Politechnika Poznańska

Wydział Informatyki

Przetwarzanie Równoległe

Równoległe sumowanie komórek pamięci za pomocą wielu wątków przetwarzania

Autorzy:
Adam Szczepański
Mateusz Czajka

Prowadzący:
dr Rafał Walkowiak

24 lutego 2014

Spis treści

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 25 lutego 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 zrównleglone. Celem zastosowania czterech różnych algorytmów sekwencyjnych było zbadanie wpływu sekcyjności pamięci podręcznej, wyprzdzającego pobrania danych do pamięcy 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
- sekcyjności 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 ($sizeof(int) = 4$), o rozmiarach 1GB ($[2^{24} - wierszy]$ na $[2^4 - kolumn]$) i 256MB ($[2^{22} - wierszy]$ na $[2^4 - kolumn]$). W badanym systemie długość lini pamięci podręcznej wynosiła 64B, co odpowiada szerokości naszej tabeli. Aby zapewnić wyrównanie do początku lini PP tabela została zdefiniowana z wykorzystaniem rozszerzenia Visual Studio:

```
__declspec(align(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). Nazwywany 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 sekcyjność pamięci

Algorytm sekwencyjny w którym badaliśmy wpływ sekcyjności 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 sekcyjność 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.

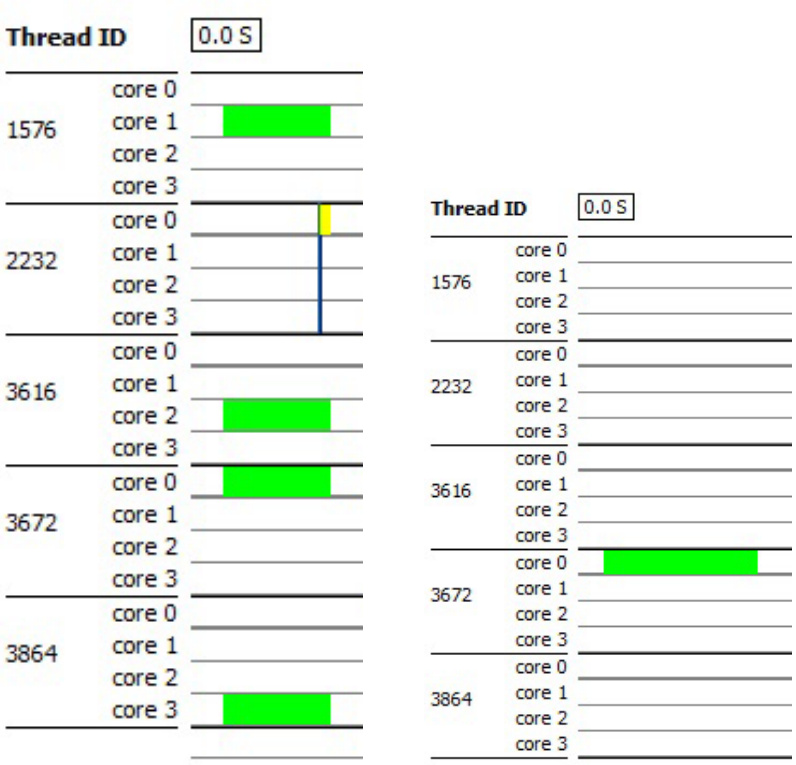
2.4.1 Kod

```
#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:



(a) sumowanie równoległe

(b) sumowanie sekwencyjne

Rysunek 1: Wykres przydziału wątków do rdzeni.

3 Pomiary efektywności

3.1 Przyspieszenie obliczeń równoległych

3.1.1 Pomiary

Algorytm	Czas wykonania w ms	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

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 uszeregowań pętli prezentują się następująco:

Algorytm	Czas wykonania w ms	Przyspieszenie względem sum_ij
sum_ij	2362	1.000
sum_par_ij	87	2.908

Tablica 2: Porównanie szybkości wybranych algorytmów (o kolejności pętli ij) wobec sum_ij

Algorytm	Czas wykonania w ms	Przyśpieszenie względem sum_ji
sum_ji	2362	1.000
sum_par_ji	1189	1.987

Tablica 3: Porównanie szybkości wybranych algorytmów (o kolejności pętli ji) wobec sum_ji

3.1.2 Podsumowanie

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

- trzykrotne przyśpieszenie dla uszeregowania pętli ij
- dwukrotne przyśpieszenie dla uszeregowania pętli ji

3.2 Ilość wykonanych instrukcji na jeden cykl procesora

IPC (insructions 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:

$$IPC = \frac{ret\_instr}{CPU\_clocks}. \tag{1}$$

CPI (cycles per instruction) jest odwrotnością IPC:

$$CPI = \frac{CPU\_clocks}{ret\_instr}. \tag{2}$$

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

Tablica 4: Wartości IPC i CPI dla poszczególnych algorytmów

Zgodnie z oczekiwaniami najwyższe wartości wzkaźnika IPC (a zarazem najniższe CPI) wysętpują 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 kilkudziesiącio krotonie większe niż dla algorytmów z pozostałej grupy.

