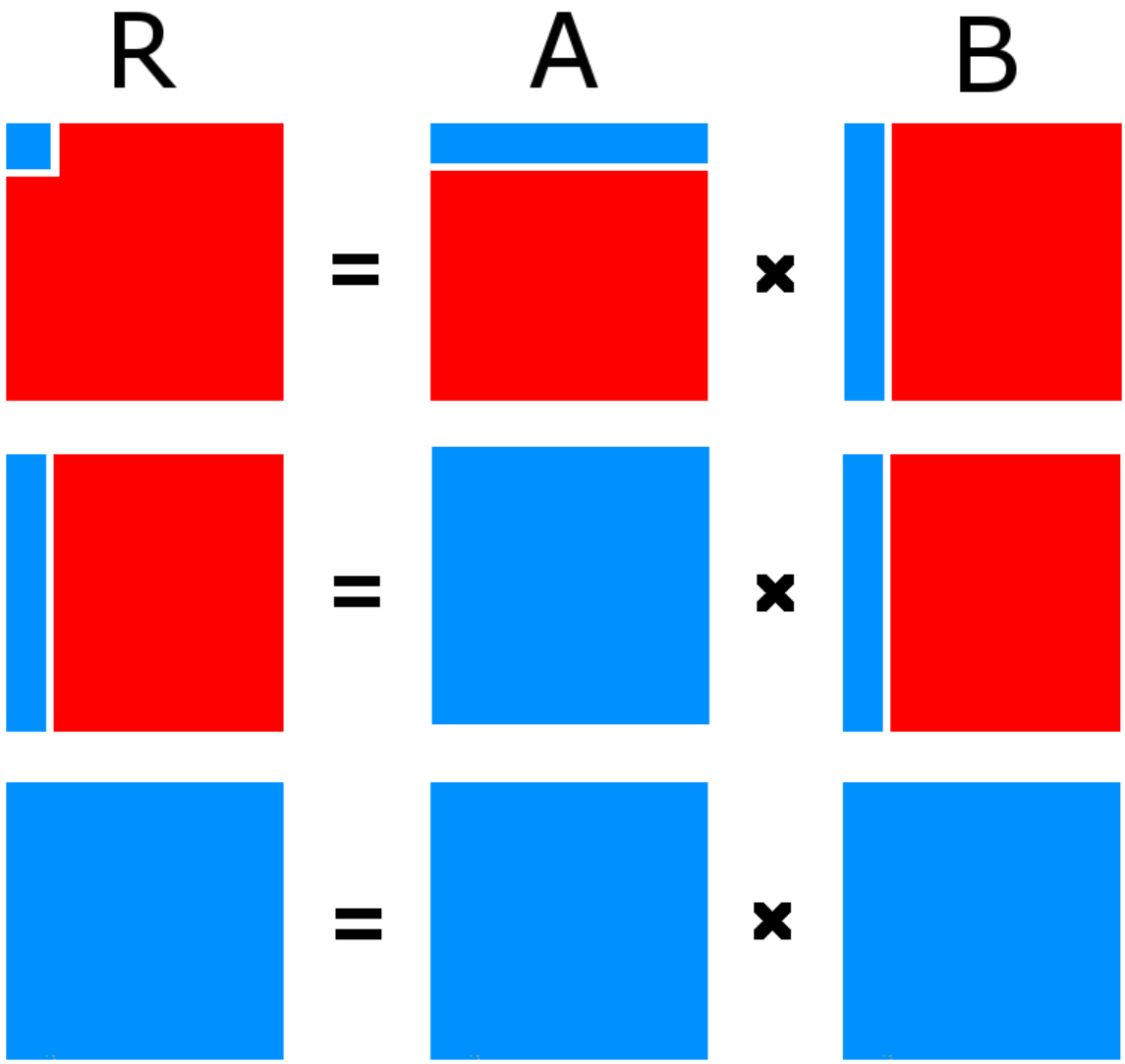
macierzy **B**. Wynikiem pętli środkowej jest jedna kolumna macierzy **R**.

