Sistemas Concurrentes y Distribuidos.
Seminario 3. Introducción al paso de mensajes con MPI.

Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Curso 15-16

Índice

Sistemas Concurrentes y Distribuidos.
Seminario 3. Introducción al paso de mensajes con MPI.

Introducción

- El objetivo de esta práctica es familiarizar al alumno con el uso de la interfaz de paso de mensajes MPI y la implementación OpenMPI de esta interfaz.
- Se indicarán los pasos necesarios para compilar y ejecutar programas usando OpenMPI.
- Se presentarán las principales características de MPI y algunas funciones básicas de comunicación entre procesos.

Enlaces para acceder a información complementaria

- Web oficial de OpenMPI.
- Instalación de OpenMPI en Linux.
- Ayuda para las funciones de MPI.
- Tutorial de MPI.

Sección 1
Message Passing Interface (MPI)

¿Qué es MPI?

- MPI es un estándar de programación paralela mediante paso de mensajes que permite crear programas portables y eficientes.
- Proporciona un conjunto de funciones que pueden ser utilizadas en programas escritos en C, C++, Fortran y Ada.
- MPI-2 contiene más de 150 funciones para paso de mensajes y operaciones complementarias (con numerosos parámetros y variantes).
- Muchos programas paralelos se pueden construir usando un conjunto reducido de dichas funciones (hay 6 funciones básicas).

Modelo de Programación en MPI

- El esquema de funcionamiento implica un número fijo de procesos que se comunican mediante llamadas a funciones de envío y recepción de mensajes.
- Se sigue como modelo básico el estilo SPMD (Single Program Multiple Data), en el que todos los procesos ejecutan un mismo programa.
- También se permite seguir un modelo MPMD (Multiple Program Multiple Data), en el que cada proceso puede ejecutar un programa diferente.
- La creación e inicialización de procesos no está definida en el estándar, depende de la implementación. En OpenMPI sería algo así:
  - `$ mpirun -np 4 -machinefile maquinas prog1`
  - Comienza 4 copias del ejecutable `prog1`.
  - El archivo `maquinas` define la asignación de procesos a máquinas.

Aspectos de implementación

- Hay que hacer: **`#include`** *`"mpi.h"`*
  - Define constantes, tipos de datos y los prototipos de las funciones MPI.
- Las funciones devuelven un código de error:
  - **`MPI_SUCCESS`**: Ejecución correcta.
- **`MPI_Status`** es una estructura que se obtiene cada vez que se completa la recepción de un mensaje. Contiene 2 campos:
  - `status.`**`MPI_SOURCE`**: proceso fuente.
  - `status.`**`MPI_TAG`**: etiqueta del mensaje.
- **Constantes** para representar tipos de datos básicos de C/C++ (para los mensajes en MPI): **`MPI_CHAR`**, **`MPI_INT`**, **`MPI_LONG`**, **`MPI_UNSIGNED_CHAR`**, **`MPI_UNSIGNED`**, **`MPI_UNSIGNED_LONG`**, **`MPI_FLOAT`**, **`MPI_DOUBLE`**, **`MPI_LONG_DOUBLE`**, etc.
- **Comunicador**: es tanto un grupos de procesos como un contexto de comunicación. Todas las funciones de comunicación necesitan como argumento un comunicador.

Sección 2
Compilación y ejecución de programas MPI

Compilación y ejecución de programas en OpenMPI

OpenMPI es una implementación portable y de código abierto del estándar MPI-2, llevada a cabo por una serie de instituciones de ámbito tanto académico y científico como industrial.
OpenMPI ofrece varios scripts necesarios para trabajar con programas aumentados con llamadas a funciones de MPI. Los más importantes son estos dos:

- **`mpicxx`**: para compilar y enlazar programas C++ que hagan uso de MPI.
- **`mpirun`**: para ejecutar programas MPI.

El programa **`mpicxx`** puede utilizarse con las mismas opciones que el compilador de C/C++ usual, p.e.:

- `$ mpicxx -c ejemplo.c`
- `$ mpicxx -o ejemplo ejemplo.o`

Compilación y ejecución de programas MPI

La forma más usual de ejecutar un programa MPI es :

- `$ mpirun -np 4 ./ejemplo`
- El argumento `-np` sirve para indicar cuántos procesos ejecutarán el programa ejemplo. En este caso, se lanzarán cuatro procesos `ejemplo`.
- Como no se indica nada más, OpenMPI lanzará dichos procesos en la máquina local.

Sección 3
Funciones MPI básicas

Funciones MPI básicas

Hay 6 funciones básicas en MPI:

- **`MPI_Init`**: inicializa el entorno de ejecución de MPI.
- **`MPI_Finalize`**: finaliza el entorno de ejecución de MPI.
- **`MPI_Comm_size`**: determina el número de procesos de un comunicador.
- **`MPI_Comm_rank`**: determina el identificador del proceso en un comunicador.
- **`MPI_Send`**: operación básica para envío de un mensaje.
- **`MPI_Recv`**: operación básica para recepción de un mensaje.

Inicializar y finalizar un programa MPI

Se usan estas dos sentencias:

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

- Llamado antes de cualquier otra función MPI.
- Si se llama más de una vez durante la ejecución da un error.
- Los argumentos `argc, argv` son los argumentos de la línea de orden del programa.

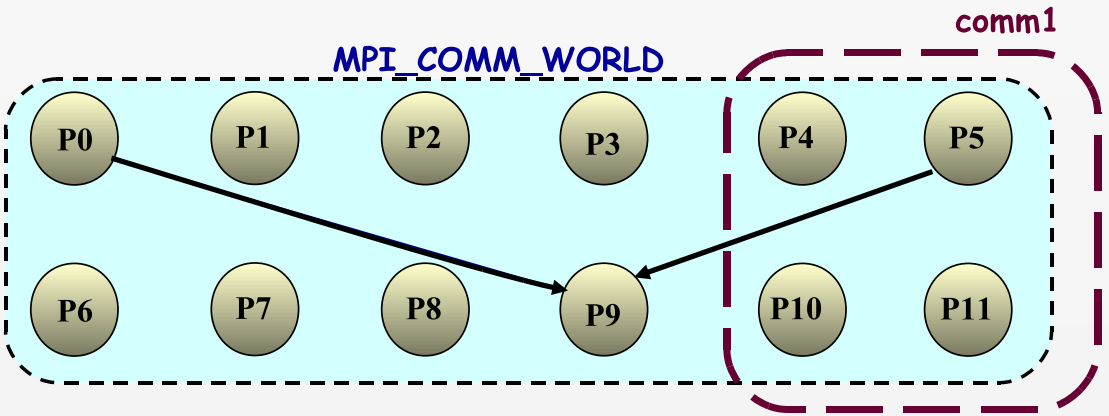
```
int MPI_Finalize( )
```

- Llamado al fin de la computación.
- Realiza tareas de limpieza para finalizar el entorno de ejecución

