Wyniki

Jakub Kowalczyk

Listopad 2023

1 Wersja jednowątkowa

Wersja jednowątkowa to właściwe typowy *quicksort* z dwoma drobnymi modyfikacjami:

1. Metoda *partition* bierze losowy *pivot* z danego przedziału i dzieli tablicę na trzy części: $[a_i < pivot \parallel a_i = pivot \parallel a_i > pivot]$. Dzięki temu w rekurencyjnch wywołaniach nie musimy uwzględniać elementów równych *pivotowi*. Wystarczy rozważyć 1. część i 3. część.

2. Jeśli rozważany przedział jest już dosyć mały ($length < 20$) wykorzystywany jest algorytm *SelectionSort*. Dzięki temu ucinamy wywołania rekurencyjne dla bardzo małych przedziałów.

2 wersja std::thread

W tej wersji zdecydowałem się zrównoleglić algorytm wykonując najwyższe wywołania rekurencyjne w oddzielnych wątkach. Określamy w argumencie wiersza polecenia maksymalną głębokość g takiego rozdzielania zadań na wątki i tak np:

1. $g = 1$: po pierwszym podzieleniu tablicy lewa strona zostanie posortowana w jednym wątku, druga strona w drugim.

2. $g = 2$: tutaj powstanie nam łącznie 6 wątków - najpierw dwa przy pierwszym podziale, następnie dwa przy podziale lewej części, dwa przy podziale prawej części.

Liczba aktywnych wątków równolegle sortujących swoje części tablicy to 2^g.

3 wersja openmp

1. Zaczynam od stworzenia regionu równoległego za pomocą #pragma omp parallel. Liczba dostępnych wątków jest znowu przekazywana poprzez argument wiersza polecenia.

2. Następnie w tym regionie za pomocą #pragma omp single wywołuję pierwsze wywołanie algorytmu (dla całej tablicy) tylko w jednym wątku.

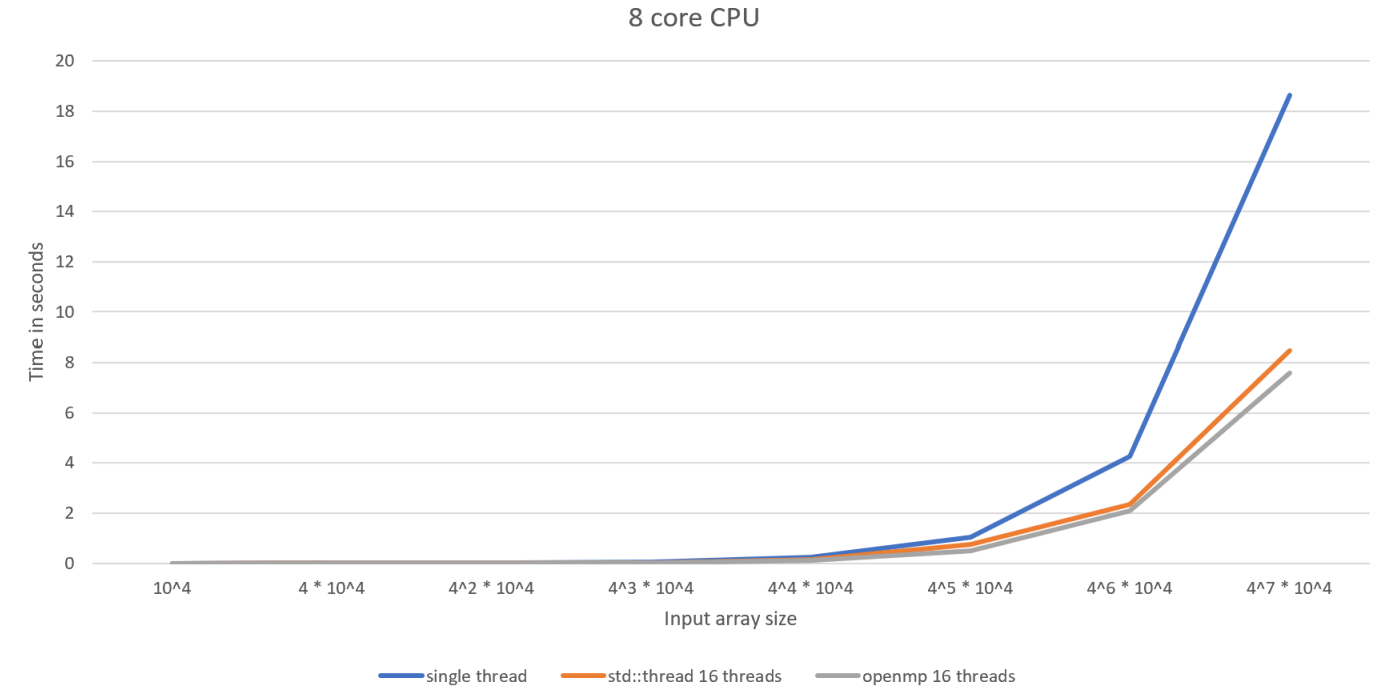
3. Dla każdego wywołania rekurencyjnego większego od pewnego minimalnego rozmiaru tworzę osobny task za pomocą #pragma omp task. Tasków nie tworzymy dla przedziałów o rozmiarze mniejszym niż 1500 elementów (najlepszy z kilku testowanych wariantów), ponieważ tworzenie tasków wiąże się z pewnym narzutem, który dla małych przedziałów po prostu się nie opłaca. Taki task ląduje w puli tasków i może być wykonany przez dowolny nieaktywny wątek. Dzięki temu to rozwiązanie działa szybciej niż poprzednie, ponieważ wykorzystuje wszystkie dostepne wątki.

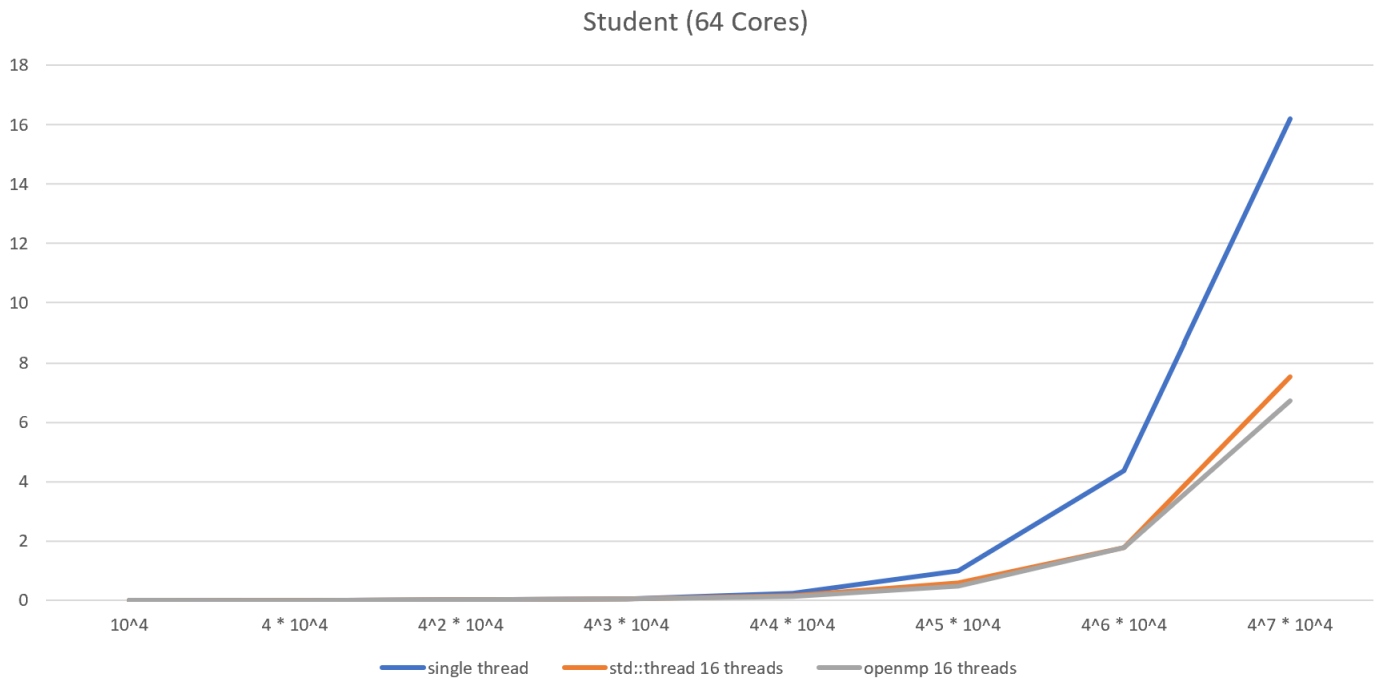
4 Kompilacja i testowanie

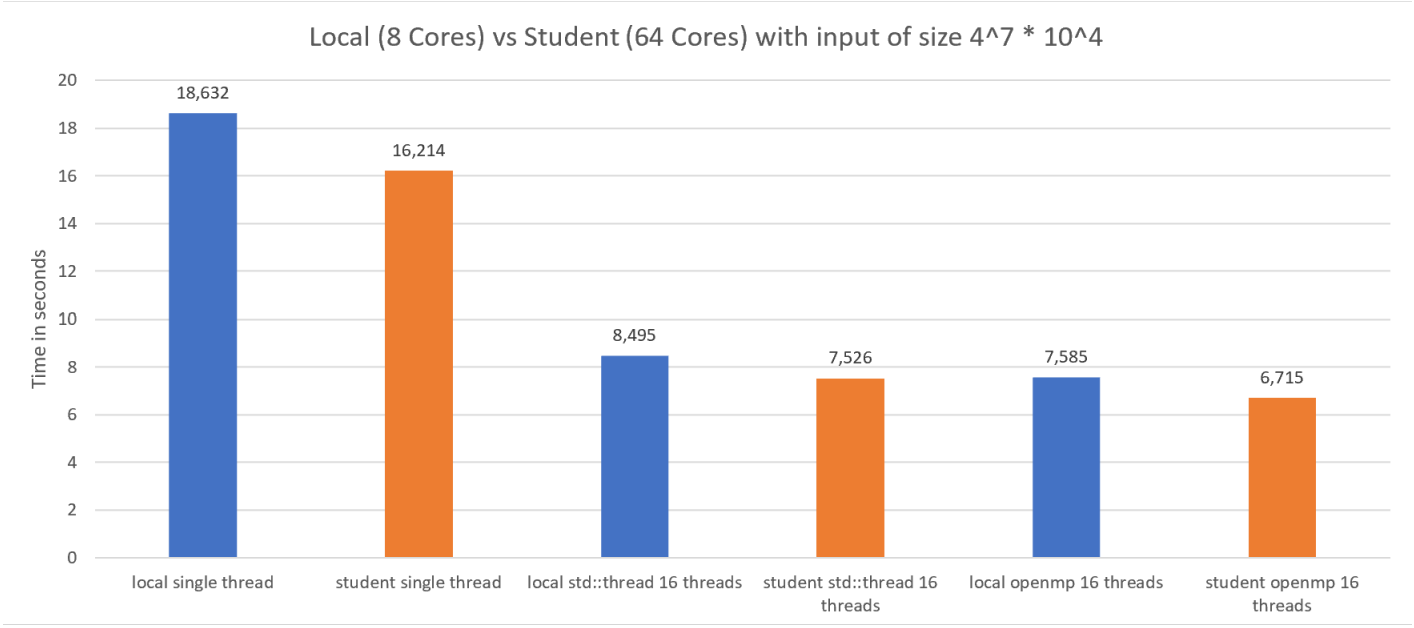
Instrukcje są dostępne w README.md

5 Wyniki

Wyniki trzech sposobów porównane są na poniższych wykresach. Na prywatnym laptopie miałem możliwość przetestowania różnych konfiguracji dostępnych procesorów za pomocą polecenia *taskset*. Na studencie mogłem testować jedynie liczbę wątków. Jestem prawie pewny, że możliwość określenia affinity procesu nie jest dla nas dostępna na studencie, a przynajmniej nie za pomocą polecenia *taskset*. Tego polecenia można natomiast użyć do sprawdzenia affinity procesu i w ten sposób określiłem, że proces na studencie ma dostępne 64 rdzenie. Wykresy przedstawiam dla najlepszych z testowanych konfiguracji. Lokalnie jest to 8 rdzeni i 16 wątków, a na Studencie jest to 64 rdzeni i 16 wątków.






Local (8 Cores) vs Student (64 Cores) with input of size 4^7 * 10^4
Time in seconds
20
18
16
14
12
10
8
6
4
2
0
18,632
16,214
8,495
7,526
7,585
6,715
local single thread
student single thread
local std::thread 16 threads
student std::thread 16 threads
local openmp 16 threads
student openmp 16 threads