Algorytmy skalowalnego przetwarzania danych — projekt 3

dr Piotr Przymus, mgr Mikołaj Fejzer, dr Krzysztof Rykaczewski

7 maja 2020

Spis treści

1 Definicje, oznaczenia **1**
1.1 Reprezentacja macierzy gęstych 1
1.2 Reprezentacja macierzy rzadkiej 1

2 Zadania: MapReduce — mnożenie macierz-wektor, macierz-macierz **2**
2.1 Macierz · wektor — wektor mieści się w pamięci (20 pkt) 2
2.2 Macierz · wektor — rozmiar wektora przekracza rozmiar pamięci (20 pkt) 4
2.3 Macierz · macierz — kilka kroków MapReduce (30 pkt) 7
2.4 Macierz · macierz — pojedynczy krok MapReduce (30 pkt) 11

1 Definicje, oznaczenia

1.1 Reprezentacja macierzy gęstych

Macierz jest zwyczajowo przetrzymywana w tablicy jedno lub dwuwymiarowej. Dostęp do elementów macierzy dany jest poprzez dwa indeksy i oraz j (dla jednowymiarowej i łatwo wyliczyć). Przyjmuje się, że i oznacza numer/indeks wiersza (od góry do dołu), a j oznacza numer kolumny (od lewej do prawej). Dla macierzy $n \times p$ pamięć potrzebna do przetrzymania jej jest proporcjonalna do iloczynu $n \cdot p$.

1.2 Reprezentacja macierzy rzadkiej

Macierz rzadka to macierz, w której większość elementów jest zerowa.

Do trzymania oraz manipulowania rzadką macierzą na komputerze stosuje się specjalistyczne algorytmy i struktury danych, które biorą pod uwagę rzadkość wypełnienia macierzy. Tradycyjne algorytmy stosowane dla gęstych macierzy są nieefektywne w przypadku macierzy rzadkich, gdyż moc obliczeniowa oraz pamięć są marnowane na przetwarzanie zer.

W przypadku rzadkiej macierzy pamięć potrzebna na przetrzymanie takiej macierzy jest zredukowana poprzez operowanie tylko na niezerowych elementach. W zależności od rozkładu tych elementów istnieje wiele różnych form reprezentacji macierzy rzadkich.

W naszym przypadku warto rozważyć następującą postać (i, j, a_{ij}). Taka postać potrafi znacząco zredukować rozmiar przetrzymywanych danych.

2 Zadania: MapReduce — mnożenie macierz-wektor, macierz-macierz

Celem zadania jest napisanie programu realizującego podstawowe operacje na macierzach, czyli mnożenie macierz-wektor oraz macierz-macierz. Te operacje są intensywnie wykorzystywane przez wiele zaawansowanych algorytmów przetwarzania danych.

2.1 Macierz · wektor — wektor mieści się w pamięci (20 pkt)

2.1.1 Wstęp

Dane są macierz $A = [a_{ij}]_{1 \leq i,j \leq n}$ (przez a_{ij} oznaczamy element i-tym wierszu oraz j-tej kolumnie) oraz wektor $v = [v_1, v_2, \ldots, v_n]$. Niech $x = A \cdot v$, tzn.

$$x_i = \sum_{j=1}^{n} a_{ji} v_j, \quad 1 \leq i \leq n. \tag{1}$$

Zakładamy, że n jest duże jednak na tyle małe, że wektor v mieści się w pamięci każdego węzła obliczeniowego, a zatem jest dostępny dla każdego zadania Map. Przyjmujemy również, że mamy bezpośredni dostęp do każdego z elementów macierzy A, albo poprzez jego pozycję w pliku lub dlatego, że jest składowany wraz z koordynatami, tzn. jako trójka (i, j, a_{ij}).

Na podobnej zasadzie mamy dostęp do elementów v_j wektora v.

2.1.2 Funkcja Map

Funkcja Map działa na jednym elemencie macierzy A. Każde zadanie Map będzie operowało na kawałku macierzy A. Dla każdego elementu a_{ij} produkuje parę klucz-wartość $(i, a_{ij} \cdot v_j)$. W ten sposób wszystkie składniki sumy definiującej x_i posiadają ten sam klucz i.

```
map(j, v_j) -> store()
map(i, j, a_ij) -> (i, a_ij* v_j)
```

2.1.3 Funkcja Reduce

Reduce sumuje wszystkie wartości powiązane z kluczem i. Rezultatem będzie para (i, x_i).

```
reduce(i, [a_ij v_j]_j) -> (i, sum_j(a_ij v_j))
```

2.1.4 Przykład

Uwaga: na początku pliku wejściowego mogą być współrzędne wektora, a później macierzy.

Wektor v mieści się w pamięci, macierz A niekoniecznie. Mamy $Av = x$, a `x[i] =` suma po `j` wyrażeń `m[i,j]*v[j]`.

```
# i to nr wiersza
map(i, j, m[i,j]) -> (i, m[i,j]*v[j])
# reduce robi sume po tych kolumnach
reduce(i, [m[i,j]*v[j]]_j) -> (i, suma_po_j(m[i,j]*v[j]))
```

Rozważmy

$$\begin{bmatrix} 1 & 0 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ -2 \end{bmatrix} = ? \tag{2}$$

W naszej rzadkiej reprezentacji będzie:

```
0;0;1
1;0;2
1;1;-1
```

`map`owanie:

```
0;  2
1;  2*2 = 4
1;  (-2)*(-1) = 2
```

`reduce`:

```
0;  2
1;  6
```

Stąd

$$\begin{bmatrix} 1 & 0 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \end{bmatrix} \tag{3}$$

W mnożeniu macierzy przez wektor, który nie mieści się w pamięci wybieramy po kolei wiersze z macierzy i mnożymy przez dany element tego wektora (wczytujemy je na bieżąco). Należy gdzieś w danych dodać część wektora, którą mnożymy, żeby wiedzieć, z którego pliku czytać odpowiednią część wektora/macierzy.

2.1.5 Zadanie

Policz na tablicy/kartce przykład mnożenia macierz-wektor.

2.1.6 Zadanie

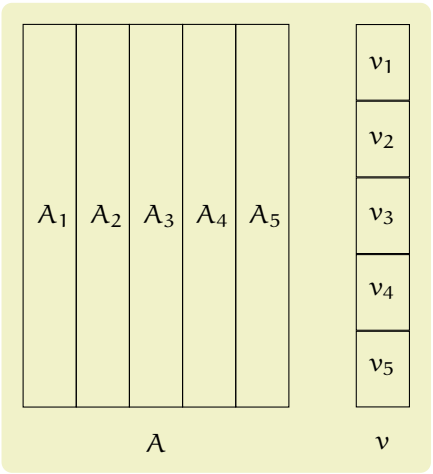
Przygotuj kod realizujący mnożenie macierz-wektor i przetestuj go.

2.2 Macierz · wektor — rozmiar wektora przekracza rozmiar pamięci (20 pkt)

2.2.1 Wstęp

W większości przypadków wektor v i macierz A będą na tyle duże, że nie będą mieścić się w pamięci węzła obliczeniowego. Oczywiście v nie musi mieścić się w pamięci, jednak narazi nas to na spory koszt związany z wymianą fragmentów pomiędzy pamięcią a dyskiem.

Alternatywnym rozwiązaniem jest podzielenie macierzy A oraz wektora v na taką samą liczbę horyzontalnych pasków równej szerokości. Celem jest taki podział danych, aby fragment wektora mieścił się w pamięci węzła obliczeniowego.



Rysunek 1: Przykładowy podział macierzy i wektora na 5 fragmentów.

Ponieważ k-ty fragment macierzy mnożony jest tylko przez k-ty fragment wektora, możemy podzielić macierz na wiele plików (jeden plik na fragment). Analogicznie postępujemy w przypadku wektora.

Każde zadanie Map ma przypisany fragment macierzy oraz odpowiadający mu fragment wektora. Map i Reduce są jak poprzednio.

2.2.2 Przykład

Wówczas k-ty fragment macierzy mnożony jest tylko przez k-ty fragment wektora.

Niech `slice_length = 3`. Załóżmy, że mamy początkową macierz A w postaci:

```
(3, 3), 7
(3, 2), 1
(2, 1), -1
(1, 1), 2.5
(0, 1), 3.2
(2, 4), 16
(1, 6), 3
(1, 5), 10.5
(3, 8), 8
```

oraz wektor v w postaci:

```
0; 2
1; -3
2; 0
3; 9
4; 1
5; 2
6; 2
7; 0
8; 1
```

Następnie za pomocą jednego wykonania MapReduce możemy podzielić macierz na paski: wystarczy brać pod uwagę j i w ten sposób dopisywać elementy do odpowiedniego pliku, tzn. element

`(i, j), m_ij,`

dopisz do pliku/fragmentu M_k, gdzie $k =$ `floor(j/slice_length)`.

Mamy wówczas paski:

```
M_0:

    (0, 1), 3.2
    (1, 1), 2.5
    (3, 2), 1
    (2, 1), -1

M_1:

    (3, 3), 7
```

```
    (2, 4), 16
    (1, 5), 10.5

M_2:

    (1, 6), 3
    (3, 8), 8
```

Te fragmenty możemy potem znów zebrać (lub nie) do wspólnego pliku (przecież i tak wszystko będzie szło strumieniem).

Podobnie dostajemy:

```
v_0:

    0; 2
    1; -3
    2; 0

v_1:

    3; 9
    4; 1
    5; 2

v_2:

    6; 2
    7; 0
    8; 1
```

Zauważmy, że elementy macierzy w jednym pliku nie muszą być posortowane.

Wówczas dla elementu, który trafił do mappera można dopasować fragment wektora, na którym mają być wykonane obliczenia za pomocą wzoru $k = \texttt{floor(j/slice\_length)}$. Wówczas $v = v_k$.

Dzięki takiemu podziałowi macierzy wymiany wektora v będziemy dokonywać tylko wtedy, gdy skończą nam się elementy z jednego pasma macierzy i będziemy musieli przejść na nowe.

2.2.3 Zadanie

Przygotuj odpowiedni przykład demonstrujący omówioną technikę i przetestuj go.

Macierz najwygodniej zapisać w kilku plikach, każdy plik to jeden pasek np. `A_1`, `A_2`, `A_3`, `A_4`, `A_5`. Wektor najwygodniej zapisać w kilku plikach np. `v_1`, `v_2`, `v_3`, `v_4`, `v_5`.

Napisać program, który będzie wczytywał odpowiednie fragmenty wektora w zależności od tego, który pasek macierzy jest przetwarzany (tzn. sprawdzamy, które kolumny analizujemy w kroku Map).

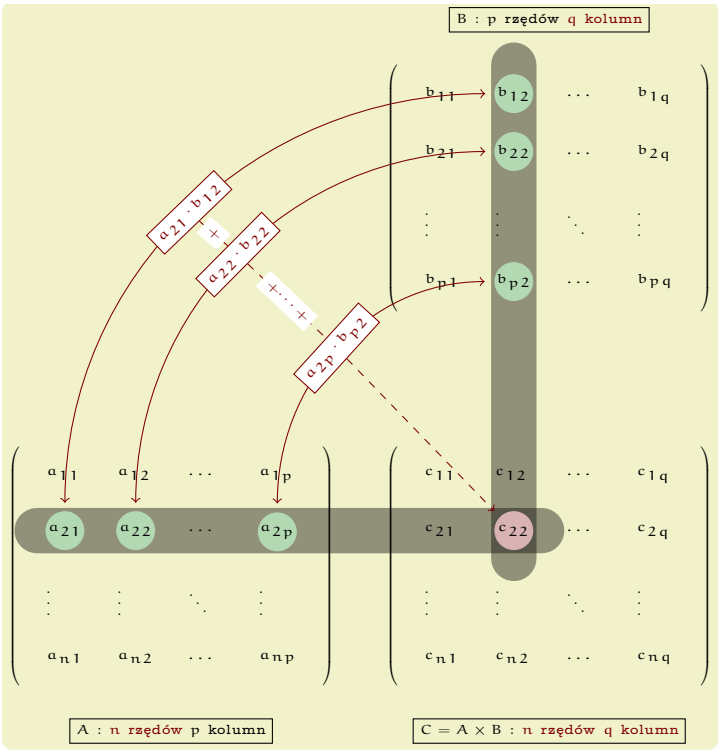
2.3 Macierz · macierz — kilka kroków MapReduce (30 pkt)

2.3.1 Wstęp

Dane są macierz $A = [a_{ij}]_{1 \le i \le n, 1 \le j \le p}$ oraz macierz $B = [b_{jk}]_{1 \le j \le p, 1 \le k \le q}$.

