
Laborator 2

Utilizarea variabilelor partajate într-un program OpenMP

Problemă de rezolvat

Să se calculeze numărul PI prin intermediul formulei:

$$\int_0^1 \frac{4.0}{(1+x^2)} = \pi$$

utilizând aproximarea:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Să se paralelizeze algoritmul secvențial implementat.

Algoritm secvențial

```
#include "stdafx.h"

int main()
{
    int i, num_steps = 2000000000;
    double step, x, pi, sum = 0.0;
    step = 1.0 / (double)num_steps;
    for (i = 0; i < num_steps; i++) {
        x = (i + 0.5)*step;
        sum = sum + 4.0 / (1.0 + x*x);
    }
    pi = step * sum;
    printf("pi=%f\n", pi);
    getchar();
    return 0;
}
```

A se observa încărcarea procesorului la execuția programului.


Algoritm paralel (tentativa 1)

Se va duplica codul algoritmului serial și se va introduce directiva (pentru a doua instanță a algoritmului):

```
#pragma omp parallel
```

Se va executa noul program. Încercați să explicați rezultatul secțiunii paralele...

```
pi=3.141593  
pi=3.135249
```



În investigarea problemei putem încerca să utilizăm următorul cod suplimentar (în cadrul algoritmului paralel):

```
num_steps = 5;  
...  
int ID = omp_get_thread_num();  
...  
printf("Thread %d calculez pentru i=%d\n", ID, i);
```

Se poate observa o primă problemă (pe lângă accentuarea erorii de calcul): pasul de calcul se execută de mai multe ori (nu de același număr de ori) pentru același index i .

```
pi=3.144926  
Thread 0 calculez pentru i=0  
Thread 1 calculez pentru i=0  
Thread 2 calculez pentru i=0  
Thread 3 calculez pentru i=0  
Thread 3 calculez pentru i=2  
Thread 3 calculez pentru i=3  
Thread 2 calculez pentru i=1  
Thread 3 calculez pentru i=4  
Thread 0 calculez pentru i=1  
Thread 1 calculez pentru i=1  
pi=6.989053
```

Rezultatul eronat al execuției este cauzat de partajarea incorectă a variabilei i . Există două soluții pentru această problemă: declararea locală (în secțiunea paralelă) a variabilei sau declararea *private* a acesteia.

Algoritm paralel (tentativa 2)

În cadrul algoritmului paralel se poate înlocui bucla for cu:

```
for (int j = 0; j < num_steps; j++) {  
    x = (j + 0.5)*step;  
    sum += 4.0 / (1.0 + x*x);  
    printf("Thread %d calculez pentru j=%d\n", ID, j);  
}
```

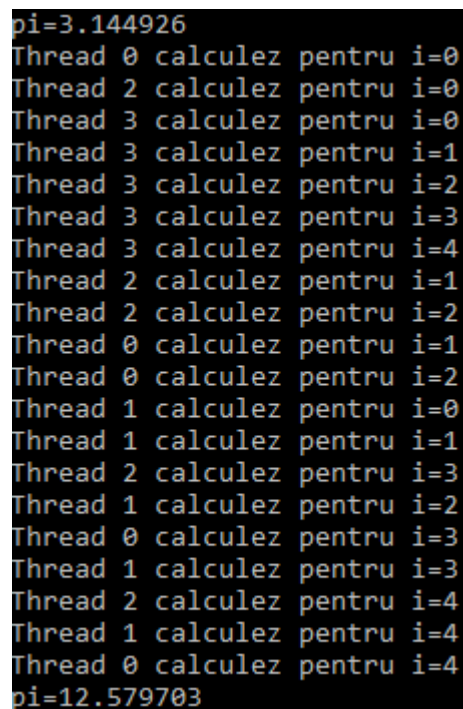
(soluția 1 – utilizarea unei variabile locale, declarată după directiva de paralelizare)

sau putem completa directiva de paralelizare:

```
#pragma omp parallel private(i)
```

(soluția 2 – impunem ca fiecare fir de execuție să utilizeze o copie proprie a variabilei i).

Din păcate rezultatul execuției este la fel de eronat:



```
pi=3.144926  
Thread 0 calculez pentru i=0  
Thread 2 calculez pentru i=0  
Thread 3 calculez pentru i=0  
Thread 3 calculez pentru i=1  
Thread 3 calculez pentru i=2  
Thread 3 calculez pentru i=3  
Thread 3 calculez pentru i=4  
Thread 2 calculez pentru i=1  
Thread 2 calculez pentru i=2  
Thread 0 calculez pentru i=1  
Thread 0 calculez pentru i=2  
Thread 1 calculez pentru i=0  
Thread 1 calculez pentru i=1  
Thread 2 calculez pentru i=3  
Thread 1 calculez pentru i=2  
Thread 0 calculez pentru i=3  
Thread 1 calculez pentru i=3  
Thread 2 calculez pentru i=4  
Thread 1 calculez pentru i=4  
Thread 0 calculez pentru i=4  
pi=12.579703
```

De această dată se observă că fiecare fir de execuție parcurge toți indecșii i. Trebuie să implementăm o soluție în care fiecare fir de execuție să execute doar o parte din plaja de valori a lui i.

Algoritm paralel (tentativa 3)

Vom efectua următoarea îmbunătățire:

```
#define N 4
#pragma omp parallel num_threads(N) private (i)
...
for (int i = ID; i < num_steps; i+=N) {
...

```

astfel încât fiecare din cele N fire de execuție impuse să calculeze doar valorile de rang ID. Executăm noua versiune a programului și putem crede că totul este rezolvat:

```
pi=3.144926
Thread 0 calculez pentru i=0
Thread 0 calculez pentru i=4
Thread 1 calculez pentru i=1
Thread 3 calculez pentru i=3
Thread 2 calculez pentru i=2
pi=3.144926
```

Pentru a fi siguri de acest lucru comentăm linia cu printf și restaurăm valoarea lui num_steps. Executăm din nou și surpriză:

```
pi=3.141593
pi=0.887387
```

Rezultatul se încăpățânează să fie eronat din cauza celorlalte variabile partajate: variabila x trebuie declarată tot private și, cel mai important, variabila sum trebuie să colecteze sumele parțiale provenite de la toate firele de execuție (operație de reducere a rezultatului). Celelalte variabile pot fi declarate *shared*.

Algoritm paralel (variantă finală)

Vom rescrie directiva de paralelizare după cum urmează:

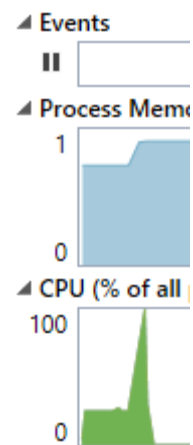
```
#pragma omp parallel num_threads(N) private(i,x)
                                shared(num_steps,step) reduction (+:sum)
```

și executăm programul din nou. Din fericire aceasta este varianta corectă de implementare a algoritmului paralel.

```
pi=3.141593
pi=3.141593
```

Clauza *reduction* permite colectarea unei valori de la mai multe fire de execuție și compunerea unei valori finale pe baza unui operator dat: +, *, -, &, &&, |, ||.

Observați modul de încărcare a procesorului (nucleelor procesorului) pe durata execuției codului secvențial și a codului paralel utilizând Diagnostic Tools. Cum estimați durata de execuție a celor două secțiuni?



Încercați să realizați o altă variantă de implementare a programului în care să nu utilizați clauza *reduction*. Utilizați, de exemplu, un vector care să stocheze valorile sumelor parțiale pentru fiecare fir de execuție în parte. Ce observați?