



Message Passing Interface

Define una API (application programming interface) para programación de máquinas de memoria distribuida. Es un estándar de facto.

MPI-Forum (<http://www.mpi-forum.org/>) elabora especificaciones de MPI para C y Fortran. Involucra a 80 personas de 40 organizaciones, principalmente de EEUU y Europa. Participan la mayoría de los fabricantes de computadoras paralelas, investigadores de universidades, laboratorios de gobiernos, e industria.

Librerías de MPI:

- MPICH (<https://www.mpich.org/>)
- OpenMPI (<https://www.open-mpi.org/>)
- Propietarias de HP, IBM, Intel, etc.

Portable: funciona en múltiples sistemas operativos y múltiples plataformas hardware.

Compatibilidad hacia atrás: NO

MPI: versiones y documentación

MPI 1.0 – 1994. Modelo de procesos estático. > 100 funciones

MPI 2.0 – 1997. Modelo de procesos dinámico. > 200 funciones

MPI 3.0 – 2012. Mejora escalabilidad, tolerancia a fallos, memoria compartida. > 400 funciones

Última versión:

MPI 3.1 – 2015

Sólo se suele utilizar un pequeño subconjunto de funciones

Documentación:

- Estándar de MPI 3.1.
868 páginas. Muy buen material de referencia.
<http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>
- Páginas del manual (comando man de linux).
- Otro material:
<https://computing.llnl.gov/tutorials/mpi/>
https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php

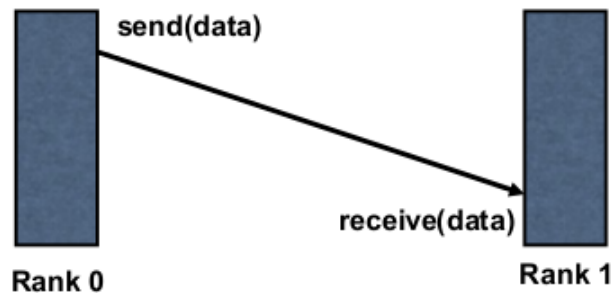
Modelo de programación y ejecución

Modelo de programación

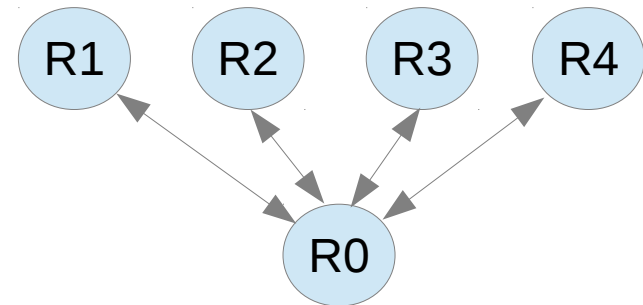
- Memoria compartida
 - Paralelismo de threads → OpenMP
- Memoria distribuida
 - Paralelismo de procesos → MPI
- Solución híbrida
 - Procesos + threads
MPI + OpenMP

Modelo de programación

- Se crean procesos, normalmente uno por core o uno por nodo si se usa junto a OpenMP. Los programas deberían escribirse para soportar un número arbitrario de procesos.
- Cada proceso tiene (al menos) un rank (identificador).
- Cada proceso ejecuta el mismo programa (SPMD). No es una restricción, y es equivalente a MIMD ya que cada proceso puede ejecutar una parte diferente del programa (basándose en el rank).
- Las comunicaciones se realizan mediante el envío y recepción explícita de mensajes. Las comunicaciones pueden ser:



Punto a Punto



Colectivas

Compilación y Ejecución

Compilación

```
mpicc prueba.c -o prueba
```

Ejecución

MPI recomienda que las librerías implementen el comando *mpiexec* para enviar un programa a ejecutar.

```
mpiexec -n 4 program_name program_arguments
```



Número de procesos

Sin embargo, muchas veces no se implementa y será necesario utilizar algún comando propio del gestor de trabajos (ej. SLURM) del sistema de cómputo.

Algunas librerías de MPI también definen el comando NO Estándar:

```
mpirun -np 4 program_name program_arguments
```

Ejecución: ¿Dónde ejecutan los procesos?

Si se utiliza un gestor de trabajos, consultar su documentación.

Si se utiliza mpiexec/mpirun, hay dos formas:

- **En un archivo:** se especifican los nodos y sus capacidades en un archivo de texto.

Ejemplo con MPI de Intel: Se crea un archivo hosts.txt con el siguiente contenido (formato nodo:cores):

```
nodo1:16
```

```
nodo2:16
```

Al ejecutar *mpirun* se pone el nombre del archivo como argumento de la opción *-machine*

```
mpirun -machine hosts.txt -np 20 ./prueba
```

- **Por línea de comandos**

```
mpirun -hosts nodo1:16,nodo2:16 -np 20 ./prueba
```

Primer programa

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

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

```
int main( int argc, char *argv[] ) {
```

```
    MPI_Init(&argc, &argv);
```

```
    printf("Hello world!\n");
```

```
    MPI_Finalize();
```

```
    return 0;
```

```
}
```

Antes de usar cualquier
función de MPI

Última operación MPI de
un programa MPI

```
Salida:  
    Hello world!  
    Hello world!
```


¿Quién es quién?

- Comunicadores

- Definen un dominio de comunicación que es un conjunto de procesos que intercambian mensajes
- `MPI_COMM_WORLD` contiene todos los procesos



- ¿ Cómo sabe cada proceso...

- cuántos procesos hay en total ?

`MPI_Comm_size`: retorna el número total de procesos en un dado comunicador

- cuál de ellos es ?

`MPI_Comm_rank`: retorna el índice del proceso en un cierto comunicador. El índice estará entre 0 y `size - 1`.

Ejemplo que identifica procesos

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

int main( int argc, char *argv[]) {
    int rank, size, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init( &argc, &argv );
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(processor_name, &namelen);
    printf("Hello from process %d out of %d on %s\n", rank,
size, processor_name);
    MPI_Finalize();
    return 0;
}
```

Salida:

```
I am 0 out of 4 on nodo1
I am 3 out of 4 on nodo2
I am 1 out of 4 on nodo2
I am 2 out of 4 on nodo1
```

Tipos de datos básicos

Tipos de datos básicos

Utilizados para transferencias de datos **homogéneos**, que se encuentran ubicados de forma **contigua** en la memoria.

Tipo en MPI	Tipo en C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	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_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wide char
MPI_PACKED	special data type for packing
MPI_BYTE	single byte value

Tipos de datos básicos

Para una comunicación, se especifican:

- dirección de memoria del **buffer**
- **tipo de dato**
- **cantidad de elementos**



¿Por qué MPI define tipos de datos básicos?

Permite soportar comunicaciones entre procesos que ejecutan en máquinas con distintas representaciones (y longitud) de datos en memoria.

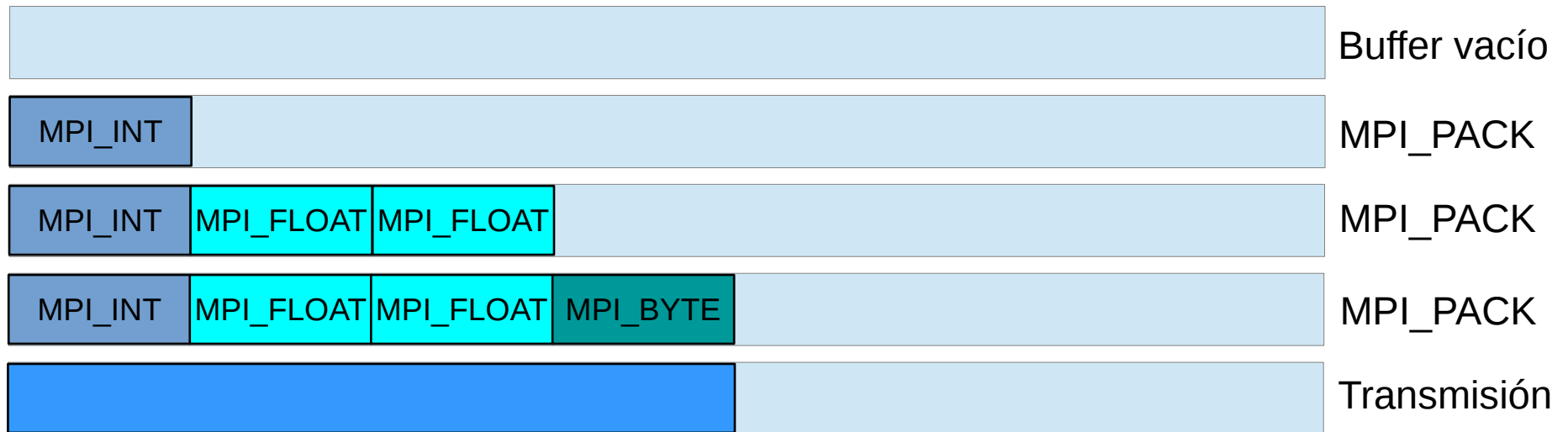
¿Y si los datos son heterogéneos o están dispersos?

Usamos el tipo de datos MPI_PACKED...

Pack - Unpack

Permite enviar/recibir datos heterogéneos y/o dispersos en memoria.

- MPI_PACK permite empaquetar datos (posiblemente heterogéneos) en un buffer contiguo para ser transmitido.
- MPI_UNPACK permite que el receptor de los datos los desempaquete en ubicaciones de memoria dispersas.



Pack - Unpack

