Równoległe mnożenie macierzy

marwyk2003

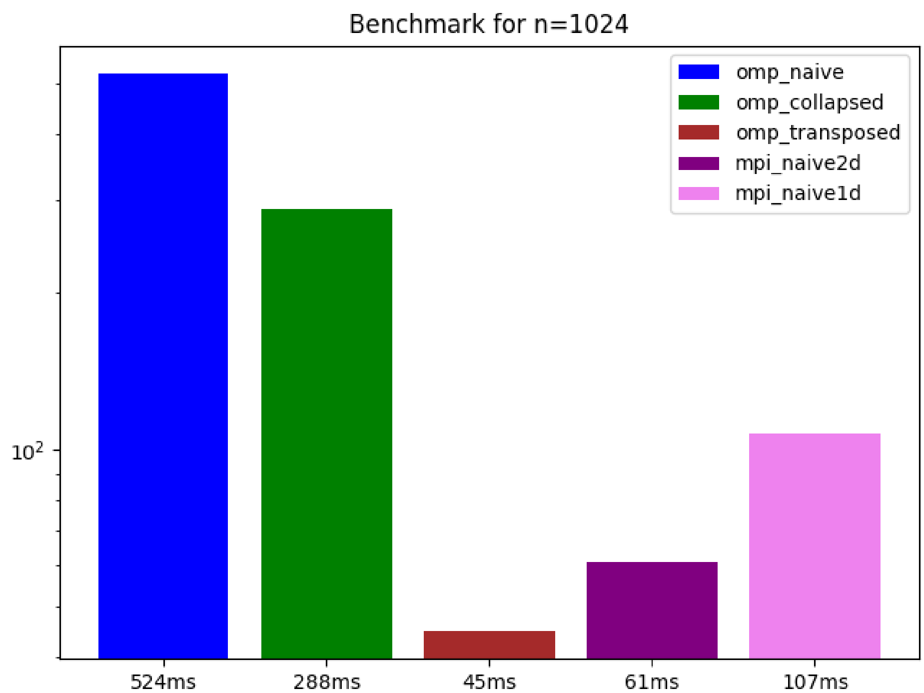
November 2023

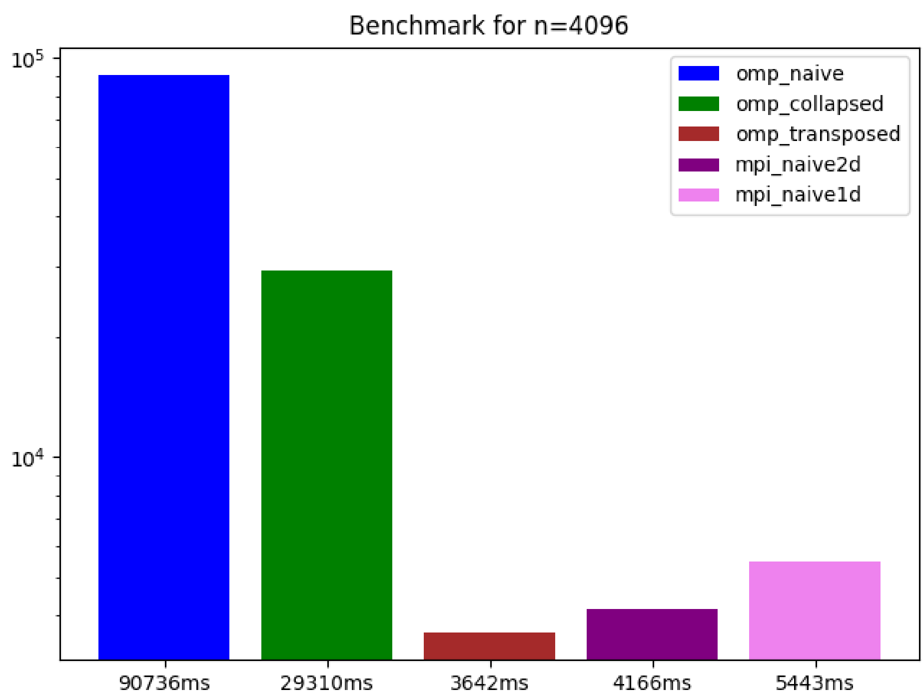
1 Metody Naiwne

Pierwszym pomysłem na zrównoleglenie naiwnego algorytmu mnożenia macierzy jest zwyczajne podzielenie operacji pomiędzy pewną pulę wątków.

W programie `omp_naive.cpp` tworzymy 1 wymiarową tablicę i wykonujemy zewnętrzną pętlę równolegle.
Rozwiąznaie to możemy jednak przyspieszyć transponując macierz B (`omp_reversed.cpp`). Wówczas iterowanie obu tablic odbywać się będzie po kolejnych współrzędnych, pozwalając tym na korzystanie z pól tablicy znajdujących się już w cachu. Optymalizacja ta okazuje się zakakująco efektywna. Już dla $n = 4096$ obserwujemy krótszy o ponad żąd wielkości czas wykonywania.
W celu daleszego przyspieszenia algorytmu, możnaby pokusić się o zastosowanie `collapse` w celu "złączenia" zagnieżdżonych pętli (`omp_collapse.cpp`) , które mogłyby wykonywać się równolegle. Okazuje się to jednak nieopłacalne. Podobnie jak w rozwiązaniu pierwszym, nie wykorzystujemy wówczas przewagi jaką daje nam cachowanie kolejnych elementów tablicy.

Do zrównoleglenia operacji mnożenia macierzy możemu również użyć bibliteki `mpthread`.
Programy `mpi_naive1d.cpp` i `mpi_naive2d.cpp` przedstawiają kolejno naiwną metodę realizowaną na kolejno 1 i 2 wymiarowej tablicy. Kolejne iteracje zewnętrznej pętli przyzielamy `NUM_THREAD` wątkom.


Benchmark for n=1024
omp_naive
omp_collapsed
omp_transposed
mpi_naive2d
mpi_naive1d
10^2
524ms
288ms
45ms
61ms
107ms


Benchmark for n=4096
omp_naive
omp_collapsed
omp_transposed
mpi_naive2d
mpi_naive1d
10^5
10^4
90736ms
29310ms
3642ms
4166ms
5443ms

2 Metoda Strassena

Algorytm Strassena polega na podziale maicerzy na 4 części, które wymnażać będziemy ze sobą rekurencyjnie. Formalnie zdefiniujmy:

$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$, $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$

Wówczas:
$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$
$C_{12} = A_{11} * B_{21} + A_{12} * B_{22}$
$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$
$C_{22} = A_{21} * B_{21} + A_{22} * B_{22}$

Jedną z metod zrównoleglenia wykonywania tych operacji jest podzielenie operacji czasochłonnych (mnożenia podmacierzy) na równoległe podzadania.
Przy pomocy `omp`, a dokładnie polecenia `task`, jesteśmy w osiągnąć taki podział. Zauważmy że dzieląc mnożenie na 4 podzadania względem macierzy C_{ij} możemy uniknąć jakichkolwiek konfliktów zapisu. Różne wątki będą współdzielić jedynie macierze A_{kl} i $B_{k'l'}$ które nie są nadpisywane w trakcie działania algorytmu.
Należy również pamiętać o wyznaczeniu pewnego kroku bazowego. Mówiąc ściślej, powinniśmy zadbać by dla odpowiednio małych macierzy nie liczyły się już rekurencyjnie. Możemy do tego użyć zwykłe, naiwnej metody sekwencyjnej, bądź jednej z opisanych metod naiwych-równoległych.

Analizując metody naiwne mnożenia macierzy, zobserwowaliśmy ich wpływ na przyspieszenie działania programu. Możemy zastanowić się czy stratgię tą dałoby się wykorzystać również w tej metodzie.
Okazuje się to dość trywialne i proste w implementacji. Wystarczy wprowadzić jedynie drobną poprawkę w implementacji: transpozycję "rozbicia" macierzy B w wywołaniu rekurencyjnym algorytmu Strassena.
Mamy więc teraz: $B = \begin{bmatrix} B_{11} & B_{21} \\ B_{12} & B_{22} \end{bmatrix}$.

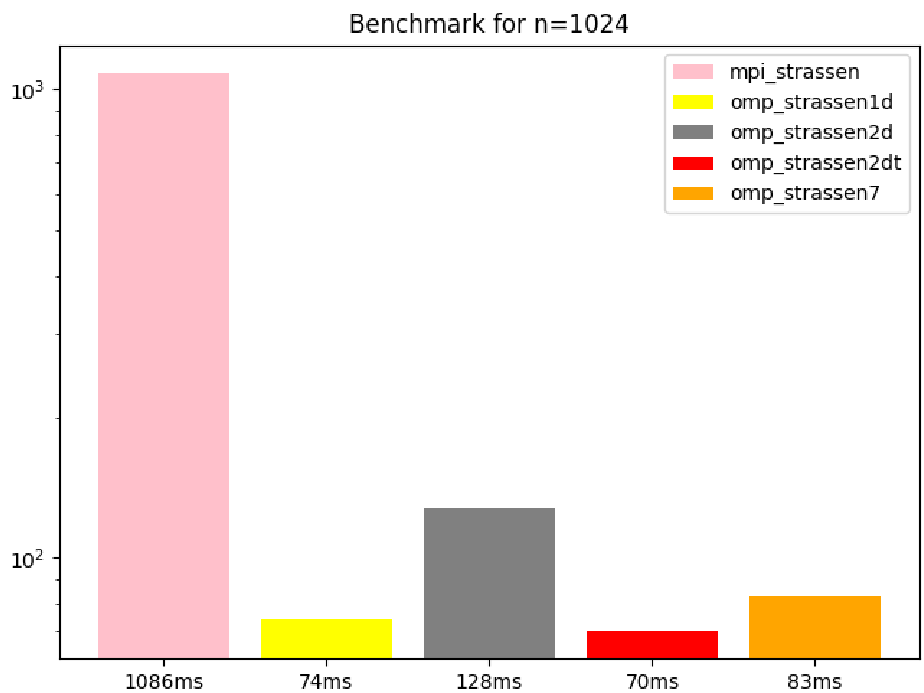
Implementacje tej metody zawarte są w plikach
`mpi_strassen.cpp`, `omp_strassen1d.cpp`, `omp_strassen2d.cpp`.

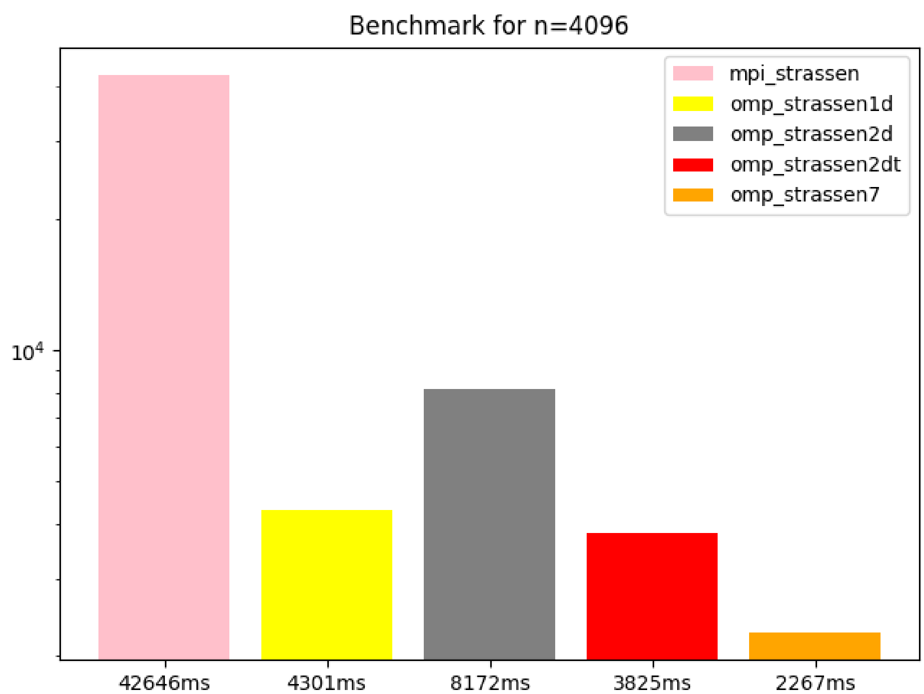
Problematyczna okazała się implementacja strassena przy pomocy `mpi`. Próbowałem limitować ilość tworzonych wątków ograniczając głębokość wywołań rekurencyjnych zanim zaczniemy liczyć mnożenia sekwencyjnie. Jednak podejście takie nie przyniosło obeicujących rezultatów.

