Continut MPI-2

- Extensii ale modelului "message passing"
  - I/E paralele
  - Operații "one-sided"
  - Gestiunea dinamică a proceselor
- **MPI-2** abordat de majoritatea producătorilor de sisteme distribuite.
  - Fujitsu, NEC au implementări complete MPI-2
  - I/E disponibile în cele mai multe implementări
  - Operații *one-sided* disponibile la SGI, HP, IBM
  - Gestiunea dinamică a proceselor în LAM (Local Area Multicomputer) MPI

Gestiunea Dinamică a Proceselor în MPI-2

- MPI_Comm_spawn
  - Porneşte **n** procese noi
- MPI_Open_port, MPI_Comm_connect, MPI_Comm_accept
  - permit ca doua programe în execuție să se conecteze şi să comunice
  - Proiectate să suporte aplicații HPC (nu client/server)
- MPI_Join
  - permite utilizarea sockets TCP pentru a conecta două aplicații

Procese dinamice

int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])

porneste **maxprocs** copii ale unui program specificat de **command**;
argv - argumentele transmise programului
info - informatii aditionale = siruri de caractere (cheie, valoare)
specifica sist runtime unde si cum sa porneasca procesele fii
operatia **executată în colectiv** de procesele grupului **comm**
MPI_COMM_SELF – contine doar procesul executant
argumentele examinate doar de **root**.
procesele fii au propriul lor MPI_COMM_WORLD.
apelul întoarce un **intercomm** pentru comunicare între părinti si fii
intercomm obtinut de copii prin **MPI_Comm_get_parent**
array_of_errcodes – cate un cod pentru fiecare fiu

intercomunicator
Contine un grup local si unul distant
Comunicare punct la punct intre un proces dintr-un grup si unul din celalalt

Exemplu master - worker

- Functia = copiere fisier in paralel
- Un master decide cati “workers” sa creeze
- Master
  - deschide fisierul de intrare
  - difuzeaza numele fisierului de iesire
  - toate procesele (inclusiv master-ul) creaza fisierul de iesire local
  - Masterul citeste si difuzeaza inregistrari din fisierul de intrare
- Workers
  - Primesc numele fisierului de iesire
  - Deschid fisierul local
  - Primesc si scriu inregistrari
- Transmitere de mesaje in paralel (MPI_Bcast)
- Toate procesele scriu in paralel
- Sintaxa comenzii “parallel copy” similara cu cp:

```
pcp 0-63 /home/progs/cpi /tmp/cpi
```

Programul Master

```
#include "mpi.h"
#include . . .
#define BUFSIZE   256*1024
#define CMDSIZE   80
int main( int argc, char *argv[] )
{
   int mystatus, allstatus, done, numread, num_hosts;
   char outfilename[128], controlmsg[80];
   int infd, outfd;
   char buf[BUFSIZE];
   MPI_Info hostinfo;                /* set de perechi (nume, valoare) */
   MPI_Comm pcpslaves;               /* intercomunicator master - slaves */

   MPI_Init( &argc, &argv );
   makehostlist( argv[1], "targets", &num_hosts );

   ... creaza si actualizeaza hostinfo ...

   MPI_Comm_Spawn( "pcp_slave", MPI_ARGV_NULL, num_hosts, hostinfo,
            0, MPI_COMM_SELF, &pcpslaves, MPI_ERRCODES_IGNORE );
   MPI_Info_free( &hostinfo );
```

```
/* command syntax: pcp 0-63 /home/progs/cpi /temp/cpi */
  strcpy( outfilename, argv[3] );
  if ( (infd = open( argv[2], O_RDONLY ) ) == -1 ) {
     fprintf( stderr, "input %s does not exist\n", argv[2] );
    sprintf( controlmsg, "exit" );
    MPI_Bcast( controlmsg, CMDSIZE, MPI_CHAR, 0, pcpslaves );
    MPI_Finalize();
    return( -1 );
  }
  else {
     sprintf( controlmsg, "ready" );
     MPI_Bcast( controlmsg, CMDSIZE, MPI_CHAR, 0, pcpslaves );
  }

  sprintf( controlmsg, outfilename );
  MPI_Bcast( controlmsg, CMDSIZE, MPI_CHAR, 0, pcpslaves );
  if ( (outfd = open( outfilename, O_CREAT|O_WRONLY|O_TRUNC,
                      S_IRWXU ) ) == -1 )
     mystatus = -1;
  else
     mystatus = 0;
  MPI_Allreduce( &mystatus, &allstatus, 1, MPI_INT, MPI_MIN, pcpslaves );
  if ( allstatus == -1 ) {
     fprintf( stderr, "output file %s could not be opened\n", outfilename );
```

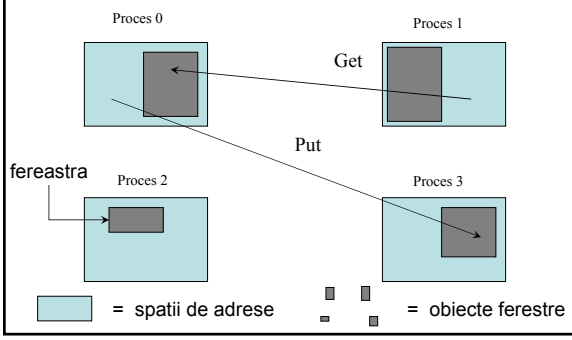
```
      MPI_Finalize();
      return( -1 );
    }

  /* at this point all files have been successfully opened */

    done = 0;
    while ( !done ) {
       numread = read( infd, buf, BUFSIZE );
      MPI_Bcast( &numread, 1, MPI_INT, 0, pcpslaves );
      if ( numread > 0 ) {
         MPI_Bcast( buf, numread, MPI_BYTE, 0, pcpslaves );
         write( outfd, buf, numread );
      }
      else {
         close( outfd );
         done = 1;
      }
    }
    MPI_Comm_free( &pcpslaves );
    MPI_Finalize();
  }
```

Programul Worker

```
/* pcp from SUT, in MPI */
#include "mpi.h"
#include . . .
#define BUFSIZE    256*1024
#define CMDSIZE    80
int main( int argc, char *argv[] )
{
   int mystatus, allstatus, done, numread;
   char outfilename[128], controlmsg[80];
   int outfd;
   char buf[BUFSIZE];
   MPI_Comm slavecomm;

   MPI_Init( &argc, &argv );
   MPI_Comm_get_parent( &slavecomm );
   MPI_Bcast( controlmsg, CMDSIZE, MPI_CHAR, 0, slavecomm );
   if ( strcmp( controlmsg, "exit" ) == 0 ) {
      MPI_Finalize();
    return( -1 );
   }

   MPI_Bcast( controlmsg, CMDSIZE, MPI_CHAR, 0, slavecomm );
   if ( (outfd = open( controlmsg, O_CREAT|O_WRONLY|O_TRUNC,
                       S_IRWXU ) ) == -1 )
```

```
  mystatus = -1;
   else
      mystatus = 0;
   MPI_Allreduce( &mystatus, &allstatus, 1, MPI_INT, MPI_MIN,
                  slavecomm );
   if ( allstatus == -1 ) {
    MPI_Finalize();
    return( -1 );
   }
   /* at this point all files have been successfully opened */
   done = 0;
   while ( !done ) {
    MPI_Bcast( &numread, 1, MPI_INT, 0, slavecomm );
    if ( numread > 0 ) {
       MPI_Bcast( buf, numread, MPI_BYTE, 0, slavecomm );
       write( outfd, buf, numread );
    }
    else {
       close( outfd );
       done = 1;
    }
   }
   MPI_Comm_free( &slavecomm );
   MPI_Finalize();
   return 0;
}
```

RMA - Remote Memory Access



Remote Memory Access

int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)

Expune memoria data de (base, size) operatiilor RMA ale altor procese din comm
win este obiectul window folosit in operatiile RMA
disp_unit unitatea de masura pentru deplasamente
1 – nu se scaleaza
sizeof(type) - unde win este un tablou de elemente de tip type

int MPI_Put (void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)
Modifica memoria la distanta

int MPI_Get (void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)
Citeste din memoria la distanta

MPI_Accumulate (void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
Actualizeaza memoria la distanta

int MPI_Win_fence (int assert, MPI_Win win)
– rol de bariera (assert = 0 este valid indicand nici o conditie speciala pentru optimizarea operatiei)

Calcul Pi cu RMA

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{  int n, myid, numprocs, i;
   double PI25DT = 3.14159265358979323846264643;
   double mypi, pi, h, sum, x;
   MPI_WIN nwin, piwin;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    if (myid == 0) {
     MPI_Win_create(&n, sizeof(int), 1, MPI_INFO_NULL,
                    MPI_COMM_WORLD, &nwin);
     MPI_Win_create(&pi, sizeof(double), 1, MPI_INFO_NULL,
                    MPI_COMM_WORLD, &piwin);
   }
   else { /* does not expose memory, size=0 */
     MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
                    MPI_COMM_WORLD, &nwin);
     MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
                    MPI_COMM_WORLD, &piwin);
   }
```

```
while (1) {
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d", &n);
    pi = 0.0;
    }
    MPI_Win_fence(0, nwin);
    if (myid != 0)
        MPI_Get(&n, 1, MPI_INT, 0, 0, 1, MPI_INT, nwin);
    MPI_Win_fence(0, nwin);
    if (n == 0)
        break;
    else {
        h   = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <= n; i += numprocs) {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
        }
        mypi = h * sum;
        MPI_Win_fence( 0, piwin);
        MPI_Accumulate(&mypi, 1, MPI_DOUBLE, 0, 0, 1, MPI_DOUBLE,
                MPI_SUM, piwin);
        MPI_Win_fence(0, piwin);
        if (myid == 0)
            printf("pi is approximately %.16f, Error is",
                                "%.16f\n", pi, fabs(pi - PI25DT));
    } /* end if */
  } /* end while */
  MPI_Win_free(&nwin);
  MPI_Win_free(&piwin);
  MPI_Finalize();
  return 0;
}
```

MPI-2 File I/O

Mai multe procese acceseaza un singur fisier
Obiectiv: acces la date si fisiere cu cat mai putine apeluri I/O

Set de extensii ale standardului MPI original
- o specificatie de interfata (fara detalii de implementare)
- rutine pentru gestiunea fisierelor si access la date
- apelurile MPI-I/O portabile pe multe arhitecturi

Definitii

etype (elementary type) unitate de baza de acces la date (similar unei inregistrari)

offset pozitie intr-un fisier, relativa la vederea curenta, exprimata ca numar de *etypes*

file pointers offset-uri in fisierul gestionat de MPI
- *Individual file pointer* - local procesului care a deschis fisierul
- *Shared file pointer* - partajat (si gestionat) de grupul de procese care au deschis fisierul

Gestiunea Fisierelor

MPI_FILE_OPEN(Comm, filename, mode, info, fh, ierr)

Deschide fisierul ***filename*** pe fiecare proces din comunicatorul *Comm*
Operatie colectiva pentru procesele din *Comm*
Fiecare procesor foloseste aceeasi valoare pentru *mode* si refera acelasi fisier

modes

MPI_MODE_CREATE	**creaza fisierul daca nu exista**
MPI_MODE_RDONLY	
MPI_MODE_RDWR	
MPI_MODE_WRONLY	
MPI_MODE_EXCL	**eroare daca se creaza fisier existent**
MPI_MODE_DELETE_ON_CLOSE	
MPI_MODE_UNIQUE_OPEN	**fisierul nu va fi deschis concurent altundeva**
MPI_MODE_SEQUENTIAL	
MPI_MODE_APPEND	**pozitia initiala la sfarsit fisier**

info hint-uri despre tiparul de acces; uzual este MPI_INFO_NULL
fh file handle
ierr cod eroare

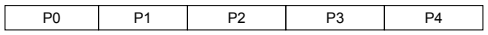
Un exemplu simplu

Un fisier partajat

Fiecare citeste / scrie **1/p** din date

Fiecare proces poate pastra propriul file pointer (fiecare scrie in alta parte)

Similar cu Unix I/O: open, seek, write (sau read)



Read dintr-un fisier comun cu *file pointers* individuali

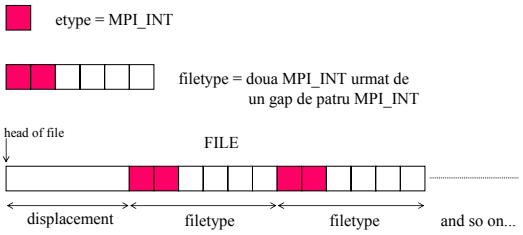
```
MPI_File fh;
MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

bufsize = FILESIZE/nprocs;
buf = (int *) malloc(bufsize);
nints = bufsize/sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", MPI_MODE_RDONLY,
        MPI_INFO_NULL, &fh);
MPI_File_seek(fh, rank*bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
```

MPI2 File Views

- ***File view*** indica partea de fisier vizibila unui proces
- Procesele pot partaja un *view* unic, sau pot avea *view*-uri diferite.
- *File view* definit de
  - ***displacement*** de la inceput fisier (in bytes)
  - ***etype*** (elementary type) unitate de baza de acces la date (similar unei inregistrari)
  - ***filetype*** tip de date folosit pentru a descrie tiparul de access al unui fisier

- Uzual: filetype = etype + gap-uri
  - Ex. filetype = secventa de etypes.
  - gauri: zone nefolosite (folosite de alte procese)
  - necesita LB, UB pentru a identifica pozitia gap-urilor

Exemplu de File View

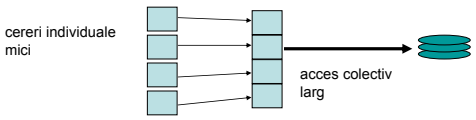


Cod pentru File View

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int); etype = MPI_INT;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
     MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

MPI I/O Colectiv

- Optimizare critica in I/O paralel
- Cadru pentru I/O in doua faze, in care comunicarea precede I/O
- Ideea de baza : construieste blocuri mari, astfel ca citirile/scrierile in sistemul I/O sa fie mari



I/O colectiv

- `MPI_File_read_all`, `MPI_File_read_at_all`, etc
  `_all` - toate procesele din grup (specificat de comunicatorul pasat la `MPI_File_open`) vor apela aceasta functie

- Fiecare proces specifica doar propria sa informatie de acces
  - lista de argumente este aceeasi ca pentru functii ne-colective
- Prin apelul functiilor colective I/O, utilizatorul permite sistemului sa optimizeze operatiile pe baza cererii combinate a proceselor
  - Implementarea poate uni cererile diferitelor procese si servi mai eficient cererea combinata
  - Eficienta cand accesele diferitelor procese sunt non-contigue si intercalate

Acces ne-contiguu cu o singura functie colectiva I/O

```
#include "mpi.h"

#define FILESIZE     1048576
#define INTS_PER_BLK  16

int main(int argc, char **argv)
{
   int *buf, rank, nprocs, nints, bufsize;
   MPI_File fh;
   MPI_Datatype filetype;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

   bufsize = FILESIZE/nprocs;
   buf = (int *) malloc(bufsize);
   nints = bufsize/sizeof(int);
```

```
    MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
                  MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);

    MPI_Type_vector(nints/INTS_PER_BLK, INTS_PER_BLK,
                INTS_PER_BLK*nprocs, MPI_INT, &filetype);
    MPI_Type_commit(&filetype);
    MPI_File_set_view(fh, INTS_PER_BLK*sizeof(int)*rank, MPI_INT,
                filetype, "native", MPI_INFO_NULL);

    MPI_File_read_all(fh, buf, nints, MPI_INT, MPI_STATUS_IGNORE);
    MPI_File_close(&fh);

    MPI_Type_free(&filetype);
    free(buf);
    MPI_Finalize();
    return 0;
}
```

Implementari MPI

MPICH, dezvoltată la Argonne National Laboratory & Mississippi State University.

Bazat pe ADI (Abstract Device Interface) pentru comunicare punct la punct.

Restul MPICH implementat pe ADI

- Gestiunea comunicatorilor
- Tipuri de date derivate
- Operatii colective

MPI-CCL (Collective Communication Library)

Orientata pe avantajele din retelele locale cu *difuzare*.

optimizează operatiile colective pentru NOW (Network Of Workstations).

Arhitectura pe două niveluri:

URTP (User-level Reliable Transport Protocol), oferă servicii de transport punct la punct si cu difuzare, sigure.

MPI-CCL oferă functionalitatea operatiilor colective ale MPI.

MPICH

Functii rutine **ADI**:

- Initializare si terminare
  - ADI MPID_Init, MPID_End
- Initializarea unei transmisii / receptii de mesaj,
  - MPID_Port_send (send_ready, send_sync),
  - MPID_Recv. . .
- Testul terminarii unei operatii sau a sosirii unui anume mesaj
  - MPID_Test_send, MPID_Test_recv, MPID_Iprobe
- Terminarea unei transmisii / receptii,
  - MPID_Complete_send, MPID_Complete_recv
- Anularea unei operatii
  - MPID_Cancel
- Verificarea unei operatii în asteptare
  - MPID_Check_device
- Aflarea unor informatii legate de rang sau numar de procese
  - MPID_Myrank, MPID_Mysize.

Implementarea ADI

- Ex folosind Fast Messages (Myrinet)
- Biblioteca FM are trei **primitive**:
  - FM_send_4 (dest, handler, i0,i1,i2,i3)
    - transmite un mesaj de 4 cuvinte
  - FM_send (dest, handler, buff, size)
    - transmite un mesaj lung
  - FM_extract ()
    - prelucrează mesaje primite

Implementare