

## Lectia 13. CARACTERISTICI GENERALE ALE OPENMP. DIRECTIVE OPENMP

### 13.1 OpenMP-caracteristici.

OpenMP este o interfață de programare a aplicațiilor (API – Application Program Interface) care poate fi utilizată în paralelismul multifir cu memorie partajată (*multi-threaded, shared memory parallelism*).

Conține trei componente API primare:

Directive de Compilare (Compiler Directives),  
Rutine de Bibliotecă la Execuție (Runtime Library Routines),  
Variabile de Mediu (Environment Variables).

Este portabilă:

Această API este realizată în C/C++ și Fortran,  
S-a implementat pe platforme variate inclusiv pe cele mai multe platforme Unix și Windows.

Este standardizată:

Este definită și aprobată în comun de un grup de producători majori de hardware și software,  
Este de așteptat a deveni în viitor un standard ANSI (American National Standards Institute).

### Ce nu este OpenMP?

Nu este prin ea însăși destinată sistemelor paralel cu memorie distribuită,  
Nu este implementată în mod necesar identic de către toți producătorii,  
Nu este garantată că ar asigura utilizarea cea mai eficientă a memoriei partajate (nu există pentru moment constructori de localizarea datelor).

### Modelul de programare OpenMP se bazează pe:

Memorie Partajată și Paralelism pe bază de fire de execuție (*Shared Memory, Thread Based Parallelism*):

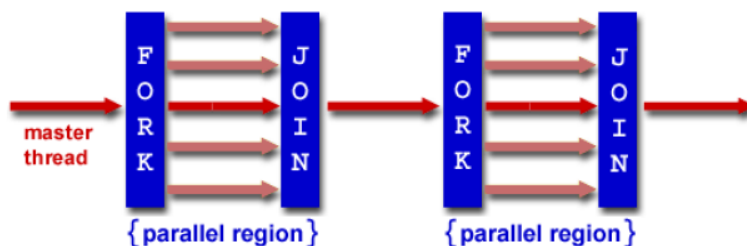
Un proces cu memorie partajată poate consta în fire de execuție multiple. OpenMP se bazează pe existența firelor multiple în paradigma programării cu partajarea memoriei.

Paralelism explicit:

OpenMP este un model de programare explicit (nu automat), care oferă programatorului deplinul control asupra paralelizării.

Modelul ramificație-joncțiune (fork-join model):

OpenMP utilizează de un model al execuției paralele alcătuit din ramificații și joncțiuni:



Toate programele OpenMP încep ca un proces unic, *firul master*. Firul master se execută secvențial până când ajunge la primul constructor numit *zonă/regiune paralelă*.

**Ramificație (fork):** firul master crează un mănunchi (fascicol) de fire paralele. Instrucțiunile din program care alcătuiesc o zonă paralelă sunt executate apoi paralel de diverite firele din fascicol.

**Joncțiune (join):** când firele din mănunchi termină de executat instrucțiunile din constructul zonă/regiune paralelă, ele se sincronizează și se încheie lăsând din nou numai firul master.

Bazat pe directive de compilare:

Virtual, tot paralelismul OpenMP este specificat prin directive de compilare care sunt încorporate în codul sursă C/C++ sau Fortran. Aceste directive reprezintă un comentariu pentru cazul neparalel (când se compilează ca un program secvențial)

Suportul pentru paralelismul stratificat:

Această interfață API asigură plasarea unui construcții paralele în interiorul altor construcții paralele.

Implementările existente pot să includă sau nu această particularitate.

Fire dinamice:

Această interfață API asigură modificarea dinamică a numărului de fire care pot fi utilizate pentru executarea unor zone paralele diferite.

Implementările existente pot să includă sau nu această particularitate.

Componentele API OpenMP pot fi reprezentate astfel:

Directives	Environment variables	Runtime environment
<ul style="list-style-type: none"> <li>◆ <b>Parallel regions</b></li> <li>◆ <b>Work sharing</b></li> <li>◆ <b>Synchronization</b></li> <li>◆ <b>Data scope attributes</b> <ul style="list-style-type: none"> <li>▢ <i>private</i></li> <li>▢ <i>firstprivate</i></li> <li>▢ <i>lastprivate</i></li> <li>▢ <i>shared</i></li> <li>▢ <i>reduction</i></li> </ul> </li> <li>◆ <b>Orphaning</b></li> </ul>	<ul style="list-style-type: none"> <li>◆ <b>Number of threads</b></li> <li>◆ <b>Scheduling type</b></li> <li>◆ <b>Dynamic thread adjustment</b></li> <li>◆ <b>Nested parallelism</b></li> </ul>	<ul style="list-style-type: none"> <li>◆ <b>Number of threads</b></li> <li>◆ <b>Thread ID</b></li> <li>◆ <b>Dynamic thread adjustment</b></li> <li>◆ <b>Nested parallelism</b></li> <li>◆ <b>Timers</b></li> <li>◆ <b>API for locking</b></li> </ul>

Mai jos este prezentat un exemplu de cod de program utilizand componentele OpenMP.

