Programowanie równoległe i rozproszone - Projekt zespołowy - Etap 1.

Adrian Cieśla, Mateusz Król, Bartłomiej Papis, Bartosz Wużyński

Styczeń 2026

1 Wstęp

W niniejszej pracy nasz zespół skupił się na rozwiązaniu zadania 11., polegającego na faktoryzacji Doolittle'a. Przypomnijmy, elementy macierzy dolnej lewostronnej L i górnej prawostronnej U takich że $A = LU$ są obliczane w nastepujący sposób k – numer iteracji:

$$u_{kj} = a_{kj} - \sum_{m=1}^{k-1} l_{km} u_{mj}, j = k, k+1, ..., n$$

$$l_{ik} = \frac{a_{ik} - \sum_{m=1}^{k-1} l_{im} u_{mk}}{u_{kk}}, i = k+1, ..., n$$

Macierz A powinna być generowana w sposób losowy. Poprawność rozwiązania powinna być liczona z wykorzystaniem normy Frobeniusa.

Praca pokrywa tylko i wyłącznie 1. etap projektu. Zadanie rozwiązane zostało z wykorzystaniem programu napisanego w języku C. Napisana została wersja sekwencyjna rozwiązania oraz powiązane z nią wersje zrównoleglające z wykorzystaniem:

- wątków współbieżnych
- procesów współbieżnych

W poniższych rozdziałach każda z tych metod zostanie dokładniej opisana.

2 Wątki współbieżne

Do implementacji rozwiązania z wykorzystaniem wątków współbieżnych wykorzystana została biblioteka OpenMP udostępniająca proste do zrozumienia przez programistę dyrektywy, które można wstawić w dowolnym miejscu w kodzie celem zrównoleglenia niektórych operacji. W przypadku faktoryzacji Doolittle'a, w każdej iteracji algortymu były dwa takie miejsca: przed obliczeniem

wiersza U oraz przed obliczeniem kolumny L. Zarówno wiersz U, jak i kolumna L obliczane są z wykorzystaniem pętli for, którą można zrównoleglić używając polecenia OpenMP:

```
#pragma omp parallel for
```

Konieczne okazało się również wykorzystanie polecenia:

```
private(m)
```

aby każdy wątek miał swoją prywatną kopię zmiennej m, zapobiegając dzięki temu warunkowi wyścigu (ang. race condition), oraz polecenia:

```
#pragma omp barrier
```

aby zapobiec sytuacji, w której w jednej iteracji wątek A celem obliczenia wartości elementu L próbuje pozyskać wartość elementu macierzy U, która nie została jeszcze obliczona przez wątek B, przez co końcowy wynik całego rozwiązania mógłby być błędny. Zamiast tego w każdej iteracji zarówno po obliczeniu wiersza U, jak i po obliczeniu kolumny L, wątki są synchronizowane z wykorzystaniem bariery.

2.1 Porównanie z wersją sekwencyjną

W niniejszej części porównamy czasy wykonania programu w wersji sekwencyjnej oraz w wersji z wątkami współbieżnymi. Niech

- n - rozmiar macierzy A
- p - liczba rdzeni oraz wątków wykorzystanych przy wykonywaniu programu
- $T(n, 1)$ - czas wykonania programu dla danego n oraz 1 rdzenia i wątku (wersja sekwencyjna)
- $T(n, p)$ - czas wykonania programu dla danego n oraz p rdzeni i wątków (wersja wątkowa)
- $S(n, p)$ - przyspieszenie

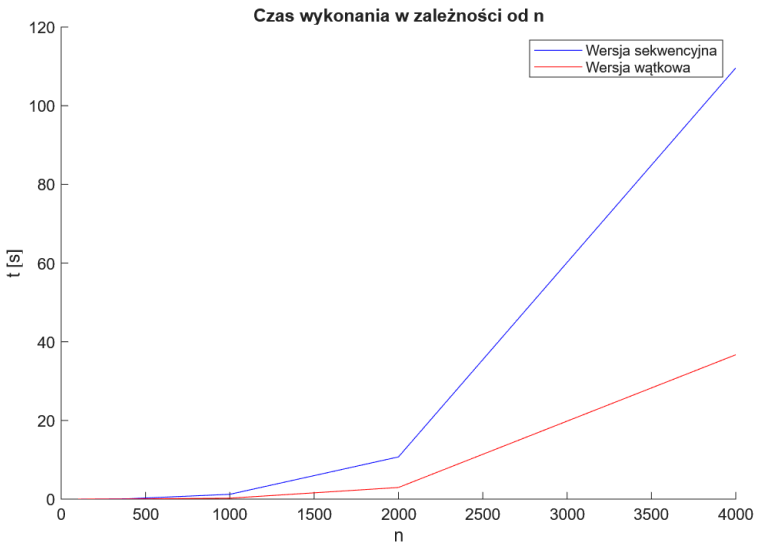
Wówczas poniższa tabela przedstawia wyniki eksperymentu przeprowadzonego w ramach niniejszego projektu dla wątków współbieżnych:

Tabela 1: Wyniki czasów działania algorytmu w wersji sekwencyjnej i współbieżnej oraz przyspieszenia w zależności od liczby rdzeni/wątków i n.

n	p	$T(n,1)[s]$	$T(n,p)[s]$	$S(n,p)$
100	1	0,002912	0,002988	0,974565
100	2	0,002912	0,002721	1,070195
100	4	0,002912	0,002014	1,445879
100	8	0,002912	0,001877	1,551412
400	1	0,092757	0,069820	1,328516
400	2	0,092757	0,052868	1,754502
400	4	0,092757	0,037478	2,474972
400	8	0,092757	0,034393	2,696973
1000	1	1,219420	1,192212	1,022821
1000	2	1,219420	0,923796	1,320010
1000	4	1,219420	0,477068	2,556072
1000	8	1,219420	0,233221	5,228603
2000	1	10,725878	10,554637	1,016224
2000	2	10,725878	5,972981	1,795733
2000	4	10,725878	3,041107	3,526965
2000	8	10,725878	2,955857	3,628686
4000	1	109,607238	108,273892	1,012315
4000	2	109,607238	54,966830	1,994061
4000	4	109,607238	36,692700	2,987167
4000	8	109,607238	45,629712	2,402102

Na podstawie powyższej tabeli możemy zauważyć, że zgodnie z intuicją wyniki czasów wykonania programu w wersji sekwencyjnej oraz czasów wykonania programu w wersji wątkowej dla 1 rdzenia i wątku są praktycznie identyczne, a przyspieszenie jest bardzo bliskie wartości 1. Największe uzyskane przyspieszenie w całym eksperymencie wyniosło $5,228603$ dla $n = 1000$ oraz $p = 8$. Z pewnością nie można powiedzieć, że poprawa wydajności po użyciu wątków współbieżnych z biblioteki OpenMP nie jest widoczna. Przyspieszenie bliskie 3 dla rozmiaru macierzy A równego 4000 może być zadowalającym wynikiem biorąc pod uwagę, że w wersji sekwencyjnej obliczenia dla tego rozmiaru trwają prawie 2 minuty. W każdym z badanych przypadków błąd poprawności rozwiązania w wersji wątkowej był niemal identyczny z błędem w wersji sekwencyjnej i najwyżej sięgał wartości rzędu 10^{-13}.

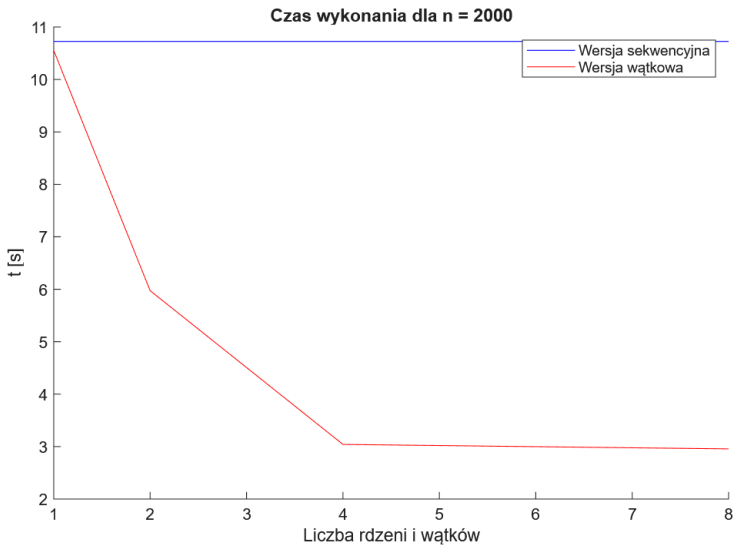
Przyjrzyjmy się teraz wykresom sporządzonym na podstawie powyższych danych. Poniżej zauważyć możemy zależność pomiędzy czasem wykonania wersji sekwencyjnej, a czasem wykonania wersji wątkowej dla różnych n. Dla każdego n wzięty został najlepszy uzyskany czas spośród wszystkich kombinacji liczby rdzeni/wątków.



Rysunek 1: Wykres czasu wykonania programu w zależności od n dla wersji sekwencyjnej i wątkowej.

