

**UNIVERSITATEA DE STAT DIN MOLDOVA**  
**Facultatea de Matematică și Informatică**

**Modele de programare paralelă pe clustere.**

***Partea II***  
***Programare OpenMP și mixtă MPI-OpenMP.***

*Note de curs*

*Aprobat de Consiliul*  
*Facultății de Matematică și Informatică*

CEP USM  
Chișinău, 2018

CZU .....

C .....

Recomandat de Departamentul Matematică și de Comisia de Asigurare a Calității

Autori: **Boris HÎNCU, Elena CALMÎȘ**

Responsabil de ediție: **Boris HÎNCU**, conferențiar universitar, doctor

Recenzent: **Secrieru Grigore**, doctor în șt. fiz.-matem., conf. univ.,  
cercetător științific coordonator IMI al AȘM

### Descrierea CIP a Camerei Naționale a Cărții

**Modele de programare paralela pe clustere** / Boris Hîncu, Elena Calmîș; Universitatea de Stat din Moldova. Facultatea de Matematică și Informatică, Departamentul Matematici – Chișinău: CEP USM, 2018.

..... ex.

ISBN .....

.....

.....

© Boris Hîncu, Elena Calmîș, 2018

© CEP USM, 2018

ISBN .....

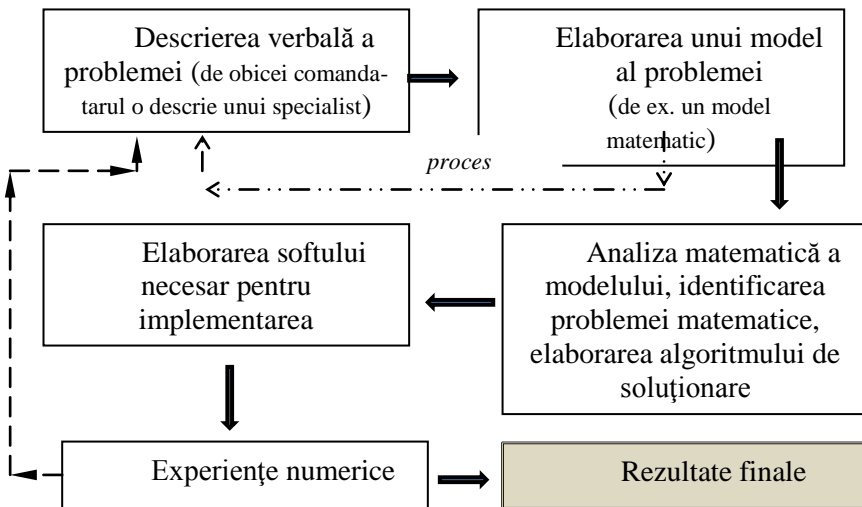
## Cuprins

Introducere.....	5
Capitolul 1. Modele de programare paralelă cu memorie comună:	
Modelul de programare OpenMP .....	9
1.1 Caracteristici generale ale OpenMP .....	9
1.2 Directive OpenMP .....	13
1.2.1 Constructorul de regiuni paralele. ....	13
1.2.2 Constructorul Work-Sharing (lucru partajat).....	16
1.2.3 Constructori de tipul PARALLEL-WORK-SHARING .....	24
1.2.4 Constructori de sincronizare .....	26
Capitolul 2. Modalitati de gestionare a datelor în OpenMP. ....	33
2.1 Clauze privind atributele de domeniu al datelor (Data Scope Attribute Clauses).....	33
2.1.1. Clauza PRIVATE .....	34
2.1.2. Clauza SHARED .....	35
2.1.3. Clauza DEFAULT .....	36
2.1.4. Clauza FIRSTPRIVATE .....	36
2.1.5. Clauza LASTPRIVATE .....	38
2.1.6. Clauza COPYIN .....	40
2.1.6. Clauza REDUCTION .....	41
Capitolul 3. Rutinele de bibliotecă run-time (Run-Time Library Routines) și variabile de mediu (Environment Variables) .....	47
3.1 Privire generală:.....	47
3.2 Rutine utilizate pentru setarea și returnarea numărului de fire ...	48
3.3 Rutina utilizata pentru returnarea identificadorul firului.....	51
3.4 Rutinele utilizate pentru generarea dinamica a firelor .....	52
3.5 Rutinele utilizate pentru generarea paralelismului unul-în-altul (nested parallelism) .....	55
3.6 Rutinele utilizate pentru blocări de domeniu a firelor .....	57
3.7 Rutine portabile pentru măsurarea timpului universal (wall clock time).....	60
3.8 Variable de mediu (de programare).....	61

Capitolul 4. Aspecte comparative ale modelelor de programare paralela MPI și OpenMP .....	63
4.1 Preliminarii .....	63
4.2 Programare paralelă mixtă MPI și OpenMP .....	65
4.3 Scenariu de execuaree mixtă MPI-OpenMP <b>Error! Bookmark not defined.</b>	
4.4 Modalitati de comunicare în sisteme paralele hibrid (MPI- OpenMP). .....	77
<b>Bibliografie</b> .....	80
<b>Anexă</b> .....	81

# Introdurre

Tehnologiile informaționale servesc drept suport (instrumentariu) pentru rezolvarea diferitor probleme teoretice și practice generate de activitatea umană. În general, procesul de rezolvare a problemelor cu caracter aplicativ poate fi reprezentat prin următoarea schemă:



O vastă clasă de probleme sunt de o complexitate foarte mare atât sub aspect de volum de date, cât și sub cel al numărului de operații. Astfel, apare o necesitate stringentă de a utiliza sisteme de calcul performant sau supercalculatoare. În prezent cele mai utilizate supercalculatoare sunt *calculatoarele paralele de tip cluster*.

În baza proiectului CRDF/MRDA „Project CERIM-1006-06” la Facultatea de Matematică și Informatică a Universității de Stat din Moldova a fost creat un laborator pentru utilizarea sistemelor paralele de calcul de tip cluster. Scopul acestui laborator este implementarea în procesul de instruire și cercetare a noilor tehnologii pentru elaborarea algoritmilor paraleli și executarea lor pe sisteme locale de tip cluster și chiar pe sisteme regionale de tip Grid. Actualmente laboratorul conține următoarele echipamente de calcul și de infrastructură:

- **1 server HP ProLiant DL380 G5**
  - ✓ CPU: 2x Dual Core Intel Xeon 5150 (2.7GHz)

- ✓ RAM: 8GB
- ✓ HDD: 2x72GB, 5x146GB
- ✓ Network: 3x1Gbps LAN

Acest echipament este utilizat la gestionarea mașinilor virtuale necesare pentru administrarea laboratorului, precum Web Server, FTP Server, PXE Server, Untangle server etc.

- **1 server HP ProLiant DL380 G5**
  - ✓ CPU: 2x QuadCore Intel Xeon 5420 (2.5 GHz)
  - ✓ RAM: 32GB
  - ✓ HDD: 2x72GB, 6x146GB
  - ✓ Network: 3x1Gbps LAN
- **2 servere HP ProLiant DL385G1**
  - ✓ CPU: 2xAMD 280 Dual-Core 2.4GHz
  - ✓ RAM: 6GB
  - ✓ HDD: SmartArray 6i, 4x146GB
  - ✓ Network: 3x1Gbps LAN

Aceste echipamente sunt utilizate pentru asigurarea logisticii necesare (la nivel de hard, soft și instruire de resurse umane) la realizarea și utilizarea MD-GRID NGI și a infrastructurii gridului European (EGI), care permit extinderea modalităților de utilizare a clusterului USM pentru realizarea unor calcule performante.

- **1 server HP ProLiant DL385G1 și 12 noduri HP ProLiant DL145R02**
  - ✓ CPU: 2xAMD 275 Dual-Core 2.2GHz
  - ✓ RAM: 4GB AECC PC3200 DDR
  - ✓ HDD: 80GB NHP SATA
  - ✓ Network: 3x1Gbps LAN
- **Storage HP SmartArray 6402**
  - ✓ hp StorageWorks MSA20
  - ✓ HDD: 4x500GB 7.2k SATA
- **Switch HP ProCurve 2650 (48 ports)**

Aceste echipamente sunt utilizate pentru elaborarea și implementarea soft a diferitor clase de algoritmi paraleli, efectuarea cursurilor de prelegeri și laborator la disciplinele corespunzătoare pentru ciclurile 1 și 2 de studii, precum și în cadrul diferitor proiecte de cercetare.

- **3 stații de management HP dx5150**
  - ✓ CPU: Athlon64 3200+

- ✓ RAM: 1GB PC3200 DDR
- ✓ Storage: 80GB SATA, DVD-RW
- ✓ Network: 1Gbps LAN
- ✓ Monitor: HP L1940 LCD

Aceste echipamente sunt utilizate pentru administrarea clusterului paralel și a infrastructurii sale de la distanță.

- **14 stații de lucru HP dx5150**

- ✓ CPU: Athlon64 3200+
- ✓ RAM: 512MB PC3200 DDR
- ✓ HDD: 80GB SATA
- ✓ Network: 1Gbps LAN
- ✓ Monitor: HP L1706 LCD

- **Tablă interactivă (Smart board) și proiectoare.**

Aceste echipamente sunt utilizate drept stații de lucru pentru realizarea procesului didactic și de cercetare la Facultatea de Matematică și Informatică.

Clusterul paralel este înzestrat cu sisteme de calcul paralel licențiate sau „open source” necesare pentru realizarea procesului didactic și de cercetare necesar.

Notele de curs „Modele de programare paralelă. Partea II. Programare OpenMP și mixtă MPI-OpenMP Programare MPI” își propun să acopere minimul de noțiuni necesare înțelegerii modalităților de implementare software a algoritmilor paraleli pe diverse sisteme paralele de calcul de tip cluster. Cursul va contribui esențial la dezvoltarea aptitudinilor și capacităților de construire, studiere a algoritmilor paraleli și implementarea lor în diferite sisteme paralele de calcul. Drept rezultat al cunoștințelor acumulate, studentul trebuie să poată aplica cele mai importante metode și rezultate expuse în lucrare, pentru implementarea celor mai moderne realizări din domeniul informaticii și tehnologiilor informaționale.

Lucrarea dată acoperă obiectivele de referință și unitățile de conținut prezente în Curriculum la disciplina *Programare Paralelă*. Această disciplină, pe parcursul a mai mulți ani, este predată la specialitățile *Informatica*, *Managementul Informațional* ale Facultății de Matematică și Informatică și la specialitatea *Tehnologii Informaționale* a Facultății de Fizică și Inginerie.

Această lucrare are drept scop dezvoltarea următoarelor competențe generice și specifice:

- cunoașterea bazelor teoretice generale ale matematicii și informaticii necesare la alcătuirea modelului problemei practice cu caracter socioeconomic;
- aplicarea de metode noi de cercetare și implementare soft a algoritmilor paraleli;
- identificarea căilor și a metodelor de utilizare a rezultatelor obținute și în alte domenii de activitate;
- elaborarea și analiza algoritmilor paraleli de soluționare a problemelor de o mare complexitate;
- implementarea metodelor noi și concepții moderne în realizarea lucrărilor proprii;
- posedarea diferitor abilități de analiză, sinteză și evaluare în abordarea și soluționarea diferitor probleme;
- deținerea capacităților și a deprinderilor necesare pentru realizarea proiectelor de cercetare, demonstrând un grad înalt de autonomie.

Programele prezentate în „Exemple” au fost elaborate și testate cu suportul studenților de la specialitatea „Informatică” a Facultății de Matematică și Informatică.

Cunoștințele acumulate la acest curs vor fi utilizate și la studierea cursului *Calcul paralel pe clustere* pentru studii de masterat.

La elaborarea acestui suport de curs au fost consultate sursele bibliografice prezente în „Bibliografie”.



## **Capitolul 1. Modele de programare paralelă cu memorie comună: Modelul de programare OpenMP**

### **Obiective**

- Să definească noțiunea de model de programare paralelă;
- Să cunoască criteriile de clasificare a sistemelor paralele de calcul;
- Să cunoască particularitățile de bază la elaborarea programelor paralele ale sistemelor de calcul cu memorie partajată, cu memorie distribuită și mixte;

### ***1.1 Caracteristici generale ale OpenMP***

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:**

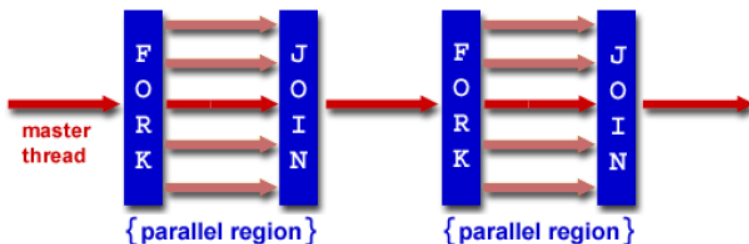
- Memorie Partajată și Paralelism pe bază de fire de execuție (*Shared Memory, Thread Based Parallelism*):
- Un proces cu memorie partajată poate consta în fire de execuție multiple. OpenMP se bazează pe existența firelor multiple în paradigma programării cu partajarea memoriei.

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 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 (joint)*: 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:

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

## 1.2 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 restrictionări).

*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.