Niech macierz C będzie wynikiem mnożenia $C = A \cdot B = [c_{ik}]_{1 \le i \le n, 1 \le k \le q}$, tzn.

$$c_{ik} = \sum_{j=1}^{p} a_{ij} b_{jk}, \quad 1 \le i \le n, 1 \le k \le q. \tag{4}$$



2.3.2 Część pierwsza

2.3.2.1 Funkcja Map

Dla macierzy A postaci $(i, j, a_{ij})_A$ generujemy pary klucz-wartość $(j, (I_A, i, a_{ij}))$, gdzie I_A jest identyfikatorem macierzy A.

Analogicznie postępujemy dla macierzy B postaci $(j, k, b_{jk})_B$, tzn. generujemy pary $(j, (I_B, k, b_{jk}))$, gdzie I_B jest identyfikatorem macierzy B.

```
map(i, j, a_ij) -> (j, (A, i, a_ij))
map(j, k, b_jk) -> (j, (B, k, b_jk))
```

2.3.2.2 Funkcja Reduce

Dla każdego klucza j przetwarza listę zasocjowanych wartości. Dla każdej z wartości pochodzącej z macierzy A (np. (I_A, i, a_{ij})) oraz dla każdej wartości pochodzącej z macierzy B (np. (I_B, k, b_{jk})) tworzy parę klucz-wartość z kluczem równym (i, k) oraz wartością elementu równą $(a_{ij} \cdot b_{jk})$

```
reduce(j, [(A, i, a_ij), (B, k, b_jk)]) -> ((i, k), a_ij b_jk)
```

2.3.3 Część druga

2.3.3.1 Funkcja Map

Funkcja identycznościowa. Dla każdej pary (k, v) produkuje dokładnie tę parę.

```
map((i, k), a_ij b_jk) -> ((i, k), a_ij b_jk)
```

2.3.3.2 Funkcja Reduce

Dla każdego klucza (i, k) wytwarza sumę wszystkich wartości zasocjowanych z tym kluczem. Wynikiem jest para $((i, k), v)$, gdzie v jest wartością elementu c_{ik} macierzy $C = AB$.

```
reduce((i, k), [a_ij b_jk]_j) -> ((i, k), sum_j(a_ij b_jk))
```

2.3.4 Przykład

Rozważmy

$$\begin{bmatrix} 1 & 2 & -3 \\ 6 & 8 & 0 \\ 0 & 0 & 3 \\ 4 & 0 & 0 \end{bmatrix} \begin{bmatrix} 9 & 0 & 4 \\ 3 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix} = ? \tag{5}$$

W reprezentacji rzadkiej przedstawia się to jako:

```
A :                 B :
    (1, 1, 1)           (1, 1, 9)
    (1, 2, 2)           (1, 3, 4)
    (1, 3, -3)          (2, 1, 3)
    (2, 1, 6)           (2, 2, 2)
    (2, 2, 8)           (3, 2, 1)
    (3, 3, 3)           (3, 3, 2)
    (4, 1, 4)
```

2.3.4.1 Krok 1

2.3.4.1.1 Map nr 1:

```
(1, 1, 1)  -> (1, (A, 1, 1))        (1, 1, 9)  -> (1, (B, 1, 9))
(1, 2, 2)  -> (2, (A, 1, 2))        (1, 3, 4)  -> (1, (B, 3, 4))
(1, 3, -3) -> (3, (A, 1, -3))       (2, 1, 3)  -> (2, (B, 1, 3))
(2, 1, 6)  -> (1, (A, 2, 6))        (2, 2, 2)  -> (2, (B, 2, 2))
(2, 2, 8)  -> (2, (A, 2, 8))        (3, 2, 1)  -> (3, (B, 2, 1))
(3, 3, 3)  -> (3, (A, 3, 3))        (3, 3, 2)  -> (3, (B, 3, 2))
(4, 1, 4)  -> (1, (A, 4, 4))
```

Sortujemy po kluczach, generujemy wszystkie możliwe kombinacje A i B, mnożymy każdą parę.

2.3.4.1.2 Reduce nr 1:

```
(1, (A, 1, 1))          [(A, 1, 1), (B, 1, 9)]          ->  ((1, 1), 1*9)
(1, (A, 2, 6))          [(A, 1, 1), (B, 3, 4)]          ->  ((1, 3), 1*4)
(1, (A, 4, 4))  ~       [(A, 2, 6), (B, 1, 9)]          ->  ((2, 1), 6*9)
(1, (B, 1, 9))          [(A, 2, 6), (B, 3, 4)]          ->  ((2, 3), 6*4)
(1, (B, 3, 4))          [(A, 4, 4), (B, 1, 9)]          ->  ((4, 1), 4*9)
                        [(A, 4, 4), (B, 3, 4)]          ->  ((4, 3), 4*4)

(2, (A, 1, 2))          [(A, 1, 2), (B, 1, 3)]          ->  ((1, 1), 2*3)
(2, (A, 2, 8))  ~       [(A, 1, 2), (B, 2, 2)]          ->  ((1, 2), 2*2)
(2, (B, 1, 3))          [(A, 2, 8), (B, 1, 3)]          ->  ((2, 1), 8*3)
(2, (B, 2, 2))          [(A, 2, 8), (B, 2, 2)]          ->  ((2, 2), 8*2)

(3, (A, 1, -3))         [(A, 1, -3), (B, 2, 1)]         ->  ((1, 2), -3*1)
(3, (A, 3, 3))  ~       [(A, 1, -3), (B, 3, 2)]         ->  ((1, 3), -3*2)
(3, (B, 2, 1))          [(A, 3, 3), (B, 2, 1)]          ->  ((3, 2), 3*1)
(3, (B, 3, 2))          [(A, 3, 3), (B, 3, 2)]          ->  ((3, 3), 3*2)
```

2.3.4.2 Krok 2

2.3.4.2.1 Map nr 2:

Identyczność, więc po prostu zwracamy w nim to samo co z poprzedniego `reduce`:

```
((1, 1), 9)
((1, 3), 4)
((2, 1), 54)
((2, 3), 24)
```

```
((4, 1), 36)
((4, 3), 16)
((1, 1), 6)
((1, 2), 4)
((2, 1), 24)
((2, 2), 16)
((1, 2), -3)
((1, 3), -6)
((3, 2), 3)
((3, 3), 6)
```

Sortujemy po kluczach, a potem robimy sumę po kluczach.

2.3.4.2.2 Reduce nr 2:

```
((1, 1), 9)
((1, 1), 6)     ->  ((1, 1), 6+9)
-----------
((1, 2), 4)
((1, 2), -3)    ->  ((1, 2), 4-3)
-----------
((1, 3), 4)
((1, 3), -6)    ->  ((1, 3), 4-6)
-----------
((2, 1), 54)
((2, 1), 24)    ->  ((2, 1), 54+24)
-----------
((2, 2), 16)    ->  ((2, 2), 16)
-----------
((2, 3), 24)    ->  ((2, 3), 24)
-----------
((3, 2), 3)     ->  ((3, 2), 3)
-----------
((3, 3), 6)     ->  ((3, 3), 6)
-----------
((4, 1), 36)    ->  ((4, 1), 36)
-----------
((4, 3), 16)    ->  ((4, 3), 16)
```

Czyli

$$\begin{bmatrix} 1 & 2 & -3 \\ 6 & 8 & 0 \\ 0 & 0 & 3 \\ 4 & 0 & 0 \end{bmatrix} \begin{bmatrix} 9 & 0 & 4 \\ 3 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 15 & 1 & -2 \\ 78 & 16 & 24 \\ 0 & 3 & 6 \\ 36 & 0 & 16 \end{bmatrix} \tag{6}$$

Istotnie

```
import numpy as np
A = np.array([[1,2,-3], [6,8,0], [0,0,3], [4,0,0]])
B = np.array([[9,0,4], [3,2,0], [0,1,2]])
A.dot(B)
```

```
array([[15,  1, -2],
       [78, 16, 24],
       [ 0,  3,  6],
       [36,  0, 16]])
```

2.3.5 Zadanie

Policz na tablicy/kartce przykład mnożenia macierz-macierz w dwu krokach.

2.3.6 Zadanie

Przygotuj odpowiedni przykład demonstrujący omówioną technikę. Przetestuj kod.

2.4 Macierz · macierz — pojedynczy krok MapReduce (30 pkt)

2.4.1 Zadanie

Zaprojektuj, zaimplementuj oraz przetestuj algorytm mnożenia macierzy, który wykorzystuje tylko pojedynczy krok Map Reduce.