Sprawozdanie z laboratorium:

Przetwarzanie równoległe

Przetwarzanie równoległe PROJEKT OMP

Jacek Graczyk 127320

Temat nr. 16

1. Wstęp

Tematem przewodnim projektu jest analiza efektywności przetwarzania równoległego realizowanego na komputerze równoległym z procesorem wielordzeniowym z pamięcią współdzieloną. Do analizy wykorzystaliśmy program wykonujący mnożenie dwóch macierzy kwadratowych o takim samym rozmiarze. Przydzielony nam temat miał numer 16 i polegał na sprawdzeniu wpływu lokalizacji dyrektywy #pragma omp for na poprawność i efektywność metod mnożenia macierzy. Nasz algorytm wykorzystuje do tego 3 pętle for w kolejności jik. Dyrektywa #pragma omp parallel znajduje się przed pierwszą pętlą.

Do przeprowadzenia analizy i zebrania potrzebnych danych wykorzystaliśmy program CodeXL. Wszystkie obliczenia zostały wykonane na komputerze laboratoryjnym.

2. Analiza różnych wersji kodu

Dyrektywa **#pragma omp for** tworzy podział iteracji pętli znajdującej się po dyrektywie na poszczególne wątki dostępne w dyrektywie **#pragma omp parallel**. Rodzaj podziału iteracji między wątki może być określony przez dyrektywę **schedule**. Domyślnie iteracje pętli rozdzielane są po równo między wątki. Różne wersje kodu będziemy badać pod kątem wystąpienia **wyścigu w dostępie do danych** i **false sharing.** Weźmiemy także pod uwagę wystąpienie lokalności wątków: przestrzennej i czasowej.

a) omp for przed pętlą j: #pragma omp for przed pierwszą pętlą for

```
void multiply_matrices_JIK_J() {
#pragma omp parallel
#pragma omp for
    for (int j = 0; j < COLUMNS; j++) {
        for (int i = 0; i < ROWS; i++) {
            float sum = 0.0;
            for (int k = 0; k < COLUMNS; k++) {
                sum = sum + matrix_a[i][k] * matrix_b[k][j];
            }
            matrix_r[i][j] = sum;
        }
    }
}
```

W kodzie występuje dyrektywa **#pragma omp for** przed zewnętrzną pętlą for, co oznacza, że kolejne iteracje zmiennej “j” zostaną podzielone równomiernie między wątki. Dzięki temu kolumny macierzy matrix_b oraz macierzy wynikowej matrix_r nie będą się powtarzać w kolejnych wątkach.

Ponieważ zapis do konkretnej komórki macierzy matrix_r nie następuje przez więcej niż jeden wątek nie występuje tutaj wyścig w dostępie do danych. możemy mieć jednak do czynienia z tzw. false sharing, występuje on tutaj, ponieważ każdy wątek operuje na innej kolumnie, ale może operować na tym samym wierszu. Doprowadza to do sytuacji, gdzie możemy zaobserwować równoczesny dostęp do danych znajdujących się blisko siebie w pamięci cache, a tym samym istnieje możliwość wystąpienia unieważnienia danych przez inne wątki.

b) omp for przed pętlą i: #pragma omp for przed drugą pętlą for

```
void multiply_matrices_JIK_I() {
#pragma omp parallel
    for (int j = 0; j < COLUMNS; j++) {
#pragma omp for
        for (int i = 0; i < ROWS; i++) {
            float sum = 0.0;
            for (int k = 0; k < COLUMNS; k++) {
                sum = sum + matrix_a[i][k] * matrix_b[k][j];
            }
            matrix_r[i][j] = sum;
        }
    }
}
```

W kodzie występuje dyrektywa **#pragma omp for** przed środkową pętlą for, co oznacza, że kolejne iteracje zmiennej "i" zostaną podzielone równomiernie między wątki. Dzięki temu wiersze macierzy martix_a oraz macierzy wynikowej martix_r nie będą się powtarzać w kolejnych wątkach.

