Dokumentacja Algorytmu Współbieżnego

Drobnoziarnista Eliminacja Gaussa-Jordana w oparciu o Teorię Śladów

Jan Dyląg

15 grudnia 2025

Spis treści

1 Wstęp Teoretyczny

1.1 Cel i Motywacja

Klasyczne podejście do zrównoleglania algorytmu Gaussa-Jordana często opiera się na dekompozycji wierszowej (ang. *row-level*), gdzie najmniejszą jednostką obliczeniową jest operacja na całym wektorze. Choć skuteczne, podejście to nie wykorzystuje pełnego potencjału równoległości dostępnego w strukturze macierzowej.

W niniejszym projekcie zastosowano podejście **drobnoziarniste** (ang. *fine-grained* lub *cell-level*). Oznacza to, że każda elementarna operacja arytmetyczna (mnożenie, odejmowanie, dzielenie) wykonywana na pojedynczej komórce macierzy $A[i][j]$ jest traktowana jako osobne zadanie w alfabecie śladu. Pozwala to na wygenerowanie znacznie gęstszego grafu zależności i teoretycznie maksymalne skrócenie czasu wykonania (minimalizację wysokości śladu).

1.2 Podstawy Matematyczne

Dla układu równań $Ax = b$, gdzie A jest macierzą $N \times N$, algorytm operuje na macierzy rozszerzonej $[A|b]$ o wymiarach $N \times (N + 1)$. Kolumna N-ta (indeksowana od 0) odpowiada wektorowi b.

Proces dla każdego pivota $k \in \{0, \ldots, N - 1\}$ składa się z dwóch faz:

1. **Normalizacja (N)**: Sprowadzenie elementu diagonalnego $A[k][k]$ do wartości 1 poprzez podzielenie całego wiersza k przez $A[k][k]$.
2. **Eliminacja (E)**: Wyzerowanie elementów w kolumnie k dla wszystkich wierszy $i \neq k$.

2 Instrukcja Obsługi i Środowisko Uruchomieniowe

Aby poprawnie przeprowadzić symulację działania algorytmu oraz zweryfikować jego wyniki, konieczne jest przygotowanie odpowiedniego środowiska plików tekstowych. Projekt został zaprojektowany do współpracy z zewnętrznym systemem generującym testy (tzw. „sprawdzarką").

2.1 Przygotowanie Danych Wejściowych

W katalogu głównym projektu (obok pliku `Gauss_Jordan.py`) należy utworzyć dwa pliki tekstowe, których zawartość pochodzi z zewnętrznego generatora:

1. `input_data.txt` – Plik zawierający dane wejściowe dla algorytmu.
   - Należy wygenerować test w sprawdzarce.
   - Skopiować zawartość sekcji wejściowej (zazwyczaj zawiera rozmiar macierzy N, elementy macierzy A oraz wektor b).
   - Wkleić zawartość do pliku `input_data.txt` i zapisać zmiany.
2. `solution.txt` – Plik zawierający wzorcowe rozwiązanie.
   - Z tego samego testu w sprawdzarce należy skopiować sekcję wyjściową/wynikową.
   - Zawiera ona oczekiwany rozmiar N, macierz jednostkową (jako dowód poprawnej eliminacji) oraz wynikowy wektor x.
   - Wkleić zawartość do pliku `solution.txt`. Plik ten jest niezbędny do automatycznej weryfikacji poprawności obliczeń przez program.

2.2 Uruchomienie i Wyniki

Program uruchamia się z poziomu terminala poleceniem:

```
python Gauss_Jordan.py
```

Po zakończeniu obliczeń program wykonuje następujące czynności:

- Wyświetla na konsoli informacje o postępie (liczba operacji, liczba klas Foaty).
- Generuje plik `my_result.txt` zawierający obliczony wynik sformatowany w sposób identyczny jak plik wzorcowy.
- Automatycznie porównuje plik `my_result.txt` z plikiem `solution.txt` i raportuje zgodność numeryczną (z dokładnością do 10^{-4}).
- Dla małych macierzy ($N \leq 5$) generuje plik graficzny `graf_komorkowy.png` z wizualizacją zależności.

3 Szczegóły Techniczne Implementacji

Program został zrealizowany w języku Python w paradygmacie programowania obiektowego. Główną strukturą sterującą jest klasa `GaussJordanFineGrained`, która realizuje proces obliczeniowy w dwóch odrębnych fazach: fazie teoretycznej (analiza śladu) i fazie praktycznej (wykonanie współbieżne).

3.1 Reprezentacja Zasobów i Operacji

W podejściu drobnoziarnistym kluczowym wyzwaniem jest identyfikacja zasobów. W programie zasób nie jest już całym wierszem, lecz konkretną komórką pamięci.

- **Identyfikator zasobu:** Krotka `(wiersz, kolumna)`. Dla wektora wyrazów wolnych b przyjęto umowny indeks kolumny równy N.
- **Słownik zasobów:** Struktura `self.op_resources` mapuje unikalną nazwę operacji (np. `"E_2_0_3"`) na dwa zbiory: `READ` (zbiór komórek odczytywanych) i `WRITE` (zbiór komórek modyfikowanych).

3.2 Faza Teoretyczna: Budowa Harmonogramu

Przed rozpoczęciem jakichkolwiek obliczeń numerycznych, program wykonuje symulację "na sucho":

1. **Generowanie Śladu:** Metoda `define_operations` iteruje po indeksach macierzy, generując sekwencję operacji w kolejności, w jakiej wykonałby je algorytm sekwencyjny.
2. **Analiza Konfliktów:** Metoda `build_dependency_graph` przegląda historię operacji. Dla każdej nowej operacji sprawdza, czy jej zbiory `READ`/`WRITE` mają część wspólną ze zbiorami wcześniejszych operacji. Jeśli tak, tworzona jest krawędź w grafie skierowanym (Graf Diekerta).
3. **Wyznaczanie Warstw (Klas Foaty):** Na podstawie grafu zależności obliczana jest "głębokość"każdej operacji. Operacje o tej samej głębokości są grupowane w listy zwane Klasami Foaty. Reprezentują one grupy zadań, które są matematycznie niezależne i mogą być wykonane równolegle.

3.3 Faza Praktyczna: Silnik Współbieżny

Do wykonania obliczeń wykorzystano bibliotekę `concurrent.futures`.

- **Pula Wątków:** Tworzony jest obiekt `ThreadPoolExecutor`. Liczba wątków jest skalowana w zależności od rozmiaru macierzy, aby obsłużyć dużą liczbę drobnych zadań.
- **Iteracja po Klasach:** Program iteruje po liście Klas Foaty. Jest to kluczowy mechanizm synchronizacji.
- **Bariera:** Wewnątrz pętli, wszystkie operacje z bieżącej klasy są zlecane do puli wątków (`executor.submit`). Następnie program główny czeka na zakończenie wszystkich zadań z tej klasy (`as_completed`), zanim przejdzie do następnej. Działa to jak bariera synchronizacyjna, gwarantująca, że np. normalizacja pivota zakończy się przed rozpoczęciem eliminacji w innych wierszach.

Dzięki takiemu podejściu, logika synchronizacji (kto na kogo czeka) wynika bezpośrednio z teorii śladów, a nie z ręcznego ustawiania blokad (mutexów) w kodzie, co eliminuje ryzyko zakleszczeń (deadlocks).

4 Specyfikacja Algorytmu Drobnoziarnistego

W podejściu komórkowym operacje wierszowe zostały rozbite na operacje atomowe. Poniżej przedstawiono zdefiniowany alfabet operacji.

4.1 Definicja Alfabetu Operacji

4.1.1 1.Normalizacja Komórkowa ($N_{k,j}$)

Operacja normalizacji wiersza k została rozbita na serię niezależnych dzieleń dla każdej kolumny $j \geq k$.

- **Symbol:** $N_{k,j}$ (gdzie k to wiersz pivota, j to aktualna kolumna).
- **Działanie:** $A[k][j] \leftarrow A[k][j]/A[k][k]$.
- **Zasoby:**
  - Odczyt: Komórka (k, k) (wartość pivota).
  - Zapis: Komórka (k, j) (aktualizowana wartość).

Szczególny przypadek: Operacja $N_{k,k}$ (normalizacja samego pivota) musi być wykonana *po* znormalizowaniu reszty wiersza, aby inne wątki nie odczytały przedwcześnie wartości 1.0. W implementacji jest ona dodawana do śladu na końcu sekwencji normalizacji dla danego k.

4.1.2 2.Eliminacja Komórkowa ($E_{i,k,j}$)

Operacja eliminacji wiersza i przy użyciu wiersza k została rozbita na aktualizację poszczególnych komórek $A[i][j]$.

- **Symbol:** $E_{i,k,j}$ (wiersz i, pivot k, kolumna j).
- **Działanie:** $A[i][j] \leftarrow A[i][j] - A[i][k] \times A[k][j]$.
- **Zasoby:**

– Odczyt: Komórka (i, k) (mnożnik wiersza eliminowanego).
– Odczyt: Komórka (k, j) (wartość z wiersza źródłowego).
– Zapis: Komórka (i, j) (aktualizowana wartość).

Szczególny przypadek: Operacja $E_{i,k,k}$ (zerowanie kolumny pivota) polega na jawnym przypisaniu $A[i][k] \leftarrow 0$. Musi się odbyć po wykorzystaniu wartości $A[i][k]$ jako mnożnika dla reszty wiersza.

5 Analiza Zależności i Implementacja

5.1 Zarządzanie Zasobami (Resource Management)

W algorytmie śladowym kluczowe jest zdefiniowanie, co jest zasobem. W wersji drobnoziarnistej zasobem jest **pojedyncza komórka macierzy**, identyfikowana przez parę współrzędnych (row, col).

Dla każdej operacji zdefiniowano zbiory `READ` i `WRITE`. Relacja zależności (konfliktu) zachodzi między dwiema operacjami u i v, jeśli:

$$(D_u \cap R_v \neq \emptyset) \vee (R_u \cap D_v \neq \emptyset) \vee (D_u \cap D_v \neq \emptyset) \tag{1}$$

gdzie D to domena zapisu (Domain), a R to domena odczytu (Range).

5.2 Przykład Konfliktu (Write-Read)

Rozważmy relację między normalizacją pivota $N_{0,0}$ a eliminacją w kolejnym wierszu $E_{1,0,0}$.

- $N_{0,0}$ pisze do $(0, 0)$ (ustawia 1.0).
- $E_{1,0,j}$ czyta $(0, 0)$ jako pivot.

Dzięki analizie historii operacji, algorytm automatycznie wykrywa, że $E_{1,0,j}$ musi czekać na zakończenie $N_{0,0}$.

5.3 Wyznaczanie Klas Foaty

Algorytm wykorzystuje postać normalną Foaty do szeregowania operacji.

1. Budowany jest graf skierowany zależności (Graf Diekerta), gdzie węzłami są operacje komórkowe.
2. Wyznaczana jest najdłuższa ścieżka do każdego węzła.
3. Operacje o tej samej długości ścieżki od źródła trafiają do jednej klasy Foaty.

Klasy te są wykonywane sekwencyjnie, ale wszystkie operacje wewnątrz jednej klasy są wykonywane w pełni równolegle.

6 Szczegóły Implementacyjne

6.1 Struktura Klasy `GaussJordanFineGrained`

Główna klasa programu przechowuje:

- `op_resources`: Słownik mapujący nazwę operacji na zbiory odczytywanych i zapisywanych komórek.
- `foata_classes`: Listę list operacji, reprezentującą harmonogram wykonania.

6.2 Mechanizm Wykonawczy (Executor)

Do współbieżnego wykonania wykorzystano `concurrent.futures.ThreadPoolExecutor`.

- Liczba wątków (*workers*) jest skalowana dynamicznie w zależności od rozmiaru problemu (N^2), aby obsłużyć dużą liczbę drobnych zadań.
- Bariera synchronizacyjna jest implementowana poprzez oczekiwanie na zakończenie wszystkich obiektów `Future` z danej klasy Foaty przed przejściem do następnej.

6.3 Obsługa Wejścia/Wyjścia

Algorytm jest zintegrowany z systemem plików:

- `input_data.txt`: Źródło danych (macierz A, wektor b).
- `solution.txt`: Wzorzec do weryfikacji.
- `my_result.txt`: Wynik działania programu.

Dodatkowo, dla małych instancji ($N \leq 5$), generowana jest wizualizacja grafu zależności (`graf_komorkowy.png`), co pozwala na empiryczną weryfikację poprawności generowanych zależności.

7 Analiza Kluczowych Fragmentów Implementacji

W tej sekcji przedstawiono i omówiono najważniejsze fragmenty kodu źródłowego, które odpowiadają za realizację logiki drobnoziarnistej (komórkowej) oraz mechanizmów współbieżności.

7.1 Definicja Operacji i Zasobów Komórkowych

W przeciwieństwie do klasycznego podejścia wierszowego, w tym algorytmie zasobem jest pojedyncza komórka macierzy. Poniższy fragment metody `define_operations` pokazuje, jak definiowana jest operacja normalizacji dla konkretnej komórki $A[k][j]$.

```
# Fragment metody define_operations
# Normalizacja komorki (k, j) przy uzyciu pivota (k, k)

op_name = f"N_{k}_{j}"
self.alphabet.append(op_name)
self.trace.append(op_name)

# Definicja zasobow:
# 'read': Odczytujemy wartosc pivota z komorki (k, k)
# 'write': Nadpisujemy wartosc w komorce (k, j)
self.op_resources[op_name] = {
    'read': {(k, k)},
    'write': {(k, j)}
}
```

Listing 1: Definicja atomowej operacji normalizacji

Kluczowym elementem jest tutaj wykorzystanie krotek `(wiersz, kolumna)` jako identyfikatorów zasobów. Dzięki temu system wie, że operacja $N_{0,1}$ jest niezależna od $N_{0,2}$ (piszą do różnych komórek), mimo że obie czytają ten sam pivot.