W jednej iteracji najbardziej zewnętrznej pętli (zmienna iteracyjna j) przemnażamy przez siebie komórki z całej macierzy **A** i danej kolumny z macierzy **B**. Iteracje po zmiennej **j** przebiegają po całej macierzy **A** i całej macierzy **B**. Wynikiem pętli zewnętrznej jest cała macierz **R**.

Poniższy rysunek ilustruje wyniki wykonania pętli dla kolejnych zmiennych iteracyjnych.



2.2. Analiza konfliktów pomiędzy wątkami

Dyrektywa **#pragma omp for** rozdziela iteracje pętli **for** następujące po dyrektywie pomiędzy wątki z dostępnej puli. Domyślnie każdy wątek otrzymuje do wykonania jedną iterację pętli, a po jej zakończeniu kolejną, dzieje się tak aż do wyczerpania wszystkich iteracji w pętli **for**. Będziemy analizować dwa różne rodzaje konfliktów:

- *Data race* – zjawisko wpływające na błędne dane wynikowe
- *False sharing* – zjawisko znacznie spowalniające wykonanie programu

W przypadku tej analizy rozważamy trzy miejsca umieszczenia dyrektywy:

- Przed pętlą zewnętrzną – każdy z wątków w danej iteracji otrzymuje do policzenia jedną kolumnę macierzy **R**. W tej sytuacji nie obserwujemy zjawiska *data race*. Wynika to z faktu iż każda z komórek macierzy jest zapisywana przez tylko i wyłącznie jeden wątek. Zaobserwujemy jednak zjawisko *false sharingu* – każdy z wątków działa na innej kolumnie, ale ponieważ komórki w tym samym wierszu, a w różnych kolumnach znajdują się w pamięci blisko siebie to może wystąpić unieważnianie danych przez inne wątki.
- Przed pętlą środkową – podział pracy na wątki odbywa się względem jednej kolumny macierzy **R**. W tej sytuacji również nie zaobserwujemy sytuacji *data race*. Przyczyna jest identyczna jak w poprzednim przypadku. W tej wariacji nie zaobserwujemy jednak również zjawiska *false sharingu*. Z racji iż w danej kolumnie każdy z wątków działa na każdym wierszu to nie unieważniają one swoich linii pamięci podręcznych.
- Przed pętlą wewnętrzną – podział pracy odbywa się na poziomie obliczeń jednej komórki macierzy **R**. W każdej iteracji zmiennej **k** wątek przemnaża wartości komórek z macierzy **A** i **B**, a następnie dodaje wynik do współdzielonej zmiennej **suma**. Z racji tego iż nie zapewniliśmy atomowego dostępu do tej zmiennej przy pomocy słowa kluczowego **atomic** ani nie użyliśmy redukcji wystąpi *data race*. Za każdym razem gdy wątek chce dodać coś do zmiennej **suma** musi wczytać ją z pamięci podręcznej do rejestru, powiększyć o wartość iloczynu i zapisać ponownie do pamięci podręcznej. W tym momencie może dojść do wyścigu w dostępie do danych. Powoduje to, iż ostateczny wynik całego programu będzie błędny. Z tego samego powodu możemy również mówić o zjawisku *false sharingu*.

2.3. Analiza lokalności wątków

Będziemy brali pod uwagę dwa rodzaje lokalności:

- Lokalność czasowa – wątek wielokrotnie odwołuje się do tej samej danej.
- Lokalność przestrzenna – wątek w kolejnych operacjach odwołuje się do danych następujących bezpośrednio po sobie

Ponownie rozważamy trzy miejsca umieszczenia dyrektywy:

- Przed pętlą zewnętrzną – lokalność czasowa nie występuje dla żadnej z macierzy w każdej kolejnej operacji związanej z macierzą wątek korzysta z innej komórki. Lokalność przestrzenna pojawia się w przypadku macierzy **A**. Podczas kolejnych iteracji wewnętrznej pętli każdy z wątków iteruje wzdłuż wiersza. Niestety nie dotyczy to macierzy **B** i **R**, które są iterowane po kolumnach.
- Przed pętlą środkową – lokalność czasowa nie występuje z tych samych powodów co w poprzednim wariancie. Podobnie z lokalnością przestrzenną, z tych samych powodów co w poprzednim wariancie występuje tylko dla macierzy **A**.
- Przed pętlą wewnętrzną – lokalność czasowa występuje dla macierzy **R**. W danej iteracji pętli wewnętrznej każdy z wątków odwołuje się do tej samej komórki. Występuje tutaj lokalność przestrzenna dla macierzy **A**, jednak jest ona słabsza niż w

poprzednich wariantach. Co prawda każdy wątek w kolejnych iteracja wykorzystuje następujące po sobie wartości z wiersza macierzy, jednak kolejne iteracje w wierszu dzielone są pomiędzy więcej niż jeden wątek.

3. Instancja graniczna macierzy

Z uwagi na ograniczenia pojemnościowe komponentów pamięci komputera możemy mówić o wielkościach instancji, które są graniczne dla danego problemu. W przypadku przekroczenia tych wielkości następuje znaczne spowolnienie czasu wykonywania programu.

Podzespół, który interesuje nas podczas tej analizy to pamięć podręczna trzeciego poziomu – L3. W przypadku komputerów laboratoryjnych ma ona pojemność 6MB, a więc przy wielkość 4B dla zmiennej typu float pozwala na przechowanie:

$$\frac{6MB}{4B} = 1,5 * 1024 * 1024 = 1\ 572\ 864$$

zmiennych typu float. Pamięć ta jest współdzielona dla wszystkich rdzeni procesora. Stąd też instancje problemu, których iteracje dla poszczególnych wątków nie zmieszczą się w pamięci L3 zostaną obciążone czasem pobrania dodatkowych danych z pamięci RAM.

Początkowo wyznaczymy instancję problemu dla wariancji ze zrównolegleniem pracy na pętli zewnętrznej. Każdy w wątków potrzebuje załadować do pamięci jedną kolumnę z macierzy **B**, całą macierz **A** i kolumnę do zapisu macierzy **R**.

$$n_a^2 + 4 * n_b + 4 * n_r < 1572864,$$

gdzie:

$n_a^2 - wielkość\ macierzy\ A$

$n_b - wielkość\ wiersza\ macierzy\ B$

$n_r - wielkość\ kolumny\ macierzy\ R$

$n_a = n_b = n_r$

Wynik rozwiązania równania to:

$$n_a = n_b = n_r = 1250$$

W programie wykorzystaliśmy jednak również inne miejsca zagnieżdżenia dyrektywy zrównoleglenia. Warto jednak zauważyć, że w tamtych przypadkach wzór opisujący wielkość instancji granicznej będzie taki sam albo lewa strona zostanie zredukowana do równania liniowego. W tym drugim przypadku instancja graniczna będzie na tyle duża, że zebranie związanych z nią statystyk zajęło by zdecydowanie za dużo czasu.

Wybrane przez nas wielkości instancji n to 1100, 1200, 1300

4. Wyniki

Poniższa tabela zawiera porównanie czasu wykonywania programu w sekundach w zależności od wielkości instancji i pozycji dyrektywy **#pragma omp parallel for**.

n	2*n^3	Przed zewnętrzną	Przed środkową	Przed wewnętrzną	czas
1100	2662000000	**2.7278**	**2.942**	**26.1778**	s
		0.975878	0.904827	0.101689	GFLOPS
1200	3456000000	**4.354**	**3.9646**	**35.1524**	s
		0.79375287	0.871715	0.098315	GFLOPS
1300	4394000000	**4.8136**	**5.7516**	**20.6094**	s
		0.91283031	0.763961	0.213204	GFLOPS