Tak jak w poprzedniej wersji kodu nie występuje tutaj wyścig w dostępie do danych, ponieważ wątki odwołują się do innych komórek w macierzy. Z racji tego, że wątki pracują na osobnych wierszach a nie kolumnach nie unieważniają one swoich linii pamięci więc zjawisko false sharing również nie występuje w tym przypadku.

c) omp for przed pętlą k: #pragma omp for przed trzecią pętlą for

```
void multiply_matrices_JIK_K() {
    // mnozenie macierzy
#pragma omp parallel
    for (int j = 0; j < COLUMNS; j++) {
        for (int i = 0; i < ROWS; i++) {
            float sum = 0.0;
            #pragma omp for
                for (int k = 0; k < COLUMNS; k++) {
                    sum = sum + matrix_a[i][k] * matrix_b[k][j];
                }
            #pragma omp atomic
                matrix_r[i][j] += sum;
        }
    }
}
```

W kodzie występuje dyrektywa **#pragma omp for** przed trzecią pętlą for, co oznacza, że wiersze macierzy martix_b oraz kolumny macierzy matrix_a zostaną rozdzielone równomiernie między wątki. Ponieważ macierzy wynikowa martix_r nie korzysta z indeksu, który został podzielony między wątki, stwierdzony został wyścig w dostępie do tej właśnie macierzy. Wyścig ten rozwiązaliśmy przez dyrektywę **#pragma omp atomic**, która gwarantuje, że dostęp do komórki macierzy wynikowej jest atomowy, czyli operacja przypisania może być realizowana równocześnie tylko przez jeden wątek dla danej komórki.

Macierz wynikowa matrix_r korzysta ze zmiennych, które nie mają zagwarantowanego współbieżnego dostępu między wątkami. Oznacza to, że w tym przypadku może wystąpić wyścig w dostępie macierzy wynikowej. Czyli przypisanie do konkretnej komórki macierzy wielokrotnie przez różne wątki. Prowadzi to do powstania błędnego wyniku. W kodzie rozwiązaliśmy ten problem zastosowując dyrektywę **#pragma omp atomic**. Innym sposobem rozwiązania tego problemu mogłoby być zastosowanie dyrektywy reduction, która tworzy prywatne

zmienne w obrębie danego wątku.

3. Analiza pod względem lokalności

Wyróżniamy dwa rodzaje lokalności - **czasowa** i **przestrzenna**. Czasowa polega na tym, że wątki wielokrotnie odwołują się do danych pobranych w pamięci podręcznej, zanim zostaną stamtąd usunięte. Przestrzenna natomiast charakteryzuje się dostępem do danych, następujących po sobie w pobranej linii pamięci co przekłada się na wiersze w macierzy w naszym programie. Przeanalizuję teraz warianty kodu pod względem lokalności. W każdym przypadku, aby występowała lokalność czasowa, suma rozmiaru danych potrzebnych do przeprowadzenia operacji musi być mniejsza od rozmiaru pamięci podręcznej.

a) **omp for przed pętlą j:**
Mamy do czynienia z lokalnością przestrzenną w przypadku macierzy matrix_a, ponieważ do pomnożenia i otrzymania jednej komórki w macierzy wynikowej potrzebujemy, całego wiersza macierzy matrix_a, czyli wykorzystujemy całe linie pamięci. Lokalność czasowa występuje jeżeli suma rozmiaru kolumny matrix_b, kolumny matrix_r oraz całej macierzy matrix_a jest mniejsza niż rozmiar pamięci podręcznej.

b) omp for przed pętlą i
W tym przypadku lokalność przestrzenna występuje ze względu na to, że do obliczenia jednej komórki wyniku potrzebujemy, całego wiersza macierzy matrix_a, czyli wykorzystania całej linii pamięci. Warunek spełnienia lokalności czasowej jest dokładnie taki sam jak w przypadku wariantu pierwszego.

c) omp for przed pętlą k
Lokalność przestrzenna tak jak poprzednio występuje dla macierzy matrix_a, ponieważ każdy wątek wykorzystuje kolejne wartości z wierszy macierzy, są one jednak podzielone pomiędzy wątki. Lokalność czasowa występuje jeżeli suma rozmiaru wiersza matrix_a oraz kolumny matrix_b jest mniejsza niż rozmiar pamięci podręcznej.

4. Uzasadnienie wyboru rozmiaru instancji macierzy

Naszym celem jest dobór takich rozmiarów instancji macierzy, aby był zbliżony do granicznego ze względu na ograniczenia pojemności podzespołów wewnętrznych. Ustalimy tą wielkość opierając się na naszym problemie mnożenia macierzy w przydzielonej nam kolejności. Jeśli dojdzie do przekroczenia wartości granicznych, czas wykonywania powinien znacząco się zwiększyć.

Interesujący nas podzespół to pamięć podręczna trzeciego poziomu "L3". Wybraliśmy ten podzespół, ponieważ pamięć ta jest współdzielona pomiędzy wszystkie rdzenie procesora. Gdy przekroczymy tą wartość czas działania algorytmu znacznie się wydłuży, gdyż wtedy będzie on zmuszony do pobrania danych z pamięci RAM, do której dostęp jest o wiele wolniejszy. To od jego rozmiaru zależy w jaki sposób dobierzemy wielkości graniczne macierzy.

