Mateusz Bednarski
177194
PR, niestacjonarne, grupa C.

Uwagi ogólne

Kod był uruchamiany w kompilacji Debug bez podłączonego debuggera (opcja Start without debugging). Środowisko VC++ 2015.

Wszystkie pomiary zostały powtórzone 3 razy i został wybrany najmniejszy czas. W czasie testów komputer nie był obciążony innymi procesami.

Testowy procesor:

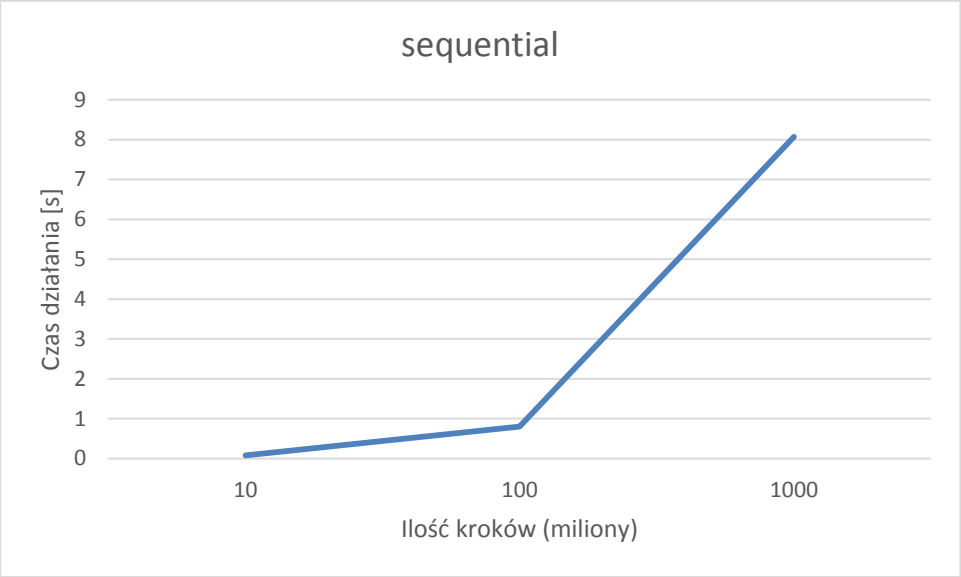
- Intel Core I5 2400
- Częstotliwość taktowania: 3,1 GHz
- Ilość rdzeni fizycznych: 4
- Ilość rdzeni logicznych: 4
- Hyper-Threading : brak

Punkt 1/2

Kod

```
void sequential()
{
    double step;
    clock_t start, stop;
    double x, pi, sum = 0.0;
    int i;
    step = 1. / (double)num_steps;
    start = clock();
    for (i = 0; i < num_steps; i++)
    {
        x = (i + .5)*step;
        sum = sum + 4.0 / (1. + x*x);
    }
    pi = sum*step;
    stop = clock();
    printf("%s\n", abs(pi - 3.1415) < 0.001 ? "OK" : "INCORRECT
RESULT");
    printf("time: %f\n", ((double)(stop - start) / 1000.0));
}
```

