

POLITECHNIKA POZNAŃSKA

WYDZIAŁ INFORMATYKI

PRZETWARZANIE RÓWNOLEGŁE

---

# Równoległe sumowanie komórek pamięci za pomocą wielu wątków przetwarzania

---

*Autorzy:*

Adam SZCZEPAŃSKI  
Mateusz CZAJKA

*Prowadzący:*

dr Rafał WALKOWIAK



2 marca 2014

## Spis treści

<b>1</b>	<b>Informacje o projekcie</b>	<b>3</b>
1.1	Dane autorów . . . . .	3
1.2	Historia projektu . . . . .	3
<b>2</b>	<b>Wstęp</b>	<b>4</b>
2.1	Opis problemu . . . . .	4
2.2	Punkt odniesienia (algorytm sekwencyjny – kolejność ij) . . . . .	4
2.3	Badane algorytmy . . . . .	5
2.3.1	Algorytm sekwencyjny – kolejność ji . . . . .	5
2.3.2	Algorytm zrównoleglony – kolejność ij . . . . .	5
2.3.3	Algorytm zrównoleglony – kolejność ji . . . . .	5
2.3.4	Algorytm na sekwencyjność pamięci . . . . .	6
2.3.5	Algorytm na pobranie z wyprzedzeniem . . . . .	6
2.4	Maska powinowactwa . . . . .	7
2.4.1	Kod . . . . .	7
2.4.2	Rezultat . . . . .	7
<b>3</b>	<b>Pomiary efektywności</b>	<b>8</b>
3.1	Przyspieszenie obliczeń równoległych . . . . .	8
3.1.1	Pomiary . . . . .	8
3.1.2	Podsumowanie . . . . .	9
3.2	Ilość wykonanych instrukcji na jeden cykl procesora . . . . .	9
3.3	Współczynniki braku trafień do pamięci . . . . .	10
3.3.1	Znaczenie danych . . . . .	10
3.3.2	Pomiary . . . . .	10
3.3.3	Omówienie . . . . .	11
<b>4</b>	<b>Wpływ rozmiaru danych</b>	<b>12</b>
4.1	Wstęp . . . . .	12
4.2	Wyniki . . . . .	12
4.3	Podsumowanie . . . . .	12
<b>5</b>	<b>Wpływ sekwencyjności pamięci</b>	<b>12</b>
5.1	Wstęp . . . . .	12
5.2	Wyniki . . . . .	13
<b>6</b>	<b>Wyprzedzające pobranie danych</b>	<b>13</b>
6.1	Klasyczne podejście . . . . .	13
6.2	Zastosowanie kwalifikatora <i>volatile</i> . . . . .	14
<b>7</b>	<b>Podsumowanie</b>	<b>14</b>
<b>8</b>	<b>Załączniki</b>	<b>15</b>
<b>9</b>	<b>Kod źródłowy</b>	<b>15</b>

Spis rysunków	19
Spis tablic	20
Spis kodów źródłowych	21

# **1 Informacje o projekcie**

## **1.1 Dane autorów**

Mateusz Czajka	106596
Adam Szczepański	106593

## **1.2 Historia projektu**

1. Jest to pierwsza wersja projektu. Dokumentacja elektroniczna została przesłana w dniu 3 marca 2014.

## 2 Wstęp

### 2.1 Opis problemu

Głównym założeniem projektu było zapoznanie się biblioteką OpenMP na podstawie równoległego sumowania komórek tablicy. W ramach projektu zrealizowaliśmy 4 algorytmy sekwencyjne oraz 2 algorytmy zrównoleglone. Celem zastosowania czterech różnych algorytmów sekwencyjnych było zbadanie wpływu sekwencyjności pamięci podręcznej, wyprzedzającego pobrania danych do pamięci podręcznej oraz kolejności uszeregowania pętli na czas realizacji zadania. W przypadku algorytmów zrównoleglonych badaliśmy wpływ kolejności uszeregowania pętli na końcowy rezultat.

Nasze badania podzieliliśmy na 3 spójne części. Badaliśmy wpływ

- rozmiaru danych
- sekwencyjności pamięci
- wyprzedzającego pobrania

na czas realizacji problemu.