Subsección 3.1
Introducción a los comunicadores

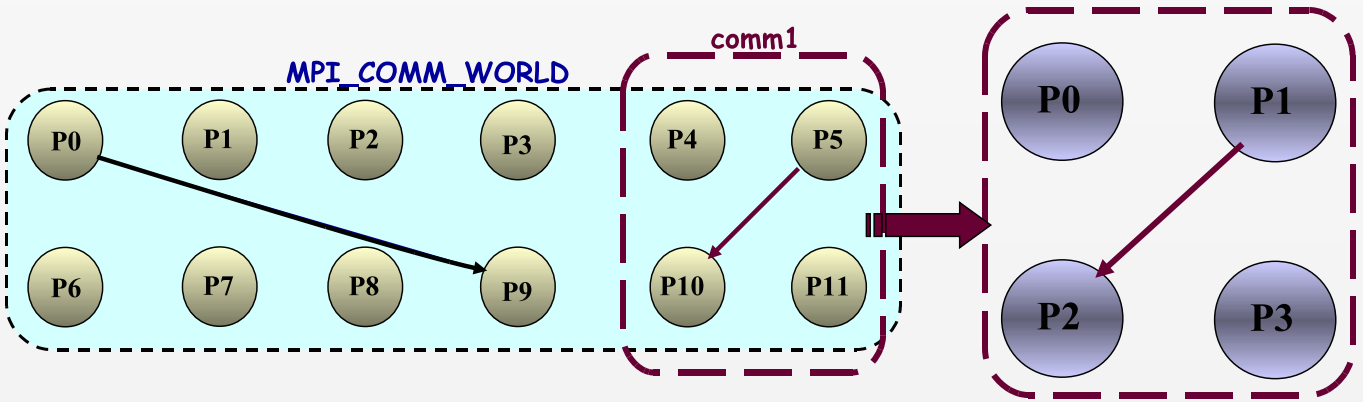
Introducción a los comunicadores MPI

- **Comunicador MPI**: es variable de tipo **`MPI_Comm`**.
- Un comunicador está constituido por:
  - Grupo de procesos: Subconjunto de procesos (pueden ser todos).
  - Contexto de comunicación: Ámbito de paso de mensajes en el que se comunican dichos procesos. Un mensaje enviado en un contexto sólo puede ser recibido en dicho contexto.
- Todas las funciones de comunicación de MPI necesitan como argumento un comunicador.



Introducción a los comunicadores

- La constante **`MPI_COMM_WORLD`** hace referencia al comunicador universal, un comunicador predefinido por MPI que incluye a todos los procesos de la aplicación (es el comunicador por defecto).
- La identificación de los procesos participantes en un comunicador es unívoca:
- Un proceso puede pertenecer a diferentes comunicadores.
- Cada proceso tiene un identificador: desde 0 a $P-1$ (P es el número de procesos del comunicador).
- Mensajes destinados a diferentes contextos de comunicación no interfieren entre sí.



Funciones para obtener información

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

- `size` : número de procesos que pertenecen al comunicador `comm`.
- ejemplo: **`MPI_Comm_size`**`(`**`MPI_COMM_WORLD`**`,&size)`.

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

- `rank`: Identificador del proceso llamador en `comm`.

```
#include "mpi.h"
#include <iostream>
using namespace std ;
int main(int argc, char *argv[])
{ int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  cout << "Hola desde proc. "
       << rank << " de " << size << endl;
  MPI_Finalize();
  return 0;
}
```

```
$ mpicxx -o hola hola.cpp
$ mpirun -np 4 hola

Hola desde proc. 0 de 4
Hola desde proc. 2 de 4
Hola desde proc. 1 de 4
Hola desde proc. 3 de 4
```

Subsección 3.2
Funciones básicas de envío y recepción de mensajes

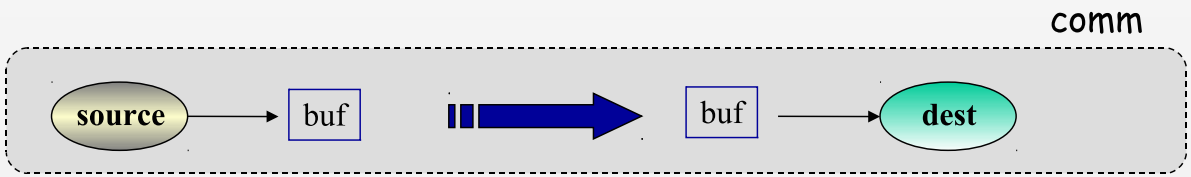
Envío y recepción de mensajes

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

- Envía los datos `num` elementos de tipo `datatype` almacenados a partir de `buf`) al proceso `dest` con la etiqueta `tag` (entero >= 0) dentro del comunicador `comm`.
- Implementa envio asíncrono seguro.

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

- Recibe mensaje de proceso `source` dentro del comunicador `comm` (recepción segura) y lo almacena en posiciones contiguas desde `buf`.
- Solo se recibe desde `source` con etiqueta `tag`, aunque existen argumentos comodín: **`MPI_ANY_SOURCE`**, **`MPI_ANY_TAG`**.



Envío y recepción de mensajes (2)

Los argumentos `num` y `datatype` determinan la longitud en bytes del mensaje. El objeto `status` es una estructura:

- Permite conocer el emisor (campo **`MPI_SOURCE`**), la etiqueta (campo **`MPI_TAG`**) y el número de items de un mensaje recibido.
- Para obtener la cuenta, el receptor debe conocer y proporcionar el tipo de los mismos. Se hace con **`MPI_Get_Count`**:

```
int MPI_Get_count( MPI_Status *status, MPI_Datatype dtype, int *num );
```

Ejemplo: Programa para dos procesos

```
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if ( rank == 0 )
{   value = 100 ;
    MPI_Send( &value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD );
} else
    MPI_Recv( &value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
MPI_Finalize( );
```

Emparejamiento de operaciones de envío y recepción

En MPI, una operación de envío (con etiqueta e) desde un proceso A **encajará** con una operación de recepción en un proceso B solo si se cumplen cada una de estas tres condiciones:

- A nombra a B como receptor y e como etiqueta.
- B especifica **`MPI_ANY_SOURCE`**, o bien nombra explictamente a A como emisor
- B especifica **`MPI_ANY_TAG`**, o bien nombra explicitamente e como etiqueta

Si una operación encaja con varias (un envío con varias posibles recepciones, o una recepción con varios posibles envíos), entonces:

- Se seleccionará de entre esas varias **la primera en iniciarse** (esto facilita al programador garantizar propiedades de equidad).

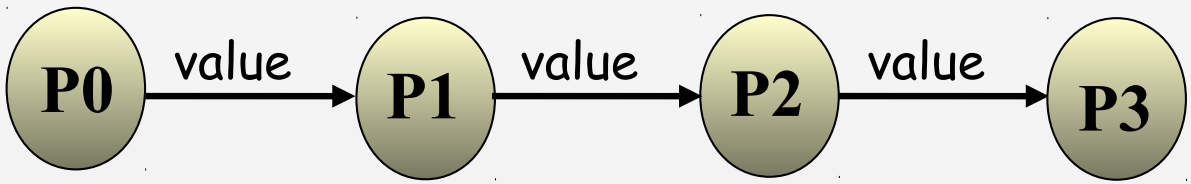
Interpretación de bytes transferidos

Es importante tener en cuenta que para determinar el emparejamiento MPI **no tiene en cuenta el tipo de datos ni la cuenta de items**. Es responsabilidad del programador asegurarse de que, en el lado del receptor:

- **Los bytes transferidos se interpretan con el mismo tipo de datos** que el emisor usó en el envío (de otra forma los valores leídos son indeterminados).
- **Se sabe exactamente cuantos items de datos** se han recibido (en otro caso el receptor podría leer valores indeterminados de zonas de memoria no escritas por MPI).
- **Se ha reservado memoria suficiente** para recibir todos los datos (de no hacerse, MPI escribiría erróneamente fuera de la memoria correspondiente a la variable especificada en el receptor).

Ejemplo: Difusión de mensaje en una cadena de procesos

```
int main(int argc, char *argv[])
{ int rank, size,value; MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);MPI_Comm_size(MPI_COMM_WORLD,&size );
  do
  { if (rank == 0)
    { scanf( "%d", &value );
      MPI_Send( &value, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD );
    }
    else
    { MPI_Recv( &value, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD, &status );
      if (rank < size-1)
        MPI_Send( &value, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD );
    }
    cout<< "Soy el proceso "<<rank<<" y he recibido "<<value<<endl;
  }
  while ( value >= 0 );
  MPI_Finalize(); return 0;
}
```



Sección 4
Paso de mensajes síncrono en MPI

Envío en modo síncrono

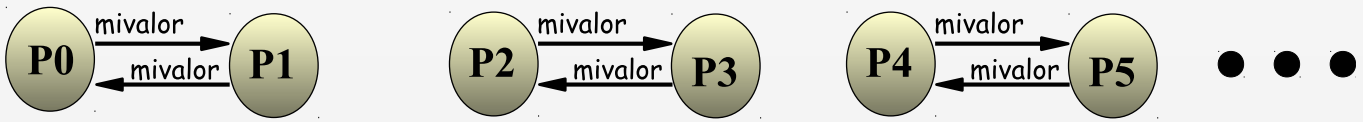
En MPI existe una función de envío **síncrono** (siempre es **seguro**):

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

- Presenta los mismos argumentos que **`MPI_Send`**
- La operación de envío finalizará solo cuando el correspondiente **`MPI_Recv`** sea invocado y el receptor haya comenzado a recibir el mensaje, y además los datos hayan terminado de leerse en el emisor.
- Por **`MPI_Ssend`** es **seguro**: cuando devuelve el control, la zona de memoria que alberga el dato podrá ser reutilizada.
- Si la correspondiente operación de recepción usada es **`MPI_Recv`**, la semántica del paso de mensajes es puramente síncrona (existe una cita entre emisor y receptor).

Ejemplo: Intercambio síncrono entre pares de procesos

```
int main(int argc, char *argv[]) {
 int rank, size, mivalor, valor;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size( MPI_COMM_WORLD, &size );
 mivalor=rank*(rank+1);
 if (rank %2 == 0) {    // El orden de las operaciones es importante
     MPI_Ssend( &mivalor, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD );
     MPI_Recv( &valor, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD, &status );}
 else {
     MPI_Recv( &valor, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD, &status );
     MPI_Ssend( &mivalor, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD );}
 cout<< "Soy el proceso "<<rank<<" y he recibido "<<valor<<endl;
 MPI_Finalize(); return 0;
}
```



Sección 5
Comunicación insegura

Comunicación insegura

- Las operaciones de comunicación vistas anteriormente son *seguras* (y por tanto pueden hacer esperar al proceso)
  - Incluso **`MPI_Send`** puede causar el bloqueo del emisor si no ha comenzado la operación de recepción correspondiente y no se dispone de memoria intermedia suficiente para copiar el mensaje completo.
- Se necesitan operaciones de comunicación no bloqueantes
  - Sondeo de mensajes:
    - **`MPI_Iprobe`**: Chequeo no bloqueante para un mensaje.
    - **`MPI_Probe`**: Chequeo bloqueante para un mensaje.
  - Envío-Recepción inseguro:
    - **`MPI_Isend`**: Inicia envío pero retorna antes de copiar en buffer.
    - **`MPI_Irecv`**: Inicia recepción pero retorna antes de recibir.
    - **`MPI_Test`**: Chequea si la operación no bloqueante ha finalizado.
    - **`MPI_Wait`**: Bloquea hasta que acabe la operación no bloqueante.

Comunicación asíncrona

El acceso no estructurado a un recurso compartido requiere comprobar la existencia de mensajes sin recibirlos.

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

- No bloquea. Si hay algún mensaje, no se recibe (se puede hacer después con **`MPI_Recv`**, p.ej.).
- Si `flag > 0`, eso indica que hay un mensaje pendiente que encaja con (`source`, `tag`, `comm`).
- El argumento `status` permite obtener más información sobre el mensaje pendiente de recepción.

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

- Retorna sólo cuando hay algún mensaje que encaje con los argumentos.
- Permite esperar la llegada de un mensaje sin conocer su procedencia, etiqueta o tamaño.

Sondeo continuo de varias fuentes desconocidas

```
int rank, size, flag, buf, src,tag;
MPI_Status status ;
...
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if (rank == 0) // proceso 0 recibe de todos los demás
{ int contador = 0;
  while ( contador < 10*(size-1) )
  { MPI_Iprobe(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD, &flag, &status);
    if ( flag > 0 )
    { MPI_Recv( &buf, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                                  MPI_COMM_WORLD, &status );
      src = status.MPI_SOURCE; tag = status.MPI_TAG ;
      cout << "mensaje de " << src << " con tag= " << tag << endl;
      contador++;
    }
  }
  cout << "total de mensajes recibidos: " << contador << endl;
}
else  // resto de procesos envían 10 items cada uno a proceso 0
  for( int i = 0 ; i < 10 ; i++ )
     MPI_Send( &buf, 1, MPI_INT, 0, i, MPI_COMM_WORLD );
...
```

Recepción con tamaño y fuente desconocidos

Se espera la llegada de un mensaje, una vez que llega, y antes de recibirlo, se reserva justo la memoria suficiente para contener el mensaje:

```
int num, *buf, source ;
MPI_Status status ;

// bloqueo hasta que se detecte un mensaje:
MPI_Probe( MPI_ANY_SOURCE, 0,comm, &status ) ;

// averigua el tamaño y el proceso emisor del mensaje:
MPI_Get_count( status, MPI_INT, &num );
source = status.MPI_SOURCE ;

// reserva memoria para recibir el mensaje:
buf = malloc( num*sizeof(int) );

// se recibe el mensaje:
MPI_Recv( buf, num, MPI_INT, source, 0, comm, &status );
```

Operaciones inseguras

Se pueden usar estas dos funciones:

```
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)
```

Los argumentos son similares a **MPI_Send** excepto:

- Argumento `request`: Identifica operación cuyo estado se pretende consultar o se espera que finalice.
- No incluye argumento `status`: Se puede obtener con otras 2 funciones de chequeo de estado.

Cuando ya no se va a usar una variable **MPI_Request**, se puede usar:

```
int MPI_Request_free( MPI_Request *request )
```

- Libera la memoria usada por la variable `request`.

Operaciones inseguras. Chequeo de estado

Para comprobar si una operación insegura ha finalizado o no, se usa:

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

- Escribe en `flag`.
- Si `flag > 0` entonces la operación identificada ha finalizado, libera `request` e inicializa `status`.

Para esperar bloqueado hasta que termine una operación, se usa:

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

- Produce bloqueo hasta que la operación chequeada finaliza (es segura).

Hay que tener en cuenta que es posible conectar operaciones inseguras con sus contrapartes seguras.

Intercambio de mensajes con operaciones inseguras

```
int main(int argc, char *argv[])
{ int rank, size, vecino, valor_env, valor_rec;
  MPI_Status status;
  MPI_Request request_env,request_rec;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  // calcular valor a enviar y número de proceso vecino:
  valor_env = rank*(rank+1);
  if (rank %2 == 0) vecino = rank+1 ;
  else              vecino = rank-1 ;
  // envio y recepción simultáneos (en cualquier orden)
  MPI_Irecv( &valor_rec,1,MPI_INT,vecino,0,MPI_COMM_WORLD,&request_rec);
  MPI_Isend( &valor_env,1,MPI_INT,vecino,0,MPI_COMM_WORLD,&request_env);
  // ... aquí se puede hacer algo que no use valor_rec ni altere valor_env
  // bloqueo hasta que sea seguro:
  MPI_Wait( &request_env, &status );
  MPI_Wait( &request_rec, &status );
  // ya esta:
  cout << "Soy el proceso "<<rank<<" y he recibido "<<valor<<endl ;
  MPI_Finalize(); return 0;
}
```