Programowanie równoległe i rozproszone

Politechnika Krakowska

Laboratorium 1

Paweł Suchanicz,
Rafał Niemczyk

22 października 2019

Spis treści

1 Wstęp

1.1 Opis laboratorium

Celem laboratorium było wykorzystanie interfejsu OpenMP do zrównoleglenia kodu C++. Interfejs OpenMP składa się głównie z dyrektyw preprocesora a także z zmiennych środowiskowych i funkcji bibliotecznych. W laboratorium wykorzystywany będzie głównie do zrównoleglania pętli.

Algorytmy, które są implementowane a następnie zrównoleglane w ramach laboratorium to normalizacja min-max, standaryzacja rozkładem normalnym i klasyfikacja KNN (k-najbliższych sąsiadów). Zaimplementowany KNN uwzględnia jednego sąsiada i używa metryki euklidesowej.

Szybkość działania każdego algorytmu została zmierzona dla implementacji w C++, implementacji w C++ po zrównolegleniu dla różnej ilości wątków (1-8) oraz impelmentacji w Python (ze skorzystaniem z funkcji z pakietu scikit-learn).

1.2 Specyfikacja sprzętowa

Przy pomiarach szybkości wykonywania algorytmów wykorzystany był sprzęt w konfiguracji (maszyna wirtualna):

- Procesor: Intel Core i7-4712MQ 4 x 2.30GHz
- Ram: 4GB DDR3
- System: Linux (Fedora 22)

1.3 Zbiór danych

Wykorzystany został zbiór obrazów ręcznie pisanych cyfr MNIST. Zbiór danych ma format .csv i zawiera 60000 rekordów, gdzie każdy rekord odpowiada za jeden obrazek 28x28 pikseli w skali szarości. Pierwsza wartość w rekordzie jest cyfrą która widnieje na obrazku, a kolejne to wartości pikseli obrazka.

Dla zadań postawionych w laboratorium zbiór danych jest dość duży, więc został on obcięty do pierwszych 6000 rekordów, z czego 4500 przeznaczono do trenowania, a pozostałe 1500 do testowania.

2 Wyniki

2.1 Normalizacja min-max

Wzór:

$x^* = \frac{x - min(x)}{max(x) - min(x)}$

2.1.1 Implementacja

W C++ normalizacja została samodzielnie zgodnie z podanym powyżej wzorem. W pętli przechodzącej tablicy (po kolumnach) wyszukiwane są wartości minimum i maxium dla każdej kolumny a następnie wyliczana nowa wartość dla każdego z elementów tablicy. Zrównoleglenie pętli za pomocą dyrektyw:

```
#pragma omp parallel default(none) private(i, j, min, max)
shared(data, rows, columns, numberOfThreads)
num_threads(numberOfThreads)
#pragma omp for schedule(dynamic, numberOfThreads)
```

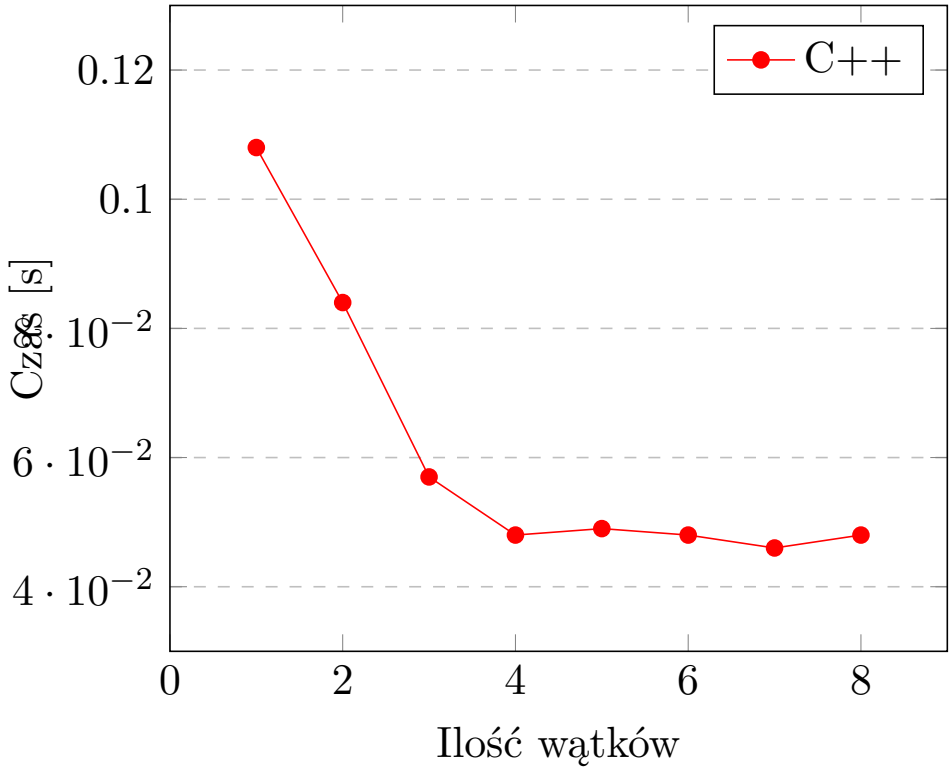
W Pythonie użyta została funkcja MinMaxScaler z pakietu sklearn .

