Lectia 2. MODELUL MPI DE PROGRAMARE PARALELĂ PE SISTEME DE CALCUL CU MEMORIE DISTRIBUITĂ

Obiective

- Să cunoască noţiunile de „standard MPI”, „programe MPI”, să poată paraleliza un cod secvenţial al unui program în limbajul C++;
- Să cunoască sintaxa și modul de utilizare a funcţiilor MPI;
- Să elaboreze programe MPI în limbajul C++ pentru implementarea diferitor algoritmi paraleli pe sisteme de calcul paralel cu memorie distribuită;
- Să poată utiliza comenzile sistemului ROCKS pentru lansarea în execuție şi monitorizarea programelor MPI pe sisteme paralele de tip cluster.

2.1 Generalități

MPI este un standard pentru comunicarea prin mesaje, elaborat de MPI Forum. În definirea lui au fost utilizate caracteristicile cele mai importante ale unor sisteme anterioare, bazate pe comunicația de mesaje. A fost valorificată experiența de la IBM, Intel (NX/2) Express, nCUBE (Vertex), PARMACS, Zipcode, Chimp, PVM, Chameleon, PICL. MPI are la bază modelul proceselor comunicante prin mesaje: un calcul este o colecție de procese secvențiale care cooperează prin comunicare de mesaje. MPI este o bibliotecă, nu un limbaj. El specifică convenții de apel pentru mai multe limbaje de programare: C, C++, FORTRAN.77, FORTRAN90, Java.

MPI a ajuns la versiunea 3, trecând succesiv prin versiunile:

- ✓ MPI 1 (1993), un prim document, orientat pe comunicarea punct la punct;
- ✓ MPI 1.0 (iunie 1994) este versiunea finală, adoptată ca standard; include comunicarea punct la punct şi comunicarea colectivă;
- ✓ MPI 1.1 (1995) conține corecții și extensii ale documentului inițial din 1994, modificările fiind univoce;
- ✓ MPI 1.2 (1997) conține extensii și clarificări pentru MPI 1.1;
- ✓ MPI 2 (1997) include funcționalități noi:
  - procese dinamice;
  - comunicarea „one-sided”;
  - operații de I/E (utilizarea fişierelor) paralele.

Obiectivele MPI:

- proiectarea unei interfețe de programare a aplicațiilor;
- comunicare eficientă;
- să fie utilizabil în medii eterogene;
- portabilitate;
- interfața de comunicare să fie sigură (erorile tratate dedesubt);
- apropiere de practici curente (PVM, NX, Express, p4;
- semantica interfeței să fie independentă de limbaj.

Astfel, MPI este o bibliotecă de funcții, care permite realizarea interacțiunii proceselor paralele prin mecanismul de transmitere a mesajelor. Este o bibliotecă complexă, formată din aproximativ 130 de funcții care includ:

- funcții de inițializare și închidere (lichidare) a proceselor MPI;
- funcții care implementează operațiunile de comunicare de tip „proces-proces”;
- funcții care implementează operațiunile colective;
- funcții pentru lucrul cu grupuri de procese și comunicatori;
- funcții pentru lucrul cu structuri de date;
- funcția care implementează generarea dinamică a proceselor;
- funcții care implementează comunicări unidirecționale între procese (accesul la distanță a memoriei);
- funcții de lucru cu fișierele.

Fiecare dintre funcțiile MPI este caracterizată prin metoda de realizare (executare):

- funcție locală – se realizează în cadrul procesului care face apel la ea, la finalizarea ei nu este nevoie de transmitere de mesaje;
- funcție nelocală – pentru finalizarea ei este necesară utilizarea unor proceduri MPI, executate de alte procese;
- funcție globală – procedura trebuie să fie executată de toate procesele grupului;
- funcție de blocare;
- funcție de non-blocare.

În tabelul de mai jos se arată corespondența dintre tipurile predefinite de date MPI și tipurile de date în limbajul de programare.

Tip de date MPI	**Tip de date C**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

2.2 Funcțiile MPI de bază

Fiecare program MPI trebuie să conțină un apel al operației **`MPI_Init.`** Rolul acesteia este de a inițializa „mediul" în care programul va fi executat. Ea trebuie executată o singură dată, înaintea altor operații MPI. Pentru a evita execuția sa de mai multe ori (și deci provocarea unei erori), MPI oferă posibilitatea verificării dacă **`MPI_Init`** a fost sau nu executată. Acest lucru este făcut prin funcţia **`MPI_Iniatialized.`** În limbajul C funcția are următorul prototip:

```
int MPI_Initialized(int *flag);
```

unde

OUT **`flag`** – este true dacă **`MPI_Init`** a fost deja executată.

Aceasta este, de altfel, singura operație ce poate fi executată înainte de **`MPI_Init`**.

Funcția *MPI_Init*

Forma generală a funcției este

```
int MPI_Init(int *argc, char ***argv);
```

Funcția de inițializare acceptă ca argumente **`argc`** și **`argv`**, argumente ale funcției **`main`**, a căror utilizare nu este fixată de standard și depinde de implementare. Ca rezultat al executării acestei funcții se creează un grup de procese, în care se includ toate procesele generate de utilitarul **`mpirun`**, și se creează pentru acest grup un mediu virtual de comunicare descris de comunicatorul cu numele **`MPI_COMM_WORLD`**. Procesele din grup sunt numerotate de la *0* la *groupsize-1*, unde *groupsize* este egal cu numărul de procese din grup. La fel se creează comunicatorul cu numele **`MPI_COMM_SELF`**, care descrie mediul de comunicare pentru fiecare proces în parte.

Perechea funcției de inițializare este **`MPI_Finalize`**, care trebuie executată de fiecare proces, pentru a „închide" mediul MPI. Forma acestei operații este următoarea:

int MPI_Finalize(void).

Utilizatorul trebuie să se asigure că toate comunicațiile în curs de desfășurare s-au terminat înainte de apelul operației de finalizare. După **MPI_Finalize**, nici o altă operație MPI nu mai poate fi executată (nici măcar una de inițializare).

Funcția *MPI_Comm_size*

În limbajul C funcția are următorul prototip:

int MPI_Comm_size(MPI_Comm comm, int *size),

unde

IN **comm** – nume comunicator,
OUT **size** – numărul de procese ale comunicatorului **comm**.

Funcția returnează numărul de procese MPI ale mediului de comunicare cu numele **comm**. Funcția este locală.

Funcția *MPI_Comm_rank*

În limbajul C funcția are următorul prototip:

int MPI_Comm_rank(MPI_Comm comm, int *rank),

unde

IN **comm** – nume comunicator,
OUT **rank** – identificatorul procesului din comunicatorul **comm.**

Un proces poate afla poziția sa în grupul asociat unui comunicator prin apelul funcției **MPI_Comm_rank.** Funcția returnează un număr din diapazonul `0,..,size-1`, care semnifică identificatorul procesului MPI care a executat această funcție. Astfel proceselor li se atribuie un număr în funcție de ordinea în care a fost executată funcția. Această funcție este locală.

Vom exemplifica utilizarea funcțiilor MPI descrise mai sus.

Exemplul 2.2.1. *Să se elaboreze un program MPI în care fiecare proces tipărește rankul său și numele nodului unde se execută.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează condițiile enunțate în exemplu 2.2.1.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc,char *argv[])
{
int size,rank,namelen;
int local_rank = atoi(getenv("OMPI_COMM_WORLD_LOCAL_RANK"));
char processor_name[MPI_MAX_PROCESSOR_NAME];
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,
    &size);
MPI_Comm_rank(MPI_COMM_WORLD,
    &rank);
if (rank ==0)
printf("\n=====REZULTATUL PROGRAMULUI '%s'\n", argv[0]);
MPI_Barrier(MPI_COMM_WORLD);
MPI_Get_processor_name(processor_name, &namelen);
if (local_rank == 0) {
```

```
printf("==Au fost generate %s MPI processes pe nodul %s ===\n",
      getenv("OMPI_COMM_WORLD_LOCAL_SIZE"), processor_name); }
MPI_Barrier(MPI_COMM_WORLD);
if (rank ==0)
printf("==Procesele comunicatorului MPI_COMM_WORLD au fost 'distribuite' pe noduri astfel: \n");
MPI_Barrier(MPI_COMM_WORLD);
printf("Hello, I am the process number %d (local runk %d) on the compute hostname %s,from total
      number of process %d\n",rank, local_rank,processor_name, size);
MPI_Finalize();
return 0;
}
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu_3_2_1.exe Exemplu_3_2_1.cpp[1]
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 7 -machinefile ~/nodes Exemplu_3_2_1.exe
=====REZULTATUL PROGRAMULUI 'Exemplu_3_2_1.exe'
==Au fost generate 4 MPI processes pe nodul compute-0-0.local ===
==Au fost generate 3 MPI processes pe nodul compute-0-1.local ===
==Procesele comunicatorului MPI_COMM_WORLD au fost 'distribuite' pe noduri astfel:
Hello, I am the process number 0 (local runk 0) on the compute hostname compute-0-0.local , from total
number of process 7
Hello, I am the process number 3 (local runk 3) on the compute hostname compute-0-0.local , from total
number of process 7
Hello, I am the process number 2 (local runk 2) on the compute hostname compute-0-0.local , from total
number of process 7
Hello, I am the process number 4 (local runk 0) on the compute hostname compute-0-1.local , from total
number of process 7
Hello, I am the process number 5 (local runk 1) on the compute hostname compute-0-1.local , from total
number of process 7
Hello, I am the process number 1 (local runk 1) on the compute hostname compute-0-0.local , from total
number of process 7
Hello, I am the process number 6 (local runk 2) on the compute hostname compute-0-1.local , from total
number of process 7
[Hancu_B_S@hpc]$
```

[1] Numele programului corespunde numelui exemplului din notele de curs Boris HÎNCU, Elena CALMÎŞ "MODELE DE PROGRAMARE PARALELĂ PE CLUSTERE. PARTEA I. PROGRAMARE MPI". Chisinau 2016.