Badanie właściwości algorytmów równoległych w sortowaniu kubełkowym

autor: Wojciech Buś

1. Opis algorytmu

Algorytmem, który miałem zaimplementować oraz przetestować był algorytm nr. 2. Algorytm ten został zaimplementowany na podstawie opisu dostępnego na UPEL-u.

Jeśli chodzi o ochronę danych wspólnych to opiszę go może na poszczególnych częściach programu:

a) wypełnianie tablicy - tutaj dzięki omp parallel for nie musimy się martwić o ochronę danych wspólnych
b) wypełnianie kubełków- tutaj wątki posiadają wspólne kubełki. Pojawia się więc problem, który rozwiązałem poprzez użycie lock'ów.
c) sortowanie kubełków - wątki mają przydzielone konkretne kubełki do posortowania. Wszystko działa niezależnie.
d) przepisywanie posortowanych kubełków do tablicy początkowej - tutaj każdy wątek przepisuje wartości przypisanych mu kubełków do tablicy. Jako że każdy element ma z góry ustaloną pozycję umieszczenia to wystarczy, że policzymy offset dla każdego kubełka (czyli od jakiego indeksu tablicy należy zacząć umieszczać elementy kubełka), żeby nie musieć się martwić o ochronę danych.

Jeśli chodzi o złożoność obliczeniową algorytmu to wprowadźmy oznaczenia:
n - rozmiar tablicy
k - liczba kubełków
p - liczba wątków
Następnie policzmy złożoność poszczególnych części algorytmu:

a) wypełnianie tablicy - O(n / p). Każdy wątek ma pewną część tablicy do wypełnienia
b) wypełnianie kubełków - O(n / p). Każdy wątek ma inną część tablicy przypisać do kubełków. Zakładając, że locki nie mają żadnego narzutu otrzymujemy podaną złożoność
c) sortowanie kubełków - O(k * n / k * log(n / k) / p). Wykonujemy quicksort o złożoności O(n / k * log(n / k)) na k różnych kubełkach przez p różnych wątków
d) scalanie - O(k + n / p). Znalezienie offsetu kubełka ma złożoność O(k). Przepisanie kubełków do tablicy początkowej ma złożoność O(n / p) (każdy kubełek ma średnio rozmiar n / k, oraz każdy wątek ma do przepisania k / p kubełków. Iloczyn z tych dwóch wartości daje żądaną złożoność)

Całkowita TEORETYCZNA złożoność wynosi więc:
$O(n / p + n / p + k * n / k * \log(n / k) / p + k + n / p) = O(n / p + n \log(n / k) / p + k)$

W zasadzie każda część algorytmu może zostać w teorii dowolnie mocno zrównoleglona. Jedyna teoretyczna bariera (wyjątek) to znalezienie offsetu dla kubełka. Można więc powiedzieć, iż ten algorytm powinien się rewelacyjnie skalować.

2. Środowisko

Testy przeprowadziłem na bastionie na vnode-01, ponieważ udostępnia on 4 rdzenie, które są konieczne do poprawnego przeprowadzenia badania.

3. Sposób przeprowadzenia doświadczenia

Programy uruchomiłem 5-krotnie. Następnie odrzuciłem pomiary, które różniły się znacząco od reszty i policzyłem średnią.

4. Wyniki badań i wnioski

a) poprawność danych wejściowych

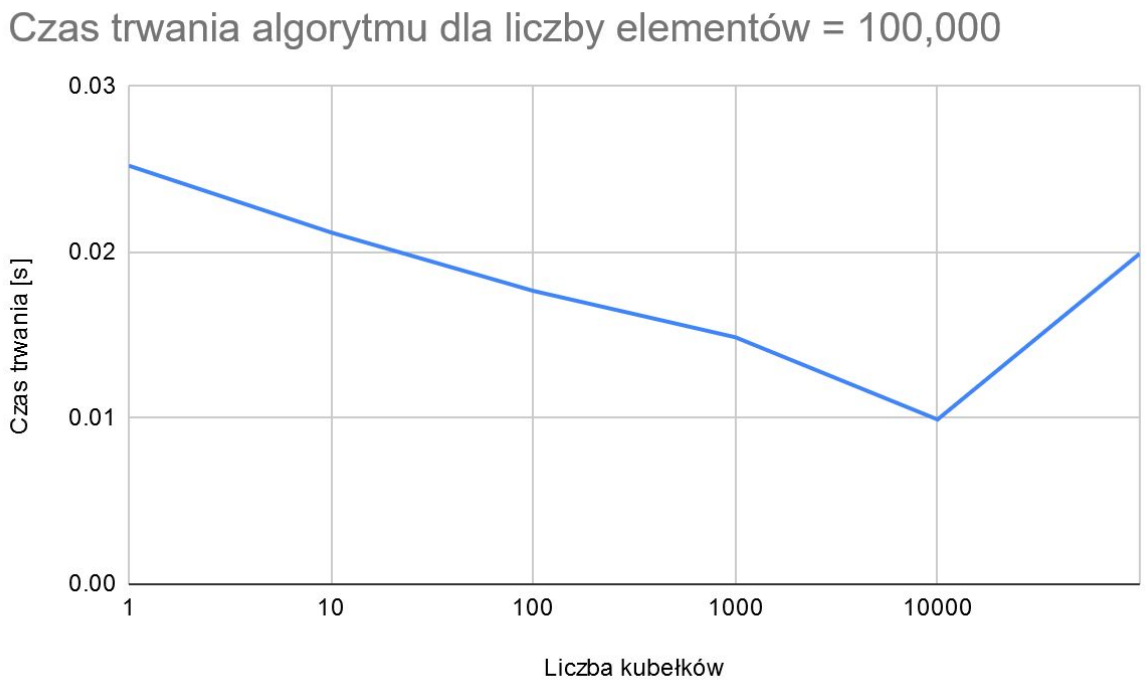
Sortowanie kubełkowe bardzo lubi dane, które są równomiernie rozłożone. Oznaczać to będzie, iż kubełki które algorytm ma do posortowania będą zbliżonej do siebie wielkości. Mój algorytm korzysta z funkcji losującej erand48(), której opis na “Linux Man Page” brzmi: *“generate uniformly distributed pseudo-random numbers”*
Przeprowadziłem 5 różnych testów i sprawdziłem ile elementów liczy najmniejszy oraz największy kubełek:

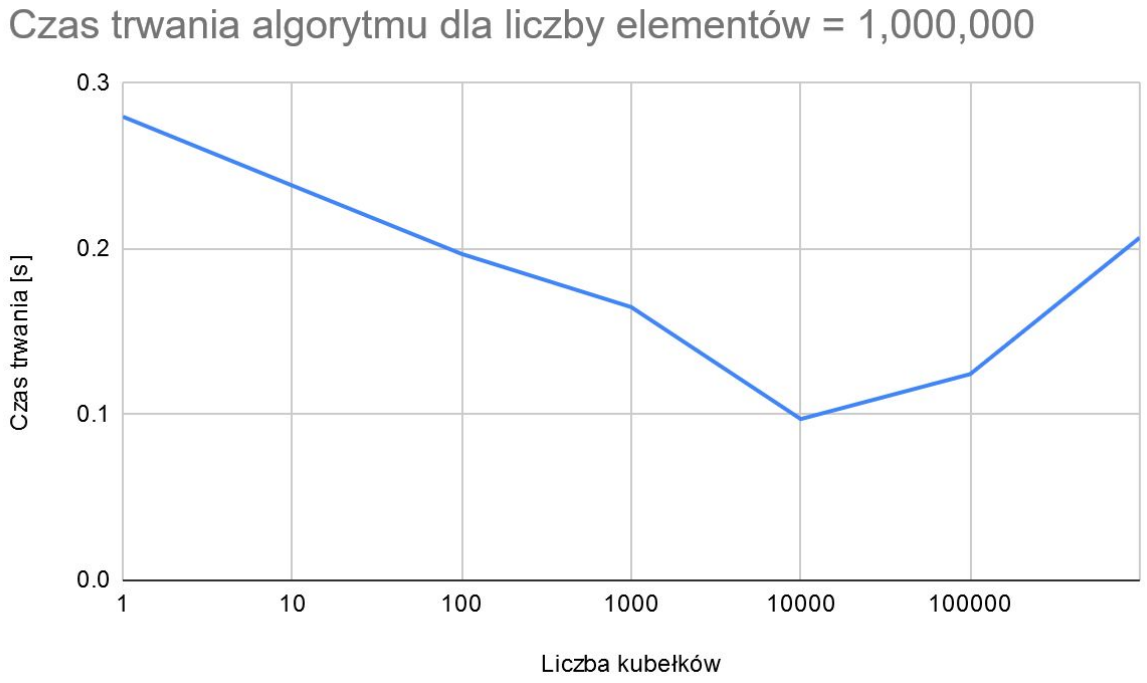
Liczba elementów	Liczba kubełków	średni kubełek	min kubełek	max kubełek
50000	100	500	434	555
250000	150	1666	1563	1792
1000000	200	5000	4827	5188
5000000	1000	5000	4781	5230
1000000	2000	5000	4760	5225

