Zbadanie poprawności danych wejściowych

Test uruchamiany z 4 wątkami, wielkość tablicy 1e6, liczba kubełków 1e6. Idealny rozkład spowodowałby, że każdy kubełek będzie miał rozmiar 1.

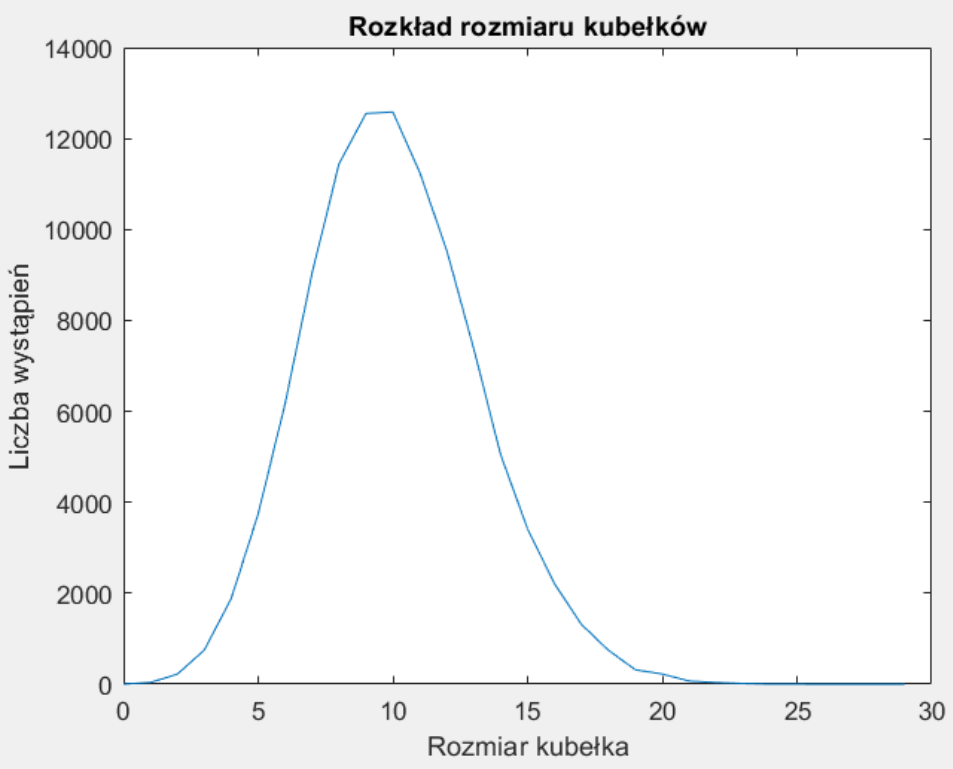
Rozmiar kubełka	0	1	2	3	4	5	6	7	8	9	10	11
Liczba wystąpień	367228	368680	184038	61275	15161	3045	481	77	11	3	1	0
Procent całości	36.7%	36.9%	18.4%	6.1%	1.5%	0.3%	0.04$	0%	0%	0%	0%	0%

Funkcje erand48(), rand() i std::uniform_real_distribution testowane dla problemu sekwencyjnego dają podobne rezultaty.

Jednak liczba kubełków równa liczbie elementów nie jest rzeczywistym przypadkiem. Kolejny test został wykonany, gdzie liczba elementów wynosił 1e6, ale liczba kubełków to 1e5, 10% z liczby elementów.

Rozmiar kubełka	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Procent wystąpień [%]	0	0	0	1	2	4	6	9	11	12	13	11	10	7	5	3	2	1	1	0	0

Oraz w formie wykresu



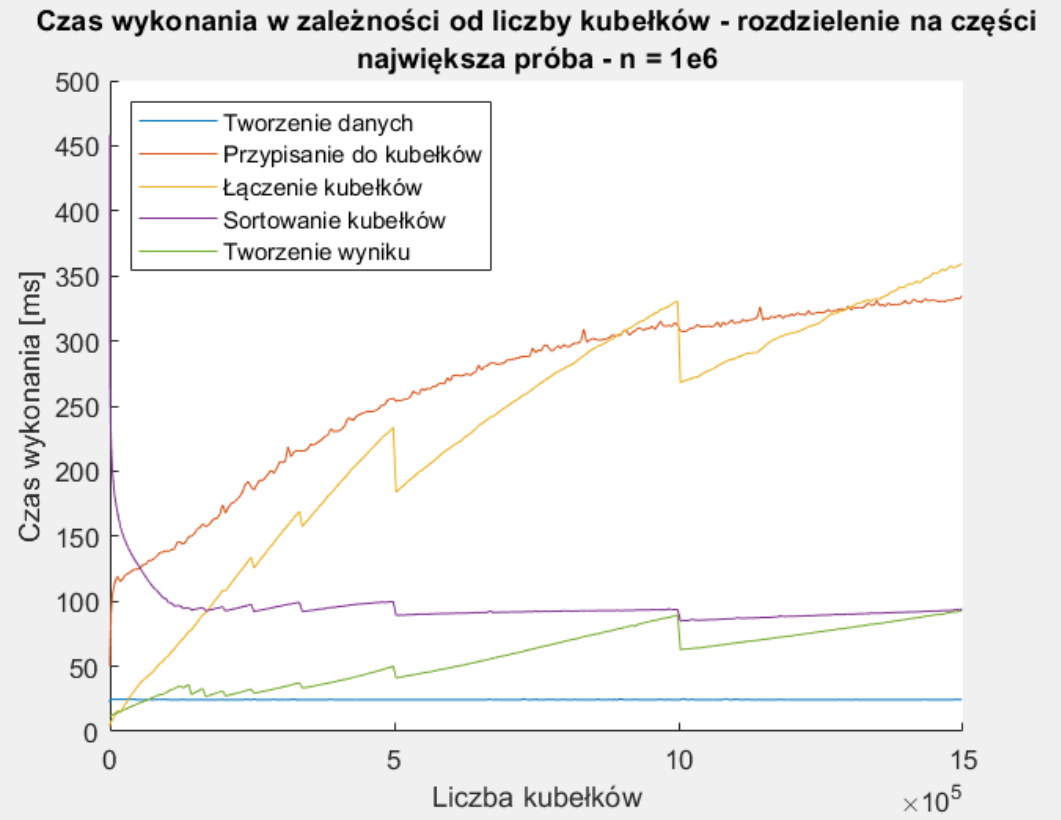
Wykres 13. Rozkład rozmiaru kubełków przy podziale

Wykres 13. Rozkład rozmiaru kubełków przy podziale Osiąga maksimum dla rozmiaru równym 10, co jest pożądanym rezultatem. Aby algorytm działał poprawnie, liczba elementów w kubełkach musi być stała i niezależna od liczby elementów. Warunek ten jest spełniony, sortowanie wewnątrz kubełka można uznać za wykonywalne w czasie stałym. Aby algorytm działał efektywnie, wszystkie kubełki powinny mieć taki sam rozmiar. Zdecydowana większość kubełków ma rozmiar zbliżony do 10, więc ten warunek też można uznać za spełniony.

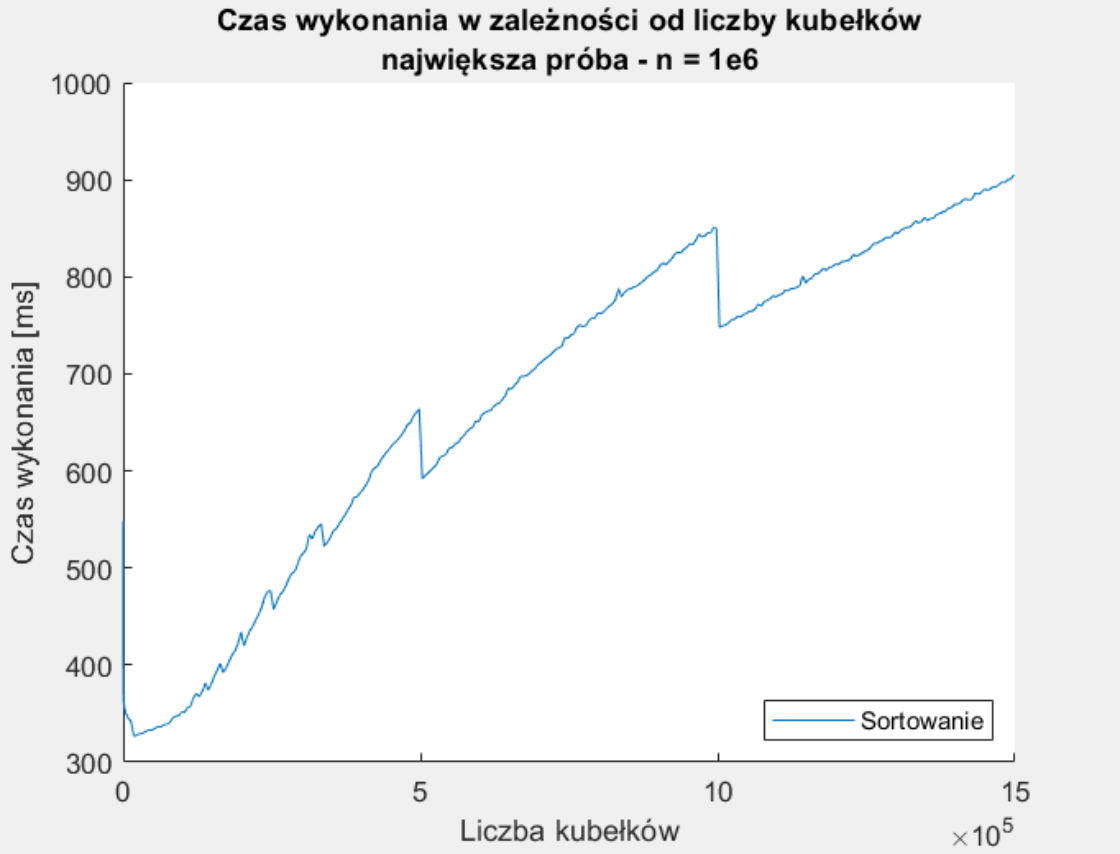
Badanie algorytmu sortowania sekwencyjnego

Algorytm 3 – Maciej Mucha

Pierwsze badania polegały na sprawdzenie zachowania algorytmu sekwencyjnego, czyli algorytmu równoległego uruchamianego na 1 wątku. Test polegał na uruchomieniu algorytmu dla 4 różnych rozmiarów danych wejściowych i ustaleniu wpływu liczby kubełków na czas sortowania.



Wykres 14. Czas wykonania w zależności od liczby kubełków, rozdzielone, n=1e6



Wykres 15. Czas wykonania w zależności od liczby kubełków, n=1e6

Wykres 15. Czas wykonania w zależności od liczby kubełków, n=1e6 osiąga minimum jeśli liczba kubełków stanowi 10%-15% liczby elementów w sortowanej tablicy. W dalszych testach liczba kubełków jest równa 15% z liczby elementów w tablicy.

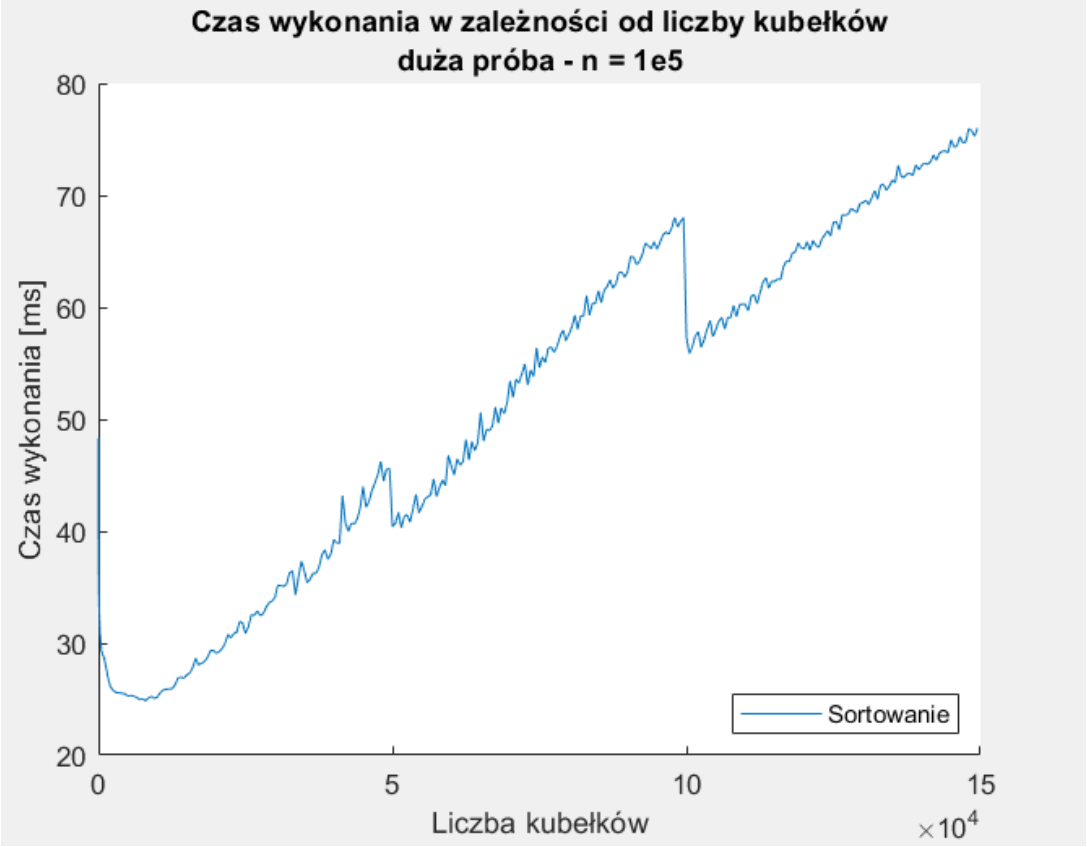
Analizując Wykres 14. Czas wykonania w zależności od liczby kubełków, rozdzielone, n=1e6 można wyciągnąć wnioski na temat poszczególnych etapów. Algorytm ma części, których złożoność jest bezpośrednio zależna od liczby kubełków. Są to etapy łączenia kubełków oraz etap finalizowania sortowania poprzez kopiowanie posortowanych danych z kubełków do tablicy. Etap przypisywania danych do odpowiednich kubełków również trwa dłużej wraz ze wzrostem liczby kubełków, co jest spowodowane większą liczbą miejsc w pamięci, gdzie dane są umieszczane, pomimo braku teoretycznej zależności.

Złożoności poszczególnych etapów. n – liczba danych, k – liczba kubełków, t – liczba wątków, S(x) – złożoność sortowania x elementów

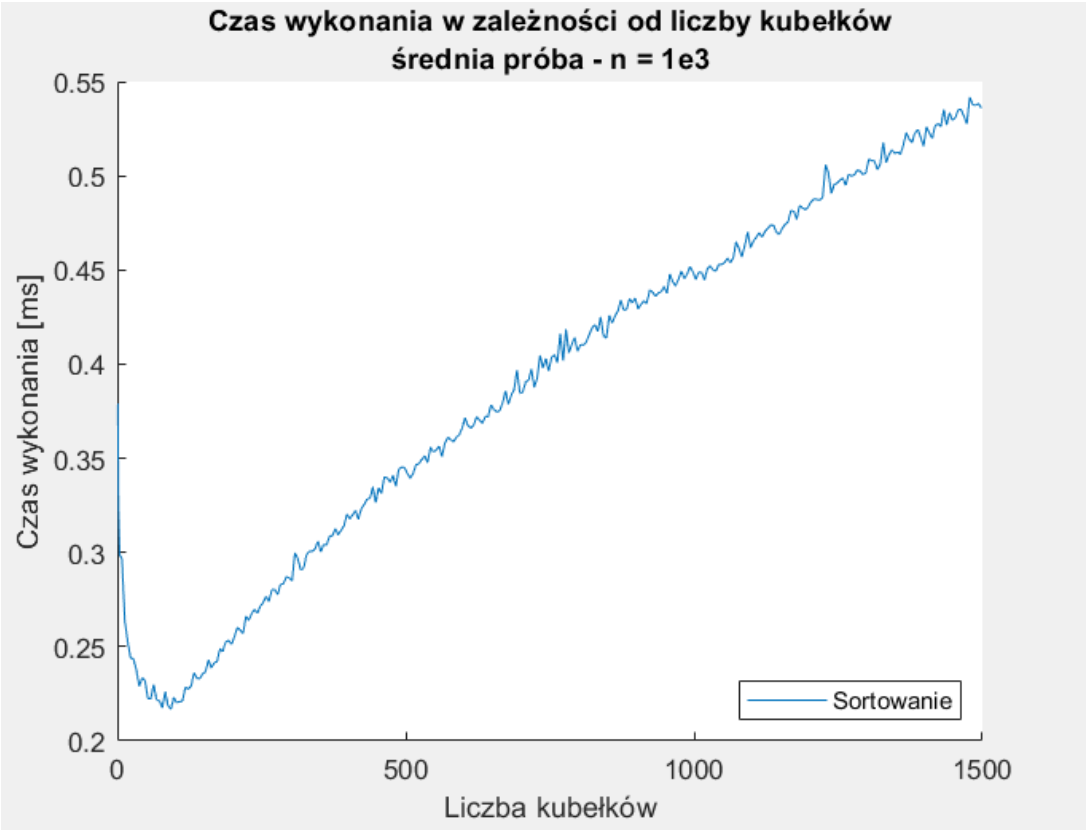
1. Przypisanie danych do kubełków – O(n)
2. Łączenie kubełków – O(k*t*n/k) = O(n*t)
3. Sortowanie kubełków – O(k*S(n/k))
4. Tworzenie wyniku – O(k+n) = O(n)

