MPI

Jesús Sánchez Cuadrado

Resumen. En este documento se describen algunas técnicas de programación en memoria distribuida y cómo implementarlas con MPI. Los objetivos principales de este capítulo son:

- Compilar y ejecutar programas MPI
- Utilización de funciones de envío y recepción de mensajes
- Funciones de comunicación colectiva
- Empaquetado de datos

Índice

1. Introducción **1**
1.1. ¿Qué es MPI? . . . 1
1.2. Características de MPI . . . 1

2. Utilización básica de MPI **2**
2.1. Compilación . . . 2
2.2. Ejecución . . . 2
2.3. Ejemplo inicial . . . 2

3. Conceptos básicos **3**
3.1. Comunicadores . . . 3
3.2. Otras funciones útiles . . . 4
3.3. Envio y recepción de mensajes . . . 4
3.4. Comunicación punto a punto . . . 4
3.4.1. Funciones . . . 5
3.4.2. Condiciones de recepción de mensajes . . . 6
3.4.3. Utilización de MPI_ANY_SOURCE . . . 7
3.4.4. Utilización de MPI_STATUS . . . 7
3.5. Otros tipos de comunicaciones . . . 7
3.6. Comunicaciones no bloqueantes . . . 8

4. Comunicaciones colectivas **8**
4.1. Broadcast . . . 9
4.2. Scattering . . . 9
4.3. Gather . . . 11
4.4. Reduction . . . 12
4.5. All-to-all . . . 12
4.6. Barreras . . . 13

5. Empaquetamiento y tipos derivados **13**
5.1. Empaquetamiento 13
5.2. Tipos derivados 15

6. Ejercicios **15**

7. Bibliografía comentada **15**

1. Introducción

1.1. ¿Qué es MPI?

MPI (Message Passing Interface) es un estándar industrial para programación paralela con paso de mensajes. El objetivo de MPI es definir una librería estándar, portable, eficiente y flexible que pueda ser implementada por diferentes entidades y para diferentes arquitecturas. La última versión del estándar y documentación adicional está disponible en `https://www.mpi-forum.org/`.

Algunas razones por las que MPI resulta relevante son las siguientes:

- Estandarización. MPI es la única librería de paso de mensajes que puede considerarse estándar, ya que hay implementaciones para cualquier plataforma de HPC.
- Portabilidad. MPI está diseñada para que el mismo programa pueda compilarse para plataformas diferentes sin ningún cambio.
- Rendimiento. Las diferentes implementaciones implementan la librería de manera diferente para optimizarla de acuerdo con las características de la plataforma destino.
- Funcionalidad. MPI-3 ofrece 430 funciones diferentes, cubriendo un amplio espectro de escenarios.

1.2. Características de MPI

MPI proporciona un modelo de programación basado en procesos que intercambian mensajes entre ellos. En este modelo la programación se realiza de manera independiente a cómo se "mapearán" los procesos a los nodos físicos de procesamiento. Así, es posible ejecutar un programa MPI en una máquina multi-core o en un cluster heterogéneo sin necesidad de modificar el código. La configuración concreta sobre cómo se debe ejecutar se realiza a través de la línea de comandos o con un fichero de configuración. Así, MPI ofrece una librería (para compilación) y un entorno de ejecución (a través de un comando para inicial la ejecución: `mpirun`) que se encargan de proporcionar las siguientes características [1, p.241–242]:

- **Identificación de procesos**. A cada proceso se le asigna un número no negativo como identificador (en MPI al identificador se denomina *rank*). Convencionalmente el identificador `0` se considera el proceso maestro.
- **Enrutamiento de mensajes**. MPI se encarga de enviar y recibir los mensajes que se produzcan durante el programa. Dependiendo del tipo de máquina física en la que se ejecute el programa se usarán unas estrategias u otras. Por ejemplo, se pueden usar sockets para comunicación entre máquinas diferentes o memoria compartida si los procesos se están ejecutando en la misma máquina.
- **Buffering de mensajes**. MPI se encarga de gestionar el buffering de los mensajes, aunque ofrece diferentes opciones para su personalización.
- **Representación de datos**. MPI ofrece una solución para construir programas paralelos interoperables, en el sentido que pueden funcionar comunicando máquinas con arquitecturas heterogéneas. Para ello MPI define tipos e datos estándar y hay que especificar explícitamente el tipo de los mensajes, de manera que sea posible realizar la conversión de datos si es necesario.

- **Diferentes sistemas paralelos**. MPI es independiente de la arquitectura en que se ejecute el programa. Un mismo programa puede ejecutarse en un cluster de ordenadores utilizando la red para realizar la comunicación entre los nodos. Igualmente, podría ejecutarse en un solo ordenador (ej., un multi-core) utilizando buffers de memoria compartida para realizar la comunicación.