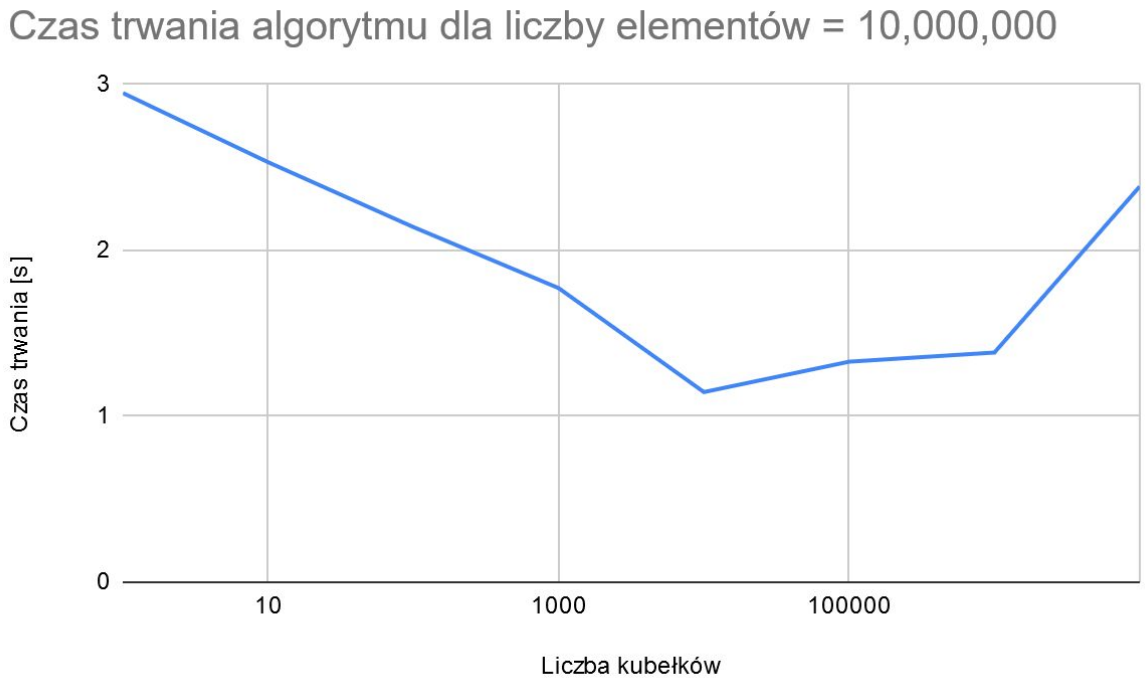
Jak można zauważyć, rozmiary kubełków są do siebie całkiem zbliżone. Nie są równe, ale są wystarczająco bliskie sobie.

b) zachowanie algorytmu dla sortowania sekwencyjnego

Testy zachowania przeprowadziłem dla 3 różnych wielkości tablicy początkowej. Tablica zawierała elementy od 0 do 9999. Wyniki prezentują się następująco:



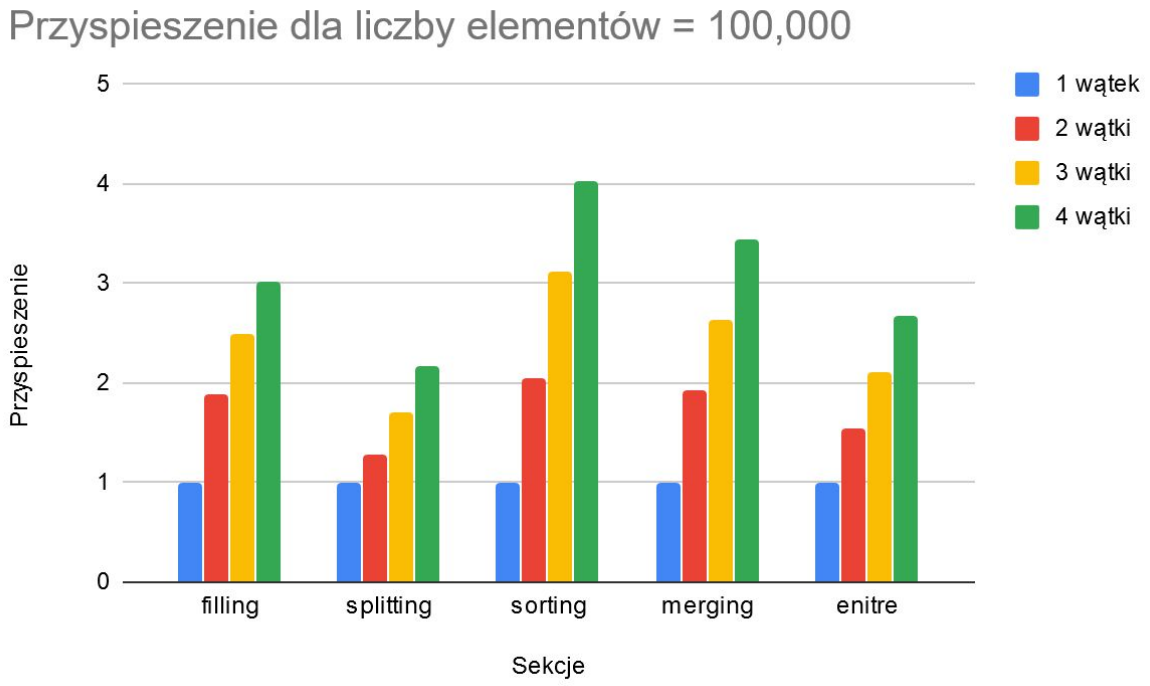


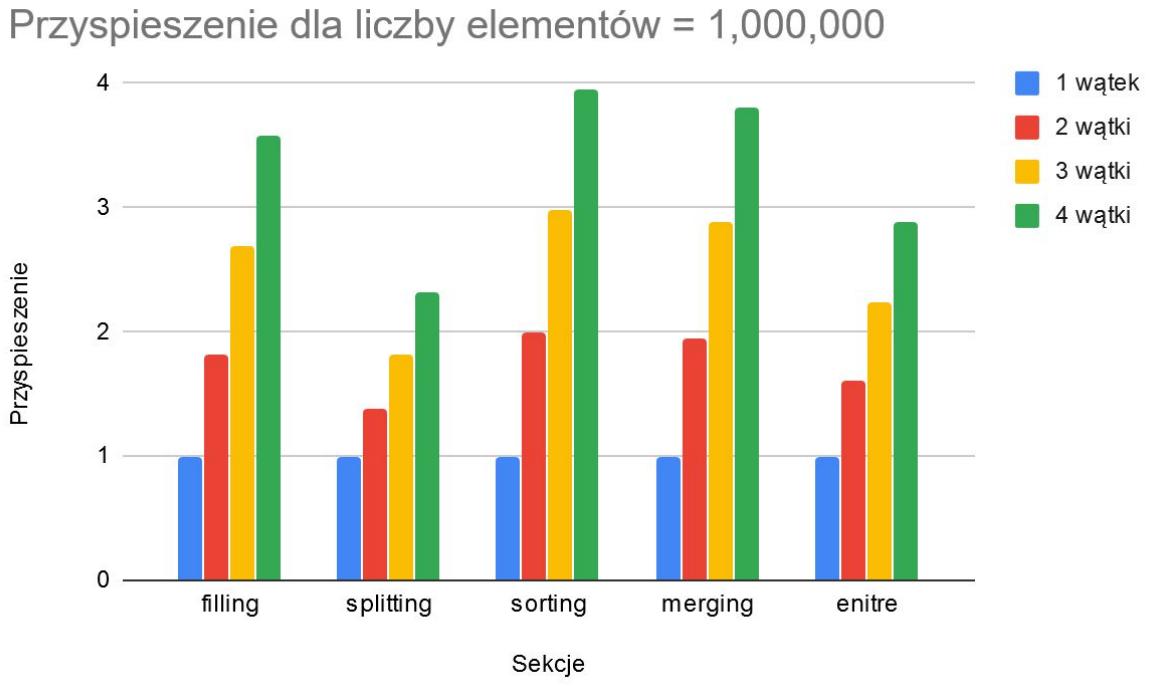


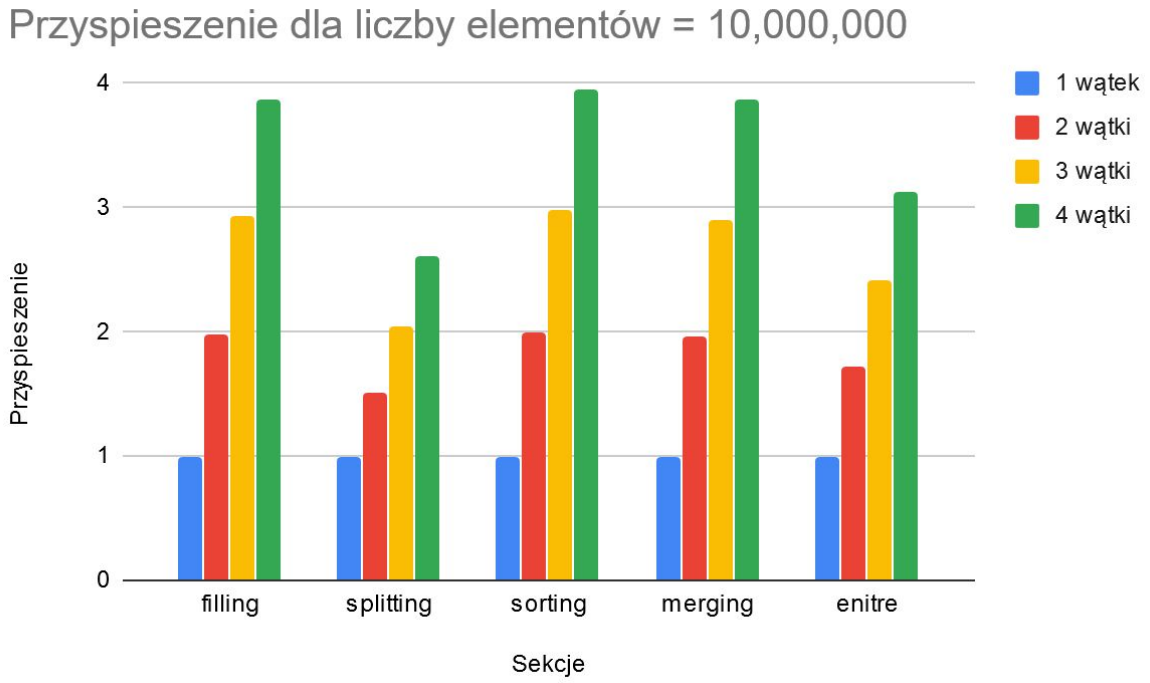
Jak widzimy, algorytm sortuje najszybciej, gdy posiada on liczbę kubełków równą liczbie możliwych do wylosowania liczb. Oczywiście użycie większej liczby kubełków nie ma jak widać sensu, gdyż tworzy to kubełki, które będą po prostu puste (wsadzenie 10,000 różnych elementów do 100,000 kubełków aby żaden nie był pusty nie jest możliwe).
Jaki ma to wpływ na rząd złożoności tego algorytmu? Maksymalne przyspieszenie jakie uzyskaliśmy to około 3, więc około pół rzędu. Możliwe, że dla większych danych udałoby się osiągnąć przyspieszenie na poziomie jednego rzędu.