Implementacja w `omp` okazała się bardziej efektywna. Wspomniane wyżej taski przyniosły pożądany efekt.
`omp_strassen1d` przedstawia moją pierwszą próbę implementacji tego algorytmu. W celu uniknięcia alokowania (i kopiowania) dodatkowych tablic dla każdego z "rozbić", stworzyłem strukturę Matrix. Struktura ta jest nakładką na jednowymiarową tablicę - spłaszczoną do 1 wymiaru macierz. Macierze Matrix można "rozbijać" ustawiając jej odpowiednie offsety po obu współrzędnych.

`omp_strassen2d` jest drobną modyfikacją poprzedniej implementacji. Tym razem przechowujemy macierze już bez użycia żadnej struktury. Imitujemy 2 wymiarową tablicę alokując n-elementową tablicę typu int*, której elementy wskazują na kolejne wiersze 1-wymiarowej macierzy.
Rozwiązanie to jest subiektywnie "czystsze" i przyjemniejsze w obsłudze. Testy wykazują, że mimo alokacji dodatkowych tablic, czas wykonania pozostaje prawie taki sam.

To jednak nie wszystko jeśli chodzi o algorytm Strassena. Istnieje pewna modykifacja tego algorytmu pozwalająca na osiągnięcie znacznie lepszych wyników. Możemy zmodyfikować algorytm Strassena tak, by wykonywac jedynie 7 (zamiast 8) mnożeń na podmacierzach. Jako że operacja mnożenia jest znacznie bardziej czasochłonna niż dodawanie/alokacja macierzy, podejście to wydaje się obiecujące. Tak też rzeczywiście możemy zaobserwować. W tym algorytmie ponownie wykorzystamy `omp` tworząc 7 podzadań, po jednym dla każdej operacji mnożenia.


Benchmark for n=1024
10^3
10^2
mpi_strassen
omp_strassen1d
omp_strassen2d
omp_strassen2dt
omp_strassen7
1086ms
74ms
128ms
70ms
83ms

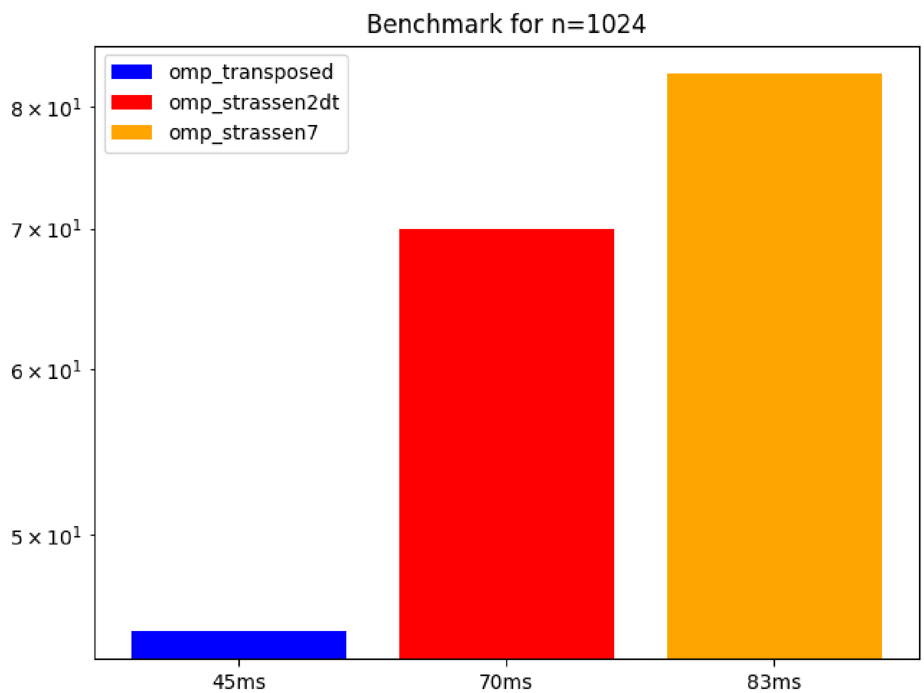

Benchmark for n=4096
10^4
mpi_strassen
omp_strassen1d
omp_strassen2d
omp_strassen2dt
omp_strassen7
42646ms
4301ms
8172ms
3825ms
2267ms

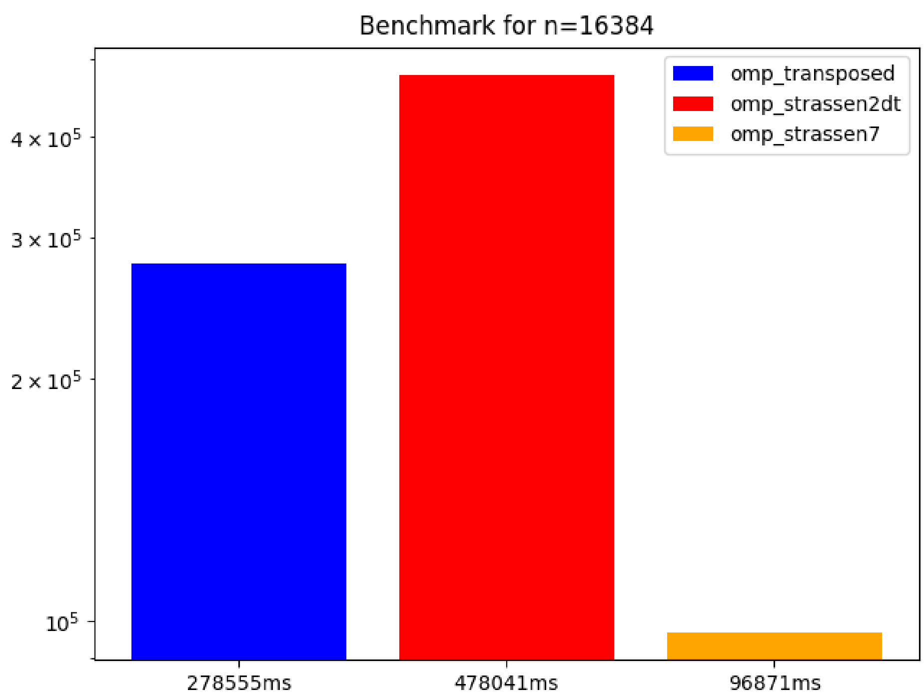
3 Podsumowanie

Otrzymane wykresy pokazują przeagę naiwnej metody w niewielkich testach i wyraźną przewagę strassena z 7 mnożeniami dla testów dużych.
Co może wydawać się ciekawe, zwykły algorytm strassena nie okazał się znacznie szybszy od naiwnej metody. Wykresy pokazują że czasy wykonywania dla dużego n są prawie identyczne.

Wykresy zostały stworzone przy użyciu tej samej paczki testu dla każdego z programów (10 testów dla $n = 1024, 4096$ oraz 5 testów dla $n = 16384$).
Wyniki osąignięte zostały przy hiperparamatrach `NUM_THREADS=64` oraz `MIN_SIZE=32`, natomiast kod uruchomiony został na serwerze `miracle.tcs.uj.edu.pl`.

Poniżej prezentują się wyniki trzech wybranych algroytmów: `omp_transposed`, `omp_strassen2dt`, `omp_strassen7` na ogromnych tesach ($n = 16384$).


Benchmark for n=1024
omp_transposed
omp_strassen2dt
omp_strassen7
8×10^1
7×10^1
6×10^1
5×10^1
45ms
70ms
83ms


Benchmark for n=16384
omp_transposed
omp_strassen2dt
omp_strassen7
4×10^5
3×10^5
2×10^5
10^5
278555ms
478041ms
96871ms