Sam problem sprowadzał się do zsumowania wartości komórek w tabeli. Dla zachowania czytelności w kolejności pętli zastosowaliśmy tablicę dwuwymiarową. Ponieważ sam problem jest prosty obliczeniowo zmuszeni byliśmy do stosowania bardzo dużych tablic typu `int` ( $\text{sizeof}(\text{int}) = 4$ ), o rozmiarach 1GB ( $[2^{24} - \text{wierszy}]$  na  $[2^4 - \text{kolumn}]$ ) i 256MB ( $[2^{22} - \text{wierszy}]$  na  $[2^4 - \text{kolumn}]$ ). W badanym systemie długość linii pamięci podręcznej wynosiła 64B, co odpowiada szerokości naszej tabeli. Aby zapewnić wyrównanie do początku linii PP tabela została zdefiniowana z wykorzystaniem rozszerzenia Visual Studio:

```
__declspec(aligned(64)) int tab[ROWS][COLS];
```

### 2.2 Punkt odniesienia (algorytm sekwencyjny – kolejność ij)

Punktem odniesienia dla wszystkich algorytmów był podstawowy algorytm sekwencyjny w którym sumowaliśmy elementy tablicy wierszami. Nazywany dalej *sum\_ij*.

```
__declspec(noinline) int sum_ij() {  
    int sum = 0;  
    for (int i=0; i<ROWS; i++) {  
        for (int j=0; j<COLS; j++) {  
            sum += tab[i][j];  
        }  
    }  
}
```

```

    return sum;
}

```

Listing 2: Algorytm sekwencyjny – kolejność ij.

## 2.3 Badane algorytmy

### 2.3.1 Algorytm sekwencyjny – kolejność ji

Algorytm sekwencyjny ze zmienioną kolejnością pętli (sumujemy kolumnami). Nazywany dalej *sum\_ji*.

```

__declspec(noinline) int sum_ji() {
    int sum = 0;
    for (int j=0; j<COLS; j++) {
        for (int i=0; i<ROWS; i++) {
            sum += tab[i][j];
        }
    }
    return sum;
}

```

Listing 3: Algorytm sekwencyjny – kolejność ji

### 2.3.2 Algorytm zrównoleglony – kolejność ij

Algorytm sumujący wierszami realizowany równolegle. Nazywany dalej *sum\_par\_ij*.

```

__declspec(noinline) int sum_par_ij() {
    int sum = 0;
    int i;
#pragma omp parallel for default(none) shared(tab) private(i)
    reduction(+:sum)
    for (i=0; i<ROWS; i++) {
        for (int j=0; j<COLS; j++) {
            sum += tab[i][j];
        }
    }
    return sum;
}

```

Listing 4: Algorytm zrównoleglony – kolejność ij.

### 2.3.3 Algorytm zrównoleglony – kolejność ji

Algorytm realizowany równolegle ze zmienioną kolejnością pętli (sumujemy kolumnami). Nazywany dalej *sum\_par\_ji*.

```

__declspec(noinline) int sum_par_ji() {

```

```

int sum = 0;
int j;
#pragma omp parallel for default(none) shared(tab) private(j)
    reduction(+:sum)
    for (j=0; j<COLS; j++) {
        for (int i=0; i<ROWS; i++) {
            sum += tab[i][j];
        }
    }
return sum;
}

```

Listing 5: Algorytm zrównoleglony – kolejność *ji*.

### 2.3.4 Algorytm na sekyjność pamięci

Algorytm sekwencyjny w którym badaliśmy wpływ sekyjności pamięci na czas wykonania zadania. Nazywany dalej *sum\_sec*.

```

__declspec(noinline) int sum_sec() {
    int sum = 0;
    for (int j=0; j<COLS; j++) {
        for (int k=0; k<CACHE_LINES_ON_PAGE; k++) {
            for (int i=k; i<ROWS; i+=CACHE_LINES_ON_PAGE) {
                sum += tab[i][j];
            }
        }
    }
    return sum;
}

```

Listing 6: Algorytm na sekyjność pamięci.

