

31 DE MARZO DE 2017

TEMA 3. PROGRAMACIÓN DISTRIBUIDA

SERVICIOS DE ALTAS PRESTACIONES Y DISPONIBILIDAD

AUTORES:

Jeferson Arboleda Gómez
Pedro Miguel Luzón Martínez
Alberto Mora Plata
Santiago Mora Soler

ÍNDICE

1	Introducción	9
1.1	Consideraciones Previas	9
1.2	Modelo de Paso de Mensajes	9
1.3	¿Qué es MPI?	10
1.4	Estructura general de un programa MPI	11
1.5	Aplicaciones de MPI	11
1.6	Consideraciones	11
2	Conceptos Básicos de MPI	13
2.1	Entorno de ejecución	13
2.2	Identificación de procesos	13
2.3	Hello World	14
2.4	Funciones Básicas	15
2.5	Tipos de datos en C	16
2.6	Comunicación bloqueante	16
2.7	Envío y recepción simultánea	18
3	Comunicación No Bloqueante	20
4	Operaciones de Comunicación Colectiva	22
4.1	Introducción	22
4.2	MPI_Barrier	23
4.3	MPI_Bcast	24
4.4	MPI_Scatter	24
4.5	MPI_Gather	25
4.6	MPI_Allgather	26
4.7	MPI_Alltoall	26
4.8	MPI_Reduce	27
4.9	MPI_Allreduce	29
4.10	Otras operaciones	29
5	Comunicadores y Grupos	30
	Referencias	32
	Anexo I. Lista de tipos de datos de MPI en C	33

ÍNDICE DE FIGURAS

Figura 1.1. Paso de mensajes entre procesos.	9
Figura 1.2: Ejemplo de paso de mensajes.	10
Figura 1.3. Estructura general de un programa MPI.....	11
Figura 2.1. Representación de comunicadores.....	14
Figura 2.2. Código hello_world.c	14
Figura 2.3. Ejecución de Hello_world	15
Figura 2.4. Código ex_send_recv.c.....	17
Figura 2.5. Ejecución de ex_send_recv	18
Figura 2.6. Representación de comunicadores.....	18
Figura 2.7. Ejecución de ex_send_recv_2	18
Figura 2.8: Ejecución del programa de envío y recepción simultánea	19
Figura 3.1: Esquema de paso de mensajes bloqueante	20
Figura 3.2: Esquema de paso de mensajes no bloqueante	21
Figura 4.1. Tipos de operaciones colectivas [6].....	22
Figura 4.2. Barrier [7]	23
Figura 4.3. Barrier - Ejemplo real	23
Figura 4.4. Broadcast	24
Figura 4.5. Broadcast - Ejemplo real	24
Figura 4.6. Scatter – Gather	25
Figura 4.7. Scatter – Gather – Ejemplo real.....	25
Figura 4.8. Allgather.....	26
Figura 4.9. Allgather - Ejemplo real	26
Figura 4.10. Alltoall.....	27
Figura 4.11. Alltoall - Ejemplo real	27
Figura 4.12. Reduce	28
Figura 4.13. Reduce - Ejemplo real	28
Figura 4.14. Allreduce	29
Figura 5.1. Comunicadores	30
Figura 5.2. Comunicadores - Ejemplo real	31

ÍNDICE DE TABLAS

Tabla 1. Equivalencia entre los tipos de datos en C y MPI.....	16
Tabla 2. Operaciones de reducción en MPI	28
Tabla 3. Tipos de datos de MPI en C [6]	33

1 INTRODUCCIÓN

1.1 CONSIDERACIONES PREVIAS

Todo el código fuente de los ejemplos se encuentra disponible en la [cuenta oficial de AirQ](#) en GitHub. En concreto, el repositorio para este proyecto es **mpi-examples**.

1.2 MODELO DE PASO DE MENSAJES

En primer lugar, debemos diferenciar dos conceptos clave en el área de la concurrencia: hilo y proceso. Un proceso se puede definir como una unidad principal de procesamiento gestionada por el sistema operativo. Esta unidad de procesamiento dispone de su propio espacio de direccionamiento en memoria principal. Los procesos pueden tener a su vez múltiples hilos de ejecución o *threads*, todos ellos comparten el mismo espacio de direccionamiento del proceso que los generó.

El paralelismo se centra en las técnicas existentes para permitir la comunicación entre procesos e hilos. Como se explicó en la unidad anterior, los hilos pueden intercambiar información mediante el uso de memoria compartida. En el caso de los sistemas distribuidos, el problema que encontramos es que la comunicación se realiza entre procesos, los cuales no comparten memoria.

Por ello, el modelo de paso de mensajes nace como un paradigma que permite la comunicación entre diferentes procesos distribuidos a través de mensajes. Esta técnica nos permite lograr:

- Sincronización en la ejecución de procesos.
- Intercambio de información entre procesos (sin necesidad de memoria compartida).

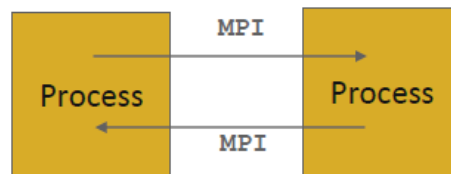


Figura 1.1. Paso de mensajes entre procesos.

La ordenación de un vector de N números enteros es el ejemplo típico de cómo funciona este modelo:

- Si el programa fuera secuencial, el proceso 1 se encargaría de ordenar el vector completo. De esta forma su coste computacional sería del orden $O(N * \log(N))$.
- Si, por el contrario, el programa utilizara paso de mensajes:
 - El proceso 1 le envía al proceso 2 mediante paso de mensajes la mitad del vector.
 - Ambos procesos ordenan su vector acarreado un coste computacional para cada proceso de orden $O(N * \log(N))$.
 - El proceso 2 le envía su vector ordenado al proceso 1 mediante paso de mensajes.
 - Finalmente, este último se encarga de combinar ambos vectores. Al estar ordenados, solo tendrá que acceder una vez a cada elemento de cada vector, esto le acarreará al proceso 1 un coste computacional de orden $O(N)$.
- Por tanto, se deduce que el coste del programa secuencial es mayor que el del programa paralelo, puesto que $O(N * \log(N)) > O(N)$.

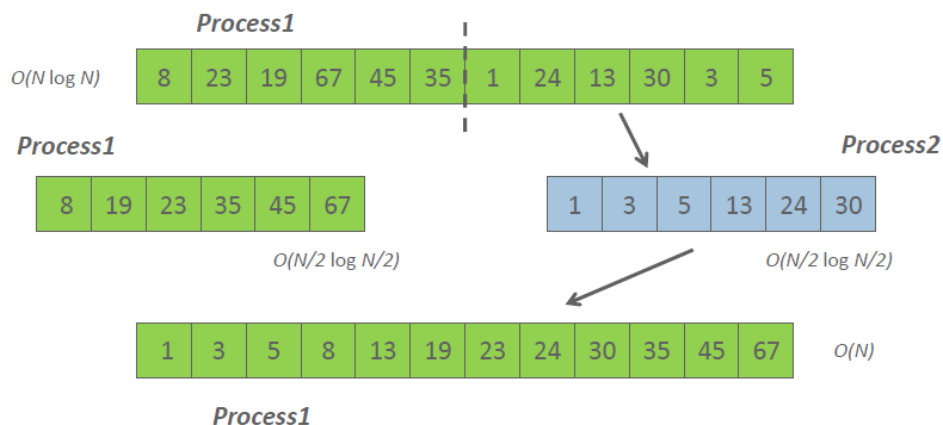


Figura 1.2: Ejemplo de paso de mensajes.

1.3 ¿QUÉ ES MPI?

Una vez se impuso el modelo de paso de mensajes en la programación paralela, cada compañía desarrolló su propio sistema de paso de mensajes (Intel NX, IBM EUI, etc.). Esto trajo consigo problemas de compatibilidad entre sistemas de diferentes fabricantes. Más tarde se desarrollaron los primeros sistemas compatibles (PVM, p4, Chameleon, etc.) pero presentaban una gran complejidad y diferentes problemas técnicos. Por estos motivos se decidió estandarizar el modelo de paso de mensajes a través de MPI Forum.

Message Passing Interface o MPI nace en 1992 como un estándar (MPI-1) de paso de mensajes desarrollado por investigadores, usuarios y las principales compañías que promovían este modelo de programación paralela. Por tanto, MPI no es un lenguaje ni una especificación de compilador, tampoco es una implementación específica ni un producto.

El estándar MPI-1 trajo consigo las siguientes características:

- Funciones básicas para la comunicación (más de 100).
- Comunicaciones Send/Receive bloqueante y no bloqueante.
- Un conjunto de funciones de comunicación colectivas:
 - Broadcast, Scatter, Gather, etc.
- Diferentes tipos de datos.
- Topologías de procesamiento.
- Librerías para C, C++ y Fortran.
- Códigos y clases de error.

