Obliczenia wielkiej skali PROJEKT 3
OMP
Sprawozdanie

Daniel Wiechetek
Adam Dachtera

Punkt 1:

Opis wykorzystywanego systemu obliczeniowego

procesor 1:
Intel(R) Core(™) i7-9750H CPU @ 2.60GHz
liczba procesów fizycznych: **368**
liczba procesów logicznych: **12**
liczba uruchomionych w systemie wątków: **3984**
oznaczenie typu procesora: **9**
wielkość i organizacja pamięci podręcznych procesora: **384 kb**
nazwa systemu operacyjnego 1: **Windows 11**
wersja i nazwa programu użytego do przygotowania kodu wynikowego i przeprowadzenia testów: **Dev-C++ 5.11 oraz Oracle VM Virtualbox ver. 7.0 z Linuxem**

Punkt 2:

Prezentacja kodów

Wersja A:

kod wersji A(3 pętlowa):

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void initialize_matrices(float **A, float **B, float **C, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = (float)(rand() % 10);
            B[i][j] = (float)(rand() % 10);
            C[i][j] = 0.0f;
        }
    }
}

float** allocate_matrix(int n) {
    float **matrix = (float **)malloc(n * sizeof(float *));
    for (int i = 0; i < n; i++) {
        matrix[i] = (float *)malloc(n * sizeof(float));
    }
    return matrix;
}

void free_matrix(float **matrix, int n) {
    for (int i = 0; i < n; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

void version_A(float **A, float **B, float **C, int n) {
    #pragma omp parallel for
    for (int i = 0; i < n; i++) { // Przechodzi przez wiersze macierzy A
        for (int j = 0; j < n; j++) { // Przechodzi przez kolumny macierzy B
            for (int k = 0; k < n; k++) { // Mnoży i sumuje odpowiednie elementy wiersza z A i kolumny z B
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <matrix_size>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);

    float **A = allocate_matrix(n);
    float **B = allocate_matrix(n);
    float **C = allocate_matrix(n);

    initialize_matrices(A, B, C, n);

    double start_time = omp_get_wtime();
    version_A(A, B, C, n);
    double end_time = omp_get_wtime();
    printf("Version A Time: %f seconds\n", end_time - start_time);

    free_matrix(A, n);
    free_matrix(B, n);
    free_matrix(C, n);