### 2.3.5 Algorytm na pobranie z wyprzedzeniem

Algorytm sekwencyjny w którym badaliśmy wpływ wyprzedzającego pobrania danych do pamięci podręcznej na czas realizacji zadania. Nazywany dalej *sum\_pf*.

```

int tmp;

__declspec(noinline) int sum_pf() {
    int sum = 0;
    int i;
    for (i=0; i<ROWS-1; i++) {
        for (int j=0; j<COLS; j++) {
            sum += tab[i][j];
            tmp = tab[i+1][j];
        }
    }
}

```

```

    }

    for (int j=0; j<COLS; j++) {
        sum += tab[i][j];
    }
    return sum;
}

```

Listing 7: Algorytm na pobranie z wyprzedzeniem.

Ostatnia iteracja jest poza pętlą w celu uniknięcia konieczności sprawdzania czy nie został przekroczony zakres tablicy przy przypisaniu to *tmp*.

## 2.4 Maska powinowactwa

Aby wyeliminować przełączanie wykonywania wątku pomiędzy rdzeniami procesora zastosowaliśmy maski powinowactwa.

### 2.4.1 Kod

```

#pragma omp parallel
{
    const int processor_count = 4;
    int th_id=omp_get_thread_num();

    DWORD_PTR mask = (1 << (th_id % processor_count));
    DWORD_PTR result = SetThreadAffinityMask(thread_handle, mask);

    if (result==0) {
        printf("error SetThreadAffnityMask\n");
    }

    else {
        printf("previous mask for thread %d : %d\n",th_id,result);
        printf("new mask for thread %d : %d\n",
            th_id,SetThreadAffinityMask(thread_handle, mask));
    }
}

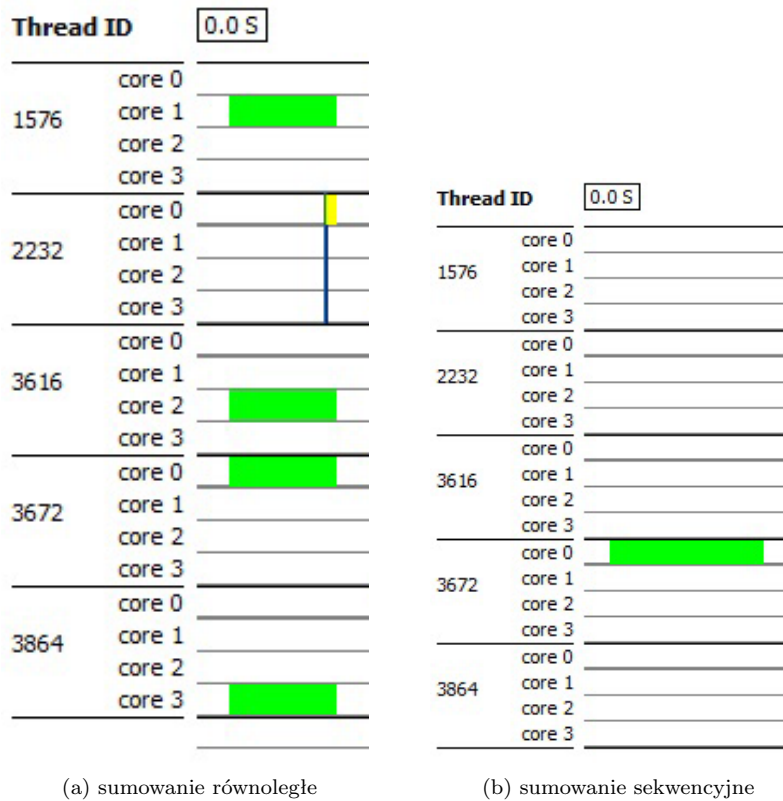
```

Listing 8: Ustawienie maski powinowactwa.

### 2.4.2 Rezultat

W rezultacie dla przetwarzania równoległego każdy wątek odpowiedzialny za sumowanie realizowany jest na oddzielnym procesorze. Co widoczne jest na poniższych wykresach:





Rysunek 1: Wykres przydziału wątków do rdzeni.

### 3 Pomiary efektywności

#### 3.1 Przyspieszenie obliczeń równoległych

##### 3.1.1 Pomiary

Algorytm	Czas wykonania w <i>ms</i>	Przyspieszenie względem <i>sum_ij</i>
<i>sum_ij</i>	253	1.000
<i>sum_ji</i>	2362	0.107
<i>sum_par_ij</i>	87	2.908
<i>sum_par_ji</i>	1189	0.213

Tablica 1: Porównanie szybkości wybranych algorytmów wobec *sum\_ij*

Warto zwrócić uwagę na fakt, że oprócz braku lub obecności zrównoleglenia znaczący wpływ na szybkość przetwarzania ma kolejność uszeregowania pętli.

Zostało to szerzej omówione w sekcji 3.3. Przyspieszenia dla poszczególnych uszeregowień pętli prezentują się następująco:

Algorytm	Czas wykonania w <i>ms</i>	Przyspieszenie względem <i>sum_ij</i>
<i>sum_ij</i>	2362	1.000
<i>sum_par_ij</i>	87	2.908

Tablica 2: Porównanie szybkości wybranych algorytmów (o kolejności pętli *ij*) wobec *sum\_ij*

Algorytm	Czas wykonania w <i>ms</i>	Przyspieszenie względem <i>sum_ji</i>
<i>sum_ji</i>	2362	1.000
<i>sum_par_ji</i>	1189	1.987

Tablica 3: Porównanie szybkości wybranych algorytmów (o kolejności pętli *ji*) wobec *sum\_ji*

### 3.1.2 Podsumowanie

Zrównoleglenie przetwarzania znacząco przyspiesza jego czas przetwarzania. Znaczny wpływ na wielkość przyspieszenia ma kolejność uszeregowania pętli. W przybliżeniu jest to:

- trzykrotne przyspieszenie dla uszeregowania pętli *ij*
- dwukrotne przyspieszenie dla uszeregowania pętli *ji*

## 3.2 Ilość wykonanych instrukcji na jeden cykl procesora

IPC (instructions per cycle) jest jednym z wyznaczników prędkości procesora. Oznacza on liczbę wykonywanych instrukcji przez procesor w jednym cyklu zegara. Wskaźnik IPC obliczaliśmy dla każdej funkcji na podstawie wzoru:

$$IPC = \frac{ret\_instr}{CPU\_clocks}. \quad (1)$$

CPI (cycles per instruction) jest odwrotnością IPC:

$$CPI = \frac{CPU\_clocks}{ret\_instr}. \quad (2)$$

Algorytm	<i>ret_instr</i>	<i>CPU_clocks</i>	<i>IPC</i>	<i>CPI</i>
<i>sum_ij</i>	12640	14736	0.86	1.17
<i>sum_ji</i>	9780	136048	0.07	13.91
<i>sum_par_ij</i>	12684	20096	0.63	1.58
<i>sum_par_ji</i>	11524	259628	0.04	22.53
<i>sum_sec</i>	14420	1139472	0.01	79.02
<i>sum_pf</i>	14996	13140	1.14	0.88

Tablica 4: Wartości IPC i CPI dla poszczególnych algorytmów

Zgodnie z oczekiwaniami najwyższe wartości wskaźnika IPC (a zarazem najniższe CPI) występują w przypadku algorytmów, które spełniają zasadę lokalności przestrzennej, są to:

- *sum\_ij*
- *sum\_par\_ij*
- *sum\_pf*

IPC dla wyżej wymienionej grupy algorytmów jest kilkunasto, a w niektórych przypadkach kilkudziesięcio krotnie większe niż dla algorytmów z pozostałej grupy.

### 3.3 Współczynniki braku trafień do pamięci

#### 3.3.1 Znaczenie danych

Nazwa	Znaczenie
DC accesses	ilość odwołań do pamięci podręcznej
DC misses	ilość chybień do pamięci podręcznej
L2 requests	ilość odwołań do pamięci L2
L2 misses	ilość chybień do pamięci L2
L2 fill write	zapis danych w L2 powodowany usunięciem ich z pp L1
DRAM accesses	ilość odwołań do pamięci dynamicznej