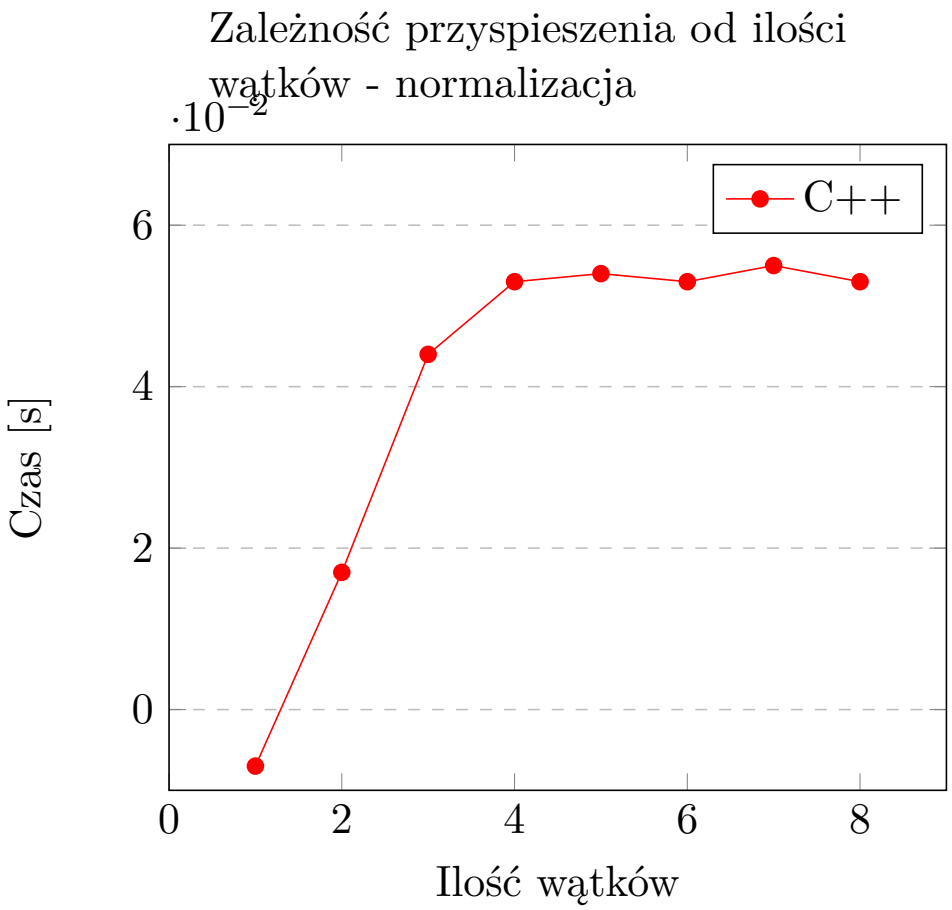
2.1.2 Porównanie wyników

Parametry	Czas [s]
C++	0.101
C++ OpenMP 1 wątek	0.108
C++ OpenMP 2 wątki	0.084
C++ OpenMP 3 wątki	0.057
C++ OpenMP 4 wątki	0.048
C++ OpenMP 5 wątki	0.049
C++ OpenMP 6 wątki	0.048
C++ OpenMP 7 wątki	0.046
C++ OpenMP 8 wątki	0.048
Pyhon sklearn	0.037

Po zastosowaniu OpenMP i zwiększeniu ilości używanych wątków widać znaczącą poprawę czasu wykonania. Czas wykonania spada gdy liczba wątków <= 4 (ilość rdzeni procesora na którym wykonywane były obliczenia). Nie udało się uzyskać czasu mniejszego niż z użyciem sklearn.

