Lectia 16. ASPECTE COMPARATIVE ALE MODELELOR DE PROGRAMARE PARALELA MPI ŞI OPENMP. PROGRAMARE MIXTA MPI-OPENMP

16.1 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, şi variabile de mediu, care pot fi folosite pentru a specifica paralelismul de memorie partajată în programele Fortran şi C/C++.

MPI vs. OpenMP	
MPI	OpenMP
Modelul de memorie distribuită pe rețea distribuită. Bazat pe mesaje. Flexibil şi expresiv	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 utilizînd 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 utilizînd 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);
  }
}
```

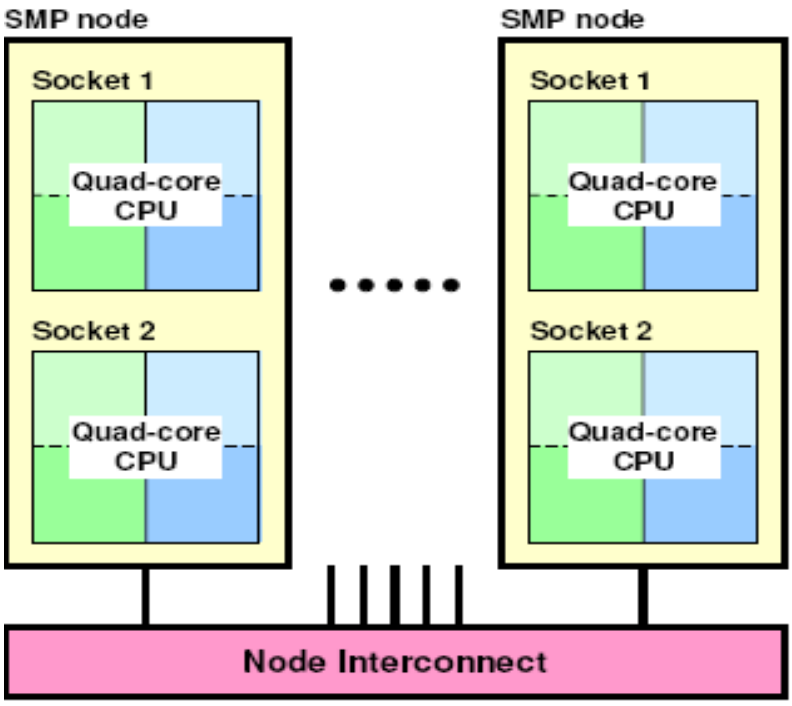
Rezultatele:

```
$/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!
```

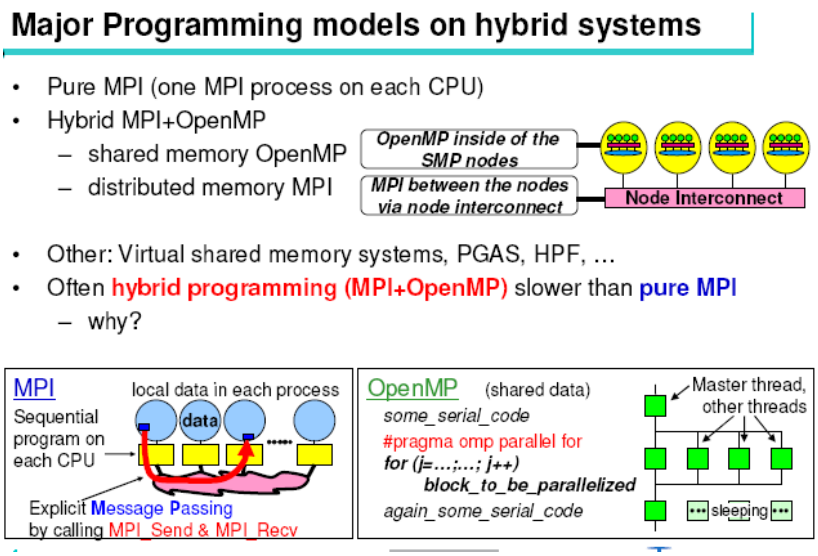
16.2 Programare paralelă mixtă MPI și OpenMP

Sistemele ***Cluster SMP*** (Symmetric Multi-Procesor) 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 şi cu memorie partajată) pot fi reprezentate astfel:



Modelele de programare paralelă sunt prezentate in urmatorul desen:



Deci, pentru modelele de programare paralela mixtă (MPI-OpenMP) se evidențiază (apare) următoarea problemă: realizarea comunicării folosind funcţiile MPI de transmitere/recepţionare a mesajelor prin intermediul firelor de execuție. Adică relațiile "apel al funcțiilor MPI și fire de execuţie". Cu alte cuvinte particularităţile programelor mixte MPI-OpenMP sunt determinate de următoarele alternative:

- procesele OpenMP (firele de execuţie) nu executa apeluri ale funcţiilor MPI,
- procesele OpenMP (firele de execuţie) executa apeluri ale funcţiilor MPI.

O abordare populară pentru sistemele HPC cu mai multe nuclee este de a utiliza în același program atât MPI cît 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.

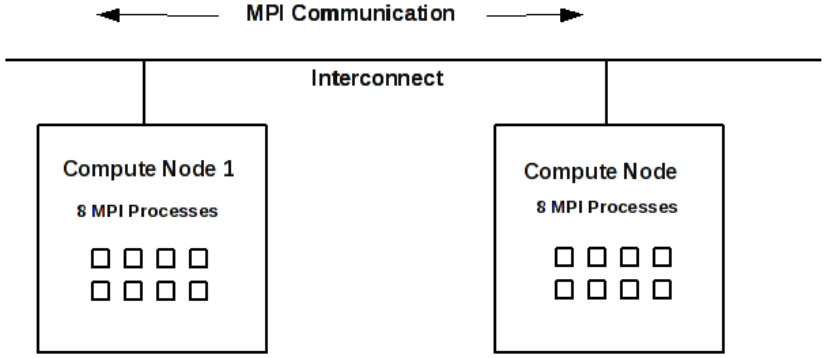


Figura 5: *Executarea programului MPI pe 2 noduri (16 procese MPI)*

Dacă s-ar utiliza atât modelul de programare MPI cît 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*. Clusterele care rulează benchmark-ul Top500 (HPL) folosesc acest tip de abordare.

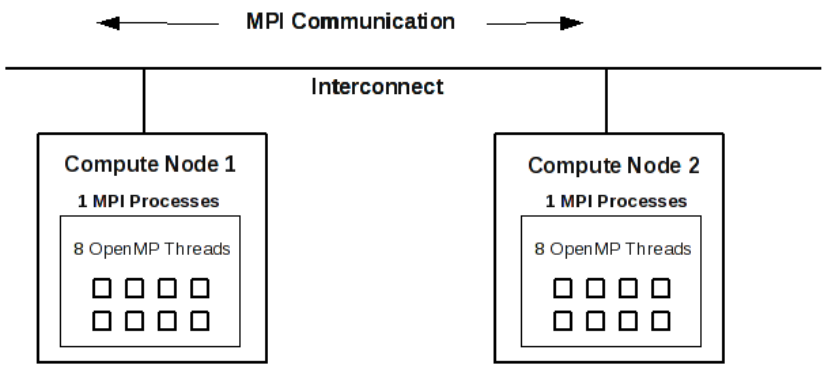


Figura 6: Executarea unui program MPI-OpenMP pe 2 noduri
(2 procese MPI, fiecare cu 8 fire OpenMP)

Vom prezenta mai jos exemple de programe în care se utilizează un model de programare paralelă mixt MPI-OpenMP astfel încât procesele OpenMP nu realizează apeluri ale funcţiilor MPI (adică în interiorul constructorilor **`#pragma omp parallel`** nu există funcţii MPI) .

Exemplu 16.2.1. *Acest exemplu ilustriază modalitatea de utilizare a funţiilor MPI şi a directivelor (rutinelor) OpenMP pentru elaborarea programelor mixte MPI-OpenMP. În programul de mai jos se genereaza diferite regiuni paralele de tip fire de execuţie în dependenţă de numele nodului pe care se execută aplicaţia. Programul se executa 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 fixeza numarul de fire pentru fiecare procesor fizic
#endif
//=======
 int numprocs, realnumprocs,rank, namelen,mpisupport;
 char processor_name[MPI_MAX_PROCESSOR_NAME];
 int iam = 0, np = 1;
omp_lock_t lock;
omp_init_lock(&lock);
MPI_Init(&argc, &argv);
MPI_Get_processor_name(processor_name, &namelen);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (strcmp(processor_name,"compute-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 theads 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 theads 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 16.2.2. *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 16.2.2