### 1.2.1 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 paralel 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();
```

```

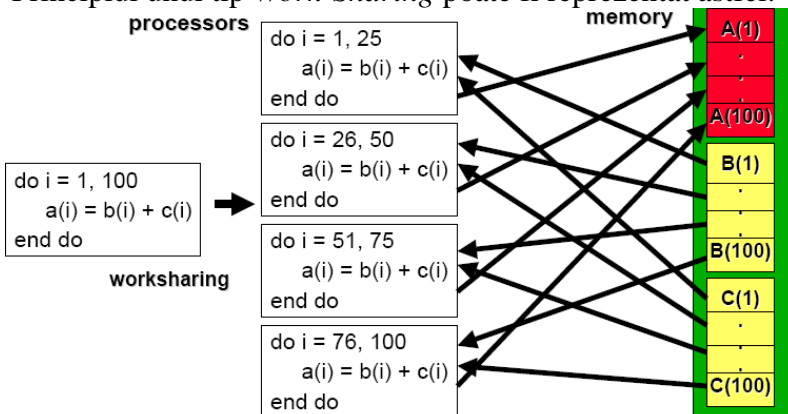
printf("Number of threads = %d\n",
      nthreads);
}
} /* All threads join master thread and
   terminate */
}

```

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

Un constructor *work-sharing* împarte execuția din regiunea de cod care îl include, între membrii fascicului 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*.

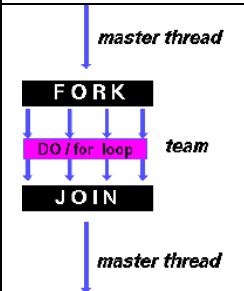
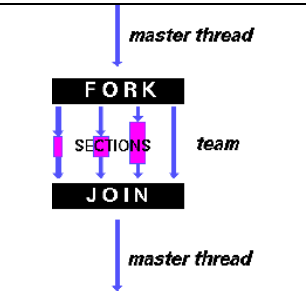
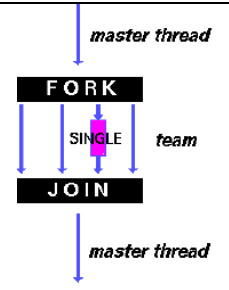
Principiul unui tip *Work-Sharing* poate fi reprezentat astfel:



### Tipuri de constructori work-sharing

<b>DO/for</b> – partajează iterațiile unei bucle din fascicul. Reprezintă un	<b>SECTIONS</b> – divizează lucrul în secțiuni separate, discrete. Fiecare secțiune este executată de un fir.	<b>SINGLE</b> – serializează o secțiune de cod.
---	--	---



tip de "paralelism pe date".	Poate fi utilizat pentru implementarea unui tip de "paraleism functional".	
 <p>The diagram shows a master thread (blue arrow) entering a <b>FORK</b> block. Inside, a team of threads (blue arrows) executes a <b>DO/for loop</b> (pink box) in parallel. After the loop, the threads join at a <b>JOIN</b> block, and the master thread continues.</p>	 <p>The diagram shows a master thread (blue arrow) entering a <b>FORK</b> block. Inside, a team of threads (blue arrows) executes different <b>SECTIONS</b> (pink boxes) in parallel. After the sections, the threads join at a <b>JOIN</b> block, and the master thread continues.</p>	 <p>The diagram shows a master thread (blue arrow) entering a <b>FORK</b> block. Inside, a team of threads (blue arrows) executes a <b>SINGLE</b> task (pink box) in parallel. After the task, the threads join at a <b>JOIN</b> block, and the master thread continues.</p>

### **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 fragmetului 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.

Consideram 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 1.2.1.** *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 1.2.1

```

#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.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 numărul 0, a determinat 500 elemente ale vectorului

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

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

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

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

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

Procesul OpenMP cu numărul 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 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 1.2.2** 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 1.2.2

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

```
}
}
```

Rezultatele executării programului. Se genereaza 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** asteaptă 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**.

### 1.2.3 Constructori de tipul PARALLEL-WORK-SHARING

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

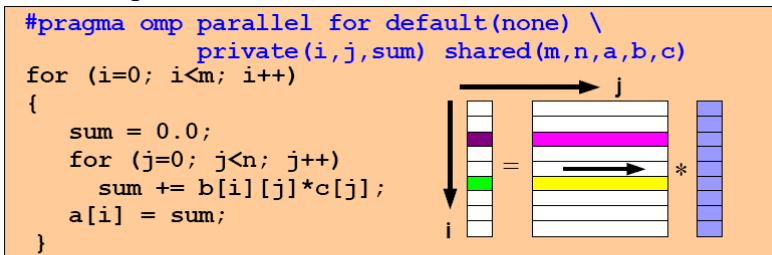


- parallel for
- parallel sections

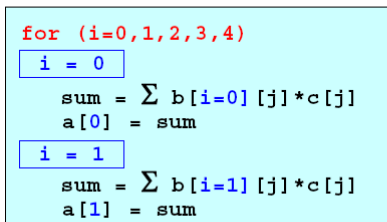
De obicei aceste directive sunt echivalente cu directiva **parallel** urmată imediat de directivele **WORK-SHARING**. Acești constructor de obicei se utilizează atunci când constructorul **parallel** conține o singură directivă

### Directiva **parallel for**

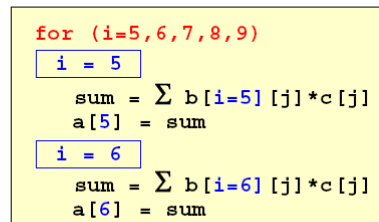
O reprezentare grafică a utilizării acestor directive pentru paralelizarea problemei înmulțirii unei matrici cu un vector:



**TID = 0**



**TID = 1**



**... etc ...**

În cazul folosirii directivei **parallel for** programul din Exemplu 1.2.1 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;
```

<pre>#pragma omp parallel for \     shared(a,b,c,chunk)     private(i,k,iam)\ schedule(static,chunk) nowait</pre>	<pre>for (i=0; i &lt; N; i++) {     c[i] = a[i] + b[i]; }</pre>
---	---

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.

### 1.2.4 Constructori de sincronizare

Se consideră un exemplu<sup>1</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):

---

<sup>1</sup> De fapt acesta-i exemplu clasic de “secvența critică”

<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 locatia  $x$ . Firul 2 depune registrul A în locatia  $x$ . 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 structuri de sincronizare care controlează cum se derulează execuția fiecărui fir în relație cu alte fire ale fascicului.

### Directiva MASTER

**Scop:** Directiva **master** specifică o regiune care trebuie executată numai de firul master al fascicului. 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 1.2.3** *In acest exemplu se ilustrează modul de gestionare la nivel de program a secțiunilor critice:*

*In C/C++:*

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

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 acea barieră. Toate firele reiau atunci executia în paralel a codului.

*Format în C/C++:*

```
#pragma omp barrier newline
```

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

GRESIT	CORECT
if (x == 0)	if (x == 0)
#pragma omp barrier	{
	#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 paralela, 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 1.2.4.** 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 1.2.4

<pre>#include &lt;stdio.h&gt; #include &lt;omp.h&gt; #include &lt;iostream&gt; 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());</pre>	<pre>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 */ }</pre>
--	--

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 **a** și **x** se pastează și în a doua regiune paralela, pe când variabila **b** nu este determinată.



## Capitolul 2. Modalitati de gestionare a datelor în OpenMP.

### Obiective

- Să definească noțiunea de model de programare paralelă;
- Să cunoască criteriile de clasificare a sistemelor paralele de calcul;
- Să cunoască particularitățile de bază la elaborarea programelor paralele ale sistemelor de calcul cu memorie partajată, cu memorie distribuită și mixte;

Să definească noțiunea de secvență critică și să poată utiliza în programe paralele astfel de secvențe.

### 2.1 Clauze privind atributele de domeniu al datelor (*Data Scope Attribute Clauses*)

O problemă importantă pentru programarea OpenMP este înțelegerea și utilizarea domeniului acoperit de date. Deoarece OpenMP se bazează pe modelul de programare cu memorie partajată, cele mai multe variabile sunt utilizate în comun (*shared*) prin default.

Clauzele privind atributele de domeniu ale datelor sunt utilizate pentru a defini explicit cum trebuie utilizate variabilele în domenii. În lista clauzelor regăsim:

- ***private***
- ***firstprivate***
- ***lastprivate***
- ***shared***
- ***default***
- ***reduction***
- ***copyin***

Clauzele privind atributele de domeniu al datelor sunt utilizate în combinație cu mai multe directive (***parallel***, ***DO/for*** și ***sections***) pentru a controla domeniul de utilizare a variabilelor incluse. Aceste clauze fac posibil controlul mediului de date în

timpul executării constructorilor paraleli. Ele definesc cum și care variabile din secțiunea secvențială a programului sunt transferate către secțiunile paralele ale programului și invers. Ele definesc care variabile vor fi vizibile tuturor firelor din secțiunile paralele și care variabile vor fi alocate privat de toate firele.

*Notă:* clauzele privind atributele de domeniu au efect numai în extinderea lor lexicală/statică.

### 2.1.1. Clauza **PRIVATE**

*Scop:* Clauza **private** declară variabile care sunt private pentru fiecare fir.

*Formatul în C/C++:*

**private (list)**

*Note:* Variabilele **private** se comportă după cum urmează:

Un obiect nou de același tip se declară o dată pentru fiecare fir din fascicul. Toate referirile la obiectul original sunt înlocuite cu referiri la obiectul nou. Variabilele declarate **private** sunt neinitializate pentru fiecare fir.

*Exemplu 2.1.1.* În acest exemplu se ilustrează modul de utilizare a variabilelor de tip **privat**.

```
#include <stdio.h>
#include <omp.h>
#include <iostream>
int main(int argc, char *argv[])
{
    int n=1,iam;
    printf("Valoarea lui n pana la
        directiva parallel cu clauza
        private: %d\n", n);
    #pragma omp parallel private(n,iam)
    {
        sleep(omp_get_thread_num());
        printf(" OpenMP procesul %d-
        valoarea lui n dupa clauza
```

```
private: %d\n",
    omp_get_thread_num(),n);
    n=omp_get_thread_num();
    printf(" OpenMP procesul %d-
    valoarea lui n dupa initializare
    de catre fir:
    %d\n",omp_get_thread_num(),
    n);
    }
    printf("Valoarea lui n dupa directiva
    parallel cu clauza private:
    %d\n", n);
}
```

Rezultatele vor fi urmatoarele:

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpiCC -fopenmp -o
Exemplu2.1.1.exe Exemplu2.1.1.cpp
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
compute-0-0,compute-0-1 Exemplu2.1.1.exe
Valoarea lui n pana la directiva parallel cu clauza private: 1
OpenMP procesul 0-valoarea lui n dupa clauza private: 0
OpenMP procesul 0-valoarea lui n dupa initializare de catre fir: 0
OpenMP procesul 1-valoarea lui n dupa clauza private: -1
OpenMP procesul 1-valoarea lui n dupa initializare de catre fir: 1
OpenMP procesul 2-valoarea lui n dupa clauza private: 0
OpenMP procesul 2-valoarea lui n dupa initializare de catre fir: 2
OpenMP procesul 3-valoarea lui n dupa clauza private: 0
OpenMP procesul 3-valoarea lui n dupa initializare de catre fir: 3
Valoarea lui n dupa directiva parallel cu clauza private: 1
[Hancu_B_S@hpc Open_MP]$
```

### 2.1.2. Clauza SHARED

*Scop:* Clauza SARED declară în lista ei variabile care sunt partajate între toate firele fascicolului.

*Formatul în C/C++:*

**shared (listă)**

*Note:* O variabilă partajată există numai într-o locație de memorie și toate firele pot citi sau scrie la acea adresă. Este în responsabilitatea programatorului a asigura că firele multiple au acces potrivit la variabilele **shared** (cum ar fi prin secțiunile **critical**).

*Exemplu 2.1.2. În acest exemplu se ilustrează modul de utilizare a variabilelor de tip **shared**.*

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    int i, m[5];
    printf("Vectorul m pana la directiva
        parallel shared:\n");
    /* Se initializeaza vectorul m */
```

```
    for (i=0; i<5; i++){
        m[i]=0;
        printf("%d\n", m[i]);
    }
    #pragma omp parallel shared(m)
    {
        /* Se atribuie valoarea 1
            elementului cu indicele egal cu
```

```

numarul      firului din
vectorul m */
m[omp_get_thread_num()]=1;
}

```

```

printf("Valoarea vectorului dupa
directiva parallel shared:\n");
for (i=0; i<5; i++) printf("%d\n", m[i]);
}

```

Rezultatele vor fi urmatoarele:

```

[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpiCC -fopenmp -o
Exemplu2.1.2.exe Exemplu2.1.2.cpp
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -machinefile
~/nodes Exemplu2.1.2.exe

```

Vectorul m pana la directiva parallel shared:

```

0
0
0
0
0
0

```

Valoarea vectorului dupa directiva parallel shared:

```

1
1
1
1
1
0

```

### 2.1.3. Clauza **DEFAULT**

*Scop:* Clauza **default** permite utilizatorului să specifice prin default un domeniu **private**, **shared** sau **none** pentru toate variabilele din extinderea lexicală a oricărei regiuni paralele.

*Formatul în C/C++:*

```
default (shared | none)
```

*Note:* Variabile specifice pot fi absolvite de default utilizând clauzele **private**, **shared**, **firstprivate**, **lastprivate** și **reduction**. Specificatia în C/C++ din OpenMP nu include “**private**” ca un default posibil. Totusi, unele implementări pot avea prevăzută această opțiune.

*Restricții:* numai clauza **default** poate fi specificată pe o directivă **parallel**.

### 2.1.4. Clauza **FIRSTPRIVATE**

*Scop:* Clauza **firstprivate** combină comportarea clauzei **private** cu inițializarea automată a variabilelor din lista ei.

*Formatul în C/C++:*

**firstprivate (listă)**

*Note:* Variabilele din listă sunt inițializate potrivit cu valorile obiectelor lor origine înainte de intrarea în construcția paralelă sau de lucru partajat.

**Exemplu 2.1.3.** În acest exemplu se ilustrează modul de utilizare a variabilelor de tip **firstprivate**.

<pre>#include &lt;stdio.h&gt; #include &lt;omp.h&gt; #include &lt;iostream&gt; int main(int argc, char *argv[]) {     int n=1;     printf("Valoarea lui n pana la            directiva parallel cu clauza            firstprivate: %d\n", n);     #pragma omp parallel firstprivate(n)     {         sleep(omp_get_thread_num());         printf(" OpenMP procesul %d-         valoarea lui n dupa clauza</pre>	<pre>firstprivate: %d\n",     omp_get_thread_num(),n);  n=omp_get_thread_num(); printf(" OpenMP procesul %d- valoarea lui n dupa initializare de catre fire : %d\n",     omp_get_thread_num(),n); } printf("Valoarea lui n dupa directiva parallel cu clauza firstprivate: %d\n", n); }</pre>
---	---

Rezultatele vor fi urmatoarele:

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o
Exemplu2.1.3.exe Exemplu2.1.3.cpp
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
compute-0-0,compute-0-1 Exemplu2.1.3.exe
Valoarea lui n pana la directiva parallel cu clauza firstprivate: 1
OpenMP procesul 0-valoarea lui n dupa clauza firstprivate: 1
OpenMP procesul 0-valoarea lui n dupa initializare de catre fire : 0
OpenMP procesul 1-valoarea lui n dupa clauza firstprivate: 1
OpenMP procesul 1-valoarea lui n dupa initializare de catre fire : 1
OpenMP procesul 2-valoarea lui n dupa clauza firstprivate: 1
OpenMP procesul 2-valoarea lui n dupa initializare de catre fire : 2
OpenMP procesul 3-valoarea lui n dupa clauza firstprivate: 1
OpenMP procesul 3-valoarea lui n dupa initializare de catre fire : 3
```

Valoarea lui `n` după directiva `parallel` cu clauza `firstprivate: 1`  
[Hancu\_B\_S@hpc Open\_MP]\$

### 2.1.5. Clauza **LASTPRIVATE**

*Scop:* Clauza **lastprivate** combină comportarea clauzei **private** cu copierea din ultima iterație din buclă sau secțiune în variabila obiect originară.