    return 0;
}
```

Podział pracy wersji A:
W wersji A, podział pracy jest realizowany na poziomie wierszy macierzy wynikowej C. Każdy wiersz macierzy C jest traktowany jako odrębne zadanie. W przetwarzaniu równoległym liczba zadań odpowiada liczbie wierszy n, więc powstaje n zadań do wykonania. Każde zadanie polega na obliczeniu wartości wszystkich elementów jednego wiersza macierzy C.

Sposób przydziału zadań wersji A:
Zadania (wiersze macierzy) są przydzielane do wątków dynamicznie, co oznacza, że wątki są przydzielane do wierszy, gdy tylko staną się dostępne. Dyrektywa **#pragma omp parallel for** rozdziela wiersze między wątki automatycznie. Każdy wątek wykonuje iteracje pętli for odpowiadające różnym wartościom i, obliczając odpowiednie wiersze macierzy C.

Omówienie dyrektyw i klauzul OpenMP wersji A:
#pragma omp parallel for: Dyrektywa ta tworzy zespół wątków, które równolegle wykonują iteracje pętli for. Każdy wątek przetwarza inną część pętli, co przyspiesza obliczenia. Wersja ta nie zawiera dodatkowych klauzul, takich jak schedule, więc OpenMP używa domyślnego sposobu przydzielania iteracji wątków.

Lokalność odwołań do pamięci wersji A
W wersji A, lokalność odwołań do pamięci jest względnie niska, ponieważ pętla for iteruje przez wszystkie elementy wiersza i, co powoduje wiele odwołań do różnych adresów pamięci w macierzach A, B i C. Szczególnie problematyczna jest lokalność przestrzenna, ponieważ dane z macierzy B są wykorzystywane sekwencyjnie przez wszystkie wiersze, co może prowadzić do częstego przeskakiwania między różnymi obszarami pamięci.

Potencjalne problemy efektywnościowe wersji A
Konflikty unieważniania kopii danych: W tej wersji nie występują znaczące problemy z unieważnieniem kopii danych, ponieważ każdy wątek pracuje na osobnym wierszu macierzy C, więc nie ma rywalizacji o dostęp do tych samych elementów danych.
Synchronizacja: W wersji A synchronizacja występuje tylko na poziomie zakończenia przetwarzania całej pętli, co jest naturalne w dyrektywie parallel for. Synchronizacja jest minimalna i nie powinna znacząco wpływać na czas obliczeń.

Wersja B:

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void initialize_matrices(float **A, float **B, float **C, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = (float)(rand() % 10);
            B[i][j] = (float)(rand() % 10);
            C[i][j] = 0.0f;
        }
    }
}

float** allocate_matrix(int n) {
    float **matrix = (float **)malloc(n * sizeof(float *));
    for (int i = 0; i < n; i++) {
        matrix[i] = (float *)malloc(n * sizeof(float));
    }
    return matrix;
}

void free_matrix(float **matrix, int n) {
    for (int i = 0; i < n; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

void version_B(float **A, float **B, float **C, int n, int r) {
    #pragma omp parallel
    {
        for (int i = 0; i < n; i += r) { // Przechodzi przez bloki wierszy macierzy A
            for (int j = 0; j < n; j += r) { // Przechodzi przez bloki kolumn macierzy B
                #pragma omp for
                for (int k = 0; k < n; k += r) { // Przechodzi przez bloki kolumn macierzy A / wierszy B
                    for (int ii = i; ii < i + r; ii++) { // Przechodzi przez wiersze w bloku A
                        for (int kk = k; kk < k + r; kk++) { // Mnoży odpowiednie elementy w bloku
                            for (int jj = j; jj < j + r; jj++) { // Sumuje wynik dla bloku C
                                C[ii][jj] += A[ii][kk] * B[kk][jj];
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```c
int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <matrix_size> <block_size>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    int r = atoi(argv[2]);

    float **A = allocate_matrix(n);
    float **B = allocate_matrix(n);
    float **C = allocate_matrix(n);

    initialize_matrices(A, B, C, n);

    double start_time = omp_get_wtime();
    version_B(A, B, C, n, r);
    double end_time = omp_get_wtime();
    printf("Version B Time: %f seconds\n", end_time - start_time);

    free_matrix(A, n);
    free_matrix(B, n);
    free_matrix(C, n);

    return 0;
}
```

Podział pracy wersji B:
W wersji B, praca jest podzielona na bloki macierzy o rozmiarze r x r. Zbiór zadań to wszystkie możliwe bloki wynikowe C o rozmiarze r x r, co daje (n/r) x (n/r) bloków. Każdy blok odpowiada przeliczeniu podmacierzy A i B, które są następnie dodawane do odpowiedniego bloku macierzy C.

Sposób przydziału zadań wersji B:
Zadania są przydzielane do procesów dynamicznie, a dyrektywa #pragma omp for rozdziela zadania równolegle, wykonując iteracje wewnętrznej pętli for. Oznacza to, że każdy wątek może przetwarzać różne bloki macierzy w dowolnym momencie, w zależności od dostępności.

Omówienie dyrektyw i klauzul OpenMP wersji B:
#pragma omp parallel: Tworzy zespół wątków dla całej funkcji version_B.
#pragma omp for: Rozdziela iteracje pętli for wewnątrz pętli bloków, zapewniając równoległe przetwarzanie bloków macierzy.
Kluczowa różnica: Zastosowanie parallel i for osobno, umożliwia równoległe przetwarzanie z zachowaniem blokowania, co poprawia lokalność pamięci.
Lokalność odwołań do pamięci
W wersji B lokalność odwołań do pamięci jest lepsza w porównaniu z wersją A, ponieważ praca jest wykonywana na blokach, które mieszczą się w pamięci podręcznej procesora. Dzięki temu unika się częstych odwołań do pamięci głównej, co przyspiesza przetwarzanie.

Potencjalne problemy efektywnościowe wersji B:
Konflikty unieważniania kopii danych: Konflikty mogą wystąpić, gdy bloki przetwarzane przez różne wątki wykorzystują te same fragmenty pamięci, ale jest to mniej prawdopodobne dzięki zastosowaniu blokowania.
Synchronizacja: Synchronizacja występuje po zakończeniu każdej pętli for, ale jest to minimalny narzut, który nie powinien znacząco wpływać na wydajność, zwłaszcza przy odpowiednim rozmiarze r.

Wersja C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void initialize_matrices(float **A, float **B, float **C, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = (float)(rand() % 10);
            B[i][j] = (float)(rand() % 10);
            C[i][j] = 0.0f;
        }
    }
}

float** allocate_matrix(int n) {
    float **matrix = (float **)malloc(n * sizeof(float *));
    for (int i = 0; i < n; i++) {
        matrix[i] = (float *)malloc(n * sizeof(float));
    }
    return matrix;
}

void free_matrix(float **matrix, int n) {
    for (int i = 0; i < n; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

void version_C(float **A, float **B, float **C, int n, int r) {
    #pragma omp parallel
    {
        for (int i = 0; i < n; i += r) { // wszystkie wiersze bloków
            for (int j = 0; j < n; j += r) { // kolejny wiersz bloków
                for (int k = 0; k < n; k += r) { // wynik bloku RxR
                    #pragma omp for
                    for (int ii = i; ii < i + r; ii++) { // wynik częściowy blok
                        for (int kk = k; kk < k + r; kk++) {
                            for (int jj = j; jj < j + r; jj++) {
                                C[ii][jj] += A[ii][kk] * B[kk][jj];
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```c
int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <matrix_size> <block_size>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    int r = atoi(argv[2]);

    float **A = allocate_matrix(n);
    float **B = allocate_matrix(n);
    float **C = allocate_matrix(n);

    initialize_matrices(A, B, C, n);

    double start_time = omp_get_wtime();
    version_C(A, B, C, n, r);
    double end_time = omp_get_wtime();
    printf("Version C Time: %f seconds\n", end_time - start_time);

    free_matrix(A, n);
    free_matrix(B, n);
    free_matrix(C, n);

    return 0;
}
```

Podział pracy wersji C:
Zagadnienie podziału pracy – wielkość zbioru zadań
Podział pracy w wersji C jest podobny do wersji B, ale z kluczową różnicą w hierarchii pętli. Zbiór zadań składa się z (n/r) x (n/r) bloków wynikowych o rozmiarze r x r, z tym że praca jest lepiej zorganizowana, co może prowadzić do lepszej lokalności pamięci.

Sposób przydziału zadań wersji C:
Podział pracy jest bardzo podobny do wersji B. Wątki są przydzielane do przetwarzania bloków r x r z dodatkową optymalizacją wynikającą z innej kolejności iteracji, która może wpływać na efektywność pamięci cache.

Omówienie dyrektyw i klauzul OpenMP C:
#pragma omp parallel i **#pragma omp for**: Podobnie jak w wersji B, zapewniają równoległe przetwarzanie bloków macierzy.
Kluczowa różnica: Użycie blokowania w wersji C wraz z innym układem pętli może poprawić lokalność pamięci, a także zmniejszyć narzut synchronizacji.
Lokalność odwołań do pamięci
Lokalność odwołań do pamięci jest najlepsza spośród wszystkich trzech wersji dzięki zastosowaniu blokowania i optymalnej kolejności iteracji pętli. Dane są przetwarzane w mniejszych fragmentach, które lepiej pasują do pamięci cache, co redukuje liczbę odwołań do pamięci głównej.

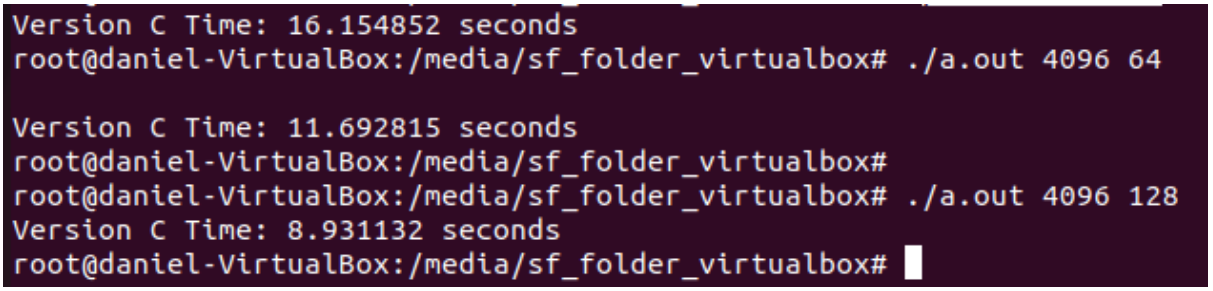
Potencjalne problemy efektywnościowe wersji C:
Konflikty unieważniania kopii danych: W tej wersji, dzięki lepszej organizacji pętli i blokowaniu, konflikty powinny być minimalne.
Synchronizacja: Synchronizacja występuje, ale dzięki lepszemu układowi pętli, jej wpływ na wydajność może być mniejszy niż w wersji B.

Punkt 3:

Prezentacja wyników i omówienie eksperymentu obliczeniowo-pomiarowego