```
#include <omp.h>
#include "mpi.h"
#define _NUM_THREADS 2
int main (int argc, char *argv[])
{
int my_rank,namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
omp_set_num_threads(_NUM_THREADS);
```

```
MPI_Init(&argc, &argv);
MPI_Get_processor_name(processor_name,&namelen);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
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(&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 Exemplu4.2.2.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 micşora numarul de executare a instricţiunilor de atribuire se poate utiliza urmatorul program în care deja nu mai folosim clauza `reduction(+:c)`.

Exemplu 16.2.2a. *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 16.2.2a

```
#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_PROCESSOR_NAME];
omp_set_num_threads(_NUM_THREADS);
omp_lock_t lock;
omp_init_lock(&lock);
/* initialize MPI stuff */
MPI_Init(&argc, &argv);
MPI_Get_processor_name(processor_name, &namelen);
MPI_Comm_size(MPI_COMM_WORLD,&p);
```

```
MPI_Comm_rank(MPI_COMM_WORLD,&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]$
```

Cum a fost menţionat mai sus, un scenariu foarte des utilizat pentru executarea programelor paralele untilizând modele de programare mixtă MPI-OpenMP pe un cluster paralel cu noduri de tip SMP (Symmetric Multi-procesor - cu memorie partajată) conţine 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 aparţin 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 iteraţie 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.

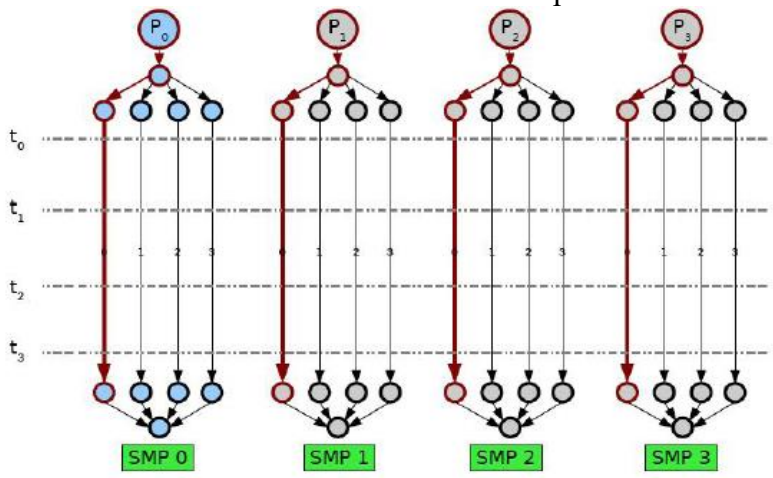


Figura 7. Scenariu de execuţie mixtă MPI-OpenMP.

Vom exemplifica acest scenariul prezentat în Figura 7, menţionând că apelurile de funcţii MPI nu se execută din interiorul constructorului de fire (procese OpenMP) paralele, prin următorul exemplu.