Złożoność całego sortowania -
O(n*t + k*S(n/k)) = [dla k = 0.15n] = O(n*(t+S(c))) = [gdzie c to stała] = O(n*t)

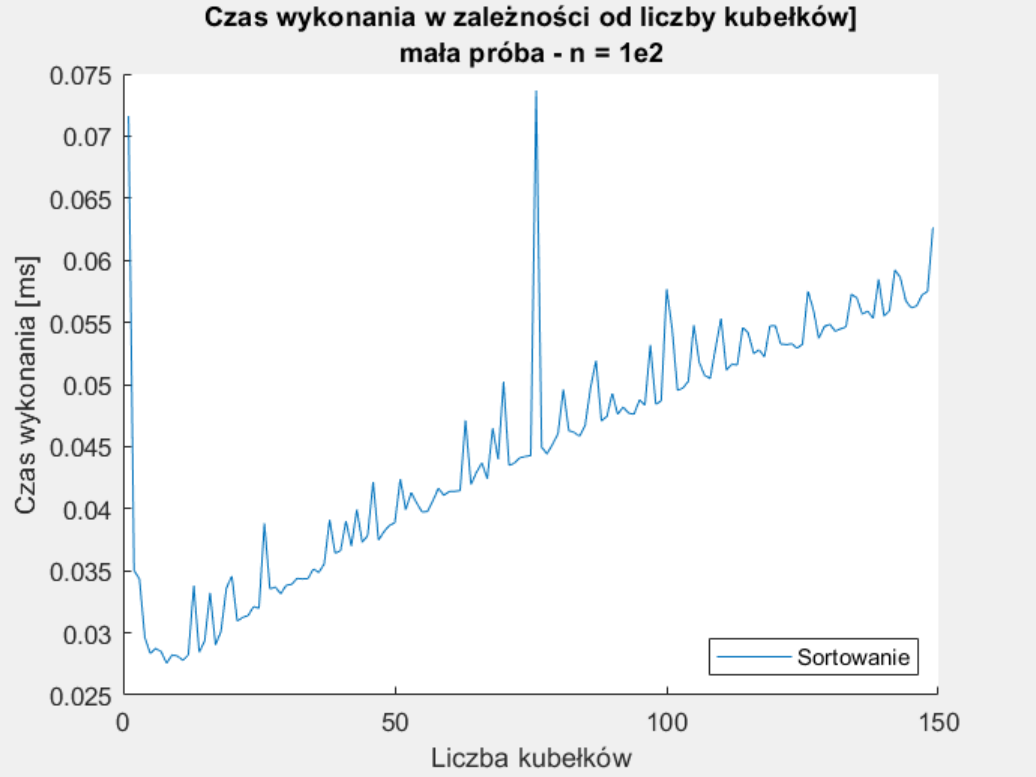
Poniżej znajdują się testy z tablicami o innych rozmiarach.



Wykres 16. Czas wykonania w zależności od liczby kubełków, n=1e5



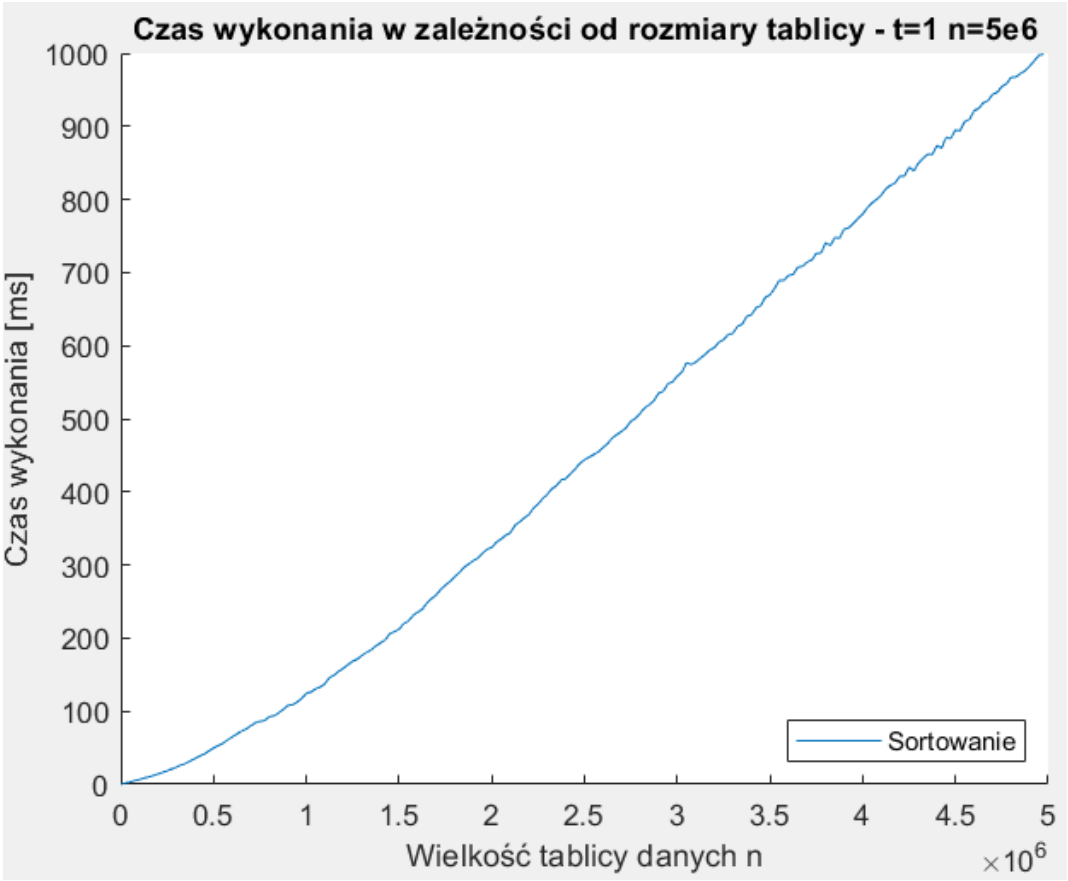
Wykres 17. Czas wykonania w zależności od liczby kubełków, n=1e3



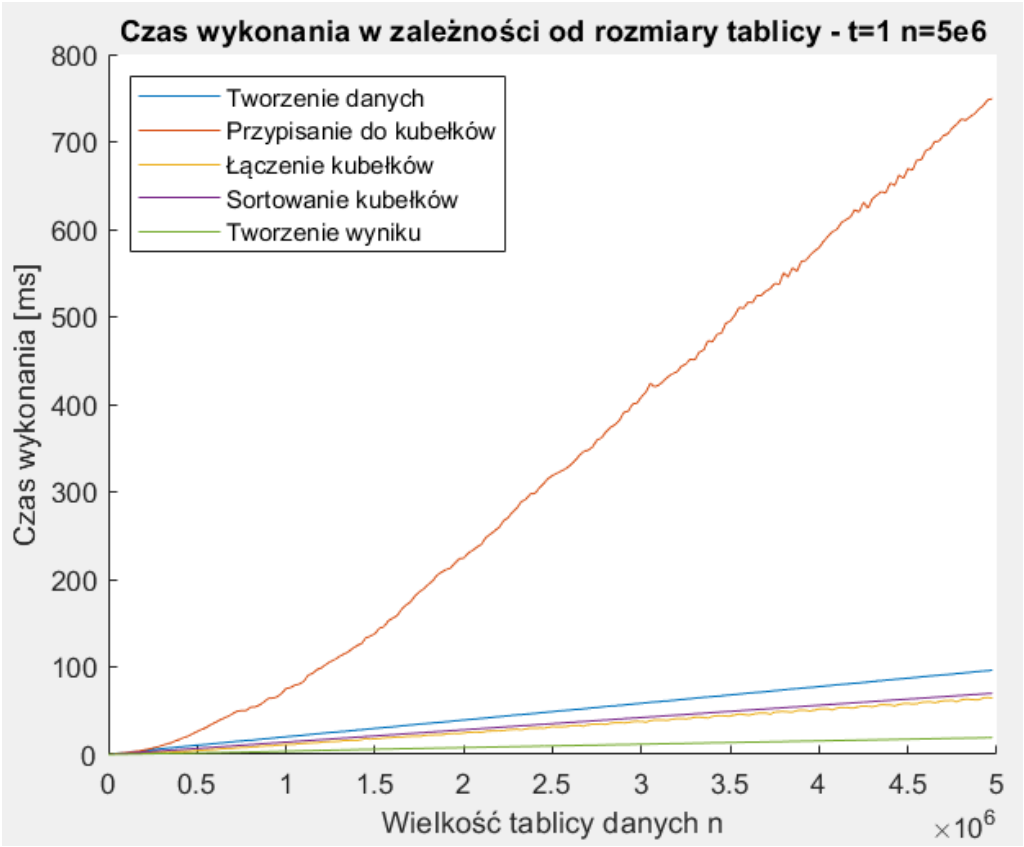
Wykres 18. Czas wykonania w zależności od liczby kubełków, n=1e2

Wykres 18. Czas wykonania w zależności od liczby kubełków, n=1e2 jest dla bardzo małej próby, która nie jest rzeczywistym przykładem zastosowania algorytmu równoległego. Zarówno Wykres 16. Czas wykonania w zależności od liczby kubełków, n=1e5jak i Wykres 17. Czas wykonania w zależności od liczby kubełków, n=1e3 mają minimum dla liczby kubełków równej 10%-15% z liczby elementów w tablicy.