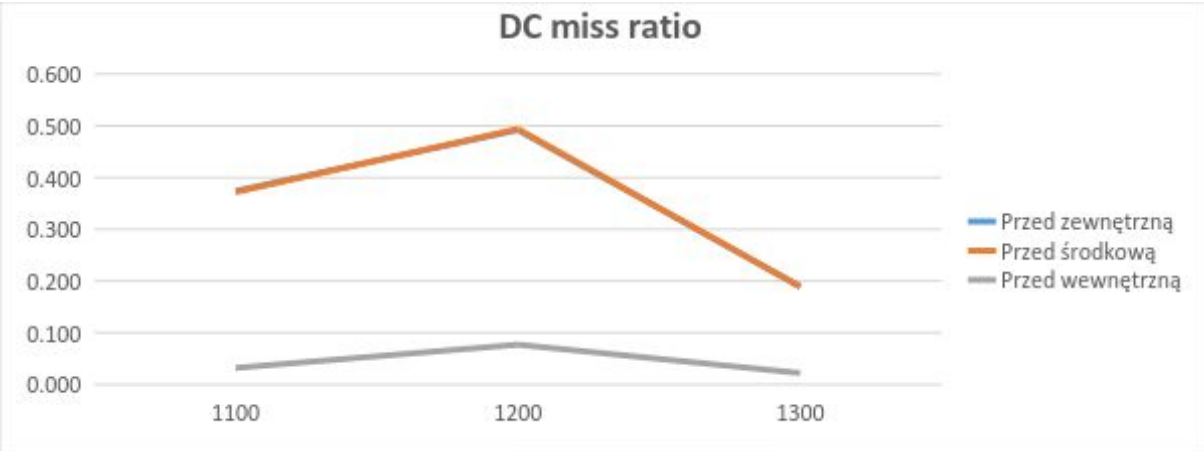
Dodatkowo przedstawiamy dane zebrane za pomocą programu CodeXL.

n	DC miss ratio	DC miss rate	DTLB L2 miss ratio	DTLB L2 miss rate	IPC
	Przed zewnętrzną				
1100	0.374	0.173	0.674	0.145	0.157
1200	0.492	0.233	0.808	0.191	0.134
1300	0.189	0.079	0.831	0.175	0.030
	Przed środkową				
1100	0.372	0.227	0.675	0.191	0.167
1200	0.494	0.267	0.809	0.219	0.134
1300	0.189	0.080	0.832	0.175	0.090
	Przed wewnętrzną				
1100	0.032	0.321	0.014	0.007	0.415
1200	0.077	0.812	0.017	0.01	0.178
1300	0.022	0.033	0.013	0.002	0.057

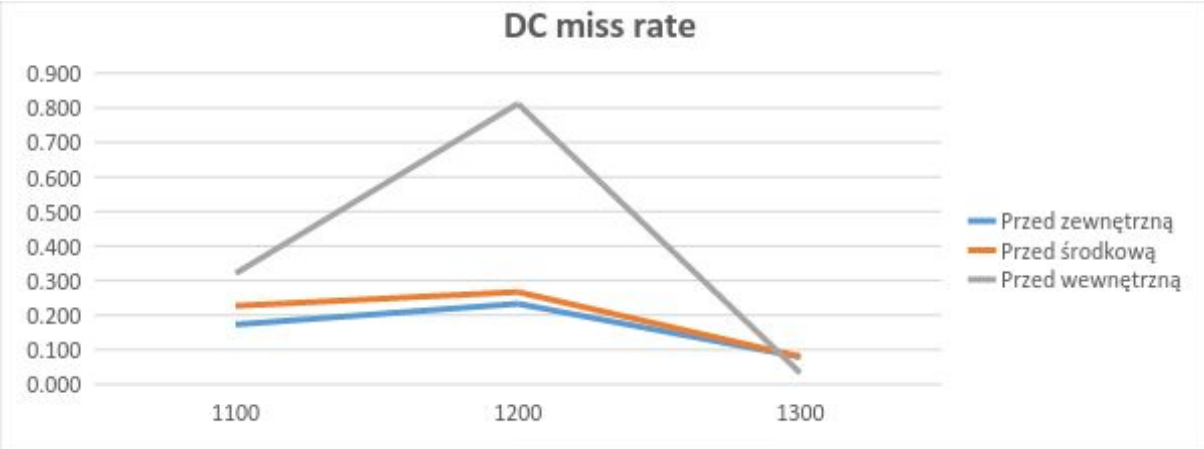
Wyniki znajdujące się w powyższej tabeli są ostateczne. Pominęliśmy wypisywanie zbędnych, surowych danych i przeszliśmy od razu do obliczonych charakterystyk