c) przyspieszenie

Badanie przeprowadziłem dla liczby kubełków równej 10,000, oraz rozmiarów tablicy analogicznych jak w podpunkcie powyżej. Wyniki prezentują się następująco:





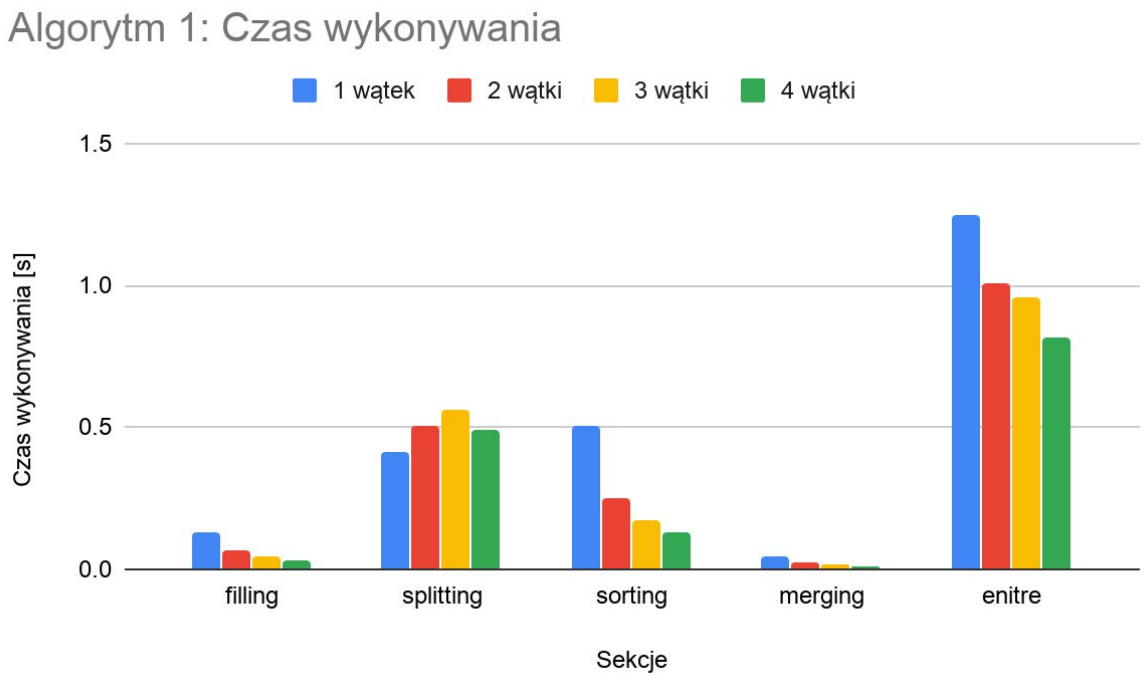


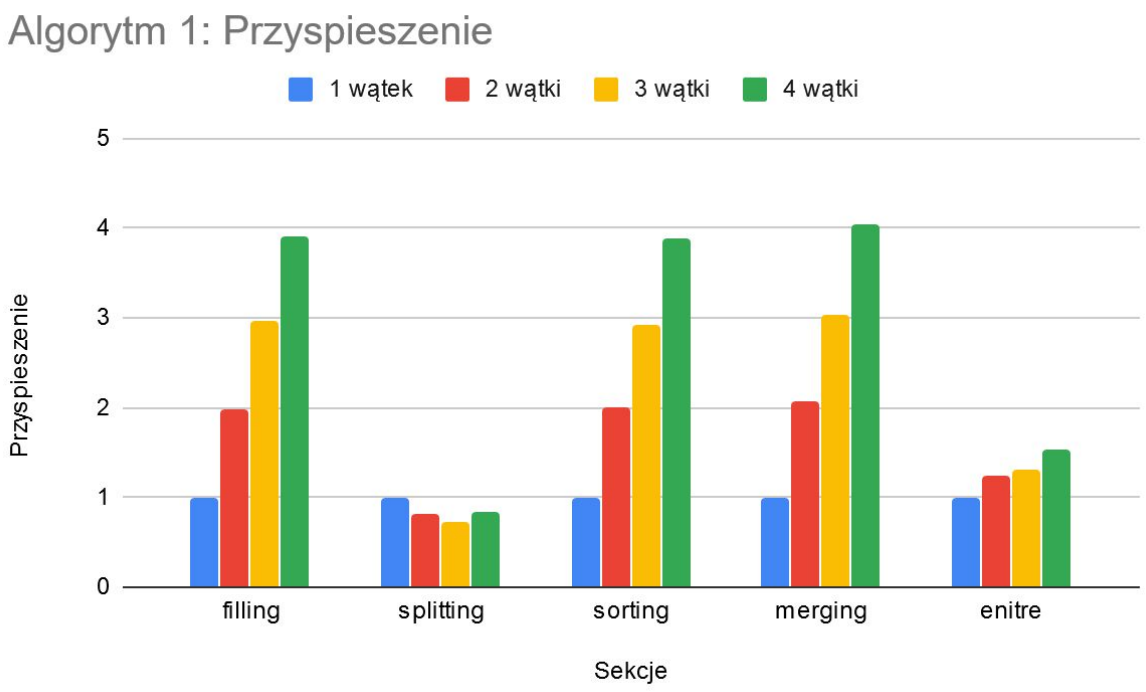
Jak widzimy, filling, sorting oraz merging są równolegle-efektywne. Widać to szczególnie przy największej tablicy - można przypuszczać, że gdyby była jeszcze większa to wyniki byłyby jeszcze lepsze.
Inaczej sprawa ma się ze splittingiem - przez dostęp do wspólnych danych byłem zmuszony do użycia locków, które niestety mają spory narzut czasowy (jak widać na załączonych