

Rozwiązanie układu równań liniowych iteracyjną metodą Richardsona Sprawozdanie, Etap I

Kacper Górka, Krzysztof Rudnicki, Aleksandra Sobala

9 grudnia 2024

1 Zadanie

Metoda Richardsona Metoda Richardsona służy do iteracyjnego rozwiązywania systemów równań liniowych postaci $Ax = b$.
Pojedyncza iteracja wygląda następująco:

$$x^{(k+1)} = x^{(k)} + \omega(b - Ax^{(k)})$$

Gdzie ω to skalar wybrany tak by $x^{(k)}$ zbiegało

Wymagania Mieliśmy za zadanie stworzyć program rozwiązujący układ równań dla wygenerowanych macierzy gęstych oraz dla macierzy rzadkich:
[nemeth12](#)
i
[poli3](#)

2 Baza

Generowanie i zapisywanie macierzy Macierze gęste są przez nas generowane przy użyciu biblioteki **numpy**, aby przyspieszyć obliczenia zapewniamy **bewzględna dominację wierszową głównej przekątnej** i upewniamy się że wygenerowana macierz jest **symetryczna i dodatnio określona**

Macierze są potem zapisywane do pliku w formacie .npz, łącznie z ich wartościami własnymi, tak by skrócić działanie programu i ujednolicić testy

Macierze nemeth12 i poli3 są pobierane ze strony podanej wyżej, dla macierzy nemeth12 aby spełnić warunki stosowalności metody musieliśmy przemnożyć ją przez -1

Testy Do testów wykorzystujemy biblioteki **numpy** oraz **pytest** oraz wbudowane w Pythona narzędzia do mierzenia czasu.

Sprawdzamy poprawność naszych algorytmów poprzez porównanie naszych wyników z wynikami policzonymi przy wykorzystaniu funkcji `np.linalg.norm` z biblioteki `numpy`. Jeżeli nasze rozwiązanie różni się od rozwiązania `numpy` o mniej niż 8×10^{-3} akceptujemy je jako poprawne

Zarówno wielkość macierzy, jej typ i typ metody użytej do zrównoleglenia Richardsona jest podawana jako parametr testów, pozwala nam to łatwo dodawać nowe metody zrównoleglenia bez zmiany kodu testów.

Funkcje pomocnicze Wszelkie podstawowe metody operacji na macierzach takie jak mnożenie wektorów, macierzy itp, napisaliśmy od zera, bez użycia zewnętrznych bibliotek, funkcje są zdefiniowane w pliku **linear_algebra_utils.py**

Metoda Richardsona Metoda Richardsona jest zaimplementowana w pliku **richardson_method.py**, sprowadza się ona do pętli:

```
1     for iteration in range(self.max_iterations):
2         Ax = self.LinAlg.matrix_vector_multiply(self.A, x)
3         residual = self.LinAlg.vector_vector_subtraction(
4             self.b, Ax)
5         x = self.LinAlg.vector_vector_addition(
6             x,
7             self.LinAlg.scalar_vector_multiply(self.omega,
8                 residual)
9         )
10    if self.LinAlg.SequentialLinearAlgebraUtils.
11        vector_norm(residual) < self.tol:
12        break
```

Listing 1: Python Code for Iterative Solver

Dla różnych metod zrównoleglenia stosujemy różne implementacje podstawowych funkcji odpowiedzialnych za mnożenie macierzy przez wektor, odejmowanie wektorów itp. Ponownie, dzięki temu możemy łatwo dodawać nowe metody zrównoleglenia bez zmiany podstawowego kodu Richardsona

3 Zrównoleglenie

Wykorzystaliśmy 3 metody zrównoleglenia:

1. Tablice rozproszone
2. Wątki
3. Procesy

3.1 Procesy

Aby wykonać obliczenia na wielu rdzeniach procesora **jednocześnie** wykorzystujemy model w którym różne frakcje danych są przetwarzane przez oddzielne procesy. Dzięki temu dla dużych zbiorów danych możemy znacznie zwiększyć wydajność obliczeń

W tym celu wykorzystujemy klasę **multiprocessing.Pool** z biblioteki **multiprocessing**. Wykorzystujemy ją do stworzenia puli procesów które potem niezależnie wykonują funkcje na różnych frakcjach danych.

Funkcje Procesy wykorzystujemy do:

1. Obliczenia iloczynu skalarnego - Metoda *dot_product* wykorzystuje pulę procesów do obliczenia iloczynów par elementów dwóch wektorów, a następnie sumuje te wyniki. Zrównoleglenie tej operacji jest korzystne, gdy mamy do czynienia z bardzo długimi wektorami.
2. Mnożenia macierzy przez wektor - *matrix_vector_multiply*, każdy wiersz macierzy jest mnożony przez wektor w osobnym procesie. Dzięki temu każde takie mnożenie może być przeprowadzane równolegle, co jest szczególnie efektywne dla macierzy o dużym rozmiarze.
3. Obliczenia normy wektora - Procesy są używane do obliczenia kwadratów poszczególnych elementów wektora, a następnie sumowanie tych

wartości (także w procesach) umożliwia obliczenie pierwiastka kwadratowego z ich sumy, co daje normę wektora.

4. Działania na wektorach i macierzach - Działania takie jak dodawanie i odejmowanie wektorów, dzielenie wektora przez skalar, czy mnożenie macierzy przez skalar, są przeprowadzane w segmentach, gdzie każdy segment jest przetwarzany przez osobny proces.

Wyzwania

- Zarządzanie procesami jest kosztowne - tworzenie i zarządzanie procesami jest droższe od wątków ze względu na większy narzut systemowy.
- Wymiana danych między procesami - wymaga serializacji i deserializacji danych, co może wprowadzić dodatkowe opóźnienia.
- Brak korzyści dla małych danych - w przypadku małych macierzy, gdzie rozmiar nie przekracza 5 tysięcy x 5 tysięcy elementów, zarządzanie procesami i koszty komunikacji międzyprocesowej mogą przewyższać korzyści wynikające z równoległego przetwarzania,

3.2 Wątki

Do implementacji wątków użyto dwóch bibliotek, `ThreadPoolExecutor` która umożliwia zarządzanie pulą wątków i delegowanie zadań do wątków w sposób równoległy oraz funkcji `partial` z biblioteki `functools` która pozwala na tworzenie częściowo zainicjalizowanych funkcji.

Funkcje Wątki zaimplementowano w mnożeniu macierzy przez wektor, odejmowaniu wektorów, dodawaniu wektorów oraz mnożeniu wektora przez skalar, metody zostają zrównoleglone poprzez podzielenie liczby wierszy macierzy między wątki, następnie `ThreadPoolExecutor` tworzy wątki i przekazuje im odpowiednie, niezależne części pracy do wykonania.

Zalety Zalety wykorzystania wątków to przede wszystkim szybki czas tworzenia i niszczenia wątków przez system operacyjny w porównaniu do procesów. Co więcej mają one dostęp do całej przestrzeni adresowej programu, co oszczędza niepotrzebne kopiowanie danych. Jedynie ich własny stos jest prywatny.

3.3 Tablice rozproszone

Tablice rozproszone dzielą macierz i przypisują każdą z jej części do konkretnego procesora. Procesory wykonują obliczenia na danych przechowywanych w ich lokalnej pamięci, co minimalizuje konieczność przesyłania danych pomiędzy węzłami.

Tablice rozproszone nie są natywnie wspierane przez Python-a, w związku z tym zostały zaimplementowane przy użyciu modułu array z biblioteki **dask**. Wszystkie podstawowe funkcje wykorzystywane w Richardsonie zostały zrównoleglone przy użyciu tablic rozproszonych.

Wyzwania

- Przy dużej zależności danych dochodzi do częstej komunikacji która obniża wydajność
- Elementy tablicy należy dobrze zbalansować aby procesory były równo obciążone

3.4 Wyniki

| Wielkość/Typ | Sekwencyjnie [s] | Procesy [s] | Wątki [s] | Tablice [s] |
|--------------|------------------|-------------|------------|-------------|
| 2 | 7.784e-05 | 2.896e+00 | 9.772e-03 | 8.817e-02 |
| 5 | 1.746e-04 | 3.897e+00 | 1.960e-02 | 9.443e-02 |
| 10 | 6.769e-04 | 7.073e+00 | 2.895e-02 | 1.674e-01 |
| 50 | 2.735e-02 | 2.153e+01 | 1.059e-01 | 4.899e-01 |
| 100 | 1.195e-01 | 2.167e+01 | 2.067e-01 | 6.921e-01 |
| 300 | 7.863e-01 | 2.363e+01 | 9.558e-01 | 7.461e-01 |
| 500 | 2.206e+00 | 2.657e+01 | 2.494e+00 | 8.521e-01 |
| 750 | 4.785e+00 | 2.939e+01 | 5.520e+00 | 9.408e-01 |
| 1000 | 8.689e+00 | 3.259e+01 | 9.672e+00 | 1.201e+00 |
| 5000 | 2.170e+02 | 9.077e+01 | 2.402e+02 | 1.368e+01 |
| 10000 | 8.615e+02 | 2.378e+02 | 9.705e+02 | 4.643e+01 |
| nemeth12 | 3.630e+02 | 1.105e+02 | 3.863e+02s | 2.133e+01 |

Tabela 1: Wyniki dla różnych zrównoleżeń (procesy, wątki i tablice rozproszone)

Wyniki dla różnych zrównolegnień