### **Exemple de structură a codurilor OpenMP.**

C / C++ – structura generală a codului:

```
#include <omp.h>
main()
{
    int var1, var2, var3;
    {
        Codul secvential. Inceputul secțiunii paralele. Ramificarea
        fluxului de fire. Specificarea domeniului variabilelor
        #pragma omp parallel private(var1,var2)shared (var3)
        {
            Secțiunea paralel executată de toate firele
            .....
            Toate firele se reunesc on firul master
        }
        {
            Se reia codul secvential
            .....
        }
    }
}
```

## **13.2 Directive OpenMP**

*Formatul directivelor C/C++:*

**#pragma omp nume de directivă [clauze ...] caracter newline**

Prefixul **#pragma omp** este obligatoriu pentru orice directivă OpenMP în C/C++. O directivă OpenMP validă trebuie să apară după **pragma** și înainte de orice clauză. Clauzele sunt optionale, pot fi în orice ordine, pot fi repetate (dacă nu sunt restricționări).

*Exemplu:*

```
#pragma omp parallel default(shared) private(beta,pi)
```

*Reguli generale:*

Directivele respectă convențiile din standardele C/C++ pentru directivele de compilare. Sensibilă la upper/lower case. Pentru o directivă, numai un nume de directivă poate fi specificat. Fiecare directivă se aplică la cel mult o declarație următoare, care poate fi un bloc structurat. Liniile-directivă lungi pot fi continuate pe linii următoare prin combinarea caracterelor **newline** (linie-nouă) cu un caracter “\” la finalul liniei-directivă.

#### ***Domeniul directivelor***

*Domeniu static (lexical):* Codul inclus textual între începutul și finalul unui bloc structurat care urmează directiva. Domeniul static al unei directive nu acoperă rutine multiple sau fișiere de cod.

*Directive orfane:* Despre o directivă OpenMP care apare independent de o altă directivă care o include se spune că este o directivă orfană. Ea există în afara domeniului static (lexical) al altei directive. Acoperă rutine și posibile fișiere de cod.

*Domeniu dinamic:* Domeniul dinamic al unei directive include atât domeniul ei static (lexical) cât și domeniile orfanelor ei.

### ***13.2.1 Constructorul de regiuni paralele.***

*Scopul acestuia:* o regiune paralela este un bloc de cod care va fi executat de mai multe fire. Este constructul paralel OpenMP fundamental.

*Formatul în C/C++:*

```
#pragma omp parallel [clause ...] newline
    if (scalar_expression)
    private (lista)
    shared (lista)
    default (shared | none)
    firstprivate (lista)
    reduction (operator: lista)
    copyin (lista)
```

***Notă:*** Când un fir ajunge la directiva **parallel**, el creează un mănunchi de fire și devine firul master al acelui fascicul. Master-ul este un membru al acelui fascicul și are numărul 0 în fascicul. La începutul acelei regiuni paralele, codul este copiat și este executat apoi de toate firele fasciculului. Există la finalul unei regiuni paralele o barieră implicită, numai firul master continuă execuția dincolo de acest punct.

***Câte fire?*** Numărul de fire dintr-o regiune paralelă este determinat de factorii următori, în ordinea prezentată:

se utilizează funcția de bibliotecă **omp\_set\_num\_threads()**,

se setează variabila de mediu **OMP\_NUM\_THREADS**,

implementarea default.

Firele sunt numerotate de la 0 (firul master) la **N - 1**.

***Fire dinamice:*** Prin default, un program cu regiuni paralele multiple utilizează același număr de fire pentru a executa fiecare dintre regiuni. Această comportare poate fi modificată pentru a permite la vremea execuției modificarea dinamică a firelor create pentru o anumită secțiune paralelă. Cele două metode disponibile pentru a permite fire dinamice sunt:

Utilizarea funcției de bibliotecă **omp\_set\_dynamic()**,

Setarea variabilei de mediu **OMP\_DYNAMIC**.

***Regiuni paralele una-în-alta (Nested Parallel Regions):*** O regiune paralelă una în alta rezultă prin crearea unui fascicul nou, constând într-un fir, prin default. Implementările pot permite mai mult de un fir în fasciculul cuprins în alt fascicul.

