Programowanie równoległe
Laboratorium 6

Sprawozdanie

Zadanie 2:

Przetestowanie opcji współdzielenia zmiennych dla bloku równoległego.

Kod źródłowy rozwiązania:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int i, a;
    a = 7;

    #pragma omp parallel for private(a) num_threads(7)
    // #pragma omp parallel for firstprivate(a) num_threads(7)
    // #pragma omp parallel for shared(a) num_threads(7)
    for(i = 0; i < 10; i++) {
       printf("Thread %d a=%d\n", omp_get_thread_num(), a);
       a++;
    }

    printf("Final a=%d\n", a);

    return 0;
}
```

Wydruk dla opcji private:

```
Thread 4 a=0
Thread 0 a=0
Thread 0 a=1
Thread 2 a=0
Thread 2 a=1
Thread 5 a=0
Thread 3 a=0
Thread 6 a=0
Thread 1 a=0
Thread 1 a=1
Final a=7
```

Każdy wątek tworzy nową zmienną odpowiedniego typu o zerowej wartości.

Wydruk dla opcji firstprivate:

```
Thread 0 a=7
Thread 0 a=8
Thread 4 a=7
Thread 2 a=7
Thread 2 a=8
Thread 5 a=7
Thread 6 a=7
Thread 1 a=7
Thread 1 a=8
Thread 3 a=7
Final a=7
```

Każdy wątek kopiuje zmienną i działa na kopii lokalnej.

Wydruk dla opcji share:

```
Thread 0 a=7
Thread 0 a=8
Thread 6 a=7
Thread 1 a=7
Thread 1 a=11
Thread 2 a=7
Thread 2 a=13
Thread 3 a=7
Thread 5 a=7
Thread 4 a=7
Final a=17
```

Każdy wątek działa na tej samej zmiennej.

Zadanie 3:

Napisanie procedury, w której znajdą się cztery pętle o 15 iteracjach testujące różne strategie przydziału iteracji czterem wątkom.

Kod źródłowy rozwiązania:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int i, a;
    a = 7;

    #pragma omp parallel for firstprivate(a) num_threads(4)
schedule(static, 3)
    for(i = 0; i < 15; i++) {
       printf("Thread %d a=%d\n", omp_get_thread_num(), a);
       a++;
    }
```

```
    printf("\n");

    #pragma omp parallel for firstprivate(a) num_threads(4)
schedule(static)
    for(i = 0; i < 15; i++) {
       printf("Thread %d a=%d\n", omp_get_thread_num(), a);
       a++;
    }

    printf("\n");

    #pragma omp parallel for firstprivate(a) num_threads(4)
schedule(dynamic, 3)
    for(i = 0; i < 15; i++) {
       printf("Thread %d a=%d\n", omp_get_thread_num(), a);
       a++;
    }

    printf("\n");

    #pragma omp parallel for firstprivate(a) num_threads(4)
schedule(dynamic)
    for(i = 0; i < 15; i++) {
       printf("Thread %d a=%d\n", omp_get_thread_num(), a);
       a++;
    }

    printf("\n");

    return 0;
}
```

Wydruk dla strategii static, rozmiar porcji=3:

```
Thread 0 a=7
Thread 0 a=8
Thread 0 a=9
Thread 0 a=10
Thread 1 a=7
Thread 1 a=8
Thread 1 a=9
Thread 0 a=11
Thread 2 a=7
Thread 2 a=8
Thread 2 a=9
Thread 0 a=12
Thread 3 a=7
Thread 3 a=8
Thread 3 a=9
```

Iteracje rozdzielane są na zbiory o rozmiarze 3 i kolejno przydzielane dostępnym wątkom.

Wydruk dla strategii static, rozmiar porcji domyślny:

```
Thread 3 a=7
Thread 3 a=8
Thread 2 a=7
Thread 2 a=8
Thread 2 a=9
Thread 2 a=10
Thread 1 a=7
Thread 1 a=8
Thread 1 a=9
Thread 1 a=10
Thread 0 a=7
Thread 0 a=8
Thread 0 a=9
Thread 0 a=10
Thread 3 a=9
```

Iteracje rozdzielane są na zbiory o rozmiarze 15 / 4 = 4 i kolejno przydzielane dostępnym wątkom.

Wydruk dla strategii dynamic rozmiar porcji=3:

```
Thread 0 a=7
Thread 0 a=8
Thread 0 a=9
Thread 0 a=10
Thread 0 a=11
Thread 0 a=12
Thread 2 a=7
Thread 2 a=8
Thread 2 a=9
Thread 1 a=7
Thread 1 a=8
Thread 1 a=9
Thread 3 a=7
Thread 3 a=8
Thread 3 a=9
```

Każdemu wątkowi przypisywana jest liczba iteracji 3, po wykonaniu obliczeń wątki otrzymują kolejną porcję iteracji do wykonania.

