Rozwiązanie układu równań liniowych iteracyjną metodą Richardsona Sprawozdanie, Etap I

Kacper Górka, Krzysztof Rudnicki, Aleksandra Sobala

13 stycznia 2025

1 Zadanie

Metoda Richardsona Metoda Richardsona służy do iteracyjnego rozwiązywania systemów równań liniowych postaci $Ax = b$.
Pojedyńcza 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 przyśpieszyć 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 popawność 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 wielkośc 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(
              self.b, Ax)
4         x = self.LinAlg.vector_vector_addition(
5             x,
6             self.LinAlg.scalar_vector_multiply(self.omega,
                  residual)
7         )
8         if self.LinAlg.SequentialLinearAlgebraUtils.
              vector_norm(residual) < self.tol:
9             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 wydajnośc 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 przeez 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.

ChatGPT Po konsultacjach z prowadzącym użyliśmy chata-gpta do wygenerowania kodu dla wątków, rozwiązanie chata po kilku (ludzkich) poprawkach zadziałało ale jego dokładność była niesatysfakcjonująca, nie spełniała wymogów naszych testów

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 rozporoszone 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 wydajnośc
- Elementy tablicy należy dobrze zbalansować aby procesory były równo obciążone

3.4 MPI

Metoda zrównoleglenia przy użyciu MPI (Message Passing Interface) zrównolegla obliczenia na wielu procesorach

3.4.1 Podział pracy

Rozdzielenie macierzy i wektora Macierz A i wektor b są dzielone na części, które każdy procesor przetwarza indywidualnie. Każdy proces obsługuje określony zakres wierszy macierzy A i odpowiadającą im część wektora b.

Residuum r Każdy proces oblicza swoją lokalną wartość residuum r, czyli różnicę pomiędzy obliczonym a rzeczywistym wynikiem.

3.4.2 Synchronizacja danych

Rozgłoszenie wspólnych wartości Parametry takie jak współczynnik relaksacji (omega) są obliczane na jednym procesie (zwykle procesie 0) i rozsyłane do wszystkich procesów za pomocą funkcji **bcast**.

Sumowanie wyników Po obliczeniu lokalnego residuum przez każdy proces, dane te są sumowane w jedną globalną wartość przy użyciu funkcji **Allreduce**, co pozwala uwzględnić wkład wszystkich procesów

3.4.3 Iteracyjna aktualizacja

Równoległe aktualizowanie rozwiązania Wektor rozwiązania x jest aktualizowany równolegle na każdym procesie na podstawie globalnego residuum. Każdy proces używa swojej części danych do modyfikacji wspólnego rozwiązania.

Sprawdzenie warunku zbieżności Norma residuum (miara błędu) jest sprawdzana po każdej iteracji. Gdy osiągnie tolerancję, algorytm przerywa dalsze obliczenia.

3.4.4 Efektywność i skalowalność

Wykorzystanie procesów Procesy wykonują większość obliczeń równolegle, co znacząco zmniejsza czas potrzebny na rozwiązanie dużych układów równań.

Minimalizacja komunikacji Dane przesyłane między procesami są ograniczone do kluczowych informacji (np. residuum, parametry), co redukuje narzut związany z komunikacją.

3.4.5 Zalety

- Skalowalność: Algorytm można stosować na dużej liczbie procesorów, co umożliwia rozwiązywanie bardzo dużych układów.
- Wydajność: Dzięki podziałowi pracy, obliczenia są szybsze niż w wersji sekwencyjnej.

- Elastyczność: Można go zastosować do różnych typów macierzy i układów równań.

3.4.6 Wady

- Narzut komunikacyjny: Przy bardzo dużej liczbie procesorów koszt komunikacji może zniwelować zysk z równoleglenia.
- Obciążenie procesorów: Jeśli rozmiar macierzy nie dzieli się równomiernie między procesy, niektóre procesory mogą być mniej obciążone.