Exemplu 16.2.3. *Să se elaboreze un program mixt MPI-OpenMP în care se calculezează valoarea aproximativă a lui* π *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 ntervalul 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 şi 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 16.2.3. Pentru elaborarea acestui program au fost utilizate programele din Exemplu 2.1.7 și Exemplul 3.5.2 (Boris HÎNCU, Elena CALMÎȘ. M*odele de programare paralelă pe clustere. Partea I. Programare MPI.* Chișinău, 2016).

```
##include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>
#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(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_WORLD_LOCAL_RANK"));
char processor_name[MPI_MAX_PROCESSOR_NAME];
MPI_Status status;
MPI_Comm com_new, ring1;
MPI_Group MPI_GROUP_WORLD, newgr;
int *ranks,*newGroup;
int namelen;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Get_processor_name(processor_name, &namelen);
if (rank == 0)
printf("=====REZULTATUL PROGRAMULUI '%s' \n", argv[0]);
MPI_Barrier(MPI_COMM_WORLD);
 if (local_rank == 0) k = 1;
MPI_Allreduce(&k, &Nodes, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
newGroup=(int *) malloc (Nodes*sizeof(int));
ranks = (int *) malloc(size*sizeof(int));
int r;
  if (local_rank == 0)
     ranks[rank] = rank;
  else
     ranks[rank] = -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_WORLD, &MPI_GROUP_WORLD);
MPI_Group_incl(MPI_GROUP_WORLD, Nodes, newGroup, &newgr);
MPI_Comm_create(MPI_COMM_WORLD, 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]$
```

16.4 Modalitati de comunicare în sisteme paralele hibrid (MPI-OpenMP).

Suportul MPI pentru firele .....

Standardul MPI definește diferitele clase de utilizare a firelor (proceselor OpenMP). Utilizatorul poate solicita o anumită clasă pentru aplicația sa, iar implementarea poate returna clasa care este de fapt suportată. MPI definește următoarele patru clase de siguranță pentru utilizarea firelor:

MPI_THREAD_SINGLE. Va fi executat doar un fir.

MPI_THREAD_FUNNELED. Procesul MPI poate fi multi-fir, dar numai firul principal va face MPI apeluri (toate apelurile MPI sunt transferate firului principal)

MPI_THREAD_SERIALIZED. Procesul poate fi multi-fir și mai multe fire pot face apeluri MPI, dar numai una la un moment dat: apelurile MPI nu sunt făcute simultan din două fire distincte (toate apelurile MPI sunt serializate)

MPI_THREAD_MULTIPLE. Multiple fire pot apela fără restricţii funcţii MPI.

Un program care intenționează să utilizeze mai multe fire și apeluri MPI ar trebui să apeleze funcţia MPI_Init_thread în loc de MPI_Init pentru a inițializarea bibliotecii MPI. Această funcție conţine două argumente suplimentare: un nivel solicitat de spuport a firului (necesar) ca intrare și un nivel furnizat ca ieșire. Ambele argumente iau unul dintre cele patru niveluri specificate mai sus. Sintaxa pentru această funcție este următoarea:

```
int MPI_Init_thread(int *argc, char *((*argv)[]), int required,
int *provided);
```

Argumenţii required și provided pot lua una dintre cele patru valori pentru clasele de supor al firelor descrise mai sus. Sub aspect numeric avem

MPI_THREAD_SINGLE<MPI_THREAD_FUNNELED<
MPI_THREAD_SERIALIZED<MPI_THREAD_MULTIPLE.

Acest lucru face posibilă testarea faptului că biblioteca oferă cel puțin nivelul solicitat de supor al firelor necesar aplicației. Aceasta se poate fece cu ajutorul următorului fragment de program

```
if (provided < requested) {
printf("Not a high enough level of thread
support!\n");
MPI_Abort(MPI_COMM_WORLD,1);}
```

De asemenea, este posibilă determinarea nivelului de suport al firelor după apelul funcţiei **MPI_Init_thread ()** apelând funcţia **MPI_Query_thread ()** care returnează valoarea furnizată. Sintaxa acestei funcţii este

```
int MPI_Query_thread(int *provided);
```

În urnătorul program MPI se verifica ce suport al firelor poate asigura versiunea data a bibliotecii de funcţii MPI.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv) {
int version,subversion;
int mpi_rank,  mpi_err, provided,required;
MPI_Comm comm = MPI_COMM_WORLD;
mpi_err = MPI_Init_thread(&argc, &argv, required, &provided);
mpi_err = MPI_Comm_rank(comm, &mpi_rank);
MPI_Get_version(&version, &subversion);
printf("==== Pe clusterul USM este instalat MPI Versiunea %d.%d\n", version, subversion);
MPI_Query_thread(&provided);
switch (provided)
 {
 case MPI_THREAD_SINGLE: printf("===== Nivelul de suport al firelor este:
  MPI_THREAD_SINGLE \n"); break;
```

```
 case MPI_THREAD_FUNNELED: printf("===== Nivelul de suport al firelor este:
  MPI_THREAD_FUNNELED \n"); break;
 case MPI_THREAD_SERIALIZED: printf("===== Nivelul de suport al firelor este:
  MPI_THREAD_SERIALIZED \n"); break;
 case MPI_THREAD_MULTIPLE: printf("===== Nivelul de suport al firelor este:
  MPI_THREAD_MULTIPLE \n" ); break;
 default: /* */ break;
 }
MPI_Finalize();
return 0;
};
```

Pentru clasterul USM rezultatele executării programului vor fi următoarele.

```
Hancu_B_S@hpc OpenMP]$ /opt/openmpi/bin/mpiCC -fopenmp -o
    MPI_Threads_Suports.exe MPI_Threads_Suports.cpp
[Hancu_B_S@hpc OpenMP]$ /opt/openmpi/bin/mpirun -n 1  -host compute-0-1
    MPI_Threads_Suports.exe
==== Pe clusterul USM este instalat MPI Versiunea 2.1
===== Nivelul de suport al firelor este: MPI_THREAD_SINGLE
[Hancu_B_S@hpc OpenMP]$
```

În baza celor menţionate mai sus este posibil ca programele mixte MPI-OpenMP să fie clasificate în cinci stiluri distincte, în funcție de modul în care și cum multiple fire OpenMP fac apeluri la funcţiile din bibliotecă MPI.

- **Stilul „Master Only”:** toate comunicările MPI (adică apelul funcţiilor de tarnsmitere/receptionare a datelor) se fac din partea secvenţială a prgramului OpenMP (fără apeluri MPI în regiuni paralele).

Astfel în stilul **„Master Only”**, toate apelurile MPI sunt făcute de firul principal al OpenMP, din afara regiunilor paralele. Trebuie să menţionăm că, strict vorbind, acest stil necesită nivelul de suport MPI_THREAD_FUNNELED, deoarece alte fire vor fi executate, dar nu vor apela funcţii MPI.

Fragmentul de program prezentat mai jos ilustrează acest stil

```
#include <omp.h>
#include "mpi.h"
# int main (int argc, char *argv[])
{
MPI_Init(&argc, &argv);
Cod de program
#pragma omp parallel
{
  Cod de program
}
Apel al funcţiilor MPI
MPI_Finalize();
return 0;
}
```

Toate exemplele prezentate în paragraful 16.2 corespund acestul stil.

- **Stilul „Funneled”:** toate comunicările MPI (adică apelul funcţiilor de tarnsmitere/receptionare a datelor) se realizează de firul de unul ţi acelaţ fir (firul master), şi pot fi în interiorul regiunilor paralele ale programului OpenMP.
- **Stilul „Serialized”:** apelurile MPI pot fi făcute de orice fir, dar în orice moment de timp numai un fir face acel apel al funcţiei
- **Stilul „Multiple”:** comunicarea MPI poate fi realizată simultan de mai multe fire OpenMP .

- **Stilul „Asynchronous Tasks”:** comunicările MPI pot avea loc din oricare fir şi ,i apelurile MPI au loc în interiorul firelor OpenMP
  AICI În cazul în care

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 execuţie 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.

```
#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
  {
// some MPI_ call as an MPI process
  }
}
printf("%d\n",c);
MPI_Finalize();
return 0;
}
```

Modalitatea 3. *Toate firele OpenMP pot itiliza 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 transmitera/recepţionare 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 şi corectitudine.

Fragmentul de program prezentat mai jos ilustrează Modalitatea 3.

```
#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 critical /* not required */
  {
  // some MPI_ call as an MPI process
}
}
printf("%d\n",c);
MPI_Finalize();
return 0;
    }
```