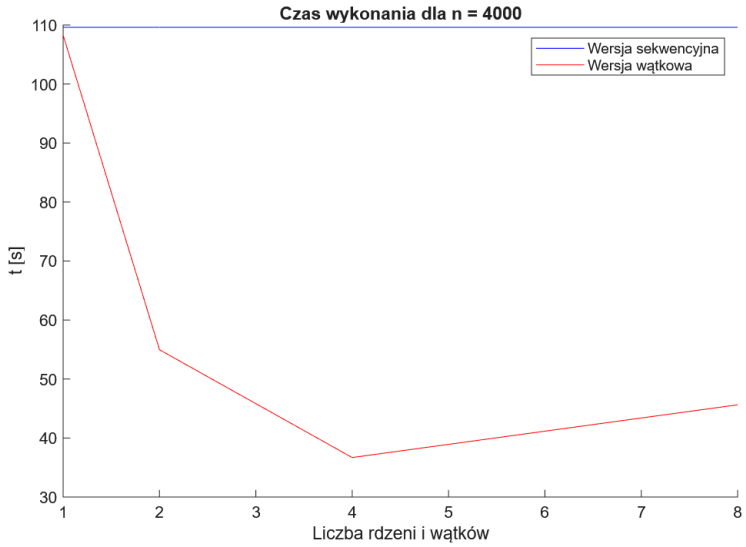
Różnica w czasach wykonania zaczyna być znacząca od $n = 2000$ i wynosi wówczas około 5 sekund. Czas wykonania dla wersji wątkowej przy $n = 4000$ jest już o około 1 minutę 20 sekund krótszy względem wersji sekwencyjnej. Można więc wysnuć wniosek, że wątki współbieżne z biblioteki OpenMP potrafią znacznie przyspieszyć działanie programu dla dużych n.

Warto również przeanalizować czas wykonania programu dla stałego n przy różnej liczbie rdzeni/wątków. Wykres takiej zależności dla $n = 2000$ znajduje się poniżej.



Rysunek 2: Wykres czasu wykonania programu w zależności od n dla wersji sekwencyjnej i wątkowej.

Zmiana liczby rdzeni i wątków nie ma większego znaczenia dla wersji sekwencyjnej, dlatego jej czas został ustalony na całym wykresie jako stały. Dla wersji wątkowej można zauważyć, że przy większej liczbie rdzeni i wątków czas wykonania programu znacząco się skraca. Początkowo przy wzroście z 1 do 2 rdzeni/wątków czas wykonania stanowił około 2 razy mniej względem wersji sekwencyjnej. Następnie przy wzroście z 2 do 4 rdzeni/wątków czas wykonania stanowił około 4 razy mniej względem wersji sekwencyjnej. Wzrost z 4 do 8 rdzeni/wątków natomiast nie przyniósł już efektu jakiego można było się spodziewać (8-krotnego przyspieszenia). Na poniższym wykresie otrzymujemy jeszcze gorszy efekt.



Rysunek 3: Wykres czasu wykonania programu w zależności od n dla wersji sekwencyjnej i wątkowej.

Dla $n = 4000$, czas wykonania programu dla wersji wątkowej jest jeszcze gorszy przy 8 rdzeniach/wątkach niż przy 4.

Oba powyższe przykłady pokrótce ilustrują działanie prawa Amdahla. Część programu zawsze pozostaje sekwencyjna, a więc nie da się go przyspieszać w nieskończoność. Im mocniej zrównoleglamy część równoległą, tym część sekwencyjna odgrywa większy udział w całkowitym czasie wykonania programu. Ponadto w przypadku biblioteki OpenMP, przy większej liczbie wątków rosną narzuty na synchronizację oraz zarządzanie zespołem wątków. To wszystko może więc wytłumaczać powyższe zachowania wykresów.

3 Procesy współbieżne

Wersja z procesami współbieżnymi została zrealizowana w języku C z wykorzystaniem modelu fork–join. Proces nadrzędny tworzy p procesów potomnych za pomocą wywołania `fork()`, po czym każdy proces potomny realizuje przydzieloną część obliczeń, a proces nadrzędny oczekuje na zakończenie pracy wszystkich potomków przez `waitpid()`. W odróżnieniu od wątków, procesy posiadają osobne przestrzenie adresowe, dlatego do współdzielenia danych konieczne było użycie mechanizmów IPC.

3.1 Pamięć współdzielona

Macierze A, L oraz U zostały umieszczone w pamięci współdzielonej utworzonej przy pomocy `shm_open()`, a następnie zmapowanej do przestrzeni adresowej

procesu przez `mmap()`. Dzięki temu każdy proces potomny ma dostęp do tych samych danych wejściowych i wyjściowych, mimo że standardowo nie współdzieli pamięci z innymi procesami. W pamięci współdzielonej umieszczono również struktury sterujące potrzebne do synchronizacji oraz ewentualnej sygnalizacji błędu pivota.