3.5 poli3 i duże macierze

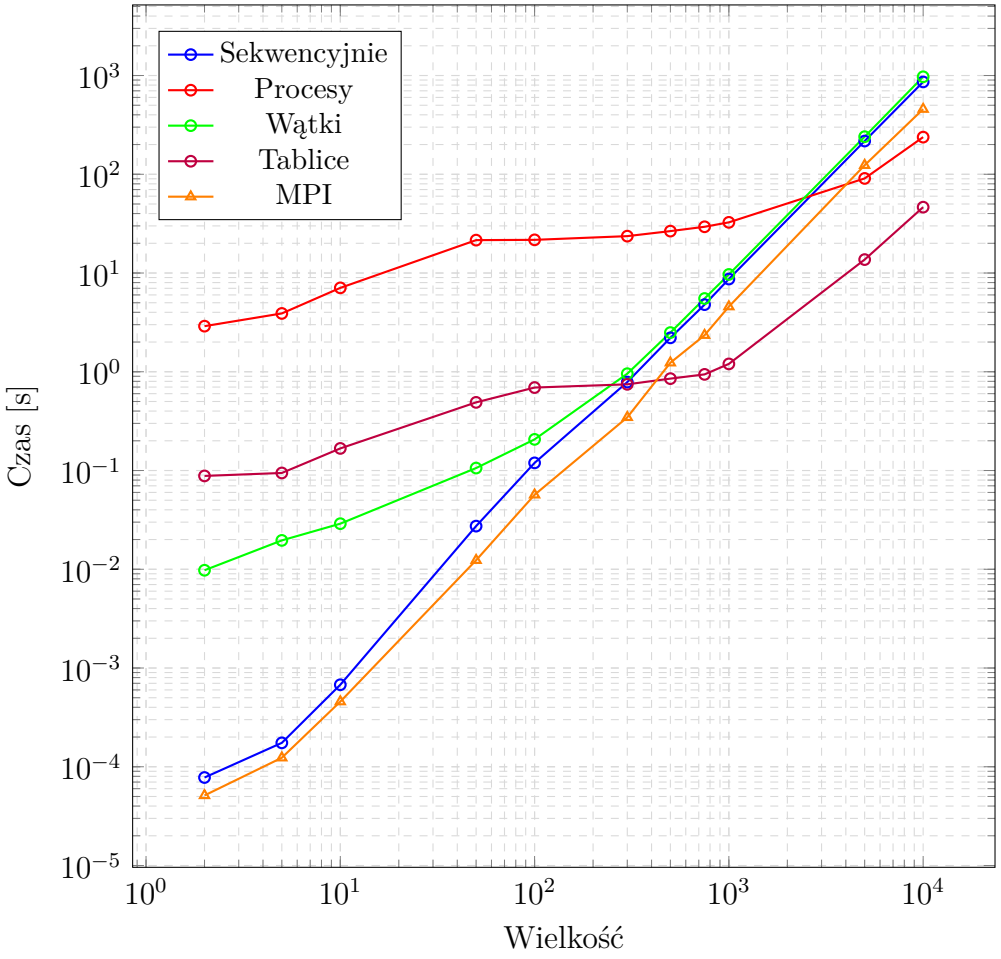
Rozwiązanie nie działa dla dużych macierzy, takich jak poli3, ze względu na ograniczenia związane z serializacją i komunikacją w bibliotekach MPI oraz sposób, w jaki macierze są rozgłaszane między procesami. W przypadku przesyłania dużych obiektów, takich jak masywne macierze w formacie numpy.ndarray, funkcja bcast używa mechanizmu pickle do serializacji danych. Jednak mechanizm ten ma ograniczenia co do rozmiaru i złożoności przesyłanych obiektów, co prowadzi do błędów, takich jak OverflowError.

3.6 Wyniki

Wielkość/Typ	**Sekwencyjnie** [s]	**Procesy** [s]	**Wątki** [s]	**Tablice** [s]	**MPI** [s]
2	7.784e-05	2.896e+00	9.772e-03	8.817e-02	1.737e-02
5	1.746e-04	3.897e+00	1.960e-02	9.443e-02	5.670e-04
10	6.769e-04	7.073e+00	2.895e-02	1.674e-01	1.766e-02
50	2.735e-02	2.153e+01	1.059e-01	4.899e-01	1.808e-02
100	1.195e-01	2.167e+01	2.067e-01	6.921e-01	1.946e-02
300	7.863e-01	2.363e+01	9.558e-01	7.461e-01	4.492e-02
500	2.206e+00	2.657e+01	2.494e+00	8.521e-01	9.526e-02
750	4.785e+00	2.939e+01	5.520e+00	9.408e-01	2.832e-01
1000	8.689e+00	3.259e+01	9.672e+00	1.201e+00	5.430e-01
5000	2.170e+02	9.077e+01	2.402e+02	1.368e+01	7.480e+01
10000	8.615e+02	2.378e+02	9.705e+02	4.643e+01	4.046e+02
nemeth12	3.630e+02	1.105e+02	3.863e+02	2.133e+01	4.427e+01
poli3	1.291e+03	1.187e+03	1.363e+03	7.029e+01	*N/A*

Tabela 1: Wyniki dla różnych rozmiarów i zrównolegleń (sekwencyjne, procesy, wątki, tablice rozproszone i MPI)

Wyniki dla różnych zrównolegleń



3.7 Przyśpieszenie według definicji z wykładu

$S(n,p) = \frac{T(n,1)}{T(n,p)}$

Wielkość/Typ	S(n,p) - Procesy	S(n,p) - Wątki	S(n,p) - Tablice	S(n,p) - MPI
2	2.69e-05	8.17e-03	8.83e-04	1.52e+00
5	4.48e-05	8.91e-03	1.85e-03	1.41e+00
10	9.57e-05	2.34e-02	4.04e-03	1.48e+00
50	1.27e-03	2.58e-01	5.58e-02	2.15e+00
100	5.52e-03	5.78e-01	1.73e-01	2.11e+00
300	3.33e-02	8.23e-01	1.05e+00	2.27e+00
500	8.30e-02	8.85e-01	2.59e+00	1.79e+00
750	1.63e-01	8.67e-01	5.08e+00	2.04e+00
1000	2.67e-01	8.98e-01	7.24e+00	1.90e+00
5000	2.39e+00	9.04e-01	1.59e+01	1.76e+00
10000	3.62e+00	8.88e-01	1.86e+01	1.89e+00
nemeth12	3.28e+00	9.40e-01	1.70e+01	1.95e+00
poli3	1.09e+00	9.47e-01	1.84e+01	1.85e+00

Wnioski Zrównoleglenie metodą tablic rozproszonych okazało się najbardziej efektywne. Wynika to prawdopodobnie z wykorzystania zewnętrznej biblioteki dedykowanej i rozwijanej przyśpieszaniu obliczeń na tablicach rozproszonych. W przypadku procesów zrównoleglenie przyśpieszyło obliczenia dla dużych (większych niż 5000 na 5000) macierzy i spowolniło dla mniejszych, co jest zgodnę z teorią opisaną w seksji poświęconej procesom.
Dla wątków nie udało się uzyskać przyśpieszenia ani dla małych ani dla dużych macierzy, podejrzewamy że jest to spowodowane przez nieefektywne zarządzanie wątkami przez Python-a