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

Javier Gómez López

Message Passing Interface (MPI)

La creación e inicialización de procesos no está definida en el estándar, depende de la implementación. En OpenMPI sería:

```
mpirun -oversubscribe -np 4 -machinefile maquinas prog1_mpi_exe
```

En este caso, comienzan 4 copias del ejecutable `prog1_mpi_exe`. El archivo `maquinas` define la asignación de procesos a ordenadores del sistema distribuido.

Es necesario hacer `#include <mpi.h>`. Las funciones de esta API devuelven un código de error:

- `MPI_SUCCESS`: Ejecución correcta.

Por otro lado, `MPI_Status` es un tipo de estructura con los metadatos de los mensajes:

- `status.MPI_SOURCE`: proceso fuente.
- `status.MPI_TAG`: etiqueta del mensaje.

Las **constantes** para representar tipos de datos básicos de C/C++ son: `MPI_CHAR, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOBULE, MPI_LONG_DOUBLE`, etc.

El **comunicador** es tanto un grupo de procesos como un contexto de comunicación. Todas las funciones de comunicación necesitan como argumento un comunicador.

Compilación y ejecución de programas MPI

`OpenMPI` es una implementación portable y *open source* del estándar MPI-2. Ofrece varios *scripts* necesarios para trabajar con programas aumentados con llamadas a funciones de MPI. Los más importantes son estos dos:

- `mpicxx`: compila y/o enlaza programas C++ con MPI.
- `mpirun`: ejectua programas MPI.

Se compila con las opciones habituales:

```
mpicxx -std=c++11 -c ejemplo.cpp
mpicxx -std=c++11 -o ejemplo_mpi_exe ejemplo.o
```

o directamente

```
mpicxx -std=c++11 -o ejemplo_mpi_exe ejemplo.cpp
```

La forma más usual de ejecutar un programa MPI es:

```
mpirun -oversubscribe -np 4 ./ejemplo_mpi_exe
```

Veamos los argumentos utilizados:

- `-np`: indica cuántos procesos ejecutarán el programa ejemplo.
- `-machinefile`: esta directiva no se usa, luego los 4 procesos del ejemplo se lanzarán en el mismo ordenador donde se ejecuta `mpirun`.
- `-oversubscribe`: puede ser necesaria si el número de procesadores disponibles en algún ordenador es inferior al número de procesos que se quieren lanzar en ese ordenador.

Funciones MPI básicas

Hay 6 funciones básicas:

- `MPI_Init`: inicializa el entorno de ejecución MPI.

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

  Es llamado antes cualquier otra función MPI. Los argumentos `argc, argv` son los argumentos de la línea de orden del programa.

- `MPI_Finalize`: finaliza el entorno de ejecución de MPI.

  ```
  int MPI_Finalize()
  ```

  Llamado al fin de la computación.

- `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.

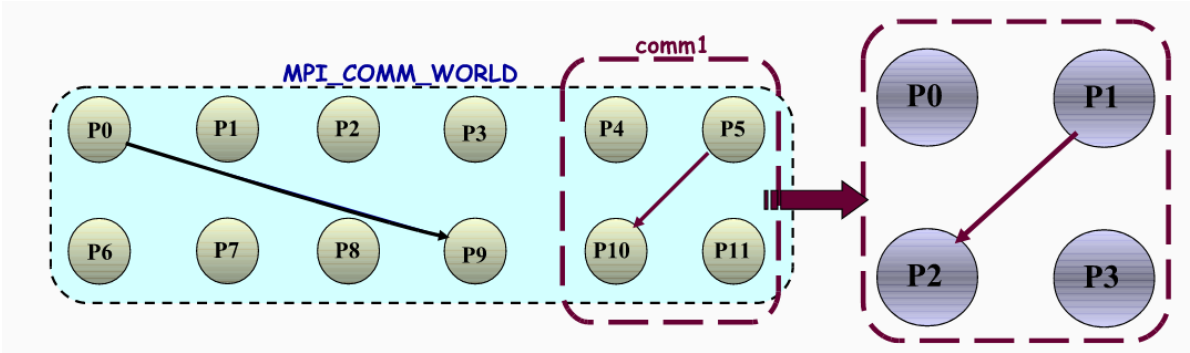
Introducción a los comunicadores

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.

La constante `MPI_COMM_WORLD` hace referencia al **comunicador universal**, está predefinido en 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.



