

POLITECHNIKA POZNAŃSKA

WYDZIAŁ INFORMATYKI

PRZETWARZANIE RÓWNOLEGŁE

Równoległe sumowanie komórek pamięci za pomocą wielu wątków przetwarzania

Autorzy:

Adam SZCZEPAŃSKI
Mateusz CZAJKA

Prowadzący:

dr Rafał WALKOWIAK



13 lutego 2014

Spis treści

1	Informacje o projekcie	3
1.1	Dane autorów	3
1.2	Historia projektu	3
2	Wstęp	4
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 jj	5
2.3.4	Algorytm na sekwencyjność pamięci	6
2.3.5	Algorytm na pobranie z wyprzedzeniem	6
2.4	Maska powinowactwa	6
2.4.1	Kod	7
2.4.2	Rezultat	7
3	Pomiary efektywności	8
3.1	Przyśpieszenie 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
4	Wpływ rozmiaru danych	11
4.1	Wstęp	11
4.2	Wyniki	12
4.3	Podsumowanie	12
5	Wpływ sekwencyjności pamięci	12
5.1	Wstęp	12
5.2	Wyniki	13
6	Wyprzedzające pobranie danych	13
6.1	Klasyczne podejście	13
6.2	Zastosowanie kwalifikatora <i>volatile</i>	14
7	Załączniki	14
8	Kod źródłowy	14
	Spis rysunków	14

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 20 stycznia 2013.

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. Kolejno badaliśmy

- rozmiar danych
- sekwencyjność pamięci
- wyprzedzające pobranie

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 maksymalnego rozmiaru tablicy tj $[2^{28} - \text{wierszy}]$ na $[2^4 - \text{kolumn}]$.

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;
}
```

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;
}
```

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;
}
```

2.3.3 Algorytm zrównoleglony - kolejność jj

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;
}
```

```
}
```

2.3.4 Algorytm na sekwencyjność pamięci

Algorytm sekwencyjny w którym badaliśmy wpływ sekwencyjności 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;
}
```

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;
}
```

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();

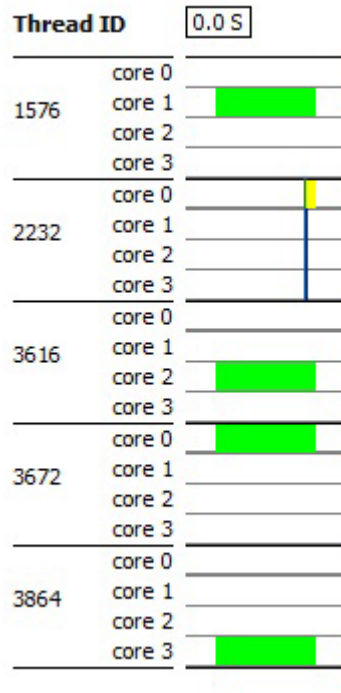
    DWORD_PTR mask = (1 << (th_id % processor_count));
    DWORD_PTR result = SetThreadAffinityMask(thread_handle, mask);

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

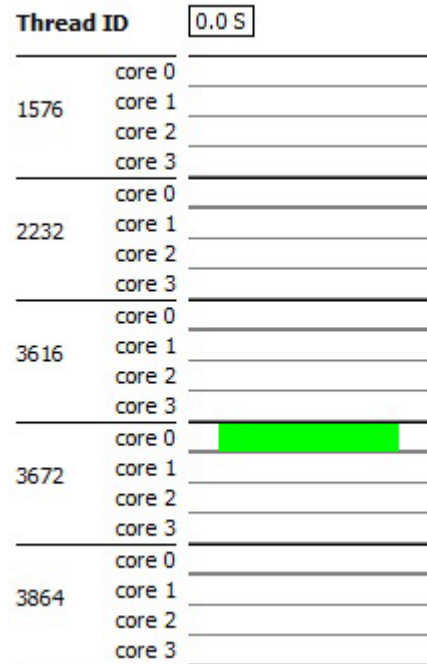
    else {
        printf("previous mask for thread %d : %d\n",th_id,result);
        printf("new mask for thread %d : %d\n",
            th_id,SetThreadAffinityMask(thread_handle, mask));
    }
}
```

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 wątków dla sum_par_ij



Rysunek 2: Wykres wątków dla sum_ij

3 Pomiary efektywności

3.1 Przyspieszenie obliczeń równoległych

3.1.1 Pomiary

Tablica 1: Porównanie szybkości wybranych algorytmów wobec sum_ij

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

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:

Tablica 2: Porównanie szybkości wybranych algorytmów (o kolejności pętli ij) wobec sum_{ij}

Algorytm	Czas wykonania w ms	Przyspieszenie względem sum_{ij}
sum_{ij}	2362	1.000
$sum_{par_{ij}}$	87	2.908

Tablica 3: Porównanie szybkości wybranych algorytmów (o kolejności pętli ji) wobec sum_{ji}

Algorytm	Czas wykonania w ms	Przyspieszenie względem sum_{ji}
sum_{ji}	2362	1.000
$sum_{par_{ji}}$	1189	1.987

3.1.2 Podsumowanie

Zrównoleglenie przetwarzania znacząco przyspiesza jego czas przetwarzania. Znaczony wpływ na wielkość wartości tego 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 CPI:

$$CPI = \frac{CPU_clocks}{ret_instr}. \quad (2)$$

Tablica 4: Wartości IPC i CPI dla poszczególnych algorytmów

Algorytm	ret_instr	CPU_clocks	IPC	CPI
sum_{ij}	12640	14736	0.86	1.17
sum_{ji}	9780	136048	0.07	13.91
$sum_{par_{ij}}$	12684	20096	0.63	1.58
$sum_{par_{ji}}$	11524	259628	0.04	22.53
sum_{sec}	14420	1139472	0.01	79.02
sum_{pf}	14996	13140	1.14	0.88

Zgodnie z oczekiwaniami najwyższe wartości wskaźnika IPC (a zarazem najniższe CPI) wysetpują 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

Tablica 5: Badane dane

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

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:

$$DC_miss_rate = \frac{DC_missess}{ret_instr} \quad (3)$$

$$DC_miss_ratio = \frac{DC_missess}{DC_accesss} \quad (4)$$

Tablica 6: Pomiary związane z odwołaniami do pamięci L1

Algorytm	DC accessess	DC missess	DC miss ratio	DC miss rate
sum_{ij}	5792	24	0%	0%
sum_{ji}	7620	4068	53%	42%
$sum_{par_{ij}}$	5188	20	0%	0%
$sum_{par_{ji}}$	7652	4456	58%	38%
sum_{sec}	18808	5256	28%	36%
sum_{pf}	9928	24	0%	0%

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:

$$L2_miss_rate = \frac{L2_misses}{ret_instr} \quad (5)$$

$$L2_miss_ratio = \frac{L2_misses}{L2_requests + L2_fill_write} \quad (6)$$

Tablica 7: Pomiary związane z odwołaniami do pamięci L2

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

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: $tab[2^{28}][2^4]$
- Instancja B: $tab[2^{24}][2^4]$

Spodziewaliśmy się, że czas obliczeń dla Instancji B będzie kilkukrotnie krótszy.

4.2 Wyniki

Wyniki pomiarów prezentują się następująco:

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

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	59	4890	57	23	304
Stosunek A/B	4.081	4.003	3.998	3.947	3.783	3.911

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 sekcyjności pamięci

5.1 Wstęp

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

- *sum_ij* - podstawowy algorytm sekwencyjny, służący jako punkt odniesienia
- *sum_sec* - algorytm sekwencyjny w którym badaliśmy sekcyjność pamięci

W wykorzystywanym systemie obliczeniowym liczba linii w pojedynczym bloku pamięci L1 wynosi 64.

W celu sprawdzenia w jaki sposób sekcyjność wpływa na czas realizacji zmodyfikowaliśmy kod w następujący sposób:

