Índice de contenido

Practica 1.- Introducción a MPI

Esta práctica pretende introducirnos en la librería MPI. Para ello lo que se pretende es lo siguiente:

1. En primer lugar se debe instalar MPI en los equipos. Para ello vamos a instalar la implementación MPICH en varias máquinas sobre las que realizaremos las prácticas.
2. A continuación se va a realizar la típica implementación de "hola mundo" donde se hagan uso de las funciones:
   a) "Hola mundo" desde el mismo procesador
      - MPI_Init
      - MPI_Comm_rank
      - MPI_Comm_size
      - MPI_Finalize
   b) "Hola Mundo" enviando mensaje de los procesadores al 0
      - MPI_Send
      - MPI_Recv
   c) "Hola mundo" desde el procesador 0 a todos los demás por difusión
      - MPI_Bcast
   d) "Hola mundo" desde el procesador 0 a todos los demás por dispersión (división del mensaje para todos)
      - MPI_Scatter
   e) Suma de los números de procesos (rank) de los procesadores que intervienen al proceso 0 por reducción (función de los resultados de todos)
      - MPI_Reduce
   f) "Hola mundo" desde todos los procesadores hacia el procesador 0 (concatenación del mensaje en el procesador 0)
      - MPI_Gather
3. A continuación vamos a realizar un programa que calcule la suma de n números almacenados en un vector. Para ello se puede utilizar las funciones probadas anteriormente.

Practica 2.- Búsqueda de errores en códigos con MPI

Esta práctica pretende resaltar posibles errores que podemos cometer a la hora de utilizar MPI para nuestros proyectos. Se trata de buscar el error y solucionarlo:

1. mpi_bug1.c
2. mpi_bug2.c
3. mpi_bug3.c
4. mpi_bug4.c
5. mpi_bug5.c
6. mpi_bug6.c
7. mpi_bug7.c

Para el código ver los anexos

Practica 3.- Multiplicación de Matrices con OpenMP

Esta práctica se pretende realizar la multiplicación de matrices de la forma A x B = C, siendo las dimensiones de las matrices de la forma A(x,y), B(y,z) y C(x,z)

Practica 4.- Problema libre y voluntario con MPI

Esta práctica será una práctica libre y voluntaria. Vosotros tenéis que pensar un problema, plantearmelo y solucionarlo.

NOTA: también tengo otros ejemplos interesantes en la carpeta otrasPracticas

ANEXOS

mpi_bug1.c

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

int main (int argc, char *argv[])
{
int numtasks, rank, dest, tag, source, rc, count;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("Task %d starting...\n",rank);

if (rank == 0) {
  if (numtasks > 2)
               printf("Numtasks=%d.    Only    2    needed.    Ignoring
extra...\n",numtasks);
  dest = rank + 1;
  source = dest;
  tag = rank;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  printf("Sent to task %d...\n",dest);
   rc  =  MPI_Recv(&inmsg,  1,  MPI_CHAR,  source,  tag,  MPI_COMM_WORLD,
&Stat);
  printf("Received from task %d...\n",source);
  }

else if (rank == 1) {
  dest = rank - 1;
  source = dest;
  tag = rank;
   rc  =  MPI_Recv(&inmsg,  1,  MPI_CHAR,  source,  tag,  MPI_COMM_WORLD,
&Stat);
  printf("Received from task %d...\n",source);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  printf("Sent to task %d...\n",dest);
  }

if (rank < 2) {
  rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
  printf("Task %d: Received %d char(s) from task %d with tag %d \n",
         rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
  }

MPI_Finalize();
}
```

mpi_bug2.c

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

int main (int argc, char *argv[])
{
int numtasks, rank, tag=1, alpha, i;
float beta;
MPI_Request reqs[10];
MPI_Status stats[10];

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

if (rank == 0) {
  if (numtasks > 2)
               printf("Numtasks=%d.    Only    2    needed.    Ignoring
extra...\n",numtasks);
  for (i=0; i<10; i++) {
    alpha = i*10;
    MPI_Isend(&alpha, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &reqs[i]);
    MPI_Wait(&reqs[i], &stats[i]);
    printf("Task %d sent = %d\n",rank,alpha);
    }
  }

if (rank == 1) {
  for (i=0; i<10; i++) {
    MPI_Irecv(&beta, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &reqs[i]);
    MPI_Wait(&reqs[i], &stats[i]);
    printf("Task %d received = %f\n",rank,beta);
    }
  }

MPI_Finalize();
}
```

mpi_bug3.c

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define  ARRAYSIZE      16000000
#define  MASTER         0

float  data[ARRAYSIZE];

int main (int argc, char *argv[])
{
int   numtasks, taskid, rc, dest, offset, i, j, tag1,
      tag2, source, chunksize;
float mysum, sum;
float update(int myoffset, int chunk, int myid);
MPI_Status status;

/***** Initializations *****/
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
if (numtasks % 4 != 0) {
   printf("Quitting. Number of MPI tasks must be divisible by 4.\n");
   MPI_Abort(MPI_COMM_WORLD, rc);
   exit(0);
   }
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
printf ("MPI task %d has started...\n", taskid);
chunksize = (ARRAYSIZE / numtasks);
tag2 = 1;
tag1 = 2;

/***** Master task only ******/
if (taskid == MASTER){

  /* Initialize the array */
  sum = 0;
  for(i=0; i<ARRAYSIZE; i++) {
    data[i] =  i * 1.0;
    sum = sum + data[i];
    }
  printf("Initialized array sum = %e\n",sum);

   /* Send each task its portion of the array - master keeps 1st part
*/
  offset = chunksize;
  for (dest=1; dest<numtasks; dest++) {
    MPI_Send(&offset, 1, MPI_INT, dest, tag1, MPI_COMM_WORLD);
         MPI_Send(&data[offset],  chunksize,  MPI_FLOAT,  dest,  tag2,
MPI_COMM_WORLD);
             printf("Sent   %d   elements   to   task   %d   offset=
%d\n",chunksize,dest,offset);
    offset = offset + chunksize;
    }

  /* Master does its part of the work */
  offset = 0;
  mysum = update(offset, chunksize, taskid);

  /* Wait to receive results from each task */
  for (i=1; i<numtasks; i++) {
```

```
    source = i;
       MPI_Recv(&offset,  1,  MPI_INT,  source,  tag1,  MPI_COMM_WORLD,
&status);
    MPI_Recv(&data[offset], chunksize, MPI_FLOAT, source, tag2,
      MPI_COMM_WORLD, &status);
    }

  /* Get final sum and print sample results */
      MPI_Reduce(&mysum,   &sum,   1,   MPI_FLOAT,   MPI_SUM,   MASTER,
MPI_COMM_WORLD);
  printf("Sample results: \n");
  offset = 0;
  for (i=0; i<numtasks; i++) {
    for (j=0; j<5; j++)
      printf("  %e",data[offset+j]);
    printf("\n");
    offset = offset + chunksize;
    }
  printf("*** Final sum= %e ***\n",sum);

  }  /* end of master section */



/***** Non-master tasks only *****/

if (taskid > MASTER) {

  /* Receive my portion of array from the master task */
  source = MASTER;
     MPI_Recv(&offset,  1,  MPI_INT,  source,  tag1,  MPI_COMM_WORLD,
&status);
  MPI_Recv(&data[offset], chunksize, MPI_FLOAT, source, tag2,
    MPI_COMM_WORLD, &status);

  mysum = update(offset, chunksize, taskid);

  /* Send my results back to the master task */
  dest = MASTER;
  MPI_Send(&offset, 1, MPI_INT, dest, tag1, MPI_COMM_WORLD);
     MPI_Send(&data[offset],  chunksize,  MPI_FLOAT,  MASTER,  tag2,
MPI_COMM_WORLD);

      MPI_Reduce(&mysum,   &sum,   1,   MPI_FLOAT,   MPI_SUM,   MASTER,
MPI_COMM_WORLD);

  } /* end of non-master */



}   /* end of main */


float update(int myoffset, int chunk, int myid) {
  int i;
  float mysum;
  /* Perform addition to each of my array elements and keep my sum */
  mysum = 0;
  for(i=myoffset; i < myoffset + chunk; i++) {
    data[i] = data[i] + i * 1.0;
```

```
    mysum = mysum + data[i];
    }
  printf("Task %d mysum = %e\n",myid,mysum);
  return(mysum);
  }
```

mpi_bug4.c

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define  ARRAYSIZE      16000000
#define  MASTER         0

float  data[ARRAYSIZE];

int main (int argc, char *argv[])
{
int   numtasks, taskid, rc, dest, offset, i, j, tag1,
      tag2, source, chunksize;
float mysum, sum;
float update(int myoffset, int chunk, int myid);
MPI_Status status;

/***** Initializations *****/
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
if (numtasks % 4 != 0) {
   printf("Quitting. Number of MPI tasks must be divisible by 4.\n");
   MPI_Abort(MPI_COMM_WORLD, rc);
   exit(0);
   }
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
printf ("MPI task %d has started...\n", taskid);
chunksize = (ARRAYSIZE / numtasks);
tag2 = 1;
tag1 = 2;

/***** Master task only ******/
if (taskid == MASTER){

  /* Initialize the array */
  sum = 0;
  for(i=0; i<ARRAYSIZE; i++) {
    data[i] =  i * 1.0;
    sum = sum + data[i];
    }
  printf("Initialized array sum = %e\n",sum);

   /* Send each task its portion of the array - master keeps 1st part
*/
  offset = chunksize;
  for (dest=1; dest<numtasks; dest++) {
    MPI_Send(&offset, 1, MPI_INT, dest, tag1, MPI_COMM_WORLD);
         MPI_Send(&data[offset],  chunksize,  MPI_FLOAT,  dest,  tag2,
MPI_COMM_WORLD);
             printf("Sent   %d   elements   to   task   %d   offset=
%d\n",chunksize,dest,offset);
    offset = offset + chunksize;
    }

  /* Master does its part of the work */
  offset = 0;
  mysum = update(offset, chunksize, taskid);

  /* Wait to receive results from each task */
```

```
  for (i=1; i<numtasks; i++) {
    source = i;
       MPI_Recv(&offset,  1,  MPI_INT,  source,  tag1,  MPI_COMM_WORLD,
&status);
    MPI_Recv(&data[offset], chunksize, MPI_FLOAT, source, tag2,
      MPI_COMM_WORLD, &status);
    }

  /* Get final sum and print sample results */
  printf("Sample results: \n");
  offset = 0;
  for (i=0; i<numtasks; i++) {
    for (j=0; j<5; j++)
      printf("  %e",data[offset+j]);
    printf("\n");
    offset = offset + chunksize;
    }
  printf("*** Final sum= %e ***\n",sum);

  }  /* end of master section */



/***** Non-master tasks only *****/

if (taskid > MASTER) {

  /* Receive my portion of array from the master task */
  source = MASTER;
     MPI_Recv(&offset,  1,  MPI_INT,  source,  tag1,  MPI_COMM_WORLD,
&status);
  MPI_Recv(&data[offset], chunksize, MPI_FLOAT, source, tag2,
    MPI_COMM_WORLD, &status);

  mysum = update(offset, chunksize, taskid);

  /* Send my results back to the master task */
  dest = MASTER;
  MPI_Send(&offset, 1, MPI_INT, dest, tag1, MPI_COMM_WORLD);
     MPI_Send(&data[offset],  chunksize,  MPI_FLOAT,  MASTER,  tag2,
MPI_COMM_WORLD);

      MPI_Reduce(&mysum,  &sum,  1,  MPI_FLOAT,  MPI_SUM,  MASTER,
MPI_COMM_WORLD);

  } /* end of non-master */


MPI_Finalize();

}   /* end of main */


float update(int myoffset, int chunk, int myid) {
  int i;
  float mysum;
  /* Perform addition to each of my array elements and keep my sum */
  mysum = 0;
  for(i=myoffset; i < myoffset + chunk; i++) {
    data[i] = data[i] + i * 1.0;
```

```
    mysum = mysum + data[i];
    }
  printf("Task %d mysum = %e\n",myid,mysum);
  return(mysum);
  }
```

mpi_bug5.c

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

#define MSGSIZE 2000

int main (int argc, char *argv[])
{
int        numtasks, rank, i, tag=111, dest=1, source=0, count=0;
char       data[MSGSIZE];
double     start, end, result;
MPI_Status status;

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

if (rank == 0) {
  printf ("mpi_bug5 has started...\n");
  if (numtasks > 2)
       printf("INFO:  Number  of  tasks=  %d.  Only  using  2  tasks.\n",
numtasks);
  }

/******************************                Send                task
*******************************/
if (rank == 0) {

  /* Initialize send data */
  for(i=0; i<MSGSIZE; i++)
     data[i] =  'x';

  start = MPI_Wtime();
  while (1) {
    MPI_Send(data, MSGSIZE, MPI_BYTE, dest, tag, MPI_COMM_WORLD);
    count++;
    if (count % 10 == 0) {
      end = MPI_Wtime();
      printf("Count= %d  Time= %f sec.\n", count, end-start);
      start = MPI_Wtime();
      }
    }
  }

/******************************              Receive               task
*******************************/

if (rank == 1) {
  while (1) {
      MPI_Recv(data,  MSGSIZE,  MPI_BYTE,  source,  tag,  MPI_COMM_WORLD,
&status);
    /* Do some work  - at least more than the send task */
    result = 0.0;
    for (i=0; i < 1000000; i++)
      result = result + (double)random();
    }
  }
```

```
MPI_Finalize();
}
```

mpi_bug6.c

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

#define COMM MPI_COMM_WORLD
#define REPS 1000
#define DISP 100

int main (int argc, char *argv[])
{
int numtasks, rank, buf, tag1=1, i, rc, dest, src, offset, nreqs;
double T1, T2;
MPI_Request reqs[REPS*2];
MPI_Status stats[REPS*2];

MPI_Init(&argc,&argv);
MPI_Comm_size(COMM, &numtasks);
MPI_Comm_rank(COMM, &rank);

/* Require 4 tasks */
if (rank == 0 ) {
  if (numtasks != 4) {
    printf("ERROR: Number of tasks must be 4. Quitting.\n");
    MPI_Abort(COMM, rc);
    }
  printf("Starting isend/irecv send/irecv test...\n");
  }

/* Use barriers for clean output */
MPI_Barrier(COMM);
printf("Task %d starting...\n", rank);
MPI_Barrier(COMM);

T1 = MPI_Wtime();     /* start the clock */

/* Tasks 0 and 1 do the isend/irecv test.
*  Determine who to send/receive with. nreqs specifies how many non-
blocking
*   operation request handles to capture. offset is where the task
should
*    store each request as it is captured in the reqs() array.
*/
if (rank < 2) {
  nreqs = REPS*2;
  if (rank == 0) {
    src = 1;
    offset = 0;
    }
  if (rank == 1) {
    src = 0;
    offset = REPS;
    }
  dest = src;

/* Do the non-blocking send and receive operations */
  for (i=0; i<REPS; i++) {
    MPI_Isend(&rank, 1, MPI_INT, dest, tag1, COMM, &reqs[offset]);
    MPI_Irecv(&buf, 1, MPI_INT, src, tag1, COMM, &reqs[offset+1]);
```

```
    offset += 2;
    if ((i+1)%DISP == 0)
      printf("Task %d has done %d isends/irecvs\n", rank, i+1);
    }
  }

/* Tasks 2 and 3 do the send/irecv test.
    Determine who to send/receive with. nreqs specifies how many non-
blocking
     operation request handles to capture. offset is where the task
should
   store each request as it is captured in the reqs() array.  */
if (rank > 1) {
  nreqs = REPS;

/* Task 2 does the blocking send operation */
  if (rank == 2) {
    dest = 3;
    for (i=0; i<REPS; i++) {
      MPI_Send(&rank, 1, MPI_INT, dest, tag1, COMM);
      if ((i+1)%DISP == 0)
        printf("Task %d has done %d sends\n", rank, i+1);
      }
    }

/* Task 3 does the non-blocking receive operation */
  if (rank == 3) {
    src = 2;
    offset = 0;
    for (i=0; i<REPS; i++) {
      MPI_Irecv(&buf, 1, MPI_INT, src, tag1, COMM, &reqs[offset]);
      offset += 1;
      if ((i+1)%DISP == 0)
        printf("Task %d has done %d irecvs\n", rank, i+1);
      }
    }

  }

/* Wait for all non-blocking operations to complete and record time */
MPI_Waitall(nreqs, reqs, stats);
T2 = MPI_Wtime();     /* end time */
MPI_Barrier(COMM);

printf("Task %d time(wall)= %lf sec\n", rank, T2-T1);

MPI_Finalize();
}
```

mpi_bug7.c

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

int main (int argc, char *argv[])
{
int   numtasks, taskid, len, buffer, root, count;
char hostname[MPI_MAX_PROCESSOR_NAME];

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Get_processor_name(hostname, &len);

printf ("Task %d on %s starting...\n", taskid, hostname);
buffer = 23;
root = 0;
count = taskid;
if (taskid == root)
   printf("Root: Number of MPI tasks is: %d\n",numtasks);

MPI_Bcast(&buffer, count, MPI_INT, root, MPI_COMM_WORLD);


MPI_Finalize();

}
```