La función `MPI_Comm_size` tiene esta declaración:

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

Escribe en `size` el número total de procesos que forman el comunicador `comm`.

La función `MPI_Comm_rank` está declarada como sigue:

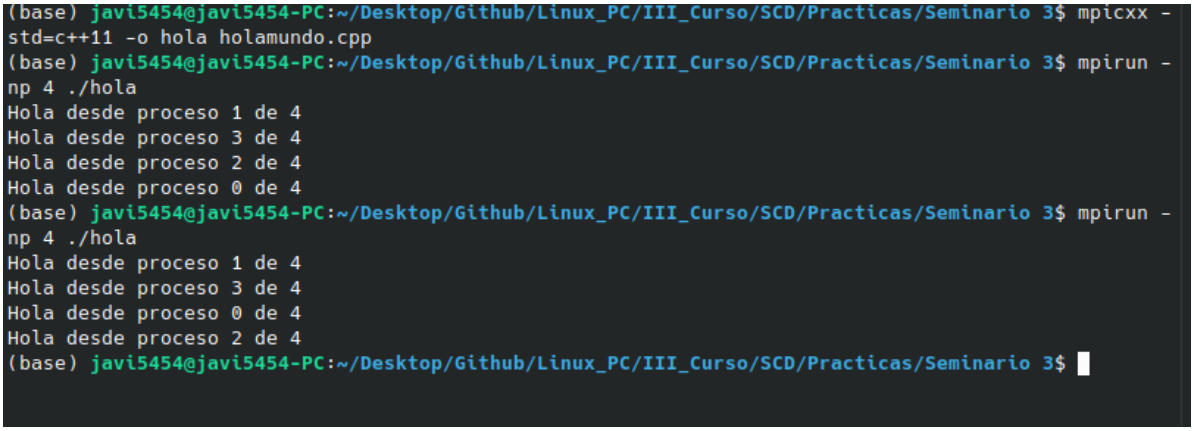
```
int MPI_Comm_rank (MPI_Comm comm, int *rank)
```

Escribe en `rank` el número de proceso que llama. De esta manera, podemos identificar cada proceso.

Veamos un ejemplo:

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

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

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

   cout << "Hola desde proceso " << id_propio << " de " << num_procesos_actual <<
endl ;

   MPI_Finalize();
   return 0;
}
```

y su compilación y ejecución es la siguiente:



Funciones básicas de envío y recepción de mensajes.

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` se transfiere junto con el mensaje y se usa para clasificar el mensaje.

Implementa envío **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 iniciado ya la recepción del mensaje (el envío es **asíncrono**).

Un proceso puede recibir un mensaje usando `MPI_Recv`:

```
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 **asíncrona**. 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.

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, la etiqueta.

Para obtener la cuenta de valores recibidos, usamos `status`:

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

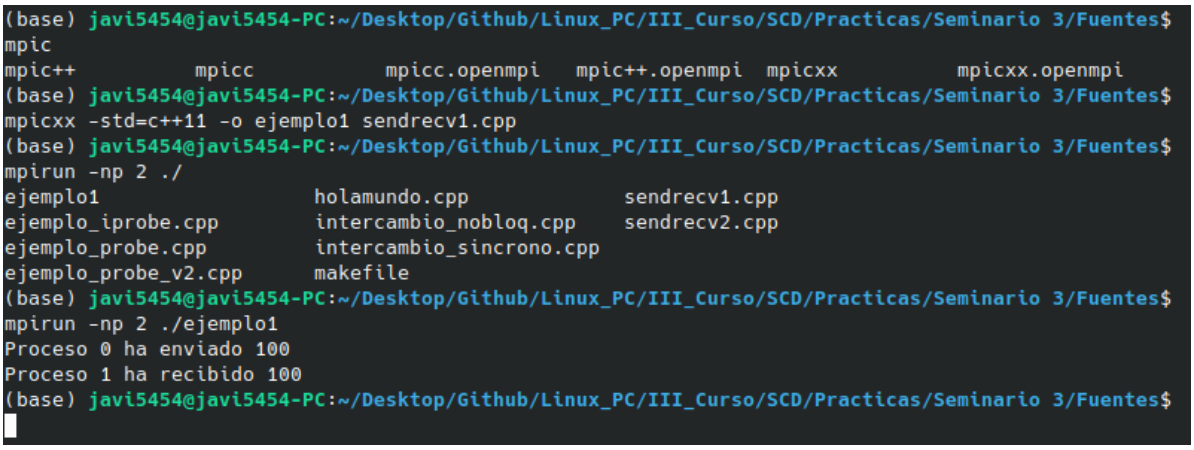
Escribe en `num` el número de items recibidos en una llamada `MPI_Recv` previa.

Veamos un ejemplo sencillo:

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

using namespace std; // incluye declaraciones de funciones, tipos y ctes. MPI

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,           // identificador de este proceso
            num_procesos_actual; // numero de procesos lanzados


 MPI_Init( &argc, &argv );                          // inicializa MPI
 MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );       // averiguar mi ident
 MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual ); // averiguar n.procs

 if ( num_procesos_esperado == num_procesos_actual ) // si num. procs. ok
 {
   // hacer envío o recepción (según id_propio)
   if ( id_propio == id_emisor ) // 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 << "Proceso " << id_propio << " ha enviado " << valor_enviado << endl
;
   }
   else // 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 << "Proceso " << id_propio << " ha recibido " << valor_recibido <<
endl ;
   }
 }
 else if ( id_propio == 0 ) // solo el primero escribe error, indep. del rol
      // escribir un mensaje de error
      cerr
```

```
         << "el número de procesos esperados es:    " << num_procesos_esperado <<
endl
         << "el número de procesos en ejecución es: " << num_procesos_actual <<
endl
         << "(programa abortado)" << endl ;


 MPI_Finalize( ); // terminar MPI: debe llamarse siempre por cada proceso.
 return 0;        // terminar proceso
}
```

cuyo resultado de ejecución es:



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 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 nombras explícitamente 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**.

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.
- Se ha reservado memoria suficiente para recibir todos los datos.

Veamos un ejemplo de un código de difusión en cadena:

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

using namespace std;

const int num_min_procesos = 2 ;
```

```
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_min_procesos <= num_procesos_actual  )
   {
      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
      {
         // recibir o leer valor
         if ( id_anterior < 0   ) // si soy el primero (no hay anterior)
         {
            cout << "Introduce un número aquí debajo (-1 para acabar): " << endl
;
            cin >> valor ;        //     pedir valor por teclado (-1 para acabar)
         }
         else                     // si no soy el primer proceso: recibirlo
            MPI_Recv( &valor, 1, MPI_INT, id_anterior, 0, MPI_COMM_WORLD, &estado
);

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

         // si no soy el último (si hay siguiente): enviar valor,
         if ( id_siguiente < num_procesos_actual )
            MPI_Send( &valor, 1, MPI_INT, id_siguiente, 0, MPI_COMM_WORLD );
      }
      while( valor >= 0 ); // acaba cuando se teclea un valor negativo
   }
   else
      cerr << "Se esperaban 2 procesos como mínimo, pero hay 1." << endl ;

   MPI_Finalize();
   return 0;
}
```

cuya ejecución es:

```
(base) javi5454@javi5454-PC:~/Desktop/Github/Linux_PC/III_Curso/SCD/Practicas/Seminario 3/Fuentes$
mpicxx -std=c++11 -o ejemplo2 sendrecv2.cpp
(base) javi5454@javi5454-PC:~/Desktop/Github/Linux_PC/III_Curso/SCD/Practicas/Seminario 3/Fuentes$
mpirun -np 4 ./ejemplo2
Introduce un número aquí debajo (-1 para acabar):
5
Proc.0: recibido/leído: 5
Introduce un número aquí debajo (-1 para acabar):
Proc.1: recibido/leído: 5
Proc.2: recibido/leído: 5
Proc.3: recibido/leído: 5
-1
Proc.0: recibido/leído: -1
Proc.1: recibido/leído: -1
Proc.2: recibido/leído: -1
Proc.3: recibido/leído: -1
(base) javi5454@javi5454-PC:~/Desktop/Github/Linux_PC/III_Curso/SCD/Practicas/Seminario 3/Fuentes$
```

Paso de mensajes síncrono en MPI

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`. Tras acabar, ya se ha iniciado en el receptor una operación de recepción que encaja con este envío (es **síncrono**) y los datos ya se han leido de `buf_emi` y se han copiado en otro lugar, luego es **seguro**.

Si la correspondiente operación de recepción usada es `MPI_Recv`, la semántica del paso de mensajes es puramente síncrona.

En el siguiente ejemplo hay un número par de procesos: estos se agrupan por parejas. Cada proceso enviará un dato a su correspondiente pareja o vecino. Los envíos se hace usando envío síncrono. Si todos los procesos hacen envío seguido de recepción (o al revés), **habría interbloqueo con seguridad**. Para evitarlo, los procesos pares hacen envío seguido de recepción y los procesos impares recepción seguida de envío.

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

      if ( id_propio % 2 == 0 ) // si proceso par: enviar y recibir
```

```
      {
         id_vecino = id_propio+1 ; // el vecino es siguiente
         MPI_Ssend( &valor_enviado,  1, MPI_INT, id_vecino, 0, MPI_COMM_WORLD );
         MPI_Recv ( &valor_recibido, 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_recibido, 1, MPI_INT, id_vecino, 0, MPI_COMM_WORLD,
&estado );
         MPI_Ssend( &valor_enviado,  1, MPI_INT, id_vecino, 0, MPI_COMM_WORLD );
      }
      cout  << "Proceso "<< id_propio <<" recibe "<< valor_recibido
            << " de "<< id_vecino << endl ;
   }
   else if ( id_propio == 0 ) // si n.procs. impar, el primero da error
      cerr << "Se esperaba un número par de procesos, pero hay "
           << num_procesos_actual << endl ;

   MPI_Finalize();
   return 0;
}
```

y su ejecución es:

```
(base) javi5454@javi5454-PC:~/Desktop/Github/Linux_PC/III_Curso/SCD/Practicas/Seminario 3/Fuentes$
mpicxx -std=c++11 -o ejemplo3 intercambio_sincrono.cpp
(base) javi5454@javi5454-PC:~/Desktop/Github/Linux_PC/III_Curso/SCD/Practicas/Seminario 3/Fuentes$
mpirun -np 4 ./ejemplo3
Proceso 2 recibe 12 de 3
Proceso 3 recibe 6 de 2
Proceso 1 recibe 0 de 0
Proceso 0 recibe 2 de 1
(base) javi5454@javi5454-PC:~/Desktop/Github/Linux_PC/III_Curso/SCD/Practicas/Seminario 3/Fuentes$
```

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 metadatos 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 de ser conocido).

Estas 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.

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 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 metadatos del mensaje, igual que se hace tras `MPI_Recv`.
- Si hay más de un mensaje disponible, los metadatos se refieren al primero que sen envió.

Veamoslo con un ejemplo:

```
void funcion_impresor( int num_emisores )
{
   MPI_Status estado ;
   int num_chars_rec ;
   // total mensajes: 2 por emisor, + 2 por cada una de sus iteraciones
   const int num_total_msgs = num_emisores*(2*num_iteraciones_por_emisor+2) ;

   cout << "inicio del impresor" << endl ;
   for( int i = 0 ; i < num_total_msgs; i++ )
   {
      // espera un mensaje de cualquier emisor, sin recibirlo
      MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &estado );

      // leer el numero de chars del mensaje
      MPI_Get_count( &estado, MPI_CHAR, &num_chars_rec );

      // reservar memoria dinámica para los caracteres (incluyendo 0 al final)
      char * buffer = new char[num_chars_rec+1] ;

      // recibir el mensake en el buffer y añadir un cero al final
      // IMPORTANTE: especificar exactamente mismo emisor detectado en el Probe
      MPI_Recv( buffer, num_chars_rec, MPI_CHAR, estado.MPI_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &estado );
      buffer[num_chars_rec] = 0 ;

      // imprimir  la cadena recibida
      cout << buffer << endl ;

      // liberar memoria dinámica ocupada por el buffer
      delete [] buffer ;
   }
   cout << "fin del impresor" << endl ;
}
```

Aquí, el receptor reserva la memoria necesaria para el mensaje que se va a recibir.

Ahora, veamos una consulta no bloqueante con `MPI_Iprobe`

```
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. Si no hay mensajes, dicho entero es 0. Los parámetros (excepto `flag`) se interpretan igual que en `MPI_Probe`.

Veamos un ejemplo de aplicación:

```
void funcion_receptor( int id_min, int id_max )
{
   int num_emisores = id_max - id_min +1 ;  // total de emisores
   int total_mensajes = num_mensajes_por_emisor*num_emisores ; // total msgs
   int cuenta[num_emisores]; // número de mensajes recibidos por emisor

   // inicializar la cuenta:
   for( unsigned i = 0 ; i < num_emisores ; i++ )
      cuenta[i] = 0 ;

   cout << "inicio del receptor" << endl ;

   for( int i = 0 ; i < total_mensajes ; i++ )
   {
      MPI_Status estado ;     // estado de la recepción
      int        hay_mensaje, // ==0 si no hay mensajes, >0 si hay mensajes
                 id_emisor ,  // identificador de emisor a recibir/recibido
                 valor ;      // valor recibido

      // comprobar si hay mensajes, en orden creciente de los posibles emisores
      // (id_emisor toma valores entre id_min e id_max, ambos incluidos)
      for( id_emisor = id_min ; id_emisor <= id_max ; id_emisor++ )
      {
         // recibir mensaje del emisor
         MPI_Iprobe( id_emisor, MPI_ANY_TAG, MPI_COMM_WORLD, &hay_mensaje,
&estado ) ;
         if ( hay_mensaje ) break ;
      }

      // comprobar si hay mensajes pendientes o no
      if ( hay_mensaje )              // si no hay mensaje:
         cout << "Hay al menos un mensaje, del emisor: " << id_emisor << endl ;
      else
      {                              // si no hay mensaje
         id_emisor = MPI_ANY_SOURCE ;  //   aceptar de cualquiera
         cout << "No hay mensajes. Recibo de cualquiera." << endl ;
      }

      // recibir el mensaje e informar
      MPI_Recv( &valor, 1,MPI_INT, id_emisor, 0, MPI_COMM_WORLD, &estado );
      id_emisor = estado.MPI_SOURCE ;
      cout << "Receptor ha recibido de " << id_emisor << endl ;

      // incrementar cuenta e imprimir estadísticas
      cuenta[id_emisor-id_min] ++ ;
      cout << "Núm. de mensajes recibidos de cada emisor:  " ;
      for( unsigned i = 0 ; i < num_emisores ; i++ )
         cout << cuenta[i] << ", " ;
      cout << endl << endl ;
```

```
      // dormir un poco
      sleep_for( milliseconds( aleatorio<8,12>() ) );
   }
}
```

Aquí, el receptor espera a que haya mensajes provenientes de procesos con id aleatorios. Si hay, recibe el del proceso con id más bajo. Si no, recibe de cualquier emisor.

Comunicación insegura

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.

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

- `MPI_Irecv`: inicia recepción pero retorna antes de recibir.

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

En ambos casos, los argumentos son similares a `MPI_Send`, excepto que `request` es un **ticket** que permitirá después identificar a la operación cuyo estado se pretende consultar o se espera a que finalice y la recepción no incluye argumento `status`.

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

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

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.

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

- `MPI_Test`: comprueba si el envío o recepción ha finalizado o no. No es una espera bloqueante.

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

comprueba la operación identificada por un `request` y escribe en `flag` el número > 0 si ha acabado, o bien 0 si no ha acabado.

En ambas funciones, una vez terminada la operación referenciada por el ticket, podemos usar el objeto `status` para consultar los metadatos y liberar la memoria usada por `request`.

Las operaciones insegurar permiten simultanear trabajo útil en el emisor y/o receptor con lectura, transmisión y recepción del mensaje, aumentando el paralelismo potencial y por tanto pueden mejorar la eficiencia en tiempo.

Veamos un ejemplo donde evitamos el interbloqueo asociado al intercambio síncrono:

```
#include <mpi.h>
#include <iostream>
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 )
   {
      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 ;

      // las siguientes dos llamadas pueden aparecer en cualquier orden
      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 );

      MPI_Wait( &ticket_envio, &estado );
      MPI_Wait( &ticket_recepcion, &estado );

      cout<< "Soy el proceso " << id_propio << " y he recibido el valor " <<
valor_recibido << " del proceso " << id_vecino  << endl ;
   }
   else if ( id_propio == 0 ) // si n.procs. impar, el primero da error
      cerr << "Se esperaba un número par de procesos, pero hay "
           << num_procesos_actual << endl ;
```

```
    MPI_Finalize();
    return 0;
}
```

cuyo resultado de ejecución es: