Programowanie równoległe i rozproszone

Politechnika Krakowska

Laboratorium 2

Paweł Suchanicz,
Rafał Niemczyk

28 października 2019

Spis treści

1 Wstęp **2**
1.1 Opis laboratorium 2
1.2 Specyfikacja sprzętowa 2
1.3 Zbiór danych 2

2 Wyniki **3**
2.1 Normalizacja min-max 3
2.1.1 Implementacja 3
2.1.2 Porównanie wyników 3
2.2 Standaryzacja rozkładem normalnym 6
2.2.1 Implementacja 6
2.2.2 Porównanie wyników 6
2.3 Klasyfikacja KNN 9
2.3.1 Implementacja 9
2.3.2 Porównanie wyników 9

1 Wstęp

1.1 Opis laboratorium

Celem laboratorium było wykorzystanie standardu MPI do zrównoleglenia kodu C++. MPI (Message Passing Interface) to standard przesyłania komunikatów pomiędzy procesami.

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 sekwencyjnej w C++, implementacji równoległej w C++ dla różnej ilości procesów (1-8) oraz implementacji w Python (ze skorzystaniem z funkcji z pakietu scikit-learn).

1.2 Specyfikacja sprzętowa

Przy pomiarach szybkości wykonywania algorytmów wykorzystany był sprzęt w konfiguracji:

- Procesor: Intel Core i7-4712MQ 4 x 2.30GHz
- Ram: 8GB 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.

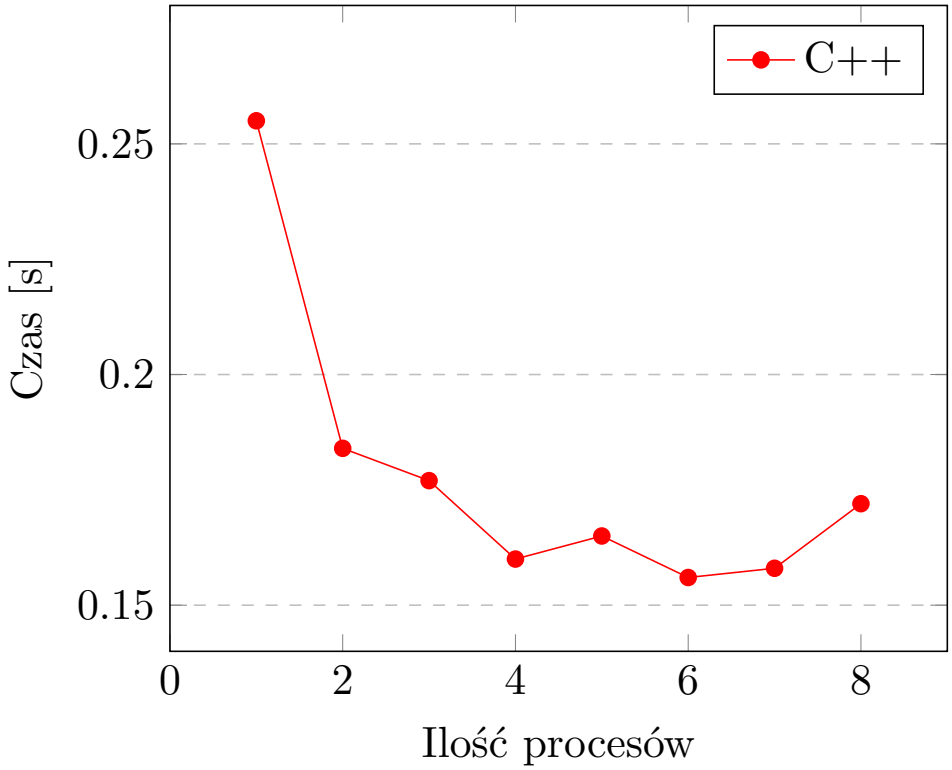
W Pythonie użyta została funkcja MinMaxScaler z pakietu sklearn .

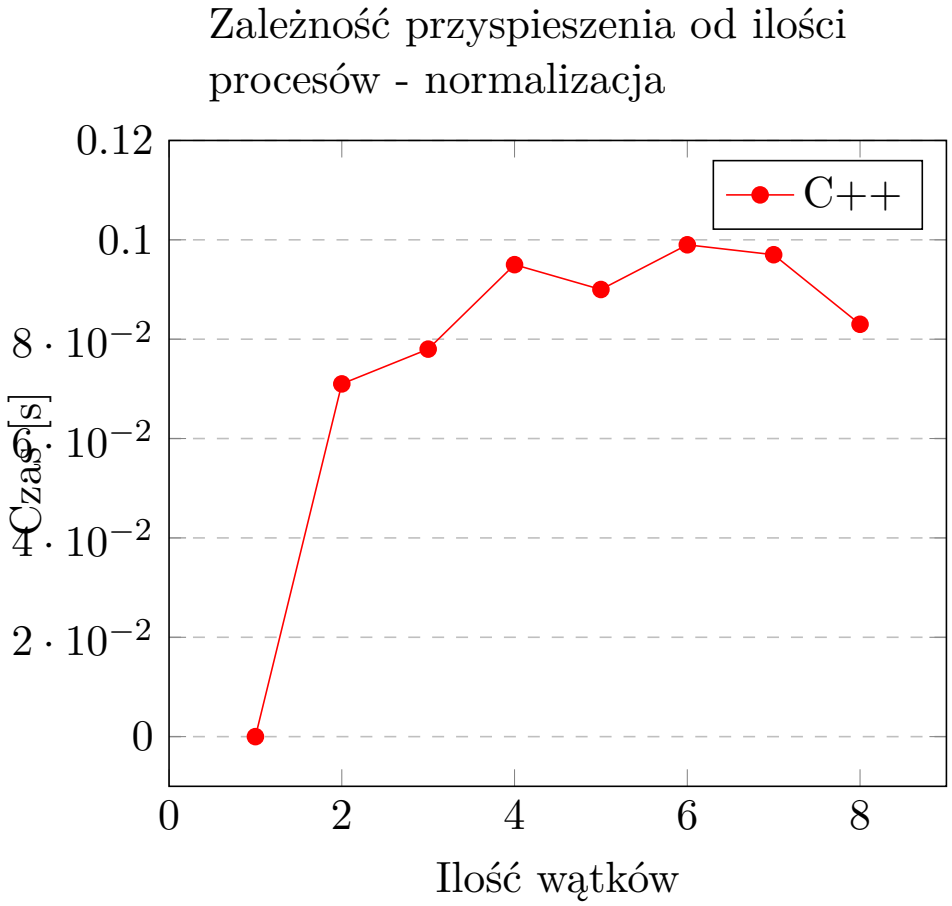
2.1.2 Porównanie wyników

Parametry	Czas [s]
C++	0.101
C++ MPI 1 proces	0.255
C++ MPI 2 procesy	0.184
C++ MPI 3 procesy	0.177
C++ MPI 4 procesy	0.160
C++ MPI 5 procesów	0.165
C++ MPI 6 procesów	0.156
C++ MPI 7 procesów	0.158
C++ MPI 8 procesów	0.172
Pyhon sklearn	0.037

Po zastosowaniu MPI i zwiększeniu ilości używanych procesów widać poprawę czasu wykonania. Czas wykonania spada gdy liczba procesów <= 4 (ilość rdzeni procesora na którym wykonywane były obliczenia). Nie udało się uzyskać czasu mniejszego niż z użyciem sklearn. Po zastosowaniu MPI można zauważyć spory wzrost czasu wykonania (0.154 sekundy co stanowi 152 początkowego czasu). Związane to jest z narzutem jaki spowodowało użycie funkcji $MPI\_Scatter$ oraz MPI_Gather. Przy zwiększeniu ilości używanych procesów widać poprawę czasu wykonania. Czas wykonania spada gdy liczba procesów <= 4 (ilość rdzeni procesora na którym wykonywane były obliczenia). Nie udało się uzyskać czasu mniejszego niż z użyciem sklearn.

Zależność czasu od ilości procesów - normalizacja




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

2.2 Standaryzacja rozkładem normalnym

Wzór:

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

2.2.1 Implementacja

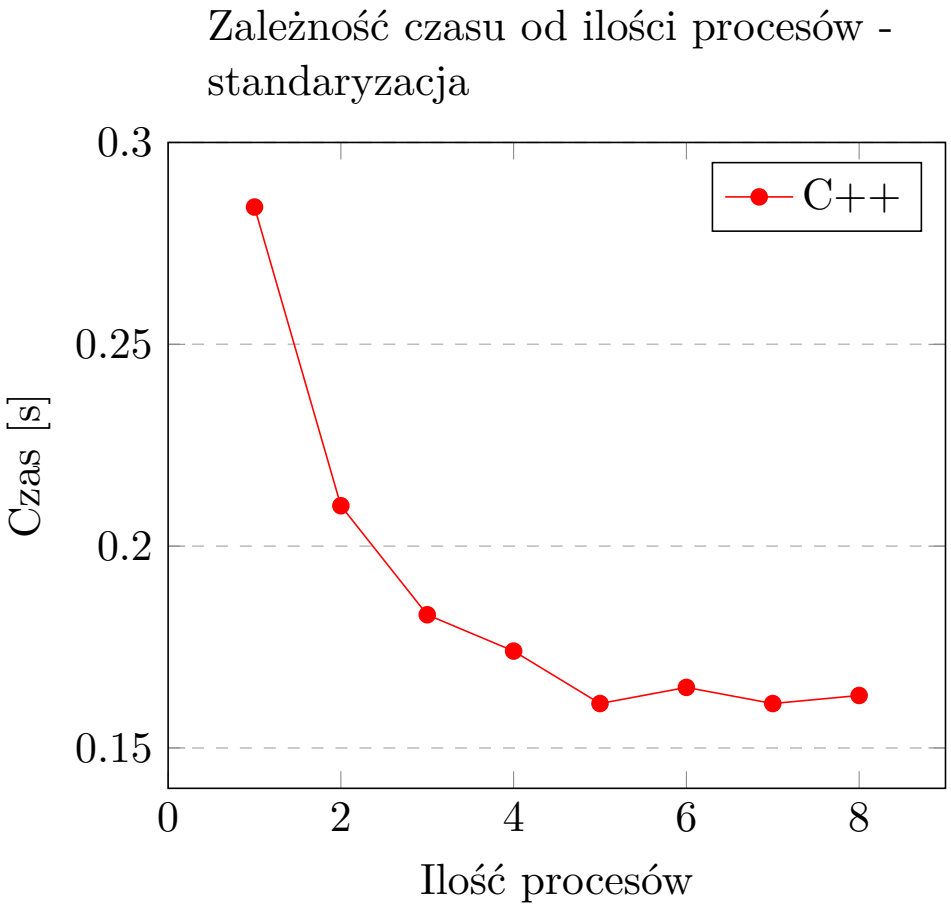
W C++ standaryzacja została zaimplementowana samodzielnie zgodnie z podanym powyżej wzorem. Przechodzimy w pętli po kolumnach i dla każdej kolumny szukamy wartości średniej i wariancji, a następnie wyliczamy nowe wartości dla każdego elementu tablicy.

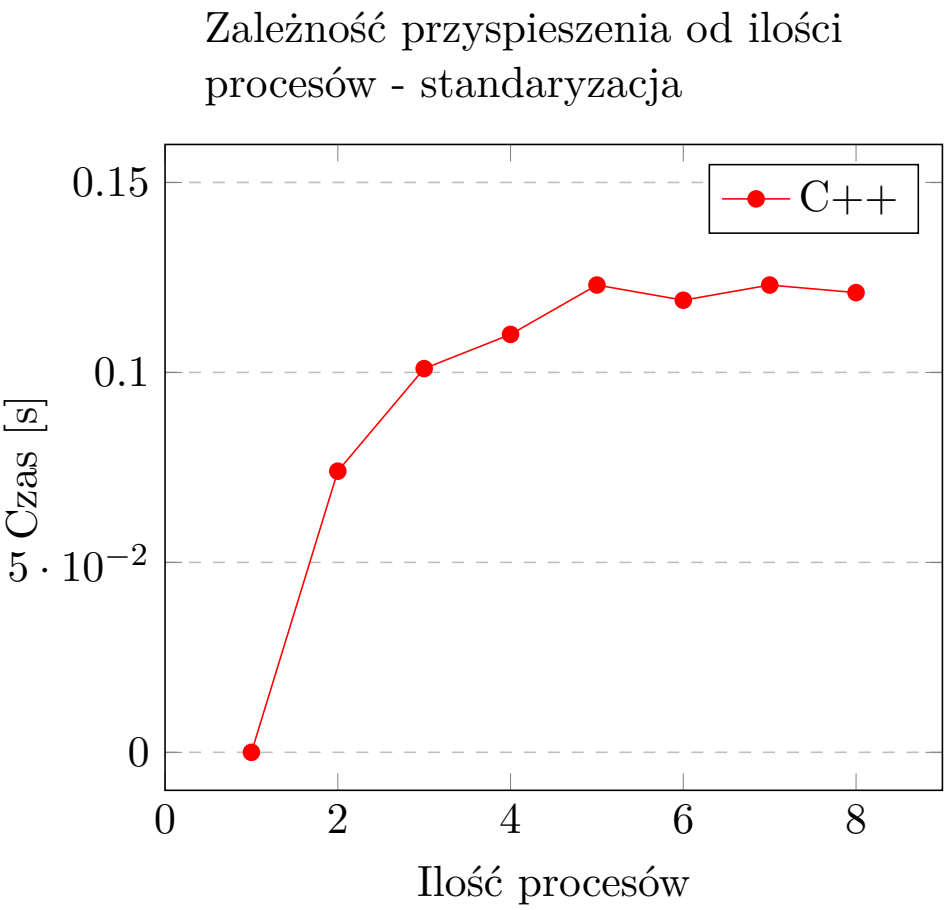
W Pythonie użyta została funkcja StandardScaler z pakietu sklearn.

2.2.2 Porównanie wyników

Parametry	Czas [s]
C++	0.157
C++ MPI 1 proces	0.284
C++ MPI 2 procesy	0.210
C++ MPI 3 procesy	0.183
C++ MPI 4 procesy	0.174
C++ MPI 5 procesów	0.161
C++ MPI 6 procesów	0.165
C++ MPI 7 procesów	0.161
C++ MPI 8 procesów	0.163
Pyhon sklearn	0.086

Podobnie jak w przypadku normalizacji samo użycie MPI spowodowało dość duży narzut (w stosunku do czasu w jakim wykonuje się normalizacja). Samo zwiększanie liczby procesów jednak przynosiło pozytywne skutki. Czas wykonania spadł o 38% przy użyciu czterech wątków w stosunku do czasu działania algorytmu na jednym wątku z użyciem MPI. Czas pozostał jednak większy niż w sklearn z powodu narzutu MPI.


Zależność czasu od ilości procesów - standaryzacja
0.3
0.25
0.2
0.15
Czas [s]
0
2
4
6
8
Ilość procesów
C++


Zależność przyspieszenia od ilości procesów - standaryzacja
C++
0.15
0.1
$5 \cdot 10^{-2}$
0
Czas [s]
0
2
4
6
8
Ilość procesów

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.

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'])
```

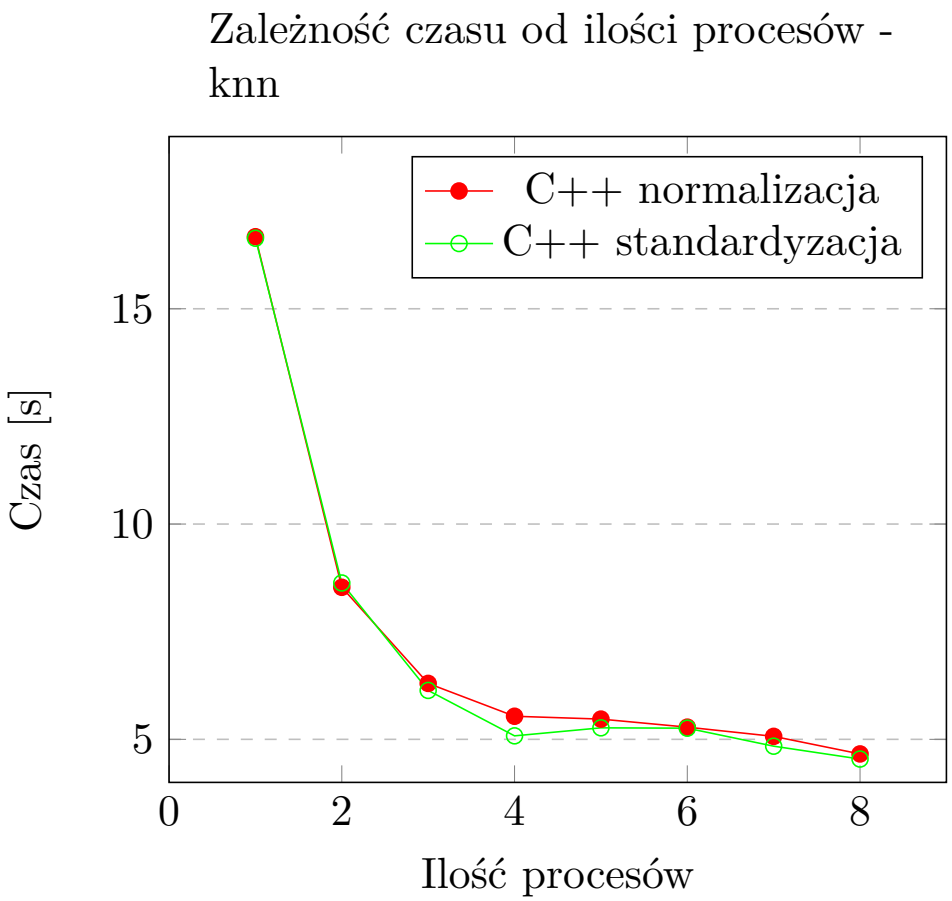
Czasy były mierzone dla wartości njobs od 1 do 4.
Dokładność accuracy wyniosła 71% dla danych po standard scalerze oraz 66% dla danych po min-max scalarze. W przypadku normalizacji w c++ otrzymano dokładność 91.67%. W przypadku standaryzacji dokładność wyniosła 90%. Można wysnuć wniosek, że algorytm 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++ MPI 1 proces normalizacja	16.673
C++ MPI 2 procesy normalizacja	8.531
C++ MPI 3 procesy normalizacja	6.299
C++ MPI 4 procesy normalizacja	5.536
C++ MPI 5 procesów normalizacja	5.470
C++ MPI 6 procesów normalizacja	5.283
C++ MPI 7 procesów normalizacja	5.069
C++ MPI 8 procesów normalizacja	4.661
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++ MPI 1 proces standaryzacja	16.638
C++ MPI 2 procesy standaryzacja	8.631
C++ MPI 3 procesy standaryzacja	6.136
C++ MPI 4 procesy standaryzacja	5.081
C++ MPI 5 procesów standaryzacja	5.267
C++ MPI 6 procesów standaryzacja	5.259
C++ MPI 7 procesów standaryzacja	4.841
C++ MPI 8 procesów standaryzacja	4.542
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

Użycie MPI w c++ przyniosło pozytywny skutek. Wyniki były niemal identyczne dla danych po normalizacji jak i standaryzacji. Już przy użyciu dwóch procesów czas zmniejszył się około dwukrotnie, przy użyciu 4 procesów prawie czterokrotnie. Wykorzystanie większej ilości wątków niż 4 (liczba rdzeni procesora na którym wykonywały się obliczenia) przynosiła już niewielkie

spadki lub czasami podniesienie się czasu wykonania. W przypadku Python zwiększanie parametru njobs algorytmu KNN przynosiło odwrotny skutek do oczekiwanego - czas wykonania wydłużał się.



Zależność czasu od parametru njobs - knn