3.2 Synchronizacja

W faktoryzacji Doolittle'a występuje zależność danych pomiędzy dwiema fazami w ramach jednej iteracji k. Najpierw musi zostać obliczony wiersz macierzy U (w szczególności element u_{kk}), a dopiero potem można wyznaczać elementy kolumny macierzy L, które dzielą przez u_{kk}. Aby zapewnić poprawną kolejność obliczeń, zastosowano bariery synchronizujące procesy po zakończeniu fazy obliczania U oraz po zakończeniu fazy obliczania L w każdej iteracji.

Synchronizacja została zrealizowana przy użyciu semaforów POSIX (`sem_init`, `sem_wait`, `sem_post`) umieszczonych w pamięci współdzielonej. Każda bariera jest zbudowana z licznika przyjść oraz semafora zwalniającego, dzięki czemu dopiero ostatni proces, który dotrze do bariery, odblokowuje przejście pozostałych procesów.

3.3 Podział pracy

W każdej iteracji k możliwe jest równoległe wyznaczanie wielu elementów, ponieważ:

- wartości u_{kj} dla różnych j są niezależne w obrębie tej samej iteracji k,
- wartości l_{ik} dla różnych i są niezależne w obrębie tej samej iteracji k.

Z tego powodu zakres indeksów $j = k, \ldots, n-1$ w fazie obliczania U oraz zakres $i = k+1, \ldots, n-1$ w fazie obliczania L został dzielony pomiędzy procesy w sposób blokowy. Każdy proces oblicza własny podzakres indeksów, a następnie przechodzi przez barierę.

3.4 Obsługa sytuacji wyjątkowych

Ponieważ w wersji bez pivotowania element u_{kk} pełni rolę pivota, wprowadzono kontrolę jego wartości. Jeżeli $|u_{kk}|$ spada poniżej ustalonego progu, ustawiana jest flaga błędu w pamięci współdzielonej. Pozwala to przerwać obliczenia i zakończyć program komunikatem diagnostycznym zamiast generować wartości niepoprawne numerycznie.

3.5 Porównanie z wersją sekwencyjną

W niniejszej części porównamy wyniki czasów wykonania programu w wersji sekwencyjnej oraz w wersji z procesami współbieżnymi. Przyjmujemy te same oznaczenia co w rozdziale dotyczącym wątków.

Tabela 2: Wyniki czasów działania algorytmu w wersji sekwencyjnej i współbieżnej oraz przyspieszenia w zależności od liczby rdzeni/wątków i n.

n	p	$T(n,1)[s]$	$T(n,p)[s]$	$S(n,p)$
100	1	0,000620	0,001531	0,378094
100	2	0,000248	0,006093	0,027725
100	4	0,000256	0,013640	0,331341
100	8	0,000251	0,026275	0,009291
400	1	0,023125	0,023207	0,985649
400	2	0,022193	0,035450	0,601508
400	4	0,022788	0,060524	0,368133
400	8	0,023183	0,116082	0,192754
1000	1	0,429953	0,407111	1,023825
1000	2	0,497631	0,289684	1,523713
1000	4	0,407532	0,262931	1,545256
1000	8	0,506266	0,431443	1,132360
2000	1	9,214057	7,949702	1,59044
2000	2	8,775058	4,299117	2,041130
2000	4	9,248994	3,314077	2,790821
2000	8	8,627136	3,985813	2,164460
4000	1	177,128880	163,337757	1,084433
4000	2	174,256576	93,844849	1,856858
4000	4	176,277225	59,724300	2,951516
4000	8	170,509996	58,632949	2,780933

Na podstawie wyników z tabeli można stwierdzić, że wariant z procesami współbieżnymi jest nieefektywny dla małych rozmiarów macierzy, gdzie przyspieszenie $S(n,p)$ przyjmuje wartości mniejsze od 1. Wynika to z narzutu związanego z tworzeniem procesów, inicjalizacją pamięci współdzielonej oraz synchronizacją. Wraz ze wzrostem rozmiaru macierzy koszt obliczeń zaczyna dominować nad narzutem, co prowadzi do uzyskania rzeczywistego przyspieszenia dla $n \geqslant 1000$. Najlepsze wyniki uzyskano dla dużych macierzy i umiarkowanej liczby procesów, przy czym dalsze zwiększanie liczby procesów nie zawsze skutkuje proporcjonalnym wzrostem przyspieszenia z uwagi na koszty synchronizacji i ograniczenia pamięci.