Más adelante, surgieron otras versiones de este estándar que trajeron cambios y nuevas características. MPI-2 apareció en el año 2000, y añadía funcionalidades como MPI+threads, MPI-I/O, acceso remoto a memoria, etc. MPI-2.1 (2008) y MPI-2.2 (2009) fueron las siguientes versiones que corrigieron algunos aspectos del estándar. Finalmente, las últimas versiones son MPI-3 (2012) y MPI-3.1 (2015), estas añadieron nuevas características. Actualmente se está trabajando en el desarrollo de la versión MPI-4 del estándar [1].

1.4 ESTRUCTURA GENERAL DE UN PROGRAMA MPI

En la Figura 1.3 se ilustra la estructura típica de un programa que hace uso de MPI.

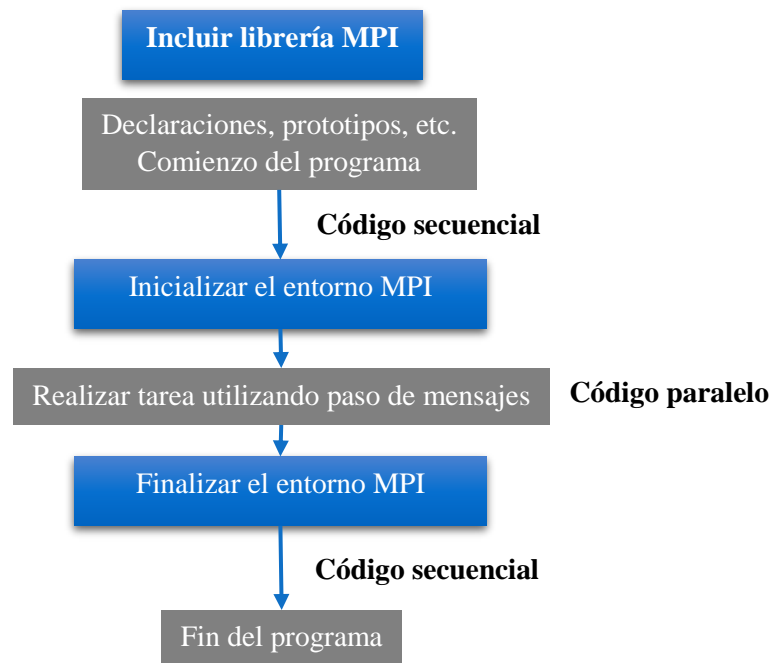


Figura 1.3. Estructura general de un programa MPI

1.5 APLICACIONES DE MPI

MPI es comúnmente aplicado en software desarrollado para las áreas de ciencia e ingeniería que requieren una gran cantidad de potencia de cómputo paralelo y hacen uso de sistemas distribuidos. Algunos ejemplos de sus principales usos son:

- Estudios acerca de la Tierra (atmósfera, climatología, etc.).
- Experimentos físicos (física aplicada, nuclear, cuántica, grandes presiones, etc.).
- Biología (biotecnología, genomas, etc.).
- Química y ciencia molecular.
- Geología y sismología.
- Ingeniería mecánica, eléctrica e informática.
- Matemáticas.

1.6 CONSIDERACIONES

Algunas de las razones por las que se debe utilizar MPI para la comunicación en sistemas distribuidos es la siguiente:

- **Estandarización:** MPI es la única librería de paso de mensajes que puede ser considerada un estándar. Está soportada por todas las plataformas de computación de altas prestaciones.
- **Portabilidad:** No es necesario modificar el código fuente de tu aplicación si deseas trasladarla a otra plataforma que cumpla con el estándar MPI.

- **Oportunidades de rendimiento:** Existen diversas implementaciones desarrolladas por las propias compañías con las que se puede explotar el hardware nativo para optimizar el rendimiento.
- **Funcionalidad:** MPI cuenta con un gran abanico de características.
- **Disponibilidad:** Hay una gran variedad de implementaciones disponibles, de dominio público y provenientes de compañías.
 - **MPICH** es la librería de software libre gratuita de MPI.
 - Las compañías y colaboradores trabajan sobre MPICH y añaden soporte a sus sistemas.
 - Intel MPI, IBM Blue Gene MPI, Cray MPI, Microsoft MPI, MVAPICH, etc.

Una consideración importante a la hora de utilizar MPI es que el programador es el responsable de identificar correctamente el tipo de paralelismo existente en la aplicación e implementar los algoritmos necesarios.

2 CONCEPTOS BÁSICOS DE MPI

2.1 ENTORNO DE EJECUCIÓN

En primer lugar, para la ejecución de programas que usan MPI es necesario disponer de **MPICH**, la cual es una implementación de alto rendimiento y portable de MPI (MPI-1, MPI-2, MPI-3) [2]. Su última versión estable es la 3.2.

En este caso, se va a utilizar **Pk2** (clúster de la UCLM) para probar los ejemplos que se encuentran en este documento, todos ellos escritos en C. En dicho clúster ya se encuentra instalado MPICH2, por lo que sólo habrá que conectarse a él a través de SSH. En caso de que se quiera instalar MPICH2, puede consultarse en su página oficial la [Guía de Instalación](#).

Una vez que se tiene el entorno configurado y se tiene un programa que hace uso de la librería MPI escrito en C, ha de compilarse el código fuente y generar el correspondiente ejecutable [3]. Para ello se utiliza el siguiente comando, que realiza ambas cosas:

```
$ mpicc test.c -o test
```

- **-o**: indica el nombre del ejecutable (por defecto *a.out*)

Por otro lado, para ejecutar de forma local el ejecutable generado anteriormente se ha de utilizar el siguiente comando:

```
$ mpiexec -n <num_processes> ./test
```

- **-n**: indica el número de procesos que se crearán (por defecto 1).

2.2 IDENTIFICACIÓN DE PROCESOS

Para conocer cómo se identifica un proceso cualquiera dentro de nuestra aplicación con MPI, es necesario entender los siguientes conceptos [4]:

- **Comunicador (communicator)**: Grupo o subconjunto de procesos definido a través del tipo **MPI_Comm**. Un proceso puede pertenecer a más de un comunicador.
- **Contexto (color o context)**: Ámbito de paso de mensajes en el que se comunican los procesos de un mismo comunicador. Un mensaje enviado en un contexto sólo conoce ese contexto y no interfiere con mensajes de otros contextos.
- **Rank**: Número entero que identifica de forma única a un proceso dentro de un comunicador comenzando en 0. Un proceso puede tener dos *rank* diferentes para diferentes comunicadores.

Cuando se crea una aplicación con MPI existe un comunicador por defecto conocido como **MPI_COMM_WORLD**, el cual puede observarse en la Figura 2.1 junto a otro que se ha creado, llamado **comm_1**. Se puede observar que los procesos 2, 3, 6 y 7 de **MPI_COMM_WORLD** pertenecen a ambos comunicadores y además poseen un *rank* diferente en cada uno de ellos.

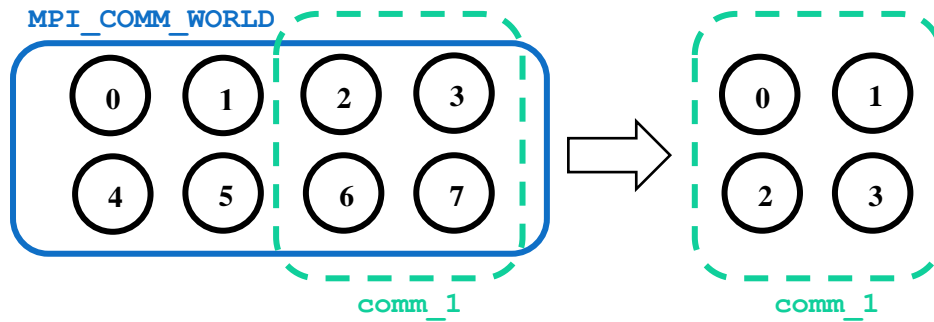


Figura 2.1. Representación de comunicadores

2.3 HELLO WORLD

Antes de conocer qué funciones contiene MPI y qué hacen cada una de ellas, se va hacer uso de los conceptos aprendidos observando cómo sería un “Hello World” en C con MPI [5]. En este programa (ver Figura 2.2) se hará que cada proceso se identifique a través de su *rank* e indique en qué procesador se está ejecutando. Además, Se han resaltado las líneas de código que componen los requisitos mínimos que ha de tener todo programa MPI para que funcione.