```
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];  
        }  
    }  
}
```

Sumujemy kolumnami, wprowadzając dodatkową, środkową pętlę która sumuje co 64 element. Dzięki temu każdy sumowany element musi być pobrany z pamięci dynamicznej.

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.

Tablica 9: Porównanie wybranych algorytmów pod kątem sekwencyjności pamięci

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

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 3: Instrukcja przypisania do zmiennej *tmp*

Tablica 10: Czasy algorytmów kluczowych dla wyprzedzającego pobrania

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

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 Załączniki

1. sum.cpp – plik z kodem źródłowym

8 Kod źródłowy

```
#include <stdio>
#include <cstdlib>
#include <ctime>
#include <Windows.h>
#include <omp.h>

#define CACHE_LINE 64
#define CACHE_ALIGN __declspec(align(CACHE_LINE))
#define CACHE_LINES_ON_PAGE 512

// ROWS * COLS = 1<<28 = 256 MB (2^8 * 2^10 * 2^10)
#define ROWS ((size_t) 1<<24)
#define COLS ((size_t) 1<<4)

CACHE_ALIGN int tab[ROWS][COLS];
double start;
```

```

HANDLE thread_handle = GetCurrentThread();

inline void start_clock() {
    start = (double) clock() / CLK_TCK;
}

inline void print_elapsed_time(const char* operation_name) {
    double elapsed;
    double resolution;

    elapsed = (double) clock() / CLK_TCK;
    resolution = 1.0 / CLK_TCK;

    printf("%s: %8.4f sec\n", operation_name, elapsed-start);
}

void init_tab() {

    start_clock();
    int i;
#pragma omp parallel for shared(tab) private(i)
    for (i=0; i<ROWS; i++) {
        for (int j=0; j<COLS; j++) {
            tab[i][j] = rand();
        }
    }
    print_elapsed_time("init");
}

typedef int (*function)();
inline void run_function(function f, const char* operation_name) {
    printf("-----\n");
    start_clock();
    int sum = f();
    printf("sum: %d\n", sum);
    print_elapsed_time(operation_name);
}

__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;
}

```



```

}

__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;
}

__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;
}

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;
}

__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;
}

__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;
}

int main(int argc, char* argv[]) {

    omp_set_num_threads(4);

#pragma omp parallel
    {

        #pragma omp single
        {
            printf("num threads: %d\n", omp_get_num_threads());
        }

        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));
        }
    }
}

```

```

init_tab();

run_function(sum_ij, "sum_ij");
run_function(sum_ji, "sum_ji");
run_function(sum_sec, "sum_sec");
run_function(sum_pf, "sum_pf");
run_function(sum_par_ij, "sum_par_ij");
run_function(sum_par_ji, "sum_par_ji");

system("pause");
}

```

Spis rysunków

1	Wykres wątków dla <i>sum_par_ij</i>	8
2	Wykres wątków dla <i>sum_ij</i>	8
3	Instrukcja przypisania do zmiennej <i>tmp</i>	13

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	9
5	Badane dane	10
6	Pomiary związane z odwołaniami do pamięci L1	10
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