Poniższe testy uruchamiane były na 1 wątku, liczbą kubełków równą 15% liczby elementów w tablicy. Wielkość tablicy rosła od 1 do 5e6. Testy powtórzone były wielokrotnie, wynik uśredniono.



Wykres 19. Algorytm sekwencyjny, czas w zależności od rozmiaru tablicy



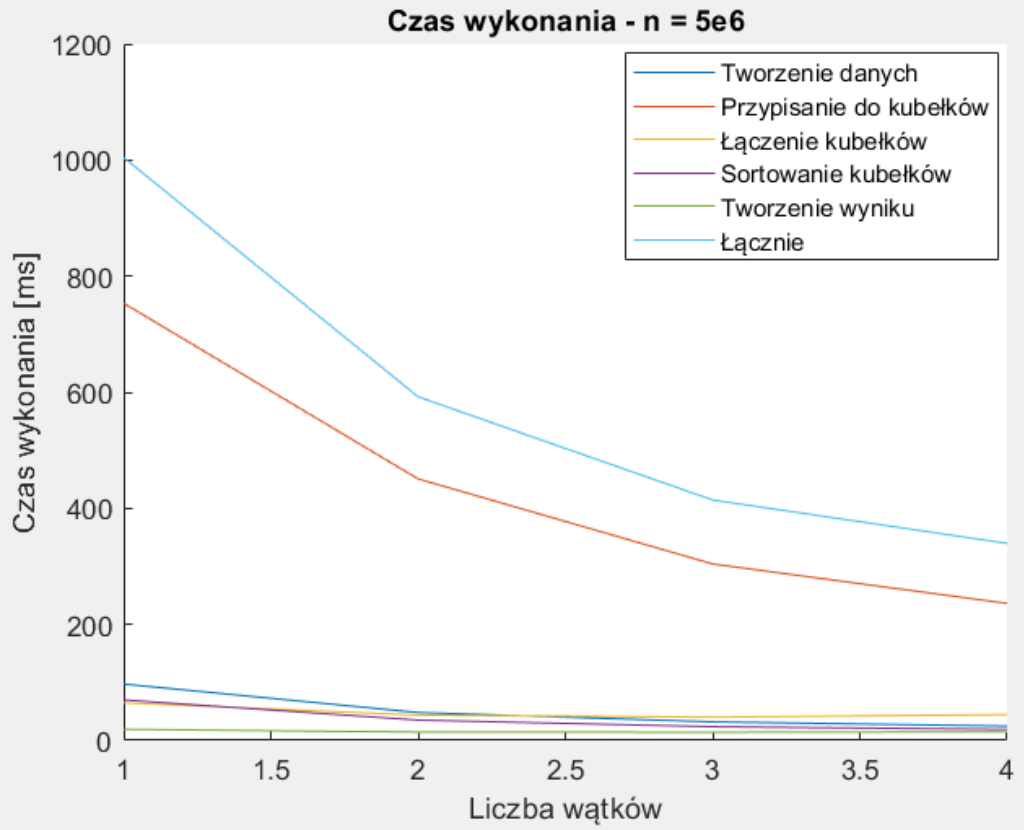
Wykres 20. Algorytm sekwencyjny, czas w zależności od rozmiaru tablicy, rozdzielone

Wykres 19. Algorytm sekwencyjny, czas w zależności od rozmiaru tablicy przedstawia jak zmienia się czas sortowania w zależności od liczby elementów w tablicy. Wykres 20. Algorytm sekwencyjny, czas w zależności od rozmiaru tablicy, rozdzielonezawiera te same dane, tylko rozdzielone na poszczególne etapy. Widać z nich, że etap przypisania do kubełków zajmuje zdecydowanie najwięcej czasu i najszybciej rośnie. Ten etap dla wielkości tablicy ponad 1e6 oraz pozostałe etapy dla wszystkich rozmiarów tablicy rosną liniowo.