Modelo SPMD (Single-Program Multiple-Data). La programación con paso de mensajes en MPI utiliza el model SPMD. Esto significa que el mismo programa se ejecuta en todos los nodos de procesamiento, y debe realizarse algún tipo de particionamiento de los datos para que cada nodo actue sobre datos diferentes (consiguiendo así el paralelismo). En la práctica, los programas tienen algún tipo de mecanismo de control (normalmente sentencias `if-else`) para que cada nodo ejecute secciones diferentes. La figura 1 muestra un esquema de funcionamento de este modelo. Cuando se inicia la ejecución con `mpirun` se crean tantas copias del programa como procesos se especifiquen. Esto significa que todos los nodos ejecutan el mismo programa. Antes de iniciar la ejecución del programa, si es necesario, éste se copia en cada uno de los nodos para su ejecución. Para poder hacer trabajo diferente en cada nodo de procesamiento el programador debe programarlo explícitamente, habitualmente a través de sentencias condicionales que comprueba el identificador del proceso y reparte el trabajo en función de eso.

Paso de mensajes

El modelo SPMD no es el único que se puede utilizar en paso de mensajes.
¿Qué otros lenguajes o sistemas de programación por paso de mensajes que no sean SPMD?

```
int id;
MPI_Comm_rank(MPI_COMM_WORLD, &id);
if (id == 0) {
 // Dar trabajo a proceso 0
} else if (id == 1 ) {
 // Dar trabajo a proceso 1
} else if (id == 2) {
 // Dar trabajo a proceso 2
} else {
 // Dar trabajo a proceso 3
}
```

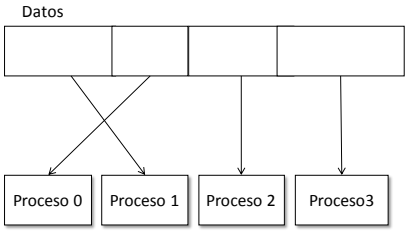

Figura 1: Esquema de funcionamiento del modelo SPMD

2. Utilización básica de MPI

2.1. Compilación

MPI es una librería de paso de mensajes que está disponible en multitud de plataformas. No requiere ningún tipo de soporte por parte del compilador.

El comando `mpicc` es un *wrapper* de `gcc` que facilita la compilación añadiendo las rutas de las librerías de MPI.

Por ejemplo:

```
mpicc -o main main.c
```

2.2. Ejecución

Para la ejecución de un programa MPI es necesario utilizar un programa especial llamado `mpirun`, que permite configurar el número de procesos a ejecutar y se encarga generar tales procesos y comenzar la ejecución del programa en cada uno de ellos. Por ejemplo, para ejecutar un programa con 4 procesos usaríamos:

```
mpirun -np 4 ./main
```

Es importante tener en cuenta que el número de procesos es independiente del número de nodos físicos en que se ejecuta el programa. Por ejemplo, es posible utilizar MPI en una máquina multi-core, así como en un cluster. En una máquina multi-core `mpirun` intentará asignar procesos a hilos del procesador para maximizar el uso de recursos.

2.3. Ejemplo inicial

El siguiente programa muestra el esqueleto básico de un programa en MPI. La función `MPI_Init` debe invocarse al principio y la función `MPI_Finalize` debe invocarse al final. Es muy importante que todos los procesos ejecuten estas dos funciones, si no el programa podría fallar de manera inesperada.

mpi/hello_world.c

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

```

```c
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    // Obtener el numero de procesos
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Obtener el identificador del proceso actual
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    printf("Hola! Procesador %s, rango %d de %d procs.\n",
           processor_name, world_rank, world_size);

    MPI_Finalize();
}
```

La librería de MPI para C sigue la convención de que las funciones comienzan con el prefijo `MPI_`. Por ejemplo, `MPI_Comm_rank`. Las constantes siguen la misma convención pero en mayúsculas. Por ejemplo, `MPI_COMM_WORLD`. Por último, los identificadores de procesos van de *0* a *N - 1* y se considera que el proceso maestro tiene identificador *0*.

Al ejecutar este programa en 4 nodos de cómputo obtendríamos la siguiente salida.

```
$ mpirun -np 4 ./ejemplo
Hola! Procesador jesus-portatil, rango 1 de 4 procs.
Hola! Procesador jesus-portatil, rango 3 de 4 procs.
Hola! Procesador jesus-portatil, rango 2 de 4 procs.
Hola! Procesador jesus-portatil, rango 0 de 4 procs.
```

3. Conceptos básicos

3.1. Comunicadores

En MPI un **comunicador** es una colección de procesos que pueden enviarse mensajes entre sí. Habitualmente se utiliza `MPI_COMM_WORLD` para referirse al comunicador principal, que incluye todos los procesos creados al inicio de la ejecución. Todas la rutinas para enviar o recibir mensajes en MPI requieren que se pase el comunicador al que hace referencia la comunicación. Aunque es posible crear comunicadores propios, en nuestro caso no será será necesario. En cambio, sí que es necesario cuando se construyen librerías programadas con MPI, para evitar conflictos con el código cliente.

Hay dos funciones básicas para consultar el estado de un comunicador: `MPI_Comm_size` y `MPI_Comm_rank`.

`int MPI_Comm_size(MPI_Comm comm, int *size)`

- Devuelve el número de procesos en el comunicador
- El resultado en el parámetro size
- Es fijo durante toda la ejecución
- Se fija al inicio de la ejecución

`int MPI_Comm_rank(MPI_Comm comm, int *rank)`

- Identificador del proceso en el comunicador
- El resultado en el parámetro rank
- El valor de rank será entre 0..procesos -1

3.2. Otras funciones útiles

Para medir el tiempo de ejecución se utiliza la función `MPI_Wtime()`.

3.3. Envio y recepción de mensajes

Cuando un proceso `A` quiere enviar un mensaje a un proceso `B`, MPI realiza las siguientes acciones. El proceso A almacena los datos en un buffer para construir un mensaje. MPI se encarga de enrutar el mensaje hasta el otro proceso. Los procesos se identifican por su rango.

Existen diversas maneras de enviar y recibir mensajes en MPI. Dependiendo de la cantidad de procesos que reciben el mensaje:

- Punto a punto. Solo hay dos procesos involucrados en la comunicación, el emisor y el receptor.
- Colectivas. Hay más de dos procesos en la comunicación, normalmente un emisor y varios receptors (o varios emisores y un receptor).

Según si se espera que las operaciones sean bloqueantes o no:

- Síncrono. La operación de envío se bloquea hasta que el mensaje haya sido enviado. La operación de recepción se bloquea hasta recibir el mensaje esperado.
- Asíncrono. La operación de envío termina inmediamente, sin esperar a que el receptor haya recibido el mensaje o ni siquiera que el mensaje esté realmente enviándose. La operación de recepción termina inmediamente incluso si todavía no ha recibido el mensaje.

3.4. Comunicación punto a punto

Hay dos funciones básicas de comunicación en MPI, para enviar y recibir datos de manera síncrona: `MPI_Send` y `MPI_Recv`.

El siguiente programa muestra un ejemplo en el que proceso maestro recibe cadenas con un saludo desde el resto de proceso. El proceso maestro se identifica con `my_rank = 0` y el el resto son procesos esclavos. Los procesos esclavos construyen un mensaje que almacenan en `greetings`, y utilizan la función `MPI_Send` para enviarlo, indicando como destinatario el proceso 0, es decir, el proceso maestro. El proceso maestro por su parte recibe los mensajes en orden desde el proceso 1 hasta `comm_sz`.

mpi/send-rec-1.c

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

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;
   int        my_rank;

```

```
12    MPI_Init(NULL, NULL);
13
14    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
15
16    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
17
18    if (my_rank != 0) {
19       sprintf(greeting, "Greetings from process %d of %d!",
20             my_rank, comm_sz);
21       MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
22             MPI_COMM_WORLD);
23    } else {
24       printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
25       for (int q = 1; q < comm_sz; q++) {
26          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
27             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
28          printf(" %s\n", greeting);
29       }
30    }
31
32    MPI_Finalize();
33
34    return 0;
35 }
```

3.4.1. Funciones

A continuación se describen los parámetros de las funciones `MPI_Send` y `MPI_Recv`

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

- Los parámetros `buffer`, `tam` y `tipo` determinan el contenido del mensaje.
- `buffer` es un puntero a inicio de los datos que se quieren enviar.
- `tam` indica el número de elementos que tiene el buffer. Es importante observar que este valor no es el número de bytes. El tamaño en bytes lo calculara MPI según el tipo de datos que se envía.
- `tipo` permite indicar el tipo de datos destino indica el proceso al que se envía. Para ello se utilizará alguno de los valores del enumerado `MPI_Datype`.
- `destino` indica el proceso al que se le envía el mensaje.
- `tag` permite marcar el mensaje especialmente para poder discernir entre ellos.

Parámetro tamaño

Un aspecto importante es que el tamaño del buffer no se indica como el tamaño total en bytes, sino como el número total de elementos del buffer. El tamaño real lo calcula MPI internamente utilizando el tipo de datos como referencia.

MPI DataType	Tipo en C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	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	-

Figura 2: Tipos de datos en MPI

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

- `buffer` es un puntero al inicio del buffer donde se recibirán los datos.
- `tam` es la capacidad máxima del buffer.
- `tipo` el tipo de datos esperado.
- `origen` indica el proceso desde el que se desea recibir. Se puede usar `MPI_ANY_SOURCE` Para indicar que no importa quién lo envie tag permite distinguir mensajes status proporciona información sobre el mensaje (ej., ¿quién lo envía?)
- `tag` igual que en `MPI_Send`. Se puede utilizar la constante `MPI_ANY_TAG` para indicar que es irrelevante el tag que lleve el mensaje que se reciba.
- `estado` es un parámetro de salida donde se almacena información sobre quién envía el mensaje. Para ignorar esta información se puede pasar `MPI_STATUS_IGNORE` como argumento.

3.4.2. Condiciones de recepción de mensajes

En un programa MPI válido todo envio que se realize con MPI_Send debe tener emparejado un MPI_Recv que procese ese envío. Las condiciones de emparejamiento de los mensajes son las siguientes:

- Número de proceso del MPI_Send debe corresponder con el número de proceso MPI_Recv o que se use MPI_ANY_SOURCE.
- Tag debe ser equivalente o MPI_ANY_TAG.
- Sólo se reciben los mensajes los procesos del mismo comunicador.
- Tipo de datos origen debe ser igual al tipo de datos destino.
- Tamaño del buffer destino debe ser mayor o igual que el buffer enviado.

Cuando la función `MPI_Send` va a enviar un mensaje, el sistema puede decidir si almacenar el buffer en una zona de almacenamiento interno o utilizar directamente el buffer para el envío. En el primer caso, la función puede terminar inmediatamente, mientras que en el segundo se bloquea hasta que ha realizado el envío. En la práctica, lo que esto significa es que cuando la función `MPI_Send` termina la única garantía que se tiene es que el buffer que se ha utilizado puede ser reutilizado (o bien se ha copiado o bien se ha enviado), pero no significa que el mensaje haya llegado ya a su destino.

En cambio, `MPI_Recv` siempre se bloquea hasta que recibe un mensaje que puede emparejar de acuerdo con las reglas indicadas anteriormente. Por tanto, el programa se quedará "colgado" si todas las operaciónes de recepción de mensajes no tienen su envío correspondiente.

Se garantiza que si un proceso p envía dos mensajes seguidos, m_1 y m_2, otro proceso q recibirá m_1 y m_2 en orden. Sin embargo, si los mensajes proceden de procesos diferentes, aunque m_1 haya sido enviado en un instante anterior a m_2, no hay garantía del orden en que llegarán.

3.4.3. Utilización de MPI_ANY_SOURCE

Supongamos que un proceso (ej., el maestro) utiliza un bucle para recibir un mensaje desde cada proceso esclavo, por ejemplo como el siguiente:

```
for (int i = 1; i < size; i++) {
  MPI_Recv(buffer, MAX_SIZE, MPI_CHAR, i,
        0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  ...
}
```

El problema que tiene este código es que obliga a recibir los mensajes en el orden del 1 a $size$, pero si el proceso 1 tarda en generar su resultado (para poder enviarlo) más tiempo que que el proceso 2, entonces el proceso maestro estará ocioso sin necesidad.

La solución es poder utilizar un "comodín" para indicar que se desea recibir un mensaje de cualquier proceso. Ese es el papel que juega la constante `MPI_ANY_SOURCE`.

Recibir cualquier mensaje.

```
for (int i = 1; i < size; i++) {
  MPI_Recv(buffer, MAX_SIZE, MPI_CHAR, MPI_ANY_SOURCE,
        0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  ...
}
```

3.4.4. Utilización de MPI_STATUS

Cuando se recibe un mensaje no se sabe la cantidad de datos el mensaje (el parámetro `tam` indica el tamaño máximo del buffer, pero no la cantidad de datos reales que se han recibido). Si se usa `MPI_ANY_SOURCE` tampoco se sabe quién es el emisor o cuál es el tag si se usa `MPI_ANY_TAG`.

El parámetro de salida `estado` en MPI_Recv tiene tipo `MPI_Status` y recoge estos datos. Por ejemplo:

```
MPI_Status status;
MPI_Recv(buffer, MAX_SIZE, MPI_CHAR, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, &status);

status.MPI_SOURCE;
status.MPI_TAG;
int count;
MPI_Get_count(&status, MPI_CHAR, &count);
```

Se utiliza esta función para obtener el número de elementos recib

3.5. Otros tipos de comunicaciones

MPI_Send y MPI_Recv son operaciones bloqueantes, esto es, su ejecución no termina hasta que la operación ha sido completada. Sin embargo, es importante entender qué significa esto. MPI_Send usa el llamado *modo de comunicación estandar* que garantiza que tras la ejecución de la función el buffer usado puede ser utilizado. Esto no significa que cuando MPI_Send termina el mensaje haya sido ya entregado en su destino. Es posible que la implementación de MPI haya copiado el buffer a su almacenamiento interno, o puede que haya realizado el envío directamente. En el caso de MPI_Recv, no devolverá el control hasta que haya recibido y copiado en el buffer el mensaje requerido.

Hay tres modos adicionales de comunicación [1, p.252]:

- Buffered. Similar al modo estándar (localmente bloqueante) pero el buffer donde se copiarán los datos lo proporciona el usuario.

Buffers y punteros

Un buffer hace referencia a una bloque contiguo de memoria que contiene los datos que se quieren enviar (o donde se quieren recibir datos). En C usaremos un puntero al inicio de esa zona de memoria para representar al buffer y poder usarlo en MPI. A continuación se muestran algunos tipos de buffers.

```
char *msj = "hola";
char c = 'a';
int array[3] = { 1, 2, 3 };

MPI_Send(msj, strlen(msj + 1), ...);
MPI_Send(&c, 1, ...);
MPI_Send(array, 3);

char *recibir =
 malloc(sizeof(char) * 10);
MPI_Recv(recibir, 10, ...);
```

```
int MPI_Bsend(void *bufffer, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

Esta función garantiza que los datos que se van a enviar van a ser copiados al buffer proporcionado, mientras que con `MPI_Send` puede ser que la implementación no utilice un buffer interno y decida bloquear hasta haber realizado el envio.
Para preparar el buffer hay que usar la funciones `MPI_Buffer_attach` para indicar qué zona de memoria hará de buffer, y `MPI_Buffer_deatach` para liberarla.

- Síncrono. La operación de envío termina sólo cuando el el proceso receptor ha comenzado la recepción del mensaje.

```
int MPI_Ssend (void *bufffer, int count, MPI_Datatype
datatype , int dest, int tag, MPI_Comm comm)
```

Al utilizar esta operación MPI no necesita utilizar ningún buffer interno porque no terminará la ejecución hasta que el mensaje haya llegado a su destino.

- Ready. La operación de envío solo tiene éxito si había una operación de recepción ya esperando. Si no, la operación devuelve un código de error. Su utilidad es reducir el coste del protocolo de comunicación.

```
int MPI_Rsend (void *bufffer, int count, MPI_Datatype
datatype , int dest, int tag, MPI_Comm comm)
```

3.6. Comunicaciones no bloqueantes

TODO: Explicar las funciones no bloqueantes como MPI_Isend y MPI_Irecv.

4. Comunicaciones colectivas

Las comunicaciones colectivas son operaciones en las que la comunicación se produce entre más de dos nodos. La idea es conseguir aprovechar mejor los recursos cuando hay que enviar o recibir de datos de más de un nodo.

Por ejemplo, un patrón muy común es que el proceso maestro disponga de un cierto dato y deba hacerlo disponible al resto de nodos del programa. Utilizando comunicaciones punto a punto el maestro podría leer el dato de la entrada estándar y enviarlo de la siguiente manera:

```
1  MPI_Comm_rank (MPI_COMM_WORLD , &rank);
2  MPI_Comm_size (MPI_COMM_WORLD , &size);
3
4  int param;
5  if (rank == 0) {
6   scanf("%d", param);
7   for(int i = 1; i < size; i ++) {
8     MPI_Send(&param , 1 , MPI_INT, i, 1, MPI_COMM_WORLD);
9   }
10 } else {
11   MPI_Recv(&param, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
         ;
12 }
```

El coste de comunicación de este trozo de código es proporcional al número de procesos, puesto que hay que realizar $O(size)$ llamadas a MPI_Send, una tras otra.

Para abordar este problema MPI ofrece las llamadas funciones de comunicación colectivas.

4.1. Broadcast

Se debe utilizar cuando un proceso dispone de datos que quiere distribuir al resto de procesos. El ejemplo anterior se podría implementar de la siguiente manera. El proceso maestro lee el dato, y todos los procesos ejecutan la función MPI_Bcast. En el caso del proceso maestro, MPI_Bcast funciona como emisor y para el resto de procesos como receptor.

```
MPI_Comm_rank (MPI_COMM_WORLD , &rank) ;

int param;
if (rank == 0) {
  scanf("%d", param);
}

MPI_Bcast (&param, 1, MPI_INT, 0, MPI_COMM_WORLD) ;
```

```
int MPI_Bcast( void * datos /* in/out */, int count /* in */,
MPI_Datatype datatype /* in */, int origen /* in */, MPI_Comm
comm /* in */)
```

- **datos** es el puntero al buffer de datos, que será de entrada para el proceso emisor y de salida para los procesos receptores.
- **count** número de elementos en el buffer
- **datatype** el tipo de datos
- **origen** es el identificador del proceso que envía el dato al resto

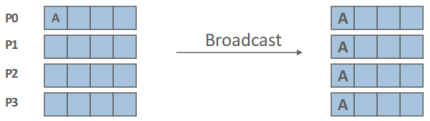

Figura 3: Funcionamiento de MPI_Broadcast

4.2. Scattering

Existen problemas para los que debemos repartir los datos entre los diferentes procesos. Para ellos es común que el buffer con los datos deba dividirse en trozos, que se envían a los procesos para que pueden realizar esa parte del trabajo. La función `MPI_Scatter` divide el buffer en trozos y lo envía en partes iguales a todos los procesos.

Por ejemplo, el siguiente código distribuye un buffer de enteros entre todos los procesos.

```
double *a;
int local_n = n / num_procesos;
double local_a = malloc(local_n*sizeof(double));
if (my_rank == 0) {
    a = malloc(n*sizeof(double));
    ... Llenar el vector ...
    MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, comm);
    free(a);
 } else {
    MPI_Scatter(NULL, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0,
     comm);
 }
... Usar local_a ...
free(local_a)
```

mpi/vector_add.c

Uso de MPI_Scatter

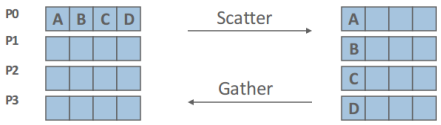

Figura 4: Funcionamiento de MPI_Scatter y MPI_Gather

```
MPI_Scatter( void* send_data, /*in*/ int send_count, /*in*/
MPI_Datatype send_type, /*in*/ void* recv_data, /*out*/ int
recv_count, /*in*/ MPI_Datatype recv_datatype,/*in*/ int
src_proc, /*in*/ MPI_Comm communicator /*in*/)
```

Divide un array (buffer) en trozos y lo envía en partes iguales a todos los procesos. El array debe tener un tamaño múltiplo del número de procesos.

- `send_data`. El buffer que se quiere enviar.
- `send_count`. El número de elementos que recibirá cada proceso, es decir, tamaño buffer/núm. procesos.
- `send_type`. El tipo de los elementos del buffer.
- `recv_data`. El buffer donde se recibirán los datos. El proceso 0 recibirá en este buffer los primeros `send_cound / num_procesos` datos, el proceso 1 los siguientes, etc.
- `recv_count`. El número de elementos que recibe el proceso. Típicamente el mismo valor que `send_count`.
- `recv_datatype`. El tipo de datos que se recibe. Típicamente el mismo valor que `send_type`.
- `src_proc`. El proceso que actúa como maestro y está enviando los datos.

La principal de limitación de MPI_Scatter es que requiere que el tamaño del buffer sea divisible por el número de procesos. Hay varias soluciones para este problema:

- Dejar el resto al proceso maestro. Esto es, si el buffer tiene tamaño t y hay p procesos, a cada proceso se le asignarán t/p datos, excepto al proceso maestro que se le asignarán $t/p + t \bmod p$. En la práctica, una forma sencilla es asumir que el buffer comienza en $t \bmod p$ de manera que el maestro trabajará de manera natural desde 0 hasta $t/p + t \bmod p$.
- El problema del método anterior es que cuando hay muchos datos y muchos procesadores es posible que el resto también sea muy grande y por tanto haya un desbalanceo de la carga. Por ejemplo, supongamos que tenemos los siguientes datos:

$$\begin{aligned} &\text{datos} = 1000 \\ &\text{procesos} = 101 \\ &\text{tamaño del trozo} = 1000/101 = 9{,}9 \text{ (redondeo a 9)} \\ &\text{resto} = 91 \end{aligned} \tag{1}$$

  Todos los procesos se harían cargo de 9 datos, excepto el maestro que tendría que hacerse cargo de ¡$91+9 = 100$ datos! Una solución es repartir dinámicamente estos datos entre algunos de los procesos para reducir la carga del maestro.
- Hacer padding the buffer para ampliar su tamaño de manera artificial. Ver ejemplo en [1, p.270]. Este método es similar al primero en cuanto a que un proceso también realizará más trabajo.
- Usar la función `MPI_Scatterv`.

4.3. Gather

Es la función "inversa" a *scatter*. Su cometido es recolectar los resultados obtenidos por los nodos hijos y centralizarlos en el nodo padre.

```
MPI_Gather( void* send_data, /*in*/ int send_count, /*in*/
MPI_Datatype send_type, /*in*/ void* recv_data, /*out*/ int
recv_count, /*in*/ MPI_Datatype recv_datatype,/*in*/ int
dest_proc, /*in*/ MPI_Comm communicator /*in*/)
```

- `send_data`. El buffer que se quiere enviar. Normalmente cada proceso tendrá un buffer de tamaño tamaño total/núm. procesos.
- `send_count`. El número de elementos que se están enviando.
- `send_type`. El tipo de los elementos del buffer.
- `recv_data`. El buffer donde se recibirán los datos. El proceso receptor (normalmente el 0) tendrá que reservar memoria para el tamaño total.
- `recv_count`. El número de elementos que recibe de cada proceso (¡no el total!). Típicamente el mismo valor que `send_count`. Solo el proceso raíz tendrá este parámetro en cuenta.
- `recv_datatype`. El tipo de datos que se recibe. Típicamente el mismo valor que `send_type`.
- `dest_proc`. El proceso que actúa como maestro y y que va a recibir los datos

El siguiente programa utiliza las operaciones *scatter* y *gather* para repartir los datos, realizar una operación, y luego devolverlos al proceso maestro que se encarga de presentar el resultado.

vector_add_colectiva.c

Uso de MPI_Scatter y MPI_Gather para paralelizar una operación de suma de dos vectores.

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

double *generar(int n) {
  double *numeros = (double *) malloc(n*sizeof(double));
  for(int i = 0; i < n; i++) {
    numeros[i] = rand() * 1000;
    // Para probarlo, no usar numeros aleatorios
    // numeros[i] = i;
  }
  return numeros;
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int n = 1024;
    // Unos pocos numeros para probar
    // int n = 16;
    int trozos = n / world_size;

    if (n % world_size != 0) {
      printf("Numero de procesos invalido.\n");
```

```
31      exit(-1);
32    }
33
34    // Operandos
35    double *local_a = malloc(trozos * sizeof(double));
36    double *local_b = malloc(trozos * sizeof(double));
37    // Resultado
38    double *local_c = malloc(trozos * sizeof(double));
39
40    // Repartir los datos
41    if (rank == 0) {
42      double *a = generar(n);
43      double *b = generar(n);
44
45      MPI_Scatter(a, trozos, MPI_DOUBLE, local_a, trozos, MPI_DOUBLE, 0,
         MPI_COMM_WORLD);
46      MPI_Scatter(b, trozos, MPI_DOUBLE, local_b, trozos, MPI_DOUBLE, 0,
         MPI_COMM_WORLD);
47
48      free(a);
49      free(b);
50    } else {
51      MPI_Scatter(NULL, trozos, MPI_DOUBLE, local_a, trozos, MPI_DOUBLE, 0,
         MPI_COMM_WORLD);
52      MPI_Scatter(NULL, trozos, MPI_DOUBLE, local_b, trozos, MPI_DOUBLE, 0,
         MPI_COMM_WORLD);
53    }
54
55    // Todos suman
56    for(int i = 0; i < trozos; i++) {
57      local_c[i] = local_a[i] + local_b[i];
58    }
59
60    // Devolver los datos al maestro
61    if (rank == 0) {
62      double *r = (double *) malloc(n * sizeof(double));
63
64      MPI_Gather(local_c, trozos, MPI_DOUBLE, r, trozos, MPI_DOUBLE, 0,
         MPI_COMM_WORLD);
65
66      for(int i = 0; i < n; i++) {
67  printf("%2.f\n", r[i]);
68      }
69
70      free(r);
71    } else {
72      MPI_Gather(local_c, trozos, MPI_DOUBLE, NULL, trozos, MPI_DOUBLE, 0,
         MPI_COMM_WORLD);
73    }
74
75    free(local_a);
76    free(local_b);
77    free(local_c);
78
79    MPI_Finalize();
80  }
```

4.4. Reduction

TODO: Explicar las funciones no bloqueantes como MPI_Isend y MPI_Irecv.

4.5. All-to-all

En las funciones colectivas que se han presentado siempre hay un proceso que actúa como maestro, bien para enviar o para recibir los datos. Sin embargo, hay ocasiones en que se desea que todos los procesos reciban los mismos datos. Por

ejemplo, tras aplicar una operación reduction se desea que todos los procesos obtengan el resultado final para así poder continuar haciéndo cálculos.

4.6. Barreras

Una barrera establece un punto de sincronización en el cual los procesos deben esperar hasta que todos los procesos alcancen ese punto.

En MPI, una barrera se especifica con `MPI_Barrier`, que bloquea al proceso hasta que todos los procesos pertenecientes al comunicador especificado lo ejecuten.

5. Empaquetamiento y tipos derivados

Hasta el momento hemos visto cómo utilizar las primitivas de comunicación como `MPI_Send` y `MPI_Recv` para enviar y recibir datos de tipo primitivo (un entero, un *double*, etc.) o bien un array de valores primitivos (utilizando el parámetro `element_count` para indicar cuántos elementos hay).

5.1. Empaquetamiento

En ocasiones es necesario tener que enviar varios datos diferentes a otro proceso. Por ejemplo, supongamos que queremos enviar dos datos: un número real (ej., 10.0) y una cadena de texto como "hola". Para ello podemos realizar varias llamadas a `MPI_Send` y los correspondientes `MPI_Recv`. Sin embargo, en un sistema de memoria distribuida el coste de enviar dos mensajes con un dato cada uno será típicamente mayor que el coste enviar un mensaje con los dos datos juntos. La alternativa es *empaquetar* los datos en un buffer contiguo antes se desempaquetan al recibirlos. Para ello utilizaremos las funciones `MPI_Pack` y `MPI_Unpack` y el tipo asociado a este tipo de buffers de empaquetado es `MPI_Packed`.

El siguiente listado muestra un ejemplo. Para poder utilizar `MPI_Pack` debemos disponer de un buffer auxiliar donde ir agregando los datos que se quieren empaquetar juntos. En este ejemplo, deseamos empaquetar un dato de tamaño fijo (un número de tipo double) y un dato de tamaño variable (una cadena de texto[1]). La estrategia que se sigue se muestra en la Figura 5.

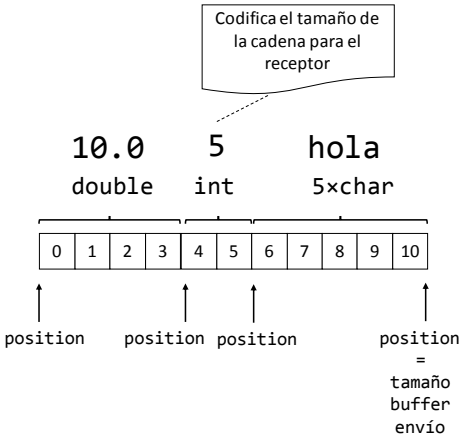

Figura 5: Organización del buffer para MPI_Pack.

```
#define MAX_TEXT 64

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int buff_size = sizeof(double) + sizeof(int) + sizeof(char) * MAX_TEXT;
    char * buffer = (char *) malloc(buff_size);

    if (rank == 0) {
      double value = 10.0;
      char *text = "hola";
      int text_len = strlen(text) + 1;
      int position = 0;

      MPI_Pack(&value, 1, MPI_DOUBLE, buffer, buff_size, &position,
       MPI_COMM_WORLD);
      MPI_Pack(&text_len, 1, MPI_INT, buffer, buff_size, &position,
       MPI_COMM_WORLD);
```

[1]En el ejemplo tiene un tamaño fijo pero en una aplicación real podría ser variable.

```
23      MPI_Pack(text, text_len, MPI_CHAR, buffer, buff_size, &position,
         MPI_COMM_WORLD);
24
25      // position contiene ahora el tam. real de los datos
26      MPI_Bcast(buffer, position, MPI_PACKED, 0, MPI_COMM_WORLD);
27    } else {
28      double value;
29      char text[MAX_TEXT];
30      int text_len;
31      int position = 0;
32
33      MPI_Bcast(buffer, buff_size, MPI_PACKED, 0, MPI_COMM_WORLD);
34
35      MPI_Unpack(buffer, buff_size, &position, &value, 1, MPI_DOUBLE,
         MPI_COMM_WORLD);
36      MPI_Unpack(buffer, buff_size, &position, &text_len, 1, MPI_INT,
         MPI_COMM_WORLD);
37      MPI_Unpack(buffer, buff_size, &position, text, text_len, MPI_CHAR,
         MPI_COMM_WORLD);
38
39      printf("Recibido: %f y %s\n", value, text);
40    }
41
42    free(buffer);
43
44    MPI_Finalize();
45  }
```

A la hora de enviar datos empaquetados es importante darse cuenta de que el tipo es `MPI_PACKED` y por tanto el número de elementos del buffer es el tamaño en bytes. El parámetro `position` nos ayudará a llevar el conteo del número de bytes correctamente.

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

- Empaqueta incount datos de tipo datatype
- `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.

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

- Desempaqueta count elementos de tipo datatype en unpack data, tomándolos de la posición `buffer+*position_ptr` del buffer.
- Hay que especificar el tamaño del buffer (`size`) en bytes, y `position_ptr` es manejado por MPI.

El empaquetamiento resulta útil cuando el mensaje que se quiere enviar tiene alguna de estas características:

- Datos no contiguos de un único tipo (ej., un sub-bloque de una matriz)
- Datos contiguos de tipos diferentes (ej., un entero seguido de una secuencia de números reales)

- Datos no contiguos de diferentes tipos

5.2. Tipos derivados

TODO: Explicar los tipos derivados.

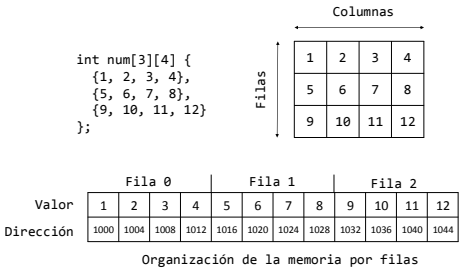

Figura 6: Organización de la memoria en C. Los arrays bidimensionales se organizan por filas.

6. Ejercicios

- Construye un sistema de memoria compartida sobre MPI. El sistema debería gestionar un espacio de direcciones virtual, por ejemplo como un mapa clave-valor distribuido.

7. Bibliografía comentada

- Tutorial de MPI: `https://mpitutorial.com/tutorials/mpi-introduction/`
- *An introduction to parallel programming* de Peter Pacheco [2]. El capítulo 3 presenta MPI e incluye varios ejemplos.

Referencias

[1] G. Barlas. *Multicore and GPU Programming: An integrated approach*. Elsevier, 2014.

[2] P. Pacheco. *An introduction to parallel programming*. Elsevier, 2011.