Laborator nr. 2
Introducere în MPI (1)

1 Standardul MPI

Acronim: *Message Passing Interface*

MPI (**M***essage* **P***assing* **I***nterface*) reprezintă o bibliotecă pentru programarea paralelă. MPI a fost creat în cadrul grupului **MPI Forum** între 1993 şi 1994, când a fost elaborat standardul 1.0. Acest standard definea, la momentul respectiv, mai mult de 120 de funcţii care oferă suport pentru:

Comunicaţii punct-la-punct – operaţiile de bază în recepţia şi trimiterea mesajelor de către procesele unui grup de lucru sunt send şi receive cu variantele de implementare a acestora.

Operaţii colective – sunt operaţii definite ca şi comunicaţii care implică un grup de procese.

Grupuri de procese şi contexte de comunicaţie

Topologia proceselor

În MPI un grup de procese este o **colecţie de n procese** în care fiecare proces are atribuit un **rang între 0 şi n-1**. În multe aplicaţii paralele o atribuire liniară a rangurilor nu reflectă logica necesarăcomunicaţiilor între procese. Adesea, procesele sunt aranjate în modele topologice (de exemplu o plasă cu 2 sau 3 dimensiuni – aplicaţii în calcul matriceal). Într-un caz şi mai general procesele pot fi descrise de un graf şi vom face referire la aranjarea acestor procese într-o **topologie virtuală**.

O distincţie clară trebuie făcută între o **topologie virtuală** şi **topologia de bază** a suportului (hardware-ul propriu-zis). **Topologia virtuală** poate fi exploatată de sistem în atribuirea proceselor către procesoarele fizice, dacă acest lucru ajută la creşterea performanţelor de comunicaţie pentru o anumită maşină, dar realizarea acestui lucru nu face obiectul MPI. Descrierea unei topologii virtuale depinde numai de aplicaţie şi este independentă de sistemul de calcul.

Legătura cu Fortran 77 şi C – aplicaţiile paralele folosind MPI pot fi scrise ăn limbaje ca Fortran 77 şi C.

Obţinerea informaţiilor despre mediu şi gestiunea acestuia – sunt introduse rutine pentru obţinerea şi setarea unor parametri pentru execuţia programelor corespunzătoare diferitelor implementări ale MPI.

Standardul MPI a ajuns la versiunea 3.0 şi oferă în plus suport pentru:

Extinderea operaţiilor colective – în MPI-1 intercomunicarea reprezenta comunicare între două grupuri de procese disjuncte. În multe funcţii argumentul de comunicare poate fi unul de **intercomunicare**, excepţie făcând operaţiile colective. Acest lucru este implementat în standardul MPI-2 şi extinde utilizarea funcţiilor pentru operaţiile colective.

Crearea proceselor şi gestiunea dinamică a acestora – obiectivele principale sunt oferirea posibilităţii de a crea aplicaţii cu un număr variat de task-uri (*task farm*) şi cuplarea aplicaţiilor după modelul client/server.

One-sided Communication – ideea de bază este aceea ca un task să aibă acces direct la memoria altui task (asemănător modelului shared memory). În plus, se oferă şi metode de sincronizare.

Parallel File I/O – mecanismele care oferă operaţii de I/O paralele, permit task-urilor să acceseze fişierele într-un mod controlat, dar sunt dependente de arhitectură. MPI-2 defineşte rutine de nivel şnalt pentru a furniza un model de programare unic pentru operaţiile de I/O paralele.

Legătura cu alte limbaje – este oferit suport pentru C++. De asemenea, există şi o variantă pentru Java.

2 Modelul de execuţie

Execuţia unui program scris cu apeluri MPI constă în **rularea simultană a mai multor procese**, definite în mod **static** (MPI-1) sau **dinamic** (începând MPI-2), identice sau diferite şi care **comunică punct-la-punct** sau **colectiv**. Procesele comunicante fac parte din acelaşi **comunicator**, care defineşte **domeniul de comunicaţie**. În cadrul domeniului de comunicaţie **fiecare proces are un rang**. De fapt comunicatorul asigură un mecanism de identificare a grupurilor de procese şi de protecţie a comunicaţiei. Un comunicator poate fi de două feluri: **intracomunicator** pentru procese din interiorul aceluiaşi grup şi **intercomunicator** pentru comunicaţiile dintre grupuri.