Tablica 5: Badane dane

#### 3.3.2 Pomiary

Wartości *DC\_accesss* oraz *DC\_missess* odczytaliśmy bezpośrednio z programu CodeAnalyst. Natomiast *DC\_miss\_ratio* oraz *DC\_miss\_rate* obliczyliśmy w następujący sposób:

$$DC\_miss\_rate = \frac{DC\_missess}{ret\_instr} \quad (3)$$

$$DC\_miss\_ratio = \frac{DC\_missess}{DC\_accesss} \quad (4)$$

Algorytm	DC accessess	DC missess	DC miss ratio	DC miss rate
<i>sum_ij</i>	5792	24	0%	0%
<i>sum_ji</i>	7620	4068	53%	42%
<i>sum_par_ij</i>	5188	20	0%	0%
<i>sum_par_ji</i>	7652	4456	58%	38%
<i>sum_sec</i>	18808	5256	28%	36%
<i>sum_pf</i>	9928	24	0%	0%

Tablica 6: Pomiary związane z odwołaniami do pamięci L1

Podobnie jak w przypadku pamięci L1 wartości *L2\_requests*, *L2\_misses* oraz *L2\_fill\_write* odczytaliśmy bezpośrednio z programu CodeAnalyst. Natomiast *L2\_miss\_ratio* oraz *L2\_miss\_rate* obliczyliśmy w następujący sposób:

$$L2\_miss\_rate = \frac{L2\_missess}{ret\_instr} \quad (5)$$

$$L2\_miss\_ratio = \frac{L2\_missess}{L2\_requests + L2\_fill\_write} \quad (6)$$

Algorytm	L2 requests	L2 missess	L2 fill write	L2 miss rate	L2 miss ratio
<i>sum_ij</i>	663	340	684	3%	25%
<i>sum_ji</i>	5452	7209	10908	56%	30%
<i>sum_par_ij</i>	1431	352	668	3%	17%
<i>sum_par_ji</i>	5472	23736	10844	47%	16%
<i>sum_sec</i>	11024	15945	21668	76%	29%
<i>sum_pf</i>	344	693	684	2%	25%

Tablica 7: Pomiary związane z odwołaniami do pamięci L2

### 3.3.3 Omówienie

Znowu można wyróżnić 2 grupy algorytmów. Pierwsza z nich składająca się z:

- *sum\_ij*
- *sum\_par\_ij*
- *sum\_pf*

cechują się one lokalnością przestrzenną. Widać to bardzo wyraźnie przy współczynnikach *DC\_miss\_ratio* oraz *DC\_miss\_rate*. Dla wyżej wymienionej grupy

wynoszą one w przybliżeniu 0%, a dla pozostałych algorytmów tj *sum\_ji*, *sum\_par\_ji* oraz *sum\_sec* po kilkadziesiąt procent. Związane to jest z faktem że grupa posiadająca cechę lokalności przestrzennej sumuje wierszami, zatem z raz wczytanej linii do pamięci podręcznej wszystkie elementy są wykorzystywane jeden po drugim.

## 4 Wpływ rozmiaru danych

### 4.1 Wstęp

W celu sprawdzenia wpływu rozmiaru danych na czas realizacji zadania dokonaliśmy pomiarów czasów dla dwóch rozmiarów danych:

- Instancja A:  $tab[2^{24}][2^4]$
- Instancja B:  $tab[2^{22}][2^4]$

Spodziewaliśmy się, że czas obliczeń dla Instancji B będzie w przybliżeniu czterokrotnie krótszy.

### 4.2 Wyniki

Wyniki pomiarów prezentują się następująco:

Problem	<i>sum_ij</i>	<i>sum_ji</i>	<i>sum_sec</i>	<i>sum_pf</i>	<i>sum_par_ij</i>	<i>sum_par_ji</i>
Instancja A	253	2352	19552	225	87	1189
Instancja B	62	590	4890	57	23	304
Stosunek A/B	4.081	4.003	3.998	3.947	3.783	3.911

Tablica 8: Czas realizacji kodu dla poszczególnych funkcji w *ms*.

### 4.3 Podsumowanie

Wyniki eksperymentu są bardzo zadowalające. W przypadku każdego algorytmu rozmiar danych ma liniowy wpływ na czas przetwarzania. Jest to zgodne ze złożonością algorytmu która jest liniowa wobec  $n$  gdzie  $n$  to *liczba\_wierszy* \* *liczba\_kolumn*.

W każdym przypadku czas realizacji dla Instancji B był około 4 razy krótszy niż dla Instancji A. Średnia tych wartości wynosi 3.954.

## 5 Wpływ sekcijności pamięci

### 5.1 Wstęp

W trakcie naszych badań sprawdziliśmy wpływ sekcijności pamięci na czas przetwarzania. Kluczowe dla tego punktu algorytmy to:

- *sum\_ij* - podstawowy algorytm sekwencyjny, służący jako punkt odniesienia
- *sum\_sec* - algorytm sekwencyjny w którym badaliśmy sekwencyjność pamięci

W wykorzystywanym systemie długość linii PP wynosi  $64B$ , a liczba linii w pojedynczej stronie wynosi 512.

W metodzie *sum\_sec* w kolejnych iteracjach odwołujemy się do komórki tabeli z tej samej kolumny, z wiersza przesuniętego o ilość linii w stronie. Powoduje to konieczność wymiany całej linii PP w każdej iteracji.

## 5.2 Wyniki

Zgodnie z założeniem, powinien to być najwolniejszy algorytm, ponieważ spodziewana jest bardzo wysoka liczba odwołań do pamięci RAM or wysoki stosunek braku trafień do pamięci podręcznej.

Algorytm	Czas [ms]	DC missess	L2 misses	DRAM access
<i>sum_ij</i>	253	24	340	352
<i>sum_sec</i>	19552	5256	11024	5920

Tablica 9: Porównanie wybranych algorytmów pod kątem sekwencyjności pamięci

Wyniki przerosły nasze oczekiwania - *sum\_sec* wykonał się ponad 77 razy wolniej. Pokazuje to jest wielkie znaczenie na szybkość realizacji przetwarzania ma zachowanie cechy lokalności przestrzennej. Wyraźnie widać, że w przypadku algorytmu użytego jako punkt odniesienia liczba braków trafień do pamięci podręcznej L1 była znacznie mniejsza. Sytuacja wygląda bardzo podobnie dla pamięci podręcznej L2. Konsekwencją braku trafień do pamięci podręcznej jest konieczność większej liczby pobrań danych z pamięci dynamicznej co odzwierciedla współczynnik *DRAM\_accesses*.

## 6 Wyprzedzające pobranie danych

Wyprzedzające pobranie danych potraktowaliśmy jako swego rodzaju eksperyment. Nasze obawy związane z optymalizacją kody okazały się być uzasadnione. W celu realizacji problemu dodaliśmy zmienną *tmp*, na której nie były realizowane żadne operacje. Eksperyment podzieliliśmy na dwie części.

### 6.1 Klasyczne podejście

Początkowo spodziewaliśmy się, że kompilator usunie nic nie robiące przepisanie. Po przejrzeniu instrukcji Assemblera okazało się, że jest inaczej.

```

  0xb41234  94      sum += tab[i][j];
  0xb41234      add eax,[ecx-04h]      03 41 FC
  0xb41237      add edx,[ecx]      03 11
  0xb41239      add esi,[ecx+04h]     03 71 04
  0xb4123c      add edi,[ecx+08h]     03 79 08
  0xb4123f  95      tmp = tab[i+1][j];
  0xb4123f      mov ebx,[ecx+48h]     8B 59 48
  0xb41242      add ecx,10h      83 C1 10
  0xb41245      dec dword [ebp-04h]   FF 4D FC
  0xb41248      mov [404033c8h],ebx   89 1D C8 33 40 40
  0xb4124e      jnz $-1ah (0xb41234)  75 E4

```

Rysunek 2: Instrukcja przypisania do zmiennej *tmp*

Algorytm	Czas [ms]
<i>sum_ij</i>	253
<i>sum_pf</i>	225

Tablica 10: Czasy algorytmów kluczowych dla wyprzedzającego pobrania

Przyspieszenie związane z wyprzedzającym pobraniem danych okazało się jednak być bardzo nieznaczne.

## 6.2 Zastosowanie kwalifikatora *volatile*

*volatile* w C++ jest kwalifikatorem typu informującym kompilator, że wartość zmiennej może się zmienić bez jego wiedzy i kontroli i że w związku z tym kompilator powinien zrezygnować z agresywnej optymalizacji i przy każdym odwołaniu do tej zmiennej wczytać nową wartość z komórki pamięci.

Kod odpowiedzialny za inicjalizację zmiennej *tmp* prezentował się następująco:

```
volatile int tmp;
```

Zastosowanie wyżej wymienionego kwalifikatora spowodowało zgodnie z oczekiwaniami pogorszenie czasu przetwarzania.

## 7 Podsumowanie

Zrealizowane zadania pozwoliły nam na lepsze poznanie przetwarzania równoległego a także wagi kolejności odwołań do pamięci. Dwa główne wnioski jakie można wyciągnąć to:

- przetwarzanie równoległe jest szybsze od sekwencyjnego (doskonale widać poprzez porównanie algorytmów *sum\_ij* oraz *sum\_par\_ij*)
- ważne podczas pisania kodu jest zaplanowanie dostępu do pamięci, zapewnienie własności lokalności czasowej i przestrzennej w znaczący sposób przyspiesza działanie programu (*sum\_ij* 77 krotnie szybszy od *sum\_sec*)

Najszybszy algorytm (*sum\_par\_ij*) był blisko 225 razy szybszy od najwolniejszego (*sum\_sec*). Było dla nas sporym zaskoczeniem, że takie różnice można uzyskać dla wydawało by się zwykłego sumowania komórek z tablicy.

Dodatkowo warto zauważyć, że rozmiar danych liniowo (w naszym przypadku) ogranicza czas wykonania programu niezależnie czy obliczenia są wykonywane równoległe czy sekwencyjnie.

## 8 Załączniki

1. sum.cpp – plik z kodem źródłowym

## 9 Kod źródłowy

```
1  #include <stdio>
2  #include <stdlib>
3  #include <ctime>
4  #include <Windows.h>
5  #include <omp.h>
6
7  #define CACHE_LINE 64
8  #define CACHE_ALIGN __declspec(align(CACHE_LINE))
9  #define CACHE_LINES_ON_PAGE 512
10
11 #define ROWS ((size_t) 1<<24)
12 #define COLS ((size_t) 1<<4)
13
14 CACHE_ALIGN int tab[ROWS][COLS];
15 double start;
16 HANDLE thread_handle = GetCurrentThread();
17
18 inline void start_clock() {
19     start = (double) clock() / CLK_TCK;
20 }
21
22 inline void print_elapsed_time(const char* operation_name) {
23     double elapsed;
24     double resolution;
25
26     elapsed = (double) clock() / CLK_TCK;
27     resolution = 1.0 / CLK_TCK;
28
29     printf("%s: %8.4f sec\n", operation_name, elapsed-start);
30 }
31
32 void init_tab() {
```



```

33     start_clock();
34     int i;
35     #pragma omp parallel for shared(tab) private(i)
36     for (i=0; i<ROWS; i++) {
37         for (int j=0; j<COLS; j++) {
38             tab[i][j] = rand();
39         }
40     }
41     print_elapsed_time("init");
42 }
43
44
45 typedef int (*function)();
46 inline void run_function(function f, const char* operation_name) {
47     printf("-----\n");
48     start_clock();
49     int sum = f();
50     printf("sum: %d\n", sum);
51     print_elapsed_time(operation_name);
52 }
53
54 __declspec(noinline) int sum_ij() {
55     int sum = 0;
56     for (int i=0; i<ROWS; i++) {
57         for (int j=0; j<COLS; j++) {
58             sum += tab[i][j];
59         }
60     }
61     return sum;
62 }
63
64 __declspec(noinline) int sum_ji() {
65     int sum = 0;
66     for (int j=0; j<COLS; j++) {
67         for (int i=0; i<ROWS; i++) {
68             sum += tab[i][j];
69         }
70     }
71     return sum;
72 }
73
74 __declspec(noinline) int sum_sec() {
75     int sum = 0;
76     for (int j=0; j<COLS; j++) {
77         for (int k=0; k<CACHE_LINES_ON_PAGE; k++) {
78             for (int i=k; i<ROWS; i+=CACHE_LINES_ON_PAGE) {
79                 sum += tab[i][j];
80             }
81         }
82     }

```

```

83     return sum;
84 }
85
86 int tmp;
87
88 __declspec(noinline) int sum_pf() {
89     int sum = 0;
90     int i;
91     for (i=0; i<ROWS-1; i++) {
92         for (int j=0; j<COLS; j++) {
93             sum += tab[i][j];
94             tmp = tab[i+1][j];
95         }
96     }
97
98     for (int j=0; j<COLS; j++) {
99         sum += tab[i][j];
100     }
101     return sum;
102 }
103
104 __declspec(noinline) int sum_par_ij() {
105     int sum = 0;
106     int i;
107     #pragma omp parallel for default(none) shared(tab) private(i)
108         reduction(+:sum)
109     for (i=0; i<ROWS; i++) {
110         for (int j=0; j<COLS; j++) {
111             sum += tab[i][j];
112         }
113     }
114     return sum;
115 }
116
117 __declspec(noinline) int sum_par_ji() {
118     int sum = 0;
119     int j;
120     #pragma omp parallel for default(none) shared(tab) private(j)
121         reduction(+:sum)
122     for (j=0; j<COLS; j++) {
123         for (int i=0; i<ROWS; i++) {
124             sum += tab[i][j];
125         }
126     }
127     return sum;
128 }
129
130 int main(int argc, char* argv[]) {
131     omp_set_num_threads(4);

```

```

131
132 #pragma omp parallel
133 {
134
135     #pragma omp single
136     {
137         printf("num threads: %d\n", omp_get_num_threads());
138     }
139
140     const int processor_count = 4;
141     int th_id=omp_get_thread_num();
142     DWORD_PTR mask = (1 << (th_id % processor_count));
143     DWORD_PTR result = SetThreadAffinityMask(thread_handle, mask);
144     if (result==0) {
145         printf("error SetThreadAffnityMask\n");
146     }
147     else {
148         printf("previous mask for thread %d : %d\n",th_id,result);
149         printf("new mask for thread %d : %d\n",
150             th_id,SetThreadAffinityMask(thread_handle, mask));
151     }
152 }
153
154 init_tab();
155
156 run_function(sum_ij, "sum_ij");
157 run_function(sum_ji, "sum_ji");
158 run_function(sum_sec, "sum_sec");
159 run_function(sum_pf, "sum_pf");
160 run_function(sum_par_ij, "sum_par_ij");
161 run_function(sum_par_ji, "sum_par_ji");
162
163 system("pause");
164 }

```

Listing 10: Cały program.

## Spis rysunków

1	Wykres przydziału wątków do rdzeni. . . . .	8
2	Instrukcja przypisania do zmiennej <i>tmp</i> . . . . .	14

## Spis tablic

1	Porównanie szybkości wybranych algorytmów wobec <i>sum_ij</i> . . .	8
2	Porównanie szybkości wybranych algorytmów (o kolejności pętli <i>ij</i> ) wobec <i>sum_ij</i> . . . . .	9
3	Porównanie szybkości wybranych algorytmów (o kolejności pętli <i>ji</i> ) wobec <i>sum_ji</i> . . . . .	9
4	Wartości IPC i CPI dla poszczególnych algorytmów . . . . .	10
5	Badane dane . . . . .	10
6	Pomiary związane z odwołaniami do pamięci L1 . . . . .	11
7	Pomiary związane z odwołaniami do pamięci L2 . . . . .	11
8	Czas realizacji kodu dla poszczególnych funkcji w <i>ms.</i> . . . .	12
9	Porównanie wybranych algorytmów pod kątem sekwencyjności pamięci	13
10	Czasy algorytmów kluczowych dla wyprzedzającego pobrania . .	14

## Listingi

1	Definicja tabeli. . . . .	4
2	Algorytm sekwencyjny – kolejność ij. . . . .	4
3	Algorytm sekwencyjny – kolejność ji . . . . .	5
4	Algorytm zrównoleglony – kolejność ij. . . . .	5
5	Algorytm zrównoleglony – kolejność ji. . . . .	5
6	Algorytm na sekwencyjność pamięci. . . . .	6
7	Algorytm na pobranie z wyprzedzeniem. . . . .	6
8	Ustawienie maski powinowactwa. . . . .	7
9	Definicja <i>tmp</i> z użyciem <i>volatile</i> . . . . .	14
10	Cały program. . . . .	15