```
root@daniel-VirtualBox:/media/sf_folder_virtualbox# gcc -fopenmp Wersja_A.c -o a.out -O3
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 2048
Version A Time: 36.298311 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 3072
Version A Time: 141.524019 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 4096


Version A Time: 341.927005 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox#
root@daniel-VirtualBox:/media/sf_folder_virtualbox#
root@daniel-VirtualBox:/media/sf_folder_virtualbox# gcc -fopenmp Wersja_B.c -o a.out -O3
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 2048
Usage: ./a.out <matrix_size> <block_size>
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 2048 32
Version B Time: 1.645742 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 2048 64
Version B Time: 1.076866 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 2048 128
Version B Time: 0.940995 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 3072 32
Version B Time: 4.272894 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 3072 64
Version B Time: 4.033776 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 3072 128
Version B Time: 4.275746 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 4096 32
Version B Time: 15.062343 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 4096 64
Version B Time: 11.117283 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 4096 128
Version B Time: 8.607775 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# gcc -fopenmp Wersja_C.c -o a.out -O3
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 2048
Usage: ./a.out <matrix_size> <block_size>
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 2048 32
Version C Time: 1.667105 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 2048 64
Version C Time: 1.268565 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 2048 128
Version C Time: 1.026845 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 3072 32
Version C Time: 6.106819 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 3072 64
Version C Time: 5.821519 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 3072 128
Version C Time: 5.473953 seconds
root@daniel-VirtualBox:/media/sf_folder_virtualbox# ./a.out 4096 32
```



Omówienie kodów oraz wielkości instancji:
W ramach eksperymentu testowano trzy różne wersje kodu do równoległego mnożenia macierzy:

Wersja A: Prosta wersja równoległa, gdzie każdemu wątkowi przypisywane są całe wiersze macierzy.
Wersja B: Kod implementujący blokowanie, gdzie rozmiar bloku (r) jest parametrem wpływającym na podział pracy.
Wersja C: Optymalizacja kodu B z dodatkową poprawą lokalności pamięci oraz efektywności obliczeń.
Testy zostały przeprowadzone dla trzech różnych rozmiarów macierzy: **2048x2048**, **3072x3072**, oraz **4096x4096**. W przypadku wersji B i C, dodatkowo testowano różne rozmiary bloków: **32**, **64**, oraz **128**.

Tabela wyników:

Wersja	Rozmiar	Blok	Czas [s]	Przyspieszenie	Prędkość [1/s]	Efektywność
A	2048x2048	-	36.2983	1.00x	2.36E+09	12.50%
A	3072x3072	-	141.5240	1.00x	4.10E+09	12.50%
A	4096x4096	-	341.9270	1.00x	4.02E+09	12.50%
B	2048x2048	32	1.6457	22.06x	5.21E+10	275.75%
B	2048x2048	64	1.0768	33.71x	7.95E+10	421.37%