Există şase apeluri de funcţii mai importante care ar putea fi suficiente pentru scrierea unui program MPI:

MPI_Init – iniţiază *o aplicaţie* MPI;

MPI_Finalize – termină *o aplicaţie* MPI;

MPI_Comm_size – determină numărul proceselor;

MPI_Comm_rank – determină identificatorul procesului;

MPI_Send – trimite un mesaj;

MPI_Receive – primeşte un mesaj.

Înainte de apelul MPI_Init şi după apelul MPI_Finalize se pot apela doar funcţiile de interogare a mediului MPI. În Listing 1 este prezentată o primă aplicaţie MPI.

```
1  #include <stdio.h>
2  #include "mpi.h"
3
4  //!!! ATENTIE: PENTRU MAIN SE VA UTILIZA TOT TIMPUL
5  //              FORMA COMPLETA A FUNCTIEI !!!
6  int main ( int argc, char ** argv ) {
7          int count, myrank;
8
9          MPI_Init (&argc, &argv);
10         MPI_Comm_size (MPI_COMM_WORLD, &count);
11         MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
12
13         printf ("Eu sunt %d din %d\n", myrank, count);
14
15         MPI_Finalize();
16
17         //!!! OBLIGATORIU IN FINAL DE APLICATIE !!!
18         return 0;
19 }
```

Listing 1: Hello World MPI

Compilarea unei aplicaţii MPI – GNU C/C++ compiler:

- sursă C

```
mpicc nume_sursa.c -o nume_executabil.bin
```

- sursă C++

```
mpic++ nume_sursa.cpp -o nume_executabil.bin
```

Rularea aplicaţiilor MPI – cazul generic simplist (## reprezintă numărul dorit de procese – valoare întreagă strict pozitivă):

```
mpirun -np ## nume_executabil.bin
```

Parametrul MPI_COMM_WORLD precizează că toate procesele aparţin aceluiaşi grup şi reprezintă comunicatorul iniţial. Procesele pot executa instrucţiuni diferite, dar pentru aceasta trebuie să se asigure un mecanism de diferenţiere (Listing 2).

```
1  #include <stdio.h>
2  #include "mpi.h"
3
4  //!!! ATENTIE: PENTRU MAIN SE VA UTILIZA TOT TIMPUL
5  //             FORMA COMPLETA A FUNCTIEI !!!
6  int main ( int argc, char ** argv ) {
7          int count, myrank;
8
9          MPI_Init (&argc, &argv);
10         MPI_Comm_size (MPI_COMM_WORLD, &count);
11         MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
12
13         /*
14         ......
15         */
16
17         if ( myrank == 0 )
18                 //cod program master
19         else
20                 //cod program slave
21         /*
22         ......
23         */
24
25         MPI_Finalize();
26
27         //!!! OBLIGATORIU IN FINAL DE APLICATIE !!!
28         return 0;
29 }
```

Listing 2: Exemplu de diferenţiere a execuţiei proceselor MPI ce aparţin de acelaşi grup

3 Comunicaţii

O mare parte a funcţiilor MPI sunt funcţii pentru comunicaţii. Comunicaţiile pot fi **punct-la-punct** (sau **individuale**) sau **colective**. În continuare, **comunicaţiile individuale** pot fi **blocante** sau **neblocante**. **Comunicaţiile colective** sunt **întotdeauna blocante**, dar pot fi împărţite în comunicaţii colective **explicite** (*sunt caracterizate de un mesaj propriu*) sau **implicite** (*nu au un mesaj propriu*).

3.1 Comunicaţii blocante

După cum sugerează şi tipul acestor comunicaţii, procesul implicat într-un astfel de transfer de date se blochează până la finalizarea acestui schimb. O funcţie de comunicare cu blocare se consideră a fi finalizată odată cu trimiterea, respectiv primirea, mesajului. Astfel, la apelul funcţiei MPI_Send execuţia poate continua după revenirea din apel, chiar dacă mesajul nu a ajuns încă la destinaţie. Asemănător, se revine din apelul funcţiei MPI_Recv imediat ce mesajul a ajuns în buffer-ul de destinaţie şi poate fi citit.

Apelul unei **funcţii send**:

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
```

unde:

IN buf adresa zonei de memorie ce conţine datele de trimis

IN count numărul de elemente de transmis

IN datatype tipul datelor pentru fiecare element din buffer

IN dest identificatorul procesului destinaţie

IN tag identificator de mesaj

IN comm comunicatorul

Prototipul funcţiei este prezentat în Listing 3.

```
1 int MPI_Send(void* buf, int count, MPI_Datatype datatype,
2               int dest, int tag, MPI_Comm comm)
```

Listing 3: Prototipul funcţiei MPI_Send (C)

Apelul unei **funcţii receive**:

```
MPI_RECV (buf, count, datatype, source, tag, comm, status)
```

unde:

OUT buf adresa zonei de memorie în care va fi stocat mesajul recepţionat în cazul în care este primit corespunzător parametrilor funcţiei

IN count numărul maxim de elemente care pot fi primite

IN datatype tipul datelor pentru fiecare element din buffer primit

IN source identificatorul procesului sursă

IN tag identificator de mesaj

IN comm comunicatorul

OUT status variabilă utilă pentru aflarea dimensiunii, identificatorului de mesaj şi a sursei

Prototipul funcţiei este prezentat în Listing 4.

```
1 int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
2               int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Listing 4: Prototipul funcţiei MPI_Recv (C)

Un exemplu de utilizare a celor două funcţii este prezentat în Listing 5.

```c
#include <stdio.h>
#include "mpi.h"

int main( int argc, char **argv ) {
        char message[20];
        int myrank;

        MPI_Status status;

        MPI_Init( &argc, &argv );
        MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

        if (myrank == 0) { /* codul pentru procesul     0 */
                strcpy(message,"Hello");
                MPI_Send(message, strlen(message), MPI_CHAR,
                        1, 99, MPI_COMM_WORLD);
        } else { /* codul pentru procesul 1 */
                MPI_Recv(message, 20, MPI_CHAR,
                        0, 99, MPI_COMM_WORLD, &status);
                printf("received :%s:\n", message);
        }

        MPI_Finalize();
        return 0;
}
```

Listing 5: Prototipul funcţiei MPI_Send (C)

3.2 Moduri de comunicaţie

Există mai multe moduri de comunicaţie, indicate în numele funcţiei printr-o literă astfel: B pentru comunicaţii bufferate, S pentru comunicaţie sincronă şi R pentru comunicaţii de tip ready.

O operaţie de tip send în mod **buffered** poate fi iniţiată indiferent dacă operaţia receive corespunzătoare a fost iniţiată sau nu. n acest caz terminarea unei operaţii de tip send nu depinde de apariţia unei operaţie receive. Pentru a termina operaţia este necesar ca datele ce trebuie recepţionate să fie într-un buffer local. În acest scop, bufferul trebuie furnizat de către aplicaţie. Spaţiul (de memorie) ocupat de buffer este eliberat atunci când mesajul este transferat către destinaţie sau când operaţia este anulată.

O operaţie de tip send în mod **sincron** poate fi iniţiată indiferent de starea operaţiei receive corespunzătoare (a fost sau nu iniţiată). Oricum, operaţia va fi încheiată cu succes numai dacă o operaţie receive corespunzătoare a fost iniţiată şi a început recepţia mesajului trimis prin send. Astfel, încheierea operaţiei send sincrone nu indică numai că buffer-ul poate fi reutilizat, dar mai indică şi faptul că receptorul a ajuns într-un anumit punct din execuţia sa şi anume faptul că a iniţiat o operaţie receive corespunzătoare.

Observaţie: *Modul sincron asigură faptul că o comunicare nu se termină la fiecare sfârşit de proces, înainte ca ambele procese să ajungă la aceeaşi operaţie comunicare.*

În modul sincron operaţia de comunicare este încheiată simultan de către ambele funcţii indiferent de primul apel.

O operaţie de tip send în mod **ready** poate fi iniţiată numai dacă o operaţie receive a fost deja iniţiată, altfel operaţia este o sursă de erori. Pe unele sisteme acest lucru poate permite înlăturarea operaţiilor de tip handshake şi au ca rezultat creşterea performanţelor. De aceea, într-un program scris corect o operaţie ready–send poate fi înlocuită de una standard fără a avea alt efect asupra comportării programului decât performanţa.

3.3 Comunicaţii neblocante

Un mecanism care adesea conferă performaţe mai bune este utilizarea comunicaţiilor neblocante. Comunicaţiile neblocante pot fi folosite în cele patru moduri ca şi cele blocante: standard, bufferate, sincrone şi ready. Modul de folosire este acelaşi cu cel de la comunicaţiile blocante.

Cu excepţia modului ready operaţia send neblocantă poate fi iniţiată chiar dacă o operaţie receive a fost sau nu iniţiată, iar una de tip ready numai dacă a fost iniţiată o operaţie receive. în toate cazurile operaţia send este iniţiată local şi returnează imediat indiferent de starea altor procese.

Apelul unei **funcţii Isend**:

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
```

unde:

IN buf adresa zonei de memorie ce conţine datele de trimis

IN count numărul de elemente de trimis

IN datatype tipul datelor pentru fiecare element din buffer

IN dest rangul destinaţiei

IN tag identificator de mesaj

IN comm comunicatorul

OUT request handler-ul comunicaţiei

Prototipul funcţiei este prezentat în Listing 6.

```
1 int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
2                           int tag, MPI_Comm comm, MPI_Request *request)
```

Listing 6: Prototipul funcţiei MPI_Isend (C)

Apelul unei **funcţii Irecv**:

```
MPI_IRECV (buf, count, datatype, source, tag, comm, request)
```

unde:

OUT buf adresa zonei de memorie în care va fi stocat mesajul recepţionat în cazul în care este primit corespunzător parametrilor funcţiei

IN count numărul de elemente de trimis din buffer

IN datatype tipul datelor pentru fiecare element din buffer

IN source identificatorul procesului sursă

IN tag identificator de mesaj

IN comm comunicatorul

OUT request handler-ul comunicaţiei

Prototipul funcţiei este prezentat în Listing 7.

```
1 int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
2                           int tag, MPI_Comm comm, MPI_Request *request)
```

Listing 7: Prototipul funcţiei MPI_Irecv (C)

Din apelul de recepţie se revine chiar dacă nu există nici un mesaj primit. Acest mod de lucru poate conduce la erori şi în acest caz se folosesc funcţii de detectare a terminării operaţiei: `MPI_Wait()` şi `MPI_Test()`. O altă observaţie importantă este legată de faptul că operaţiile send/receive blocante pot fi completate de perechi neblocante (exemplu în Listing 8).

```
1  //...
2  MPI_Comm_rank (MPI_COMM_WORLD,&myrank);
3  MPI_Status status; MPI_Request req1;
4  int a,b;
5
6  if (myrank == 0) {
7          a = somevalue();
8          MPI_Isend(&a,1,MPI_INT,msgtag,MPI_COMM_WORLD,&req1);
9          calcul();
10         MPI_Wait(&req1, &status);
11 } else if (myrank == 1) {
12         MPI_Irecv(&b,1,MPI_INT,0,mesgtag,MPI_COMM_WORLD,&req1);
13         MPI_Wait(&req1, &status);
14 }
15 //...
```

Listing 8: Exemplu de utilizare – send neblocant/recv blocant

Funcţia MPI_Wait() se termină după finalizarea operaţiei, iar MPI_Test() se termină imediat şi returnează un indicator a cărui stare arată dacă operaţia s-a încheiat sau nu.

În afară de aceste funcţii mai există MPI_Waitany(), MPI_Testany(), MPI_Waitall(), MPI_Testall(), MPI_Waitsome(), MPI_Testsome().

Pentru mai multe detalii despre aceste tipuri de funcţii consultaţi help-ul distribuţiei MPI cu care lucraţi (ex: man MPI_Waitsome).

Observaţie: *Funcţiile de comunicaţie fără blocare oferă posibilitatea suprapunerii în timp a operaţiilor de comunicaţie cu cele de calcul, ceea ce poate reduce timpul de execuţie al aplicaţiei.*

Observaţie: *Sistemele de comunicaţii cu transfer de mesaje sunt în general nedeterministe, adică nu garantează că ordinea în care se transmit mesajele este aceeaşi cu ordinea în care mesajele sunt recepţionate, programatorului revenindu-i sarcina găsirii mecanismelor necesare pentru obţinerea unui calcul determinist.*

Listing-urile 9 şi 10 prezintă o variantă posibilă de implementare a problemei ”producători–consumatori” (cazul unui singur consumator) utilizând comunicaţii simple blocante şi neblocante.

```
1  //...
2  typedef struct{
3          char data[MAXSIZE];
4          int datasize;
5          MPI_Request req;
6  }Buffer;
7
8  Buffer *buffer;
9  MPI_Status status;
10
11 //...
12
13 MPI_Comm_rank(comm, &rank);
14 MPI_Comm_size(comm, &size);
15 if (rank!=size-1) { //codul pentru producator
16         //initializare   producatorul aloca un buffer
17         buffer = (Buffer *)malloc(sizeof(Buffer));
18         //producatorul umple bufferul cu date si intoarce numarul
```

```
19          //de bytes din buffer
20          produce(buffer->data, &buffer->datasize);
21          //trimite datele
22          MPI_Send(buffer->data, buffer->datasize, MPI_CHAR,
23                                  size-1, tag, comm);
24  } else { // rank==size-1; codul pentru consumator
25          //initializare   consumatorul aloca un buffer pentru fiecare
26          //producator
27          buffer=(Buffer )malloc(sizeof(Buffer)*(size-1));
28          // se apleleaza un receive pentru fiecare producator
29          for(i=0; i<size-1; i++) {
30                  MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i,
31                          tag, comm, &(buffer[i].req));
32          }
33          for(i=0; i<size-1; i++) { //bucla principala
34                  MPI_Wait(&(buffer[i].req), &status);
35                  // gaseste numarul de bytes primiti
36                  MPI_Get_count(&status, MPI_CHAR, &(buffer[i].datasize));
37                  // consumatorul elibereaza bufferul de date
38                  consume(buffer[i].data, buffer[i].datsize);
39          }//end for
40  }//end if-else
```

Listing 9: Problema producători–consumatori – cu utilizare MPI_Wait

```
1  //...
2  typedef struct{
3          char data[MAXSIZE];
4          int datasize;
5          MPI_Request req;
6  }Buffer;
7
8  Buffer *buffer;
9  MPI_Status status;
10
11 //...
12
13 MPI_Comm_rank(comm, &rank);
14 MPI_Comm_size(comm, &size);
15
16 if (rank!=size-1) { //codul pentru producator
17          //initializare   producatorul aloca un buffer
18          buffer= (Buffer *)malloc(sizeof(Buffer));
19          while(1) { //bucla principala
20                  //producatorul umple bufferul cu date si intoarce numarul
21                  //de bytes din buffer
22                  produce(buffer->data, &buffer->datasize);
23                  //trimite datele
24                  MPI_Send(buffer->data, buffer->datasize, MPI_CHAR,
25                                          size-1, tag, comm);
26          } //end while
27  } else { // rank==size-1; codul pentru consumator
28          //initializare   consumatorul aloca un buffer pentru fiecare
29          //producator
30          buffer=(Buffer )malloc(sizeof(Buffer)*(size-1));
31          // se apleleaza un receive pentru fiecare producator
32          for(i=0; i<size-1; i++)
33                  MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i,
34                                          tag, comm, &(buffer[i].req));
```

```
35
36          i=0;
37          while(1){ //bucla principala
38                  for(flag=0;!flag; i=(i+1)%(size-1)) {
39                          MPI_Test(&(buffer[i].req),&flag, &status);
40                  }//end for
41                  MPI_Get_count(&status, MPI_CHAR, &(buffer[i-1].datasize));
42                  // consumatorul elibereaza bufferul de date
43                  consume(buffer[i-1].data, buffer[i-1].datsize);
44                  //reinitializare
45                  //(in cazul in care mai sunt mesaje de la acelasi producator)
46                  MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i,
47                                  tag, comm, &(buffer[i].req));
48          }//end while
49  }//end else
```

Listing 10: Problema producători–consumatori – cu utilizare MPI_Test