```
int MPI_Pack(const void* inbuf,  
            int incount,  
            MPI_Datatype datatype,  
            void* outbuf,  
            int outsize,  
            int* position,  
            MPI_Comm comm)
```

```
int MPI_Unpack(const void* inbuf,  
              int insize,  
              int* position,  
              void* outbuf,  
              int outcount,  
              MPI_Datatype datatype,  
              MPI_Comm comm)
```

Ejemplo de MPI_Pack:

```
int position, i, j, a[2];  
char buff[1000];
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

```
if (myrank == 0)
```

```
{ /* SENDER CODE */
```

```
    position = 0;
```

```
    MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
```

```
    MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
```

```
    //Aquí va una operación de envío cuyo tipo de dato es MPI_PACKED
```

```
}
```

```
else
```

```
{ /* RECEIVER CODE */
```

```
    //Aquí va una operación de recepción con tipo de dato MPI_INT o MPI_PACKED
```

```
}
```

El programador puede crear sus propios tipos de datos, que se denominan Tipos de Datos Derivados (se explica más adelante)

Comunicaciones punto a punto

Esquema de comunicación

En esta comunicación participan dos procesos:

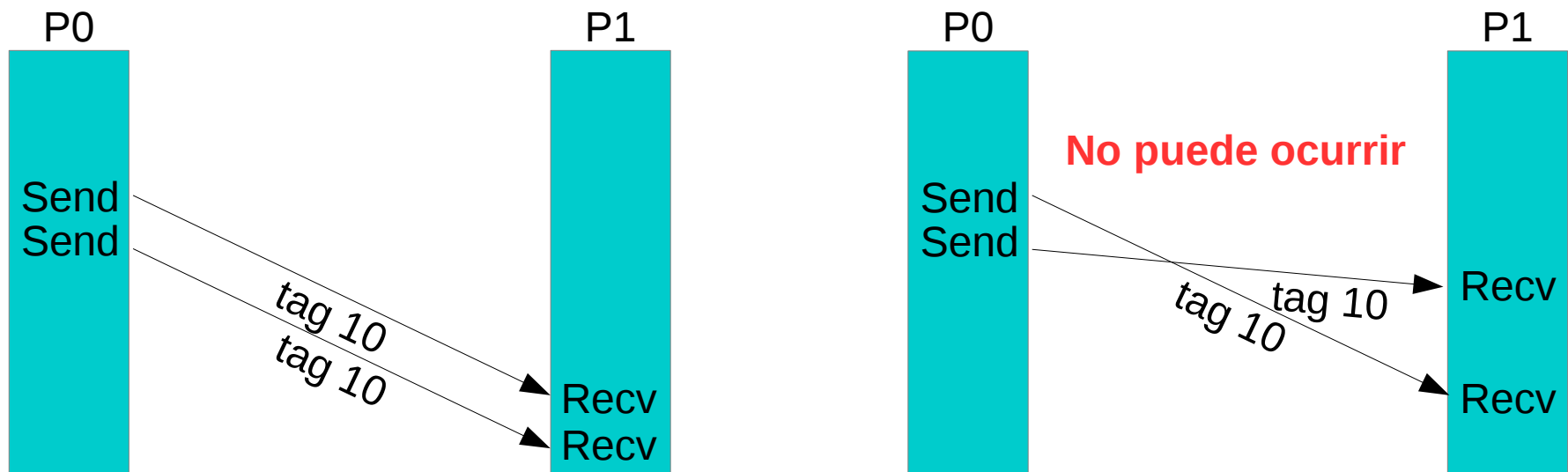
- Un proceso emisor que ejecuta una operación de envío (send)
- Un proceso receptor que ejecuta una operación de recepción (receive)

Los mensajes tienen una etiqueta (tag) de identificación.

Orden de los mensajes

Único caso garantizado:

Si un emisor envía varios mensajes (uno después de otro) al mismo receptor, y los mensajes concuerdan con una misma operación de recepción, MPI asegura que los mensajes se recibirán en el orden en que se enviaron.



Operaciones de comunicaciones bloqueantes y no bloqueantes

Los términos **bloqueante** y **no bloqueante** describen el comportamiento de las operaciones de comunicación desde el **punto de vista local** al proceso que las ejecuta.

- **Operación bloqueante:**

El retorno de la operación indica que todos los recursos (buffers) especificados en la llamada pueden reutilizarse.

- **Operación No-bloqueante:**

El retorno podría ocurrir antes de que la operación finalice, y por lo tanto los recursos no pueden reutilizarse hasta que se haya comprobado que la operación ha finalizado.

Modos de comunicaciones

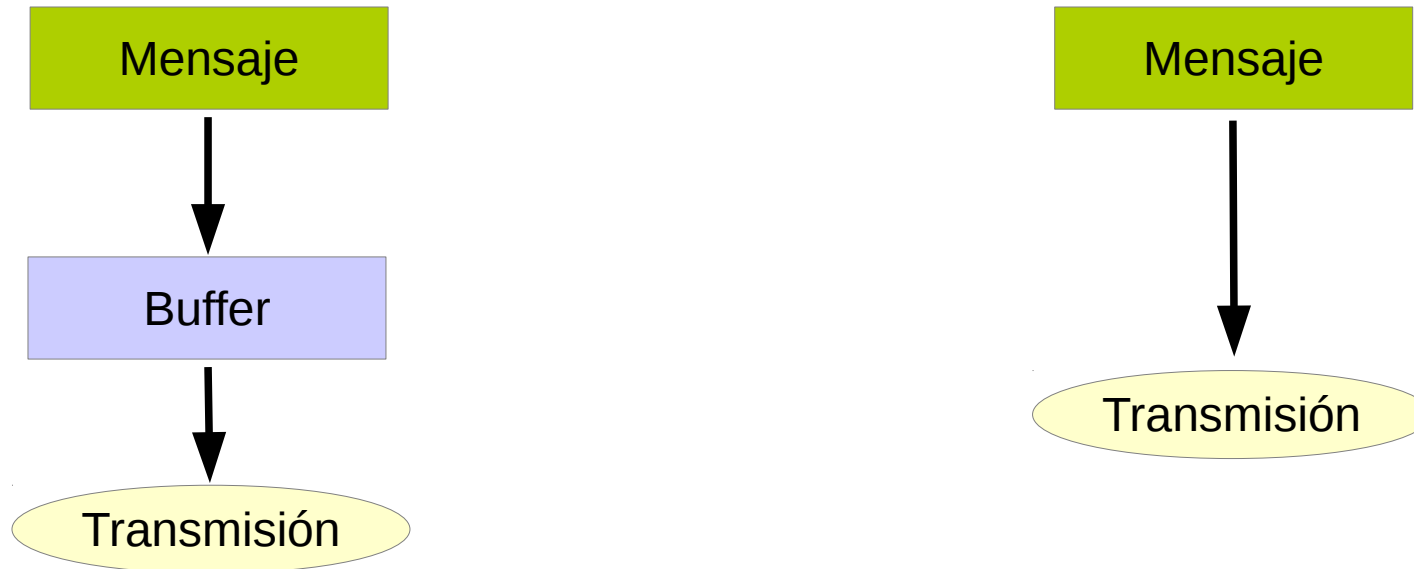
Se definen, para operaciones bloqueantes y no bloqueantes, los siguientes modos de comunicaciones:

- Standard Mode
- Buffered Mode
- Synchronous Mode
- Ready Mode

Modos de comunicaciones

Standard Mode

La implementación es libre de decidir si los mensajes, antes de ser transmitidos, se almacenan o no en un buffer del sistema local.



La elección de crear o no un buffer podría estar basada en cuestiones de rendimiento y de espacio disponible en memoria.