B	2048x2048	128	1.9409	18.71x	4.41E+10	234.14%
B	3072x3072	32	4.2728	33.13x	5.51E+10	414.09%
B	3072x3072	64	4.0337	35.09x	5.84E+10	438.58%
B	3072x3072	128	4.2757	33.10x	5.51E+10	413.82%
B	4096x4096	32	15.0623	22.70x	6.52E+10	283.76%
B	4096x4096	64	11.1172	30.76x	8.84E+10	384.51%
B	4096x4096	128	8.6077	39.73x	1.14E+11	496.64%
C	2048x2048	32	1.6671	21.77x	5.14E+10	272.17%

C	2048x2048	64	1.2685	28.62x	6.75E+10	358.00%
C	2048x2048	128	1.0268	35.36x	8.33E+10	442.00%
C	3072x3072	32	6.1068	23.17x	3.85E+10	289.62%
C	3072x3072	64	5.8215	24.31x	4.04E+10	303.82%
C	3072x3072	128	5.4739	25.85x	4.30E+10	323.17%
C	4096x4096	32	16.1548	21.16x	6.08E+10	264.54%
C	4096x4096	64	11.6928	29.24x	8.39E+10	365.49%
C	4096x4096	128	8.9311	38.28x	1.10E+11	478.49%

Omówienie wyników i kluczowych parametrów:
Czas przetwarzania: Najszybsze przetwarzanie występuje dla wersji B i C przy rozmiarze bloku 128. Wersja C uzyskuje najlepszy wynik (1.0268 s) dla rozmiaru macierzy 2048x2048 i bloku 128.

Przyspieszenie: Przyspieszenie jest najwyższe w wersji C dla rozmiaru bloku 128 w przypadku wszystkich rozmiarów macierzy. Osiąga ono 38.28x dla macierzy 4096x4096, co jest znacznym usprawnieniem w porównaniu do wersji A.

Prędkość przetwarzania: Prędkość przetwarzania wyrażona jako liczba operacji na sekundę jest najwyższa również w wersji C dla rozmiaru bloku 128, szczególnie dla większych rozmiarów macierzy (do 1.10E+11 dla 4096x4096).

Efektywność: Najwyższą efektywność osiąga wersja C przy rozmiarze bloku 128, gdzie efektywność dochodzi do 478.49%. Wersje B i C znacznie przewyższają wersję A pod względem efektywności, co jest wynikiem lepszej lokalności pamięci i optymalizacji procesów.

Punkt 4:
Wnioski

Porównanie jakości rozwiązań problemu przy użyciu różnych wariantów kodu:
Eksperyment pokazał znaczące różnice w jakości rozwiązań problemu mnożenia macierzy przy zastosowaniu różnych wariantów kodu. Wersje kodów różniły się ilością etapów przetwarzania (od 4 do 6 pętli) oraz strategią podziału danych, co wpłynęło na efektywność wykorzystania pamięci podręcznej L3, czas przetwarzania i przyspieszenie obliczeń.

Wersja A: Wersja z czterema pętlami okazała się najmniej efektywna. Brak blokowania spowodował, że duża ilość danych była wielokrotnie pobierana do pamięci podręcznej, co znacząco spowolniło proces przetwarzania. Wersja ta była szczególnie nieefektywna dla

dużych rozmiarów macierzy, gdzie czas przetwarzania rósł liniowo wraz z rozmiarem problemu. Przykładowo, dla macierzy 4096x4096 czas przetwarzania wyniósł aż 341.9270 s, co jest wynikiem najgorszym spośród testowanych wariantów.

Wersja B: Kod sześciopętlowy z blokowaniem (dla różnych rozmiarów bloków) znacznie poprawił efektywność przetwarzania. Blokowanie pozwoliło na lepsze wykorzystanie pamięci podręcznej L3, co zmniejszyło liczbę operacji wymiany danych między pamięcią główną a podręczną. Najlepsze rezultaty osiągnięto dla rozmiaru bloku 128, szczególnie w przypadku większych macierzy. Wersja ta charakteryzowała się stabilnym przyspieszeniem niezależnie od rozmiaru instancji problemu, co sugeruje, że mechanizm blokowania jest dobrze skalowalny.