Najpierw musimy obliczyć ile zmiennych typu float jesteśmy w stanie zmieścić w naszej pamięci L3. Tą wartość uzyskujemy przez podzielenie wielkości pamięci przez wielkość pojedynczej zmiennej typu float. Analizy dokonywaliśmy na komputerach laboratoryjnych, pamięć L3 ma tam pojemność 6 MB. Wielkość zmiennej typu float ma 4 B. Otrzymujemy zatem:

Wielkość zmiennej typu float:
F = 4 B

Pojemność pamięci trzeciego poziomu:
L3 = 6MB = 6 * 1024 * 1024 = 6291456 B

$\frac{L3}{F} = \frac{6291456\ B}{4\ B} = 1572864$ - Ilość zmiennych typu float, które jesteśmy w stanie zmieścić do analizowanej pamięci podręcznej.

Rozpatrzmy teraz ile takich zmiennych potrzebujemy załadować do pamięci w przypadku różnych wersji algorytmu.

Wariant 1 - **omp for przed pętlą j:**
Każdy wątek musi załadować do pamięci kolumnę z macierzy matrix_b i macierzy wynikowej matrix_r oraz całą macierz matrix_a, co nam daje:

$$Ma^2 + Mb + Mr \leq 1572864$$
$$Ma^2 + Ma + Ma \leq 1572864$$
$$Ma^2 + 2 * Ma \leq 1572864$$
$$Ma * (Ma + 2) \leq 1572864$$

$Ma \in [1572862; 1572864]$ - Przedział granicznej liczby komórek macierzy w przypadku omawianej wersji algorytmu.

$\sqrt{Ma} \sim 1254$ - Graniczny rozmiar wszystkich macierzy dla omawianej wersji algorytmu.

Ma - liczba komórek macierzy matrix_a
Mb - liczba komórek macierzy matrix_b
Mr - liczba komórek macierzy wynikowej matrix_r

W przypadku wariantu **omp for przed pętlą i**, mamy dokładnie taką samą ilość potrzebnych danych pobranych do pamięci podręcznej. Jeśli chodzi o **omp for przed pętlą k**, gdzie dyrektywa jest przed najbardziej zagnieżdżoną pętlą każdy wątek musi załadować do pamięci jedynie kolumnę z macierzy matrix_a oraz wiersz z macierzy

matrix_b.

Przy obliczeniach dla wariantu **omp for przed pętlą i** byłaby taka sama. Dla wariantu **omp for przed pętlą k** wielkość graniczna macierzy byłaby o wiele większa. Aby zredukować czas potrzebny do przeprowadzenia analizy, zdecydowałem się przeanalizować wyniki w oparciu o pierwszy wariant algorytmu. Biorąc pod uwagę wcześniejsze założenia oraz sposób nadpisywania starszych danych nowymi w pamięci podręcznej możemy stwierdzić, że ustalona wielkość macierzy nie powinna mieć wpływu na czas wykonywania wariantu 3 - **omp for przed pętlą k**.

Na podstawie powyższych obliczeń wybrane przez nas wielkości macierzy to 1100, 1200 i 1300.

5. Prezentacja miar efektywności

5.1. Zestawienie czasów wykonywania programu w sekundach w zależności od rozmiaru macierzy i pozycji dyrektywy **#pragma omp parallel for**.

GFLOPS zostało obliczone jako iloraz: $\frac{2*n^3*10^{-9}}{Czas\ [s]}$.

a) Rozmiar macierzy **n = 1100.**

	Czas [s]	**GFLOPS**
przed zewnętrzną	2,734	0,97366496
przed środkową	2,956	0,90054127
przed wewnętrzną	26,911	0,09891866

Tabela 1. Czas przetwarzania instancji dla rozmiaru macierzy 1100x1100 przy użyciu trzech wariantów kodu oraz odpowiadający obliczona jednostka GFLOPS.

b) Rozmiar macierzy **n = 1200**.

	Czas [s]	**GFLOPS**
przed zewnętrzną	4,352	0,79411765
przed środkową	3,928	0,87983707
przed wewnętrzną	34,997	0,09875132

Tabela 2. Czas przetwarzania instancji dla rozmiaru macierzy 1200x1200 przy użyciu trzech wariantów kodu oraz odpowiadający współczynnik GFLOPS.

c) Rozmiar macierzy **n = 1300**.

	Czas [s]	**GFLOPS**
przed zewnętrzną	4,809	0,91370347
przed środkową	5,796	0,75810904
przed wewnętrzną	21,611	0,20332238

Tabela 3. Czas przetwarzania instancji dla rozmiaru macierzy 1300x1300 przy użyciu trzech wariantów kodu oraz odpowiadający współczynnik GFLOPS.

5.2. Zestawienie ostatecznych danych opracowanych na podstawie pomiarów w programie CodeXL.

a) Dyrektywa **#pragma omp parallel for przed pętlą zewnętrzną**

n	1100	1200	1300
DC miss ratio	0,372	0,499	0,189
DC miss rate	0,169	0,237	0,087
DTLB L2 miss ratio	0,674	0,810	0,833
DTLB L2 miss rate	0,149	0,192	0,175
IPC	0,150	0,135	0,031

Tabela 4. Zestawienie ostatecznych danych w zależności od wielkości instancji dla wariantu *przed pętlą zewnętrzną*.

b) Dyrektywa **#pragma omp parallel for przed pętlą środkową**

	1100	1200	1300
DC miss ratio	0,371	0,495	0,192
DC miss rate	0,230	0,275	0,089
DTLB L2 miss ratio	0,671	0,823	0,842
DTLB L2 miss rate	0,190	0,226	0,176
IPC	0,164	0,132	0,095

Tabela 5. Zestawienie ostatecznych danych w zależności od wielkości instancji dla wariantu *przed pętlą środkową*.

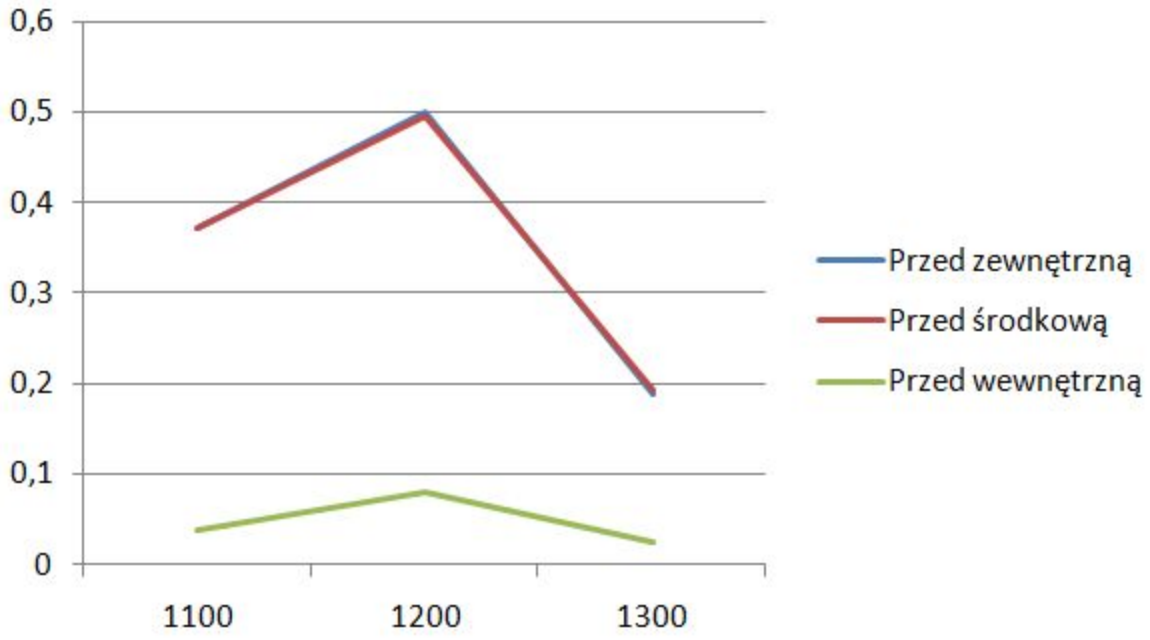
c) Dyrektywa **#pragma omp parallel for przed pętlą wewnętrzną**

	1100	1200	1300
DC miss ratio	0,037	0,080	0,024
DC miss rate	0,331	0,840	0,059
DTLB L2 miss ratio	0,020	0,022	0,022
DTLB L2 miss rate	0,010	0,011	0,007
IPC	0,428	0,200	0,084

Tabela 6. Zestawienie ostatecznych danych w zależności od wielkości instancji dla wariantu *przed pętlą wewnętrzną*.

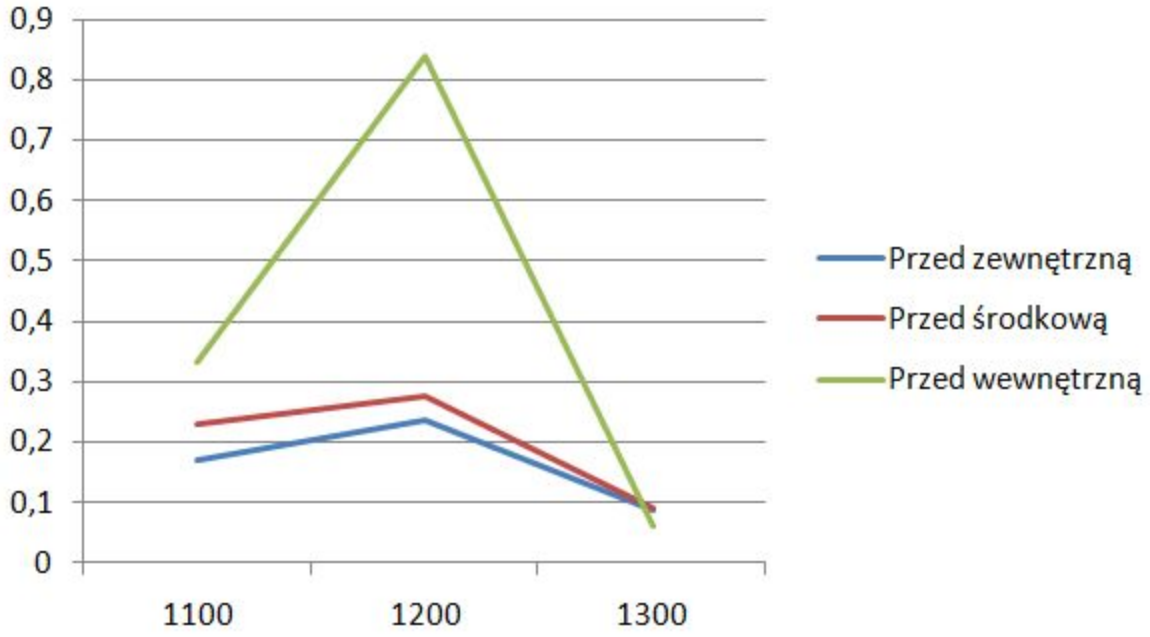
5. Analiza pomiarów

5.1. Data Cache miss ratio - jest to stosunek nietrafień do pamięci podręcznej do dostępów zrealizowanych. Miara ta pokazuje nam jak często następowały nietrafienia do pamięci w stosunku do wszystkich odwołań.



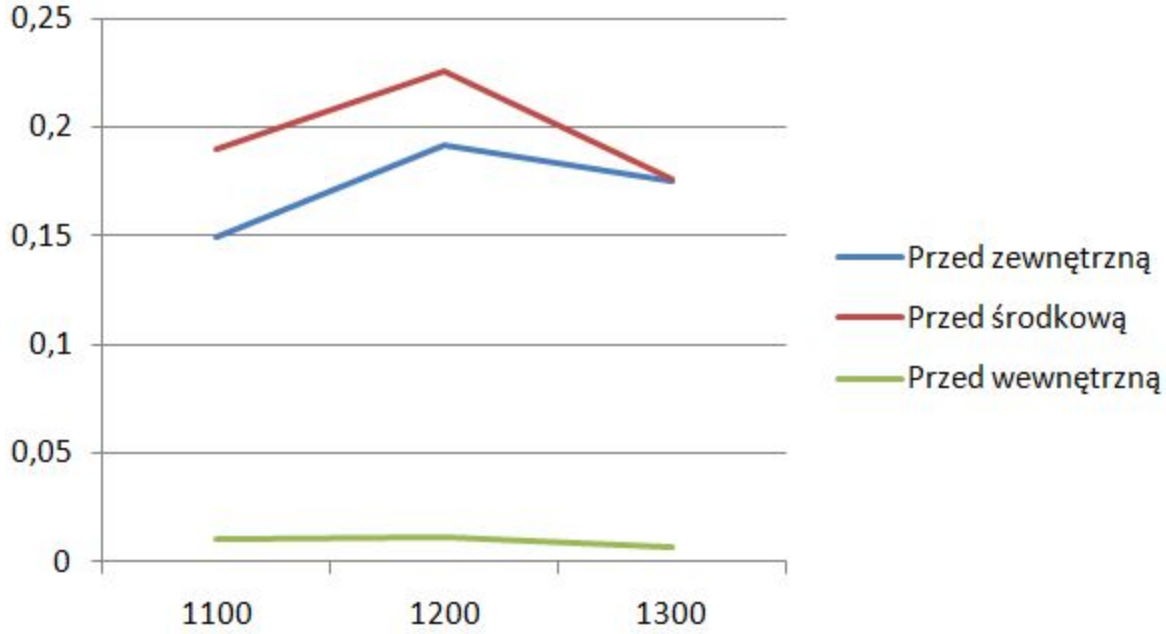
Wykres 1. Wykres przedstawiający wartość Data Cache miss ratio dla poszczególnych rozmiarów macierzy.

5.2. Data Cache miss rate – stosunek nietrafień do pamięci podręcznej i wykonanych instrukcji. Miara ta mówi, jak często nie możemy pobrać danych z pamięci podręcznej i musimy korzystać z wolniejszej pamięci. Im większy ten stosunek tym wolniej działa program.



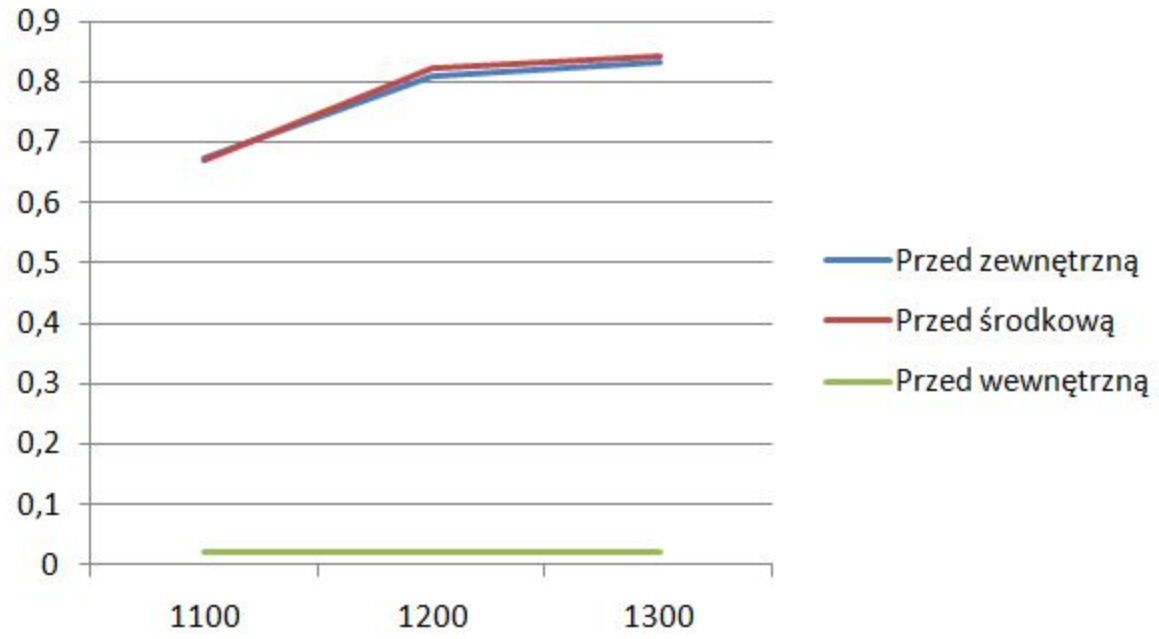
Wykres 2. Wykres przedstawiający wartość Data Cache miss rate dla poszczególnych rozmiarów macierzy.

5.3. Data Translation Lookaside Buffer L2 miss rate - określa częstość nietrafień do bufora translacji adresów. Im jest wyższy, tym częściej musimy sięgać do pamięci operacyjnej. Naszym celem jest jego minimalizacja.