5. Analiza wyników

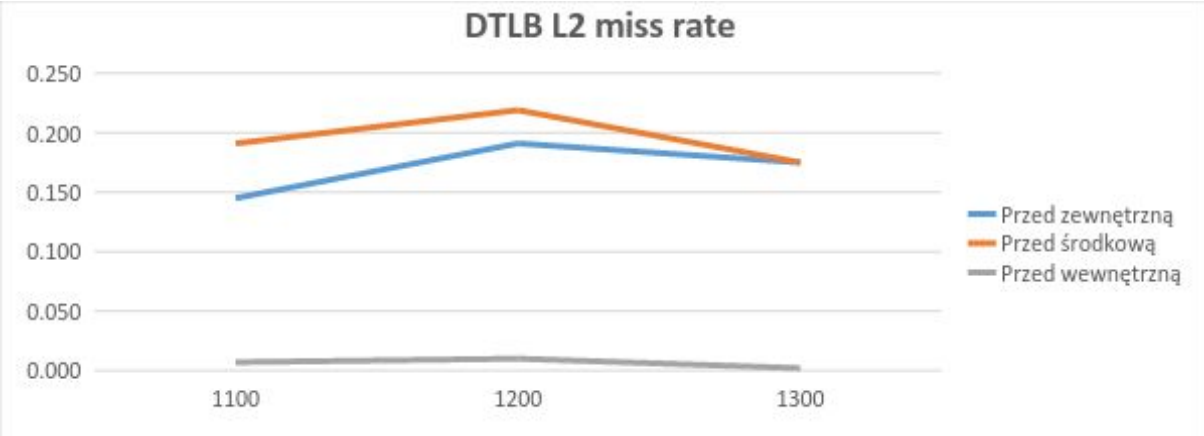
Data Cache miss ratio to stosunek nietrafień do pamięci podręcznej i dostępów zrealizowanych. Ta miara pokazuje jak często nie trafiamy do pamięci w stosunku do wszystkich odwołań do pamięci.



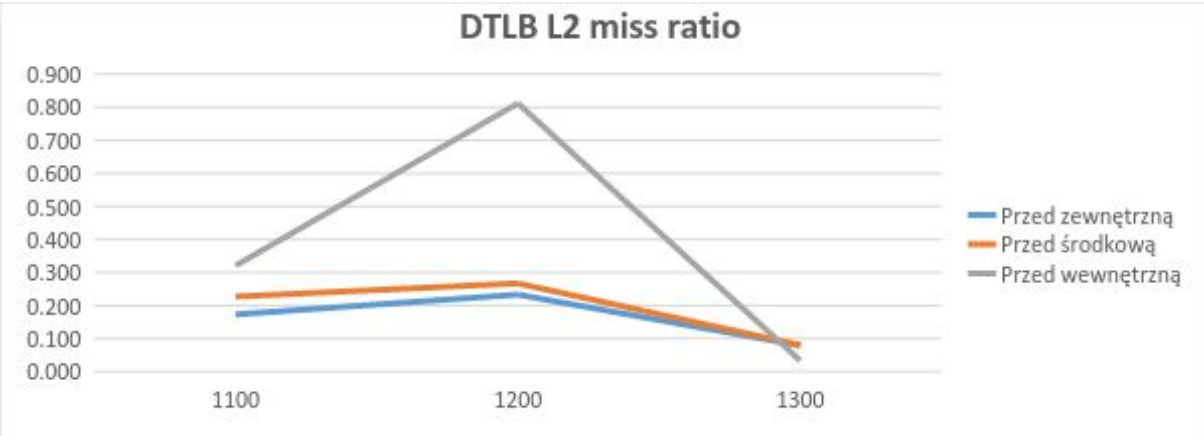
Data Cache miss rate to stosunek nietrafień do pamięci podręcznej i wykonanych instrukcji. Miara ta mówi, jak często nie możemy pobrać danych z pamięci podręcznej i musimy korzystać z wolniejszej pamięci. Im większy ten stosunek tym wolniej działa program.



