

POLITECHNIKA POZNAŃSKA

WYDZIAŁ INFORMATYKI

PRZETWARZANIE RÓWNOLEGŁE

Równoległe sumowanie komórek pamięci za pomocą wielu wątków przetwarzania

Autorzy:

Adam SZCZEPAŃSKI
Mateusz CZAJKA

Prowadzący:

dr Rafał WALKOWIAK



11 lutego 2014

Spis treści

1	Informacje o projekcie	2
1.1	Dane autorów	2
1.2	Historia projektu	2
2	Wstęp	3
2.1	Opis problemu	3
2.2	Punkt odniesienia (algorytm sekwencyjny - kolejność ij)	3
2.3	Badane algorytmy	4
2.3.1	Algorytm sekwencyjny - kolejność ji	4
2.3.2	Algorytm zrównoleglony - kolejność ij	4
2.3.3	Algorytm zrównoleglony - kolejność jj	4
2.3.4	Algorytm na sekwencyjność pamięci	5
2.3.5	Algorytm na pobranie z wyprzedzeniem	5
3	Wpływ rozmiaru danych	6
3.1	Wstęp	6
3.2	Wyniki	6
3.3	Podsumowanie	6
	Spis rysunków	6
	Spis tablic	6

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 sekcynność pamięci

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

3 Wpływ rozmiaru danych

3.1 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.

3.2 Wyniki

Wyniki pomiarów prezentują się następująco:

Tablica 1: 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

3.3 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.

Spis rysunków

Spis tablic

1	Czas realizacji kodu dla poszczególnych funkcji w <i>ms</i>	6
---	-----------------------------------------------------------------------	---