wykresach). Dobra wiadomość jest taka, iż przy wzroście rozmiaru tablicy, przyspieszenie dla splittingu wydaje się rosnąć.
Jeśli chodzi o prawa Amdahla oraz Gustafsona - w naszym programie w teorii wszystkie części są równolegle-efektywne (może oprócz wyliczania offsetu przy mergowaniu, ale jest to pomijalnie mała część uzależniona od liczby bucketów). Oznacza to, iż w teorii nie mamy tutaj żadnych części sekwencyjnych, więc jesteśmy w stanie przyspieszać nasz algorytm w nieskończoność (powtarzam - w TEORII).

d) porównanie

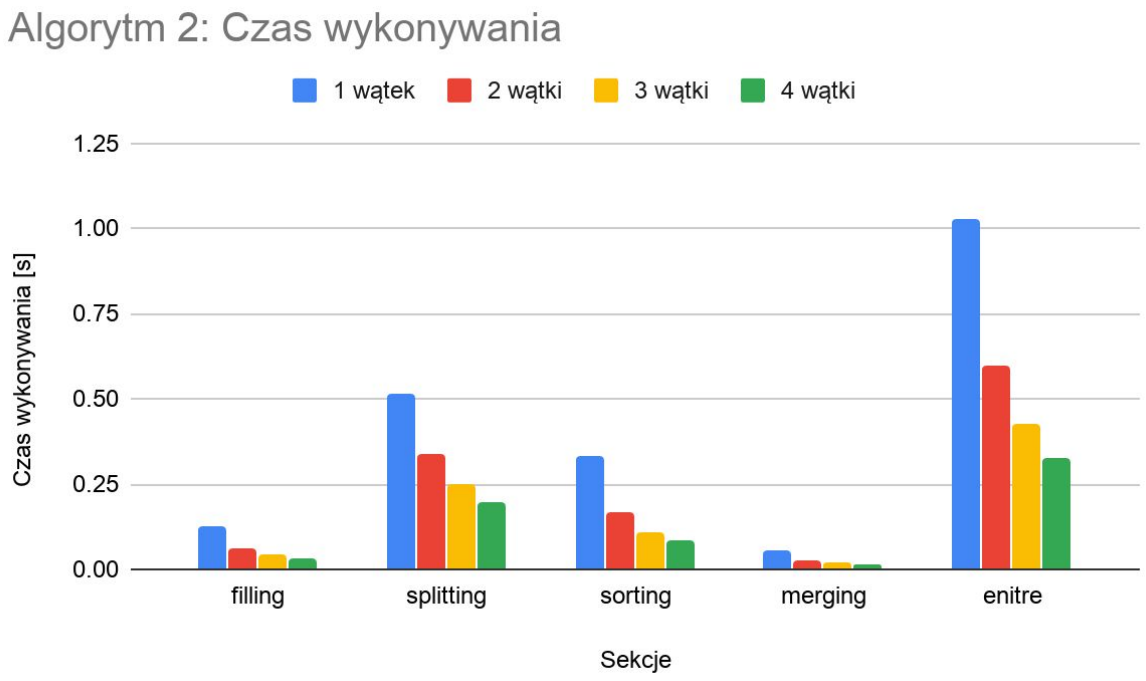
Porównajmy działanie naszych algorytmów dla tablicy o wielkości 10,000,000:

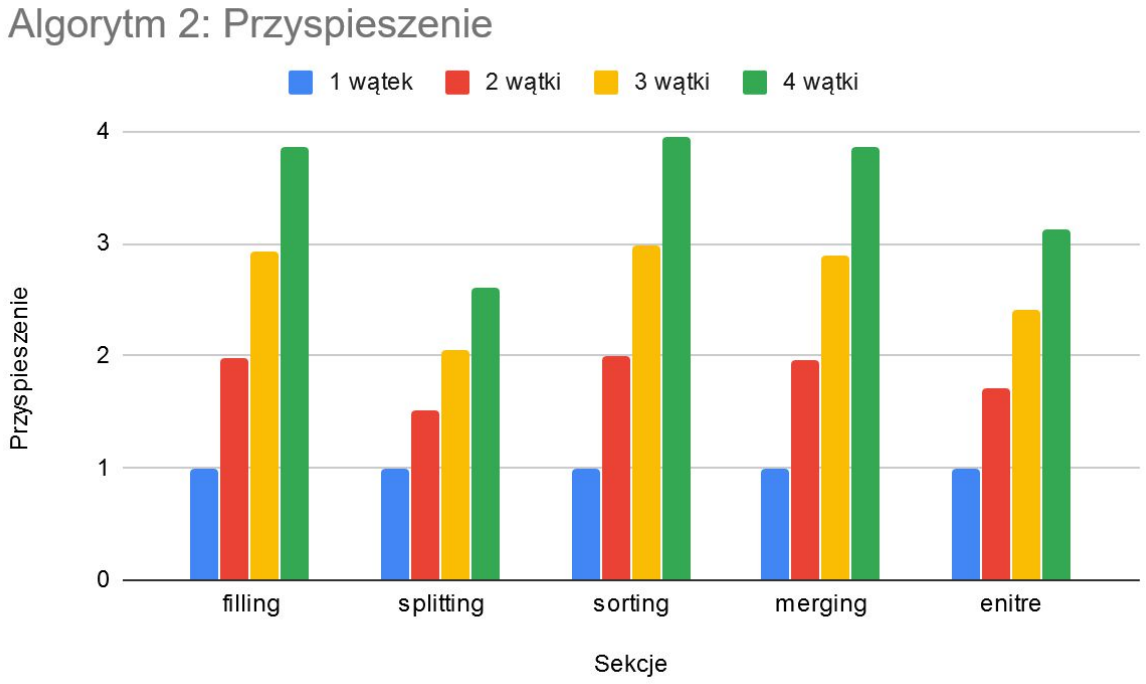
Algorytm 1 (autor - Paweł Mendroch):



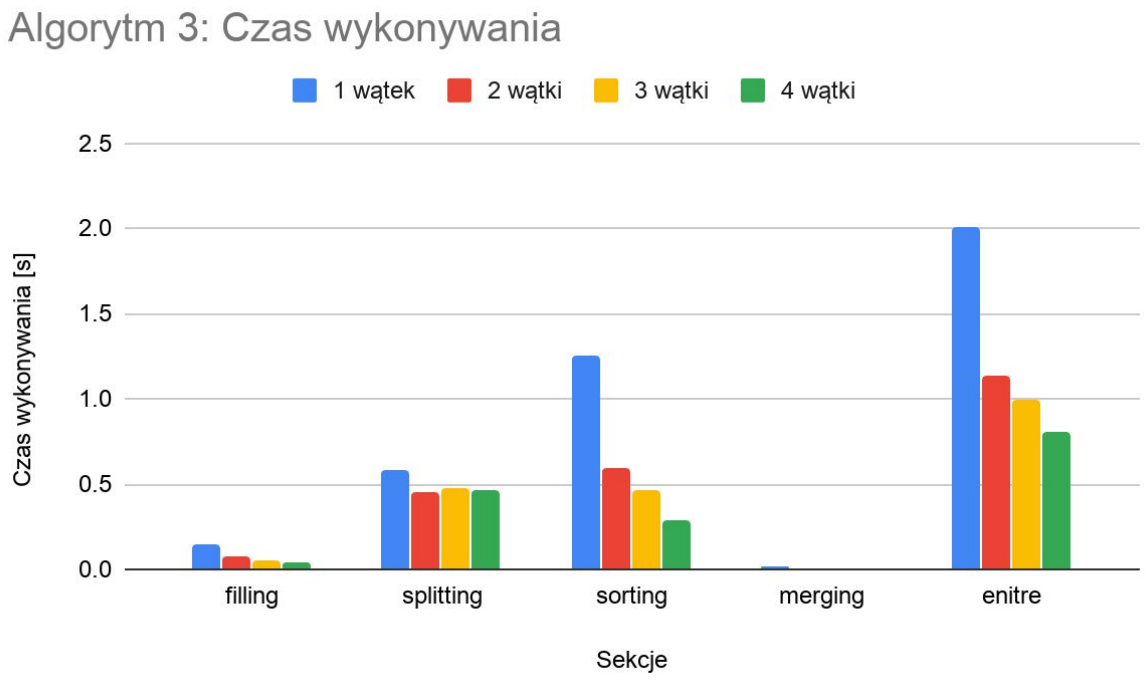


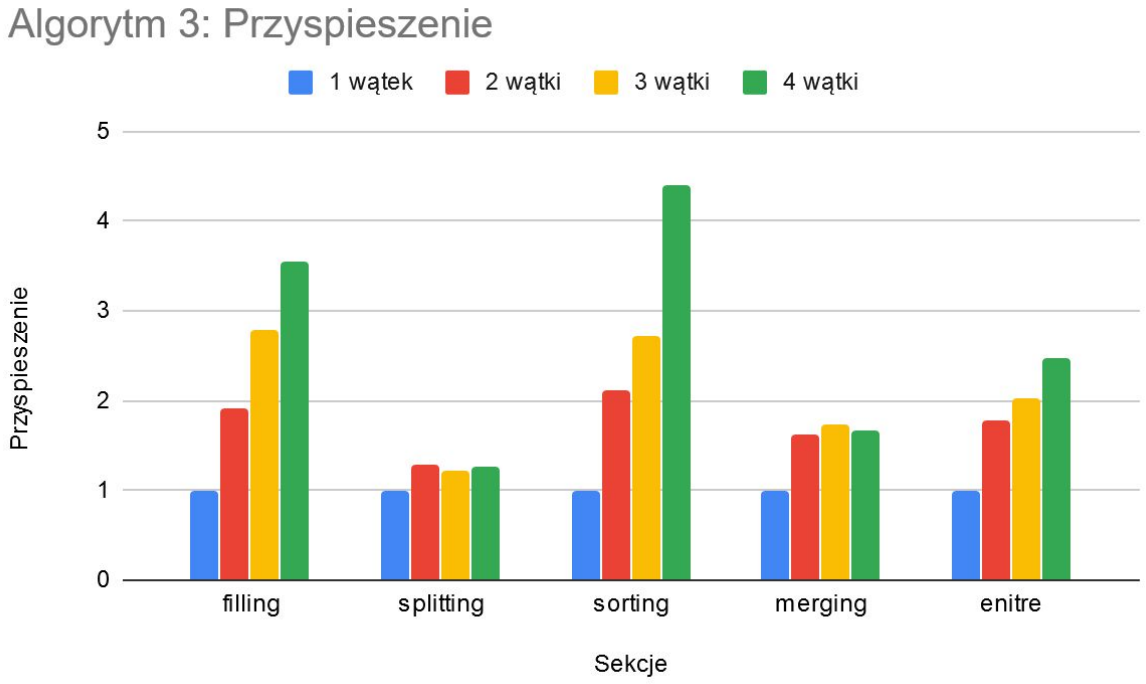
Algorytm 2 (autor - Wojciech Buś (ja)):





Algorytm 3 (autor - Bartłomiej Pluta):





Algorytm 1 - wszystkie części się ładnie skalują oprócz splitowania. Splitowanie nie skaluje się ponieważ każdy wątek czyta całą tablicę. Przez to cały algorytm na tym cierpi.

Algorytm 2 - wszystkie części się skalują. Splitowanie trochę mniej (użycie locków - sekcja krytyczna).

Algorytm 3 - tutaj znowu powtórka z rozrywki z algorytmu 1 - konkatenacja kubełków poszczególnych wątków jest trudna do optymalnego zrównoleglenia, przez co skalowalność całego algorytmu na tym cierpi.

Jak widzimy, najlepiej poradził sobie algorytm nr. 2. Będzie się on najlepiej skalował. Dlaczego? Ponieważ wszystkie jego części dobrze się skalują, gdy dodajemy nowe wątki (pozostałe algorytmy mają problem ze splitowaniem). Czyli jak widzimy - oczekiwania z punktu pierwszego *“ten algorytm powinien się rewelacyjnie skalować”* sprawdziły się! :D