Introducere în OpenMP

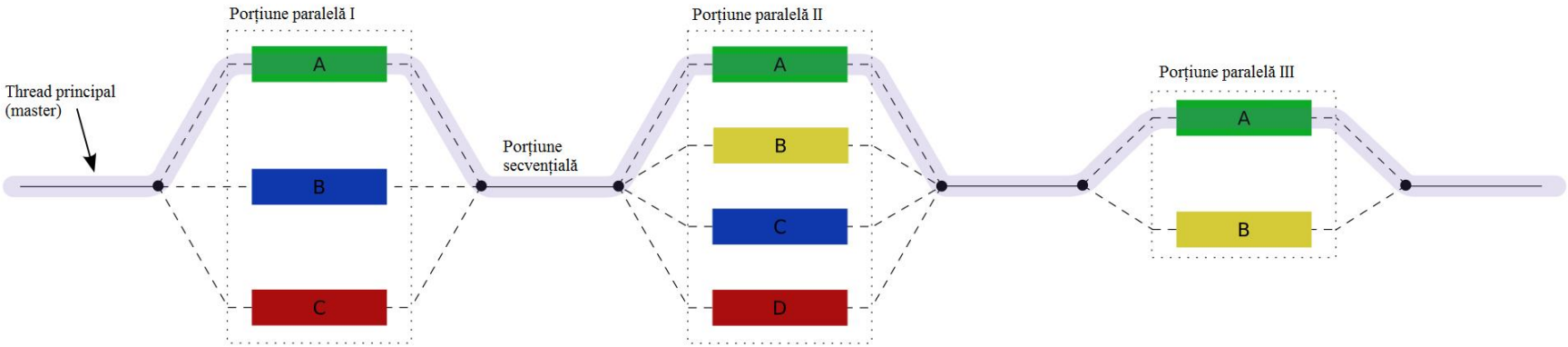
Cosmin – Ioan Niță

OpenMP

- API (Application Programming Interface) pentru crearea de aplicații ce se execută pe mai multe fire de execuție
- Conține directive de preprocesare, funcții și variabile specifice
- Simplifică mult programarea aplicațiilor multithreaded

Modelul fork-join

- Firul de execuție principal crează alte fire de execuție



- Firele de execuție au spațiu de memorie comun
- Proces vs. fir de execuție

Noțiuni introductive

- Activarea suportului pentru OpenMP la compilare (opțiunea */openmp*)
- Funcțiile specifice sunt definite în *omp.h*

  #include <omp.h>
- Se folosesc directive de preprocesare pentru controlul paralelismului

  #pragma omp [...]

  –Exemplu

  #pragma omp parallel num_threads(4)

Crearea firelor de execuție

- Se folosește directiva de preprocesare

  #pragma omp parallel

  {//porțiune paralelă}
- Fiecare fir de execuție va executa blocul de cod din {}
- Numărul de fire de execuție se specifică cu funcția *omp_set_num_threads();*
- Identificatorul firului de execuție curent se obține cu funcția *omp_get_thread_num();*

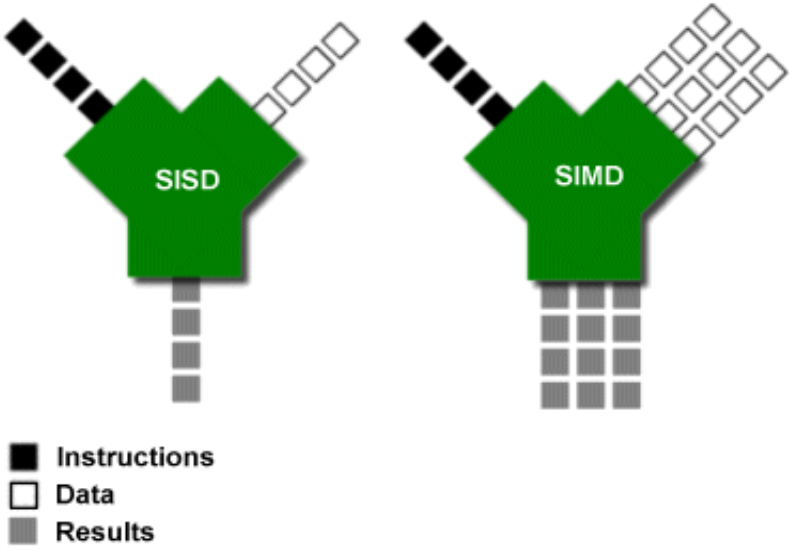
Exemplu

```
#include <omp.h>
#include <stdio.h>
void main()
{
 omp_set_num_threads(8);
#pragma omp parallel
{
int thread_id = omp_get_thread_num();
printf("Hello from thread %d\n",thread_id);
}

}
```