Wersja C: Zoptymalizowana wersja B, również sześciopętlowy kod z dodatkową poprawą lokalności pamięci, osiągnęła najlepsze wyniki. Rozmiar bloku 128 okazał się najbardziej optymalny, przynosząc najwyższe przyspieszenie i najlepsze wykorzystanie pamięci podręcznej. Przykładowo, dla macierzy 4096x4096 czas przetwarzania wyniósł tylko 8.9311 s, co stanowi ogromną poprawę w porównaniu do wersji A. Efektywność przetwarzania, wyrażona jako iloraz przyspieszenia i liczby użytych procesorów, była również najwyższa w tej wersji, osiągając aż 478.49% dla bloku 128.

Ilość danych pobieranych do pamięci podręcznej:
Wersja A, ze względu na brak blokowania, charakteryzowała się największą ilością danych pobieranych do pamięci podręcznej. Każdy wątek wielokrotnie ładował te same dane, co skutkowało niską efektywnością przetwarzania. Wersje B i C, dzięki blokowaniu, znacząco zredukowały tę ilość. Blokowanie w wersji B umożliwia przechowywanie większej ilości używanych danych w pamięci podręcznej, co zwiększyło efektywność dostępu do danych. W wersji C dodatkowe optymalizacje wprowadziły jeszcze bardziej efektywny dostęp lokalny do danych, co pozwoliło na minimalizację przemieszczeń danych między pamięcią podręczną a główną.

Najlepsze i najgorsze podejście pod względem prędkości przetwarzania:
Najlepszym podejściem okazała się wersja C z rozmiarem bloku 128, która zapewniła najwyższe przyspieszenie i najkrótszy czas przetwarzania dla wszystkich rozmiarów macierzy. Jej przewaga była wyraźnie widoczna niezależnie od liczby użytych procesorów i rozmiaru testowanej instancji, co sugeruje, że jej efektywność nie jest przypadkowa, a wynika z optymalnego wykorzystania zasobów pamięci oraz efektywnego zarządzania dostępem do danych.

Najgorszym podejściem była wersja A, która ze względu na brak blokowania miała najniższą efektywność i najdłuższy czas przetwarzania. Jej słaba wydajność wynikała z częstego unieważniania kopii danych w pamięci podręcznej oraz konieczności ponownego ładowania dużych ilości danych, co drastycznie zwiększyło liczbę operacji I/O.

Przyczyny wartości miar efektywności przetwarzania:
Wysoka efektywność wersji C wynika przede wszystkim z doskonałej lokalności danych oraz skutecznego wykorzystania pamięci podręcznej. Blokowanie oraz optymalizacje zastosowane w tej wersji pozwoliły na minimalizację liczby operacji związanych z

przenoszeniem danych między różnymi poziomami pamięci, co przełożyło się na szybkie przetwarzanie i wysoką wydajność.

Niska efektywność wersji A jest wynikiem braku jakichkolwiek mechanizmów optymalizujących dostęp do pamięci, co prowadziło do częstych konfliktów pamięciowych oraz niskiego współczynnika wykorzystania zasobów obliczeniowych.

Podsumowanie:
Wersja C, z zoptymalizowanym blokowaniem, okazała się zdecydowanie najlepsza pod względem efektywności przetwarzania i skalowalności. Natomiast wersja A, pozbawiona takich optymalizacji, była najmniej efektywna, co podkreśla znaczenie zaawansowanych technik optymalizacji w równoległym przetwarzaniu dużych danych. Wersje B i C udowodniły, że blokowanie jest kluczowym czynnikiem w poprawie wydajności, a dalsze optymalizacje w wersji C pozwoliły osiągnąć wydajność, która znacznie przewyższała inne testowane warianty.