Modos de comunicaciones

Buffered/Synchronous/Ready

Buffered Mode

La diferencia con el Standard Mode es que la finalización de un envío en ningún caso dependerá de la ocurrencia de una recepción.

Si se hace un envío y no hay una recepción esperando, entonces la implementación está obligada a utilizar un buffer. La operación de envío podrá finalizar al terminar de colocar el mensaje en el buffer.

Synchronous Mode

El envío finalizará cuando el receptor haya comenzado a recibir el mensaje.

Ready Mode

El usuario garantiza que en el momento del envío ya hay una operación de recepción esperando (si no se cumple, el comportamiento es indefinido). Esto permite mejorar el rendimiento.

Comunicaciones
punto a punto
bloqueantes

MPI_Send

MPI_Send realiza un envío standard-mode bloqueante

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

- El buffer de envío se define con los parámetros *buf*, *count* y *datatype*.
buf es un buffer que contiene los elementos a enviar
count es la cantidad de elementos a enviar
datatype es el tipo de dato de los elementos
- *dest* indica el rank del proceso, dentro del comunicador *comm*, a quien va dirigido el mensaje
- *tag* es una etiqueta del mensaje que puede ser utilizada por el receptor para distinguir diferentes mensajes del mismo emisor.

MPI_Recv

MPI_Recv realiza una recepción bloqueante (retorna después de que el buffer contiene el nuevo mensaje recibido)

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

Entrada:

- El buffer de recepción se define con los parámetros *buf*, *count* y *datatype*.
buf es el buffer que contendrá los elementos a recibir
count es la cantidad máxima de elementos a recibir
datatype es el tipo de dato de los elementos
- *source* indica el rank del proceso, dentro del comunicador *comm*, de quien proviene el mensaje. MPI_ANY_SOURCE: acepta cualquier emisor
- *tag* es una etiqueta del mensaje que puede ser utilizada por el receptor para distinguir diferentes mensajes del mismo emisor. MPI_ANY_TAG: acepta cualquier tag

Salida:

- *status* se retorna información sobre la operación como el estado de la recepción y la cantidad de elementos recibidos.

Status de MPI_Recv

Después de completar la operación MPI_Recv(), el *estatus* contiene la siguiente información:

<code>status.MPI_SOURCE</code>	especifica el rank del emisor
<code>status.MPI_TAG</code>	especifica el tag del mensaje recibido
<code>status.MPI_ERROR</code>	contiene un código de error

También contiene información sobre la longitud del mensaje:

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

Entrada:

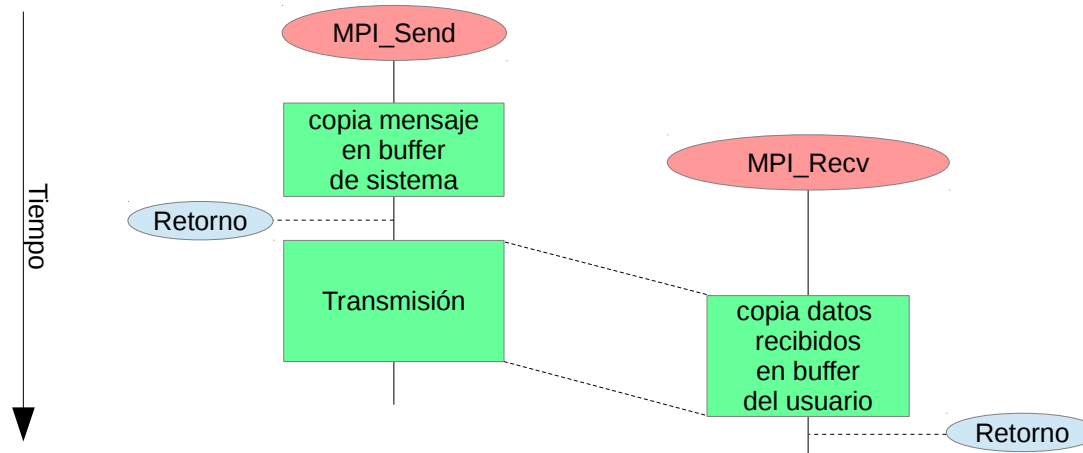
status estado retornado por la operación de recepción
datatype datatype de los elementos recibidos en el buffer

Salida:

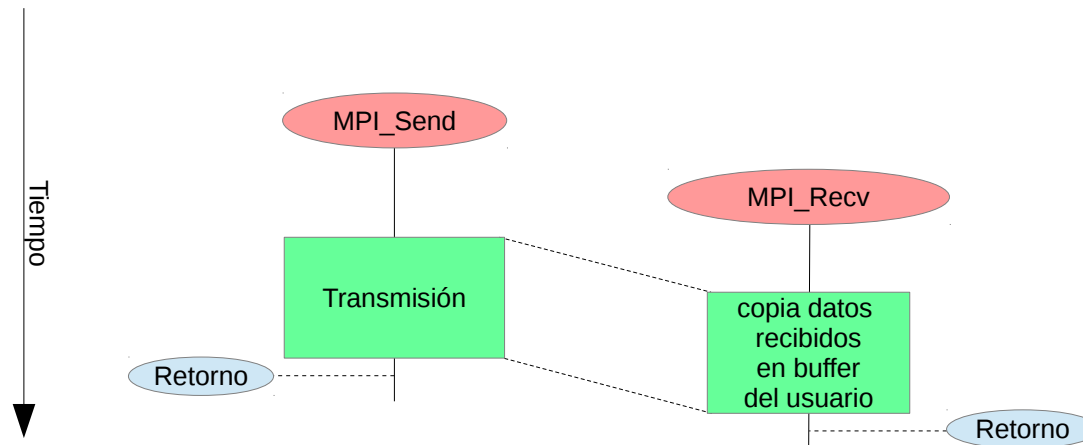
count cantidad de elementos recibidos

Punto de retorno de operaciones MPI_Send y MPI_Recv

Con buffer de sistema:



Sin buffer de sistema:



Ejemplo de MPI_Send y MPI_Recv

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

int main(int argc, char *argv[]) {
    int rank;
    char msg[20];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        strcpy(msg, "hola");
        MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    }
    else if (rank == 1) {
        MPI_Recv(msg, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status);
        printf("Recibido: %s\n", msg);
    }

    MPI_Finalize();
    return 0;
}
```

Ejemplo de MPI_Send y MPI_Recv

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

int main(int argc, char *argv[]) {
    int rank;
    wchar_t msg[20];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    setlocale(LC_ALL, "es_AR.UTF-8");

    if (rank == 0) {
        wcscpy(msg, L"ñandú");
        MPI_Send(msg, wcslen(msg)+1, MPI_WCHAR, 1, 0, MPI_COMM_WORLD);
        printf("Enviado: %ls\n", msg);
    }
    else if (rank == 1) {
        MPI_Recv(msg, 20, MPI_WCHAR, 0, 0, MPI_COMM_WORLD, &status);
        printf("Recibido: %ls\n", msg);
    }

    MPI_Finalize();
    return 0;
}
```


Deadlocks

El siguiente programa siempre causa un deadlock:

```
if (rank == 0) {  
    MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);  
    MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);  
}  
else if (rank == 1) {  
    MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);  
    MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);  
}
```

Deadlocks

En el siguiente programa la ocurrencia de deadlock depende de la política definida por la implementación de MPI que se utilice:

```
if (rank == 0) {  
    MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);  
    MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);  
}  
else if (rank == 1) {  
    MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);  
    MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);  
}
```

Un programa debería ser escrito de tal forma que, para cualquier implementación de MPI, siempre funcione correctamente.

Deadlocks

El siguiente programa nunca entra en deadlock. Es un programa “seguro” que no depende de la implementación.

```
if (rank == 0) {  
    MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);  
    MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);  
}  
else if (rank == 1) {  
    MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);  
    MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);  
}
```

MPI_Sendrecv

MPI_Sendrecv hace un envío y recepción bloqueantes

Permite no preocuparnos por el orden de los Send y Recv

```
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)
```

MPI_Sendrecv_replace

MPI_Sendrecv_replace es como Sendrecv pero usa el mismo buffer para envío y recepción

```
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)
```

Modos

Buffered/Synchronous/Ready

`MPI_Send (buf, count, datatype, dest, tag, comm)` **Standard**

`MPI_Bsend(buf, count, datatype, dest, tag, comm)` **Buffered**

`MPI_Buffer_attach`
`MPI_Buffer_detach` } Funciones complementarias para MPI_Bsend

`MPI_Ssend(buf, count, datatype, dest, tag, comm)` **Synchronous**

`MPI_Rsend(buf, count, datatype, dest, tag, comm)` **Ready**

Comunicaciones
punto a punto
no bloqueantes

MPI_Isend y MPI_Irecv

- Operaciones que inician comunicaciones no bloqueantes
- Aumenta el rendimiento mediante el solapamiento de comunicaciones con cómputo (en caso de ser posible).

MPI_Isend

MPI_Isend realiza un envío standard-mode no bloqueante

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

Entrada:

- El buffer de envío se define con los parámetros *buf*, *count* y *datatype*.
buf es un buffer que contiene los elementos a enviar
count es la cantidad de elementos a enviar
- *datatype* es el tipo de dato de los elementos
- *dest* indica el rank del proceso, dentro del comunicador *comm*, a quien va dirigido el mensaje
- *tag* es una etiqueta del mensaje que puede ser utilizada por el receptor para distinguir diferentes mensajes del mismo emisor.

Salida:

- *request* retorna un objeto identificador de la comunicación que será necesario suministrar a la operación de fin de la comunicación.

MPI_Irecv

MPI_Irecv realiza una recepción no bloqueante

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

Entrada:

- El buffer de recepción se define con los parámetros *buf*, *count* y *datatype*.
buf es el buffer que contendrá los elementos a recibir
count es la cantidad máxima de elementos a recibir
datatype es el tipo de dato de los elementos
- *source* indica el rank del proceso, dentro del comunicador *comm*, de quien proviene el mensaje. MPI_ANY_SOURCE: acepta cualquier emisor
- *tag* es una etiqueta del mensaje que puede ser utilizada por el receptor para distinguir diferentes mensajes del mismo emisor. MPI_ANY_TAG: acepta cualquier tag

Salida:

- *request* retorna un objeto identificador de la comunicación que será necesario suministrar a la operación de fin de la comunicación.

MPI_Wait, MPI_Test y derivados

El programa no debe modificar/leer el buffer hasta **comprobar** que la **operación** de envío/recepción ha sido **completada**.

La comprobación se puede hacer de diversas formas:

- Con espera:
 - por una comunicación: `MPI_Wait`
 - por cualquier comunicación: `MPI_Waitany`
 - por algunas comunicaciones: `MPI_Waitsome`
 - por todas las comunicaciones: `MPI_Waitall`
- Sin espera:
 - por una comunicación: `MPI_Test`
 - por cualquier comunicación: `MPI_Testany`
 - por algunas comunicaciones: `MPI_Testsome`
 - por todas las comunicaciones: `MPI_Testall`

MPI_Wait

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

↑
entrada

↓
salida

Ejemplo:

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

```
int main(int argc, char *argv[]) {
    int rank;
    char msg[20];
    MPI_Request request;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        strcpy(msg, "hola");
        MPI_Isend(msg, strlen(msg)+1, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &request);
        // Aquí va el cómputo a solapar con la comunicación...
        MPI_Wait(&request, &status);
    }
    else if (rank == 1) {
        MPI_Irecv(msg, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &request);
        // Aquí va el cómputo a solapar con la comunicación...
        MPI_Wait(&request, &status);
        printf("mensaje: %s\n", msg);
    }

    MPI_Finalize();
    return 0;
}
```

MPI_Waitall

```
int MPI_Waitall(int count,                ← entrada
                MPI_Request array_of_requests[], ← entrada
                MPI_Status array_of_statuses[]) → salida
```

Ejemplo:

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

main(int argc, char *argv[]) {
    int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
    MPI_Request reqs[4];
    MPI_Status stats[4];

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

    prev = rank-1;
    next = rank+1;
    if (rank == 0) prev = numtasks - 1;
    if (rank == (numtasks - 1)) next = 0;

    MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

    MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

    // se hace algún trabajo mientras se realizan las comunicaciones

    MPI_Waitall(4, reqs, stats);
    MPI_Finalize();
}
```

MPI_Waitany y MPI_Waitsome

```
int MPI_Waitany(int count,
                MPI_Request array_of_requests[],
                int *indx,
                MPI_Status *status)
```

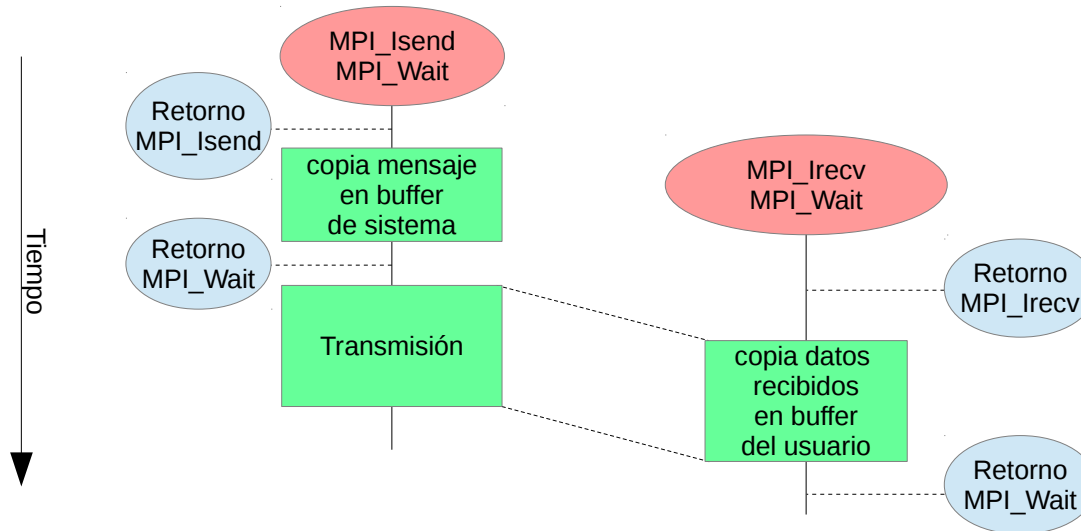
`count` es la longitud del `array_of_requests`
`array_of_requests` es un arreglo de requests
`indx` indica el índice de la operación completada
`status` es el objeto estado de esa comunicación

```
int MPI_Waitsome(int incount,
                  MPI_Request array_of_requests[],
                  int *outcount,
                  int array_of_indices[],
                  MPI_Status array_of_statuses[])
```

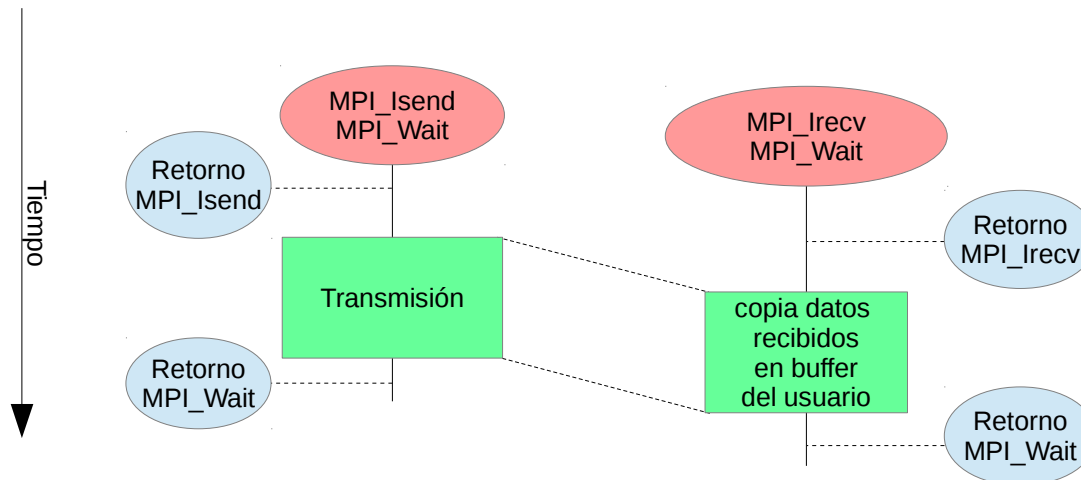
`incount` es la longitud del `array_of_requests`
`array_of_requests` es un arreglo de requests
`outcount` indica la cantidad operaciones completadas
`array_of_indices` indica los índices de las operaciones completadas
`array_of_statuses` contiene los objetos de estado de las operaciones completadas

Punto de retorno de operaciones MPI_Isend y MPI_Irecv

Con buffer de sistema:



Sin buffer de sistema:



Modos

Buffered/Synchronous/Ready

`MPI_Isend (buf, count, datatype, dest, tag, comm, request)` **Standard**

`MPI_Ibsend(buf, count, datatype, dest, tag, comm, request)` **Buffered**

`MPI_Buffer_attach`
`MPI_Buffer_detach` } Funciones complementarias para MPI_Ibsend

`MPI_Issend(buf, count, datatype, dest, tag, comm, request)` **Synchronous**

`MPI_Irsend(buf, count, datatype, dest, tag, comm, request)` **Ready**

Comunicaciones Colectivas

Introducción

- Las **Comunicaciones Colectivas** involucran comunicaciones y cálculos coordinados entre los procesos de un comunicador
- Ventajas:

Al reemplazar múltiples llamadas de envío y recepción, y algunos cálculos adicionales, se logra:

 - Código más legible
 - Posiblemente código más optimizado
- Clases de colectivas:
 - **Sincronización**: barrier
 - **Movimiento de datos**: broadcast, scatter, gather
 - **Cálculo colectivo**: reductions. Un miembro del grupo recolecta datos de los otros y realiza una operación sobre los datos (mínimo, máximo, suma, etc.)

Generalidades de las operaciones

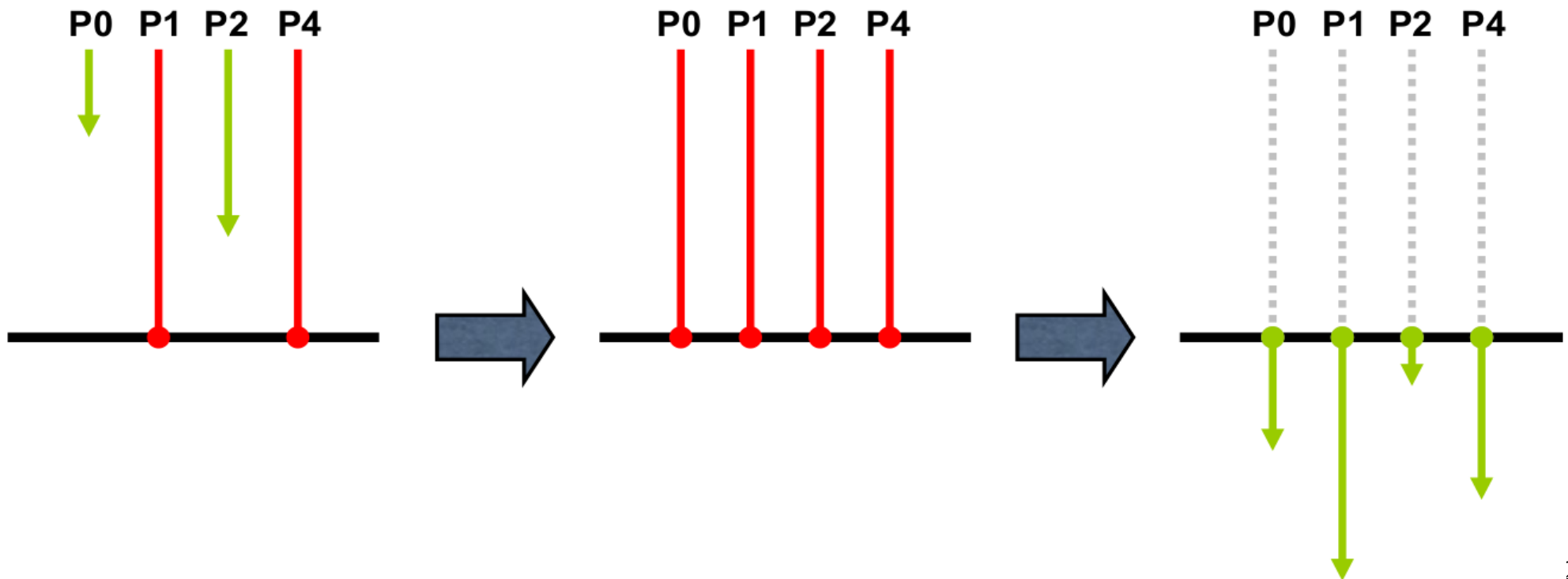
- Todos los procesos en un comunicador deben participar en todas las operaciones colectivas. Si no ocurre, el comportamiento del programa queda indefinido (el programa podría fallar)
- Hay colectivas bloqueantes y no bloqueantes (MPI 3)
- No se utilizan tags de mensajes

Sincronización: MPI_Barrier

Operación de sincronización del tipo barrier

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

Cuando un proceso alcanza esta función, se bloquea hasta que todos los procesos en el comunicador hayan llamado a la función. En ese instante todos los procesos se desbloquean.

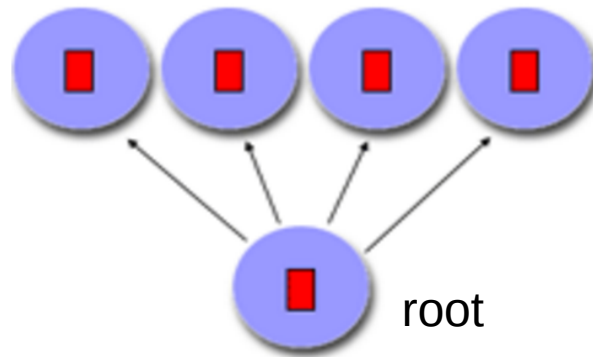


Movimiento de datos: MPI_Bcast

Operación bloqueante (de broadcast), que envía un mensaje de un proceso a los demás asociados al comunicador

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

`root` es el rank del proceso que tiene los datos a enviar



Ejemplo: El proceso 0 hace un broadcast a los procesos restantes. Todos los procesos ejecutan el siguiente código.

```
int buf[size];  
int MPI_Bcast(buf, size, MPI_INT, 0, MPI_COMM_WORLD);
```

Movimiento de datos: MPI_Scatter

Operación bloqueante que reparte datos de un proceso a todos los procesos (inclusive al mismo que tenía los datos a distribuir) asociados a un comunicador. Reparte segmentos de datos, de un buffer, de forma equitativa y ordenada. El orden es por número de rank, es decir, primer segmento al rank 0, segundo al rank 1, etc.

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

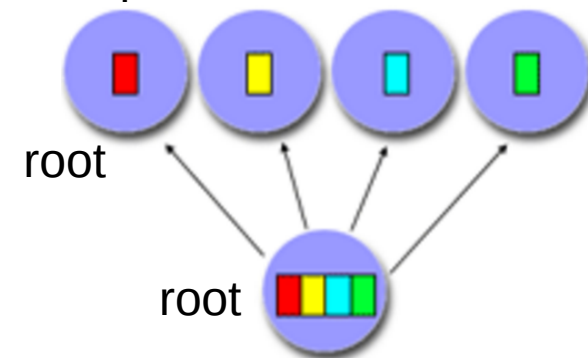
`root` es el rank del proceso que tiene los datos a repartir

`sendbuf` es el buffer con los datos a repartir y solo tienen significado en el proceso `root`

`sendcount` es la cantidad de elementos a enviar a cada proceso

`recvbuf` es el buffer donde se recibirán los datos

`recvcount` es la cantidad de elementos del buffer de recepción



Ejemplo: El proceso 0 hace un scatter. Todos los procesos ejecutan el siguiente código.

```
MPI_Comm comm;  
int size, *sendbuf;  
int rbuf[100];  
MPI_Comm_size(comm, &size);  
if (rank == 0) sendbuf = (int*)malloc(100 * size * sizeof(int));  
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, 0, comm);
```

Movimiento de datos: MPI_Alltoall

Operación bloqueante que reparte datos de todos los procesos a todos los procesos asociados a un comunicador. Los segmentos se reparten de forma equitativa y ordenada por número de rank.

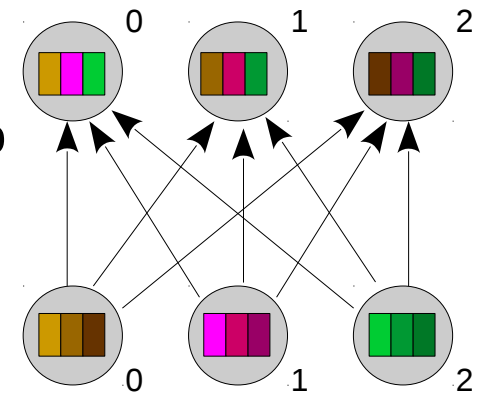
```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,  
                void *recvbuf, int recvcount, MPI_Datatype recvtype,  
                MPI_Comm comm)
```

sendbuf es el buffer con los datos a repartir

sendcount es la cantidad de elementos a enviar a cada proceso

recvbuf es el buffer donde se recibirán los datos

recvcount es la cantidad de elementos a recibir de cada proceso



Ejemplo: Todos los procesos ejecutan el siguiente código.

```
MPI_Comm comm;  
int size,*sendbuf;  
int rbuf[100];  
MPI_Comm_size(comm, &size);  
sendbuf = (int*)malloc(100 * size * sizeof(int));  
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, 0, comm);
```

Movimiento de datos: MPI_Gather

Operación bloqueante que recolecta, en un proceso, datos de todos los procesos (inclusive los que tiene él mismo) asociados a un comunicador. La recolección de segmentos de datos es ordenada por número de rank. Es la inversa de MPI_Scatter.

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

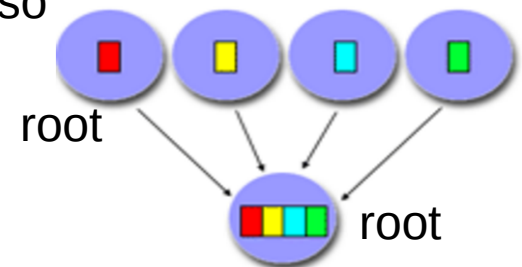
root es el rank del proceso que recibirá los datos

sendbuf es el buffer con los datos a enviar

sendcount es la cantidad de elementos a enviar

recvbuf es el buffer donde se recibirán los datos, solo con significado en root

recvcount es la cantidad de elementos a recibir de cada proceso



Ejemplo: El proceso 0 hace un gather. Todos los procesos ejecutan el siguiente código.

```
MPI_Comm comm;  
int size,*sendbuf;  
int sbuf[100];  
MPI_Comm_size(comm, &size);  
if (rank == 0) recvbuf = (int*)malloc(100 * size * sizeof(int));  
MPI_Gather(sbuf, 100, MPI_INT, recvbuf, 100, MPI_INT, 0, comm);
```


Movimiento de datos: MPI_Allgather

Operación de movimiento de datos del tipo Allgather bloqueante. Concatena los datos de los procesos y se los coloca en cada proceso asociado a un comunicador. Es decir, cada proceso realiza una operación de broadcast de sus datos.

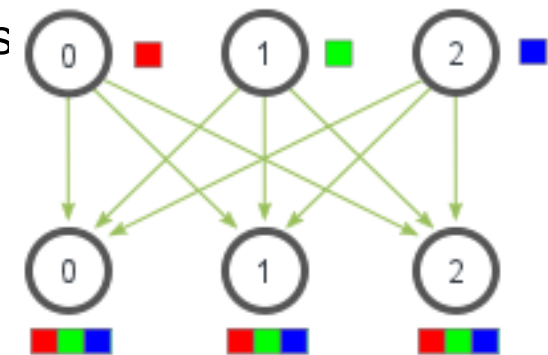
```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype,  
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,  
                 MPI_Comm comm)
```

`sendbuf` es el buffer con los datos a enviar

`sendcount` es la cantidad de elementos a enviar

`recvbuf` es el buffer donde se recibirán los datos

`recvcount` es la cantidad de elementos a recibir de cada proces



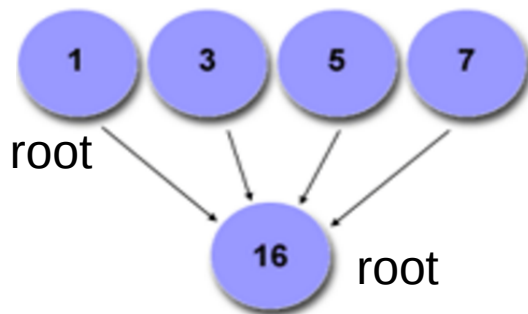
Ejemplo: Todos los procesos ejecutan el siguiente código.

```
MPI_Comm comm;  
int size, *sendbuf;  
int sbuf[100];  
MPI_Comm_size(comm, &size);  
recvbuf = (int*)malloc(100 * size * sizeof(int));  
MPI_Allgather(sbuf, 100, MPI_INT, recvbuf, 100, MPI_INT, comm);
```

Cálculo: MPI_Reduce

Operación de cálculo del tipo reducción todos a uno, bloqueante. Combina los elementos enviados por cada uno de los procesos (inclusive el destino), en un cierto comunicador, aplicando una cierta operación.

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

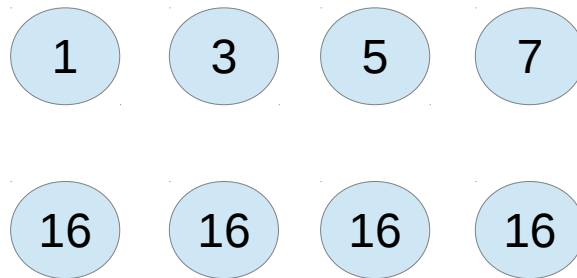


Nombre	Significado
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or (xor)
MPI_BXOR	bit-wise exclusive or (xor)
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

Cálculo: MPI_Allreduce

Operación de cálculo del tipo reducción todos a todos, bloqueante. Pone el resultado de una operación de reducción en todos los procesos asociados a un cierto comunicador.

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



Más sobre colectivas

- MPI proporciona más funciones colectivas
- Muchas funciones son derivadas de otras existentes:
 - Permiten que las partes a enviar sean de distinto tamaño
MPI_Scatterv, MPI_Gatherv, etc.
- Se pueden definir funciones de reducción personalizadas
 - MPI_Op_create, MPI_Op_free

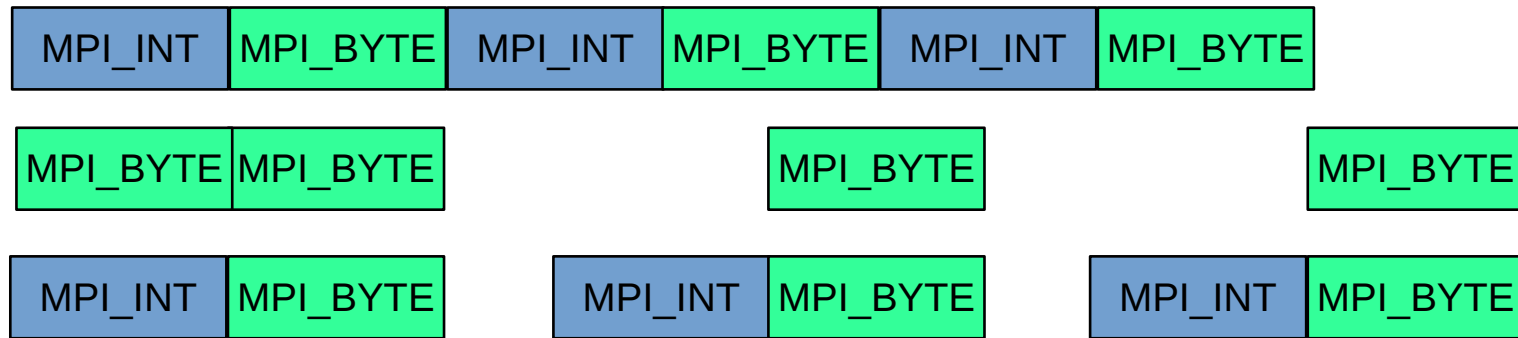
Tipos de datos derivados

¿Por qué MPI define tipos de datos derivados?

Conociendo la disposición de los datos en memoria, se reduce la copia de datos de memoria a memoria, permitiendo utilizar hardware especialmente diseñado para este propósito.

Tipos de datos derivados

- Utilizados para transferencias de datos heterogéneos y/o dispersos en memoria.



- La implementación de MPI podría utilizar algún mecanismo del subsistema de comunicación que permita envíos de datos dispersos o, en caso de no soportarse, los podría empaquetar.

Tipos de datos derivados

Para crear un nuevo tipo de dato derivado se requieren los siguientes pasos:

- **Declarar** del nuevo nombre del tipo de dato:

```
MPI_Datatype mi_tipo;
```

- **Construir** del tipo de dato derivado. Se pueden construir tipos de datos derivados a partir de tipos de datos básicos o derivados. Algunos constructores son: Contiguous, Vector, Indexed, Struct.
- **Agregar** el nuevo tipo de dato al sistema:

```
MPI_Type_commit(&mi_tipo)
```

Es posible **liberar** un tipo de dato derivado:

```
MPI_Type_free (&mi_tipo)
```

Se asigna el valor `MPI_DATATYPE_NULL` a `mi_tipo`. Las comunicaciones actuales que utilizan el tipo se completarán normalmente. La liberación de un tipo de dato no afecta a otro tipo de dato que fue construido a partir de él.

Tipo de dato Contiguous

Define un arreglo de elementos contiguos en memoria.

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

Tipo de dato Contiguous

Ejemplo:



`count = 8`

```
MPI_Datatype mi_tipo;  
MPI_Type_contiguous(8, MPI_INT, &mi_tipo);  
MPI_Type_commit(&mi_tipo);
```

Tipo de dato Vector

Define un arreglo de bloques (de elementos contiguos), uniformemente distanciados unos de otros.

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

`count` es el número de bloques

`blocklength` es el número de elementos de los bloques

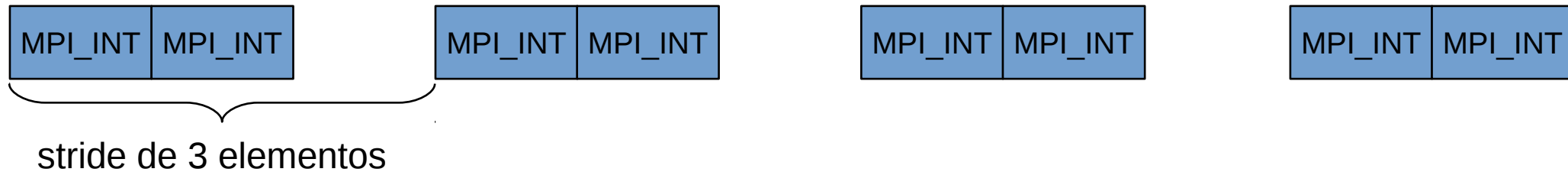
`stride` es la distancia en número de elementos entre los inicios de los bloques

`old_type` es el tipo de los elementos que conforman los bloques

`MPI_Type_create_hvector` es una variante con el stride medido en bytes.

Tipo de dato Vector

Ejemplo:



`count = 4`

`blocklength = 2`

`stride = 3`

```
MPI_Datatype mi_tipo;  
MPI_Type_vector(4, 2, 3, MPI_INT, &mi_tipo);  
MPI_Type_commit(&mi_tipo);
```

Tipo de dato Indexed

Define un arreglo de bloques (de elementos) y desplazamientos (entre bloques). Es más general que Vector, ya que define tamaños y desplazamientos particulares a cada uno de los bloques.

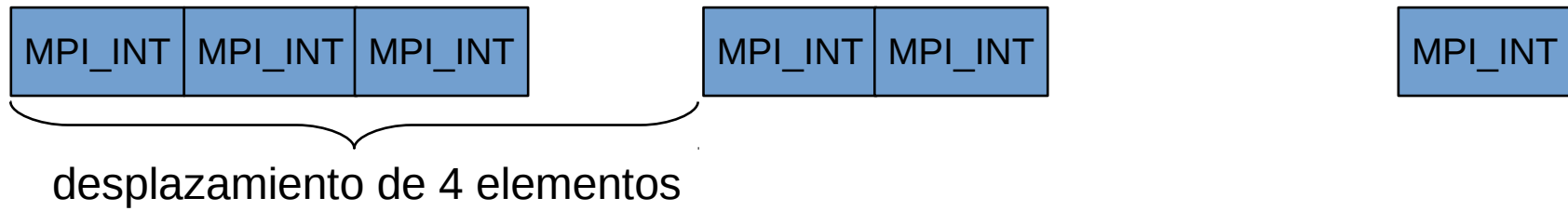
```
int MPI_Type_indexed(int count,  
                    const int *array_of_blocklengths,  
                    const int *array_of_displacements,  
                    MPI_Datatype oldtype,  
                    MPI_Datatype *newtype)
```

`count` es el número de bloques (y tamaño de los arreglos `array_of...`),
`blocklengths` define la longitud de cada bloque en número de elementos, y
`displacements` define el desplazamiento de cada bloque en número de elementos.

`MPI_Type_create_hindexed` es una variante con desplazamientos medidos en bytes.

Tipo de dato Indexed

Ejemplo:



`count = 3`

`array_of_blocklengths = { 3, 2, 1 }`

`array_of_displacements = { 0, 4, 8 }`

```
MPI_Datatype mi_tipo;
```

```
MPI_Type_indexed(3,
```

```
    array_of_blocklengths,
```

```
    array_of_displacements,
```

```
    MPI_INT,
```

```
    &mi_tipo);
```

```
MPI_Type_commit(&mi_tipo);
```

Tipo de dato Struct

Define un arreglo de bloques (de elementos) y desplazamientos (entre bloques). Es más general que Indexed, ya que cada bloque puede estar conformado por elementos de diferente tipo de dato.

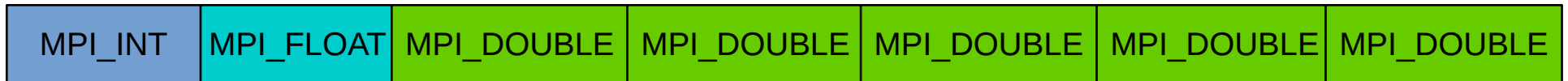
```
int MPI_Type_create_struct(int count,
                           const int array_of_blocklengths[],
                           const MPI_Aint array_of_displacements[],
                           const MPI_Datatype array_of_types[],
                           MPI_Datatype *newtype)
```

`count` es el número de bloques (y tamaño de los arreglos `array_of...`),
`blocklengths` define el número de elementos de cada bloque,
`displacements` define el desplazamiento de cada bloque en bytes,
`array_of_types` define el tipo de dato de cada bloque.

Tipo de dato Struct

Ejemplo:

```
struct mi_reg {  
    int a;  
    float b;  
    double c[4];  
}
```

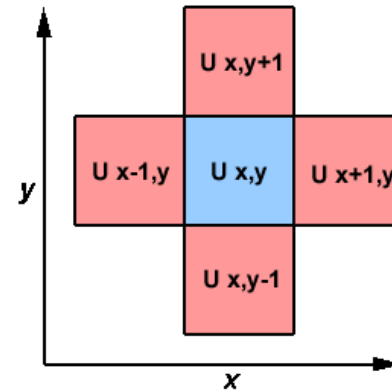
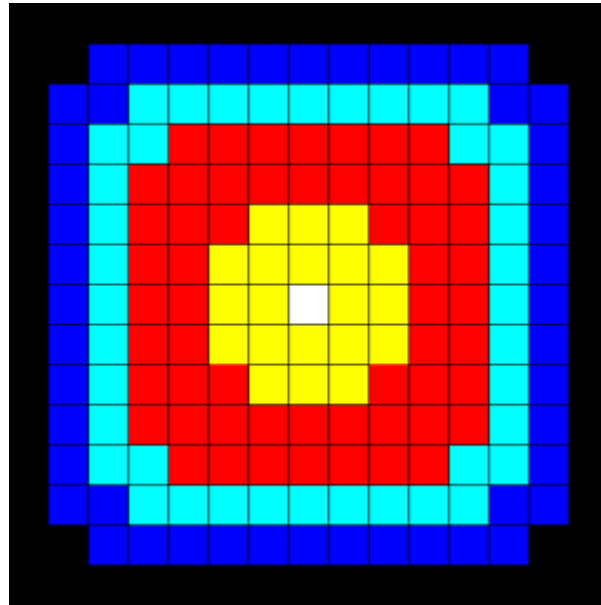


```
count = 3  
array_of_blocklengths = { 1, 1, 4 }  
array_of_displacements = { 0, 4, 8 }  
array_of_types = { MPI_INT, MPI_FLOAT, MPI_DOUBLE }
```

```
MPI_Datatype mi_tipo;  
MPI_Type_create_struct(3,  
    array_of_blocklengths,  
    array_of_displacements,  
    array_of_types,  
    &mi_tipo);  
MPI_Type_commit(&mi_tipo);
```

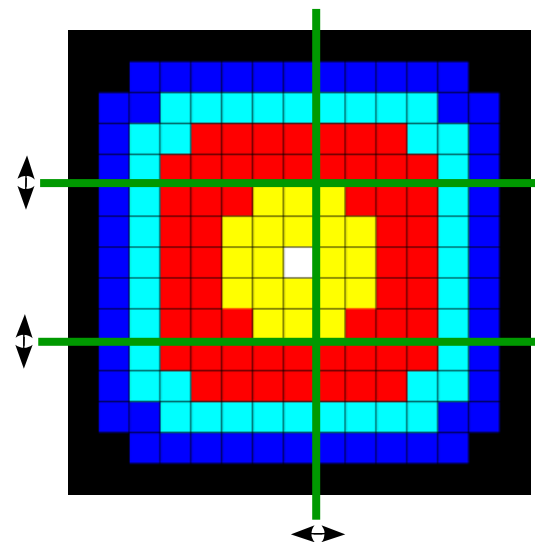
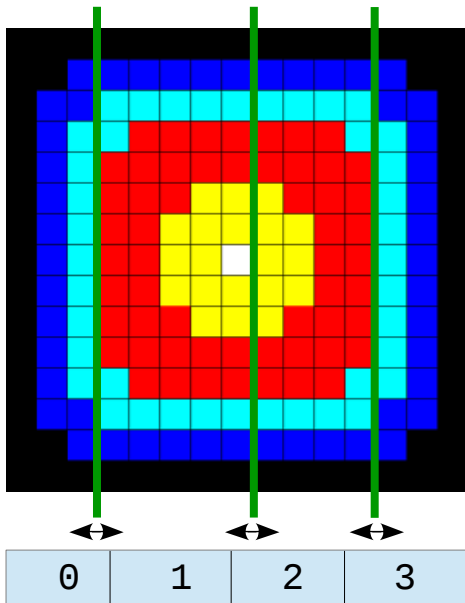
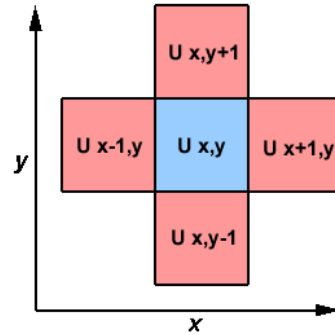
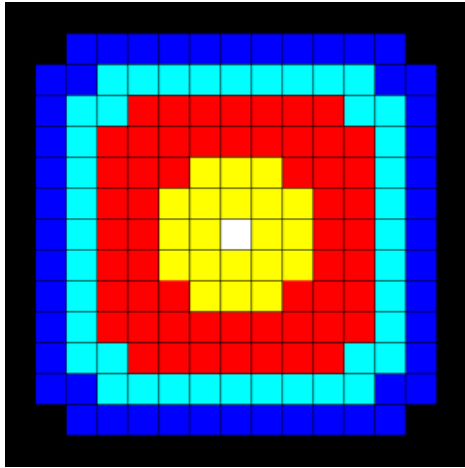

Topologías de procesos

Topología Cartesiana



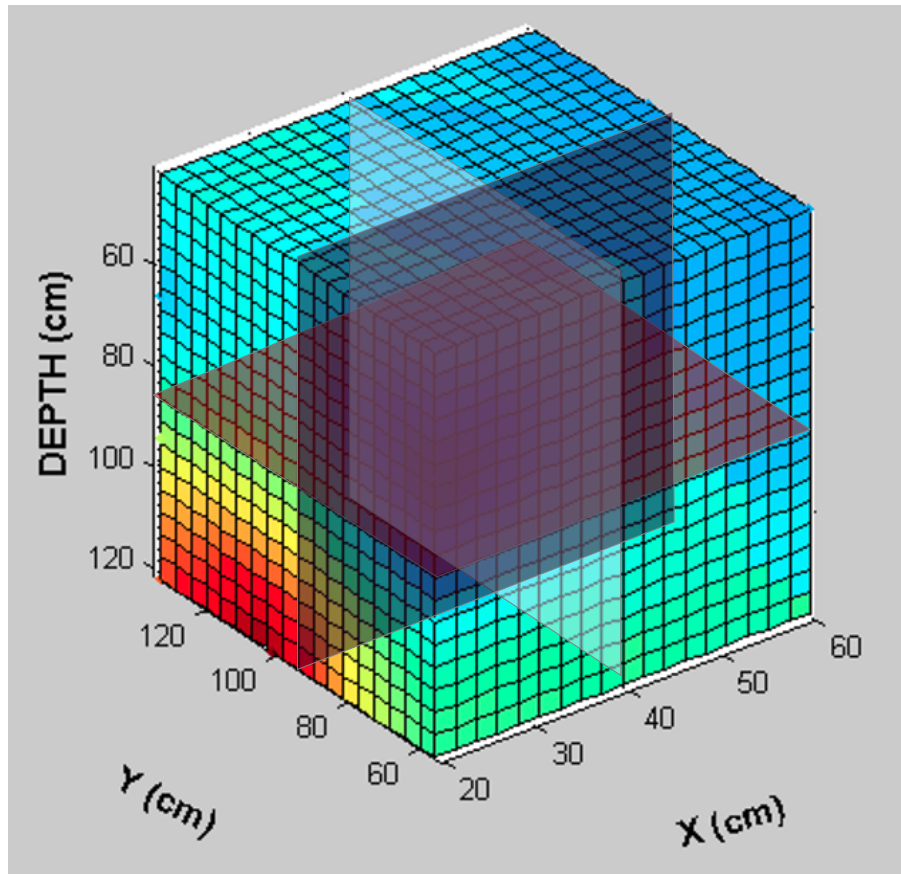
$$\begin{aligned} U_{x,y} &= U_{x,y} \\ &+ C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{xy}) \\ &+ C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y}) \end{aligned}$$

Topología Cartesiana



0 (0, 0)	1 (0, 1)
2 (1, 0)	3 (1, 1)
4 (2, 0)	5 (2, 1)

Topología Cartesiana



0 (0, 0, 0)	1 (0, 1, 0)
2 (1, 0, 0)	3 (1, 1, 0)
4 (0, 0, 1)	5 (0, 1, 1)
6 (1, 0, 1)	7 (1, 1, 1)


Constructor Cartesiano

```
int MPI_Cart_create(MPI_Comm comm_old,  
                   int ndims,  
                   const int dims[],  
                   const int periods[],  
                   int reorder,  
                   MPI_Comm *comm_cart)
```

`comm_old` es el comunicador inicial.

`ndims` número de dimensiones de grilla cartesiana.

`dims` es un arreglo de tamaño `ndims` que contiene el número de procesos de cada dimensión.

`periods` es un arreglo de tamaño `ndims` que indica si cada dimensión es circular  (true) o no (false).

`reorder` indica si se permite que la librería de MPI reasigne (true) o no (false) los ranks de los procesos para mejorar las comunicaciones.

`comm_cart` es el comunicador con la nueva topología cartesiana.

Ejemplo:

```
ndims = 2  
dims = {3, 4}  
periods = {1, 1}
```

//vecinos de 3: arriba 11, abajo 7, izquierda 2, derecha 0

0	1	2	3
(0, 0)	(0, 1)	(0, 2)	(0, 3)
4	5	6	7
(1, 0)	(1, 1)	(1, 2)	(1, 3)
8	9	10	11
(2, 0)	(2, 1)	(2, 2)	(2, 3)

MPI_Cart_shift

Provee los ranks de dos procesos vecinos equidistantes a un determinado número de saltos en una dirección.

```
int MPI_Cart_shift(MPI_Comm comm,
                  int direction,
                  int disp,
                  int *rank_source,    —► salida
                  int *rank_dest)     —► salida
```

direction indica la dimensión (de 0 a ndim-1) en la cual se encuentran los dos vecinos que están a una distancia de **disp** saltos. Se retorna el rank del vecino anterior en **rank_source** y posterior en **rank_dest**. Si el vecino está fuera de rango (solo para dimensiones no circulares), el rank contendrá el valor MPI_PROC_NULL.

Ejemplo:

```
int arriba, abajo, derecha, izquierda;
MPI_Cart_shift(cartcomm, 0, 1, &arriba, &abajo);
MPI_Cart_shift(cartcomm, 1, 1, &izquierda, &derecha);
```

Estando en rank 6:

por dim 0, a un salto (-1 y +1), se llega a 2 y 10.

por dim 1, a un salto (-1 y +1), se llega a 5 y 6.

0	1	2	3
(0,0)	(0,1)	(0,2)	(0,3)
4	5	6	7
(1,0)	(1,1)	(1,2)	(1,3)
8	9	10	11
(2,0)	(2,1)	(2,2)	(2,3)

MPI_Cart_coords

Determina las coordenadas cartesianas de un proceso a partir de su rank.

```
int MPI_Cart_coords(MPI_Comm comm,  
                    int rank,  
                    int maxdims,  
                    int coords[]) → salida
```

rank es el número de rank del proceso del que se quieren conocer las coordenadas cartesianas.

maxdims es el tamaño del vector coords.

En **coords** se retornan las coordenadas.

Ejemplo:

```
// Obtiene coordenadas de dos dimensiones del proceso con rank 6  
MPI_Cart_coords(cartcomm, 6, 2, coords);
```