Czas działania

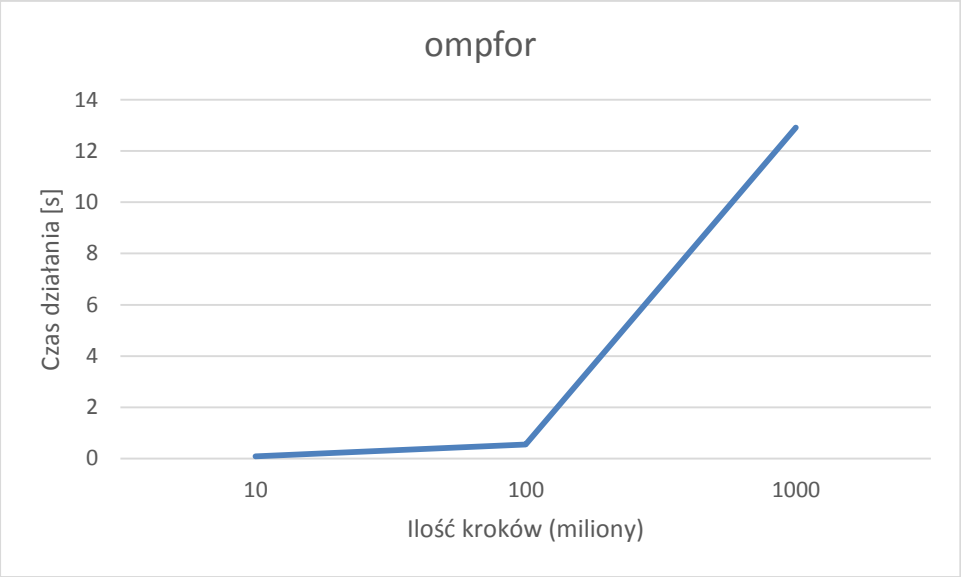


Punkt 3.

Kod:

```
void ompfor(void)
{
    double step;
    clock_t start, stop;
    double pi, sum = 0.0;
    int i;
    step = 1. / (double)num_steps;
    start = clock();
#pragma omp parallel for num_threads(4) schedule(static)
    for (i = 0; i < num_steps; i++)
    {
        double x = (i + .5)*step;
        sum = sum + 4.0 / (1. + x*x);
    }
    pi = sum*step;
    stop = clock();
    printf("%s\n", abs(pi - 3.1415) < 0.001 ? "OK" : "INCORRECT
RESULT");
    printf("time: %f\n", ((double)(stop - start) / 1000.0));
}
```

Czas działania



Zmienne i oraz x są prywatne, sum jest współdzielona między wątki. Podany kod daje błędne wyniki. Jest to faktem , że aktualizacja sumy (zmienna sum) nie jest dokonywana atomowo. Jedyną zmienną prywatną jest i – licznik pętli. Przyśpieszenie jest niewielkie, jest to spowodowane prawdopodobnie tym, że wszystkie wątki piszą sobie nawzajem po tej samej zmiennej ciągle unieważniając linię pamięci dla innych, okazuje się że dla 100 milionów kroków jest nawet wolniej niż sekwencyjnie.

Punkt 4.

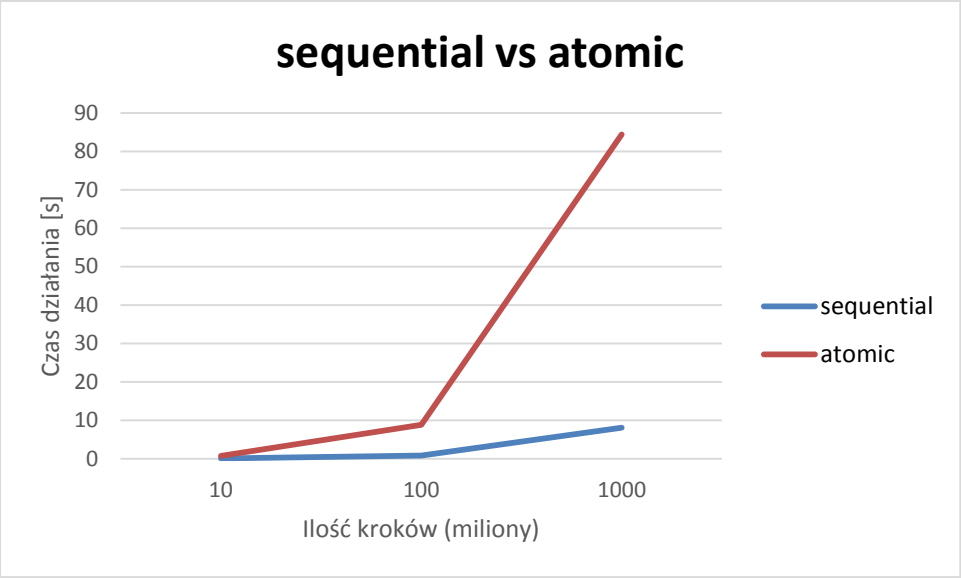
Kod

```
void atomic(void)
{
    double step;
    clock_t start, stop;
    double pi, sum = 0.0;
    int i;
    step = 1. / (double)num_steps;
    start = clock();
#pragma omp parallel for num_threads(4) schedule(static)
    for (i = 0; i < num_steps; i++)
    {
        double  x = (i + .5)*step;
#pragma omp atomic
        sum += 4.0 / (1. + x*x);
    }

    pi = sum*step;
    stop = clock();
```

```
    printf("%s\n", abs(pi - 3.1415) < 0.001 ? "OK" : "INCORRECT
RESULT");
    //printf("val:  %15.12f\n", pi);
    printf("time: %f\n", ((double)(stop - start) / 1000.0));
}
```