Zależność czasu od ilości wątków - normalizacja




Zależność przyspieszenia od ilości wątków - normalizacja
$\cdot 10^{-2}$
6
4
2
0
Czas [s]
C++
0
2
4
6
8
Ilość wątków

2.2 Standaryzacja rozkładem normalnym

Wzór:

$x^* = \frac{x-\mu}{\sigma}$

2.2.1 Implementacja

W C++ standaryzacja została zaimplementowana samodzielnie zgodnie z podanym powyżej wzorem. Przechodzimy w pętli po kolumnach i dla każdej kolumny szukamy wartości średniej i wariancji, a następnie wyliczamy nowe wartości dla każdego elementu tablicy. Zrównoleglenie pętli za pomocą dyrektyw:

```
#pragma omp parallel default(none) private(i, j, ave, amo, var)
shared(data, rows, columns, numberOfThreads)
num_threads(numberOfThreads)
#pragma omp for schedule(dynamic, numberOfThreads)
```

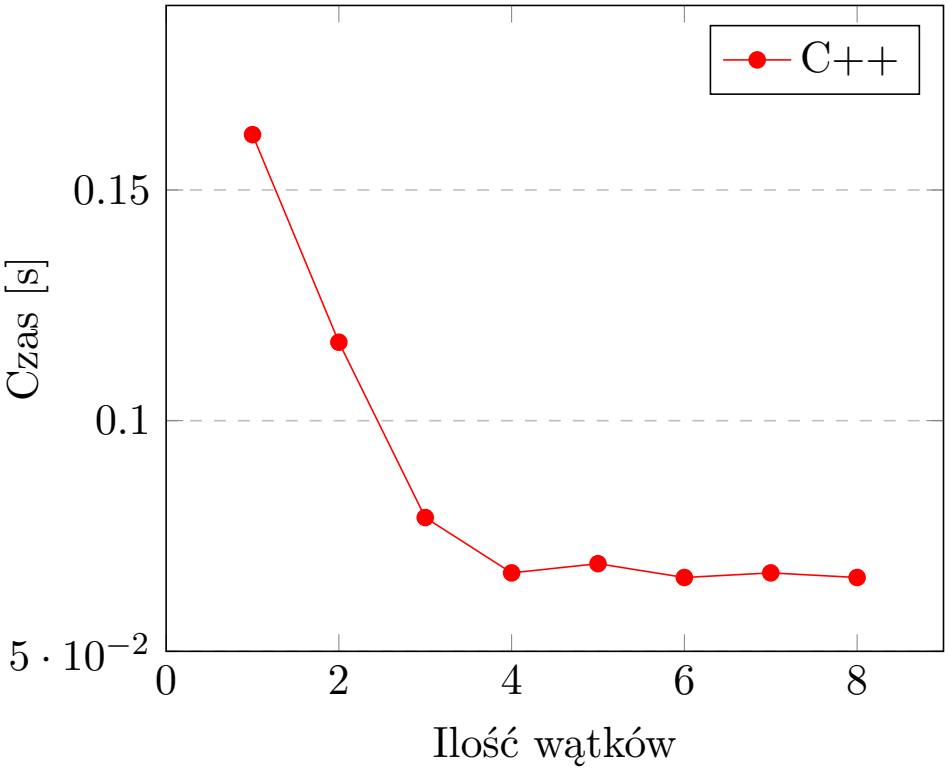
W Pythonie użyta została funkcja StandardScaler z pakietu sklearn.