*Formatul în C/C++:*

**lastprivate (listă)**

*Note:* Valorile copiate înapoi în variabilele obiect origine se obțin din ultima iterație sau secțiune (secvențială) a constructului care o conține.

**Exemplu 2.1.4.** În acest exemplu se ilustrează modul de utilizare a variabilelor de tip **lastprivate**.

```
#include <stdio.h>
#include <omp.h>
#include <iostream>
int main(int argc, char *argv[])
{
    int n=0;
    printf("Valoarea n în zona
           secvențială a programului
           (înainte de constructorul
           paralel): %d\n", n);
    #pragma omp parallel
    {
        #pragma omp sections
        lastprivate(n)
        {
            sleep(omp_get_thread_num());
            #pragma omp section
            {
                printf("Valoarea n pentru firul %d (în
                       #pragma omp section-pana la
                       initializare):
                       %d\n",omp_get_thread_num(),
                       n);
            }
        }
    }
}
```

```
n=1;
printf("Valoarea n pentru firul %d (în
#pragma omp section-dupa
initializare): %d\n",
omp_get_thread_num(), n);
}
#pragma omp section
{
    printf("Valoarea n pentru firul %d (în
#pragma omp section- pana la
initializare):
%d\n",omp_get_thread_num(),
n);
n=2;
printf("Valoarea n pentru firul %d (în
#pragma omp section- dupa
initializare):
%d\n",omp_get_thread_num(),
n);
}
#pragma omp section
{
    printf("Valoarea n pentru firul %d (în
#pragma omp section-pana la
initializare):
%d\n",omp_get_thread_num(),
n);
}
```

<pre>printf("Valoarea n pentru firul %d (în #pragma omp section-pana la initializare): %d\n",omp_get_thread_num(), n); n=3; printf("Valoarea n pentru firul %d (în #pragma omp section-dupa initializare): %d\n",omp_get_thread_num(), n); }</pre>	<pre>} sleep(omp_get_thread_num()); printf("Valoarea n pentru firul %d: %d (dupa #pragma omp section)\n",omp_get_thread_nu m(), n); } printf("Valoarea n în zona secventiala a programului(dupa constructorul paralel): %d\n", n); }</pre>
--	--

Rezultatele vor fi urmatoarele:

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o
Exemplu2.1.4.exe Exemplu2.1.4.cpp
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
compute-0-0,compute-0-1 Exemplu2.1.3.exe
Valoarea n în zona secventiala a programului (inainte de constructorul
paralel): 0
Valoarea n pentru firul 1 (în #pragma omp section-pana la initializare): -1
Valoarea n pentru firul 1 (în #pragma omp section-dupa initializare): 1
Valoarea n pentru firul 3 (în #pragma omp section-pana la initializare): 0
Valoarea n pentru firul 3 (în #pragma omp section-dupa initializare): 3
Valoarea n pentru firul 0 (în #pragma omp section- pana la initializare): 0
Valoarea n pentru firul 0 (în #pragma omp section- dupa initializare): 2
Valoarea n pentru firul 0: 3 (dupa #pragma omp section)
Valoarea n pentru firul 1: 3 (dupa #pragma omp section)
Valoarea n pentru firul 2: 3 (dupa #pragma omp section)
Valoarea n pentru firul 3: 3 (dupa #pragma omp section)
Valoarea n în zona secventiala a programului(dupa constructorul paralel): 3
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
compute-0-0,compute-0-1 Exemplu2.1.3.exe
Valoarea n în zona secventiala a programului (inainte de constructorul
paralel): 0
Valoarea n pentru firul 3 (în #pragma omp section-pana la initializare): 0
Valoarea n pentru firul 3 (în #pragma omp section-dupa initializare): 1
Valoarea n pentru firul 0 (în #pragma omp section- pana la initializare): 0
Valoarea n pentru firul 0 (în #pragma omp section- dupa initializare): 2
Valoarea n pentru firul 3 (în #pragma omp section-pana la initializare): 1
Valoarea n pentru firul 3 (în #pragma omp section-dupa initializare): 3
Valoarea n pentru firul 0: 3 (dupa #pragma omp section)
```

Valoarea n pentru firul 1: 3 (dupa #pragma omp section)  
 Valoarea n pentru firul 2: 3 (dupa #pragma omp section)  
 Valoarea n pentru firul 3: 3 (dupa #pragma omp section)  
 Valoarea n în zona secventiala a programului(dupa constructorul paralel): 3  
 [Hancu\_B\_S@hpc Open\_MP]\$

Astfel, în regiune paralela (până la inițializare) valoarea lui **n** nu este determinată, la ieșirea din regiunea paralelă valoarea lui **n** este egală cu ultima valoare inițializată

### 2.1.6. Clauza **COPYIN**

*Scop:* Clauza **copyin** asigură un mijloc de a atribui aceeași valoare variabilelor **threadprivate** pentru toate firele unui fascicul.

*Formatul în C/C++:*

**copyin (listă)**

*Note:* Lista conține numele variabilelor de copiat. Variabilele firului master sunt sursa tuturor cópiilor. Firele fasciculului sunt inițializate cu valorile lor la intrarea în constructul paralel.

**Exemplu 2.1.5.** În acest exemplu se ilustrează modul de utilizare a variabilelor de tip **copyin**.

<pre>#include &lt;stdio.h&gt; #include &lt;omp.h&gt; #include &lt;iostream&gt; int n; #pragma omp threadprivate(n) int main(int argc, char *argv[]) {     n=1;     #pragma omp parallel copyin(n)     {         sleep(omp_get_thread_num());         printf("Valoarea n în prima regiune             paralela a firului %d: %d\n",             omp_get_thread_num(),n);     } }</pre>	<pre>printf("*****\n");     printf("Aici firul Master executa un         cod serial\n");      printf("*****\n");     n=2;     #pragma omp parallel copyin(n)     {         sleep(omp_get_thread_num());         printf("Valoarea n în a doua regiune             paralela a firului %d: %d\n",             omp_get_thread_num(),n);     } }</pre>
---	---



<pre>printf("*****\n"); printf("Aici firul Master executa un cod serial\n");  printf("*****\n"); #pragma omp parallel</pre>	<pre>{ sleep(omp_get_thread_num()); printf("Valoarea n în a treia regiune paralela a firului %d: %d\n", omp_get_thread_num(),n); }</pre>
---	--

Rezultatele vor fi urmatoarele:

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o
Exemplu2.1.5.exe Exemplu2.1.5.cpp
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
compute-0-0,compute-0-1 Exemplu2.1.4.exe
Valoarea n în prima regiune paralela a firului 0: 1
Valoarea n în prima regiune paralela a firului 1: 1
Valoarea n în prima regiune paralela a firului 2: 1
Valoarea n în prima regiune paralela a firului 3: 1
*****
Aici firul Master executa un cod serial
*****
Valoarea n în a doua regiune paralela a firului 0: 2
Valoarea n în a doua regiune paralela a firului 1: 2
Valoarea n în a doua regiune paralela a firului 2: 2
Valoarea n în a doua regiune paralela a firului 3: 2
*****
Aici firul Master executa un cod serial
*****
Valoarea n în a treia regiune paralela a firului 0: 2
Valoarea n în a treia regiune paralela a firului 1: 2
Valoarea n în a treia regiune paralela a firului 2: 2
Valoarea n în a treia regiune paralela a firului 3: 2
[Hancu_B_S@hpc Open_MP]$
```

### 2.1.6. Clauza **REDUCTION**

*Scop:* Clauza **reduction** execută o operație de reducere pe variabilele care apar în listă. Pentru fiecare fir se crează o copie privată pentru fiecare variabilă din listă. La sfârșitul operației de

reducere, variabila de reducere este aplicată tuturor copiiilor private ale variabilelor partajate și rezultatul final este scris în variabila globală folosită partajat.

*Formatul în C/C++:*

### **reduction (operator: listă)**

*Restrictii:* Variabilele din listă trebuie să fie variabile scalare cu nume. Ele nu pot fi masive sau variabile de tipul structurilor. Ele trebuie de asemenea să fie declarate **shared** în contextul care le include. Operațiile de reducere pot să nu fie asociative pentru numere reale.

**Exemplu 2.1.6** În acest exemplu se ilustrează modul de utilizare a clauzei **reduction**. Se determina produsul scalar a doi vectori. Iterațiile buclei paralele vor fi distribuite în blocuri fiecărui fir din fascicol. La finalul construcției buclei paralele toate firele vor aduna valorile “rezultatelor” lor pentru a actualiza copia globală din firul **master**.

```
##include <stdio.h>
#include <omp.h>
#include <iostream>
main ()
{
    int i, n, chunk,k;
    float a[100], b[100], result;
    /* Se initializeaza valorile */
    n = 100;
    chunk = 10;
    result = 0.0;
    for (i=0; i < n; i++)
    {
        a[i] = 1.0/i * 1.0;
        b[i] = 1.0/i * 2.0;
    }
    omp_set_num_threads(2);
    #pragma omp parallel
        default(shared) private(i,k)
        reduction(+:result)
    {
```

```
        k=0;
        #pragma omp for
        schedule(dynamic,chunk)
        nowait
        for (i=0; i < n; i++)
        {
            k=k+1;
            result = result + (a[i] * b[i]);
        }

        sleep(omp_get_thread_num
        ());
        printf("Procesul OpenMP cu
        numarul %d, a determinat %d
        elemente ale produsului scalar
        egal cu %f\n",
        omp_get_thread_num(),k,result
        );
    }
```

```
printf("Produsul scalar este=
    %f\n",result);
}
```

Rezultatele vor fi urmatoarele:

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

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o
    Exemplu2.1.6.exe Exemplu2.1.6.cpp
```

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
    compute-0-0,compute-0-1 Exemplu2.1.5.exe
```

Procesul OpenMP cu numarul 0, a determinat 50 elemente ale produsului scalar egal cu 50.000000

Procesul OpenMP cu numarul 1, a determinat 50 elemente ale produsului scalar egal cu 50.000000

Produsul scalar este= 100.000000

```
[Hancu_B_S@hpc Open_MP]$
```

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

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o
    Exemplu2.1.6.exe Exemplu2.1.6.cpp
```

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
    compute-0-0,compute-0-1 Exemplu2.1.6.exe
```

Procesul OpenMP cu numarul 0, a determinat 90 elemente ale produsului scalar egal cu 90.000000

Procesul OpenMP cu numarul 1, a determinat 10 elemente ale produsului scalar egal cu 10.000000

Produsul scalar este= 100.000000

```
[Hancu_B_S@hpc Open_MP]$
```

Operatiile de reducere pot fi numai de urmatoarea forma:

**C/C++**

```
x = x op expr
```

```
x = expr op x (except subtraction)
```

```
x binop = expr
```

```
x++
```

```
++x
```

```
x--
```

```
--x
```

unde:

**x** este o variabilă scalară din listă

**expr** este o expresie scalară care nu face referire la **x**

**op** nu este overloaded și este +, \*, -, /, &, ^, |, && sau ||

**binop** nu este overloaded și este +, \*, -,

## Un sumar al clauzelor/directivelor OpenMP

Clauze	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
<b>IF</b>	X				X	X
<b>PRIVATE</b>	X	X	X	X	X	X
<b>SHARED</b>	X	X			X	X
<b>DEFAULT</b>	X				X	X
<b>FIRSTPRIVATE</b>	X	X	X	X	X	X
<b>LASTPRIVATE</b>	X	X	X		X	X
<b>REDUCTION</b>	X	X	X		X	X
<b>COPYIN</b>	X				X	X
<b>SCHEDULE</b>		X			X	
<b>ORDERED</b>		X			X	
<b>NOWAIT</b>		X	X	X		

Următoarele directive OpenMP nu admit clauze:

**master**  
**critical**  
**barrier**  
**atomic**  
**flush**  
**ordered**  
**threadprivate**

Implementările pot diferi și diferă uneori de standard în ceea ce privește acceptarea clauzelor de către fiecare directivă.

**Exemplu 2.1.7.** Să se calculeze valoarea aproximativă a lui  $\pi$  prin integrare numerică cu formula  $\pi = \int_0^1 \frac{4}{1+x^2} dx$ , folosind formula dreptunghiurilor. Intervalul închis  $[0,1]$  se împarte într-un număr de  $n$  subintervale și se însumează ariile dreptunghiurilor având ca bază fiecare subinterval.

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele menționate în exemplul 2.1.7<sup>2</sup>.

---

<sup>2</sup> Semnificația variabilelor `MPIrank`, `Nodes` va fi explicată în capitolul 4.

```

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#ifdef _OPENMP
    #include <omp.h>
    #define TRUE 1
    #define FALSE 0
#else
    #define omp_get_thread_num() 0
#endif
double f(double y)
    {return(4.0/(1.0+y*y));}
int main()
{
    double w, x, sum, pi,a,b;
    int i,MPIrank;
    int n = 1000000;
    int Nodes=1;
    MPIrank=0;
    w = 1.0/n;
    sum = 0.0;
    a=(MPIrank+0.0)/Nodes;
    b=(MPIrank+1.0)/Nodes;
    omp_set_num_threads(2);
    #pragma omp parallel private(x)
        shared(w,a,b) reduction(+:sum)

```

```

{
    #pragma omp master
    {
        printf("Pentru fiecare proces MPI se
            genereaza %d procese
            OpenMP (fire)\n",
            omp_get_num_threads());
    }
    #pragma omp for nowait
    for(i=0; i < n; i++)
    {
        x = a+(b-a)*w*(i-0.5);
        sum = sum + f(x);
    }
    sleep(omp_get_thread_num());
    printf("Procesul OpenMP cu numarul
        %d, a determinat integrala
        egala cu %f\n",
        omp_get_thread_num(),(b-
        a)*w*sum);
}
pi = (b-a)*w*sum;
printf("Valoare finala , pi = %f\n", pi);
}

```

### Rezultatele executarii programului.

```

[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o
Exemplu2.1.7.exe Exemplu2.1.7.cpp

```

```

[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
compute-0-0,compute-0-1 Exemplu2.1.7.exe

```

Pentru fiecare proces MPI se genereaza 2 procese OpenMP (fire)

Procesul OpenMP cu numarul 0, a determinat integrala egala cu 1.854591

Procesul OpenMP cu numarul 1, a determinat integrala egala cu 1.287003

Valoare finala , pi = 3.141595

```

[Hancu_B_S@hpc Open_MP]$

```

### Capitolul 3. Rutinele de bibliotecă run-time (Run-Time Library Routines) și variabile de mediu (Environment Variables)

#### Obiective

- Să definească noțiunea de model de programare paralelă;
- Să cunoască criteriile de clasificare a sistemelor paralele de calcul;
- Să cunoască particularitățile de bază la elaborarea programelor paralele ale sistemelor de calcul cu memorie partajată, cu memorie distribuită și mixte;

#### 3.1 Privire generală:

Standardul OpenMP definește o interfață API pentru apeluri la bibliotecă care execută următoarea varietate de funcții:

- Află prin chestionare numărul de fire/procesoare, stabilește numărul de fire utilizate.
- Blocări de domeniu general prin rutine adecvate (semafoare).
- Rutine portabile pentru măsurarea timpului universal (wall clock time).
- Setarea funcțiilor de mediu de execuție: paralelism unul-în-altul, ajustarea dinamică a firelor.

Pentru C/C++ poate fi necesară specificarea fișierului de incluziune "omp.h".

Pentru funcțiile/rutinele de închidere (Lock):

- Variabilele de tip **lock** trebuie să fie accesate numai prin rutinele de închidere.
- Pentru C/C++, variabila de tip **lock** trebuie să aibă tipul **omp\_lock\_t** sau tipul **omp\_nest\_lock\_t**, în funcție de modul de utilizare.

Note de implementare:

- Implementarea poate să suporte sau poate să nu suporte paralelismul unul-în-altul și/sau firele dinamice. Dacă paralelismul unul-în-altul este suportat, este adesea numai

nominal prin aceea că o regiune paralela una-în-alta poate avea numai un fir.

- Documentația implementării trebuie consultată pentru aceste detalii –sau se pot experimenta unele aspecte pentru a afla ceea ce nu este scris explicit în documentație.

## ***3.2 Rutine utilizate pentru setarea și returnarea numărului de fire***

### ***3.2.1 OMP\_SET\_NUM\_THREADS***

*Scop:* Setează numărul de fire care vor fi utilizate în regiunea paralela. Trebuie să fie un întreg pozitiv.

*Formatul în C/C++:*

```
#include <omp.h>
void omp_set_num_threads(int num_threads)
```

*Note și restricții:* Mecanismul firelor dinamice modifică efectul acestei rutine.

*Enabled:* specifică numărul maxim de fire care pot fi utilizate pentru orice regiune paralelă prin mecanismul firelor dinamice.

*Disabled:* specifică numărul exact de fire de utilizat până la apelul următor la această rutină.

Această rutină poate fi apelată numai din porțiunile secvențiale ale codului.

Acest apel are precedență față de variabila de mediu **OMP\_NUM\_THREADS**.

### ***3.2.2 OMP\_GET\_NUM\_THREADS***

*Scop:* returnează numărul de fire care sunt în fasciculul curent și execută regiunea paralelă din care este apelată.

*Format în C/C++:*

```
#include <omp.h>
int omp_get_num_threads(void)
```

*Note și restricții:* Dacă acest apel este făcut dintr-o porțiune secvențială a programului sau dintr-o regiune paralelă una-în-alta



care este serializată, returnul este 1. Numărul prin default al firelor este dependent de implementare.

**Exemplu 3.2.1.** În acest exemplu se ilustrează modul de utilizare a rutinelor

a) **omp\_get\_num\_threads()** -returnează numărul de fire care sunt în fasciculul curent

b) **omp\_set\_num\_threads()** -setează numărul de fire care vor fi utilizate în regiunea paralela.

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#ifdef _OPENMP
    #include <omp.h>
    #define TRUE 1
    #define FALSE 0
#else
    #define omp_get_thread_num() 0
    #define omp_get_num_threads() 1
#endif
int main()
{
    int TID;
#ifdef _OPENMP
    (void) omp_set_dynamic(1);
    if (omp_get_dynamic())
        {printf("Warning: dynamic
        adjustment of threads has been
        set\n");}
    // (void) omp_set_num_threads(4);
#endif
    for (int n=5; n<11; n+=5)
    {
        #pragma omp parallel if (n > 5)
            num_threads(n) default(none) \
            private(TID) shared(n)
        {
            TID = omp_get_thread_num();
        }
        #pragma omp single
        {
            printf("Value of n = %d\n",n);
            printf("Number of threads în
            parallel region: %d\n",
            omp_get_num_threads());
        }
        sleep(omp_get_thread_num());
        printf("Print statement executed
        by thread %d\n",TID);
    } /*-- End of parallel region --*/
    return(0);
}
```

Rezultatele vor fi următoarele:

a) Cazul cand (void) omp\_set\_dynamic(0)- desabilitează ajustarea dinamică (de sistemul de execuție) a numărului de fire disponibile pentru executarea regiunilor paralele.

[Hancu\_B\_S@hpc Open\_MP]\$ /opt/openmpi/bin/mpiCC -fopenmp -o Exemplu3.2.1.exe Exemplu3.2.1.cpp

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
compute-0-0,compute-0-1 Exemplu3.2.1.exe
Value of n = 5
Number of threads în parallel region: 1
Print statement executed by thread 0
Value of n = 10
Number of threads în parallel region: 10
Print statement executed by thread 0
Print statement executed by thread 1
Print statement executed by thread 2
Print statement executed by thread 3
Print statement executed by thread 4
Print statement executed by thread 5
Print statement executed by thread 6
Print statement executed by thread 7
Print statement executed by thread 8
Print statement executed by thread 9
[Hancu_B_S@hpc Open_MP]$
```

- b) Cazul cand (void) `omp_set_dynamic(1)`- abilitază ajustarea dinamică (de sistemul de execuție) a numărului de fire disponibile pentru executarea regiunilor paralele.

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpiCC -fopenmp -o
Exemplu3.2.1.exe Exemplu3.2.1.cpp
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
compute-0-0,compute-0-1 Exemplu3.2.1.exe
Warning: dynamic adjustment of threads has been set
Value of n = 5
Number of threads în parallel region: 1
Print statement executed by thread 0
Value of n = 10
Number of threads în parallel region: 4
Print statement executed by thread 0
Print statement executed by thread 1
Print statement executed by thread 2
Print statement executed by thread 3
[Hancu_B_S@hpc Open_MP]$
```

### ***3.2.3 OMP\_GET\_MAX\_THREADS***

*Scop:* returnează valoarea maximă care poate fi returnată de un apel la funcția **`omp_get_num_threads`**.

*Format în C/C++:*

```
#include <omp.h>
int omp_get_max_threads(void)
```

*Note și restricții:* Reflectă în general numărul de fire setat de variabila de mediu OMP\_NUM\_THREADS sau de rutina omp\_set\_num\_threads() din bibliotecă. Poate fi apelată atât din regiunile seriale ale codului cât și din cele paralele.

### 3.2.4 OMP\_GET\_NUM\_PROCS

*Scop:* returnează numărul de procesoare care sunt la dispoziția programului.

*Format în C/C++:*

```
#include <omp.h>
int omp_get_num_procs(void)
```

## 3.3 Rutina utilizată pentru returnarea identificatorului firului

### 3.3.1 OMP\_GET\_THREAD\_NUM

*Scop:* returnează numărul de fir al firului, în interiorul fasciculului, prin acest apel. Numărul acesta va fi între 0 și omp\_get\_num\_threads-1. Firul master din fascicul este firul 0.

*Format în C/C++:*

```
#include <omp.h>
int omp_get_thread_num(void)
```

*Note și restricții:* dacă este apelată dintr-o regiune paralel una-în-alta, funcția aceasta returnează 0.

*Exemple de determinare a numărului de fire dintr-o regiune paralelă:*

Exemplul 1 este modul corect de a determina identificatorul de fir într-o regiune paralelă.

Exemplul 2 este incorect – variabila **TID** trebuie să fie **private**.

Exemplul 3 este incorect – apelul **omp\_get\_thread\_num** este în afara unei regiuni paralele.

### *Exemplul 1:*

<pre>#include &lt;stdio.h&gt; #include &lt;omp.h&gt; int main(int argc, char *argv[]) {     int TID;     #pragma omp parallel private(TID)</pre>	<pre>{     TID=omp_get_thread_num();     printf("Hello from thread number            %d\n", TID); }</pre>
--	---

### *Exemplul 2*

<pre>#include &lt;stdio.h&gt; #include &lt;omp.h&gt; int main(int argc, char *argv[]) {     int TID;     #pragma omp parallel</pre>	<pre>{     TID=omp_get_thread_num();     printf("Hello from thread number            %d\n", TID); }</pre>
---	---

### *Exemplul 3:*

<pre>#include &lt;stdio.h&gt; #include &lt;omp.h&gt; int main(int argc, char *argv[]) {     int TID;     TID=omp_get_thread_num();</pre>	<pre>printf("Hello from thread number        %d\n", TID); #pragma omp parallel { } }</pre>
--	--

## ***3.4 Rutinele utilizate pentru generarea dinamica a firelor***

### ***3.4.1 OMP\_IN\_PARALLEL***

*Scop:* poate fi apelată pentru a determina dacă secțiunea de cod în execuție este paralelă sau nu.

*Formatul în C/C++:*

```
#include <omp.h>
int omp_in_parallel(void)
```

*Note și restricții:* În C/C++ ea returnează un întreg nenul dacă apelul este dintr-o zonă paralela, zero în caz contrar .

**Exemplu 3.4.1.** În acest exemplu se ilustrează modul de utilizare a rutinei **omp\_in\_parallel**

```
#include <stdio.h>
#include <omp.h>
void mode(void)
{
    if(omp_in_parallel()) printf("Se
        executa instructiuni din
        regiunea paralela\n");
    else printf("Se executa instructiuni
        din regiunea segventiala\n");
}
int main(int argc, char *argv[])
{
    mode();
    #pragma omp parallel
    {
        #pragma omp master
        {
            mode();
        }
    }
}
```

Rezultatele vor fi urmatoarele:

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o
Exemplu3.4.1.exe Exemplu3.4.1.cpp
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
compute-0-0,compute-0-1 Exemplu3.4.1.exe
Se executa instructiuni din regiunea segventiala
Se executa instructiuni din regiunea paralela
[Hancu_B_S@hpc Open_MP]$
```

### 3.4.2 OMP\_SET\_DYNAMIC

*Scop:* abilitază sau desabilitază ajustarea dinamică (de sistemul de execuție) a numărului de fire disponibile pentru executarea regiunilor paralele.

*Formatul în C/C++:*

```
#include <omp.h>
void omp_set_dynamic(int dynamic_threads)
```

*Note și restricții:* Pentru C/C++, dacă argumentul **dynamic\_threads** este nenul, atunci mecanismul este abilitat, altminteri este desabilitat. Subrutina **omp\_set\_dynamic** are întâietate față de variabila de mediu **OMP\_DYNAMIC**. Setarea prin

default depinde de implementare. Poate fi apelată dintr-o secțiune serială/secvențială a programului.

### 3.4.3 OMP\_GET\_DYNAMIC

*Scop:* determinarea stării abilitat/desabilitat a modificării dinamice a firelor.

*Formatul în C/C++:*

```
#include <omp.h>
int omp_get_dynamic(void)
```

*Note și restricții:* Pentru C/C++, rezultatul apelului este non-zero/zero pentru cele două situații.

**Exemplu 3.4.2.** Aceste exemplu ilustrează modalitatea de utilizare a rutinelor `omp_set_dynamic()` și `omp_get_dynamic()`

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    printf("Valoarea initiala
    (prestabilita) a variabilei de
    mediu OMP_DYNAMIC: %d\n",
    omp_get_dynamic());
    omp_set_dynamic(0);
    printf("Valoarea variabilei de
    mediu dupa executarea rutinei
    omp_set_dynamic()");
}
```

```
OMP_DYNAMIC : %d\n",
omp_get_dynamic());
#pragma omp parallel
num_threads(128)
{
#pragma omp master
{
    printf("Regiunea paralela
    contine, %d fire\n",
    omp_get_num_threads());
}
}
}
```

Rezultatele vor fi următoarele:

a) Cazul `omp_set_dynamic(1)`:

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o
Exemplu3.4.2.exe Exemplu3.4.2.cpp
```

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
compute-0-0,compute-0-1 Exemplu3.4.2.exe
```

```
Valoarea initiala (prestabilita) a variabilei de mediu OMP_DYNAMIC: 0
Valoarea variabilei de mediu dupa executarea rutinei omp_set_dynamic()
OMP_DYNAMIC : 1
```

Regiunea paralela contine, 4 fire

a) Cazul `omp_set_dynamic(0)`:

```
[Hancu_B_S@hpc Open_MP]$
```

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o  
Exemplu3.4.2.exe Exemplu3.4.2.cpp
```

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host  
compute-0-0,compute-0-1 Exemplu3.4.2.exe
```

Valoarea initiala (prestabilita) a variabilei de mediu `OMP_DYNAMIC`: 0

Valoarea variabilei de mediu dupa executarea rutinei `omp_set_dynamic()`

```
OMP_DYNAMIC : 0
```

Regiunea paralela contine, 128 fire

```
[Hancu_B_S@hpc Open_MP]$
```

### ***3.5 Rutinele utilizate pentru generarea paralelizmului unul-în-altul (nested parallelism)***

#### ***3.5.1 OMP\_SET\_NESTED***

*Scop:* abilitarea sau desabilitarea paralelizmului unul-în-altul.

*Formatul în C/C++:*

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

```
void omp_set_nested(int nested)
```

*Note și restrictii:* Pentru C/C++, dacă variabila **nested** este nenulă, atunci paralelizmul unul-în-altul este abilitat; dacă este nulă îl desabilitează. Defaultul este cu paralelizmul unul-în-altul desabilitat. Apelul este mai tare ca precedentă față de variabila de mediu **OMP\_NESTED**.

#### ***3.5.2 OMP\_GET\_NESTED***

*Scop:* determină dacă paralelizmul unul-în-altul este abilitat sau nu.

*Formatul în C/C++:*

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

```
int omp_get_nested (void)
```

*Note și restrictii:* În C/C++, se returnează o valoare nenulă dacă paralelizmul unul-în-altul este abilitat și zero în caz contrar.

*Exemplu 3.5.1. Aceste exemplu ilustreaza modalitatea de  
utilizare a rutinelor **omp\_get\_nested()** și  
**omp\_set\_nested()**.*

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#ifdef OPENMP
#include <omp.h>
#define TRUE 1
#define FALSE 0
#else
#define omp_get_thread_num() 0
#define omp_get_num_threads() 1
#define omp_get_nested() 0
#endif
int main()
{
int id;
#ifdef OPENMP
(void) omp_set_dynamic(FALSE);
if (omp_get_dynamic())
{printf("Warning: dynamic
adjustment of threads has been
set\n");}
(void) omp_set_num_threads(3);
(void) omp_set_nested(1);
}
else
{
if (! omp_get_nested())
{printf("Warning: nested
parallelism not set\n");}
#endif
printf("Nested parallelism is %s\n",
omp_get_nested() ?
"supported" : "not supported");
#pragma omp parallel private(id)
{
id=omp_get_thread_num();
#pragma omp parallel
num_threads(2)
{
sleep(id);
printf(" Thread %d from outer
parallel region executes the
thread %d from inner parallel
region\n",id,omp_get_thread_n
um());
} //End of inner parallel region
} // End of outer parallel region -
return(0);
}
```

Rezultatele vor fi urmatoarele:

a) Cazul (void) **omp\_set\_nested(1)**:

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o
Exemplu3.5.1.exe Exemplu3.5.1.cpp
```

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
compute-0-0,compute-0-1 Exemplu3.5.1.exe
```

Nested parallelism is supported

Thread 0 from outer parallel region executes the thread 0 from inner parallel region

Thread 0 from outer parallel region executes the thread 1 from inner parallel region

Thread 1 from outer parallel region executes the thread 1 from inner parallel region

Thread 1 from outer parallel region executes the thread 0 from inner parallel region

Thread 2 from outer parallel region executes the thread 0 from inner parallel region

Thread 2 from outer parallel region executes the thread 1 from inner parallel region

```
[Hancu_B_S@hpc Open_MP]$
```



## 2) Cazul (void) omp\_set\_nested(0):

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpiCC -fopenmp -o  
Exemplu3.5.1.exe Exemplu3.5.1.cpp
```

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host  
compute-0-0,compute-0-1 Exemplu3.5.1.exe
```

Warning: nested parallelism not set

Thread 0 from outer parallel region executes the thread 0 from inner  
parallel region

Thread 1 from outer parallel region executes the thread 0 from inner  
parallel region

Thread 2 from outer parallel region executes the thread 0 from inner  
parallel region

```
[Hancu_B_S@hpc Open_MP]$
```

Astfel, fiecare regiune paralela “exterioara” din numarul total de 3 fire “genereaza” la randul sau 2 fire care executa regiuni paralele “interioare”.

## 3.6 Rutinele utilizate pentru blocări de domeniu a firelor

### 3.6.1 OMP\_INIT\_LOCK

*Scopul:* subrutina initializează un lacăt asociat cu variabila de tip **lock**.

*Formatul în C/C++:*

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

```
void omp_init_lock(omp_lock_t *lock)
```

```
void omp_init_nest_lock(omp_nest_lock_t *lock)
```

*Note și restricții:* Starea inițială este unlocked.

### 3.6.2 OMP\_DESTROY\_LOCK

*Scop:* această subrutină disociază o variabilă lacăt de orice lacăt.

*Formatul în C/C++:*

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

```
void omp_destroy_lock(omp_lock_t *lock)
```

```
void omp_destroy_nest_lock(omp_nest_lock_t  
*lock)
```

*Note și restricții:* Este nepermis a apela această subrutină cu o variabilă lock care nu este inițializată.

### 3.6.3 OMP\_SET\_LOCK

*Scop:* această subrutină obligă firul executant să aștepte până când un lacăt specificat este disponibil. Un fir este proprietar al unui lock când acesta devine disponibil.

*Formatul în C/C++:*

```
#include <omp.h>
void omp_set_lock(omp_lock_t *lock)
void omp_set_nest_lock(omp_nest_lock_t *lock)
```

*Note și restricții:* Este nepermis a apela această rutină cu o variabilă lock neinițializată.

### 3.6.4 OMP\_UNSET\_LOCK

*Scop:* această subrutină eliberează(descurie) un **lock** dintr-o subrutină în execuție.

*Formatul în C/C++:*

```
#include <omp.h>
void omp_unset_lock(omp_lock_t *lock)
void omp_unset_nest_lock(omp_nest_lock_t
    *lock)
```

*Note și restricții:* Nu este permis a se apela această subrutină cu o variabilă lock e inițializată.

**Exemplu 3.6.1.** Aceste exemplu ilustrează modalitatea de utilizare a rutinelor **omp\_unset\_lock ()**, **omp\_set\_lock ()**, **omp\_destroy\_lock ()**, **omp\_init\_lock**.

<pre>#include &lt;stdio.h&gt; #include &lt;omp.h&gt; #include &lt;iostream&gt; omp_lock_t lock, lock1; int main(int argc, char *argv[]) {     int n;     omp_init_lock(&amp;lock);     omp_init_lock(&amp;lock1);</pre>	<pre>    omp_set_num_threads(3);     #pragma omp parallel private (n)     {         omp_set_lock(&amp;lock1);         #pragma omp master         {             printf("Se generează %d procese                 OpenMP (fire)\n",                 omp_get_num_threads());</pre>
---	--

<pre> } omp_unset_lock(&amp;lock1); n=omp_get_thread_num(); omp_set_lock(&amp;lock); printf("=====\n"); printf("Inceputul sectiei inchise, firul %d\n", n); printf("Sfarsitul sectiei inchise, firul %d\n", n); </pre>	<pre> printf("=====\n");     printf("\n");     omp_unset_lock(&amp;lock); } omp_destroy_lock(&amp;lock1); omp_destroy_lock(&amp;lock); } </pre>
--	---

Rezultatele vor fi urmatoarele:

```

[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpiCC -fopenmp -o
Exemplu3.6.2.exe Exemplu3.6.2.cpp
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
compute-0-0,compute-0-1 Exemplu3.6.2.exe
Se genereaza 3 procese OpenMP (fire)
=====
Inceputul sectiei inchise, firul 0
Sfarsitul sectiei inchise, firul 0
=====

=====
Inceputul sectiei inchise, firul 2
Sfarsitul sectiei inchise, firul 2
=====

=====
Inceputul sectiei inchise, firul 1
Sfarsitul sectiei inchise, firul 1
=====

[Hancu_B_S@hpc Open_MP]$

```

Astfel utilizand aceste rutine se poate “sincroniza,, procesul de scoatere la tipar: un fir începe să tipărească numai după ce un alt fir a terminat să tipărească.

### 3.6.5 OMP\_TEST\_LOCK

*Scop:* această subrutină încearcă a seta un **lock** dar nu blochează dacă lacătul este indisponibil.

*Formatul în C/C++:*

```
#include <omp.h>
int omp_test_lock(omp_lock_t *lock)
int omp_test_nest_lock(omp_nest_lock_t *lock)
```

*Note și restricții:* Pentru C/C++ este restituită o valoare nenulă dacă lacătul a fost setat cu succes, altminteri este restituit zero. Este interzis a apela această subrutină cu o variabilă lock neinitializată.

### ***3.7 Rutine portabile pentru măsurarea timpului universal (wall clock time)***

#### ***3.7.1 OMP\_GET\_WTIME***

*Scop:* furnizează o rutină portabilă de temporizare în timp universal (wall clock). Returnează o valoare în dublă precizie egală cu numărul de secunde scurse de la un anumit punct în trecut. Uzual, este folosită în pereche cu valoarea de la primul apel scăzută din valoarea apelului al doilea pentru a obține timpul scurs pentru un bloc de cod. Proiectată a da timpii “pe fir” și de aceea nu poate fi consistentă global pe toate firele unui fascicul; depinde de ceea ce face un fir comparativ cu alte fire.

*Formatul în C/C++:*

```
#include <omp.h>
double omp_get_wtime(void)
```

*Note și restricții:* Necesită suportul versiunii OpenMP 2.0.

#### ***3.7.2 OMP\_GET\_WTICK***

*Scop:* furnizează o rutină portabilă de temporizare în timp universal (wall clock). Returnează o valoare în dublă precizie egală cu numărul de secunde între două tic-uri succesive ale ceasului.

*Formatul în C/C++:*

```
#include <omp.h>
double omp_get_wtick(void)
```

*Note și restricții:* Necesită suportul versiunii OpenMP 2.0. Variabile de mediu (de programare) OpenMP furnizează patru variabile de mediu pentru controlul executiei unui cod paralel. Toate

numele variabilelor de mediu sunt scrise cu majuscule. Valorile atribuite lor nu sunt sensibile la upper/lower case.

### ***3.8 Variable de mediu (de programare)***

OpenMP furnizează patru variabile de mediu pentru controlul execuției unui cod paralel. Toate numele variabilelor de mediu sunt scrise cu majuscule. Valorile atribuite lor nu sunt sensibile la upper/lower case.

#### ***3.8.1 OMP\_SCHEDULE***

Se aplică numai la directivele `for`, `parallel for` (C/C++) care au clauzele lor `schedule` setată în timpul execuției (runtime). Valoarea acestei variabile determină modul cum sunt planificate iteratiile buclei pe procesoare. De exemplu:

```
setenv OMP_SCHEDULE "guided, 4"  
setenv OMP_SCHEDULE "dynamic"
```

#### ***3.8.2 OMP\_NUM\_THREADS***

Fixează numărul maxim de fire utilizate în timpul execuției. De exemplu:

```
setenv OMP_NUM_THREADS 8
```

#### ***3.8.3 OMP\_DYNAMIC***

Abilitează sau desabilitează ajustarea dinamică a numărului de fire disponibile pentru executarea unei regiuni paralel. Valorile valide sunt `TRUE` sau `FALSE`. De exemplu:

```
setenv OMP_DYNAMIC TRUE
```

#### ***3.8.4 OMP\_NESTED***

Abilitează sau desabilitează paralelismul unul-în-altul. Valorile valide sunt `TRUE` sau `FALSE`. De exemplu:

```
setenv OMP_NESTED TRUE
```

*Note de implementare:* O implementare sau alta poate să permită sau nu paralelismul unul-în-altul si/sau firele dinamice. Dacă paralelismul unul-în-altul este suportat, el este adesea numai nominal, în aceea că o regiune paralel una-în-alta poate avea numai un fir. Pentru detalii trebuie consultată documentatia implementării utilizate sau se poate experimenta pentru a afla ceea ce nu se poate găsi în documentatie.

## Capitolul 4. Aspecte comparative ale modelelor de programare paralela MPI și OpenMP

### Obiective

- Să definească noțiunea de model de programare paralelă;
- Să cunoască criteriile de clasificare a sistemelor paralele de calcul;
- Să cunoască particularitățile de bază la elaborarea programelor paralele ale sistemelor de calcul cu memorie partajată, cu memorie distribuită și mixte;

### 4.1 Preliminarii

#### *Message Passing Interface (MPI)*

MPI este o specificație de bibliotecă pentru message-passing (mesaje-trecere), propus ca standard de către un comitet bazat în mare parte de furnizori, implementatori și utilizatori..

#### *Open Multi Processing (OpenMP)*

OpenMP este o specificație pentru un set de directive compilator, rutine de bibliotecă, și variabile de mediu, care pot fi folosite pentru a specifica paralelismul de memorie partajată în programele Fortran și C/C++.

MPI vs. OpenMP	
<b>MPI</b> Modelul de memorie distribuită pe rețea distribuită Bazat pe mesaje Flexibil și expresiv	<b>OpenMP</b> Modelul de memorie partajată pe procesoare multi-core Bazat pe directivă Mai ușor de programat și debug

Exemplificăm cele spuse mai sus.

Un program serial

```
#include<stdio.h>
#define PID 0
main () {
```

```
int i;
printf("Greetings from process %d!\n",
PID);
}
```

Rezultatele:

```
$/opt/openmpi/bin/mpiCC -o TT1.exe TT1.cpp
$/opt/openmpi/bin/mpirun -n 4 -machinefile
~/nodes TT1.exe
Greetings from process 0!
Greetings from process 0!
Greetings from process 0!
Greetings from process 0!
```

Programul paralel utilizând MPI

```
#include<stdio.h>
#include <stdlib.h>
#include<mpi.h>
int main(int argc,char *argv[])
{
int my_rank;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
// Parallel Region
if ( my_rank != 0)
printf("Greetings from process %d!\n",
my_rank);
MPI_Finalize();
}
```

Rezultatele:

```
$ /opt/openmpi/bin/mpiCC -o TT2.exe TT2.cpp
$ /opt/openmpi/bin/mpirun -n 4 -machinefile
~/nodes TT2.exe
Greetings from process 3!
Greetings from process 2!
Greetings from process 1!
```



Programul paralel utilizând OpenMP

```
##include<stdio.h>
#include<omp.h>
main()
{
    int id;
    #pragma omp parallel
    {
        id = omp_get_thread_num();
        printf("Greetings from process %d!\n",
id);
    }
}
```

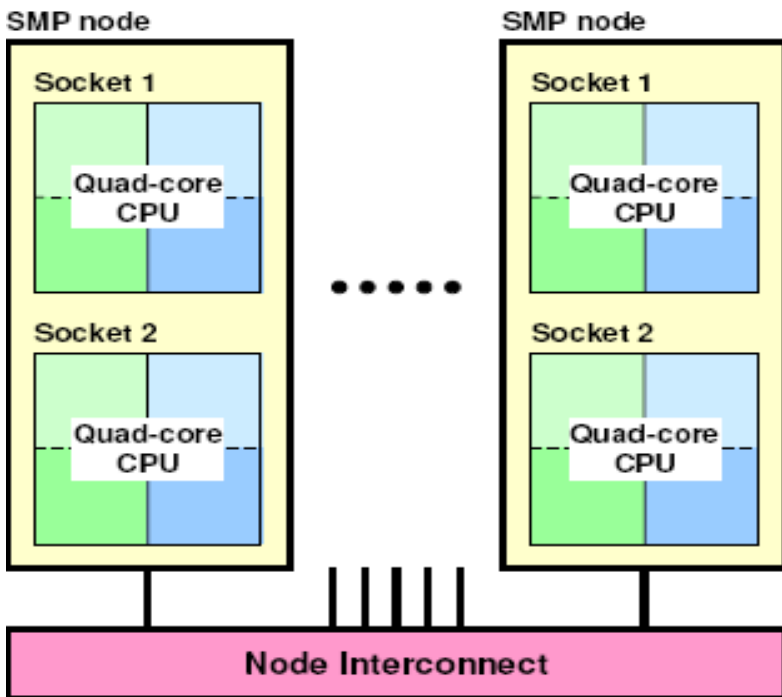
Rezultatele:

```
$/opt/openmpi/bin/mpicc -fopenmp -o
TT3.exe TT3.cpp
$/opt/openmpi/bin/mpirun -n 1 -machinefile
~/nodes TT3.exe
Greetings from process 2!
Greetings from process 3!
Greetings from process 1!
Greetings from process 0!
```

## 4.2 Programare paralelă mixtă MPI și OpenMP

Sistemele *Cluster SMP* (Symmetric Multi-Processor) pot fi descrise ca un hibrid de sisteme cu memorie partajată și sisteme cu memorie distribuită. Clusterul este format dintr-un număr de noduri SMP, fiecare conținând un număr de procesoare partajînd un spațiu de memorie globală.

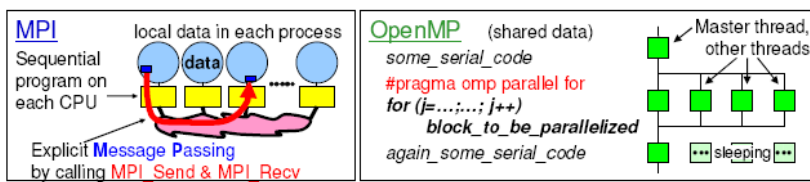
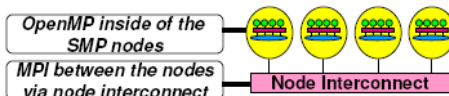
Sistemele de calcul paralel mixte (cu memorie distribuită și cu memorie partajată) pot fi reprezentate astfel:



Modelele de programare paralelă sunt prezentate în următorul desen:

## Major Programming models on hybrid systems

- Pure MPI (one MPI process on each CPU)
- Hybrid MPI+OpenMP
  - shared memory OpenMP
  - distributed memory MPI
- Other: Virtual shared memory systems, PGAS, HPF, ...
- Often **hybrid programming (MPI+OpenMP)** slower than **pure MPI**
  - why?



Deci, pentru modelele de programare paralela mixtă (MPI-OpenMP) se evidențiază (apare) următoarea problemă: realizarea comunicării folosind funcțiile MPI de transmitere/recepționare a mesajelor prin intermediul firelor de execuție. Adică relațiile “apel al funcțiilor MPI și fire de execuție”. Mai jos vom prezenta fragmente de cod care realizează această “interacțiune”.

O abordare populară pentru sistemele HPC cu mai multe nuclee este de a utiliza în același program atât MPI cât și OpenMP. De exemplu, un program MPI tradițional care rulează pe două servere 8-core (fiecare server conține două procesoare quad-core) este prezentată în figura de mai jos (Figura. 5). În total vor fi 16 procese independente pentru acest job MPI.

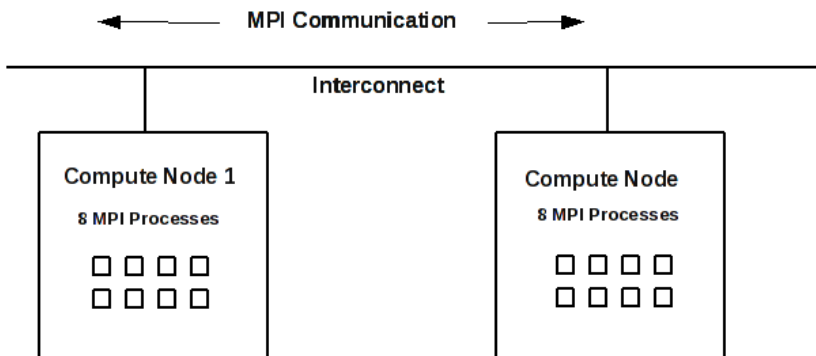


Figura 5: Executarea programului MPI pe 2 noduri (16 procese MPI)

Dacă s-ar utiliza atât modelul de programare MPI cât și modelul de programare OpenMP, atunci modalitatea prezentată în Figura 6 ar fi cea mai bună pentru a realiza un program mixt MPI-OpenMP. După cum se arată în figura 6, fiecare nod *execută doar un singur proces MPI*, care generează apoi *opt fire OpenMP*. Clusterelor care rulează benchmark-ul Top500 (HPL) folosesc acest tip de abordare, dar divizarea pe fire se face la un nivel inferior în biblioteca BLAS (Basic Linear Algebra Subroutines).

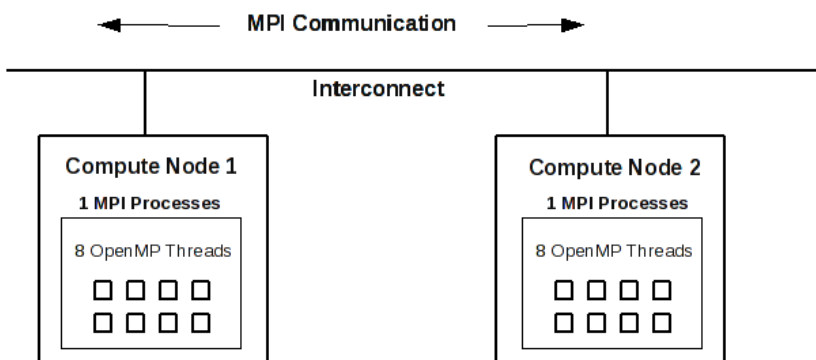


Figura 6: Executarea unui program MPI-OpenMP pe 2 noduri (2 procese MPI, fiecare cu 8 fire OpenMP)

Vom prezenta urmatorul exemplu de program în care se utilizează un model de programare paralelă mixt MPI-OpenMP.

**Exemplu 4.2.1.** Acest exemplu ilustrează modalitatea de utilizare a funcțiilor MPI și a directivelor (rutinelor) OpenMP pentru elaborarea programelor mixte MPI-OpenMP. În programul de mai jos se generează diferite regiuni paralele de tip fire de execuție în dependență de numele nodului pe care se execută aplicația. Programul se execută pe nodurile "compute-0-0, compute-0-1".

```
#include <stdio.h>
#include "mpi.h"
//=====
#ifdef OPENMP
#include <omp.h>
#define TRUE 1
#define FALSE 0
#else
#define omp_get_thread_num() 0
#endif
int main(int argc, char *argv[])
{
//=====
#ifdef OPENMP
(void)
omp_set_dynamic(FALSE);
if (omp_get_dynamic())
{printf("Warning: dynamic
adjustment of threads has
been set\n");}
(void) omp_set_num_threads(4);//
se fixeaza numarul de fire
pentru fiecare procesor fizic
#endif
//=====
int numprocs, realnumprocs,rank,
namelen,mpisupport;
char
processor_name[MPI_MAX_P
ROCESSOR_NAME];
int iam = 0, np = 1;
omp_lock_t lock;
omp_init_lock(&lock);
MPI_Init(&argc, &argv);
MPI_Get_processor_name(proces
sor_name, &namelen);
MPI_Comm_size(MPI_COMM_WO
RLD, &numprocs);
```

```
MPI_Comm_rank(MPI_COMM_W
ORLD, &rank);
if
(strncmp(processor_name,"co
mpute-0-0.local")==0)
{
omp_set_num_threads(4);
#pragma omp parallel
default(shared) private(iam,
np,realnumprocs)
{ //begin parallel construct
np = omp_get_num_threads();
//returneaza numarul total de fire
realnumprocs=
omp_get_num_procs();
//returneaza numarul de procesoare
disponibile
iam = omp_get_thread_num();
//returneaza 'eticheta" firului
omp_set_lock(&lock);
#pragma omp master
{
printf("\n");
printf(" ===Procesul MPI cu
rankul %d al nodului cu
numele '%s' a executat %d fire
=== \
\n",rank,processor_name,omp_get
_num_threads());
}
//#pragma omp barrier
printf("Hello from thread number
%d,total number of threads are
%d, MPI process rank is %d,
real number of processors is
%d on node %s\n", iam, np,
rank, realnumprocs,
processor_name);
omp_unset_lock(&lock);
```

```

//end parallel construct
omp_destroy_lock(&lock);
MPI_Barrier(MPI_COMM_WORLD
);
}
else
{
if (strcmp(processor_name,
"compute-0-1.local")==0)
{
omp_set_num_threads(2);
#pragma omp parallel
default(shared) private(iam,
np,realnumprocs)
{ //begin parallel construct
np = omp_get_num_threads();
//returneaza numarul total de fire
realnumprocs =
omp_get_num_procs();
//returneaza numarul de procesoare
disponibile
iam = omp_get_thread_num();
//returneaza 'eticheta' firului
omp_set_lock(&lock);

#pragma omp master
{
printf("\n");

```

```

printf(" ===Procesul MPI
cu rankul %d al nodului cu
numele '%s' a executat %d fire
=== \n",rank, processor_
name,
omp_get_num_threads());
}
omp_unset_lock(&lock);
printf("Hello from thread
number %d,total number of
thead's are %d, MPI process
rank is %d, real number"
"processors is %d on
node %s\n", iam, np, rank,
realnumprocs,
processor_name);
omp_unset_lock(&lock);
} //end
parallel construct
}
omp_destroy_lock(&lock);
MPI_Barrier(MPI_COMM_WORLD
);
}
MPI_Finalize();
return 0;
}

```

Rezultatele vor fi urmatoarele:

```

[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpiCC -fopenmp -o
Exemplu4.2.1.exe Exemplu4.2.1.cpp

```

```

[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 2 -host
compute-0-0,compute-0-1 Exemplu4.2.1.exe

```

===Procesul MPI cu rankul 1 al nodului cu numele 'compute-0-1.local' a executat 2 fire ===

Hello from thread number 1,total number of theads are 2, MPI process rank is 1, real number processors is 4 on node compute-0-1.local

Hello from thread number 0,total number of theads are 2, MPI process rank is 1, real number processors is 4 on node compute-0-1.local

===Procesul MPI cu rankul 0 al nodului cu numele 'compute-0-0.local' a executat 4 fire ===

Hello from thread number 0,total number of theads are 4, MPI process rank is 0, real number processors is 4 on node compute-0-0.local

```

Hello from thread number 1,total number of theads are 4, MPI process rank
    is 0, real number processors is 4 on node compute-0-0.local
Hello from thread number 3,total number of theads are 4, MPI process rank
    is 0, real number processors is 4 on node compute-0-0.local
Hello from thread number 2,total number of theads are 4, MPI process rank
    is 0, real number processors is 4 on node compute-0-0.local
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 1 -host
    compute-0-0,compute-0-1 Exemplu4.2.1.exe

```

===Procesul MPI cu rankul 0 al nodului cu numele 'compute-0-0.local' a executat 4 fire ===

```

Hello from thread number 0,total number of theads are 4, MPI process rank
    is 0, real number processors is 4 on node compute-0-0.local
Hello from thread number 1,total number of theads are 4, MPI process rank
    is 0, real number processors is 4 on node compute-0-0.local
Hello from thread number 3,total number of theads are 4, MPI process rank
    is 0, real number processors is 4 on node compute-0-0.local
Hello from thread number 2,total number of theads are 4, MPI process rank
    is 0, real number processors is 4 on node compute-0-0.local
[Hancu_B_S@hpc Open_MP]$

```

Vom analiza în continuare următorul exemplu.

**Exemplu 4.2.2.** *Să se elaboreze un program în limbajul C++ în care se determină numărul total de fire generate pe nodurile unui cluster.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează condițiile enunțate în exemplu 4.2.2

<pre> #include &lt;omp.h&gt; #include "mpi.h" #define _NUM_THREADS 2 int main (int argc, char *argv[]) {     int my_rank,namelen;     char         processor_name[MPI_MAX_PR         OCESSOR_NAME];     omp_set_num_threads(_NUM_THR     EADS);     MPI_Init(&amp;argc, &amp;argv);     MPI_Get_processor_name(process     or_name,&amp;namelen);     MPI_Comm_rank(MPI_COMM_WO     RLD, &amp;my_rank); </pre>	<pre> int c=0, Sum_c;     #pragma omp parallel     reduction(+:c)     {         c = 1;     }     printf("The count of threads on the     MPI process %d of the compute     node '--%s--' is %d \n",     my_rank, processor_name,c);     MPI_Reduce(&amp;c, &amp;Sum_c, 1,     MPI_INT, MPI_SUM, 0,     MPI_COMM_WORLD);     if (my_rank == 0)     printf("Total number of threads=%d     \n", Sum_c); </pre>
--	--

```
MPI_Barrier(MPI_COMM_WORLD); | return 0;
MPI_Finalize();                | }
```

Rezultatele vor fi urmatoarele:

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpiCC -fopenmp -o
Exemplu4.2.2.exe Exemplu4.2.2.cpp
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 8 -machinefile
~/nodes Exemplu4.2.2.exe
The count of threads on the MPI process 7 of the compute node '--compute-
0-1.local--' is 2
The count of threads on the MPI process 6 of the compute node '--compute-
0-1.local--' is 2
The count of threads on the MPI process 4 of the compute node '--compute-
0-1.local--' is 2
The count of threads on the MPI process 5 of the compute node '--compute-
0-1.local--' is 2
The count of threads on the MPI process 0 of the compute node '--compute-
0-0.local--' is 2
The count of threads on the MPI process 1 of the compute node '--compute-
0-0.local--' is 2
The count of threads on the MPI process 2 of the compute node '--compute-
0-0.local--' is 2
The count of threads on the MPI process 3 of the compute node '--compute-
0-0.local--' is 2
Total number of threads=16
```

În aceasta variantă a programului instructiunea de atribuire `c = 1`; se executa de 16 ori ((8 procesoare)\*(2 fire)). Pentru a micșora numarul de executare a instructiunilor de atribuire se poate utiliza urmatorul program în care deja nu mai folosim clauza `reduction(+:c)`.

**Exemplu 4.2.2a.** *Să se elaboreze un program în limbajul C++ în care se determină numărul total de fire generate pe nodurile unui cluster.*

**Indicație.** Numărul de instrucțiuni de atribuire să fie egal cu numărul de procese MPI.

Mai jos este prezentat codul programului în limbajul C++ în care se realizează condițiile enunțate în exemplu 4.2.2a



```
#include <omp.h>
#include "mpi.h"
#define NUM_THREADS 2
int main (int argc, char *argv[])
{
    int p,my_rank,namelen;
    char
        processor_name[MPI_MAX_PRO-
        CESSOR_NAME];
    omp_set_num_threads(_NUM_THR-
    EADS);
    omp_lock_t lock;
    omp_init_lock(&lock);
    /* initialize MPI stuff */
    MPI_Init(&argc, &argv);
    MPI_Get_processor_name(process-
    or_name, &namelen);
    MPI_Comm_size(MPI_COMM_WO-
    RLD,&p);
    MPI_Comm_rank(MPI_COMM_WO-
    RLD,&my_rank);
    int c, Sum_c;
    sleep(my_rank);
    #pragma omp parallel
    {
```

```
        omp_set_lock(&lock);
        #pragma omp master
        {
            c = omp_get_num_threads();
            printf("    In regiunea paralela-
            curenta s-au generat %d fire\n",
                omp_get_num_threads());
        }
        omp_unset_lock(&lock);
    }
    printf("The count of threads on the
        MPI process %d of the compute
        node '--%s--' is %d \n",
        my_rank,processor_name,c);
    omp_destroy_lock(&lock);
    MPI_Reduce(&c, &Sum_c, 1,
        MPI_INT, MPI_SUM, 0,
        MPI_COMM_WORLD);
    if (my_rank == 0) printf("Total
        number of threads=%d \n",
        Sum_c);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

Rezultatele vor fi urmatoarele:

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpicc -fopenmp -o
Exemplu4.2.2a.exe Exemplu4.2.2a.cpp
```

```
[Hancu_B_S@hpc Open_MP]$ /opt/openmpi/bin/mpirun -n 6 -machinefile
~/nodes6 Exemplu4.2.2a.exe
```

In regiunea paralela curenta s-au generat 2 fire

The count of threads on the MPI process 0 of the compute node '--compute-0-0.local--' is 2

In regiunea paralela curenta s-au generat 2 fire

The count of threads on the MPI process 1 of the compute node '--compute-0-1.local--' is 2

In regiunea paralela curenta s-au generat 2 fire

The count of threads on the MPI process 2 of the compute node '--compute-0-3.local--' is 2

In regiunea paralela curenta s-au generat 2 fire

The count of threads on the MPI process 3 of the compute node '--compute-0-4.local--' is 2

In regiunea paralela curenta s-au generat 2 fire

The count of threads on the MPI process 4 of the compute node '--compute-0-5.local--' is 2

Total number of threads=12

In regiunea paralela curenta s-au generat 2 fire

The count of threads on the MPI process 5 of the compute node '--compute-0-11.local--' is 2

```
[Hancu_B_S@hpc Open_MP]$
```

Un scenariu foarte des utilizat pentru executarea programelor paralele utilizând modele de programare mixtă MPI-OpenMP pe un cluster paralel cu noduri de tip SMP (Symmetric Multi-processor - cu memorie partajată) conține următoarele etape.

- un singur proces MPI este lansat pe fiecare nod SMP în cluster;
- fiecare proces MPI generează  $N$  fire pe fiecare nod SMP;
- la un moment dat de sincronizare la nivel global, firul de bază pe fiecare SMP comunică unul cu altul;
- firele care aparțin fiecărui proces continuă executarea până la un alt punct de sincronizare sau completare.

În Figura 7 schematic este prezentat acest scenariu. Fiecare proces generează 4 fire pe fiecare dintre nodurile SMP. După fiecare iterație OMP paralelă în cadrul fiecărui nod SMP, firul de bază al fiecărui nod SMP comunică cu alte fire de bază ale nodurilor MPI folosind MPI apeluri. Din nou, iterația în OpenMP în cadrul fiecărui nod se realizează cu fire până când este completă.

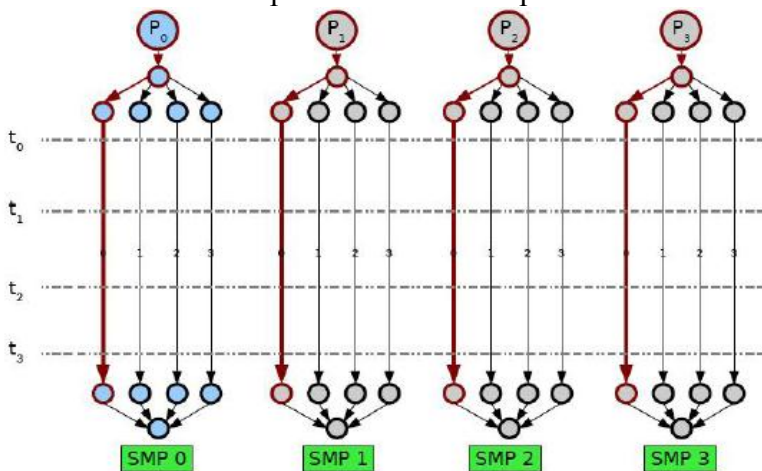


Figura 7. Scenariu de execuție mixtă MPI-OpenMP.

Vom exemplifica acest scenariu prezentat în Figura 7 prin următorul exemplu.

**Exemplu 4.2.3.** Să se elaboreze un program mixt MPI-OpenMP în care se calculează valoarea aproximativă a lui  $\pi$  prin integrare

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

dreptunghiurilor.

**Indicație.** Pe fiecare nod al clusterului se utilizează un singur proces MPI. Inițial intervalul închis  $[0,1]$  în subintervale  $[a_k, b_k]$  unde  $k$  este indicele nodului. Subintervalele  $[a_k, b_k]$  se împart într-un număr de  $n$  subintervale și se însumează ariile dreptunghiurilor având ca bază fiecare subinterval.

Mai jos este prezentat codul programului în limbajul C++ în care se realizează condițiile enunțate în exemplu 4.2.3. Pentru elaborarea acestui program au fost utilizate programele din Exemplu 2.1.7 și Exemplul 3.5.2 (Boris HÎNCU, Elena CALMÎȘ. *Modele de programare paralelă pe clustere. Partea I. Programare MPI*. Chișinău, 2016).

<pre> #include &lt;stdio.h&gt; #include &lt;stdlib.h&gt; #include &lt;mpi.h&gt; #include &lt;math.h&gt; #ifdef OPENMP #include &lt;omp.h&gt; #define TRUE 1 #define FALSE 0 #else #define omp_get_thread_num() 0 #endif double f(double y) {return(4.0/(1.0+y*y));} int main(int argc, char *argv[]) { int i, p, k=0, size, rank, rank_new; double PI25DT = 3.14159265358979323846264 3; int Node_rank; int Nodes; //numarul de noduri int local_rank = atoi(getenv("OMPI_COMM_W ORLD_LOCAL_RANK")); </pre>	<pre> char processor_name[MPI_MAX_PR OCESSOR_NAME]; MPI_Status status; MPI_Comm com_new, ring1; MPI_Group MPI_GROUP_WORLD, newgr; int *ranks, *newGroup; int namelen; MPI_Init(&amp;argc, &amp;argv); MPI_Comm_size(MPI_COMM_WO RLD, &amp;size); MPI_Comm_rank(MPI_COMM_WO RLD, &amp;rank); MPI_Get_processor_name(process or_name, &amp;namelen); if (rank == 0) printf("====REZULTATUL PROGRAMULUI '%s' \n", argv[0]); MPI_Barrier(MPI_COMM_WORLD); if (local_rank == 0) k = 1; MPI_Allreduce(&amp;k, &amp;Nodes, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD); </pre>
---	--

```

newGroup=(int *) malloc
(Nodes*sizeof(int));
ranks = (int *)
malloc(size*sizeof(int));
int r;
if (local_rank == 0)
    ranks[r] = rank;
else
    ranks[r] = -1;
for (int i = 0; i < size; ++i)
    MPI_Bcast(&ranks[i], 1,
    MPI_INT, i,
    MPI_COMM_WORLD);
for (int i = 0, j = 0; i < size; ++i)
{
    if (ranks[i] != -1)
    {
        newGroup[j] = ranks[i];
        ++j;
    }
}
MPI_Comm_group(MPI_COMM_W
ORLD,
&MPI_GROUP_WORLD);
MPI_Group_incl(MPI_GROUP_WO
RLD, Nodes, newGroup,
&newgr);
MPI_Comm_create(MPI_COMM_W
ORLD, newgr, &com_new);
MPI_Group_rank(newgr,
&rank_new);
if (rank_new != MPI_UNDEFINED)
{
    double w, x, sum, pi, Final_pi, a, b;
    int i, MPIrank;
    int n = 1000000;
    w = 1.0/n;
    sum = 0.0;
    a=(rank_new+0.0)/Nodes;
    b=(rank_new+1.0)/Nodes;
    omp_set_num_threads(2);

    #pragma omp parallel private(x)
    shared(w) reduction(+:sum)
    {
        if (rank_new==0)
        #pragma omp master
        {
            printf("Pentru fiecare proces MPI se
            genereaza %d procese
            OpenMP (fire)\n",
            omp_get_num_threads());
        }
        #pragma omp for nowait
        for(i=0; i < n; i++)
        {
            //x = w*(i-0.5);
            x = a+(b-a)*w*(i-0.5);
            sum = sum + f(x);
        }
        pi = (b-a)*w*sum;
        sleep(rank_new);
        printf ("Procesul %d al
        comunicatorului 'com_new' de
        pe nodul %s a calculat valoarea
        pi=%f pe [%f,%f]. \n",
        rank_new,processor_name,pi,a,
        b);
        MPI_Barrier(com_new);
        MPI_Reduce(&pi, &Final_pi, 1,
        MPI_DOUBLE, MPI_SUM, 0,
        com_new);
        if (rank_new == 0)
        {
            printf("Valoarea finala este
            %.16f, Error is %.16f\n",
            Final_pi, fabs(Final_pi -
            PI25DT));
        }
    }
    MPI_Finalize();
    return 0;
}

```

Rezultatele vor fi următoarele:

```

[Hancu_B_S@hpc OpenMP]$ /opt/openmpi/bin/mpiCC -fopenmp -o
Exemplu4.2.3New.exe Exemplu4.2.3.cpp
[Hancu_B_S@hpc OpenMP]$ /opt/openmpi/bin/mpirun -n 24 -machinefile
~/nodes6 Exemplu4.2.3.exe
=====REZULTATUL PROGRAMULUI 'Exemplu4.2.3New.exe'
Pentru fiecare proces MPI se genereaza 2 procese OpenMP (fire)
Procesul 0 de pe nodul compute-0-0.local a calculat valoarea pi=0.660595
pe [0.000000,0.166667].

```

```

Procesul 1 de pe nodul compute-0-1.local a calculat valoarea pi=0.626408
pe [0.166667,0.333333].
Procesul 2 de pe nodul compute-0-3.local a calculat valoarea pi=0.567588
pe [0.333333,0.500000].
Procesul 3 de pe nodul compute-0-4.local a calculat valoarea pi=0.497420
pe [0.500000,0.666667].
Procesul 4 de pe nodul compute-0-5.local a calculat valoarea pi=0.426943
pe [0.666667,0.833333].
Procesul 5 de pe nodul compute-0-11.local a calculat valoarea
pi=0.362640 pe [0.833333,1.000000].
Valoarea finala este 3.1415929869231181, Error is 0.0000003333333249
[Hancu_B_S@hpc OpenMP]$

```

#### ***4.4 Modalitati de comunicare în sisteme paralele hibrid (MPI-OpenMP).***

***Modalitatea 1.*** Un proces MPI de bază controlează toate comunicările.

Această modalitate este caracterizată prin:

- cea mai simplă paradigmă;
- procesul MPI este reprezentat (interpretat) ca un nod SMP;
- fiecare proces MPI generează un număr dat de fire de execuție cu memorie partajată;
- comunicarea între procesele MPI este gestionată doar de către procesul MPI principal, la intervale fixe predeterminate;
- permite un control strict tuturor comunicărilor.

Fragmentul de program prezentat mai jos ilustrează Modalitatea

1.

```

#include <omp.h>
#include "mpi.h"
#define _NUM_THREADS 4
int main (int argc, char *argv[]) {
    int p, my_rank;
    omp_set_num_threads(_NUM_THREADS);
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

```

```

#pragma omp parallel
{
    #pragma omp master
    {
        if ( 0 == my_rank)
            // some MPI_call as ROOT process
        else
            // some MPI_call as non-ROOT process
        }
    }
    printf("%d\n", c);
}

```

/* finalize MPI */ MPI_Finalize();		return 0; }
---------------------------------------	--	----------------

**Modalitatea 2.** *Firul de bază OpenMP controlează toate comunicările.*

Această modalitate este caracterizată prin:

- fiecare proces MPI utilizează propriul fir de bază OpenMP (1 pentru un nod SMP) pentru a comunica;
- permite mai multe comunicări asincrone;
- nu este la fel de rigid ca Modalitatea 1;
- necesita mai multă atenție pentru a asigura comunicarea eficientă, în schimb flexibilitatea poate rezulta în eficiență în altă parte.

Fragmentul de program prezentat mai jos ilustrează Modalitatea 2.

<pre>#include &lt;omp.h&gt; #include "mpi.h" #define _NUM_THREADS 4 int main (int argc, char *argv[]) {     int         p, my_rank; omp_set_num_threa ds(_NUM_THREADS); MPI_Init(&amp;argc, &amp;argv); MPI_Comm_size(MPI_COMM_WO RLD, &amp;p); MPI_Comm_rank(MPI_COMM_WO RLD, &amp;my_rank);</pre>		<pre>#pragma omp parallel {     #pragma omp master     {         // some MPI_ call as an MPI process     } } printf("%d\n", c); MPI_Finalize(); return 0; }</pre>
---	--	---

**Modalitatea 3.** *Toate firele OpenMP pot utiliza apeluri MPI*

Această modalitate este caracterizată prin:

- acest mod reprezintă cea mai flexibilă schemă de comunicare;
- permite un comportament distribuit adevărat similar cu cel dacă am folosi doar MPI;
- presupune cel mai mare risc de ineficiență dacă se folosește această abordare;
- necesită un efort în cazul cînd se calculează care fir al cărui proces MPI realizează transmitere/recepție de date;

- necesită o schemă de adresare care denotă: care procese MPI participă la comunicare și care fir al procesului MPI este implicat; de exemplu . <my\_rank,omp\_thread\_id>;
- nici MPI, nici OpenMP nu au facilități;
- pot fi utilizate secțiuni critice pentru un anumit nivel de control și corectitudine.

Fragmentul de program prezentat mai jos ilustrează Modalitatea

3.

<pre>#include &lt;omp.h&gt; #include "mpi.h" #define _NUM_THREADS 4 int main (int argc, char *argv[]) {     int p, my_rank;     omp_set_num_threads(_NUM_THREADS);     MPI_Init(&amp;argc, &amp;argv);     MPI_Comm_size(MPI_COMM_WORLD, &amp;p);     MPI_Comm_rank(MPI_COMM_WORLD, &amp;my_rank);     #pragma omp parallel</pre>	<pre>{     #pragma omp critical /* not         required */     {         // some MPI_call as an MPI         process     } } printf("%d\n",c); MPI_Finalize(); return 0; }</pre>
---	---

## Bibliografie

1. B. Wilkinson, M. Allen. *Parallel Programming*. Printice-Hall, 1999.
2. B. Dumitrescu. *Algoritmi de calcul paralel*. București, 2001.
3. Gh. Dodescu, B. Oancea, M. Raceanu. *Procesare paralelă*. București: Economica, 2002.
4. Gh.M. Panăitescu. *Arhitecturi paralele de calcul*. Ploiești: Universitatea “Petroil-Gaze”, 2007.
5. R.W. Hockney, C.R. Jesshope. *Calculatoare paralele: Arhitectură, programare și algoritmi*. București, 1991.
6. <http://www.mpi-forum.org/docs/docs.html>.
7. *MPI-2. Extension to the Message-Passing Interface*: <http://www.mpi-forum.org/docs/mpi20-html/>
8. P.S. Pacheco. *An Introduction to Parallel Programming*. 2011. pp. 370. [www.mkp.com](http://www.mkp.com) or [www.elsevierdirect.com](http://www.elsevierdirect.com)
9. Ph.M. Papadopoulos. *Introduction to the Rocks Cluster Toolkit – Design and Scaling*. San Diego Supercomputer Center, University of California, San Diego, <http://rocks.npaci.edu>
10. А.А. Букатов, В.Н. Дацюк, А.И. Жегуло. *Программирование многопроцессорных вычислительных систем*. Ростов-на-Дону: Издательство ООО ЦВБР, 2003.
11. А.С. Антонов. *Параллельное программирование с использованием технологии MPI*. Издательство Московского университета, 2004.
12. В.П. Гергель, Р.Г. Стронгин. *Основы параллельных вычислений для многопроцессорных вычислительных систем*. Нижний Новгород: Издательство Нижегородского государственного университета, 2003.
13. В.В. Воеводин. *Параллельные вычисления*. Москва, 2000.
14. Г.И. Шпаковский, Н.В. Сериков. *Программирование для многопроцессорных систем в стандарте MPI*, 2003.
15. В.П. Гергель. *Теория и практика параллельных вычислений*. <http://www.software.unn.ac.ru/ccam/kurs1.htm>
16. *Краткое руководство пользователя по работе с вычислительным кластером ТТИ ЮФУ*. Таганрог, 2010. <http://hpc.tti.sfedu.ru>
17. М.Л. Цымблер, Е.В. Аксенова, К.С. Пан. *Задания для практических работ и методические указания по их выполнению по дисциплине „Технологии параллельного программирования”*. Челябинск, 2012.



## Anexă

Vom prezenta o mostră de lucrare de laborator pentru implementarea soft pe clusterul USM a unui algoritmul paralel.

**Lucrare de laborator. Să se elaboreze un program MPI în limbajul C++ pentru determinarea în paralel a mulțimii tuturor situațiilor de echilibru în strategii pure pentru un joc bimatriceal.**

Fie dat un joc bimatriceal  $\Gamma = \langle I, J, A, B \rangle$  unde  $I$  – mulțimea de indici ai liniilor matricelor,  $J$  – mulțimea de indici ai coloanelor matricelor, iar  $A = \|a_{ij}\|_{\substack{i \in I \\ j \in J}}$ ,  $B = \|b_{ij}\|_{\substack{i \in I \\ j \in J}}$  reprezintă matricele de câștig ale jucătorilor.

Situația de echilibru este perechea de indici  $(i^*, j^*)$ , pentru care se verifică sistemul de inegalități:

$$(i^*, j^*) \Leftrightarrow \begin{cases} a_{i^* j^*} \geq a_{ij^*} \dots \forall i \in I \\ b_{i^* j^*} \geq b_{i^* j} \quad \forall j \in J \end{cases}$$

Vom spune că *linia  $i$  strict domină linia  $k$*  în matricea  $A$  dacă și numai dacă  $a_{ij} > a_{kj}$  pentru orice  $j \in J$ . Dacă există  $j$  pentru care inegalitatea nu este strictă, atunci vom spune că linia  $i$  domină linia  $k$ . Similar, vom spune: *coloana  $j$  strict domină coloana  $l$*  în matricea  $B$  dacă și numai dacă  $b_{ij} > b_{il}$  pentru orice  $i \in I$ . Dacă există  $i$  pentru care inegalitatea nu este strictă, atunci vom spune: *coloana  $j$  domină coloana  $l$* .

### Algoritmul de determinare a situației de echilibru

a) Eliminarea, în paralel, din matricea  $A$  și  $B$  a liniilor care sunt dominate în matricea  $A$  și din matricea  $A$  și  $B$  a coloanelor care sunt dominate în matricea  $B$ .

b) Se determină situațiile de echilibru pentru matricea

$$(A', B'), A' = \|a'_{ij}\|_{\substack{i \in I' \\ j \in J'}} \text{ și } B' = \|b'_{ij}\|_{\substack{i \in I' \\ j \in J'}} \text{ obținută din pasul a). Este}$$

clar că  $|I'| \leq |I|$  și  $|J'| \leq |J|$ .

- Pentru orice coloană fixată în matricea  $A'$  notăm (evidențiem) toate elementele maxime după linie. Cu alte cuvinte, se determină  $i^*(j) = \arg \max_{i \in I'} a'_{ij}$  pentru orice  $j \in J'$ . Pentru aceasta

se va folosi funcția **MPI\_Reduce** și operația **ALLMAXLOC**<sup>3</sup> (a se vedea exercițiul 4 din paragraful 3.4.4).

- Pentru orice linie fixată în matricea  $B'$  notăm toate elementele maximele de pe coloane. Cu alte cuvinte, se determină  $j^*(i) = \arg \max_{j \in J'} b'_{ij}$  pentru orice  $i \in I'$ . Pentru aceasta se va folosi funcția **MPI\_Reduce** și operația **ALLMAXLOC** (a se vedea Exercițiul 4 din paragraful 3.4.4).
- Selectăm acele perechi de indici care concomitent sunt selectate atât în matricea  $A'$  cât și în matricea  $B'$ . Altfel spus, se determină 
$$\begin{cases} i^* \equiv i^*(j^*) \\ j^* \equiv j^*(i^*) \end{cases}.$$

c) Se construiesc situațiile de echilibru pentru jocul cu matricele inițiale  $A$  și  $B$ .

Vom analiza următoarele exemple.

**Exemplul 1.** Situația de echilibru se determină numai în baza eliminării liniilor și a coloanelor dominate. Considerăm următoarele matrici:

$$A = \begin{pmatrix} 400 & 0 & 0 & 0 & 0 & 0 \\ 300 & 300 & 0 & 0 & 0 & 0 \\ 200 & 200 & 200 & 0 & 0 & 0 \\ 100 & 100 & 100 & 100 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -100 & -100 & -100 & -100 & -100 & -100 \end{pmatrix},$$

$$B = \begin{pmatrix} 0 & 200 & 100 & 0 & -100 & -200 \\ 0 & 0 & 100 & 0 & -100 & -200 \\ 0 & 0 & 0 & 0 & -100 & -200 \\ 0 & 0 & 0 & 0 & -100 & -200 \\ 0 & 0 & 0 & 0 & 0 & -200 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

<sup>3</sup> În cazul utilizării operației MAXLOC rezultatele pot fi incorecte.

Vom elimina liniile și coloanele dominate în următoarea ordine: *linia 5, coloana 5, linia 4, coloana 4, coloana 3, linia 3, coloana 0, linia 0, coloana 1, linia 1*. Astfel obținem matricele  $A' = (200)$ ,  $B' = (0)$  și situația de echilibru este  $(i^*, j^*) = (2, 2)$  și câștigul jucătorului 1 este 200, al jucătorului 2 este 0.

**Exemplul 2.** Considerăm următoarele matrice

$$A = \begin{pmatrix} 2 & 0 & 1 \\ 1 & 2 & 0 \\ 0 & 1 & 2 \end{pmatrix}, B = \begin{pmatrix} 1 & 0 & 2 \\ 2 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix}.$$

În matricea  $A$  nu există linii dominate, în matricea  $B$  nu există coloane dominate. Pentru comoditate, vom reprezenta acest joc astfel:  $AB = \begin{pmatrix} (\underline{2}, 1) & (0, 0) & (1, \underline{2}) \\ (1, \underline{2}) & (\underline{2}, 1) & (0, 0) \\ (0, 0) & (1, 2) & (\underline{2}, 1) \end{pmatrix}$ . Ușor se observă că în acest joc nu există situații de echilibru în strategii pure.

**Exemplul 3.** Considerăm următoarele matrice:

$$A = \|a_{ij}\|_{i \in I}^{j \in J}, B = \|b_{ij}\|_{i \in I}^{j \in J},$$

unde  $a_{ij} = c, b_{ij} = k$  pentru orice  $i \in I, j \in J$  și orice constante  $c, k$ . Atunci mulțimea de situații de echilibru este  $\{(i, j): \forall i \in I, \forall j \in J\}$ .

**Pentru realizarea acestui algoritm pe clustere paralele sunt obligatorii următoarele:**

- 1) Paralelizarea la nivel de date se realizează astfel:
  - a) Procesul cu rankul 0 inițializează valorile matricelor  $A$  și  $B$ . Dacă  $n > 5, m > 5$ , atunci procesul cu rankul 0 citește matricele dintr-un fișier text.
  - b) Distribuirea matricelor pe procese se face astfel încât să se realizeze principiul *load balancing*.
- 2) Paralelizarea la nivel de operații se realizează și prin utilizarea funcției **MPI\_Reduce** și a operațiilor nou create.

Boris HÂNCU, Elena CALMÎȘ

MODELE DE PROGRAMARE PARALELĂ PE CLUSTERE  
PARTEA II. Programare OpenMP și mixtă MPI-OpenMP

*Note de curs*

Redactare:  
Machetare computerizată:

Bun de tipar . Formatul  
Coli de tipar . Coli editoriale .  
Comanda 53. Tirajul 50 ex.

Centrul Editorial – Poligrafic al USM  
str. Mateevici, 60, Chișinău, MD