3.3 Współczynniki braku trafień do pamięci

3.3.1 Znaczenie danych

Nazwa	Znaczenie
DC acceesses	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:

$$DC\_miss\_rate = \frac{DC\_missess}{ret\_instr} \tag{3}$$

$$DC\_miss\_ratio = \frac{DC\_missess}{DC\_accesss} \tag{4}$$

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%

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:

$$L2\_miss\_rate = \frac{L2\_missess}{ret\_instr} \tag{5}$$

$$L2\_miss\_ratio = \frac{L2\_missess}{L2\_requests + L2\_fill\_write} \tag{6}$$

Algorytm	L2 requests	L2 missess	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%

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	sum_ij	sum_ji	sum_sec	sum_pf	sum_par_ij	sum_par_ji
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 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 odeniesienia
- *sum_sec* - algorytm sekwencyjny w którym badaliśmy sekcyjność pamięci

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

W metodzie *sum_sec* w kolejnych iteracjach odwołujemy się do komórki tabeli z tej samej kolumny, z wiersza przesuniętego o ilość lini w stronie. Powoduje to konieczność wymiany całej lini 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
sum_ij	253	24	340	352
sum_sec	19552	5256	11024	5920

Tablica 9: Porównanie wybranych algorytmów pod kątem sekcyjności pamięci

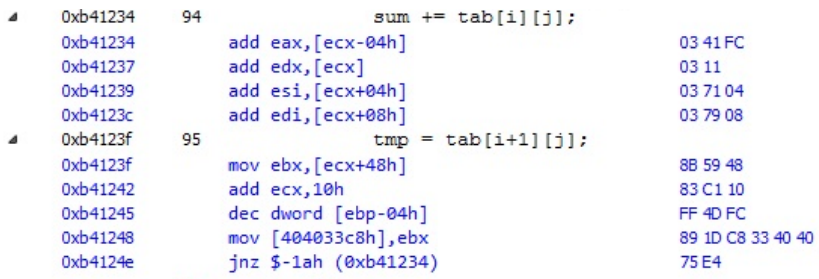
Wyniki przerosły nasze oczekiwania - *sum_sec* wykonał się ponad 77 razy wolniej. Pokazuje to jest wielkie znaczenie na szybkość realizacji przetwarzania ma zachowanie cechy lokalności przestrzennej. Wyraźnie widać, że w przypadku algorytmu użytego jako punkt odniesienia liczba braków trafień do pamięci 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 optymlizacją 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.



Rysunek 2: Instrukcja przypisania do zmiennej *tmp*

Algorytm	Czas [ms]
sum_ij	253
sum_pf	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, ze 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 równoległwego 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 dostępów do pamięci, zapewnienie własności lokalności czasowej i przestrzennej w znaczący sposób przyspiesza działanie progoramu (*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 równolegle czy sekwencyjnie.

8 Załączniki

1. sum.cpp – plik z kodem źródłowym

9 Kod źródłowy

```
#include <cstdio>
#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

#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() {
```

```
33
34    start_clock();
35    int i;
36  #pragma omp parallel for shared(tab) private(i)
37    for (i=0; i<ROWS; i++) {
38      for (int j=0; j<COLS; j++) {
39        tab[i][j] = rand();
40      }
41    }
42    print_elapsed_time("init");
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)
      reduction(+:sum)
108   for (i=0; i<ROWS; i++) {
109     for (int j=0; j<COLS; j++) {
110       sum += tab[i][j];
111     }
112   }
113   return sum;
114 }
115
116 __declspec(noinline) int sum_par_ji() {
117   int sum = 0;
118   int j;
119 #pragma omp parallel for default(none) shared(tab) private(j)
      reduction(+:sum)
120   for (j=0; j<COLS; j++) {
121     for (int i=0; i<ROWS; i++) {
122       sum += tab[i][j];
123     }
124   }
125   return sum;
126 }
127
128 int main(int argc, char* argv[]) {
129
130   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

Spis tablic

Listingi