Programación de aplicaciones paralelas con MPI (*Message Passing Interface*)

José Miguel Alonso
Facultad de Informática UPV/EHU
miguel@si.ehu.es
13/1/97

1. Introducción a MPI

MPI (*Message Passing Interface*) es un Interfaz estandarizado para la realización de aplicaciones paralelas basadas en paso de mensajes. El modelo de programación que subyace tras MPI es MIMD (*Multiple Instruction streams, Multiple Data streams*) aunque se dan especiales facilidades para la utilización del modelo SPMD (*Single Program Multiple Data*), un caso particular de MIMD en el que todos los procesos ejecutan el mismo programa, aunque no necesariamente la misma instrucción al mismo tiempo.

MPI es, como su nombre indica, un interfaz, lo que quiere decir que el estándar no exige una determinada implementación del mismo. Lo importante es dar al programador una colección de funciones para que éste diseñe su aplicación, sin que tenga necesariamente que conocer el hardware concreto sobre el que se va a ejecutar, ni la forma en la que se han implementado las funciones que emplea.

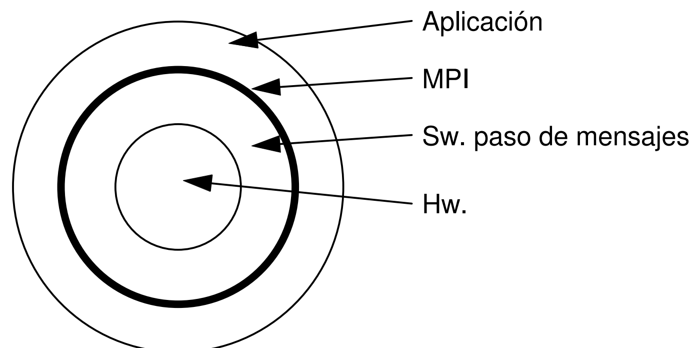

Figura 1. Ubicación de MPI en el proceso de programación de aplicaciones paralelas.

MPI ha sido desarrollado por el MPI Forum, un grupo formado por investigadores de universidades, laboratorios y empresas involucrados en la computación de altas prestaciones. Los objetivos fundamentales del MPI Forum son los siguientes:

1. Definir un entorno de programación único que garantice la portabilidad de las aplicaciones paralelas.
2. Definir totalmente el interfaz de programación, sin especificar cómo debe ser la implementación del mismo

3. Ofrecer implementaciones de calidad, de dominio público, para favorecer la extensión del estándar.
4. Convencer a los fabricantes de computadores paralelos para que ofrezcan versiones de MPI optimizadas para sus máquinas (lo que ya han hecho fabricantes como IBM y Silicon Graphics).

Los elementos básicos de MPI son una definición de un interfaz de programación independiente de lenguajes, más una colección de *bindings* o concreciones de ese interfaz para los lenguajes de programación más extendidos en la comunidad usuaria de computadores paralelos: C y FORTRAN.

Un programador que quiera emplear MPI para sus proyectos trabajará con una implementación concreta de MPI, que constará de, al menos, estos elementos:

- Una biblioteca de funciones para C, más el fichero de cabecera **mpi.h** con las definiciones de esas funciones y de una colección de constantes y macros.
- Una biblioteca de funciones para FORTRAN + **mpif.h**.
- Comandos para compilación, típicamente **mpicc**, **mpif77**, que son versiones de los comandos de compilación habituales (cc, f77) que incorporan automáticamente las bibliotecas MPI.
- Comandos para la ejecución de aplicaciones paralelas, típicamente **mpirun**.
- Herramientas para monitorización y depuración.

MPI no es, evidentemente, el único entorno disponible para la elaboración de aplicaciones paralelas. Existen muchas alternativas, entre las que destacamos las siguientes:

- Utilizar las bibliotecas de programación propias del computador paralelo disponible: NX en el Intel Paragon, MPL en el IBM SP2, etc.
- PVM (*Parallel Virtual Machine*): de características similares a MPI, se desarrolló con la idea de hacer que una red de estaciones de trabajo funcionase como un multicomputador. Funciona también en multicomputadores, normalmente como una capa de software encima del mecanismo de comunicaciones nativo.
- Usar, si es posible, lenguajes de programación paralelos (FORTRAN 90) o secuenciales (C, FORTRAN 77) con directivas de paralelismo.
- Usar lenguajes secuenciales junto con compiladores que paralelicen automáticamente.

MPI está aún en sus comienzos, y aunque se está haciendo un hueco creciente en la comunidad de programadores de aplicaciones científicas paralelas, no es probable que desplace a corto plazo a los entornos de programación ya existentes (como los anteriormente citados) o impida la aparición de otros nuevos. El MPI Forum es consciente de que MPI todavía adolece de algunas limitaciones, e incluso ha identificado bastantes de ellas:

- Entrada/salida: no se establece un mecanismo estandarizado de E/S paralela.
- Creación dinámica de procesos. MPI asume un número de procesos constante, establecido al arrancar la aplicación.
- Variables compartidas. El modelo de comunicación estandarizado por MPI sólo tiene en cuente el paso de mensajes.
- *Bindings* para otros lenguajes, además de C y FORTRAN. Se piensa, en concreto, en C++ y Ada.
- Soporte para aplicaciones de tiempo real. MPI no recoge en ningún punto restricciones de tiempo real.
- Interfaces gráficos. No se define ningún aspecto relacionado con la interacción mediante GUIs con una aplicación paralela.
- Etc.