Czas działania



Tym razem wynik jest poprawny ze względu na zapewnienie poprawności aktualizacji zmiennej sum, natomiast odbywa się to bardzo dużym kosztem wydajnościowym. Za każdym razem gdy jeden z wątków aktualizuje sumę, pozostałe muszą zaczekać aż on przeprowadzi tą operację – operacje synchronizujące zajmują niewspółmiernie dużo czasu w stosunku do tego który jest potrzebny do przeprowadzenia właściwych obliczeń;

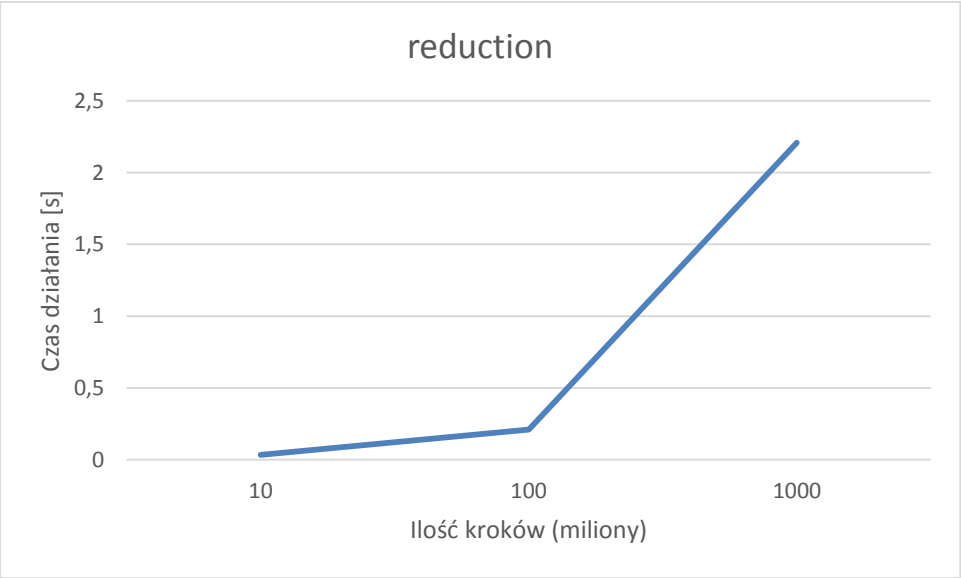
Punkt 5.

Kod:

```
void reduction(void)
{
    double step;
    clock_t start, stop;
    double pi, sum = 0.0;
    int i;
    step = 1. / (double)num_steps;
    start = clock();
#pragma omp parallel for reduction(+:sum) num_threads(4)
schedule(static)
    for (i = 0; i < num_steps; i++)
    {
        double x = (i + .5)*step;
        sum = sum + 4.0 / (1. + x*x);
```

```
    }
    pi = sum*step;
    stop = clock();
    printf("%s\n", abs(pi - 3.1415) < 0.001 ? "OK" : "INCORRECT
RESULT");
    printf("time: %f\n", ((double)(stop - start) / 1000.0));
}
```

Czas działania



Użycie dyrektywy reduction daje najlepsze wyniki. Jej sens można opisowo przedstawić następująco: każdy wątek dostaje własną, prywatną zmienną sum i aktualizuje ją w trakcie pracy. Jako, że każdy ma własną, wątki nie nadpisują jej sobie nawzajem. Dopiero na końcu wszystkie lokalne sumy są sumowane aby otrzymać ostateczny wynik. Dzięki temu raz unikamy wzajemnego unieważniania pamięci, dwa unikamy kosztu synchronizacji sumy za każdym krokiem (jest to odroczone do momentu zakończenia pracy – do synchronizacji pozostają 4 sumy lokalne zamiast wcześniejszego synchronizowania miliony razy. W najlepszym przypadku uzyskano przyśpieszenie na poziomie 3,88 co jest bliskie teoretycznie optymalnemu wynikowi 4.