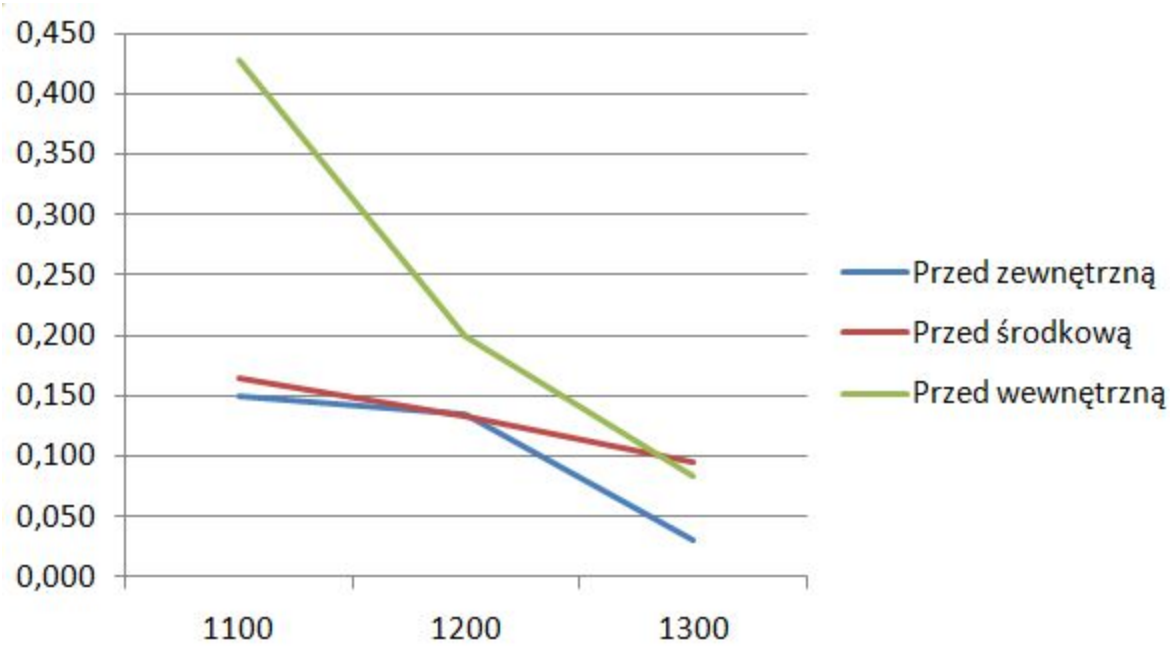
Wykres 3. Wykres przedstawiający wartość Data Translation Lookaside Buffer L2 miss rate dla poszczególnych rozmiarów macierzy.

5.4. Data Translation Lookaside Buffer L2 miss ratio - To wskaźnik żądań dostępu do pamięci podręcznej L2 po nietrafieniu do L1. Staramy się go minimalizować.



Wykres 4. Wykres przedstawiający wartość Data Translation Lookaside Buffer L2 miss rate dla poszczególnych rozmiarów macierzy.

5.5. Instruction per Cycle (IPC) - liczba instrukcji na cykl. Pozwala nam to zmierzyć efektywność zrównoleglenia dla poziomu przetwarzania instrukcji. Niska wartość IPC może oznaczać, że jest jest wiele czynników, które zmniejszają efektywność.

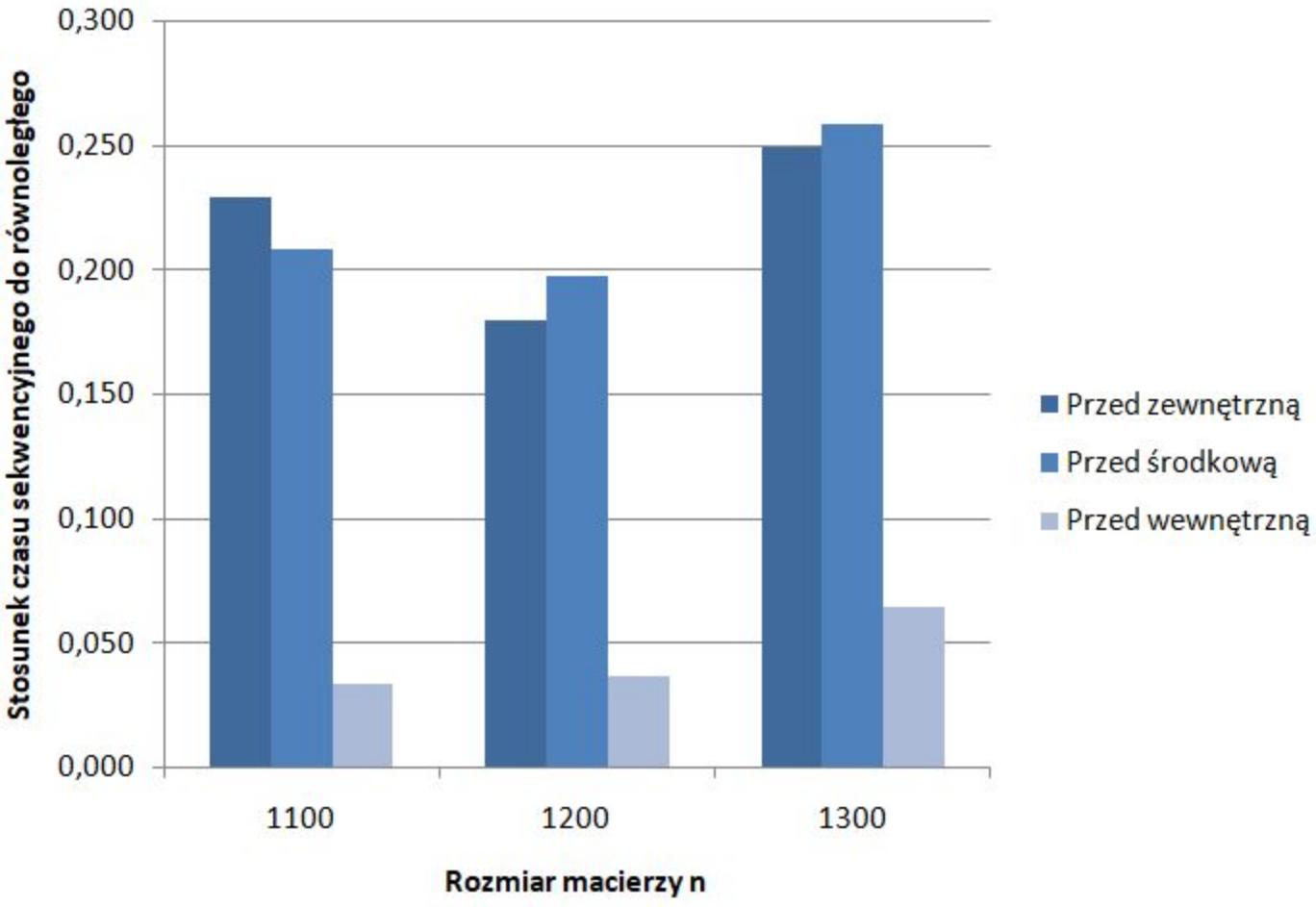


Wykres 5. Wykres przedstawiający wartość Data Translation Lookaside Buffer L2 miss rate dla poszczególnych rozmiarów macierzy.

6. Analiza przyśpieszenia

Aby przeanalizować przyśpieszenie otrzymane w wyniku zrównoleglenia obliczeń, potrzebujemy czas wykonywania obliczeń w sposób sekwencyjny. Dzielimy ten czas przez czas w przypadku wersji równoległej.

Miara liczona ze wzoru: $\frac{Tseq}{Trow}$



Wykres 6. Porównanie przyspieszeń wykonywania programu równoległego dla różnych rozmiarów instancji macierzy oraz umiejscowienia dyrektywy przed pętlami.

Powyższy wykres przedstawia bardzo istotne zjawisko. Czas wykonywania programu sekwencyjnego jest w każdym przypadku krótszy niż wykonywanie równoległe. Obliczenia sekwencyjne wykonują się od 4 do 30 razy szybciej. Możemy uznać, że konfiguracja JIK nie poprawia wyników obliczeń równoległych, gdyż:

1. False sharing – wątki muszą pobierać unieważnione dane tracąc przy tym czas, w którym mogłyby wykonywać więcej obliczeń.
2. Słaba lokalność przestrzenna danych – zgodnie z analizą algorytmu mnożenia w kolejności **JIK** większość iteracji obydwa się wzdłuż kolumn co wymaga sprowadzanie kolejnych wierszy i mocno zwiększa ilość nietrafień do pamięci podręcznej.

7. Wnioski

Obliczenia jasno pokazują, że funkcja JIK okazała się skrajnie nieefektywna. Jest to spowodowane kilkoma problemami. Po pierwsze kolejność dostępu do danych, czyli słaba lokalność przestrzenna, co można było zauważyć już przy początkowej analizie. Występuje ona tylko dla jednego wiersza w macierzy przy wykonaniu każdego obliczenia.

Drugi problem to false sharing. Przy obliczeniach wątki często unieważniały sobie dane, co skutkuje ich aktualizacją i pobraniem z pamięci operacyjnej. To z kolei znacznie zwiększa czas przetwarzania.

Kolejną kwestią jest nie wzięcie pod uwagę, czy podczas obliczeń procesor jest obciążony. W przypadku moich obliczeń nie uważałem na to, co mogło mieć spory wpływ na wynik.

Maksymalne otrzymane przyspieszenie wynosi 0.25, co oznacza, że w przypadku każdego rozmiaru macierzy sekwencyjna wersja obliczeń wykonała się w mniejszym czasie. Na tej podstawie stwierdzam, że kolejność wykonania pętli JIK jest wysoce nieefektywna w przypadku zrównoleglenia.