- Fiecare thread execută același cod
- Barieră de sincronizare la sfârșit

Exemplu

```
#include <omp.h>
#include <stdio.h>
void main()
{
            omp_set_num_threads(8);
           #pragma omp parallel
           {
                       int thread_id =
omp_get_thread_num();
                       printf("Hello from thread
%d\n", thread_id);
           }

}
```



Interactiunea firelor de executie

- Firele de execuție au spațiu de adrese (memorie) comună → comunică prin partajarea memoriei.
- Problema: ordinea în care firele de execuție accesează o resursă partajată
- Soluția: Sincronizarea firelor de execuție

Interactiunea firelor de executie

- Firele de execuție au spațiu de adrese (memorie) comună → comunică prin partajarea memoriei.
- Problema: ordinea în care firele de execuție accesează o resursă partajată
- Soluția: Sincronizarea firelor de execuție

Interactiunea firelor de execuție

- Clauza *private(lista)*: Declară variabilele din listă ca fiind private fiecărui fir de execuție

  #pragma omp parallel private(x,y,z)
- Clauza *shared(lista)*: Declară variabilele din listă ca fiind partajate fiecărui fir de execuție

  #pragma omp parallel shared(x,y,z)
- Iteratorul unei structuri repetitive va fi implicit privat
- Firstprivate și lastprivate

```
#include <omp.h>
#include <stdio.h>
void main()
{
int a=0;
 omp_set_num_threads(8);
#pragma omp parallel shared(a)
{
        a = omp_get_thread_num();
}
#pragma omp parallel private(a)
{
                a = omp_get_thread_num();
}
}
```

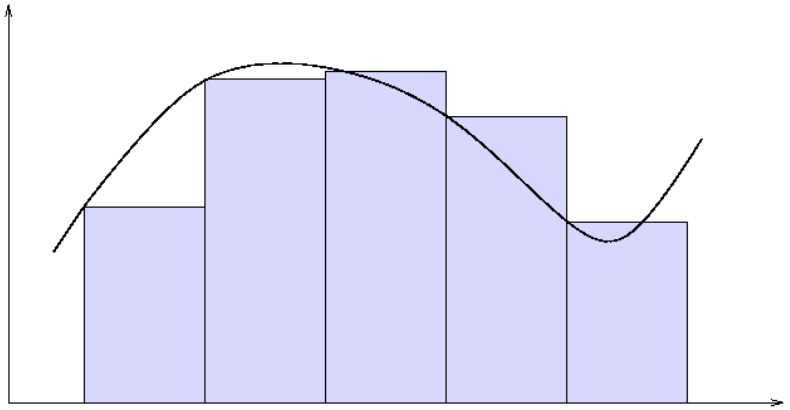
Exemplu 2: Estimarea lui π

- Valoarea exactă a lui π poate fi scrisă ca:

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

- Prin discretizare:

$$\sum_{i=0}^{N} \frac{4}{1+x_i^2}\,\Delta x \approx \pi$$



Exemplu 2: Varianta secvențială

```
#include <omp.h>
#include <stdio.h>
void main()
{
	int num_steps = 100000;
	double pi, sum = 0.0;
	double step = 1 / (double)num_steps;
	for (int i = 0; i < num_steps; i++){
		double x = i*step;
		sum += 4.0 / (1.0 + x*x);
	}
	pi = sum * step;
}
```

Exemplu 2: Varianta paralelă

- Fiecare fir de execuție va calcula o sumă parțială
- Sumele parțiale sunt ținute într-un tablou și sunt adunate după finalizarea secțiunii paralele

Exemplu 2: Varianta paralelă

- Fiecare fir de execuție va calcula o sumă parțială
- Sumele parțiale sunt ținute într-un tablou și sunt adunate după finalizarea secțiunii paralele

Sincronizarea firelor de execuție

- Este folosită pentru a controla ordinea în care fire de execuție diferite accesează o resursă comună
- Mai multe tipuri de sincronizare:
  - Secțiuni critice
  - Operații atomice
  - Barieră de sincronizare

Sincronizarea firelor de execuție – Excludere mutuală

- Secțiune critică – un singur fir de execuție poate intra intr-o sectiune critică la un moment dat

#pragma omp critical

```
int rezultat;
#pragma omp parallel
{
int id = omp_get_thread_num();
int rezultat_partial = calculeaza_partial(id);
#pragma omp critical
{
        rezultat =
calculeaza(rezultat_partial);
}
}
```

Sincronizarea firelor de execuție – Excludere mutuală

- Operație atomică – similar cu o secțiune critică doar că e valabilă pentru scrierea unei singure locații de memorie (*x [operator binar]=[expresie]*)

#pragma omp atomic

- Exemplu – estimarea lui pi

```
int rezultat;
#pragma omp parallel
{
int id = omp_get_thread_num();
int rezultat_partial = calculeaza_partial(id);
#pragma omp atomic
        rezultat += rezultat_partial;
}
```

Sincronizarea firelor de execuție – Directiva *barrier*

#pragma omp barrier

- Sincronizează toate firele de execuție.
- Bariera de sincronizare trebuie să fie întâlnită de toate firele de execuție sau nici unul.
- Exemplu

Sincronizarea firelor de execuție – Directiva *master*

```
#pragma omp master
{...}
```

- Specifică o regiune care este executata doar de thread-ul master.

Structuri repetitive paralele

- Iterațiile unei structuri repetitive sunt distribuite pe mai multe fire de execuție

```
#pragma omp parallel for
        for (int i = 0; i < N; i++)
                a[i] = b[i] + c[i];
```

- Iterațiile trebuie să fie independente

Structuri repetitive paralele

Cod secvențial:

```
for(int i = 0; < N; i++){do_stuff(i)};
```

Cod paralel (varianta 1):

```
#pragma omp parallel
{
        id = omp_get_thread_num();
        Nthreads = omp_get_num_threads();
        istart = id*N/Nthreads;
        iend(id+1)*N/Nthreads;
        for( int i = istart; i < iend; i++ )
        {
                do_stuff(i);
        }
}
```

Cod paralel (varianta 2) :

```
#pragma omp parallel for
for(int i = 0; < N; i++){do_stuff(i)};
```

Structuri repetitive paralele

- Modul prin care iteratiile sunt atribuite fiecărui fir de execuție poate fi specificat prin directiva *schedule(tip, chunk)*

```
#pragma omp parallel for schedule(dynamic,chunk)
        for (int i = 0; i < N; i++)
                a[i] = b[i] + c[i];
```

- Static vs. dinamic

Reducere paralelă

```
double sum = 0.0; A[N];
for (int i = 0; i < N; i++)
        sum += A[i];
```

- Iterațiile nu sunt independente. Fiecare fir de execuție modifică variabila *sum*
- Soluția: reducere paralelă
- Fiecare fir de execuție calculează o sumă parțială.

Reducere paralelă

- Clauza OpenMP pentru reducere paralelă

  reduction([operator, listă variabile])

- Exemplu

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < num_steps; i++){
  sum += A[i];
}
```

- Problema valorilor inițiale. OpenMP inițializează variabilele în funcție de operatorul specificat eg. „0” pentru „+” sau „1” pentru „*”

Exemplul 2: Reducere paralelă

- Estimarea lui pi prin reducere paralelă

Directiva *sections*

- Permite execuția în paralel a unor sectiuni diferite

```
#pragma omp sections
{
#pragma omp section{/*secțiune 1*/}
#pragma omp section{/*secțiune 2*/}
}
```

- Fiecare secțiune va fi executată de un singur thread

Exemplu 3: Estimarea valorii lui π printr-o metodă de tip Monte-Carlo

- Se generează puncte aleatoare în interiorul pătratului
- Se numără punctele ce se află în interiorul cercului
- Valoarea π poate fi aproximată prin
  $\pi \approx 4 \frac{N_c}{N_{total}}$
- Precizia crește cu cât numărul de iteratii crește