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

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Get the number of processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Get the rank of the process
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processes\n",
           processor_name, rank, size);

    // Finalize the MPI environment.
    MPI_Finalize(); return 0;
}
```

Figura 2.2. Código hello_world.c

En la Figura 2.3 se compila y ejecuta el *Hello World* en Pk2. Puede comprobarse que, al no existir ningún tipo de sincronización, los mensajes no tienen por qué estar ordenados.

```
$ mpicc hello_world.c -o hello_world
$ mpiexec -n 4 ./hello_world
```

Output:

```
Hello world from processor avaricia, rank 0 out of 4 processes
Hello world from processor avaricia, rank 3 out of 4 processes
Hello world from processor avaricia, rank 2 out of 4 processes
Hello world from processor avaricia, rank 1 out of 4 processes
```

Figura 2.3. Ejecución de Hello_world

2.4 FUNCIONES BÁSICAS

Como se ha visto en el ejemplo anterior de la Figura 2.2, todo programa MPI ha de comenzar por la función

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

, la cual inicializa el entorno de ejecución de MPI, como por ejemplo la creación del comunicador `MPI_COMM_WORLD` y los *ranks* de todos los procesos. Si se llama más de una vez a esta función durante la ejecución, se lanza un error.

Después de `MPI_Init`, se han realizado dos llamadas a funciones que se usarán en casi todos los programas MPI. En primer lugar, se tiene la función

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

que devuelve el tamaño (*size*) del comunicador `comm`, es decir, el número de procesos que contiene. En el ejemplo, se utiliza el comunicador `MPI_COMM_WORLD`, por lo que se obtiene el número total de procesos que ejecutan el programa. En segundo lugar, se llama a la función

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

que devuelve el *rank* dentro del comunicador `comm` del proceso que está ejecutando el programa. Tras estas llamadas, se tiene por un lado la función

```
int MPI_Get_processor_name (char *name, int *resultlen)
```

que devuelve el actual nombre del procesador donde se está ejecutando el proceso, y por otro, la última llamada a la función

```
int MPI_Finalize()
```

que finaliza y limpia el entorno de ejecución. Al igual que ocurría con `MPI_Init`, no se puede volver a llamar a esta función o se lanzará un error [5].

Si todo funciona correctamente, las funciones anteriores devolverán **MPI_SUCCESS** (cero). Sin embargo, de acuerdo al estándar de MPI, en el caso de error durante una llamada, ésta se aborta [6], por lo que técnicamente no se podría recibir otro código más allá del de éxito en la operación.

2.5 TIPOS DE DATOS EN C

La librería MPI contiene un equivalente *MPI Datatype* a los tipos de datos primitivos de lenguajes como C (en la Tabla 1 se muestran algunos de ellos). El objetivo de crear estos tipos es poder realizar la especificación de los mensajes que se envían entre los procesos sea a alto nivel.

En los ejemplos de este documento sólo se utilizarán tipos de datos básicos. No obstante, es importante destacar que MPI ofrece la posibilidad de crear estructuras de datos propias basadas en tipos de datos primitivos. Estos nuevos tipos de datos se conocen como *Tipos de datos derivados* [6].

Tabla 1. Equivalencia entre los tipos de datos en C y MPI

MPI DATATYPE	EQUIVALENTE EN C
<code>MPI_SHORT</code>	short int
<code>MPI_INT</code>	int
<code>MPI_LONG</code>	long int
<code>MPI_LONG_LONG</code>	long long int
<code>MPI_CHAR</code>	signed char
<code>MPI_UNSIGNED</code>	unsigned int
<code>MPI_UNSIGNED_LONG</code>	unsigned long int
<code>MPI_UNSIGNED_LONG_LONG</code>	unsigned long long int
<code>MPI_FLOAT</code>	float
<code>MPI_DOUBLE</code>	double
<code>MPI_LONG_DOUBLE</code>	long double
<code>MPI_BYTE</code>	char

2.6 COMUNICACIÓN BLOQUEANTE

La comunicación en MPI se asemeja al envío de un email. Un proceso envía (**emisor**) una copia de datos a otro proceso (o grupo de procesos) y el otro proceso lo recibe (**receptor**). Ambos extremos de la comunicación necesitan conocer la siguiente información [3]:

- **Emisor (Sender):** A quién se envía la información (*rank* del receptor), qué tipo de datos se van a enviar (100 enteros, 200 caracteres, etc.) y una *etiqueta* para el mensaje (sería como el asunto del mensaje e indica al receptor qué tipo de mensaje está recibiendo).
- **Receptor (Receiver):** Quién envía la información (*rank* del emisor, o en caso de no conocerlo `MPI_ANY_SOURCE`, que indica cualquier emisor), qué tipo de datos recibe (puede ser información parcial, por ejemplo, hasta 200 enteros) y la *etiqueta* del mensaje (en caso de no conocerse, `MPI_ANY_TAG`).

En MPI ofrece la posibilidad de utilizar tanto comunicación bloqueante como no bloqueante, que se explicará más tarde. En cuanto a la primera, tenemos dos funciones [4]. La función

```
int MPI_Send (const void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)
```

envía los datos almacenados en el buffer **buf**, que contiene **count** elementos del tipo **datatype**, al proceso **dest**(*rank*) con la etiqueta **tag** (entero > 0) dentro del comunicador **comm** [4]. Cuando la función devuelve la ejecución, el dato ha sido enviado, aunque puede que aún no haya sido recibido por el receptor.

La función

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

espera hasta que un mensaje (**buf**, **count**, **datatype**) que coincide con lo que exactamente espera (**source**, **tag**, **comm**) es recibido por el sistema. Como se ha comentado anteriormente, si el receptor no conoce el emisor o la etiqueta del mensaje puede hacerse uso de **MPI_ANY_SOURCE** y **MPI_ANY_TAG** respectivamente. En el caso de recibir más datos que los esperados (**count**) se lanzará un error.

En cuanto al parámetro **status**, es una estructura de datos de tipo **MPI_Status** que contiene información extra del mensaje, aunque se puede utilizar **MPI_STATUS_IGNORE** si no se necesita información adicional. La estructura es:

- **MPI_SOURCE**: *Rank* del emisor. Puede ser utilizado cuando se utiliza **MPI_ANY_SOURCE**.
- **MPI_TAG**: Etiqueta del mensaje.
- **Tamaño del mensaje recibido**: A través de la función

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

EJEMPLO 1 – EJEMPLO DE PASO DE MENSAJES

En la Figura 2.4, se recoge un ejemplo muy sencillo de comunicación bloqueante entre dos procesos en el que el ‘proceso 0’ envía un mensaje al ‘proceso 1’ [3].

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

int main(int argc, char** argv)
{
    int rank, data[100];
    MPI_Status status;

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

    if (rank == 0){
        printf("Process %d: Sending message to process 1\n", rank);
        MPI_Send(data, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1){
        MPI_Recv(data, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        printf("Process %d: Recieving message from process %d\n",
            rank, status.MPI_SOURCE);
    }

    MPI_Finalize(); return 0;
}
```

Figura 2.4. Código *ex_send_recv.c*

```
$ mpicc ex_send_recv.c -o ex_send_recv
$ mpiexec -n 2 ./ex_send_recv
```

Output:

```
Process 0: Sending message to process 1
Process 1: Receiving message from process 0
```

*Figura 2.5. Ejecución de ex_send_recv***EJEMPLO 2 – PASO DE MENSAJES EN CADENA**

En este caso se tienen N procesos ($N > 2$) entre los que se irán enviando un número entero pasado como argumento al programa (por defecto se pasará un 0) [4]. En la Figura 2.6 se ilustra el funcionamiento anterior y en la Figura 2.7 se puede observar la ejecución de este programa, cuyo código fuente puede encontrarse en el fichero *ex_send_recv_2.c* (ver *Consideraciones Previas*).