7.2 Implementacja Warunków Bernsteina

Sercem algorytmu wyznaczającego graf zależności jest metoda `build_dependency_graph`. Sprawdza ona historyczne konflikty dostępu do pamięci, implementując tzw. warunki Bernsteina.

```
# Fragment metody build_dependency_graph
dependent = False

# 1. Write-Read (True Dependency) - Uzycie wyniku poprzedniej operacji
if not res_prev['write'].isdisjoint(res_curr['read']):
    dependent = True

# 2. Read-Write (Anti Dependency) - Nadpisanie danych uzywanych przez inna operacje
elif not res_prev['read'].isdisjoint(res_curr['write']):
    dependent = True

# 3. Write-Write (Output Dependency) - Nadpisanie tego samego zasobu
elif not res_prev['write'].isdisjoint(res_curr['write']):
    dependent = True

if dependent:
    self.dependencies[op_curr].add(op_prev)
```

Listing 2: Wykrywanie zaleznosci miedzy operacjami

Użycie metody `isdisjoint` na zbiorach (Set) w Pythonie pozwala na błyskawiczne sprawdzenie, czy dwie operacje rywalizują o tę samą komórkę pamięci.

7.3 Współbieżne Wykonanie z Barierą

Metoda `solve` realizuje fizyczne wykonanie obliczeń. Wykorzystuje ona wyznaczone wcześniej Klasy Foaty jako naturalne punkty synchronizacji (bariery).

```
# Fragment metody solve
with concurrent.futures.ThreadPoolExecutor(max_workers=max_threads) as executor:
    # Iteracja po warstwach (klasach Foaty)
    for cls in self.foata_classes:
        futures = []

        # Zlecenie wszystkich zadan z biezacej klasy do watkow
        for op in cls:
            # Parsowanie nazwy operacji i uruchamianie funkcji
            # ... (kod parsowania pominiety) ...
            futures.append(executor.submit(self.execute_eliminate_cell, ...)
    )

        # BARIERA SYNCHRONIZACYJNA
        # Program czeka, az WSZYSTKIE watki z tej klasy zakoncza prace,
        # zanim przejdzie do nastepnej klasy (kolejnej iteracji petli).
        for f in concurrent.futures.as_completed(futures):
            f.result()
```

Listing 3: Silnik wspolbiezny oparty na Klasach Foaty

Ten fragment kodu gwarantuje poprawność matematyczną. Na przykład: żadna operacja eliminacji (z następnej klasy) nie rozpocznie się, dopóki wszystkie niezbędne normalizacje (z obecnej klasy) nie zostaną zakończone i zatwierdzone w pamięci.

8 Podsumowanie i Wnioski

Zastosowanie podejścia drobnoziarnistego skutkuje znacznym wzrostem liczby operacji w alfabecie (złożoność $O(N^3)$ w porównaniu do $O(N^2)$ dla wersji wierszowej).

Zalety:

- Maksymalizacja teoretycznego stopnia zrównoleglenia.
- Możliwość równoległego wykonywania operacji wewnątrz tego samego wiersza.

Wady:

- Znaczny narzut czasowy (overhead) związany z analizą zależności i zarządzaniem tysiącami małych zadań w języku Python.
- Graf zależności staje się bardzo gęsty i trudny do wizualizacji dla dużych N.

Algorytm poprawnie rozwiązuje układy równań, co potwierdzono weryfikacją numeryczną. Stanowi on doskonały przykład akademicki ilustrujący teorię śladów Mazurkiewicza w zastosowaniach numerycznych.

Zalecam użycia macierzy (NxN) małych rozmiarów np. N=5, żeby algorytm działał sprawnie i budowanie grafu diekerta nie zajęło dużo czasu. Warto na koniec wspomnieć, że cały algorytm nie jest wydajny/najszybszy ze względu na budowanie rozległego grafu diekerta, zauważyłem, że istnieje szybsze rozwiązanie, które jest mniej drobnoziarniste i wykorzystuje operację na całych wierszach, a nie na pojedynczych komórkach. Rozwiązanie wykorzystujące wiersze radziło sobie z macierzami rozmiaru N=80, podczas gdy podejście komórkowe na rozmiarze N=20 ma już problem.