**Clauze:** Dacă apare clauza **IF**, ea trebuie să producă **.TRUE.** (în Fortran) sau o valoare nenulă (în C/C++) pentru a fi creat un fascicul de fire. Altminteri, regiunea este executată de firul master în mod secvențial.

**Restricții:** O regiune paralelă trebuie să fie un bloc structurat care nu acoperă rutine multiple sau fișiere de cod multiple. În Fortran, instrucțiunile I/O nesincrone prin care mai multe fire se referă la aceeași unitate au o comportare nespecific(at)ă. Este ilegală ramificarea din sau într-o regiune paralelă. Nu este permisă decât o singură clauză IF.

*Exemplu de regiune paralelă*

Programul “Hello World”. Fiecare fir execută întregul cod inclus în regiunea paralelă. Rutinele de bibliotecă OpenMP sunt utilizate pentru a obține identificatori de fire și numărul total de fire.

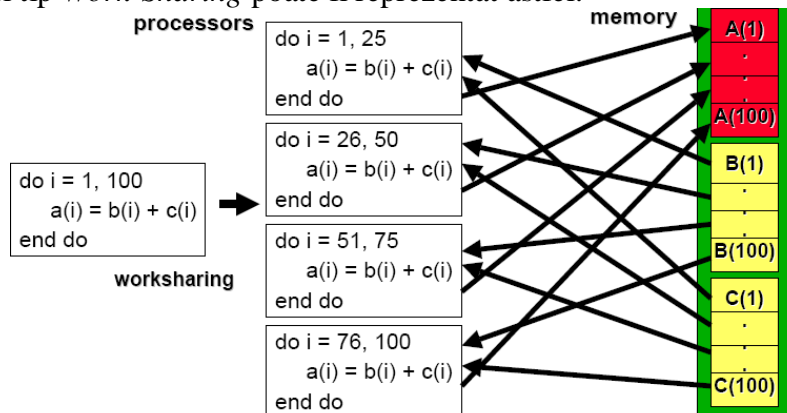
În C/C++:

```
main ()
{
  int nthreads, tid;
  /* Fork a team of threads giving them their own copies of
     variables */
  #pragma omp parallel private(tid)
  {
    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  } /* All threads join master thread and terminate */
}
```

### 13.2.2 Constructorul Work-Sharing (lucru partajat)

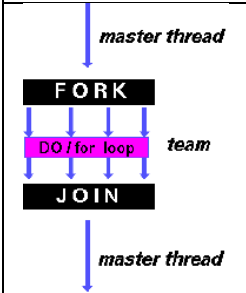
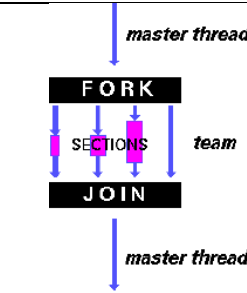
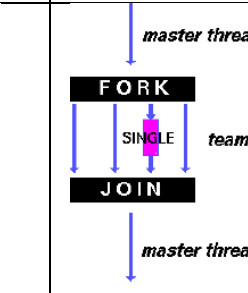
Un constructor *work-sharing* împarte execuția din regiunea de cod care îl include, între membrii fasciculului care îl întâlnesc. Constructorul *work-sharing* nu lansează fire noi. Nu există barieră implicită la intrarea într-un construct *work-sharing*, totuși există o barieră implicită la sfârșitul unui construct *work-sharing*.

Principiul unui tip *Work-Sharing* poate fi reprezentat astfel:



#### Tipuri de constructori work-sharing

<b>DO/for</b> – partajează	<b>SECTIONS</b> – divizează lucrul în	<b>SINGLE</b> – serializează o
-------------------------------	--	-----------------------------------

iterațiile unei bucle din fascicul. Reprezintă un tip de "paralelism pe date".	secțiuni separate, discrete. Fiecare secțiune este executată de un fir. Poate fi utilizat pentru implementarea unui tip de "paraleism functional".	secțiune de cod.
		

### Restricții:

Un constructor *work-sharing* trebuie să fie inclus dinamic într-o regiune *paralela* pentru ca directivele să fie executate în paralel. Constructorii *work-sharing* trebuie să fie «întâlnite» de toate firele membre ale unui fascicul sau de niciunul. Constructorii *work-sharing* trebuie să fie întâlnite în aceeași ordine de toate firele membre ale unui fascicul.

### Directiva for

**Scopul:** Directiva **for** specifică faptul că iterațiile buclei trebuie să fie executate de fascicul în paralel. Aceasta presupune că o regiune paralela a fost deja inițiată, altminteri ea se execută secvențial pe un singur procesor.