Wydruk dla strategii dynamic, rozmiar porcji domyślny:

```
Thread 3 a=7
Thread 3 a=8
Thread 3 a=9
Thread 3 a=10
Thread 3 a=11
Thread 3 a=12
```

```
Thread 0 a=7
Thread 0 a=8
Thread 0 a=9
Thread 0 a=10
Thread 0 a=11
Thread 0 a=12
Thread 3 a=13
Thread 2 a=7
Thread 1 a=7
```

Każdemu wątkowi przypisywana jest liczba iteracji 1, po wykonaniu obliczeń wątki otrzymują kolejną porcję iteracji do wykonania.

Zadanie 4:

Dodanie procedur pomiaru czasu (z biblioteki OpenMP) i zmierzenie czasów dla różnych strategii podziału pętli (przy zwiększeniu ilości iteracji i wyłączeniu wyświetlania). Porównanie i analiza otrzymanych wyników.

Kod źródłowy rozwiązania:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    double start, end;
    int i, a;
    a = 7;

    start = omp_get_wtime();

    #pragma omp parallel for firstprivate(a) num_threads(4)
schedule(static, 3)
    for(i = 0; i < 150; i++) {
       a++;
    }

    end = omp_get_wtime();
    printf("Static 3:\t%f\n", end - start);

    start = omp_get_wtime();

    #pragma omp parallel for firstprivate(a) num_threads(4)
schedule(static)
    for(i = 0; i < 150; i++) {
       a++;
    }

    end = omp_get_wtime();
```

```
    printf("Static auto:\t%f\n", end - start);

    start = omp_get_wtime();

    #pragma omp parallel for firstprivate(a) num_threads(4)
schedule(dynamic, 3)
    for(i = 0; i < 150; i++) {
       a++;
    }

    end = omp_get_wtime();
    printf("Dynamic 3:\t%f\n", end - start);

    start = omp_get_wtime();

    #pragma omp parallel for firstprivate(a) num_threads(4)
schedule(dynamic)
    for(i = 0; i < 150; i++) {
       a++;
    }

    end = omp_get_wtime();
    printf("Dynamic auto:\t%f\n", end - start);

    return 0;
}
```

Wydruk:

```
Static 3:    0.000087
Static auto:    0.000002
Dynamic 3:    0.000004
Dynamic auto:    0.000006
```

Parametry "static, 3" są najmniej wydajne; różnica pomiędzy innymi parametrami nie jest taka zauważalna.

Zadanie 5:

Opracowanie programu równoległego liczącego sumę pięciuset kwadratów dowolnej liczby.

Kod źródłowy rozwiązania:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int i, a, sum;
    a = 7;
    sum = 0;
```

```
    #pragma omp parallel for schedule(static) num_threads(4) shared(a,
sum)
    for(i = 0; i < 500; i++) {
       #pragma omp critical
       {
             sum += a * a;
       }
    }

    printf("Sum is %d\n", sum);

    return 0;
}
```

Wydruk:

```
Sum is 24500
```

Zadanie 6:

Przetestowanie i pomiar czasu wykonania programu dla różnych strategii zabezpieczenia zmiennej wspólnej (reduction, critical, lock itp.). Porównanie i analiza otrzymanych wyników.

Kod źródłowy rozwiązania:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    double start, end;
    int i, a, sum;
    omp_lock_t writelock;
    omp_init_lock(&writelock);
    a = 7;
    sum = 0;

    start = omp_get_wtime();

    #pragma omp parallel for schedule(static) num_threads(4) shared(a)
reduction(+:sum)
    for(i = 0; i < 500; i++) {
       short square = a * a;
       sum += square;
    }

    end = omp_get_wtime();
    printf("Reduction:\t%f\n", end - start);
```

```
    sum = 0;

    start = omp_get_wtime();

    #pragma omp parallel for schedule(static) num_threads(4) shared(a,
sum)
    for(i = 0; i < 500; i++) {
       short square = a * a;

       #pragma omp critical
       {
             sum += square;
       }
    }

    end = omp_get_wtime();
    printf("Critical:\t%f\n", end - start);

    sum = 0;

    start = omp_get_wtime();

    #pragma omp parallel for schedule(static) num_threads(4) shared(a,
sum)
    for(i = 0; i < 500; i++) {
       short square = a * a;
       omp_set_lock(&writelock);
       sum += square;
       omp_unset_lock(&writelock);
    }

    end = omp_get_wtime();
    printf("Lock:\t\t%f\n", end - start);

    return 0;
}
```

Wydruk:

```
Reduction:    0.000094
Critical:    0.000039
Lock:        0.000053
```

Strategia critical jest najbardziej wydajna, strategia reduction jest najmniej wydajna.