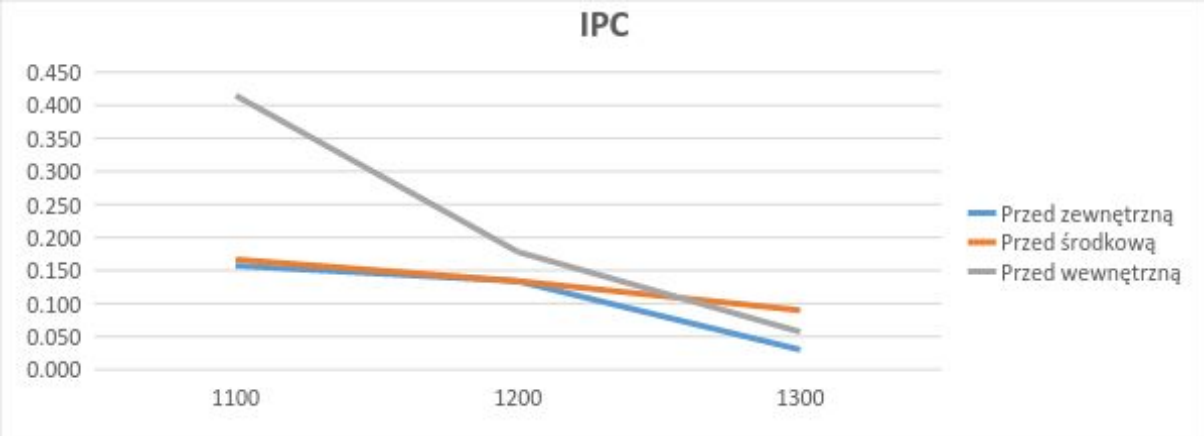
Data Translation Lookaside Buffer L2 miss rate mówi, jak często nie trafiamy do bufora translacji adresów. Im wyższy ten współczynnik, tym częściej musimy sięgać do wolnej pamięci. Staramy się go minimalizować.



Data Translation Lookaside Buffer L2 miss rate jest wskaźnikiem żądań dostępu do poziomu L2 pamięci podręcznej, pokazuje jak często występuje brak trafienia w L1 i odwołanie do L2. Powinien być jak najmniejszy.