*Figura 2.6. Representación de comunicadores*

```
$ mpicc ex_send_recv.c -o ex_send_recv_2
$ mpiexec -n 4 ./ex_send_recv_2 33
```

Output:

```
Process 1 got 33 from 0
Process 3 got 33 from 2
Process 2 got 33 from 1
```

*Figura 2.7. Ejecución de ex_send_recv_2***2.7 ENVÍO Y RECEPCIÓN SIMULTÁNEA**

Podemos hacer envío y recepción simultánea en MPI con dos funciones, que son **MPI_Sendrecv** y **MPI_Sendrecv_replace**. A continuación, se explica cada una de ellas:

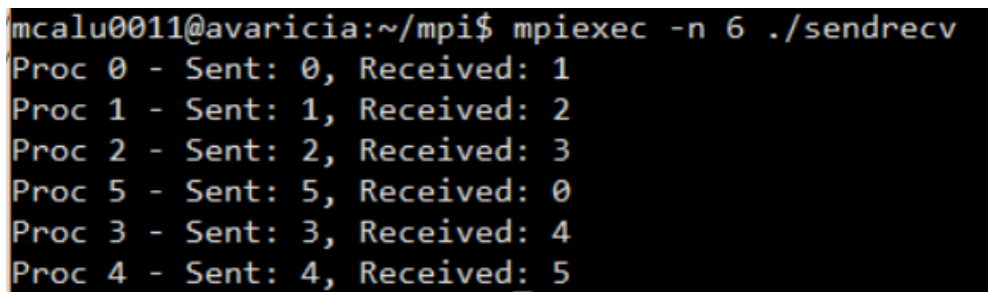
```
int MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype
sendtype, int dest, int sendtag, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,
MPI_Status *status )
```

Envía un mensaje a **dest** y recibe de **source** simultáneamente. Como vemos, combina los parámetros de las funciones **MPI_Send** y **MPI_Recv**, ya que se usan buffers distintos y disjuntos para el emisor y el receptor y permite enviar datos de diferente tipo. No obstante, comparten el mismo comunicador y estado. Esta función es muy útil para patrones de comunicación circulares. Se permite que **source** sea igual a **dest**, es decir, se permite paso de mensajes simultáneo en un mismo nodo.

```
int MPI_Sendrecv_replace( void *buf, int count, MPI_Datatype
datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm
comm, MPI_Status *status )
```

Esta función es igual a la anterior, pero sólo usa un único buffer. Igualmente, el emisor y el receptor tratan con datos del mismo tipo. Debido a que sólo usa un buffer, los datos enviados son reemplazados por los recibidos. Como en el caso anterior, el destino puede ser igual al origen.

El código del programa de ejemplo *sendrecv.c* (ver *Consideraciones Previas*) es un envío circular de mensajes. Un número de procesos (dispuestos de forma circular) envían su id. Cada proceso envía su id al proceso de su izquierda y recibe el id del proceso del de la derecha al mismo tiempo. Para ello usamos **MPI_Sendrecv**, ya que disponemos de dos buffers para agilizar la ejecución del programa. En el fichero *sendrecv2.c* se realiza la misma comunicación circular, pero usando **MPI_Send** y **MPI_Recv** por separado, mientras que en el programa *sendrecv3.c* se usa la instrucción **MPI_Sendrecv_replace**.



```
mcalu0011@avaricia:~/mpi$ mpiexec -n 6 ./sendrecv
Proc 0 - Sent: 0, Received: 1
Proc 1 - Sent: 1, Received: 2
Proc 2 - Sent: 2, Received: 3
Proc 5 - Sent: 5, Received: 0
Proc 3 - Sent: 3, Received: 4
Proc 4 - Sent: 4, Received: 5
```

Figura 2.8: Ejecución del programa de envío y recepción simultánea

3 COMUNICACIÓN NO BLOQUEANTE

Las funciones que se han visto hasta ahora, **MPI_Send** y **MPI_Recv**, son bloqueantes. **MPI_Send** no termina hasta que el buffer está vacío (es decir, hasta que haya terminado de enviar todo y esté libre para usarlo de nuevo). Por otra parte, **MPI_Recv** no termina hasta que el buffer está lleno (ha terminado de recibir todo). Este esquema de comunicación bloqueante es simple de implementar, pero puede llevar a interbloqueos muy fácilmente.

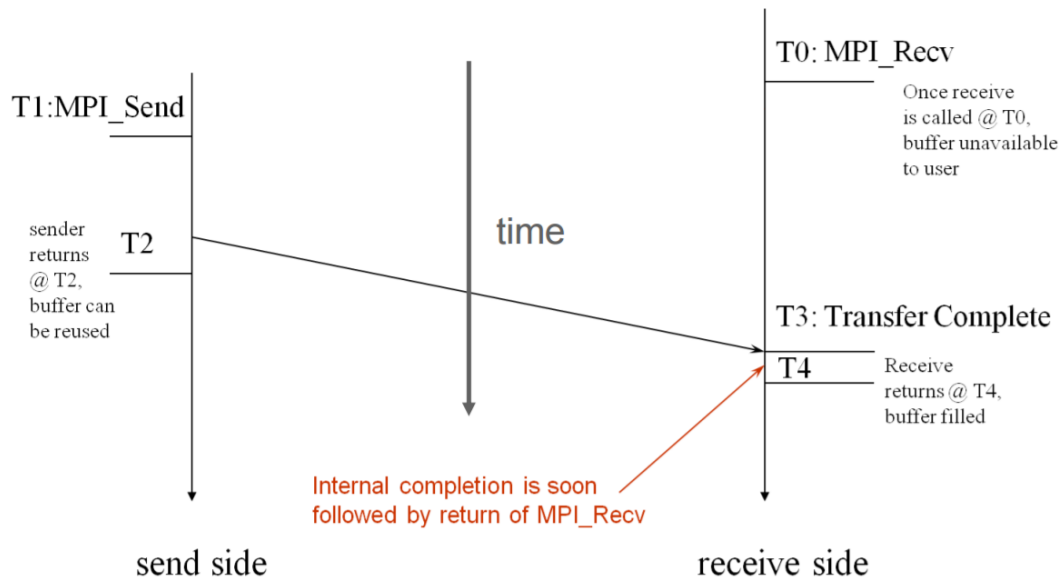


Figura 3.1: Esquema de paso de mensajes bloqueante

La comunicación no bloqueante permite el solapamiento entre computación (procesamiento de datos) y comunicación, mejorando así el rendimiento del sistema. Las funciones no terminan hasta que toda la comunicación se completa, sino que devuelven el control inmediatamente al iniciar las operaciones de paso de mensajes. Devuelven un manejador de petición que se puede consultar o usarse para esperar a que la operación se complete.

Las funciones asíncronas por excelencia son **MPI_Isend**, **MPI_Irecv**, **MPI_Test** y **MPI_Wait**.

```
int MPI_Isend( buffer, count, datatype, dest, tag, comm, request)
```

Inicia el envío y devuelve el control antes de que el buffer se haya vaciado.

```
int MPI_Irecv( buffer, count, datatype, src, tag, comm, request)
```

Inicia la recepción y devuelve el control antes de que el buffer se haya llenado.

```
int MPI_Test( request, flag, status )
```

Consulta el estado de **request** (el manejador generado por **MPI_Isend** y **MPI_Irecv**). Un valor 0 en **flag** indica que la operación se completó con éxito. En ese caso, la función libera **request** e inicializa **status**.

Para consultar varias operaciones a la vez se usa `int MPI_Testall (int count, MPI_Request *array_of_requests, int *flag, MPI_Status *array_of_statuses)`.

```
int MPI_Wait(request, status)
```

Espera (bloquea) hasta que la operación perteneciente al manejador **request** finaliza. Para espera múltiple se usa `int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])`.

Si se está esperando un mensaje, no es necesario bloquear directamente con **MPI_Recv**, sino que se puede usar **MPI_Iprobe**, que únicamente bloquea si hay mensajes para recibir pertenecientes a **source**. En ese caso, **flag** tendrá un valor distinto de 0, y podremos proceder a usar **MPI_Recv** para recibirlos. Su sintaxis es:

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

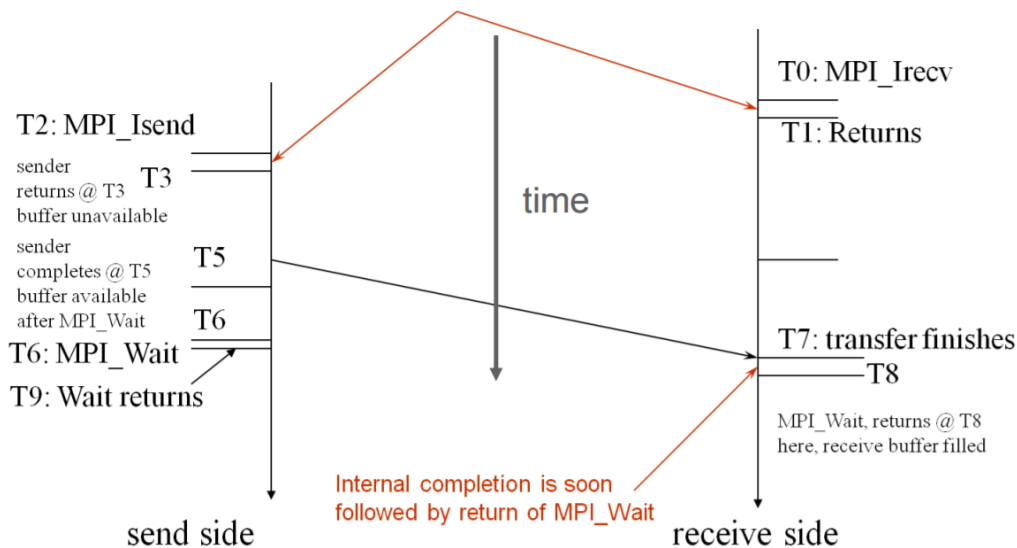


Figura 3.2: Esquema de paso de mensajes no bloqueante

Dondequiera que se use **MPI_Send** o **MPI_Recv** se pueden usar las combinaciones **MPI_Isend/MPI_Wait** y **MPI_Irecv/MPI_Wait**. Se pueden combinar funciones bloqueantes y no bloqueantes. Es decir, podemos hacer envío bloqueante y recepción no bloqueante y viceversa. Hay que tener especial cuidado de no cambiar los parámetros de las funciones no bloqueantes antes de que las operaciones se completen totalmente. Si por ejemplo modificamos los datos que enviamos con **MPI_Isend** justo después de la llamada a la función, será incierta la recepción de un dato u otro en el destino, ya que devuelve el control antes de completar el envío.

El fichero *nobloq.c* (ver *Consideraciones Previas*) implementa una comunicación circular usando paso de mensajes no bloqueante.

4 OPERACIONES DE COMUNICACIÓN COLECTIVA

4.1 INTRODUCCIÓN

Una operación colectiva es un concepto de la computación paralela en el cual se envían o se reciben datos simultáneamente entre varios nodos. En la Figura 4.1 se muestran ejemplos de operaciones colectivas.

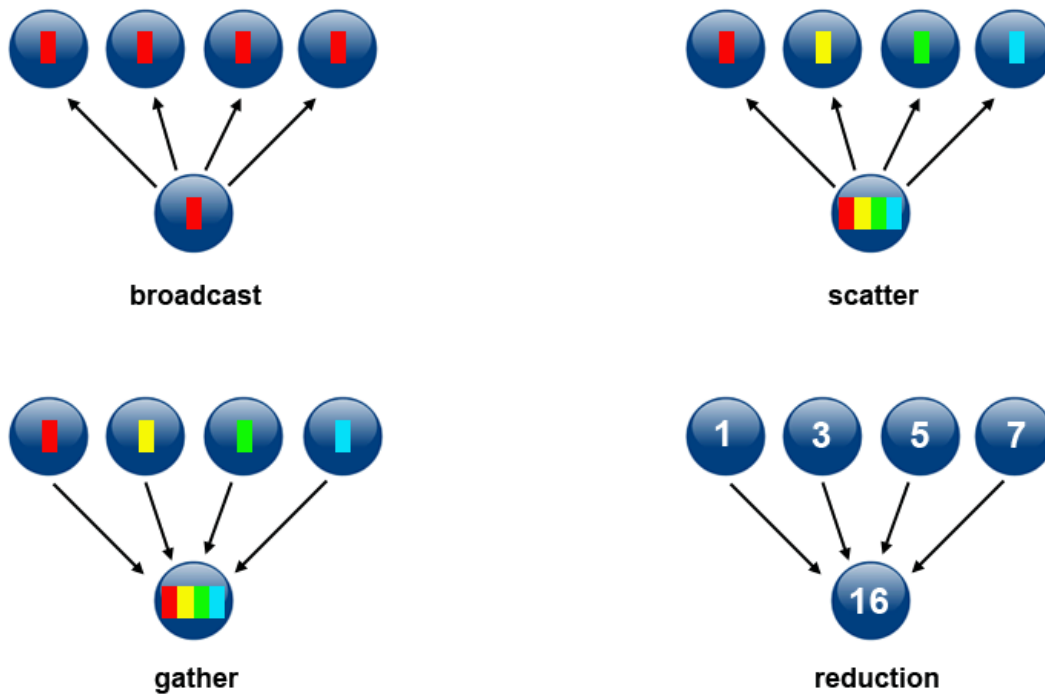


Figura 4.1. Tipos de operaciones colectivas [6]

Existen tres tipos de operaciones colectivas:

- **Sincronización.** Los procesos esperan hasta que todos los miembros del grupo alcanzan una barrera de sincronización.
- **Movimiento de datos.** Envío de datos entre procesos.
- **Reducciones.** Un miembro del grupo recoge datos de otros miembros del grupo y ejecuta una operación (suma, hallar el máximo, etc.) sobre esos datos.

El alcance de estas operaciones se define de la siguiente manera:

- Estas operaciones deben involucrar a todos los procesos de un comunicador.
- Todos los procesos, por defecto, son miembros del comunicador *MPI_COMM_WORLD*. Además, se pueden definir comunicadores adicionales.
- Si algún proceso no participa en la operación se producirá un comportamiento inesperado, por lo que es responsabilidad del programador que todos los procesos de un comunicador participen en las operaciones colectivas.

A la hora de utilizar operaciones colectivas, existen una serie de consideraciones a tener en cuenta:

- Las operaciones colectivas no utilizan *tags* de mensajes.
- Sólo se pueden utilizar con los tipos de datos predefinidos.

4.2 MPI_BARRIER

Esta operación bloquea todos los procesos en un comunicador hasta que todos la alcanzan. En la Figura 4.2 se muestra un esquema de cómo funciona esta operación [6].

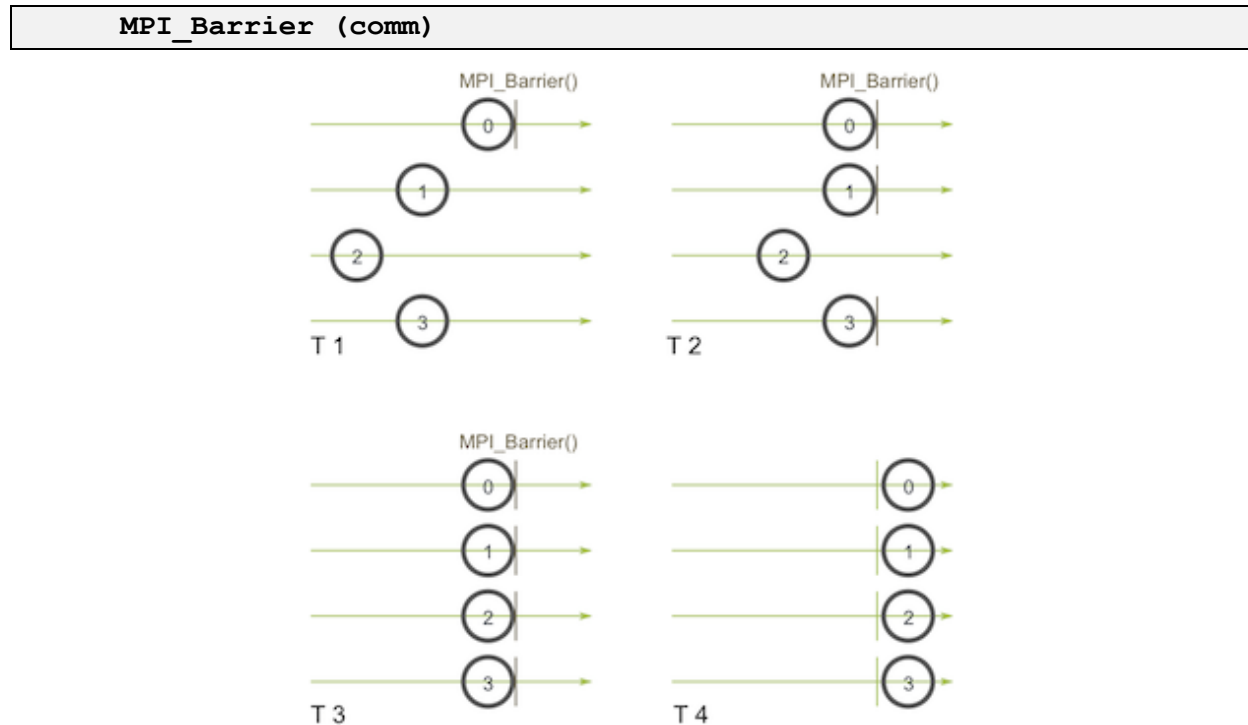


Figura 4.2. Barrier [7]

En el repositorio del proyecto se encuentra un ejemplo de esta operación (**barrier.c**, ver *Consideraciones Previas*). En él, todos los procesos imprimen un *hello world* por pantalla, y luego uno de ellos se queda parado durante 1s. Así, como se utiliza MPI_Barrier, todos los procesos del comunicador tienen que esperar a dicho proceso para poder finalizar su ejecución. (Figura 4.3).

```
jag ... > repos > airq > mpi-examples > ./mpi_exec.sh collective_comm/barrier.c 4
Hello, world. I am 1 of 4
Hello, world. I am 0 of 4
Hello, world. I am 2 of 4
Hello, world. I am 3 of 4
root process: wait for me...
Bye, world. I was 0 of 4
Bye, world. I was 1 of 4
Bye, world. I was 3 of 4
Bye, world. I was 2 of 4
```

Figura 4.3. Barrier - Ejemplo real

4.3 MPI_Bcast

Operación de movimiento de datos. Envía un mensaje desde un proceso a todos los procesos del grupo. En la Figura 4.4 se muestra un diagrama de esta operación [6].

MPI_Bcast (*&buffer, count, datatype, root, comm*)

Parámetros:

- Buffer. Contiene los datos a enviar.
- Count. Número de entradas en el buffer.
- Datatype. Tipo de datos que se envían.
- Root. *Rank* del proceso que hace el *broadcast*.
- Comm. Comunicador.

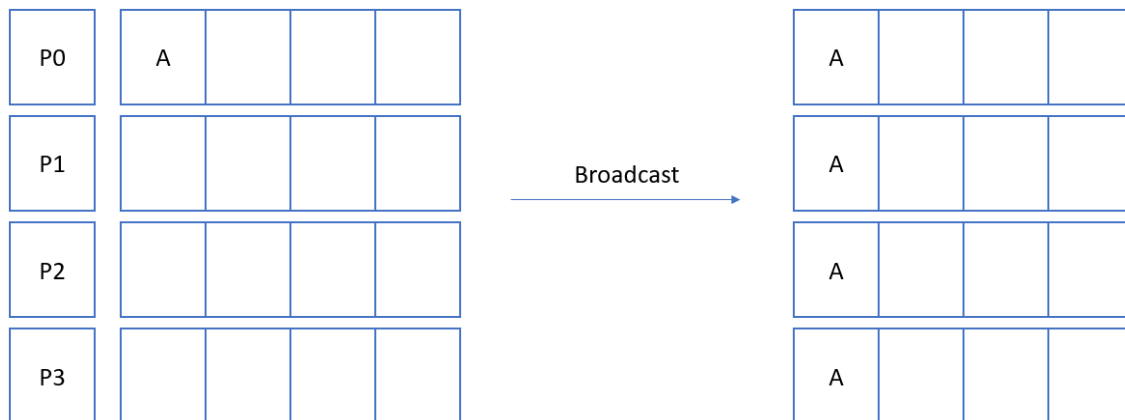


Figura 4.4. Broadcast

En el repositorio del proyecto se encuentra un ejemplo de esta operación (*bcast.c*, ver *Consideraciones Previas*). En él, el proceso *root* envía el dato “100” al resto de procesos (Figura 4.5).

```
mcalu0001@avaricia:~/mpi_examples$ ./mpi_exec.sh bcast.c 4
Process 0 broadcasting data 100
Process 0 received data 100 from root process
Process 1 received data 100 from root process
Process 3 received data 100 from root process
Process 2 received data 100 from root process
```

Figura 4.5. Broadcast - Ejemplo real

4.4 MPI_SCATTER

Operación de movimiento de datos. Distribuye mensajes distintos desde un proceso a todos los procesos del grupo. En la Figura 4.6 se muestra un diagrama de esta operación [6].

MPI_Scatter
(*&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm*)

Parámetros:

- Sendbuf. Contiene los datos a enviar.
- Sendcount. Número de elementos que se envía a cada proceso.
- Sendtype. Tipo de datos.
- Recvcount. Número de elementos en el buffer de recepción.
- Recvtype. Tipo de datos del buffer de recepción.

4.5 MPI_GATHER

Operación de movimiento de datos. Recoge distintos mensajes desde cada proceso del grupo y los envía a un único proceso. Esta operación es la inversa de MPI_Scatter. En la Figura 4.6 se muestra un diagrama de esta operación [6].

MPI_Gather
 (&sendbuf, sendcnt, sendtype, &recvbuf, recvcount, recvtype, root, comm)

Los parámetros son iguales que en la operación anterior y se mantienen en las siguientes operaciones salvo que se indique lo contrario.

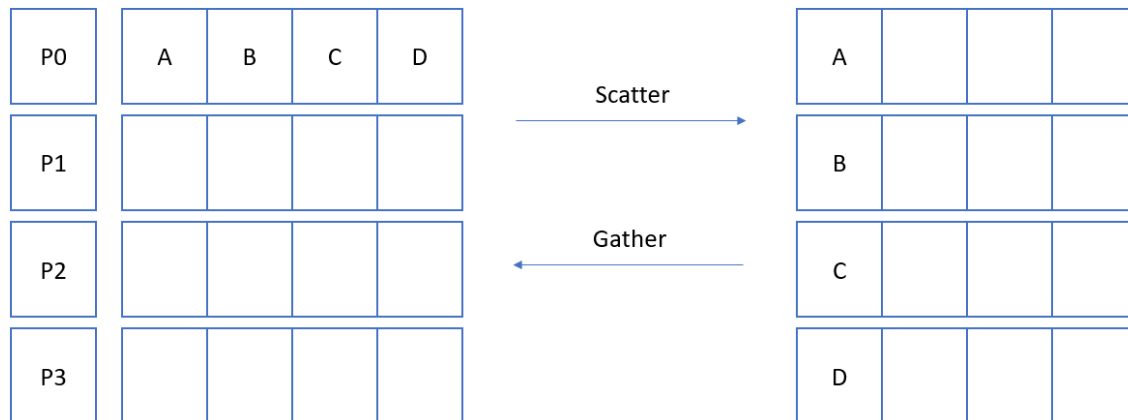


Figura 4.6. Scatter – Gather

En el repositorio del proyecto se encuentra un ejemplo de esta operación (*scatter_gather.c*, ver *Consideraciones Previas*). En él, el proceso root genera una lista de números aleatorios, envía un subconjunto de igual tamaño a cada proceso del grupo y estos devuelven la media aritmética de dicho subconjunto. La media global puede obtenerse como la media de los subconjuntos. (Figura 4.7).

```
jag ... > data > repos > mpi_examples ./mpi_exec.sh scatter_gather.c 4
Process 0: Data = 5.00 1.00 1.00
Process 1: Data = 6.00 5.00 0.00
Process 2: Data = 8.00 6.00 0.00
Process 3: Data = 6.00 4.00 3.00
Process 0: Avg of subset is 2.333333
Process 2: Avg of subset is 4.666667
Process 3: Avg of subset is 4.333333
Process 1: Avg of subset is 3.666667
Process 0: Avg of all elements is 3.750000
Process 0: Avg of original data is 3.750000
```

Figura 4.7. Scatter – Gather – Ejemplo real

4.6 MPI_Allgather

Operación de movimiento de datos. Concatena todos los datos de un grupo. Equivale a que cada proceso del grupo ejecute un *broadcast*, es decir, todos los procesos acaban teniendo sus datos y los de todos los demás. En la Figura 4.8 se muestra un diagrama de esta operación [6].

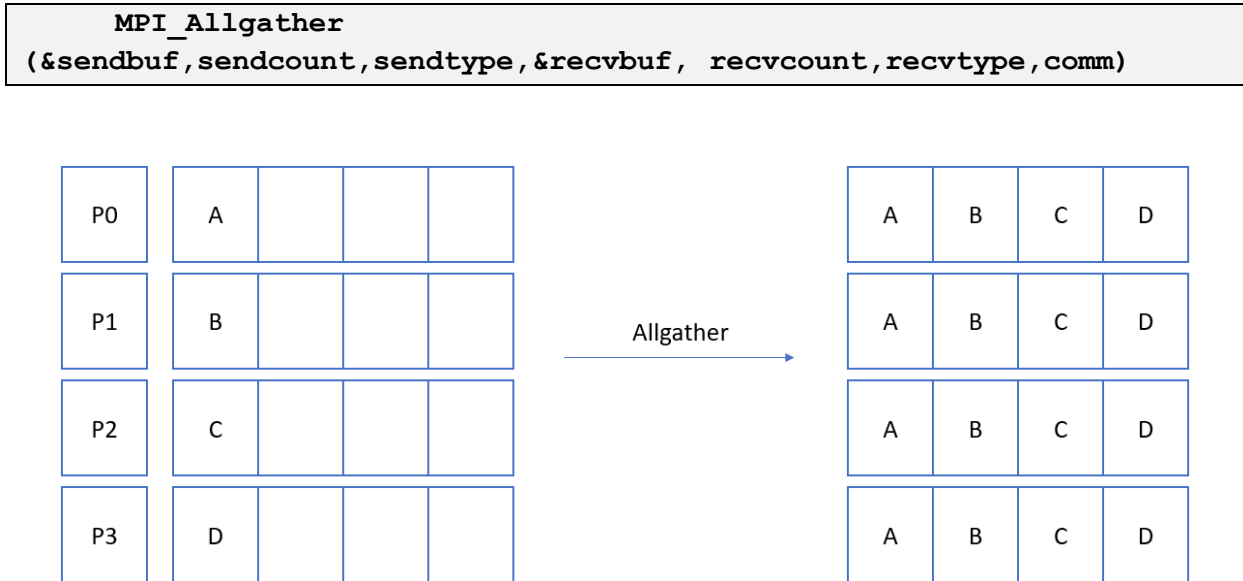


Figura 4.8. Allgather

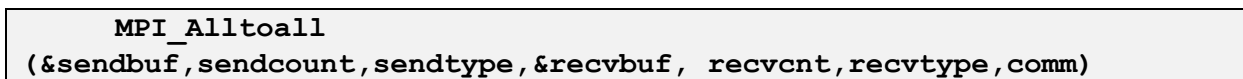
En el repositorio del proyecto se encuentra un ejemplo de esta operación (*allgather.c*, ver *Consideraciones Previas*). En él, el proceso *root* hace un `MPI_Scatter` al resto de procesos, y después se utiliza `MPI_Allgather` para que todos los procesos obtengan los mismos datos. Finalmente, cada proceso calcula la media aritmética de sus datos, que debe ser igual al del resto (Figura 4.9).

```
jag ... > data > repos > mpi_examples ./mpi_exec.sh allgather.c 4
Process 0: Data = 1.00 6.00 5.00
Process 1: Data = 0.00 8.00 6.00
Process 2: Data = 0.00 6.00 4.00
Process 3: Data = 3.00 8.00 4.00
Process 0: Avg of all elements is 4.250000
Process 1: Avg of all elements is 4.250000
Process 3: Avg of all elements is 4.250000
Process 2: Avg of all elements is 4.250000
```

Figura 4.9. Allgather - Ejemplo real

4.7 MPI_Alltoall

Operación de movimiento de datos. Cada proceso del grupo ejecuta un `MPI_Scatter`, enviando un mensaje distinto a cada miembro del grupo. En la Figura 4.10 se muestra un diagrama de esta operación [6].



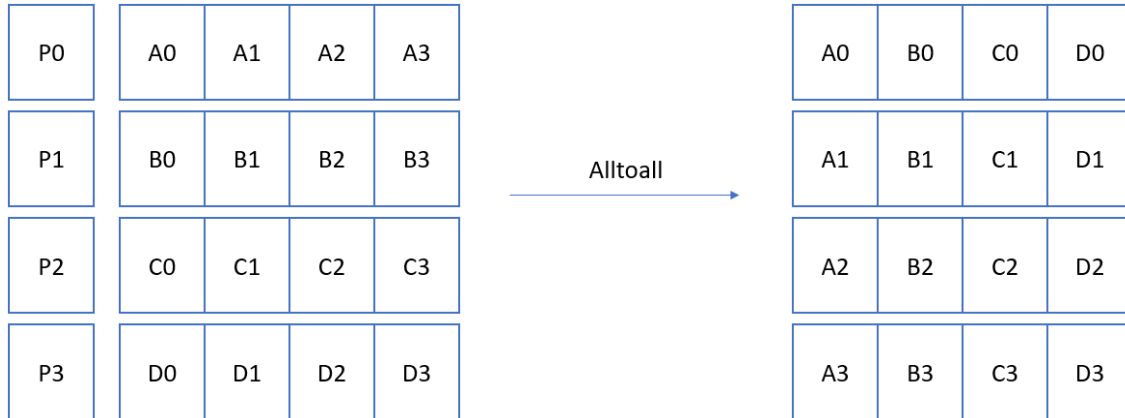


Figura 4.10. Alltoall

En el repositorio del proyecto se encuentra un ejemplo de esta operación (*alltoall.c*, ver *Consideraciones Previas*). En él, se presenta el mismo ejemplo de la Figura 4.10, donde cada proceso genera un *array* de datos en el que cada dato se forma de la siguiente manera:

- Primer dígito: *Rank* del proceso.
- Segundo dígito: grupo del proceso.
- Tercer y cuarto dígito: contador desde 0 hasta N - 1 procesos.

En la Figura 4.11 se muestran los datos antes de realizar la operación y cómo quedarían distribuidos después de hacer la operación.

```
jag ... > repos > airq > mpi-examples > ./mpi_exec.sh collective_comm/alltoall.c 4
Process 0 : Before = 0000 0100 0200 0300
Process 1 : Before = 1000 1100 1200 1300
Process 2 : Before = 2000 2100 2200 2300
Process 3 : Before = 3000 3100 3200 3300
Process 0 : After  = 0000 1000 2000 3000
Process 1 : After  = 0100 1100 2100 3100
Process 2 : After  = 0200 1200 2200 3200
Process 3 : After  = 0300 1300 2300 3300
```

Figura 4.11. Alltoall - Ejemplo real

4.8 MPI_REDUCE

Operación de computación colectiva. Aplica una operación de reducción a todos los procesos de un grupo y guarda el resultado en uno de ellos. En la Figura 4.12 se muestra un diagrama de esta operación [6].

MPI_Reduce (&sendbuf, &recvbuf, count, datatype, op, root, comm)

Ahora aparece un nuevo parámetro *op*, que representa la operación que se va a realizar sobre los datos. En la Tabla 2 se muestran las principales operaciones de MPI [8].

NOMBRE DE LA OPERACIÓN	SIGNIFICADO
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Sumatorio
MPI_PROD	Producto
MPI_LAND	AND lógico
MPI_BAND	Bit-wise AND
MPI_LOR	OR lógico
MPI_BOR	Bit-wise OR
MPI_LXOR	XOR lógico
MPI_BXOR	Bit-wise XOR
MPI_MAXLOC	Valor máximo y localización
MPI_MINLOC	Valor mínimo y localización

Tabla 2. Operaciones de reducción en MPI

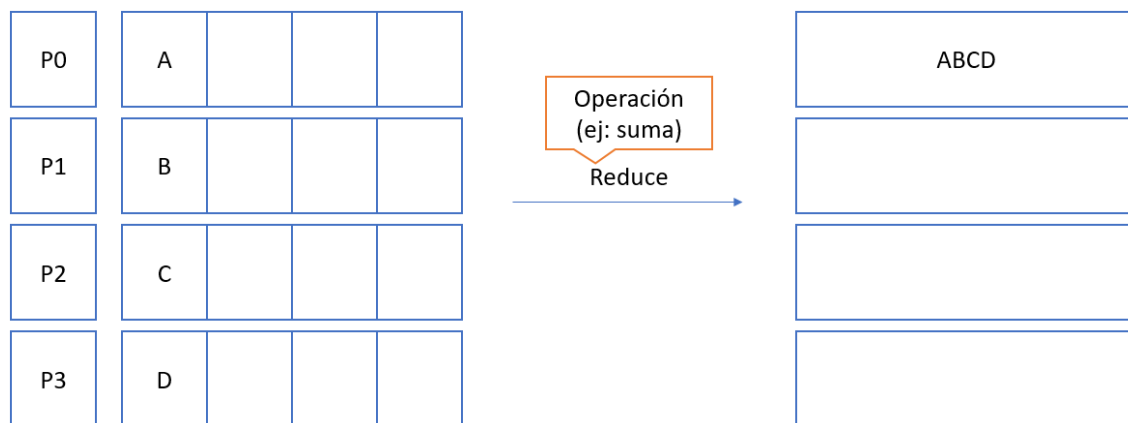


Figura 4.12. Reduce

En el repositorio del proyecto se encuentra un ejemplo de esta operación (**reduce.c**, ver *Consideraciones Previas*). En él, cada proceso genera un *array* de números aleatorios, y luego se aplica una reducción con la operación `MPI_SUM`, de manera que al final es posible calcular la media aritmética sobre el total de los datos (Figura 4.13).

```
jag ... > data > repos > mpi_examples > ./mpi_exec.sh reduce.c 4
Process 0: Data = 5.00 1.00 1.00
Process 1: Data = 3.00 5.00 1.00
Process 2: Data = 3.00 5.00 1.00
Process 3: Data = 3.00 5.00 1.00
Process 0: Local sum = 7.000000, avg = 2.333333
Process 1: Local sum = 9.000000, avg = 3.000000
Process 2: Local sum = 9.000000, avg = 3.000000
Process 3: Local sum = 9.000000, avg = 3.000000
Process 0: Total sum = 34.000000, avg = 2.833333
```

Figura 4.13. Reduce - Ejemplo real

4.9 MPI_ALLREDUCE

Operación de computación colectiva y de movimiento de datos. Aplica una operación de reducción a todos los procesos de un grupo y guarda el resultado en todos ellos. En la Figura 4.14 se muestra un diagrama de esta operación [6].

```
MPI_Reduce_scatter(&sendbuf, &recvbuf, recvcount, datatype, op, comm)
```

