UNIVERSIDAD
DE GRANADA

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

Carlos Ureña / Jose M. Mantas / Pedro Villar

2020-21

Grado en Ingeniería Informática / Grado en Ingeniería Informática y Matemáticas.
Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Seminario 3. Introducción a paso de mensajes con MPI.
Índice.

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 que define una API para 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.

- El modelo básico es SPMD (*Single Program Multiple Data*): todos los procesos ejecutan un mismo programa.
- Permite el modelo MPMD (*Multiple Program Multiple Data*): 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:
  `mpirun -np 4 -machinefile maquinas prog1`
  - Comienza 4 copias del ejecutable `prog1`.
  - El archivo `maquinas` define la asignación de procesos a ordenadores del sistema distribuido.

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 un tipo estructura con los metadatos de los mensajes:
  - `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/o 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.:

- `$mpicxx -std=c++11 -c ejemplo.cpp`
- `$mpicxx -std=c++11 -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 la opción `-machinefile`, OpenMPI lanzará dichos 4 procesos en el mismo ordenador donde se ejecuta `mpirun`.
- Con la opción `machinefile`, podemos realizar asociaciones de procesos a distintos ordenadores.

Sección 3.
Funciones MPI básicas.

3.1. Introducción a los comunicadores
3.2. Funciones básicas de envío y recepción de mensajes

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

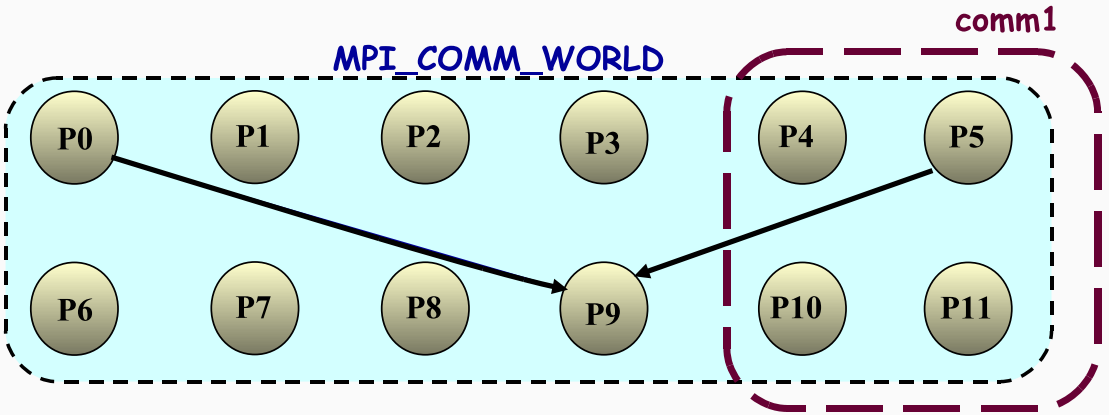
Subsección 3.1.
Introducción a los comunicadores.

Introducción a los comunicadores (1)

Un **Comunicador MPI** es una variable de tipo **`MPI_Comm`**. 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.

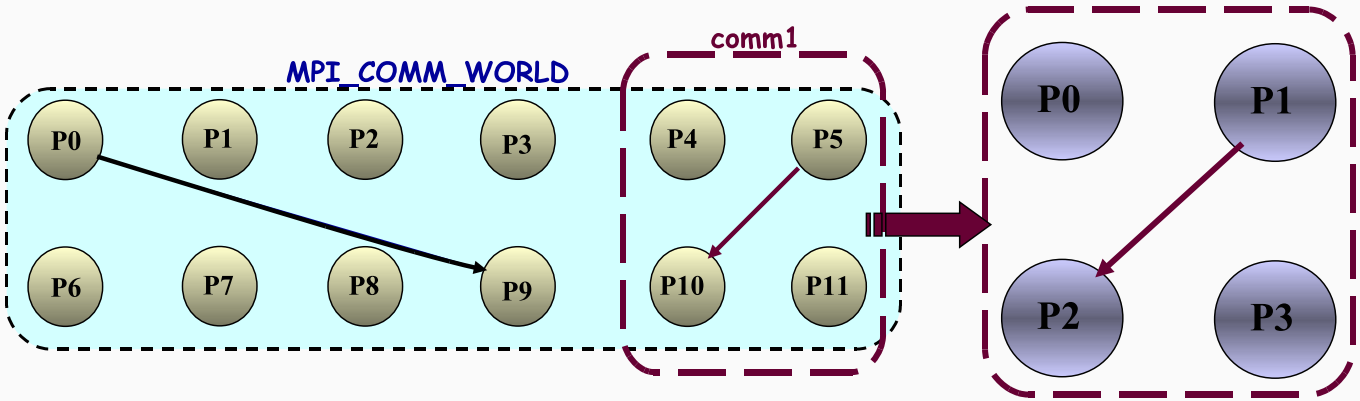
Todas las funciones de comunicación de MPI necesitan como argumento un comunicador.



Introducción a los comunicadores (2)

La constante **MPI_COMM_WORLD** hace referencia al **comunicador universal**, está predefinido e incluye todos los procesos lanzados:

- 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.



Consulta del número de procesos

La función **MPI_Comm_size** tiene esta declaración:

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

- Escribe en el entero apuntado por **size** el número total de procesos que forman el comunicador **comm**.
- Si usamos el comunicador universal, podemos saber cuantos procesos en total se han lanzado en una aplicación, por ejemplo:

```
int num_procesos ; // contendrá el total de procesos de la aplic.
MPI_Comm_size( MPI_COMM_WORLD, &num_procesos );
cout << "El numero total de procesos es: " << num_procesos << endl ;
```

Consulta del identificador del proceso

La función **MPI_Comm_rank** está declarada como sigue:

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

- Escribe en el entero apuntado por **rank** el número de proceso que llama. Este número es el número de orden dentro del comunicador **comm** (empezando en 0). Ese número se suele llamar *rank* o *identificador* del proceso en el comunicador.
- Se suele usar al inicio de una aplicación MPI, con el comunicador universal, como en este ejemplo:

```
int id_propio ; // contendrá el número de proceso que llama
MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
cout << "mi número de proceso es:"  << id_propio << endl ;
```

Normalmente, usaremos este número de orden en el comunicador universal para identificar cada proceso.

Ejemplo de un programa simple

En el archivo **holamundo.cpp** vemos un ejemplo sencillo:

```
#include <mpi.h>
#include <iostream>
using namespace std;

int main( int argc, char *argv[] )
{
  int id_propio, num_procesos ;
  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &num_procesos );
  MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
  cout <<"Hola desde proceso " <<id_propio <<" de " <<num_procesos <<endl;
  MPI_Finalize();
  return 0;
}
```

Si se compila y ejecuta con 4 procesos obtenemos esta salida:

```
$mpicxx -std=c++11 -o hola holamundo.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 asíncrono seguro de un mensaje (MPI_Send)

Un proceso puede enviar un mensaje usando **MPI_Send**:

```
int MPI_Send( void *buf_emi, 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_emi**) al proceso **dest** dentro del comunicador **comm**.
- El entero **tag** (*etiqueta*) se transfiere junto con el mensaje, y suele usarse para clasificar los mensajes en distintas categorías o tipos, en función de sus etiquetas. Es no negativo.
- Implementa envio **asíncrono seguro**: tras acabar **MPI_Send**
  - MPI ya ha leído los datos de **buf_emi**, y los ha copiado a otro lugar, por tanto podemos volver a escribir sobre **buf_emi** (el envío es **seguro**).
  - el receptor no necesariamente ha inciado ya la recepción del mensaje (el envío es **asíncrono**)

Recepción segura síncrona de un mensaje (MPI_Recv)

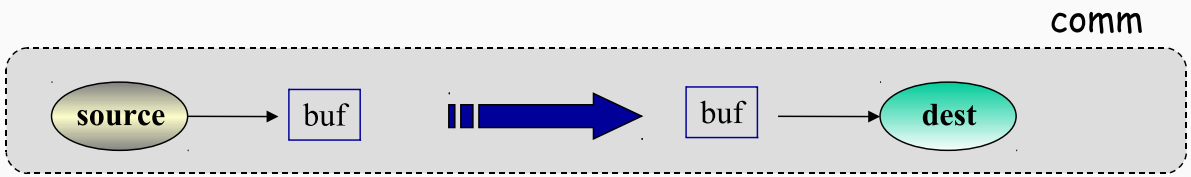
Un proceso puede recibir un mensaje usando **MPI_Recv**, que se declara como sigue:

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

- Espera hasta recibir un mensaje del proceso **source** dentro del comunicador **comm** con la etiqueta **tag**, y escribe los datos en posiciones contiguas desde **buf_rec**.
- Puesto que se espera a que el emisor envíe, es una recepción **síncrona**. Puesto que al acabar ya se pueden leer en **buf_rec** los datos transmitidos, es una recepción **segura**.
- Se pueden dar valores especiales o *comodín*:
  - Si **source** es **MPI_ANY_SOURCE**, se puede recibir un mensaje de cualquier proceso en el comunicador
  - Si **tag** es **MPI_ANY_TAG**, se puede recibir un mensaje con cualquier etiqueta.

Envío y recepción de mensajes (2)

Los datos se copian desde **buf_emi** hacia **buf_rec**:



- Los argumentos **num** y **datatype** determinan la longitud en bytes del mensaje. El objeto **status** es una estructura con el emisor (campo **MPI_SOURCE**), la etiqueta (campo **MPI_TAG**).

Para obtener la cuenta de valores recibidos, usamos **status**

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

- Escribe en el entero apuntado por **num** el número de items recibidos en una llamada **MPI_Recv** previa. El receptor debe conocer y proporcionar el tipo de los datos (**dtype**).

Ejemplo sencillo (1): estructura del programa

Dos procesos: emisor y receptor (archivo `sendrecv1.cpp`)

```
#include <iostream>
#include <mpi.h>  // incluye declaraciones de funciones, tipos y ctes. MPI
using namespace std;
const int id_emisor             = 0,  // identificador de emisor
          id_receptor           = 1,  // identificador de receptor
          num_procesos_esperado = 2;  // numero de procesos esperados

int main( int argc, char *argv[] )
{
  int id_propio, num_procesos_actual; // identificador propio, núm.procesos
  MPI_Init( &argc, &argv );           // inicializa MPI
  MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );           // lee mi ident.
  MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual ); // lee num.procs.
  if ( num_procesos_esperado == num_procesos_actual )    // si n.procs. ok
  {
     ..... // hacer envío o recepción (según id_propio)
  }
  else if ( id_propio == 0 ) // si error, el primer proceso informa
    cerr<< "Esperados 2 procs., hay: "<< num_procesos_actual<< endl;
  MPI_Finalize();  // terminar MPI: debe llamarse siempre por cada proceso.
  return 0;        // terminar proceso
}
```

Ejemplo sencillo (2): envío y recepción

Cuando el número de procesos es correcto, el comportamiento de cada proceso depende de su rol, es decir, de su identificador propio (**id_propio**). En este caso, uno emite (**id_emisor**) y otro recibe (**id_receptor**):

```
if ( id_propio == id_emisor ) // soy emisor: enviar
{
  int valor_enviado = 100 ; // buffer del emisor (tiene 1 entero: MPI_INT)
  MPI_Send( &valor_enviado, 1, MPI_INT, id_receptor, 0, MPI_COMM_WORLD );
  cout << "Emisor ha enviado: " << valor_enviado << endl ;
}
else // soy receptor: recibir
{
  int valor_recibido; // buffer del receptor (tiene 1 entero: MPI_INT)
  MPI_Status estado;  // estado de la recepción
  MPI_Recv( &valor_recibido,1,MPI_INT,id_emisor,0,MPI_COMM_WORLD,&estado);
  cout << "Receptor ha recibido: " << valor_recibido << endl ;
}
```

Emparejamiento de operaciones de envío y recepción

En MPI, una operación de envío (con etiqueta e) realizada por un proceso emisor A **encajará** con una operación de recepción realizada por un proceso receptor B si y 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 explíctamente a A como emisor
- B especifica **MPI_ANY_TAG**, o bien nombra explícitamente e como etiqueta

Si al iniciar una operación de recepción se determina que encaja con varias operaciones de envío ya iniciadas:

- Se seleccionará de entre esos varios envíos **el primero que se inició** (esto facilita al programador garantizar propiedades de equidad).

Interpretación de bytes transferidos

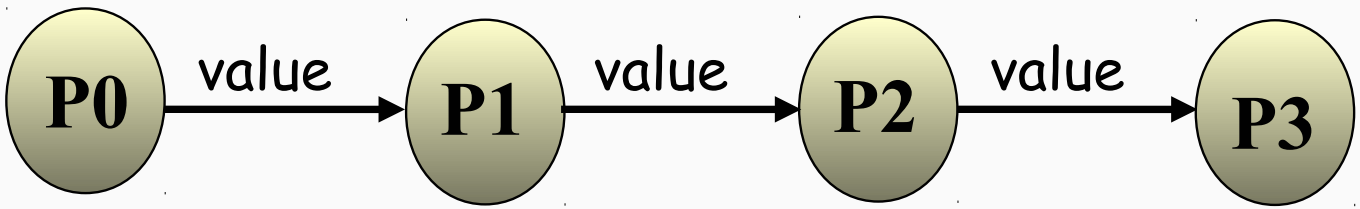
Es importante tener en cuenta que para determinar el emparejamiento MPI **no tiene en cuenta el tipo de datos ni la cuenta de items**. Es responsabilidad del programador asegurarse de que, en el lado del receptor:

- **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

En este ejemplo

- Funciona con un número de procesos como mínimo igual a 2.
- Todos los procesos ejecutan un bucle, en cada iteración:
  - El primer proceso (identificador $= 0$) pide un valor entero por teclado.
  - El resto de procesos (identificador > 0), reciben cada uno un valor del proceso anterior.
  - Todos los procesos (excepto el último) envían su valor al siguiente proceso.
- El bucle acaba cuando se ha recibido o leído un valor negativo.



Difusión en cadena: estructura del programa

El ejemplo está en el archivo `sendreceive2.cpp`:

```
const int num_procesos_min = 2 ; // número mínimo de procesos

int main( int argc, char *argv[] )
{
  int id_propio, num_procesos_actual ;

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
  MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );

  if ( num_procesos_min <= num_procesos_actual ) // núm. procs. ok
  {
     ........ // bucle de envío/recepción
  }
  else if ( id_propio == 0 ) // si error, el primero proceso informa
    cerr << "Número de procesos menor que mínimo ("
         << num_procesos_min << ")" << endl;

  MPI_Finalize();
  return 0;
}
```

Difusión en cadena: bucle de envío/recepción

Si el número de procesos es correcto, cada proceso hace un bucle. El código tiene esta forma:

```
const int  id_anterior  = id_propio-1,  // ident. proceso anterior
           id_siguiente = id_propio+1 ; // ident. proceso siguiente
int        valor;  // valor recibido o leído, y enviado
MPI_Status estado; // estado de la recepción

do
{
  if ( id_anterior < 0 ) // si soy el primer proceso (id_anterior es -1):
    cin >> valor ;       //    pedir valor por teclado
  else                   // si no soy el primero: recibir valor de anterior
    MPI_Recv( &valor, 1,MPI_INT, id_anterior, 0, MPI_COMM_WORLD, &estado);

  cout<< "Proc."<< id_propio<< ": recibido/leído: "<< valor<< endl ;

  if ( id_siguiente < num_procesos_actual ) // so no soy último: enviar
    MPI_Send( &valor, 1, MPI_INT, id_siguiente, 0, MPI_COMM_WORLD );
}
while( valor >= 0 ); // acaba cuando se teclea un valor negativo
```

Sección 4.
Paso de mensajes síncrono en MPI.

Envío en modo síncrono (y seguro) (MPI_Ssend)

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

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

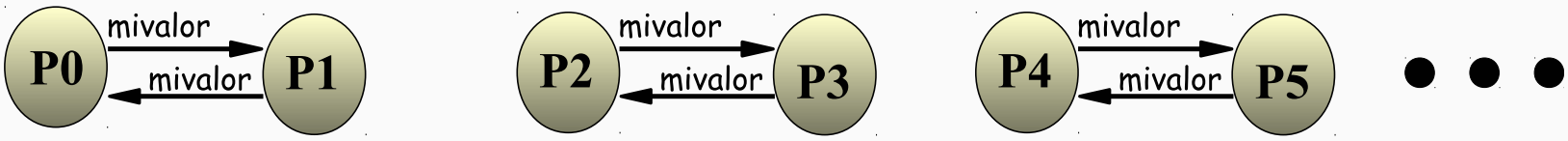
- Inicia un envío, lee datos y espera el inicio de la recepción, con los mismos argumentos que **MPI_Send**.
- Es **síncrono y seguro**. Tras acabar **MPI_Ssend**
  - ya se ha iniciado en el receptor una operación de recepción que encaja con este envío (es **síncrono**),
  - los datos ya se han leído de **buf_emi** y se han copiado a otro lugar. Por tanto se puede volver a escribir en **buf_emi** (es **seguro**)
- 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 de intercambio síncrono

En este ejemplo hay un número par de procesos:

- Los procesos se agrupan por parejas. Cada proceso enviará un dato a su correspondiente pareja o vecino.
- Los envíos se hacen usando envío síncrono (con **`MPI_Ssend`**).
- Si todos los procesos hacen envío seguido de recepción (o todos lo hacen al revés), **habría interbloqueo con seguridad**.
- Para evitarlo, los procesos pares hacen envío seguido de recepción y los procesos impares recepción seguida de envío.

Por tanto, el esquema de funcionamiento se puede ver así:



Intercambio síncrono: estructura del programa

La estructura de **main** (en `intercambio_sincrono.cpp`) es esta:

```
#include <iostream>
#include <mpi.h>
using namespace std;

int main( int argc, char *argv[] )
{
   int id_propio, num_procesos_actual;
   MPI_Init( &argc, &argv);
   MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
   MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );

   if ( num_procesos_actual % 2 == 0 ) // si número de procesos correcto (par)
   {
      .... // envío/recepción hacia/desde proceso vecino
   }
   else if ( id_propio == 0 ) // si n.procs. impar, el primero da error
      cerr << "Error: se esperaba un número par de procesos" << endl ;

   MPI_Finalize();
   return 0;
}
```

Intercambio síncrono: envío/recepción

El envío y recepción con el proceso vecino se puede hacer así:

```
MPI_Status estado ;
int valor_env = id_propio*(id_propio+1),// dato a enviar (cualquiera vale)
    valor_rec,                          // valor recibido
    id_vecino ;                         // identificador de vecino

if ( id_propio % 2 == 0 )  // si proceso par: enviar y recibir
{
   id_vecino = id_propio+1 ; // el vecino es el siguiente
   MPI_Ssend( &valor_env,1,MPI_INT,id_vecino,0,MPI_COMM_WORLD );
   MPI_Recv ( &valor_rec,1,MPI_INT,id_vecino,0,MPI_COMM_WORLD,&estado );
}
else  // si proceso impar: recibir y enviar
{
   id_vecino = id_propio-1 ; // el vecino es el anterior
   MPI_Recv ( &valor_rec,1,MPI_INT,id_vecino,0,MPI_COMM_WORLD,&estado );
   MPI_Ssend( &valor_env,1,MPI_INT,id_vecino,0,MPI_COMM_WORLD );
}
cout << "El proceso " << id_propio << " ha recibido " << valor_rec
     << " del proceso  " << id_vecino << endl ;
```

Sección 5.
Sondeo de mensajes.

Sondeo de mensajes

MPI incorpora dos operaciones que permiten a un proceso receptor averiguar si hay algún mensaje pendiente de recibir (en un comunicador), y en ese caso obtener los metadados de dicho mensaje. Esta consulta:

- no supone la recepción del mensaje.
- se puede restringir a mensajes de un emisor.
- se puede restringir a mensajes con una etiqueta.
- cuando hay mensaje, permite obtener los metadatos: emisor, etiqueta y número de items (el tipo debe ser conocido).

Las dos operaciones son

- **`MPI_Iprobe`**: consultar si hay o no algún mensaje pendiente en este momento.
- **`MPI_Probe`**: esperar bloqueado hasta que haya al menos un mensaje.

Espera bloqueada con MPI_Probe

La función **MPI_Probe** tiene esta declaración:

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

El proceso que llama queda bloqueado hasta que haya al menos un mensaje enviado a dicho proceso (en el comunicador **comm**) que encaje con los argumentos.

- **source** puede ser un identificador de emisor o **MPI_ANY_SOURCE**
- **tag** puede ser una etiqueta o bien **MPI_ANY_TAG**.
- **status** permite conocer los metadados del mensaje, igual que se hace tras **MPI_Recv**.
- Si hay más de un mensaje disponible, los metadatos se refieren al primero que se envió.

Consulta previa a recepción, con `MPI_Probe`

En el archivo **`ejemplo_probe.cpp`** vemos un programa en el cual los procesos mandan cadenas de texto a un proceso receptor, que las imprime al recibirlas. El receptor reserva justo la memoria necesaria para cada cadena:

```
int num_chars_rec ; // número de caracteres del mensaje (sin el cero al final)
MPI_Status estado ; // contiene metadatos del mensaje

// esperar mensaje y leer la cuenta de caracteres (sin recibirlo)
MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &estado );
MPI_Get_count( &estado, MPI_CHAR, &num_chars_rec );

// reservar memoria para el mensaje y recibirlo
char * buffer = new char[num_chars_rec+1] ;
MPI_Recv( buffer, num_chars_rec, MPI_CHAR, estado.MPI_SOURCE, MPI_ANY_TAG,
          MPI_COMM_WORLD, &estado );
buffer[num_chars_rec] = 0 ; // añadir un cero al final

// imprimir el mensaje y liberar la memoria que ocupa
cout << buffer << endl ;
delete [] buffer ;
```

Consulta no bloqueante con MPI_Iprobe

La función **MPI_Iprobe** tiene esta declaración:

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

Al terminar, el entero apuntado por **flag** será mayor que 0 solo si hay algún mensaje enviado al proceso que llama, y que encaje con los argumentos (en el comunicador **comm**). Si no hay mensajes, dicho entero es 0.

- No supone bloqueo alguno.
- La consulta se refiere a los mensajes pendientes en el momento de la llamada.
- Los parámetros (excepto **flag**) se interpretan igual que en **MPI_Probe**.

Recepción con prioridad usando `MPI_Iprobe`

Aquí (`ejemplo_iprobe.cpp`) un proceso receptor determina si hay mensajes pendientes de recibir de los emisores con idents. desde `id_min` hasta `id_max`, ambos incluidos.

Si hay más de uno, recibe del emisor con identificador más bajo. Si no hay mensajes pendientes, queda a la espera hasta recibir el primero de cualquier emisor:

```
MPI_Status estado ; int hay_mens, id_emisor, valor ;

// comprobar si hay mensajes, en orden creciente de los posibles emisores
for( unsigned id_emisor = id_min ; id_emisor <= id_max ; id_emisor++ )
{
  MPI_Iprobe( id_emisor, MPI_ANY_TAG, MPI_COMM_WORLD, &hay_mens, &estado);
  if ( hay_mens ) break ; // si hay mensaje: terminar consulta
}
if ( ! hay_mens )                  // si no hay mensaje:
   id_emisor = MPI_ANY_SOURCE ;    // aceptar de cualquiera

// recibir el mensaje del emisor concreto o de cualquiera
MPI_Recv( &valor, 1,MPI_INT, id_emisor, 0, MPI_COMM_WORLD, &estado );
```

Sección 6.
Comunicación insegura.

Operaciones inseguras.

MPI ofrece la posibilidad de usar **operaciones inseguras** (asíncronas). Permiten el inicio de una operación de envío o recepción, y después el emisor o el receptor puede continuar su ejecución de forma concurrente con la transmisión:

- **`MPI_Isend`**: inicia envío pero retorna antes de leer el buffer.
- **`MPI_Irecv`**: inicia recepción pero retorna antes de recibir.

En algún momento posterior se puede comprobar si la operación ha terminado o no, se puede hacer de dos formas:

- **`MPI_Wait`**: espera bloqueado hasta que acabe el envío o recepción.
- **`MPI_Test`**: comprueba si el envio o recepción ha finalizado o no. no supone espera bloqueante.

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:

- **request** es un **ticket** que permitirá después identificar la operación cuyo estado se pretende consultar o se espera que finalice.
- La recepción no incluye argumento **status** (se obtiene con las operaciones de consulta de estado de la operación).

Cuando ya no se va a usar una variable **MPI_Request**, se puede liberar la memoria que usa con **MPI_Request_free**, declarada así:

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

Consulta de estado de operaciones inseguras

La función **MPI_Test** comprueba la operación identificada por un ticket (**request**) y escribe en **flag** un número > 0 si ha acabado, o bien 0 si no ha acabado:

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

MPI_Wait sirve para esperar bloqueado hasta que termine una operación:

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

En ambas funciones, una vez terminada la operación referenciada por el ticket:

- podemos usar el objeto **status** para consultar los metadatos del mensaje.
- la memoria usada por **request** es liberada por MPI (no hay que llamar a **MPI_Request_free**).

Ventajas y seguridad en operaciones inseguras

Las operaciones inseguras:

- Permiten simultanear trabajo útil en el emisor y/o receptor con la lectura, transmisión y recepción del mensaje.
- Aumentan el paralelismo potencial y por tanto pueden mejorar la eficiencia en tiempo.

Las operaciones de consulta de estado (`MPI_Wait` y `MPI_Test`) permiten saber cuando **es seguro** volver a usar el buffer de envío o recepción, ya que nos dicen que la operación ha acabado cuando

- se han leído y copiado los datos del buffer del emisor (si el ticket se refiere a una operación `MPI_Isend`).
- se han recibido los datos en el buffer del receptor (si el ticket se refiere a una operación `MPI_Irecv`).

Una operación insegura se puede emparejar con una operación segura y/o síncrona.

Intercambio de mensajes con operaciones inseguras

A modo de ejemplo, vemos una solución (archivo `intercambio_nobloq.cpp`) con operaciones inseguras que evita el interbloqueo asociado al intercambio síncrono:

```
int         valor_enviado = id_propio*(id_propio+1), // dato a enviar
            valor_recibido, id_vecino ;
MPI_Status  estado ;
MPI_Request ticket_envio, ticket_recepcion;

if ( id_propio % 2 == 0 ) id_vecino = id_propio+1 ;
else                      id_vecino = id_propio-1 ;

// iniciar ambas operaciones simultáneamente (el orden es irrelevante)
MPI_Irecv( &valor_recibido, 1, MPI_INT, id_vecino, 0, MPI_COMM_WORLD,
           &ticket_recepcion );
MPI_Isend( &valor_enviado,  1, MPI_INT, id_vecino, 0, MPI_COMM_WORLD,
           &ticket_envio );

// esperar hasta que acaben ambas operaciones
MPI_Wait( &ticket_envio, &estado );
MPI_Wait( &ticket_recepcion, &estado );
```

Fin de la presentación.