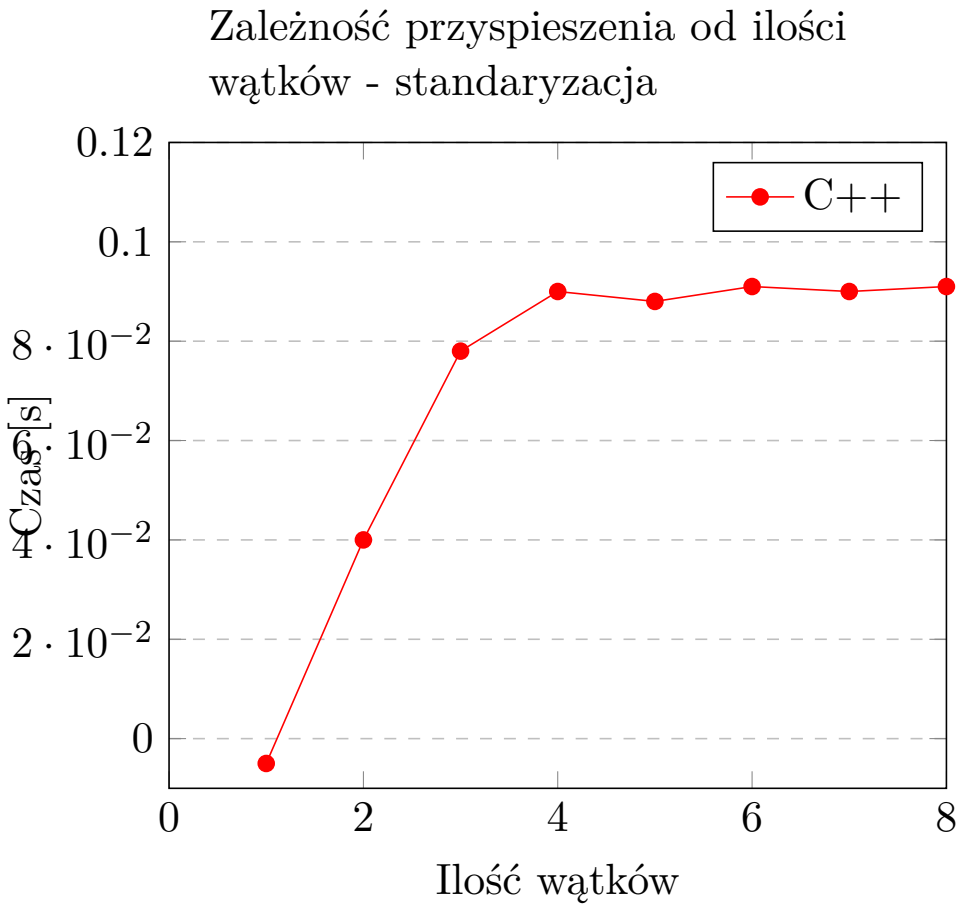
2.2.2 Porównanie wyników

Parametry	Czas [s]
C++	0.157
C++ OpenMP 1 wątek	0.162
C++ OpenMP 2 wątki	0.117
C++ OpenMP 3 wątki	0.079
C++ OpenMP 4 wątki	0.067
C++ OpenMP 5 wątki	0.069
C++ OpenMP 6 wątki	0.066
C++ OpenMP 7 wątki	0.067
C++ OpenMP 8 wątki	0.066
Pyhon sklearn	0.086

Zrównoleglenie w C++ przyniosło pozytywne skutki. Czas wykonania spadł o 59% przy użyciu czterech wątków w stosunku do czasu działania algorytmu na jednym wątku. Już wykorzystanie trzech wątków dało czas mniejszy niż implementacja z sklearn.

Zależność czasu od ilości wątków - standaryzacja




Zależność przyspieszenia od ilości wątków - standaryzacja
0.12
0.1
$8 \cdot 10^{-2}$
$6 \cdot 10^{-2}$
$4 \cdot 10^{-2}$
$2 \cdot 10^{-2}$
0
Czas [s]
C++
0
2
4
6
8
Ilość wątków

2.3 Klasyfikacja KNN

2.3.1 Implementacja

W C++ algorytm k najbliższych sąsiadów zaimplementowany samodzielnie. Algorytm uwzględnia tylko najbliższego sąsiada i korzysta z metryki euklidesowej. Zrównoleglenie za pomocą dyrektyw:

```
#pragma omp parallel default(none) private(current_test_row,
closest_neighbour_distance, closest_neighbour_index)
shared(max_float, numberOfThreads)
num_threads(numberOfThreads) reduction(+ : accurate_predictions)
#pragma omp for schedule(dynamic, numberOfThreads)
```

W Pythonie użyta została funkcja KNeighborsClassifier z pakietu sklearn z parametrami:

```
KNeighborsClassifier(n_neighbors=1, algorithm='brute', p=2, metric='minkowski',
n_jobs=app_conf['jobs_number'])
```

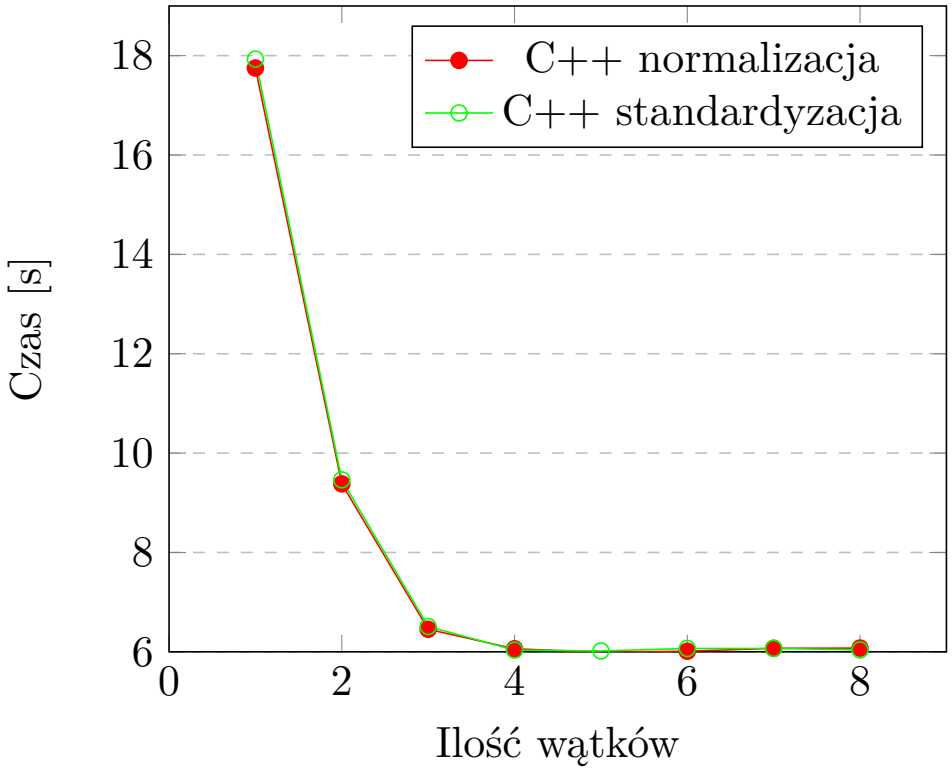
Czasy były mierzone dla wartości njobs od 1 do 4.
Dokładność accuracy wyniosła 71% dla danych po standard scalerze oraz 66% dla danych po min-max scalarze. W przypadku normalizacji w c++ otrzymano dokładność 93.67%. W przypadku standaryzacji dokładność wyniosła 90Można wysnuć wniosek, że algorytm w pakiecie sklearn działa w nieco inny sposób stąd mniejszy czas wykonania, ale i mniejsza dokładność. Użycie równoległości oczywiście nie miało wpływu na dokładność działania Knn.

2.3.2 Porównanie wyników

Parametry	Czas [s]
C++ OpenMP 1 wątek normalizacja	17.750
C++ OpenMP 2 wątki normalizacja	9.381
C++ OpenMP 3 wątki normalizacja	6.454
C++ OpenMP 4 wątki normalizacja	6.061
C++ OpenMP 5 wątek normalizacja	5.993
C++ OpenMP 6 wątki normalizacja	6.012
C++ OpenMP 7 wątki normalizacja	6.071
C++ OpenMP 8 wątki normalizacja	6.075
Pyhon sklearn njobs=1 normalizacja	0.215
Pyhon sklearn njobs=2 normalizacja	0.323
Pyhon sklearn njobs=3 normalizacja	0.455
Pyhon sklearn njobs=4 normalizacja	0.386
C++ OpenMP 1 wątek standaryzacja	17.930
C++ OpenMP 2 wątki standaryzacja	9.462
C++ OpenMP 3 wątki standaryzacja	6.510
C++ OpenMP 4 wątki standaryzacja	6.035
C++ OpenMP 5 wątek standaryzacja	6.018
C++ OpenMP 6 wątki standaryzacja	6.067
C++ OpenMP 7 wątki standaryzacja	6.068
C++ OpenMP 8 wątki standaryzacja	6.039
Pyhon sklearn 1 wątek standaryzacja	0.208
Pyhon sklearn 2 wątki standaryzacja	0.326
Pyhon sklearn 3 wątki standaryzacja	0.329
Pyhon sklearn 4 wątki standaryzacja	0.328

Zrównoleglenie w c++ przyniosło pozytywny skutek. Wyniki były niemal identyczne dla danych po normalizacji jak i standaryzacji. Już przy użyciu dwóch wątków czas zmniejszył się ponad dwukrotnie. Wykorzystanie większej ilości wątków niż 4 (liczba rdzeni procesora na którym wykonywały się obliczenia) nie przynosiła już żadnych zmian w szybkości działania. Zależność między ilością wątków a czasem wykonania ma charakter wykładniczy. W przypadku Python zwiększanie parametru njobs algorytmu KNN przynosiło odwrotny skutek do oczekiwanego - czas wykonania wydłużał się.

Zależność czasu od ilości wątków - knn



Zależność czasu od parametru njobs - knn