Como ya hemos comentado, MPI está especialmente diseñado para desarrollar aplicaciones SPMD. Al arrancar una aplicación se lanzan en paralelo N copias del mismo programa[*] (procesos). Estos procesos no avanzan sincronizados instrucción a instrucción sino que la sincronización, cuando sea necesaria, tiene que ser explícita. Los procesos tienen un espacio de memoria completamente separado. El intercambio de información, así como la sincronización, se hacen mediante paso de mensajes.

Se dispone de funciones de comunicación punto a punto (que involucran sólo a dos procesos), y de funciones u operaciones colectivas (que involucran a mútiples procesos). Los procesos pueden agruparse y formar *comunicadores*, lo que permite una definición del ámbito de las operaciones colectivas, así como un diseño modular.

La estructura típica de un programa MPI, usando el *binding* para C, es la siguiente:

```
# include "mpi.h"
main (int argc, char **argv) {
     int nproc; /* Número de procesos */
     int yo; /* Mi dirección: 0<=yo<=(nproc-1) */

     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &nproc);
     MPI_Comm_rank(MPI_COMM_WORLD, &yo);

     /* CUERPO DEL PROGRAMA */

     MPI_Finalize();
}
```

Este segmento de código ya nos presenta cuatro de las funciones más

[*] Las implementaciones de MPI también suelen permitir lanzar aplicaciones en las que no todos los procesos ejecutan el mismo programa.

utilizadas de MPI: MPI_Init() para iniciar la aplicación paralela, MPI_Comm_size() para averiguar el número de procesos que participan en la aplicación, MPI_Comm_rank(), para que cada proceso averigue su dirección (identificador) dentro de la colección de procesos que componen la aplicación, y MPI_Finalize() para dar por finalizada la aplicación.

```
int MPI_Init(int *argc, char ***argv);
int MPI_Comm_size (MPI_Comm comm, int *size);
int MPI_Comm_rank (MPI_Comm comm, int *rank);
int MPI_Finalize(void);
```

El ejemplo nos sirve también para que prestemos atención a algunas convenciones de MPI. Los nombres de todas las funciones empiezan con "MPI_", la primera letra que sigue siempre es mayúscula, y el resto son minúsculas.

La mayor parte de las funciones MPI devuelven un entero, que es un diagnóstico. Si el valor devuelto es MPI_SUCCESS, la función se ha realizado con éxito. No se han estandarizado otros posibles valores.

La palabra clave MPI_COMM_WORLD hace referencia al comunicador universal, un comunicador predefinido por MPI que incluye a todos los procesos de la aplicación. Más adelante veremos cómo definir otros comunicadores. Todas las funciones de comunicación de MPI necesitan como argumento un comunicador.

En el resto de este tutorial vamos a ir presentando las diferentes funciones que MPI ofrece para la comunicación y sincronización entre procesos. En la sección 2 presentamos los mecanismos para la comunicación entre pares de procesos (Comunicación punto a punto). En la sección 3 presentamos las funciones para comunicación entre grupos de procesos (Operaciones colectivas). La sección 4 discute aspectos de MPI relacionados con la Modularidad. En la sección 5 se describen las funciones disponibles para definir Tipos de datos derivados. Por último, la sección 6 (Bibliografía) aporta referencias para la localización de información adicional sobre MPI.

2. Comunicación punto a punto

Un buen número de funciones de MPI están dedicadas a la comunicación entre pares de procesos. Existen múltiples formas distintas de intercambiar un mensaje entre dos procesos, en función del *modelo* y el *modo* de comunicación elegido.

2.1 Modelos y modos de comunicación

MPI define dos modelos de comunicación: bloqueante (blocking) y no bloqueante (nonblocking). El modelo de comunicación tiene que ver con el tiempo que un proceso pasa bloqueado tras llamar a una función de comunicación, sea ésta de envío o de recepción. Una función bloqueante mantiene a un proceso bloqueado hasta que la operación solicitada finalice. Una no bloqueante supone simplemente "encargar" al sistema la realización de una operación, recuperando el control inmediatamente. El proceso tiene

que preocuparse, más adelante, de averiguar si la operación ha finalizado o no.

Queda una duda, sin embargo: ¿cuándo damos una operación por finalizada? En el caso de la recepción está claro: cuando tengamos un mensaje nuevo, completo, en el buffer asignado al efecto. En el caso de la emisión la cosa es más compleja: se puede entender que la emisión ha terminado cuando el mensaje ha sido recibido en destino, o se puede ser menos restrictivo y dar por terminada la operación en cuanto se ha hecho una copia del mensaje en un buffer del sistema en el lado emisor. MPI define un envío como finalizado cuando el emisor puede reutilizar, sin problemas de causar interferencias, el buffer de emisión que tenía el mensaje. Dicho esto, podemos entender que tras hacer un send (envío) bloqueante podemos reutilizar el buffer asociado sin problemas, pero tras hacer un send no bloqueante tenemos que ser muy cuidadosos con las manipulaciones que se realizan sobre el buffer, bajo el riesgo de alterar inadvertidamente la información que se está enviando.

Al margen de si la función invocada es bloqueante o no, el programador puede tener un cierto control sobre la forma en la que se realiza y completa un envío. MPI define, en relación a este aspecto, 4 modos de envío: básico (basic), con buffer (buffered), síncrono (synchronous) y listo (ready).

Cuando se hace un envío **con buffer** se guarda inmediatamente, en un buffer al efecto en el emisor, una copia del mensaje. La operación se da por completa en cuanto se ha efectuado esta copia. Si no hay espacio en el buffer, el envío fracasa.

Si se hace un envío **síncrono**, la operación se da por terminada sólo cuando el mensaje ha sido recibido en destino. Este es el modo de comunicación habitual en los sistemas basados en Transputers. En función de la implementación elegida, puede exigir menos copias de la información conforme ésta circula del buffer del emisor al buffer del receptor.

El modo de envío **básico** no especifica la forma en la que se completa la operación: es algo dependiente de la implementación. Normalmente equivale a un envío con buffer para mensajes cortos y a un envío síncrono para mensajes largos. Se intenta así agilizar el envío de mensajes cortos a la vez que se procura no perder demasiado tiempo realizando copias de la información.

En cuanto al envío en modo **listo**, sólo se puede hacer si antes el otro extremo está preparado para una recepción inmediata. No hay copias adicionales del mensaje (como en el caso del modo con buffer), y tampoco podemos confiar en bloquearnos hasta que el receptor esté preparado.

2.2 Comunicación básica

El resultado de la combinación de dos modelos y cuatro modos de comunicación nos da 8 diferentes funciones de envío. Funciones de recepción sólo hay dos, una por modelo. Presentamos a continuación los

prototipos de las funciones más habituales. Empezamos con MPI_Send() y MPI_Recv que son, respectivamente, las funciones de envío y recepción básicas bloqueantes.

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

El significado de los parámetros es como sigue. **Buf**, **count** y **datatype** forman el mensaje a enviar o recibir: count copias de un dato del tipo datatype que se encuentran (o se van a dejar) en memoria a partir de la dirección indicada por buf. **Dest** es, en las funciones de envío, el identificador del proceso destinatario del mensaje. **Source** es, en las funciones de recepción, el identificador del emisor del cual esperamos un mensaje. Si no nos importa el origen del mensaje, podemos poner MPI_ANY_SOURCE. **Tag** es una etiqueta que se puede poner al mensaje. El significado de la etiqueta lo asigna el programador. Suele emplearse para distinguir entre diferentes clases de mensajes. El emisor pone siempre una etiqueta a los mensajes, y el receptor puede elegir entre recibir sólo los mensajes que tengan una etiqueta dada, o aceptar cualquier etiqueta, poniendo MPI_ANY_TAG como valor de este parámetro. **Comm** es un comunicador; en muchas ocasiones se emplea el comunicador universal MPI_COMM_WORLD. **Status** es un resultado que se obtiene cada vez que se completa una recepción, y nos informa de aspectos tales como el tamaño del mensaje recibido, la etiqueta del mensaje y el emisor del mismo. La definición de la estructura MPI_Status es la siguiente:

```
typedef struct {
    int MPI_SOURCE;
    int MPI_TAG;
    /* otros campos no accesibles */
} MPI_Status;
```

Podemos acceder directamente a los campos .MPI_SOURCE y .MPI_TAG, pero a ningún otro más. Si queremos saber el tamaño de un mensaje, lo haremos con la función MPI_Get_count():

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype
datatype, int *count);
```

MPI_Isend() y MPI_Irecv() son las funciones de emisión/recepción básicas no bloqueantes.

```
int MPI_Isend(void* buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm, MPI_Request
*request);
int MPI_Irecv(void* buf, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm, MPI_Request
*request);

int MPI_Wait(MPI_Request *request, MPI_Status *status);
int MPI_Test(MPI_Request *request, int *flag, MPI_Status
```

```
*status);
int MPI_Cancel(MPI_Request *request);
```

Las funciones no bloqueantes manejan un objeto **request** del tipo MPI_Request. Este objeto es una especie de “recibo” de la operación solicitada. Más adelante se podrá utilizar este recibo para saber si la operación ha terminado o no.

La función MPI_Wait() toma como entrada un recibo, y bloquea al proceso hasta que la operación correspondiente termina. Por lo tanto, hacer un MPI_Isend() seguido de un MPI_Wait() equivale a hacer un envío bloqueante. Sin embargo, entre la llamada a la función de envío y la llamada a la función de espera el proceso puede haber estado haciendo cosas útiles, es decir, consigue solapar parte de cálculo de la aplicación con la comunicación.

Cuando no interesa bloquearse, sino simplemente saber si la operación ha terminado o no, podemos usar MPI_Test(). Esta función actualiza un **flag** que se le pasa como segundo parámetro. Si la función ha terminado, este flag toma el valor 1, y si no ha terminado pasa a valer 0.

Por último, MPI_Cancel() nos sirve para cancelar una operación de comunicación pendiente, siempre que ésta aún no se haya completado.

2.3 Comunicación con buffer

Una de los problemas que tiene el envío básico es que el programador no tiene control sobre cuánto tiempo va a tardar en completarse la operación. Puede que apenas tarde, si es que el sistema se limita a hacer una copia del mensaje en un buffer, que saldrá más tarde hacia su destino; pero también puede que mantenga al proceso bloqueado un largo tiempo, esperando a que el receptor acepte el mensaje.

Para evitar el riesgo de un bloqueo no deseado se puede solicitar explícitamente que la comunicación se complete copiando el mensaje en un buffer, que tendrá que asignar al efecto el propio proceso. Así, se elimina el riesgo de bloqueo mientras se espera a que el interlocutor esté preparado. La función correspondiente es MPI_Bsend(), que tiene los mismos argumentos que MPI_Send().

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

Para que se pueda usar el envío con buffer es necesario que el programador asigne un buffer de salida. Para ello hay reservar previamente una zona de memoria (de forma estática o con malloc()) y luego indicarle al sistema que la emplee como buffer. Esta última operación la hace MPI_Buffer_attach(). Ese buffer se puede recuperar usando MPI_Buffer_detach().

```
int MPI_Buffer_attach(void* buffer, int size);
int MPI_Buffer_detach(void* buffer, int* size);
```

Recordemos que una peculiaridad de los envíos con buffer es que fracasan en el caso de que el buffer no tenga suficiente espacio como para contener un mensaje, y el resultado es que el programa aborta. El programador puede decidir entonces que el buffer asignado es muy pequeño, y asignar uno más grande para una ejecución posterior, o bien cambiar el modo de comunicación a otro con menos requerimientos de buffer pero con más riesgo de bloqueo.

2.4 Recepción por encuesta

Las funciones de recepción de mensajes engloban en una operación la sincronización con el emisor (esperar a que haya un mensaje disponible) con la de comunicación (copiar ese mensaje). A veces, sin embargo, conviene separar ambos conceptos. Por ejemplo, podemos estar a la espera de mensajes de tres clases, cada una asociada a un tipo de datos diferente, y la clase nos viene dada por el valor de la etiqueta. Por lo tanto, nos gustaría saber el valor de la etiqueta antes de leer el mensaje. También puede ocurrir que nos llegue un mensaje de longitud desconocida, y resulte necesario averiguar primero el tamaño para así asignar dinámicamente el espacio de memoria requerido por el mensaje.

Las funciones MPI_Probe() y MPI_Iprobe() nos permiten saber si tenemos un mensaje recibido y listo para leer, pero sin leerlo. A partir de la información de estado obtenida con cualquiera de estas "sondas", podemos averiguar la identidad del emisor del mensaje, la etiqueta del mismo y su longitud. Una vez hecho esto, podemos proceder a la lectura real del mensaje con la correspondiente llamada a MPI_Recv() o MPI_Irecv().

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int
*flag, MPI_Status *status);
int MPI_Probe(int source, int tag, MPI_Comm comm,
MPI_Status *status);
```

MPI_Probe() es bloqueante: sólo devuelve el control al proceso cuando hay un mensaje listo. MPI_Iprobe() es no bloqueante: nos indica en el argumento **flag** si hay un mensaje listo o no—es decir, realiza una encuesta.

2.5 Tipos de datos

Los mensajes gestionados por MPI son secuencias de **count** elementos del tipo **datatype**. MPI define una colección de tipos de datos primitivos, correspondientes a los tipos de datos existentes en C. Hay otra colección, distinta, para FORTRAN.

Tabla 1. Tipos de datos MPI

Tipos MPI	Tipos C equivalentes
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_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	Sin equivalente

Aunque una aplicación desarrollada en C trabaja con los tipos de datos habituales, cuando se realice un paso de mensajes habrá que facilitar a MPI un descriptor del tipo equivalente, tal como lo define MPI. La idea de fondo es la siguiente: los procesadores que participan en una aplicación MPI no tienen por qué ser todos iguales. Es posible que existan diferencias en la representación de los datos en las distintas máquinas. Para eliminar los problemas que puedan surgir, MPI realiza, si son necesarias, transformaciones de sintaxis que posibilitan la comunicación en entornos heterogéneos. La excepción la constituyen los datos de tipo MPI_BYTE, que se copian sin más de una máquina a otra.

Aparte de los tipos simples definidos de forma primitiva por MPI, se permite la definición de tipos de usuario, más complejos. Esto lo estudiaremos más adelante (5. Tipos de datos derivados, pág. 17)

2.6 Etiquetas y comunicadores

Todo mensaje que se envía con MPI va etiquetado: parámetro **tag**. El proceso receptor puede elegir entre aceptar mensajes con una cierta etiqueta (parámetro **tag** con un valor concreto), o decir que acepta un mensaje cualquiera (MPI_ANY_TAG). Tras la recepción, se puede saber la etiqueta concreta accediendo a status.MPI_TAG. Las etiquetas permiten diferenciar las diferentes clases de información que pueden intercambiarse los procesos.

La comunicación se produce dentro de un *comunicador* (entorno de comunicación), que es básicamente un conjunto de procesos. El comunicador universal MPI_COMM_WORLD está siempre definido, e incluye a todos los procesos que forman parte de la aplicación. Los comunicadores permiten que diferentes grupos de procesos actúen sin interferir, así como dividir la aplicación en fases no solapadas. MPI garantiza la entrega ordenada de mensajes dentro de un mismo comunicador.

3. Operaciones colectivas

Muchas aplicaciones requieren de la comunicación entre más de dos procesos. Esto se puede hacer combinando operaciones punto a punto, pero para que le resulte más cómodo al programador, y para posibilitar implementaciones optimizadas, MPI incluye una serie de operaciones colectivas:

- Barreras de sincronización
- Broadcast (difusión)
- Gather (recolección)
- Scatter (distribución)

- Operaciones de reducción (suma, multiplicación, mínimo, etc.)
- Combinaciones de todas ellas

Una operación colectiva tiene que ser invocada por *todos* los participantes, aunque los roles que juegen no sean los mismos. La mayor parte de las operaciones requieren la designación de un proceso como *root*, o raíz de la operación.

3.1 Barreras y broadcast

Dos de las operaciones colectivas más comunes son las barreras de sincronización (MPI_Barrier()) y el envío de información en modo difusión (MPI_Broadcast()). La primera de estas operaciones no exige ninguna clase de intercambio de información. Es una operación puramente de sincronización, que bloquea a los procesos de un comunicador hasta que todos ellos han pasado por la barrera. Suele emplearse para dar por finalizada una etapa del programa, asegurándose de que todos han terminado antes de dar comienzo a la siguiente.

```
int MPI_Barrier(MPI_Comm comm);
```

MPI_Broadcast() sirve para que un proceso, el raíz, envíe un mensaje a todos los miembros del comunicador. Esta función ya muestra una característica peculiar de las operaciones colectivas de MPI: todos los participantes invocan la misma función, designando al mismo proceso como raíz de la misma. Una implementación alternativa sería tener una función de envío especial, en modo broadcast, y usar las funciones de recepción normales para leer los mensajes.

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype
datatype, int root, MPI_Comm comm);
```

Este esquema representa el significado de un broadcast. En las filas representamos los procesos de un comunicador, y en las columnas los datos que posee cada proceso. Antes del broadcast, el procesador raíz (suponemos que es el 0) tiene el dato A0. Tras realizar el broadcast, todos los procesos (incluyendo el raíz) tienen una copia de A0.

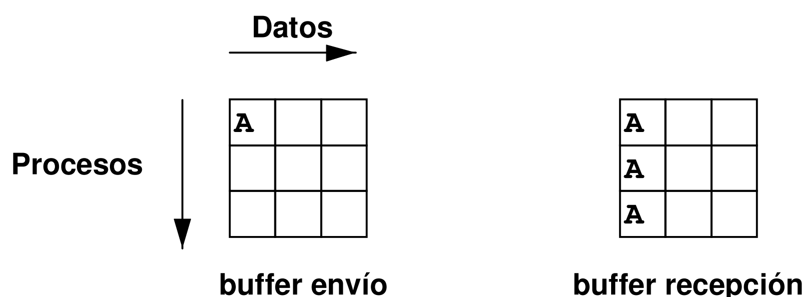

Figura 2. Esquema de la operación colectiva MPI_Broadcast(). “Buffer envío” indica los contenidos de los bufferes de envío antes de proceder a la operación colectiva.“Buffer recepción” indica los contenidos de los bufferes de recepción tras completarse la operación.

3.2 Recolección (gather)

MPI_Gather realiza una recolección de datos en el proceso raíz. Este proceso recopila un vector de datos, al que contribuyen todos los procesos del comunicador con la misma cantidad de datos. El proceso raíz almacena las contribuciones de forma consecutiva.

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm);
```

Figura 3. Esquema de la operación colectiva MPI_Gather().

Sólo el proceso raíz tiene que preocuparse de los parámetros **recbuf**, **recvcount** y **recvtype**. Sin embargo, todos los procesos (raíz incluido) tienen que facilitar valores válidos para **sendbuf**, **sendcount** y **sendtype**.

Existe una versión de la función de recolección, llamada MPI_Gatherv(), que permite almacenar los datos recogidos en forma no consecutiva, y que cada proceso contribuya con bloques de datos de diferente tamaño . La tabla **recvcounts** establece el tamaño del bloque de datos aportado por cada proceso, y **displs** indica cuál es el desplazamiento entre bloque y bloque

```
int MPI_Gatherv(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
int *displs, MPI_Datatype recvtype, int root, MPI_Comm
comm);
```

Muchas veces interesa distribuir a todos los procesos el resultado de una recolección previa. Esto se puede hacer concatenando una recolección con un broadcast. Existen funciones que se encargan de hacer todo esto en un único paso: MPI_Allgather() (si los bloques de datos son de tamaño fijo y se almacenan consecutivamete) y MPI_Allgatherv() (si los tamaños de los datos son variables y/o se almacenan de forma no consecutiva).

```
int MPI_Allgather(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm);

int MPI_Allgatherv(void* sendbuf, int sendcount,
MPI_Datatype sendtype,void* recvbuf, int *recvcounts, int
*displs, MPI_Datatype recvtype, MPI_Comm comm);
```

Figura 4. Esquema de la operación colectiva MPI_Allgather().

3.3 Distribución (scatter)

MPI_Scatter() realiza la operación simétrica a MPI_Gather(). El proceso raíz posee un vector de elementos, uno por cada proceso del comunicador. Tras realizar la distribución, cada proceso tiene una copia del vector inicial. Recordemos que MPI permite enviar bloques de datos, no sólo datos individuales.

```
int MPI_Scatter(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm);
```

Si los datos a enviar no están almacenados de forma consecutiva en memoria, o los bloques a enviar a cada proceso no son todos del mismo tamaño, tenemos la función MPI_Scatterv(), con un interfaz más complejo — pero más flexible.

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int
*displs, MPI_Datatype sendtype, void* recvbuf, int
recvcount, MPI_Datatype recvtype, int root, MPI_Comm
comm);
```

Figura 5. Esquema de la operación colectiva MPI_Scatter().

3.4 Comunicación todos con todos (All-to-all)

La comunicación de todos con todos supone que, inicialmente, cada proceso tiene un vector con tantos elementos como procesos hay en el comunicador. Para i, j y k entre 0 y N-1 (donde N es el número de procesos del comunicador), cada proceso i envía una copia de sendbuf[j] al proceso j, y recibe del proceso k un elemento, que almacena en recvbuf[k]. MPI_Alltoall() equivale, por tanto, a una sucesión de N operaciones de distribución, en cada una de las cuales el proceso i toma el rol de raíz.

```
int MPI_Alltoall(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm);

int MPI_Alltoallv(void* sendbuf, int *sendcounts, int
*sdispls, MPI_Datatype sendtype, void* recvbuf, int
*recvcounts, int *rdispls, MPI_Datatype recvtype,
MPI_Comm comm);
```

Figura 6. Esquema de la operación colectiva MPI_Alltoall().

3.5 Reducción

Una reducción es una operación realizada de forma cooperativa entre todos los procesos de un comunicador, de tal forma que se obtiene un resultado final que se almacena en el proceso raíz.

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm
comm);
```

MPI define una colección de operaciones del tipo MPI_Op que se pueden utilizar en una reducción: MPI_MAX (máximo), MPI_MIN (mínimo), MPI_SUM (suma), MPI_PROD (producto), MPI_LAND (and lógico), MPI_BAND (and bit a bit), MPI_LOR (or lógico), MPI_BOR (or bit a bit), MPI_LXOR (xor lógico), MPI_BXOR (xor bit a bit), etc.

En ocasiones el programador puede necesitar otra operación de reducción distinta, no predefinida. Para ello MPI ofrece la función MPI_Op_create(), que toma como argumento de entrada una función de usuario y devuelve un objeto de tipo MPI_Op que se puede emplear con MPI_Reduce(). Ese objeto se puede destruir más adelante con MPI_Op_free().

```
int MPI_Op_create(MPI_User_function *function, int
commute, MPI_Op *op );
```

```
int MPI_Op_free(MPI_Op *op);
```

La operación a realizar puede ser cualquiera. En general se emplean operaciones conmutativas y asociativas, es decir, cuyo resultado no depende del orden con el que se procesan los operandos. Eso se indica con el valor 1 en el parámetro **commute**. Si la operación no es conmutativa (**commute** = 0) entonces se exige que la reducción se realice en orden de dirección (se empieza con el proceso 0, luego con el 1, con el 2, etc.).

En ocasiones resulta útil que el resultado de una reducción esté disponible para todos los procesos. Aunque se puede realizar un broadcast tras la reducción, podemos combinar la reducción con la difusión usando MPI_Allreduce(). Si el resultado de la reducción es un vector que hay que distribuir entre los procesadores, podemos combinar la reducción y la distribución usando MPI_Reduce_scatter().

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int
count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

```
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int
*recvcounts,MPI_Datatype datatype,MPI_Op op,MPI_Comm
comm);
```

Una variante más de la reducción nos la da MPI_Scan(). Es similar a MPI_Allreduce(), excepto que cada proceso recibe un resultado parcial de la reducción, en vez del final: cada proceso i recibe, en vez del resultado de OP(0..N-1)—siendo N el número total de procesos—el resultado de OP(0..i).

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm );
```

4. Modularidad

MPI permite definir grupos de procesos. Un grupo es una colección de procesos, y define un espacio de direcciones (desde 0 hasta el tamaño del grupo menos 1). Los miembros del grupo tienen asignada una dirección dentro de él. Un proceso puede pertenecer simultáneamente a varios grupos, y tener una dirección distinta en cada uno de ellos.

Un comunicador es un *universo de comunicación*. Básicamente consiste en un grupo de procesos, y un *contexto* de comunicación. Las comunicaciones producidas en dos comunicadores diferentes nunca interfieren entre sí.

El concepto de comunicador se introdujo para facilitar la elaboración de bibliotecas de programas: el programa de un usuario se ejecuta en un comunicador, y las funciones de la biblioteca en otro diferente (posiblemente, con el mismo grupo de procesos, pero con un contexto diferente). Así no hay riesgo de que se mezclen mensajes del usuario con mensajes privados de las funciones de la biblioteca. Además, así tampoco hay problemas a la hora de usar etiquetas.

Describimos ahora algunas de las funciones sobre comunicadores más comunes. Dos de ellas ya han sido presentadas: MPI_Comm_size() para averiguar el tamaño (número de procesos) de un comunicador, y MPI_Comm_rank() para que un proceso obtenga su identificación dentro del comunicador.

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

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

La función MPI_Comm_dup() permite crear un nuevo comunicador, con el mismo grupo de procesos, pero diferente contexto. Se puede usar antes de llamar a una función de una biblioteca. La Figura 7 (izquierda) muestra cómo puede ocurrir que un mensaje del usuario (flecha fina) interfiera con los mensajes propios de una función de biblioteca (flechas gruesas). Si la biblioteca trabaja en un comunicador diferente (derecha) entonces no hay problemas de interferencia.

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm);
```

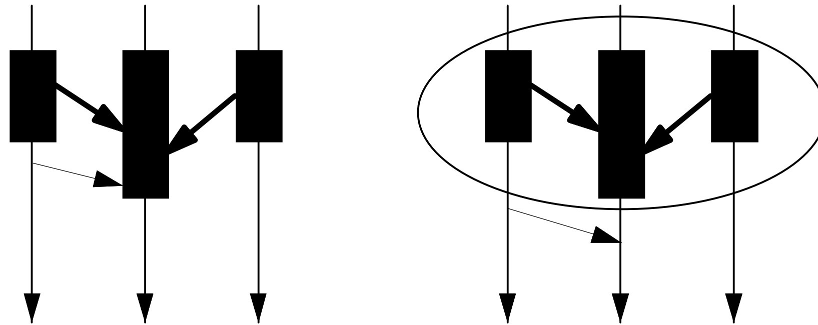

Figura 7. Utilización de un comunicador para aislar una función de biblioteca.

```
…
MPI_Comm_dup (MPI_COMM_WORLD, &newcomm);
/* llamada a función, que se ejecutará dentro */
/* de newcomm */
MPI_Comm_free (&newcomm);
…
```

La operación MPI_Comm_dup(), así como todas las demás que crean comunicadores, son operaciones colectivas: tienen que ser invocadas por todos los procesos implicados, con argumentos compatibles.

MPI_Comm_free() destruye un comunicador que ya no se va a emplear. Por su parte, MPI_Comm_split() crea, a partir de un comunicador inicial, varios comunicadores diferentes, cada uno con un conjunto disjunto de procesos. El **color** determina en qué comunicador queda cada uno de los procesos.

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
MPI_Comm *newcomm);
```

Veamos un ejemplo de creación de comunicadores con MPI_Comm_split(). Partiendo del comunicador universal, definimos dos comunicadores: uno con aquellos procesos que tienen dirección par en el comunicador MPI_COMM_WORLD, y otro con los procesos impares. Cada proceso pertenecerá a dos comunicadores: al universal y al recién creado, y tendrá por lo tanto dos direcciones, una por comunicador.

```
int myglobalid, myotherid, color;
MPI_Comm newcom;

MPI_Comm_rank (MPI_COMM_WORLD, &myglobalid);
if (myglobalid%2 == 0) color = 0;
else color = 1;
MPI_Comm_split (comm, color, myglobalid, &newcom);
MPI_Comm_rank (newcom, &myotherid);

/* operaciones entre pares e impares, por separado */

MPI_Comm_free (&newcom);
```

La Figura 8 muestra una colección de seis procesos que forman inicialmente parte del comunicador universal y que, tras MPI_Comm_split() pasan a formar dos nuevos comunicadores disjuntos.

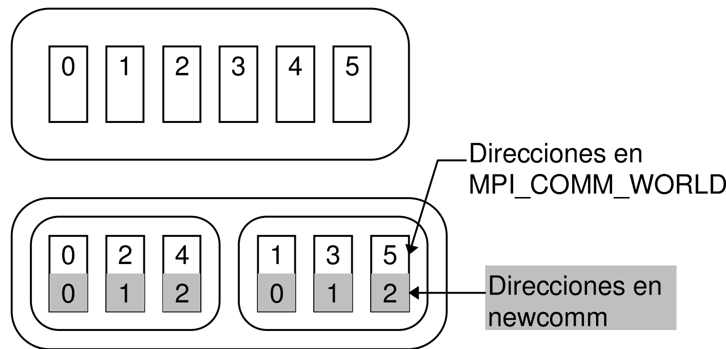

Figura 8. Arriba: una colección de procesos en el comunicador MPI_COMM_WORLD. Abajo, dos comunicadores: uno con los procesos con dirección global par, y otro con los procesos con dirección global impar.

Los procesos que se encuentran en dos grupos diferentes no pueden comunicarse entre sí, a no ser que se cree un *intercomunicador* (los comunicadores vistos hasta ahora se denominan *intracomunicadores*). La función MPI_Intercomm_create() nos permite crear un intercomunicador.

```
int MPI_Intercomm_create(MPI_Comm local_comm, int
local_leader, MPI_Comm peer_comm, int remote_leader, int
tag, MPI_Comm *newintercomm);
```

Todos los procesos de los dos comunicadores que quieren interconectarse deben hacer una llamada a MPI_Intercomm_create() con argumentos compatibles. Interpretar los argumentos de esta función no es trivial. En primer lugar, cada proceso tiene que facilitar el comunicador (**local_comm**), y nombrar a un proceso local como *lider* del intercomunicador (**local_leader**). Luego hay que facilitar un comunicador que englobe a los dos comunicadores que queremos intercomunicar (**peer_comm**; MPI_COMM_WORLD siempre resulta válido), y la dirección de un proceso del otro comunicador que va a actuar como líder (**remote_leader**), teniendo en cuenta que esa dirección corresponde al comunicador global. Con esto ya se tiene tendido un "puente" entre los dos comunicadores. También hay que facilitar un número de etiqueta (**tag**) que esté libre y que el programador no debe emplear para ningún otro propósito. El intercomunicador lo obtenemos en **newintercomm**. Una vez que se tiene un intercomunicador, se puede usar para acceder al comunicador que está "al otro lado". Las direcciones de los procesos receptores que hay que facilitar en las funciones de envío, o las de los procesos emisores que se obtienen tras una función de recepción, hacen referencia a ese "otro lado".

El siguiente ejemplo toma como punto de partida los dos comunicadores creados en el ejemplo anterior: uno con los procesos que en MPI_COMM_WORLD tienen dirección par, y otro con los que tienen en ese comunicador dirección impar. Cada proceso tiene almacenado en newcom el comunicador al que pertenece, y en myotherid su dirección en ese comunicador. Tras crear, de forma colectiva, el intercomunicador, cada proceso del comunicador "par" envía un mensaje a su semejante en el comunicador "impar".

```
if (myglobalid%2 == 0) {
```

```
        MPI_Intercomm_create (newcom, 0, MPI_COMM_WORLD,
             1, 99, &intercomm);
        MPI_Send(msg, 1, type, myotherid, 0,
             intercomm);
}
else {
        MPI_Intercomm_create (newcom, 0, MPI_COMM_WORLD,
             0, 99, &intercomm);
        MPI_Recv(msg, 1, type, myotherid, 0,
             intercomm, &status);
}
```

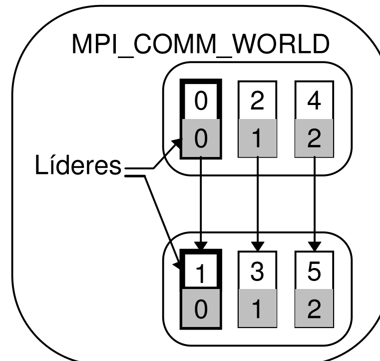

Figura 9. Creación de un intercomunicador entre los dos comunicadores de la Figura 8.

5. Tipos de datos derivados

MPI maneja en todas sus funciones de envío/recepción vectores de tipos simples. En general, se asume que los elementos de esos vectores están almacenados consecutivamente en memoria. En ocasiones, sin embargo, es necesario el intercambio de tipos estructurados (structs), o de vectores no almacenados consecutivamente en memoria (p.ej: envío de una columna de un array, en vez de envío de una fila).

MPI incluye la posibilidad de definir tipos más complejos (objetos del tipo MPI_Datatype), empleando constructores. Antes de usar un tipo de usuario, hay que ejecutar MPI_Type_commit(). Cuando ya no se necesite un tipo, se puede liberar con MPI_Type_free().

```
int MPI_Type_commit(MPI_Datatype *datatype);

int MPI_Type_free(MPI_Datatype *datatype);
```

5.1 Definición de tipos homogéneos

Son tipos homogéneos aquellos tipos construidos en los que todos los elementos constituyentes son del mismo tipo. Se pueden definir tipos homogéneos con dos funciones distintas: MPI_Type_contiguous() y MPI_Type_vector(). La primera versión es la más sencilla, y permite definir un tipo formado por una colección de elementos de un tipo básico, todos ellos del mismo tamaño y almacenados consecutivamente en memoria.

```
int MPI_Type_contiguous (int count, MPI_Datatype oldtype,
```

```
MPI_Datatype *newtype);
```

Newtype es un nuevo tipo que consiste en **count** copias de **oldtype**. Si los elementos constituyentes del nuevo tipo no están almacenados consecutivamente en memoria, sino que están espaciados a intervalos regulares, la función a emplear es MPI_Type_vector().

```
int MPI_Type_vector (int count, int blocklength, int
stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

Newtype es un nuevo tipo que consiste en **count** bloques de datos. Cada bloque consta de **blocklength** elementos del tipo **oldtype**. La distancia entre bloques, medida en múltiplos del tamaño del elemento básico, la da **stride**.

Una llamada a MPI_Type_contiguous(c, o, n) equivale a una llamada a MPI_Type_vector (c, 1, 1, o, n), o a una llamada a MPI_Type_vector (1, c, x, o, n), siendo x un valor arbitrario.

5.2 Definición de tipos heterogéneos

Si nos interesa trabajar con tipos no homogéneos, podemos usar una función más genérica: MPI_Type_struct().

```
MPI_Type_struct (int count, int *array_of_blocklenghts,
MPI_Aint *array_of_displacements, MPI_Datatype
*array_of_types, MPI_Datatype *newtype);
```

Count determina el número de bloques, así como el tamaño de los arrays **array_of_blocklenghts**, **array_of_displacements** y **array_of_types**. Cada bloque i tiene array_of_blocklenghts[i] elementos, está desplazado array_of_displacements[i] bytes del anterior, y es de tipo array_of_types[i].

Veamos un ejemplo. Si B = {2, 1, 3}, D = {0, 16, 26} y T = {MPI_FLOAT, MPI_INT, MPI_CHAR}, entonces MPI_Type_struct(3, B, D, T, newtype) devuelve un tipo que en memoria tiene el aspecto mostrado en la Tabla 2. La primera columna hace referencia a la posición de memoria que ocupa cada campo de una instancia de este tipo de datos, relativa al origen del dato.

Tabla 2. Tipo de datos de ejemplo.

Desplazamiento	Tipo
0	float
4	float
16	int
26	char
27	char
28	char

6. Bibliografía

La mayor fuente de información sobre MPI es la página WWW mantenida por

Argonne National Laboratory en http://www. mcs.anl.gov/mpi/index.html/. En ella podemos encontrar enlaces al texto del estándar, así como diferentes recursos para aprendizaje de MPI. Entre ellos, se encuentra un enlace a la versión electrónica de libro "Designing and building parallel programs", publicado por Ian Foster (y editado por Addison-Wesley).

También podemos encontrar aquí enlaces con implementaciones de dominio público de MPI. Una de las más conocidas es MPICH, desarrollada por Argonne National Laboratory y Mississippi State University, con versiones para múltiples sistemas paralelos (SGI, IBM, Intel, Cray, etc.) y para diferentes redes de estaciones de trabajo (IBM, HP, Sun, etc.).