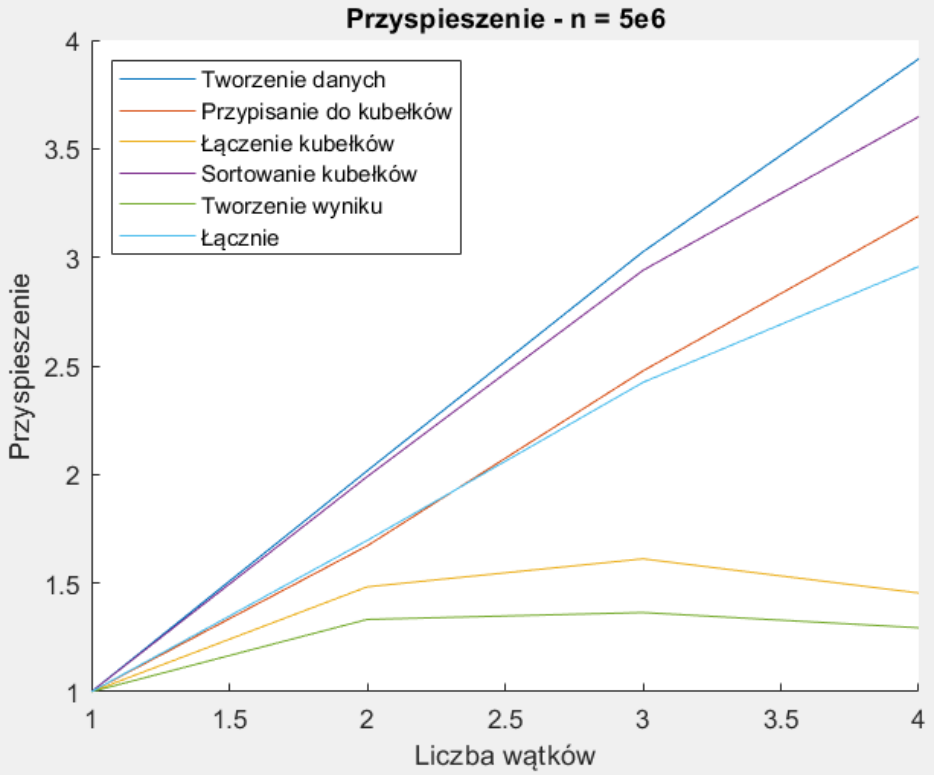
Badanie przyśpieszenia

Algorytm 3 – Maciej Mucha

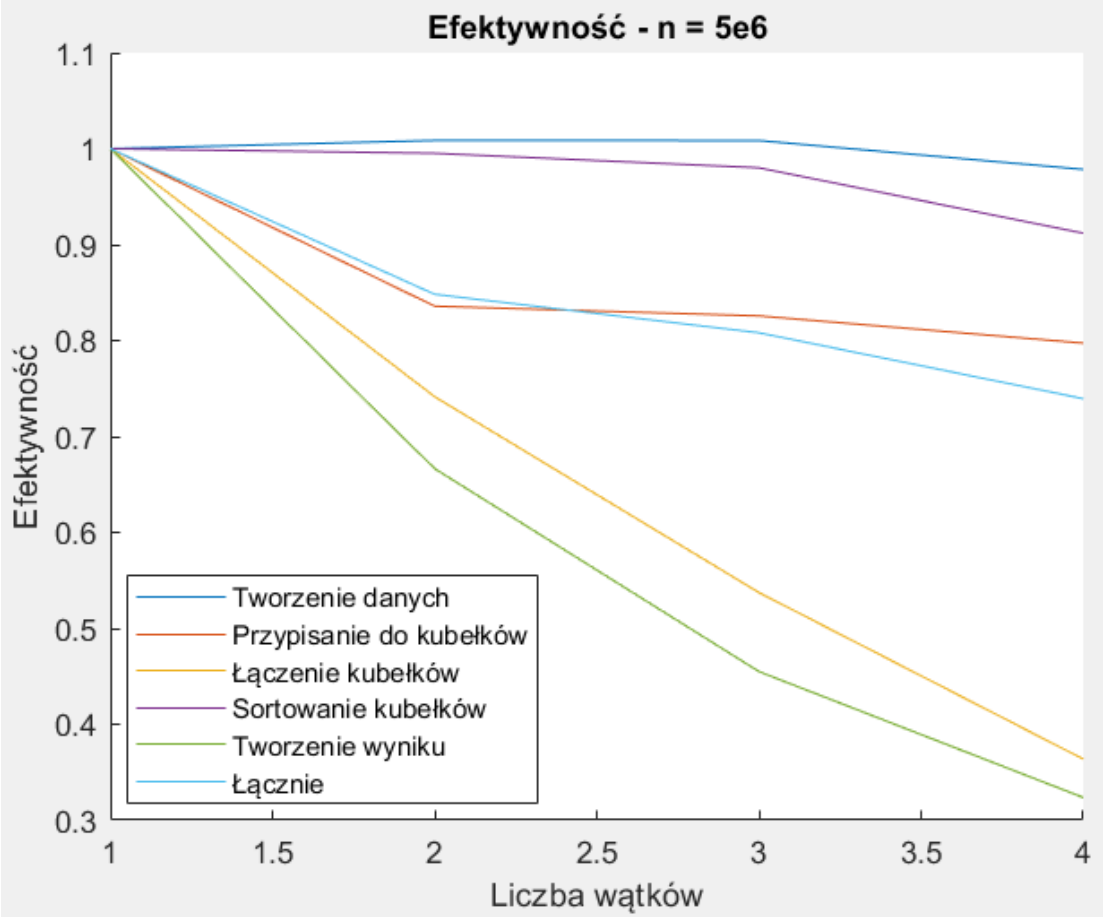
Test z tablicą o rozmiarze 5e6, wykonywana na v-klastrze. Test został wykonany 100 razy, wyniki uśredniono. Liczba kubełków stanowiła 15% czyli 750000.



Wykres 21. Czas wykonania sortowania z użyciem wielu wątków - n=5e6



Wykres 22. Przyspieszenie wykonania sortowania z użyciem wielu wątków - n=5e6



Wykres 23. Efektywność wykonania sortowania z użyciem wielu wątków - n=5e6

Czas widoczny na wykresie Wykres 21. Czas wykonania sortowania z użyciem wielu wątków - n=5e6posłużył do wyznaczenia przyspieszenia widocznego na wykresie Wykres 22. Przyspieszenie wykonania sortowania z użyciem wielu wątków - n=5e6, które zostało użyte do wyznaczenia efektywności widocznej na Wykres 23. Efektywność wykonania sortowania z użyciem wielu wątków - n=5e6 Zastosowano prawo Amdahla.

Efektywność całego sortowania osiągnięta wartość 74% dla 4 wątków.

Praca algorytmu równoległego jest to iloczyn „czasu działania" algorytmu przez liczbę procesorów, jest określona liczbą obliczeń, jakie wykonuje sekwencyjna maszyna RAM symulując działanie algorytmu równoległego. Praca i złożoność obliczeniowa każdego etapu została określona osobno

Tworzenie danych

```
double DoubleRand() {
    static thread_local std::mt19937 generator;
    std::uniform_real_distribution<double> distribution(0.0,1.0);
    return distribution(generator);
}
(…)
#pragma omp parallel for schedule(runtime) firstprivate(Array)
for (int s = 0; s < size; s++)
{
    Array[s] = DoubleRand();
}
```

Operacja łatwa do zrównoleglenia. Wysoka efektywność dla 4 wątków równa 98%. Złożoność O(n). Praca algorytmu równoległego O(n).

1. Przypisanie do kubełków

```
#pragma omp parallel firstprivate(Buckets, Array, bucketCount)
{
    const int myThreadNum = omp_get_thread_num();
    std::vector<double>* myBucket = Buckets[myThreadNum];
    #pragma omp for schedule(runtime)
    for (int s = 0; s < size; s++)
    {
        const double val = Array[s];
        const int valBucket = val * bucketCount;
        myBucket[valBucket].push_back(val);
    }
   }
```

Etap zajmujący najwięcej czasu ze wszystkich. Efektywność dla 4 wątków równa 80%. Jak pokazywały poprzednie badania w wersji sekwencyjnej, wraz ze wzrostem liczby kubełków, czas wykonywania tego etapu wzrastał. Wzrost liczby wątków również zwiększa liczbę kubełków, ponieważ każdy wątek ma swoje kubełki na wyłączność. Spowodowało to że efektywność nieco zmalała od ideału, jednak nadal utrzymuje się na wysokim poziomie. Złożoność O(n). Praca O(n)

2. Łączenie kubełków

```
#pragma omp parallel for firstprivate(threadNum, SingleBuckets, Buckets)
schedule(runtime)
for (int b = 0; b < bucketCount; b++)
{
    std::vector<double>* mySingleBucket = &SingleBuckets[b];
    for (int t = 0; t < threadNum; t++)
```

```
    {
        std::vector<double>* mergeBucket = &Buckets[t][b];
        mySingleBucket->insert(mySingleBucket->end(), mergeBucket-
>begin(), mergeBucket->end());
    }
}
```

Efektywność równa 36% dla 4 wątków, co stanowiło wynik lepszy jedynie od ostatniego etapu. Złożoność tego etapu wynosi O(n*t) i jest bezpośrednio zależne od liczby wątków, co wyjaśnia tak słaby wynik. Jest to etap najtrudniejszy do zrównoleglenia.
Złożoność O(n*t). Praca O(n*t).

3. Sortowanie kubełków

```
#pragma omp parallel for schedule(runtime) firstprivate(SingleBuckets,
bucketSizes)
for (int b = 0; b < bucketCount; b++)
{
    std::vector<double>* mySingleBucket = &SingleBuckets[b];
    std::sort(mySingleBucket->begin(), mySingleBucket->end());
    bucketSizes->at(b) = (int)mySingleBucket->size();
}
```

Efektywność dla 4 wątków równa 91%, a dla 3 wątków równa 98%. Etap łatwy do zrównoleglenia, składa się z wielu, podobnych czasowo operacji wykonywanych na każdym elemencie tablicy (kubełku), co wyjaśnia wysoką efektywność. Dobranie odpowiedniej ilości kubełków zapewniło podobny rozmiar każdego z nich.
Złożoność O(k*S(n/k)) =[dla k=0.15*n]= O(n) (sortowanie w czasie stałym). Praca O(n)

4. Tworzenie wyniku

```
#pragma omp parallel for schedule(runtime) firstprivate(SingleBuckets,
bucketSizes, Array)
for (int b = 0; b < bucketCount; b++)
{
    int insertId = b == 0 ? 0 : bucketSizes->at(b - 1);
    for (int i = 0; i < SingleBuckets[b].size(); i++)
    {
        Array[insertId + i] = SingleBuckets[b][i];
    }
}
```

Wpisywanie danych do tablicy wejściowej. Zawiera część sekwencyjną, a w części równoległej, Podobnie jak w punktach 2 i 3, liczba kubełków, z których musimy przepisać wzrasta wraz ze wzrostem liczby wątków, co nie zwiększa liczby kopiowań, tylko zwiększa liczbę źródeł w pamięci.
Złożoność O(n). Praca O(n).

5. Łącznie
   Dla 4 wątków efektywność całego algorytmu wyniosła 74%. Łączenie kubełków to etap najbardziej wpływający na końcową efektywność, ponieważ trwa najdłużej. Złożoność całego sortowania wynosi O(n * t). Praca algorytmu równoległego jest rzędu O(n * t).

Można uznać, że t – liczba wątków jest stałą, zależną od używanego sprzętu, a nie od wielkości tablicy z danymi. Oznacza to, że złożoność jest rządu O(n), oraz praca algorytmu równoległego jest również rzędu O(n). Najszybszy możliwy algorytm sekwencyjny miałby również złożoność O(n).

Pozwala to na stwierdzenie, że algorytm równoległego sortowania kubełkowego jest sekwencyjnie-efektywny w odniesieniu do algorytmu sekwencyjnego sortowania kubełkowego.

Podsumowanie

Algorytm 3 różni się od pozostałych dwóch całkowitym brakiem kolizji zarówno przy czytaniu jak i zapisywaniu danych pomiędzy wątkami. W każdym etapie każdy wątek operuje na rozłącznym zbiorze danych. Nie jest wymagana ochrona kubełków, brak locków oraz aktywnego czekania na zwolnienie zasobów. Jest to opłacone koniecznością dodatkowego etapu, opisywanego wcześniej jako łączenie kubełków. Ten dodatkowy etap osiągną słabą efektywność zrównoleglenia, ponieważ ma złożoność O(n*t), dodatkowy wątek zwiększy złożoność obliczeniową.