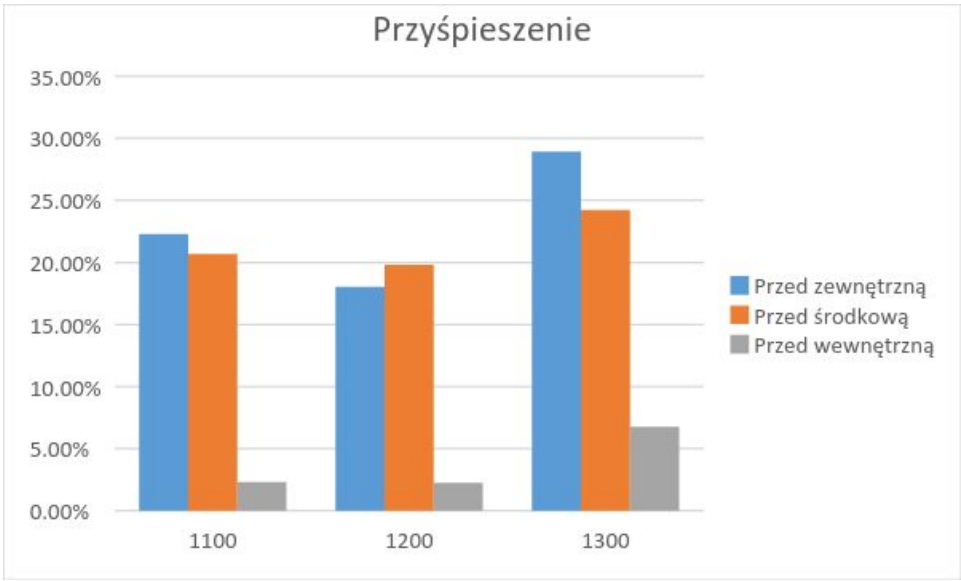


Instruction per Cycle (IPC) – stosunek wykonanych instrukcji do liczby cykli zegara, które minęły. Staramy się, aby wartość ta była jak największa, bo im więcej instrukcji wykonujemy na cykl, tym lepsze jest wykorzystanie procesora.



6. Analiza przyspieszenia

W ramach analizy przyspieszenia, które uzyskaliśmy przy pomocy zrównoleglenia obliczeń skorzystamy z pomiarów czasu wykonania programu sekwencyjnego. Na ich podstawie możemy wyznaczyć przyspieszenie zdefiniowane jako iloraz czasu obliczeń sekwencyjnych najlepszą metodą i czasu obliczeń równoległych.



Jak widać w żadnych z mierzonych przypadków nie doszło do sytuacji, w której zrównoleglenie dałoby jakikolwiek zysk. W najgorszym przypadku – umieszczenie zrównoleglenia przed pętlą wewnętrzną w instancji o wielkości 1200 – program działał 40 razy wolniej. Można stwierdzić, że kolejność **JIK** zdecydowanie nie jest optymalna dla obliczeń równoległych. Oto powody, które chcielibyśmy wyróżnić:

- 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.
- Słaba lokalność czasowa – występuje ona tylko i wyłącznie w jednej sytuacji, dodatkowo jest to wariant algorytmu, który daje błędne wyniki.

- Występowanie *false sharingu* – we wcześniejszej analizie wykazaliśmy istnienie zjawiska w problemie. Jest to kolejna niekorzystna cecha, która spowalnia wykonywanie programu. Wątki często muszą pobierać unieważnione dane do swojej pamięci podręcznej, a czas ten mogłyby poświęcić na wykonywanie kolejnych operacji i szybsze obliczenie wyniku końcowego.

Co więcej, analizując przyspieszenie, można zauważyć, że przekroczenie wyliczonej wcześniej wartości granicznej, rzeczywiście wpływa na czas mnożenia macierzy i tym samym przyspieszenie. Jest to przede wszystkim widoczne dla najbardziej zagnieżdżonego zrównoleglenia.

7. Podsumowanie

Dodatkowo podczas pomiarów wykonywanych w laboratorium sprawdziliśmy również obciążenie poszczególnych procesorów w czasie wykonywania programu. Odczytane wartości wskazywały na równomierne obciążenie procesorów.

Podsumowując, nieodpowiednie przetworzenie problemu do postaci równoległej powoduje znaczące pogorszenie wydajności przetwarzania. Dlatego pojawia sie konieczność dogłębnej analizy problemu przed implementacja programu równoległego.

KOD:

Wariant 1 – dyrektywa przed pętlą zewnętrzną

```
#pragma omp parallel 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;
                }
        }
```

Wariant 2 – dyrektywa przed pętlą środkową

```
for (int j = 0; j < COLUMNS; j++) {
#pragma omp parallel 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;
        }
}
```

Wariant 3 – dyrektywa przed pętlą wewnętrzną

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

Wariant sekwencyjny

```
for (int i = 0; i < ROWS; i++) {
        for (int k = 0; k < COLUMNS; k++) {
                for (int j = 0; j < COLUMNS; j++) {
                        matrix_r[i][j] += matrix_a[i][k] * matrix_b[k][j];
                }
        }
}
```