PROGRAMACIÓN PARALELA

Modelos de programación paralela
Programación en memoria distribuida: MPI

- MPI-1.1: www.mpi.org
- www.mpich.org
- Curso de Wilkinson
- Curso de Dongarra
- Curso de Francisco Almeida

¿Qué es MPI?

Previamente PVM: Parallel Virtual Machine.

MPI: Message Passing Interface.

Una especificación para paso de mensajes.

La primera librería de paso de mensajes estándar y portable.

Por consenso MPI Forum. Participantes de unas 40 organizaciones.

Acabado y publicado en mayo 1994. Actualizado en junio 1995.

MPI2, HMPI

¿Qué ofrece MPI?

Estandarización.

Portabilidad: multiprocesadores, multicomputadores, redes, heterogéneos, ...

Buenas prestaciones, ..., si está disponible para el sistema.

Amplia funcionalidad.

Implementaciones libres (mpich, lam, ...)

MPI hello.c

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
main(int argc, char*argv[]) {

int name, p, source, dest, tag = 0;
char message[100];
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&name);
MPI_Comm_size(MPI_COMM_WORLD,&p);
```

```
if (name != 0) {
   printf("Processor %d of %d\n",name, p);
   sprintf(message,"greetings from process %d!",
            name);
   dest = 0;
   MPI_Send(message, strlen(message)+1,
            MPI_CHAR, dest, tag,
            MPI_COMM_WORLD);
 }  else {
   printf("processor 0, p = %d ",p);
   for(source=1; source < p; source++) {
     MPI_Recv(message,100, MPI_CHAR, source,
          tag,  MPI_COMM_WORLD, &status);
     printf("%s\n",message);
   }
 }
 MPI_Finalize();
}
```

```
Processor 2 of 4
Processor 3 of 4
Processor 1 of 4
processor 0, p = 4 greetings
from process 1!
greetings from process 2!
greetings from process 3!
```

```
mpicc –o hello hello.c

mpirun –np 4 hello
```

mpich

- Compilación:

mpicc programa.c

- Ejecución:

mpirun –np numpro -machinefile fichmaq programa

- **Fichero cabecera:**

#include <mpi.h>

- **Formato de las funciones:**

error=MPI_nombre(parámetros ...)

- **Inicialización:**

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

- **Comunicador:**

Conjunto de procesos en que se hacen comunicaciones.

MPI_COMM_WORD , el mundo de los procesos MPI

- **Identificación de procesos:**

MPI_Comm_rank (MPI_Comm comm , int *rank)

- **Procesos en el comunicador:**

MPI_Comm_size (MPI_Comm comm , int *size)

- **Finalización:**

int MPI_Finalize ( )

MENSAJE

- **Formado por un cierto número de elementos de un tipo MPI.**

- **Tipos MPI:**

  Básicos:

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_SHOT	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	

  Derivados: los construye el programador.

- **Envío:**

```
int MPI_Send ( void  *buffer , int contador ,
    MPI_Datatype  tipo , int destino , int tag ,
    MPI_Comm  comunicador )
```

MPI_ANY_TAG

- **Recepción:**

```
int MPI_Recv ( void  *buffer , int contador ,
    MPI_Datatype  tipo , int origen , int tag ,
    MPI_Comm  comunicador , MPI_Status  *estado)
```

MPI_ANY_SOURCE

Tipos de comunicación

• Envío síncrono: MPI_Ssend
Acaba cuando la recepción empieza.

• Envío con buffer: MPI_Bsend
Acaba siempre, independiente del receptor.

• Envío estándar: MPI_Send
Síncrono o con buffer.

• Envío “ready”: MPI_Rsend
Acaba independiente de que acabe la recepción.

• Recepción: MPI_Recv
Acaba cuando se ha recibido un mensaje.

Comunicación asíncrona

MPI_Isend(buf, count, datatype, dest, tag, comm, request)

MPI_Irecv(buf, count, datatype, source, tag, comm, request)

request para saber si la operación ha acabado

MPI_Wait( ) vuelve si la operación se ha completado, espera hasta que se completa.

MPI_Test( ) devuelve un flag diciendo si la operación se ha completado.

Regla del trapecio

p procesos

se divide el intervalo [a,b] en *n* subintervalos

cada proceso debe saber:
- número total de procesos (MPI_Comm_size)
- identificador de proceso (MPI_Comm_rank)
- intervalo de integración
- número de subintervalos

I/0 por el proceso 0
- lee datos
- los envía
- recibe resultados parciales

```
main(int argc,char **argv)
{   int  my_rank , p , n , local_n , source , dest=0 , tag=50;
    float a , b ,  h , local_a , local_b, integral , total;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    Get_data(my_rank,p,&a,&b,&n);
    h=(b-a)/n ; local_n=n/p;
    local_a=a+my_rank*local_n*h ; local_b=local_a+local_n*h;
    integral=Trap(local_a,local_b,local_n,h);
    if(my_rank==0) {
        total=integral;
        for(source=1;source<p;source++) {
            MPI_Recv(&integral,1,MPI_FLOAT,source,tag,MPI_COMM_WORLD,&status);
           total+=integral;
        };
        printf("With n= %d trapezoides\n la estimacion",n);
        printf("de la integral entre %f y %f\n es= %f \n",a,b,total);
    } else {
        MPI_Send(&integral,1,MPI_FLOAT,dest,tag,MPI_COMM_WORLD);
    };
    MPI_Finalize();
}
```

```
float f(float x)
{
  float return_val;
  ...
  return return_val;
}

float Trap(float local_a,float local_b,int local_n,float h)
{
  float integral;
  float x;
  int i;

  integral=(f(local_a)+f(local_b))/2.0;
  x=local_a;
  for(i=1;i<=local_n-1;i++)
  {
    x+=h;
    integral+=f(x);
  };
  integral*=h;
  return integral;
}
```

```
void Get_data(int my_rank,int p,float *a_ptr,float *b_ptr,int *n_ptr)
{   int source=0 , dest , tag;
    MPI_Status status;
    if(my_rank==0)
    {
        printf("Enter a, b, and n\n");
        scanf("%f %f %d",a_ptr,b_ptr,n_ptr);
        for(dest=1;dest<p;dest++) {
            tag=30;
            MPI_Send(a_ptr,1,MPI_FLOAT,dest,tag,MPI_COMM_WORLD);
            tag=31;
            MPI_Send(b_ptr,1,MPI_FLOAT,dest,tag,MPI_COMM_WORLD);
            tag=32;
            MPI_Send(n_ptr,1,MPI_INT,dest,tag,MPI_COMM_WORLD);
        };
    } else {
        tag=30;
        MPI_Recv(a_ptr,1,MPI_FLOAT,source,tag,MPI_COMM_WORLD,&status);
        tag=31;
        MPI_Recv(b_ptr,1,MPI_FLOAT,source,tag,MPI_COMM_WORLD,&status);
        tag=32;
        MPI_Recv(n_ptr,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
    };
}
```

Coste envío:

$3(p-1)(t_s+t_w)$

Comunicaciones colectivas

MPI_Barrier( ) bloquea los procesos hasta que la llaman todos

MPI_Bcast( ) broadcast del proceso raíz a todos los demás

MPI_Gather( ) recibe valores de un grupo de procesos

MPI_Scatter( ) distribuye un buffer en partes a un grupo de procesos

MPI_Alltoall( ) envía datos de todos los procesos a todos

MPI_Reduce( ) combina valores de todos los procesos

MPI_Reduce_scatter( ) combina valores de todos los procesos y distribuye

MPI_Scan( ) reducción prefija (0,...,i-1 a i)

MPI Collective Operations

MPI Operator	Operation
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bitwise and
MPI_LOR	logical or
MPI_BOR	bitwise or
MPI_LXOR	logical exclusive or
MPI_BXOR	bitwise exclusive or
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

```
void Get_data(int my_rank,float *a_ptr,float *b_ptr,int *n_ptr)
{
  int root=0;
  int count=1;

  if(my_rank==0)
  {
    printf("Enter a, b y n\n");
    scanf("%f %f %d",a_ptr,b_ptr,n_ptr);
  };
  MPI_Bcast(a_ptr,1,MPI_FLOAT,root,MPI_COMM_WORLD);
  MPI_Bcast(b_ptr,1,MPI_FLOAT,root,MPI_COMM_WORLD);
  MPI_Bcast(n_ptr,1,MPI_FLOAT,root,MPI_COMM_WORLD);
}
```

Coste envío:

$3(t_{sb}+t_{wb})$

Agrupamiento de datos

- Con contador en rutinas de envío y recepción agrupar los tres datos a enviar

- Con tipos derivados

- Con empaquetamiento

Coste envío:

$(p-1)(t_s+t_w)$

Tipos derivados

- Se crean en tiempo de ejecución

- Se especifica la disposición de los datos en el tipo:

```
Int MPI_Type_Struct(    int count,
                        int *array_of_block_lengths,
                         MPI_Aint  *array_of_displacements,
      MPI_Datatype *array_of_types,
                        MPI_Datatype *newtype )
```

- Se pueden construir tipos de manera recursiva

- La creación de tipos requiere trabajo adicional

```
void Build_derived_type(INDATA_TYPE *indata,MPI_Datatype *message_type_ptr)
{   int block_lenghts[3];
    MPI_Aint displacements[3];
    MPI_Aint addresses[4];
    MPI_Datatype typelist[3];

    typelist[0]=MPI_FLOAT;  typelist[1]=MPI_FLOAT;  typelist[2]=MPI_INT;

    block_lenghts[0]=1;  block_lenghts[1]=1;  block_lenghts[2]=1;

    MPI_Address(indata,&addresses[0]);
    MPI_Address(&(indata->a),&addresses[1]);
    MPI_Address(&(indata->b),&addresses[2]);
    MPI_Address(&(indata->n),&addresses[3]);

    displacements[0]=addresses[1]-addresses[0];
    displacements[1]=addresses[2]-addresses[0];
    displacements[2]=addresses[3]-addresses[0];

    MPI_Type_struct(3,block_lenghts,displacements,typelist,message_type_ptr);

    MPI_Type_commit(message_type_ptr);
}
```

• Hay que llamar a la rutina

MPI_Type_commit

antes de poder usar message_type_ptr como un tipo de MPI.

• Para calcular las direcciones del parámetro indata (suponemos que los parámetros a, b y n están en una estructura indata) se usa la rutina

MPI_Address

y no se calcula la dirección como en C.

```
void Get_data(INDATA_TYPE *indata , int my_rank)
{
  MPI_Datatype message_type;
  int root=0;
  int count=1;

  if(my_rank==0)
  {
    printf("Enter a, b y n\n");
    scanf("%f %f %d",&(indata->a),&(indata->b),&(indata->n));
  };

  Build_derived_type(indata,&message_type);
  MPI_Bcast(indata,count,message_type,root,MPI_COMM_WORLD);
}
```

Tipos derivados

• Si los datos que constituyen el nuevo tipo son un subconjunto de entradas hay mecanismos especiales para construirlos:

```
int MPI_Type_contiguous( int count,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```

crea un tipo derivado formado por count elementos del tipo oldtype contiguos en memoria.

Tipos derivados

```
int MPI_Type_vector( int count,
    int block_lenght,
    int stride,
    MPI_Datatype element_type,
    MPI_Datatype *newtype)
```

crea un tipo derivado formado por count elementos, cada uno de ellos con block_lenght elementos del tipo element_type. stride es el número de elementos del tipo element_type entre elementos sucesivos del tipo new_type. De este modo, los elementos pueden ser entradas igualmente espaciadas en un array.

Tipos derivados

```
int MPI_Type_indexed( int count,
    int *array_of_block_lengths,
    int *array_of_displacements,
    MPI_Datatype element_type,
    MPI_Datatype *newtype)
```

crea un tipo derivado con count elementos, habiendo en cada elemento array_of_block_lengths[i] entradas de tipo element_type, y el desplazamiento array_of_displacements[i] unidades de tipo element_type desde el comienzo de newtype.

Empaquetamiento

Los datos se pueden empaquetar para ser enviados, y desempaquetarse tras ser recibidos:

```
int MPI_Pack( void *pack_data, int in_count,
    MPI_Datatype datatype, void *buffer, int size,
    int *position_ptr, MPI_Comm comm)}
```

Se empaquetan in_count datos de tipo datatype, y pack_data referencia los datos a empaquetar en el buffer, que debe consistir de size bytes (puede ser una cantidad mayor a la que se va a ocupar). El parámetro position_ptr es de entrada y salida. Como entrada, el dato se copia en la posición buffer+*position_ptr. Como salida, referencia la siguiente posición en el buffer después del dato empaquetado. De esta manera, el cálculo de dónde se sitúa en el buffer el siguiente elemento a empaquetar lo hace MPI automáticamente.

Empaquetamiento

```
int MPI_Unpack( void *buffer,
    int size,
    int *position_ptr,
    void *unpack_data,
    int count,
    MPI_Datatype datatype,
    MPI_Comm comm)
```

Copia count elementos de tipo datatype en unpack_data, tomándolos de la posición buffer+*position_ptr del buffer. Hay que decir el tamaño del buffer (size) en bytes, y position_ptr es manejado por MPI de manera similar a como lo hace en MPI_Pack.

```
void Get_data(int my_rank,float *a_ptr,float *b_ptr,int *n_ptr)
{   int root=0 ; char buffer[100] ; int position;

    if(my_rank==0) {
       printf("Enter a, b y n\n");
       scanf("%f %f %d",a_ptr,b_ptr,n_ptr);

       position=0;
       MPI_Pack(a_ptr,1,MPI_FLOAT,buffer,100,&position,MPI_COMM_WORLD);
       MPI_Pack(b_ptr,1,MPI_FLOAT,buffer,100,&position,MPI_COMM_WORLD);
       MPI_Pack(n_ptr,1,MPI_INT,buffer,100,&position,MPI_COMM_WORLD);

       MPI_Bcast(buffer,100,MPI_PACKED,root,MPI_COMM_WORLD);
    } else {
       MPI_Bcast(buffer,100,MPI_PACKED,root,MPI_COMM_WORLD);

       position=0;
       MPI_Unpack(buffer,100,&position,a_ptr,1,MPI_FLOAT,MPI_COMM_WORLD);
       MPI_Unpack(buffer,100,&position,b_ptr,1,MPI_FLOAT,MPI_COMM_WORLD);
       MPI_Unpack(buffer,100,&position,n_ptr,1,MPI_INT,MPI_COMM_WORLD);
    };
}
```

Comunicadores

• MPI_COMM_WORLD incluye a todos los procesos

• Se puede definir comunicadores con un número menor de procesos: para comunicar datos en cada fila de procesos en la malla, ...

• Dos tipos de comunicadores:

- intra-comunicadores, se utilizan para enviar mensajes entre los procesos en ese comunicador,
- inter-comunicadores, se utilizan para enviar mensajes entre procesos en distintos comunicadores.

Consta de:

- un grupo, que es una colección ordenada de procesos a los que se asocia identificadores entre 0 y p-1
- un contexto, que es un identificador que asocia el sistema al grupo. Adicionalmente, a un comunicador se le puede asociar una topología virtual.

Comunicadores

• Creación de un comunicador cuyos procesos son los de la primera fila de nuestra malla virtual.

MPI_COMM_WORLD consta de p=q^2 procesos agrupados en q filas y columnas. El proceso número x tiene las coordenadas (x div q,x mod q).

```
MPI_Group  MPI_GROUP_WORLD;
MPI_Group  first_row_group;
MPI_Comm  first_row_comm;
int  row_size;
int  *process_ranks;

process_ranks=(int *) malloc(q*sizeof(int));

for(proc=0;proc<q;proc++)
    process_ranks[proc]=proc;

MPI_Comm_group(MPI_COMM_WORLD,&MPI_GROUP_WORLD);

MPI_Group_incl(MPI_GROUP_WORLD,q,process_ranks,&first_row_group);

MPI_Comm_create(MPI_COMM_WORLD,first_row_group,&first_row_comm);
```

Comunicadores

• Las rutinas MPI_Comm_group y MPI_Group_incl son locales y no hay comunicaciones,

• MPI_Comm_create es una operación colectiva, y todos los procesos en old_comm deben ejecutarla aunque no vayan a formar parte del nuevo grupo.

Comunicadores

• Para crear varios comunicadores disjuntos

int MPI_Comm_split(MPI_Comm old_comm, int split_key,
int rank_key, MPI_Comm *new_comm)

crea un nuevo comunicador para cada valor de split_key.
Los procesos con el mismo valor de split_key un grupo.
Si dos procesos a y b tienen el mismo valor de split_key y el rank_key de a es menor que el de b, en el nuevo grupo a tiene identificador menor que b.
Si los dos tienen el mismo rank_key el sistema asigna los identificadores arbitrariamente.
Es una operación colectiva. Todos los procesos en el comunicador deben llamarla. Los procesos que no se quiere incluir en ningún nuevo comunicador pueden utilizar el valor MPI_UNDEFINDED en rank_key, con lo que el valor de retorno de new_comm es MPI_COMM_NULL.

Crear q grupos de procesos asociados a las q filas:

```
MPI_Comm my_row_comm;
int my_row;
my_row=my_rank/q;
MPI_Comm_split(MPI_COMM_WORLD,my_row,my_rank,&my_row_comm);
```

Topologías

- A un grupo se le puede asociar una topología virtual:
  topología de grafo en general,
  de malla o cartesiana.

- Una topología cartesiana se crea:

```
int MPI_Card_create( MPI_Comm  old_comm,
     int  number_of_dims, int *dim_sizes, int *periods,
     int reorder, MPI_Comm *cart_comm)
```

El número de dimensiones de la malla es number_of_dims,
el número de procesos en cada dimensión está en dim_sizes.
Con periods se indica si cada dimensión es circular o lineal.
Un valor de 1 en reorder indica al sistema que se reordenen los procesos para optimizar la relación entre el sistema físico y el lógico.

Topologías

• Las coordenadas de un proceso conocido su identificador se obtienen con

```
int MPI_Cart_coords( MPI_Comm comm, int rank,
    int number_of_dims, int *coordinates)
```

• El identificador conocidas las coordenadas con

```
int MPI_Cart_rank( MPI_Comm comm, int *coordinates, int *rank)
```

• Una malla se puede particionar en mallas de menor dimensión

```
int MPI_Cart_sub(MPI_Comm old_comm, int *varying_coords,
    MPI_Comm *new_comm)
```

en varying_coords se indica para cada dimensión si pertenece al nuevo comunicador.

Si varying_coords[0]=0 y varying_coords[1]=1 para obtener el nuevo comunicador no se varía la primera dimensión pero sí la segunda. Se crean q comunicadores, uno por cada fila.

Es colectiva.