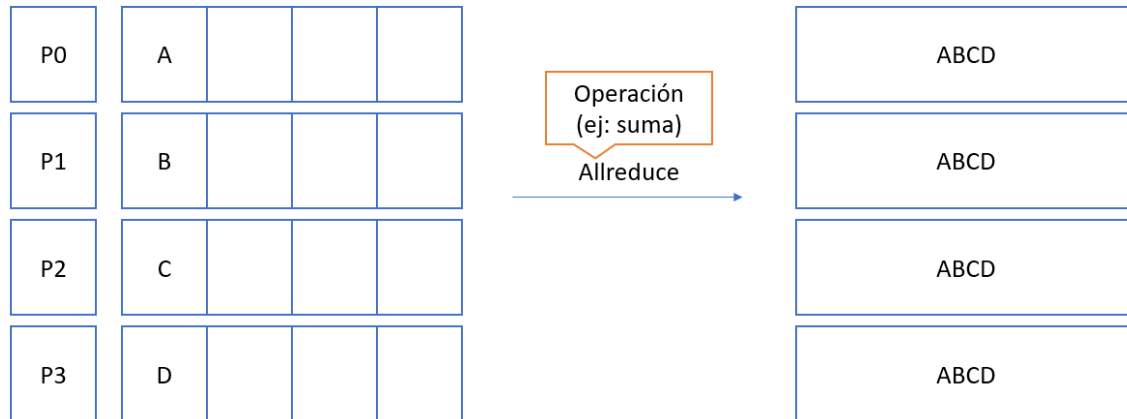


Figura 4.14. Allreduce

Más adelante se verá un ejemplo en el que se utiliza esta operación (*comm.c*, ver *Consideraciones Previas*).

4.10 OTRAS OPERACIONES

Cabe destacar que existen otras operaciones: `MPI_Reduce_scatter`, `MPI_Scan`. Algunas de estas operaciones las veremos en ejemplos posteriores o en la sesión de prácticas.

5 COMUNICADORES Y GRUPOS

Un grupo es un conjunto ordenado de procesos. Dentro de cada grupo, cada proceso es asociado con un *Rank* único, que es un número entero. Estos *ranks* van desde cero hasta $N - 1$, donde N es el número de procesos en el grupo. Por último, un grupo siempre se asocia a un comunicador.

Un comunicador encapsula a un grupo de procesos que pueden comunicarse entre ellos. Todos los mensajes MPI deben especificar un comunicador. Simplificándolo mucho, un comunicador es una etiqueta que debe ser incluida en todas las llamadas MPI. Por defecto, existe un comunicador que incluye a todos los procesos: *MPI_COMM_WORLD*.

En la Figura 5.1 se muestra un ejemplo de cómo pueden utilizarse los comunicadores y grupos [6].

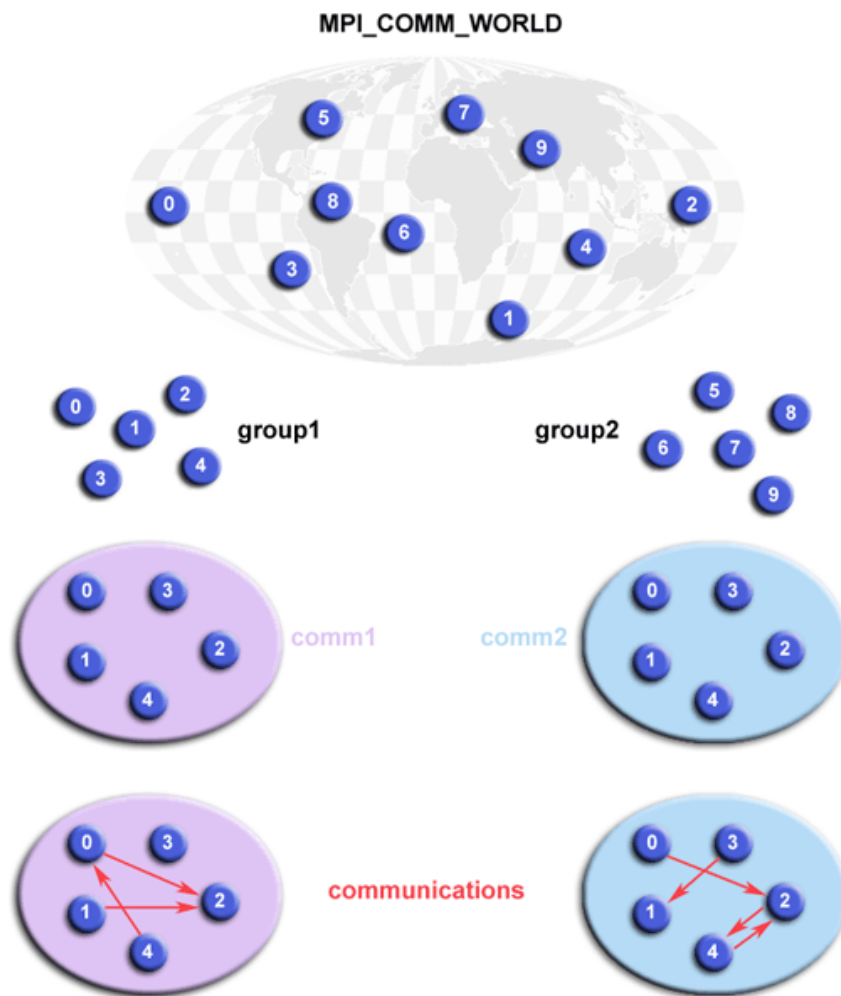


Figura 5.1. Comunicadores

En el repositorio del proyecto se encuentra un ejemplo de esta operación (*comm.c*, ver *Consideraciones Previas*). En él, se generan dos grupos de procesos, de manera parecida a la Figura 5.1. Una vez tenemos dos comunicadores, ejecutamos una operación *MPI_Allreduce* con los *ranks* de los procesos, para que todos los procesos dentro de cada comunicador tengan todos los datos. En esta operación de reducción, se

suman los *ranks* de los procesos, de manera que cada proceso debe obtener la suma de su *rank* y el del resto de procesos en su grupo (Figura 5.2).

```
jag ... > data > repos > mpi_examples > ./mpi_exec.sh comm.c 4  
world_rank = 2 newrank = 0 recvbuf = 5  
world_rank = 3 newrank = 1 recvbuf = 5  
world_rank = 1 newrank = 1 recvbuf = 1  
world_rank = 0 newrank = 0 recvbuf = 1
```

Figura 5.2. Comunicadores - Ejemplo real

REFERENCIAS

- [1] MPI Forum, «MPI Forum website,» [En línea]. Available: <http://mpi-forum.org/>. [Último acceso: Marzo 2017].
- [2] MPICH, «MPICH Overview | MPICH,» [En línea]. Available: <https://www.mpich.org/about/overview/>. [Último acceso: Marzo 2017].
- [3] P. B. T. H. K. R. W. B. y X. Z. , «Introduction to MPI. Guides | MPICH,» 6 Junio 2014. [En línea]. Available: <https://www.mpich.org/documentation/guides/>. [Último acceso: Marzo 2017].
- [4] J. M. M. Ruiz, «Introducción a la Interfaz de paso de mensajes (MPI),» [En línea]. Available: http://lsi.ugr.es/jmantas/pdp/teoria/descargas/PDP_MPI.pdf. [Último acceso: Marzo 2017].
- [5] W. Kendall, «MPI Hello World | MPI Tutorial,» [En línea]. Available: <http://mpitutorial.com/tutorials/mpi-hello-world/>. [Último acceso: Marzo 2017].
- [6] Lawrence Livermore National Laboratory, «Message Passing Interface,» [En línea]. Available: <https://computing.llnl.gov/tutorials/mpi>. [Último acceso: Febrero 2017].
- [7] W. Kendall, «MPI Broadcast and Collective Communication,» [En línea]. Available: <http://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/>. [Último acceso: Marzo 2017].
- [8] mpi-forum, «Predefined reduce operations,» [En línea]. Available: <http://mpi-forum.org/docs/mpi-1.1/mpi-1.1-html/node78.html>. [Último acceso: Abril 2017].
- [9] University of Mary Washington, «Parallel sorting,» [En línea]. Available: <http://cs.umw.edu/~finlayson/class/fall14/cpsc425/notes/18-sorting.html>. [Último acceso: Marzo 2017].

ANEXO I. LISTA DE TIPOS DE DATOS DE MPI EN C

En la Tabla 3 se recoge la lista completa de todos los tipos de datos que la librería MPI contiene para programas escritos en C. Junto a cada uno de ellos se indica su descripción.

Tabla 3. Tipos de datos de MPI en C [6]

TIPO DE DATO	DESCRIPCIÓN
MPI_CHAR	signed char
MPI_WCHAR	wchar_t - wide character
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG	
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX	
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_C_BOOL	_Bool
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack