MPI - MESSAGE PASSING INTERFACE

MPI este un standard international, care foloseste una dintre cele mai utilizate paradigme din lumea sistemelor paralele si distribuite: **comunicatia prin mesaje**. El a fost creat in momentul in care "rivalul" sau PVM (Parallel Virtual Machine), standard "de facto" dealtfel, era in plina glorie, tocmai pentru a oferi avantajele unui standard. Spre deosebire de PVM, exista mai multe implemenatri ale acestui standard (mpich, lam, chimp si chiar winmpi). Standardul a ajuns la versiunea 2.0, insa implemetarile sunt doar la versiunea 1.2. Varianta 2.0 a aparut pentru a corecta o serie de neajunsuri ale variantei 1.0, cea mai importanta fiind imposibilitatea creerii dinamice de procese.

Introducere in MPI

Pentru aplicatii uzuale, MPI este la fel de usor de folosit ca orice alt sistem bazat pe comunicatia prin mesaje. Scheletul unei aplicatii MPI de tip master-slave cu incarcare echilibrata este prezentat in continuare:

```
#include <mpi.h>
/* definesc tipuri posibile de mesaje */
#define WORKTAG         1
#define DIETAG          2

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

    MPI_Init(&argc, &argv);     /* initializare MPI */
    MPI_Comm_rank(              /* afla identificatorul procesului
curent */
            MPI_COMM_WORLD,     /* cel mai des folosit comunicator,
format din toate procesele MPI */
            &myrank);           /* identificatorul procesului curent,
intre 0 si N-1 */

    if (myrank == 0) master();
    else slave();

    MPI_Finalize();             /* termina "sesiunea" MPI */
}

master()
{   int                 ntasks, rank, work;
    double              result;
    MPI_Status          status;
```

```
    MPI_Comm_size(
            MPI_COMM_WORLD,     /* cel mai des folosit comunicator */
            &ntasks);           /* numarul de procese din aplicatia MPI
*/
/* Starteaza procesele slave */
    for (rank = 1; rank < ntasks; ++rank) {
        work = pregateste_urmatoarea_sarcina();
        MPI_Send(&work,         /* buffer pentru mesaj */
                1,              /* trimite o entitate de date */
                MPI_INT,        /* care data este de tip integer */
                rank,           /* identificatorul (rangul) procesului
destinatie */
                WORKTAG,        /* tip (identificator) mesaj */
                MPI_COMM_WORLD);/* cel mai des folosit comunicator */
    }
/* Receptioneaza raspunsul de la orice slave si pregateste urmatoarea
cerere */
    work = pregateste_urmatoarea_sarcina();

    while (sarcina_valida())
    {  MPI_Recv(&result,        /* buffer pentru mesaj */
                1,              /* receptie o entitate de date */
                MPI_DOUBLE,     /* care este de tip real */
                MPI_ANY_SOURCE, /* si care se accepta de la orice
emitator */
                MPI_ANY_TAG,    /* si care poate veni in orice tip de
mesaj */
                MPI_COMM_WORLD, /* cel mai des folosit comunicator */
                &status);       /* informatii de stare asupra mesajului
receptionat */
       MPI_Send(&work, 1, MPI_INT, status.MPI_SOURCE,WORKTAG,
MPI_COMM_WORLD);
        work = pregateste_urmatoarea_sarcina();
    }
/* Receptie rezultate pentru cererile in asteptare */
    for (rank = 1; rank < ntasks; ++rank) {
        MPI_Recv(&result, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
    }
/* Anunta toate procesele slave ca "munca" s-a terminat */
    for (rank = 1; rank < ntasks; ++rank) {
        MPI_Send(0, 0, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);
    }
}

slave()

{
    double              result;
    int                 work;
    MPI_Status          status;

    for (;;) {
        MPI_Recv(&work, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
/* Verifica tipul mesajului receptionat*/
```

```
        if (status.MPI_TAG == DIETAG) {return;}

        result = functie_slave();

        MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
}
```

Universul MPI

Procesele MPI au asociat un rang (identificator) **unic,** de tip intreg si cu valori intre 0, 1, 2, ..., N-1. MPI_COMM_WORLD este comunicatorul care cuprinde *toate procesel din aplicatia MPI*, si care contine toate informatiile necesare pentru a realiza comunicarea prin mesaje. Exista functii care lucreaza cu comunicatori, permitand operatii suplimentare asupra acestora.

Pornirea si parasirea MPI

Ca in orice sistem, exista doua functii speciale, care permit unui proces normal sa revendice, respectiv sa renunte la statutul de **proces MPI**.

```
MPI_Init(&argc, &argv);
MPI_Finalize( );
```

Cine sunt eu? Cine sunt ceilalti?