Punkt 6

Kod

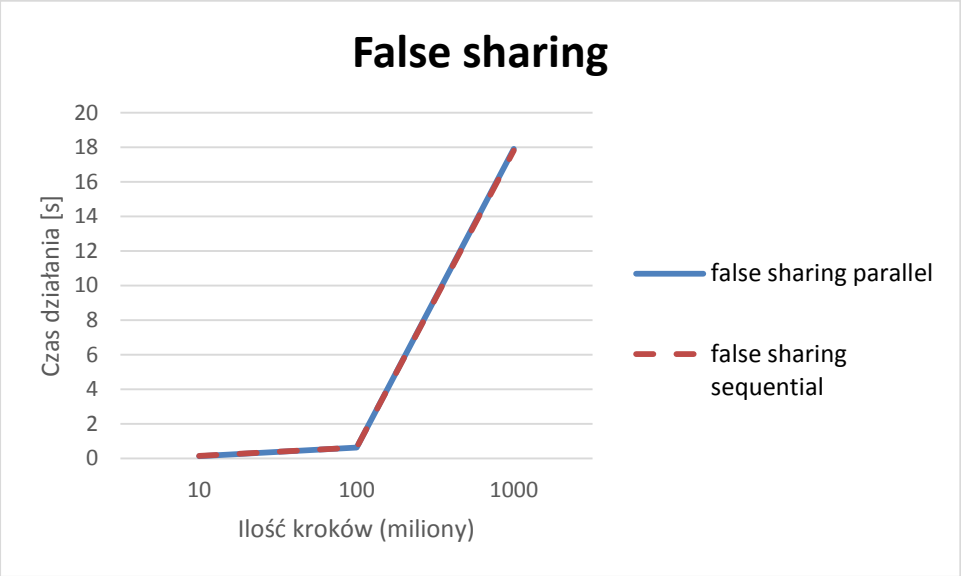
```
void false_sharing_parallel(void)
{
    double step;
    volatile double tab[4] = { 0 };
    clock_t start, stop;
    double pi, sum = 0.0;
    int i;
    step = 1. / (double)num_steps;
    start = clock();
#pragma omp parallel num_threads(4) // 1 dla wersji sekwencyjnej
```

```
    {
        int id = omp_get_thread_num();
#pragma omp for reduction(+:sum) schedule(static)
        for (i = 0; i < num_steps; i++)
        {
            double x = (i + .5)*step;
            tab[id] += 4.0 / (1. + x*x);
        }
#pragma omp atomic
        sum += tab[id];
    }

    pi = sum*step;
    stop = clock();

    printf("%s\n", abs(pi - 3.1415) < 0.001 ? "OK" : "INCORRECT
RESULT");
    printf("time: %f\n", ((double)(stop - start) / 1000.0));
}
```

Czas działania



W poprzednio przeprowadzonych testach lokalne zmienne x dla każdego z wątków mogły być potencjalnie w rożnych miejscach pamięci. Używając czteroelementowej tablicy (ponieważ używamy 4 wątków) gwarantujemy że będą one znajdowały się obok siebie. Mamy nadzieję, że w ten sposób wszystkie z nich znajdą się w jednej linii pamięci, przez co za każdy, razem, każdy z wątków będzie musiał ją pobrać na nowo. Tak jest w istocie, liczby pokazują że podany kod znacząco zmniejsza korzyści z używania reduction. Co ciekawe różnice miedzy 4 a jednym wątkiem są marginalne. Wnioskuje na tej podstawie, że dominuje koszt odwołań do pamięci.

Punkt 7

Kod

```
void memory_line(void)
{
    float step;
    typedef float float_t;
    volatile float_t tablica[10000] = { 0 };
    for (int q = 0; q < 10000; q++)
    {
        ZeroMemory((char *)tablica, 10000 * sizeof(float_t));
        clock_t start, stop;
        float_t x, pi, sum = 0.0;
        int i;
        step = 1. / (float_t)num_steps;
        start = clock();
#pragma omp parallel num_threads(2)
        {
            int id = omp_get_thread_num();
#pragma omp for schedule(static)
            for (i = 0; i < num_steps; i++)
            {
                float_t x = (i + .5)*step;
                tablica[q + id] += 4.0 / (1. + x*x);
            }
#pragma omp atomic
            sum += tablica[q + id];
        }

        pi = sum*step;
        stop = clock();

        printf("%s\n", abs(pi - M_PI) < 0.001 ? "OK" : "INCORRECT
RESULT");
        printf("%d time: %f\n", q, ((double)(stop - start) /
1000.0));
    }
    exit(0);
}
```

Niewielka modyfikacja poprzedniego kodu pozwala wyznaczyć długość linii pamięci podręcznej. Alokujemy dość dużą tablicę w której będą zapisywane lokalne sumy wątków. Przesuwamy się po niej, taka by dwa wątki pracowały na jej sąsiednich elementach, kolejno 0 i 1, 1 i 2, 2 i 3, 3 i 4… Jako, że tablica jest ciągłym obszarem pamięci osiągamy efekt „sunięcia” przez kolejne adresy. Uruchomienie kodu pokazuje, że co którąś iterację kod wykonuje się znacznie szybciej. Dzieje się tak kiedy jeden wątek pracuje w jednej, a drugi w drugiej linii pamięci. Obliczając częstotliwość

pojawiania się przyśpieszeń wyznaczamy długość linii pamięci. Dla typu double jest to co ósma iteracja. Stąd długość linii = 8 * sizeof(double) = 8 * 8 = 64B. Zmiana typu na float potwierdza: jest to co 16 iteracja: 16 * sizeof(float) = 16 * 4 = 64B. Co ciekawe zaobserwowano, że typ float posiada zbyt małą precyzje na poprawne wyznaczeni liczby PI przy większej ilości iteracji. Dzieje się tak ponieważ w linii step = 1. / (float_t)num_steps; dla dużych num_steps, step jest bardzo małe (zbyt małe aby float mógł przechować z wystarczającą precyzją).

Punkt 8

Kod

```
void affinity_1_to_1(void)

{
    double step;
    clock_t start, stop;
    double x, pi, sum = 0.0;
    int i;
    step = 1. / (double)num_steps;
    start = clock();
#pragma omp parallel num_threads(4)
    {
        int id = omp_get_thread_num();
        DWORD_PTR mask = (1 << id);
        // DWORD_PTR mask = (1 << id / 2); dla powinowactwa parami
        SetThreadAffinityMask(GetCurrentThread(), mask);

#pragma omp for reduction(+:sum) schedule(static)
        for (i = 0; i < num_steps; i++)
        {
            x = (i + .5)*step;
            sum = sum + 4.0 / (1. + x*x);
        }
    }
    pi = sum*step;
    stop = clock();

    printf("%s\n", abs(pi - M_PI) < 0.001 ? "OK" : "INCORRECT
RESULT");

    printf("time: %f\n", ((double)(stop - start) / 1000.0));
}
```

Sprawdzony został również wpływ powinowactwa wątków na czasy wykonania. Wyniki zostały zaprezentowane w tabeli:

Powinowactwo	Czas dla 1000 milionów kroków [s]
1 do 1	5,812
2 do 1	6,946
automatyczne	6,914

Tabela pomiarowa 1

Nie jest zaskoczeniem, że 1 do 1 zajęło mniej czasu niż 2 do 1. Autor natomiast jest zaskoczony, że przydział automatyczny nie jest tożsamy z 1 do 1. Prawdopodobnie planista systemowy oszacował, że różnica nie jest wystarczająco duża aby zajmować wszystkie rdzenie.

Oryginalne wyniki

Num steps	10	100	1000
sequential	0,082	0,805	8,07
ompfor	0,092	0,552	12,919
atomic	0,74	8,813	84,389
reduction	0,033	0,21	2,208
false sharing parallel	0,136	0,622	17,911
false sharing sequential	0,139	0,621	17,804

Tabela pomiarowa 2

Przyśpieszenie

ompfor	0,891	1,458	0,625
atomic	0,111	0,091	0,096
reduction	2,485	3,833	3,655
false sharing parallel	0,603	1,294	0,451
false sharing sequential	0,590	1,296	0,453

Tabela pomiarowa 3