Formatul on C/C++:

```
#pragma omp for [clause ...] newline
    schedule (type [,chunk])
    ordered
    private (list)
    firstprivate (list)
    lastprivate (list)
    shared (list)
    reduction (operator: list)
    nowait

for_loop
```

### Clauzele utilizate în directiva DO/for:

**Clauza schedule:** descrie cum sunt partajate iterațiile buclei între firele fasciculului. Atât pentru Fortran cât și pentru C/C++ clauza poate fi de tipul:

**static:** Iterațiile buclei sunt împărțite în bucăți de dimensiunea **chunk** și apoi atribuite static firelor. Dacă **chunk** nu este precizată, iterațiile sunt împărțite (dacă este posibil) egal și continuu între fire.

**dynamic:** Iterațiile buclei sunt divizate în bucăți de dimensiunea **chunk** și distribuite dinamic între fire; când un fir încheie o bucată, i se atribuie dinamic alta. Dimensiunea bucăților prin default este 1.

**guided:** Dimensiunea fragmentului este redusă exponențial cu fiecare bucată distribuită a spațiului de iterații. Dimensiunea fragmentului specifică numărul minim de iterații de distribuit de fiecare dată. Dimensiunea bucăților prin default este 1.

**runtime:** Decizia de repartizare este amânată până la timpul execuției de variabila de mediu **OPM\_SCHEDULE**. Este ilegal a specifica dimensiunea fragmentului pentru această clauză. Repartizarea prin default este dependentă de implementare. Implementarea poate fi totodată întrucâtva variabilă în modul în care repartizările variate sunt implementate.

**Clauza ordered:** Trebuie să fie prezentă când în directiva **DO/for** sunt incluse directive **ordered**.

**Clauza NO WAIT (Fortran)/nowait (C/C++):** Dacă este specificată, atunci firele nu se sincronizează la finele buclei paralele. Firele trec direct la instrucțiunile următoare de după buclă.

*Restricții:*

Bucula **DO** nu poate fi o buclă **DO WHILE** sau o buclă fără control. Totodată, variabila de iterație a buclei trebuie să fie un întreg și parametrii de control ai buclei trebuie să fie aceiași pentru toate firele. Corectitudinea programului trebuie să nu depindă de câte fire execută o iterație particulară. Este ilegal a ramifica controlul înafara unei bucle asociate cu directiva **DO/for**. Dimensiunea fragmentului trebuie să fie specificată ca o expresie întreagă invariantă, ca și când nu există vreo sincronizare în timpul evaluării ei de fire diferite. Directiva **for** din C/C++ necesită ca bucla *for* să aibă forma canonică. Clauzele **ordered** și **schedule** pot apărea fiecare numai o dată.

Considerăm urmatorul fragment de program:

```
#pragma omp for schedule(static,16)
for (i=1; i < 128; i++)
    c(i) = a(i) + b(i);
```

În cazul când numărul de fire este 4 atunci fiecare fir va executa următoarele iterații:

<p><b>Thread 0: DO I = 1, 16</b>  C(I) = A(I) + B(I)  ENDDO</p> <p>DO I = 65, 80  C(I) = A(I) + B(I)  ENDDO</p>	<p><b>Thread 2: DO I = 33, 48</b>  C(I) = A(I) + B(I)  ENDDO</p> <p>DO I = 97, 112  C(I) = A(I) + B(I)  ENDDO</p>
<p><b>Thread 1: DO I = 17, 32</b>  C(I) = A(I) + B(I)  ENDDO</p> <p>DO I = 81, 96  C(I) = A(I) + B(I)  ENDDO</p>	<p><b>Thread 3: DO I = 49, 64</b>  C(I) = A(I) + B(I)  ENDDO</p> <p>DO I = 113, 128  C(I) = A(I) + B(I)  ENDDO</p>

**Exemplu 13.1.** Să se elaboreze un program OpenMP în care se determină suma a doi vectori. Firele vor calcula câte 100 de elemente ale vectorului. Să se calculeze câte elemente ale vectorului sumă vor fi determinate de fiecare fir în parte. Firele nu se vor sincroniza la încheierea lucrului lor.

Mai jos este prezentat codul programului în limbajul C++ în care se realizează condițiile enunțate în exemplu 13.1

```
#include <omp.h>
#include <stdio.h>
#include <iostream>
#define CHUNKSIZE 100
#define N 1000
main ()
{
```

```

int i, k, chunk, iam;
float a[N], b[N], c[N];
/* initializare vectorilor */
for (i=0; i < N; i++)
a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
#pragma omp parallel \ shared(a,b,c,chunk) private(i,k,iam)
{
k=0;
iam = omp_get_thread_num();
#pragma omp for schedule(static,chunk) nowait
for (i=0; i < N; i++)
{
c[i] = a[i] + b[i];
k=k+1;
}
sleep(iam);
printf("Procesul OpenMP cu numarul %d, a determinat %d elemente ale vectorului \n",
      iam, k);
}
return 0;
}

```

Rezultatele executării programului

Cazul ...schedule(dynamic,chunk)...

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o Exemplu1.2.1.exe Exemplu1.2.1.cpp1
```

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host compute-0-0,compute-0-1 Exemplu1.2.1.exe
```

Procesul OpenMP cu numarul 0, a determinat 500 elemente ale vectorului

Procesul OpenMP cu numarul 1, a determinat 400 elemente ale vectorului

Procesul OpenMP cu numarul 2, a determinat 0 elemente ale vectorului

Procesul OpenMP cu numarul 3, a determinat 100 elemente ale vectorului

Cazul ...schedule(static,chunk)...

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o Exemplu1.2.1.exe Exemplu1.2.1.cpp
```

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host compute-0-0,compute-0-1 Exemplu1.2.1.exe
```

Procesul OpenMP cu numarul 0, a determinat 300 elemente ale vectorului

Procesul OpenMP cu numarul 1, a determinat 300 elemente ale vectorului

Procesul OpenMP cu numarul 2, a determinat 200 elemente ale vectorului

Procesul OpenMP cu numarul 3, a determinat 200 elemente ale vectorului

```
[Hancu_B_S@hpc Open_
```

## Directiva **SECTIONS**

**Scop:** Directiva **sections** este un constructor de divizare a lucrului neiterativ. Ea specifică faptul că secțiunea/secțiunile de cod incluse sunt distribuite între firele din fascicol. Directive **section** independente pot fi așezate una într-alta în directiva **sections**. Fiecare **section** este executată o dată de un fir din fascicol. Secțiuni diferite pot fi executate de fire diferite. Este posibil

<sup>1</sup> Numele programului corespunde numelui exemplului din notele de curs Boris HÎNCU, Elena CALMÎȘ "MODELE DE PROGRAMARE PARALELĂ PE CLUSTERE. PARTEA II. Programare OpenMP și mixtă MPI-OpenMP". Chisinau 2018.



ca un fir să execute mai mult de o secțiune dacă firul este suficient de rapid și implementarea permite așa ceva.

*Format în C/C++:*

```
#pragma omp sections [clause ...] newline
private (list)
firstprivate (list)
lastprivate (list)
reduction (operator: list)
nowait
{
    #pragma omp section newline
    structured_block
    #pragma omp section newline
    structured_block
}
```

**Restricții:** La finalul unei directive **sections** există o barieră implicită cu excepția cazului în care se utilizează o clauză **nowait**. Clauzele sunt descrise în detaliu mai jos.

**Exemplu 13.2** Vom exemplifica modul de utilizare a directivei **sections** printr-un program de adunare simplă a vectorilor – similar exemplului utilizat mai sus pentru directiva **DO/for**. Primele  $n/2$  iterații ale buclei **for** sunt distribuite primului fir, restul se distribuie firului al doilea. Când un fir termină blocul lui de iterații, trece la executarea a ceea ce urmează conform codului (**nowait**).

Mai jos este prezentat codul programului în limbajul C++ în care se realizează condițiile enunțate în exemplu 13.2

```
#include <stdio.h>
#include <omp.h>
#include <iostream>
#define N 1000
main ()
{
    int i,iam;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    #pragma omp parallel shared(a,b,c) private(i,iam)
    {
        #pragma omp sections nowait
        {
            sleep(omp_get_thread_num());
            #pragma omp section
            {
                iam = omp_get_thread_num();
                for (i=0; i < N/2; i++)
                    c[i] = a[i] + b[i];
                printf("Procesul OpenMP cu numarul %d, a determinat %d elemente ale vectorului \n",
                    iam, N/2);
            }
        }
        #pragma omp section
        {
            iam = omp_get_thread_num();
```



```

for (i=N/2; i < N; i++)
c[i] = a[i] + b[i];
printf("Procesul OpenMP cu numarul %d, a determinat %d elemente ale vectorului \n",
    iam, N/2);
}
}
}

```

Rezultatele executării programului. Se generează 4 fire.

```

[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o Exemplu1.2.2.exe
Exemplu1.2.2.cpp
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host compute-0-
0,compute-0-1 Exemplu1.2.2.exe
Procesul OpenMP cu numarul 2, a determinat 500 elemente ale vectorului
Procesul OpenMP cu numarul 3, a determinat 500 elemente ale vectorului
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host compute-0-
0,compute-0-1 Exemplu1.2.2.exe
Procesul OpenMP cu numarul 0, a determinat 500 elemente ale vectorului
Procesul OpenMP cu numarul 1, a determinat 500 elemente ale vectorului
[Hancu_B_S@hpc Open_MP]$

```

### Directiva **SINGLE**

*Scop:* Directiva **single** specifică faptul că secvența de cod inclusă trebuie executată numai de un fir din fascicul. Poate fi utilă în tratarea secțiunilor codului care nu sunt sigure pe orice fir (cum sunt operațiile I/O).

*Formatul în C/C++:*

```

#pragma omp single [clause ...] newline
private (list)
firstprivate (list)
nowait
    structured_block

```

*Clauze:* Firele din fascicul care nu execută directiva **single** așteaptă la finalul blocului de cod inclus, cu excepția cazului în care este specificată o clauză **nowait** (C/C++).

*Restricții:* Este inacceptabil a ramifica în sau înafara unui bloc **single**.

### 13.2.3 Constructori de tipul **PARALLEL-WORK-SHARING**

OpenMP prezintă două (pentru C++) directive “mixte” pentru realizarea paralelizmului prin partajarea operațiilor (comenzilor):

parallel for  
parallel sections

De obicei aceste directive sunt echivalente cu directiva **parallel** urmată imediat de directivele **WORK-SHARING**. Acești constructor de obicei se utilizează atunci când constructorul **parallel** conține o singură directivă

### Directiva **parallel for**

O reprezentare grafică a utilizării aceste directive pentru paralelizarea problemei înmulțirii unei matrici cu un vector:

```
#pragma omp parallel for default(none) \
        private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
    sum = 0.0;
    for (j=0; j<n; j++)
        sum += b[i][j]*c[j];
    a[i] = sum;
}
```

TID = 0

```
for (i=0,1,2,3,4)
i = 0
sum =  $\sum b[i=0][j]*c[j]$ 
a[0] = sum
i = 1
sum =  $\sum b[i=1][j]*c[j]$ 
a[1] = sum
```

TID = 1

```
for (i=5,6,7,8,9)
i = 5
sum =  $\sum b[i=5][j]*c[j]$ 
a[5] = sum
i = 6
sum =  $\sum b[i=6][j]*c[j]$ 
a[6] = sum
```

... etc ...

În cazul folosirii directivei **parallel for** programul din Exemplu 13.1 va avea următoarea formă

```
#include <omp.h>
#include<stdio.h>
#include <iostream>
#define CHUNKSIZE 100
#define N 1000
main ()
{
    int i, chunk, iam;
    float a[N], b[N], c[N];
    /* initializare vectorilor */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
    #pragma omp parallel for \ shared(a,b,c,chunk) private(i,k,iam)\
    schedule(static,chunk) nowait
    for (i=0; i < N; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```

Observăm că în acest caz nu vom putea determina câte elemente ale vectorului sumă vor fi determinate de fiecare fir în parte.

### Directiva PARALLEL SECTIONS

*Scop:* Directiva **parallel sections** specifică o regiune paralela care contine o directivă **sections** unică. Directiva **sections** unică trebuie să urmeze imediat, ca declaratie imediat următoare.

*Format în C/C++:*

```
#pragma omp parallel sections [clause ...] newline
default (shared | none)
shared (list)
private (list)
firstprivate (list)
lastprivate (list)
reduction (operator: list)
```

```
copyin (list)
ordered
    structured_block
```

**Clauze:** Clauzele acceptate pot fi oricare din cele acceptate de directivele **parallel** și **sections**. Clauzele neanalizate încă sunt descrise în detaliu mai jos.

#### 13.2. 4 Constructori de sincronizare

Se consideră un exemplu<sup>2</sup> simplu în care două fire pe două procesoare diferite încearcă ambele să incrementeze o variabilă **x** în același timp (se presupune că **x** se initializează cu 0):

<b>THREAD 1:</b> increment(x) { x = x + 1; } <b>THREAD 1:</b> 10 LOAD A, (x address) 20 ADD A, 1 30 STORE A, (x address)	<b>THREAD 2:</b> increment(x) { x = x + 1; } <b>THREAD 2:</b> 10 LOAD A, (x address) 20 ADD A, 1 30 STORE A, (x address)
--	--

O variantă de execuție posibilă este: Firul 1 încarcă valoarea lui **x** în registrul A. Firul 2 încarcă valoarea lui **x** în registrul A. Firul 1 adună 1 la registrul A. Firul 2 adună 1 la registrul A. Firul 1 depune registrul A în locația **x**. Firul 2 depune registrul A în locația **x**. Valoarea rezultantă pentru **x** va fi 1 nu 2 cum ar trebui. Pentru a evita situațiile de acest gen, incrementarea lui **x** trebuie să fie sincronizată între cele două fire pentru a ne asigura de rezultatul corect. OpenMP asigură o varietate de constructe de sincronizare care controlează cum se derulează execuția fiecărui fir în relație cu alte fire ale fasciculului.

#### Directiva MASTER

**Scop:** Directiva **master** specifică o regiune care trebuie executată numai de firul master al fasciculului. Toate celelalte fire din fascicul sar această secțiune a codului. Nu există o barieră implicită asociată cu această directivă.

*Formatul C/C++:*

```
#pragma omp master newline
    structured_block
```

**Restricții:** Este interzis a ramifica în sau în afara blocului **master**.

#### Directiva CRITICAL

**Scop:** Directiva **critical** specifică o regiune de cod care trebuie executată succesiv (nu concomitent) de firele din fascicul.

*Formatul C/C++:*

```
#pragma omp critical [ name ] newline
    structured_block
```

**Note:** Dacă un fir execută curent o regiune **critical** și un altul ajunge la acea regiune **critical** și încearcă să o execute, el va sta blocat până când primul fir părăsește regiunea **critical**. Un nume optional face posibilă existența regiunilor **critical** multiple: numele acționează ca identificatori globali. Regiunile **critical** diferite cu același nume sunt tratate ca aceeași regiune. Toate secțiunile **critical** fără nume sunt tratate ca o aceeași secțiune.

**Restricții:** Nu este permis a se ramifica controlul în sau înafara unui bloc **critical**.

**Exemplu 13.3** In acest exemplu se ilustrează modul de gestionare la nivel de program a secțiunilor critice:

<sup>2</sup> De fapt acesta-i exemplu clasic de “secvența critică”

*In C/C++:*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    int x,i;
    x=0;
    omp_set_num_threads(2);
    #pragma omp parallel shared(x) private(i)
    {
        for(i =0;i<1000;i++)
        {
            #pragma omp critical {
                x=x+1;
            }
        }
    }
    printf("Valoarea lui x=%d \n", x);
}
```

Rezultatele executării programului. Cazul cand se utilizeaza directiva **#pragma omp critical**

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpiCC -fopenmp -o Exemplu1.2.3.exe Exemplu1.2.3.cpp
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host compute-0-0,compute-0-1 Exemplu1.2.3.exe
Valoarea lui x=2000
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host compute-0-0,compute-0-1 Exemplu1.2.3.exe
Valoarea lui x=2000
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 2 -host compute-0-0,compute-0-1 Exemplu1.2.3.exe
Valoarea lui x=2000
Valoarea lui x=2000
[Hancu_B_S@hpc Open_MP]$
```

Cazul cand nu se utilizeaza directiva **#pragma omp critical**

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpiCC -fopenmp -o Exemplu1.2.3.exe Exemplu1.2.3.cpp
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host compute-0-0,compute-0-1 Exemplu1.2.3.exe
Valoarea lui x=1492
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host compute-0-0,compute-0-1 Exemplu1.2.3.exe
Valoarea lui x=1127
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 2 -host compute-0-0,compute-0-1 Exemplu1.2.3.exe
Valoarea lui x=1128
Valoarea lui x=1118
[Hancu_B_S@hpc Open_MP]$
```

## Directiva BARRIER

*Scop:* Directiva **barrier** sincronizează toate firele unui fascicul. Când o directivă **barrier** este atinsă, orice fir așteaptă în acel punct până când toate celelalte fire ating și ele aceea barieră. Toate firele reiau atunci execuția în paralel a codului.

*Format în C/C++:*

```
#pragma omp barrier newline
```

*Restricții:* în C/C++, cea mai mică declarație care conține o barieră trebuie să fie un bloc structurat. De exemplu:

```
GRESIT
if (x == 0)
#pragma omp barrier
```

```
CORECT
if (x == 0)
{
#pragma omp barrier
}
```

### Directiva ATOMIC

*Scop:* Directiva **atomic** specifică faptul că o locație particulară de memorie trebuie să fie actualizată atomic și interzice ca mai multe fire să încerce să scrie în ea. În esență, această directivă asigură o mini-sectiune **critical**.

*Format în C/C++:*

```
#pragma omp atomic newline
statement_expression
```

*Restricții:* Directiva se aplică numai unei declarații, cea imediat următoare.

### Directiva ORDERED

*Scop:* Directiva **ordered** specifică faptul că iterațiile buclei incluse vor fi executate în aceeași ordine ca și când ar fi executate de un procesor secvențial.

*Formatul în C/C++:*

```
#pragma omp ordered newline
structured_block
```

*Restricții:* O directivă **ordered** poate apărea numai în extensia dinamică a directivelor **do** sau **parallel do** din Fortran și **parallel for** din C/C++. Numai un fir este permis într-o secțiune “**ordered**” la un moment dat. Nu este permisă ramificarea în sau din blocurile **ordered**. O iterație a unei bucle nu trebuie să execute aceeași directivă **ordered** mai mult decât o dată și nu trebuie să execute mai mult de o directivă **ordered**. O buclă care conține o directivă **ordered** trebuie să fie o buclă cu o clauză **ordered**.

### Directiva THREADPRIVATE

*Scopul:* Directiva **threadprivate** este folosită pentru a face variabilele de domeniu fișier global (în C/C++) locale și persistente pentru un fir în execuție de regiuni paralele multiple.

*Formatul în C/C++:*

```
#pragma omp threadprivate (list)
```

*Note:* Directiva trebuie să apară după declarația listei de variabile. Fiecare fir își ia apoi propria sa copie a variabilelor, astfel datele scrise de un fir nu sunt vizibile celorlalte fire.

La prima intrare într-o regiune paralelă, datele din variabilele **threadprivate** ar trebui presupuse nedefinite, cu excepția cazului în care în directiva **parallel** este menționată clauza **copyin**. Variabilele **threadprivate** diferă de variabilele **private** (discutate mai jos) deoarece ele sunt abilitate să persiste între secțiuni paralele diferite ale codului.

*Restricții:* Datele din obiectele **threadprivate** sunt garantate a persista numai dacă mecanismul firelor dinamice este închis (**turned off**) și numărul de fire în regiuni paralele diferite rămâne constant. Setarea prin default a firelor dinamice este nedefinită. Directiva **threadprivate** trebuie să apară după fiecare declarație a unei variabile private/unui bloc comun din fir.

**Exemplu 13.4.** În acest exemplu se ilustrează modul de utilizare a directivei **threadprivate**.

Mai jos este prezentat codul programului în limbajul C++ în care se realizează condițiile enunțate în exemplu 13.4

```
#include <stdio.h>
#include <omp.h>
#include <iostream>
int a, b, i, tid;
float x;
#pragma omp threadprivate(a, x)
main ()
{
/* Explicitly turn off dynamic threads */
omp_set_dynamic(0);
printf("Prima regiune paralela:\n");
#pragma omp parallel private(b,tid)
{
tid = omp_get_thread_num();
a = tid;
b = tid;
x = 1.1 * tid + 1.0;
sleep(omp_get_thread_num());
printf("Procesul OpenMP %d: a,b,x= %d %d %f\n",tid,a,b,x);
} /* end of parallel section */
printf("*****\n");
printf("Aici firul Master executa un cod serial\n");
printf("*****\n");
printf("A doua regiune paralela:\n");
#pragma omp parallel private(tid)
{
tid = omp_get_thread_num();
sleep(omp_get_thread_num());
printf("Procesul OpenMP %d: a,b,x= %d %d %f\n",tid,a,b,x);
} /* end of parallel section */
}Rezultatele executării programului. Se generează 4 procese OpenMP (fire).
[[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpiCC -fopenmp -o Exemplu1.2.4.exe
Exemplu1.2.4.cpp
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host compute-0-0,compute-
0-1 Exemplu1.2.4.exe
Prima regiune paralela:
Procesul OpenMP 0: a,b,x= 0 0 1.000000
Procesul OpenMP 1: a,b,x= 1 1 2.100000
Procesul OpenMP 2: a,b,x= 2 2 3.200000
Procesul OpenMP 3: a,b,x= 3 3 4.300000
*****
Aici firul Master executa un cod serial
*****
A doua regiune paralela:
Procesul OpenMP 0: a,b,x= 0 0 1.000000
Procesul OpenMP 1: a,b,x= 1 0 2.100000
Procesul OpenMP 2: a,b,x= 2 0 3.200000
Procesul OpenMP 3: a,b,x= 3 0 4.300000
[Hancu_B_S@hpc Open_MP]$
```

Din acest exemplu se observă că valorile variabilelor  $\mathbf{a}$  și  $\mathbf{x}$  se pastează și în a doua regiune paralela, pe când variabila  $\mathbf{b}$  nu este determinată.