O informatie deosebit de utila pentru un proces MPI este identificatorul cu care este cunoscut in sistem. De asemenea,.un proces MPI trebuie sa cunoasca numarul de procese existente in sistem. Aflarea primei informatii se face cu apelul MPI_Comm_rank( ):

```
int         myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

Numarul total de procese este dat de functia MPI_Comm_size( ):

```
int         nprocs;
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

Trimiterea de mesaje

Un mesaj MPI este un vector de elemente de un tip bine precizat. Tipurile valide au denumirile prefixate de "MPI_". MPI cunoaste toate tipurile uzuale si permite construirea de tipuri complexe. Un mesaj este adresat unui proces identificat prin rangul sau si are atasat un **tip**, precizat de utilizator. Acest tip permite diferentierea intre mesajele pe care o aplicatie le-ar putea trimite sau primi in timpul executiei. In exemplul anterior existau doua tipuri de mesaje, de lucru si de terminare a lucrului.

```
    MPI_Send(buffer, count, datatype, destination, tag,
MPI_COMM_WORLD);
```

Receptia mesajelor

Un proces trebuie sa specifice tipul mesajului asteptat, precum si sursa acestuia. Se pot folosi constantele MPI_ANY_TAG si MPI_ANY_SOURCE pentru a putea receptiona orice tip de mesaj de la orice sursa.

```
    MPI_Recv(buffer, maxcount, datatype, source, tag, MPI_COMM_WORLD,
&status);
```

Parametrul status contine informatii despre mesajul receptionat.Tipul mesajului receptionat este in campul status.MPI_TAG, iar identificatorul sursei este in status.MPI_SOURCE.

Pentru a afla numarul de elemente receptionate se foloseste functia

```
    MPI_Get_count(&status, datatype, &nelements);
```

Functia se foloseste atunci cand numarul de elemente receptionate poate fi mai mic decat "maxcount".

Tipuri de date in MPI

Datele schimbate ca mesaje pot avea diverse reprezentari intr-un mediu heterogen. De aceea trebuie sa fie disponibil un mecanism prin care datele sa fie transmise fara nici un fel de probleme. MPI poate trimite date avand o larga varietate de tipuri, de la tipuri simple pana la structuri complexe.

Fuctille MPI care se ocupa cu transferul de mesaje admit un parametru de tip MPI_Datatype pentru a preciza tipul datelor. De exemplu, pentru apelul de trimitere a unui mesaj:

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

Cele mai utilizate tipuri sunt urmatoarele:

```
    MPI_CHAR (char)
    MPI_INT (int)
    MPI_FLOAT (float)
    MPI_DOUBLE (double)
```

Numarul parametrilor precizati in functia MPI_Send() se refera la numarul de elemente de un anumit tip, si nu la numarul de octeti.

Daca datele vehiculate reprezinta un vector omogen de elemente contigue, nu exista probleme deosebite. Daca insa mesajul contine date de tipuri diferite, sau care nu sunt contigue in memorie, atunci mai trebuie facut ceva.

Vectori si matrici

O aplicatie care trebuie sa trimita *coloane* ale unei matrici bidimensionale stocate *pe linii* trebuie sa foloseasca tipuri de date MPI derivate. Un exemplu este prezentat in continuare:

```
#include <mpi.h>
{   float               mesh[10][20];
    int                 dest, tag;
    MPI_Datatype        newtype;

/* Actiuni care se executa o singura data */
    MPI_Type_vector(10,         /* numarul de coloane */
            1,                  /* prima coloana */
            20,                 /* sari peste 20 elemente */
            MPI_FLOAT,          /* de tip float */
            &newtype);          /* tipul MPI derivat */
    MPI_Type_commit(&newtype);
/* Se executa pentru fiecare mesaj */
    MPI_Send(&mesh[0][19], 1, newtype, dest, tag, MPI_COMM_WORLD);
}
```

Un tip de date MPI derivat dupa creere poate fi folosit ulterior fara setari suplimentare. MPI are insa si alti constructori pentru tipuri derivate.

Structuri C

Fie o aplicatie de perlucrare imagini, care transfera linii de lungime fixa, cu pixeli avand maxim 8 culori. Impreuna cu matricea de pixeli se asociaza numarul liniei, de tip intreg. Structura C corespunzand unui mesaj este:

```
    struct {     int            lineno;
                 char           pixels[1024];
           } scanline;
```

In plus, operatia de impachetare a datelor derivate poate fi utila la trimiterea datelor ne-contigue si/sau heterogene. Un fragment de cod care impacheteaza si trimite structura anterioara este prezentat in continuare:

```
#include <mpi.h>
{   unsigned int        membersize, maxsize;
    int                 position;
    int                 dest, tag;
```

```
    char                *buffer;
/* Actiuni care se executa o singura data */
    MPI_Pack_size(1,            /* un element */
            MPI_INT,            /* tip intreg */
            MPI_COMM_WORLD,     /* communicator */
            &membersize);       /* spatiul maxim de impachetare necesar
*/

    maxsize = membersize;
    MPI_Pack_size(1024, MPI_CHAR, MPI_COMM_WORLD, &membersize);
    maxsize += membersize;
    buffer = malloc(maxsize);
/* Se executa pentru fiecare mesaj */
    position = 0;

    MPI_Pack(&scanline.lineno,  /* impacheteaza acest element */
            1,                  /* un element */
            MPI_INT,            /* tip int */
            buffer,             /* buffer de impachetare */
            maxsize,            /* dimensiune buffer */
            &position,          /* pozitia urmatorului byte liber */
            MPI_COMM_WORLD);    /* communicator */

    MPI_Pack(scanline.pixels, 1024, MPI_CHAR, buffer, maxsize,
&position, MPI_COMM_WORLD);

    MPI_Send(buffer, position, MPI_PACKED, dest, tag, MPI_COMM_WORLD);
}
```

Buffer-ul trebuie alocat pentru a avea dimensiunea structurii. Dimensiunea trenuie calculata din cauza overhead-ului specific implementarii care se ataseaza fiecarui mesaj. MEsajele de lungime variabila pot fi trimise daca se aloca suficient spatiu pentru cel mai mare mesaj posibil. Parametrul *position* al apelului MPI_Pack( ) returneaza intotdeauna dimensiunea actuala a datelor impachetate.

Despachetarea datelor intr-un buffer alocat deja se face ca in exemplul urmator:

```
{   int             src;
    int             msgsize;
    MPI_Status      status;

    MPI_Recv(buffer, maxsize, MPI_PACKED, src, tag, MPI_COMM_WORLD,
&status);

    position = 0;
    MPI_Get_count(&status, MPI_PACKED, &msgsize);

    MPI_Unpack(buffer,          /* buffer */
            msgsize,            /* dimensiune buffer */
            &position,          /* pozitia urmatorului element */
            &scanline.lineno,   /* despacheteaza acest element */
            1,                  /* un element */
            MPI_INT,            /* tip int */
            MPI_COMM_WORLD);    /* comm. */
```

```
    MPI_Unpack(buffer, msgsize, &position, scanline.pixels, 1024,
MPI_CHAR, MPI_COMM_WORLD);
}
```

Instalare MPI

Pentru a instala MPICH pe o masina UNIX trebuie facute urmatoarele actiuni:

1. Se obtin sursele uneia dintre cele cateva distributii existente (mpich, lam, chimp). De exemplu, sursele pentru varianta mpich le gasiti chiar aici:
   - Fisierul mpich_98.tar.gz (varianta 1998) complet ocupa cam 2.8 MB
2. Se dezarhiveaza fisierul mpich.tar.gz, de exemplu cu comenzile
3.
```
gzip -d mpich.tar.gz
tar xvf mpich.tar
```

   In urma acestei operatii rezulta un director mpich, care contine o serie de subdirectoare. In aceste subdirectoare se afla, in afara de sursele MPI o serie de exemple si fisiere postscript cu documentatie. Pentru o documentatie suplimentara, incercati si fisierul MPI-report.ps.gz (cam 230 pagini). Instalarea este bine sa o faca administratorul, dar nu este obligatoriu acest lucru. Fie MPI_ROOT calea catre directorul mpich.

4. Pentru Linux RedHat 6.x e bine sa se foloseasca urmatorul patch Modul de folosire e continut in fisierul respectiv.
5. Se lanseaza comanda

```
./configure
```

   care verifica daca se poate instala MPI si construieste fisierele Makefile.

6. Se lanseaza comanda

```
make
```

   care va realiza instalarea propriu-zisa.

   ATENTIE: Operatia de instalare dureaza cam mult: (cam 30 minute pe 486DX4, 16MB RAM, cam 15 minute pe K6II, 32MB RAM,)

7. Pentru ca totul sa mearga bine trebuie ca
   - La variabila PATH sa se adauge directoarele $MPI_ROOT/bin si $MPI_ROOT/lib/LINUX/ch_p4 (daca instalarea se face pe LINUX)

- Masina sa permita apeluri rsh (fie exista un fisier .rhosts in directorul home al user-ului, fie se foloseste fisierul /etc/hosts.equiv)

Utilizare MPI

- Compilarea programelor care folsoesc MPI se realizeaza cu comanda

  ```
  mpicc
  ```

  (de preferat unui make), care functioneaza asemanator compilatorului uzual

  ```
  cc
  ```

  sau

  ```
  gcc
  ```

  . De exemplu:

  ```
  mpicc -o program program.c
  ```

- Rularea programelor MPI se face cu comanda

  ```
  mpirun
  ```

  , specificandu-se numarul de procese (**identice**) care se executa:

  ```
  mpirun -np 4 program
  ```

  Procesele dorite vor fi lansate pe masinile care se gasesc in fisierul $MPI_ROOT/bin/machines/machines.LINUX. In acest fisier este permisa specificarea repetata a aceleiasi masini (la limita se poate sa existe o singura masina).

- Pentru fep:
  - Pentru a putea compila cu mpcc, este necesar sa completati PATH in fisierul .profile. Includeti /opt/SUNWhpc/bin si opt/SUNWspro/bin inainte de /usr/ucb
  - Pentru compilare, **mpcc -lmpi *.c -o fisier_out**
  - Pentru executie (de pe fep):

- fep: **mprun -np #NP ./fisier_out**
- hpcdom: **hpcrun mprun -np #NP ./fisier_out**

#NP = numarul de procesoare; pe fep maxim 2, pe hpcdom maxim 24.