DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS

1. Motivación y aspectos de la programación paralela.
2. Tipos de sistemas paralelos. Paradigmas de programación paralela.
3. Conceptos básicos y medidas de paralelismo.
4. Diseño de programas paralelos.
5. La Interface de paso de mensaje: el estándar MPI.
   - Programación mediante paso de mensajes.
   - Introducción al estándar MPI (Message Passing Interface).
     - Evolución y origen.
     - ¿Qué es MPI?
     - Funciones básicas.
       - Inicialización.
       - Finalización.
       - Información.
       - Comunicación.
     - Comunicaciones colectivas.
     - Problemas de interbloqueo.
     - Mediciones de tiempo.
     - Caso de estudio. Cálculo de pi.
6. Paralelización de algoritmos: ejemplos y aplicaciones.

Evolución y origen

- Generaciones previas de computadores paralelos comerciales se han basado en la arquitectura de memoria distribuida, con lo que el paradigma natural es el modelo de paso de mensajes.
- Esto dio lugar al desarrollo de diferentes entornos de trabajo para paso de mensajes.
- Cada fabricante proporcionaba sus propias librerías con buenos rendimientos en sus máquinas pero incompatibles con las proporcionadas por otros fabricantes: con diferencias no sólo sintácticas sino también semánticas:
  - NX: librería propietario de Intel.
  - MPL: librería de paso de mensajes de IBM.
- Desarrollo de paquetes de dominio público: PICL, PARMACS, P4, PVM, …
- PVM: quizá la primera librería ampliamente adoptada y aceptada como librería de paso de mensajes.
- Esfuerzo por crear un estándar para la programación de paso de mensajes: MPI.

Evolución y origen

- La primera versión del estándar MPI se finalizó y publicó en 1994, después de dos años de reuniones y discusiones.
- En 1995 se publicó una nueva especificación del estándar en la que se modificaron aspectos de orden menor de la primera versión.
- En 1997 se presentó la especificación MPI-2 que extiende MPI sin introducir cambios en la versión 1:
  - Se introdujeron características para incrementar la conveniencia y robustez de MPI, como operaciones en memoria remota, entrada/salida paralela, gestión dinámica de procesos, etc.

- Todo ello ha permitido que MPI sea la primera librería estándar portable con buenos rendimientos.

¿Qué es MPI?

- Message Passing Interface: Librería de funciones para el paso de mensajes entre procesadores.
- MPI NO ES un lenguaje!
- Puede ser utilizado en computadores paralelos o redes de computadores personales/estaciones de trabajo,…
- Aplicable desde Fortran o C.
- Todos los procesadores reciben la misma copia del programa a ejecutar.
- Requiere sus propios comandos de compilación/ejecución.

¿Qué es MPI?

MPI ofrece:

- **Estandarización** a muchos niveles, reemplazando virtualmente a todas las implementaciones de paso de mensajes utilizadas para producción.
- **Portabilidad** a los sistemas existentes y a los nuevos. La mayoría de las plataformas (si no todas) ofrecen, al menos, una implementación de MPI.
- **Rendimiento** comparable a las librerías propietarias de los vendedores. Incluso para arquitecturas de memoria compartida, las implementaciones MPI podrían no hacer uso de la red de interconexión, sino de la memoria compartida.
- **Riqueza.** Posee una extensa funcionalidad y muchas implementaciones de calidad.

Funciones básicas

- MPI incluye más de 125 funciones.
- Sin embargo, se puede efectuar trabajo productivo en paralelo con sólo 6:
  - 2 de inicio y finalización del programa.
  - 2 de control del número de procesos.
  - 2 de comunicación.

Funciones básicas

Variables MPI predefinidas

- MPI predefine una serie de variables y de estructuras de datos inherentes a su funcionamiento.
- Se encuentran en un archivo de encabezado que debe ser incluido en todo código que use MPI.

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

Funciones básicas

Sintaxis:

`MPI_Funcion(…)`

- Los parámetros de las funciones MPI son de tres tipos:

IN:	la función **lee** el argumento
OUT:	la función **modifica** el argumento
IN/OUT:	la función **lee y modifica** el argumento

Funciones básicas

Sintaxis: MPI_Funcion(…)

- Todas las rutinas MPI utilizan el prefijo **MPI_** . En las rutinas de C, a continuación del prefijo se coloca el nombre de la rutina con la primera letra en mayúscula.
- Las funciones MPI (todas excepto MPI_Wtime y MPI_Wtick) **devuelven un entero** como **código de error**.
- error = MPI_Funcion(...)
- Por defecto, el manipulador de errores de MPI aborta un trabajo cuando se produce un error.
- Si no ha ocurrido ningún error, se devuelve **MPI_SUCCESS** (0 en esta implementación).
- En caso de errores, el valor que indica el tipo de error depende de la implementación.

Funciones básicas

- Necesitamos:
  - un método para crear procesos: estático / dinámico.
  - un método para enviar y recibir mensajes, punto a punto y de manera global.
- El objetivo de MPI es explicitar la comunicación entre procesos, es decir:
  - el movimiento de datos entre procesadores.
  - la sincronización de procesos.

Funciones básicas

- El modelo de paralelismo que implementa MPI es **SPMD** (Single Program Multiple Data).

```
if (pid==1)             ENVIAR_a_pid2
else if (pid==2)            RECIBIR_de_pid1
```

- Recordamos que cada proceso dispone de su propio espacio de direcciones.

- También se puede trabajar con un modelo **MPMD** (Multiple Program Multiple Data): se ejecutan programas diferentes en los nodos.

Funciones básicas

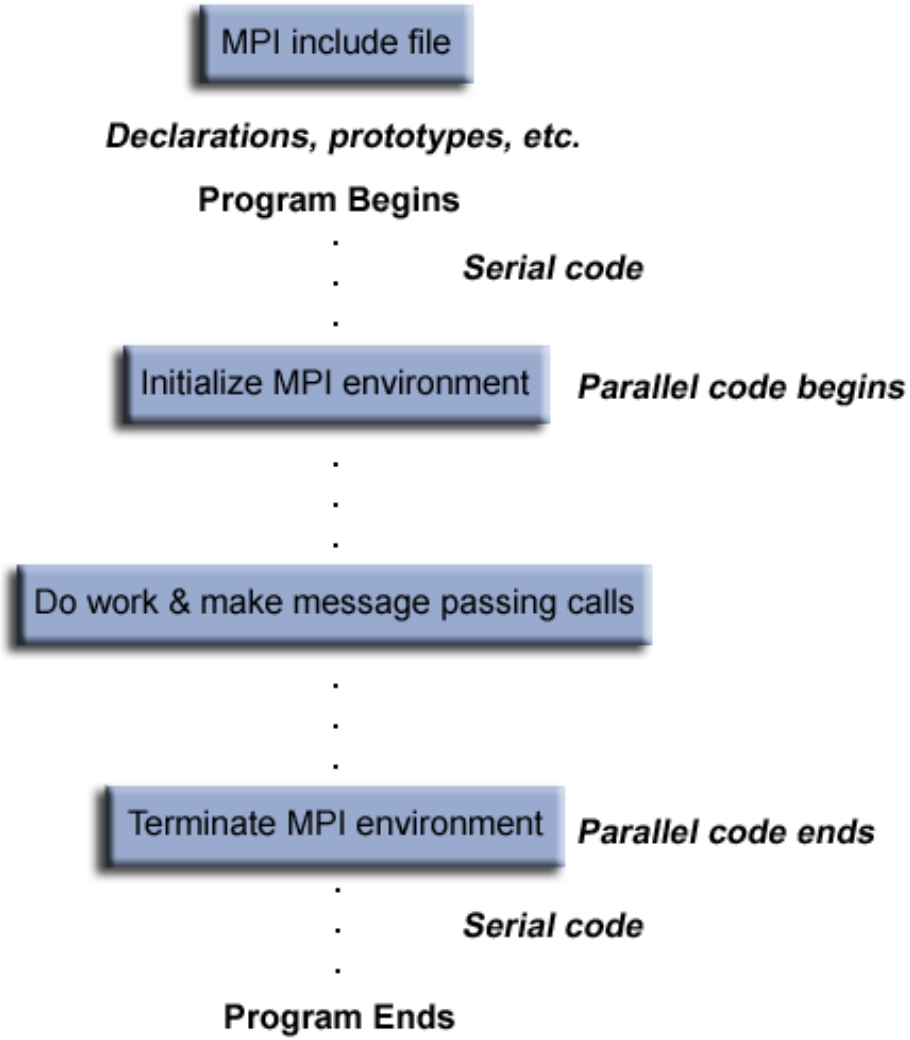
- En todo caso, hay que tener en cuenta que la **eficiencia en la comunicación** va a ser determinante en el **rendimiento del sistema** paralelo, sobre todo en aquellas aplicaciones en las que la comunicación juega un papel importante (paralelismo de grano medio / fino).

- Además de implementaciones específicas, existen dos implementaciones libres de uso muy extendido: LAM/ MPI (OpenMPI) y MPICH.

- Nosotros vamos a usar **MPICH**

Funciones básicas

Estructura general de un programa MPI



Funciones básicas

Comienzo y **final**:

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

> MPI_Finalize();
```

Estas dos funciones son la **primera** y **última** función MPI que deben ejecutarse en un programa.

No se pueden utilizar funciones MPI antes de **`_Init`**. No se puede invocar a otras rutinas MPI después de esta llamada **`_Finalize`**. Si un proceso no ejecuta **`_Finalize`** el programa queda como “colgado”.

Funciones básicas: Inicialización

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

- int **argc:** Puntero al número de argumentos.
- char **argv:** Puntero al vector de argumentos.

- Esta función inicializa varias estructuras de datos inherentes al ambiente de trabajo MPI: **inicializa todas las estructuras de datos necesarias para permitir la comunicación** entre procesos basados en el envío de mensajes MPI. Reserva el canal de comunicación, asigna un valor a la constante MPI_COMM_WORLD.
- Si el ambiente no se puede inicializar, el programa se detiene por completo.
- Esta rutina, debe ser llamada **antes** que cualquier otra rutina de MPI. Debe ser llamada una única vez, en caso de no hacerlo así sería erroneo. Los argumentos que recibe no son ni modificados, ni interpretados ni distribuidos.
- El estándar MPI no indica nada sobre lo que un programa puede hacer antes de la inicialización de MPI. Sin embargo, en la implementación MPICH se debe **hacer lo menos posible** (evitar sobretodo: abrir archivos, lectura y escritura).
- Errores:
  - MPI_SUCCESS: No ha habido error, la función MPI se ha realizado satisfactoriamente.
  - MPI_ERR_OTHER: Significa que se ha llamado más de una vez a la función MPI_Init.

Funciones básicas: Inicialización

- Todas las funciones MPI (excepto MPI_Wtime y MPI_Wtick) devuelven un valor de error.
- Por defecto, el manipulador de errores de MPI aborta un trabajo cuando se produce un error.
- Si no ha ocurrido ningún error, se devuelve **MPI_SUCCESS**.

Funciones básicas: Finalización

```
int MPI_Finalize( void )
```

- Cierra el ambiente de trabajo en paralelo una vez finalizado el trabajo: **Corta la comunicación entre los procesos y elimina los tipos de datos** creados para ello.
- Todos los procesos deben de llamarla antes de salir.
- No se puede invocar a otras funciones MPI después de esta llamada.
- El número de procesos que se ejecutan después de que se llama a esta rutina es indefinido. Lo mejor es que sea la **última sentencia del programa** y no realizar nada más después de llamar a MPI_Finalize. En caso contrario, se debería elegir un único proceso para que continúe la ejecución del programa. MPI no asegura lo que hacen las hebras tras el MPI_Finalize con lo que lo más seguro es encerrar el resto del código tras la ejecución con una condición que asegure que sólo un proceso lo ejecute. Es altamente recomendable proteger la zona de código que sigue al MPI_Finalize. Una forma puede ser como sigue.

Funciones básicas: Finalización

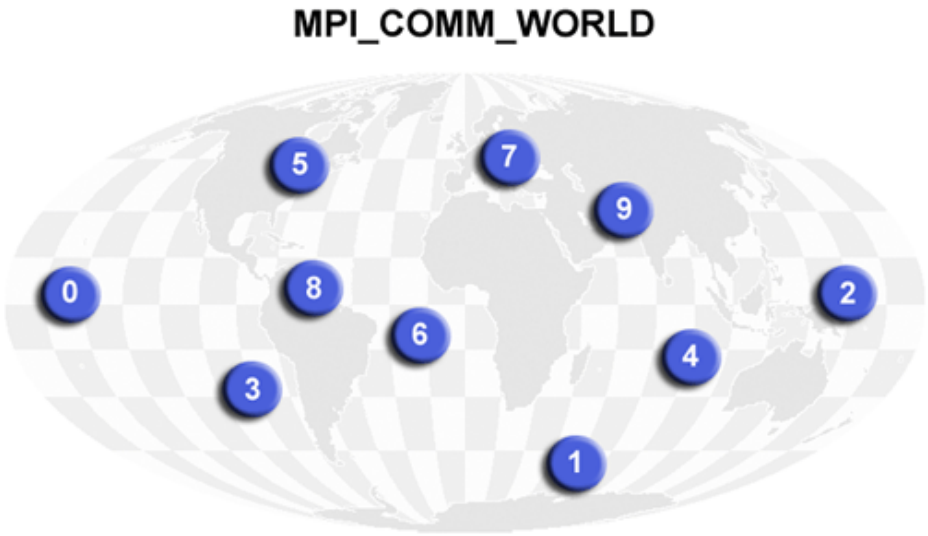
```
int MPI_Finalize( void )
```

```
int main(int argc, char* argv[ ])
{
    /* Declaracion de variables */
    MPI_Init(&argc, &argv);
    /* Reparto de trabajo */
    /* Bucle principal del programa */
    // Finalizamos el entorno MPI
    MPI_Finalize();
     /* Proteger la zona de código que sigue al MPI_Finalize: elegimos
un único proceso para que continúe la ejecución del programa */
    if(/*es proceso elegido*/){
        /* Codigo que ejecuta unicamente el proceso elegido */
      }
}
```

Funciones básicas: Información

- Los procesos que se van a ejecutar se agrupan en conjuntos denominados **comunicadores**.
- Cada proceso tiene un **identificador** o pid en cada comunicador.

- El comunicador **MPI_COMM_WORLD** (un objeto de tipo MPI_COMM) se crea por defecto y engloba a todos los procesos.



Funciones básicas: Información

Identificación de procesos

> MPI_Comm_rank(comm, &pid);

Devuelve en **pid** (int) el **identificador del proceso** dentro del grupo de procesos, comunicador **comm**, especificado.

Recuerda que un proceso se identifica mediante dos parámetros: identificador (**pid**) y grupo (**comm**).

> MPI_Comm_size(comm, &npr);

Devuelve en **npr** (int) el **número de procesos** del comunicador especificado.

Funciones básicas: Información

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

- MPI_Comm **comm:** variable de tipo MPI_Comm suministrado por el usuario indicando el comunicador utilizado. Por defecto, es MPI_COMM_WORLD que no hace falta declarar.
- int **size:** variable de tipo entero devuelto por la función indicando el número de procesos asignados al sistema, i.e., el número de procesos en el grupo de comm.
- **Nota sobre MPI_Comm:** Este tipo de dato guarda toda la información relevante sobre un comunicador específico. Se utiliza para especificar el comunicador por el que se desea realizar las operaciónes de transmisión o recepción (como MPI_Send o MPI_Recv).

Determina el **número total de procesos** existentes dentro de un mismo comunicador. El usuario puede definir otros comunicadores para designar subconjuntos de procesos.

Funciones básicas: Información

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

- MPI_Comm **comm:** variable de tipo MPI_Comm suministrado por el usuario indicando el comunicador utilizado. Por defecto, es MPI_COMM_WORLD.
- int **rank:** variable de tipo entero devuelto por la función indicando el número lógico que corresponde a cada proceso.

Devuelve el **número lógico** que corresponde a cada **proceso**. Este valor siempre empieza en cero y alcanza un valor máximo igual al número de procesos menos uno.

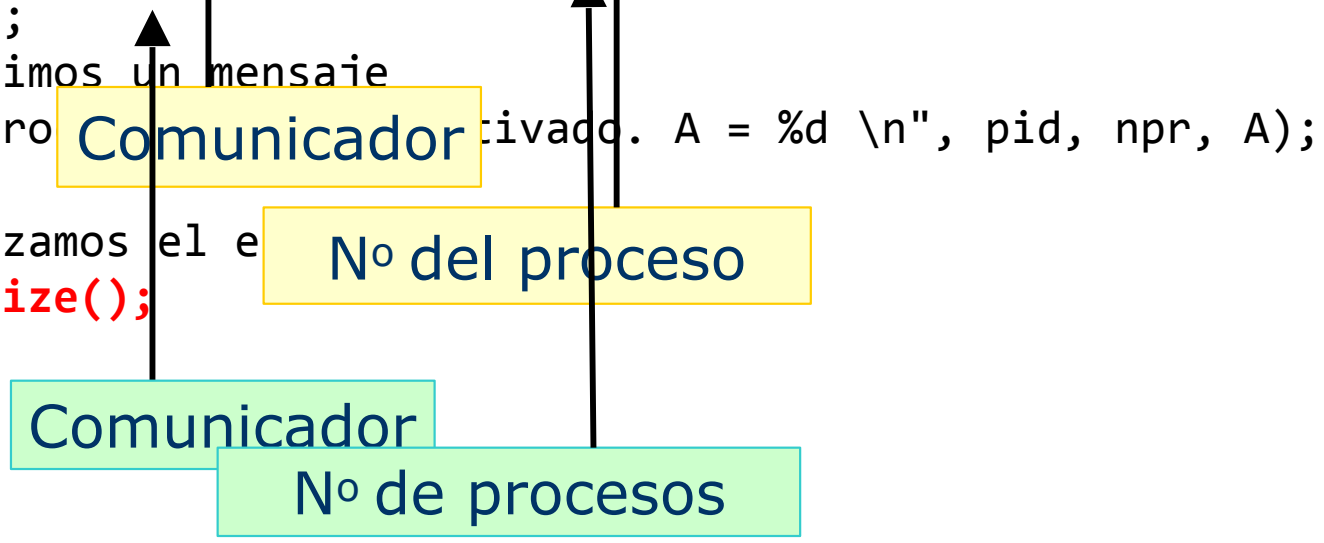
Funciones básicas: Información

ejemplo0.c

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

int main (int argc, char **argv)
{
    int pid, npr, A;
    // Inicializamos el entorno MPI
    MPI_Init(&argc, &argv);
    A = 2;
    // Obtenemos el rango de los procesos
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    // Obtenemos el número de procesos
    MPI_Comm_size(MPI_COMM_WORLD, &npr);
    A = A + 1;
    // Imprimimos un mensaje
    printf("Pro                    tivado. A = %d \n", pid, npr, A);

    // Finalizamos el e
    MPI_Finalize();
}
```



Salida del Ejemplo 0

- Compilación: `mpicc -o ejemplo0 ejemplo0.c`
- Después de la ejecución (`mpirun -np 4 ejemplo0`), la salida que produce el *ejemplo0* es:

```
Proceso 0 de 4 activado. A = 3
Proceso 1 de 4 activado. A = 3
Proceso 2 de 4 activado. A = 3
Proceso 3 de 4 activado. A = 3
```

Funciones básicas: Información

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

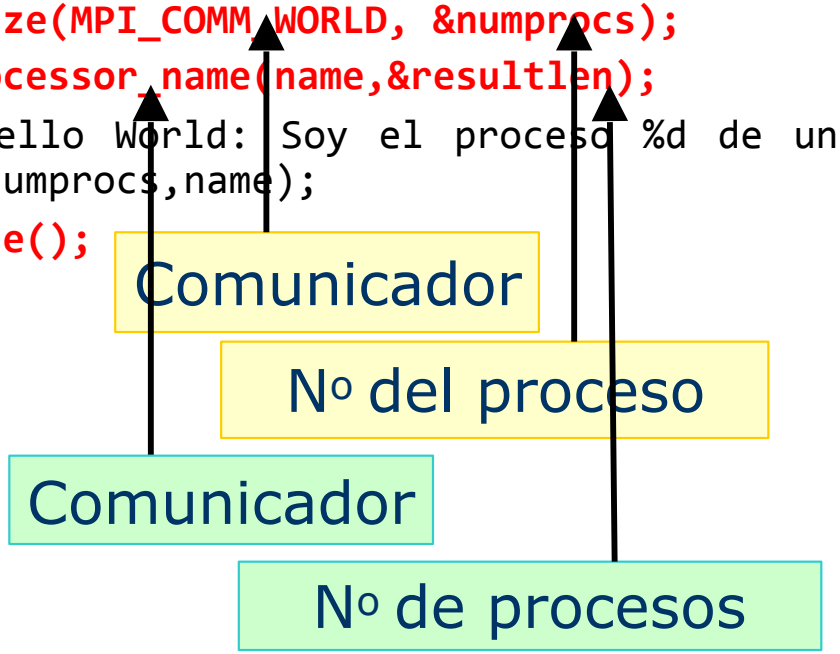
- char **name:** **variable de tipo carácter** devuelto por la función que indica el **nombre del nodo** en el que se está ejecutando la tarea. Debe ser un array de tamaño al menos
- `MPI_MAX_PROCESSOR_NAME`.
- int **resultlen:** **variable de tipo entero** devuelto por la función indicando **la longitud (en caracteres) de name**.

Obtiene el **nombre** del nodo en el que se está ejecutando la tarea que hace la llamada.

Funciones básicas: Información

ejemplo1.c

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv)
{
   int myrank, numprocs, resultlen;
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Get_processor_name(name,&resultlen);
     printf("Hello World: Soy el proceso %d de un total de %d. Estoy en el nodo
%s\n",myrank,numprocs,name);
   MPI_Finalize();
}
```



Salida del Ejemplo 1

- Compilación: `mpicc -o ejemplo1 ejemplo1.c`
- Después de la ejecución (`mpirun -np 3 ejemplo1`), la salida que produce el *ejemplo1* es:

```
Hello World: Soy el proceso  0 de un total de  3 procesos. Estoy en el nodo cluster1
Hello World: Soy el proceso  2 de un total de  3 procesos. Estoy en el nodo cluster3
Hello World: Soy el proceso  1 de un total de  3 procesos. Estoy en el nodo cluster2
```

- Notar que los procesos se ejecutan asíncronamente.

Compilación y ejecución

- Para la compilación se usan los scripts:

  mpif77, mpif90, mpicc

- De esta forma, la compilación del ejemplo 1 sería:

  mpicc –o ejemplo1 ejemplo1.c

- La ejecución de trabajos mpi se realiza con el script:

  mpirun, mpiexec

- La ejecución del ejemplo 1 sería:

  mpirun –np 3 ejemplo1

Le indicamos el número de procesos a utilizar: el número de copias del ejecutable, *ejemplo1*

Funciones básicas: Comunicación

- MPI ofrece dos (tres) tipos de comunicación:
  - **punto a punto**, del proceso i al j (participan ambos).
  - **en grupo** (colectiva): entre un grupo de procesos, de uno a todos, de todos a uno, o de todos a todos.
  - one-sided: del proceso i al j (participa uno solo).

- Además, básicamente en el caso de comunicación entre dos procesos, hay múltiples variantes en función de cómo se implementa el proceso de envío y de espera.

Funciones básicas: Comunicación

- La comunicación entre procesos requiere (al menos) de dos participantes: emisor y receptor.

- El emisor ejecuta una función de **envío** y el receptor otra de **recepción**.

- La comunicación es un proceso **cooperativo**: si una de las dos funciones no se ejecuta, no se produce la comunicación (y podría generarse un **deadlock**).

Funciones básicas: Comunicación

Para enviar o recibir un mensaje es necesario especificar:

- **A quién** se envía (o **de quién** se recibe):
  - los **datos** a enviar (dirección de **comienzo** y **cantidad**).
  - el **tipo** de los datos.
  - la **clase** de mensaje (tag).

Todo lo que no son los datos forma el "sobre" del mensaje.

- Las dos funciones estándar para enviar y recibir mensajes son:
  - `MPI_Send`
  - `MPI_Recv`

Funciones básicas:

Comunicación (Blocking Standard Send)

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

- **buf:** Variable que contiene la información a comunicar.
- int **count:** Cantidad de elementos contenidos en buf.
- MPI_Datatype **datatype:** Tipo de la variable buf.
- int **dest:** Número lógico del proceso al cual se ha transferido información.
- int **tag:** Identifica el envío. Generalmente es cero y sólo cambia cuando se ha de comunicar más de un envío.
- MPI_Comm **comm:** Comunicador.

Un send/receive bloqueante suspende la ejecución del programa hasta que el buffer que se está enviando/recibiendo es seguro de usar. En el caso de un send bloqueante, esto significa que los datos que tienen que ser enviados han sido **copiados en el buffer de envío** (no necesariamente han sido recibidos por la tarea que recibe).

Funciones básicas: Comunicación (Blocking Standard Send)

int MPI_Send(*buf, count, datatype, dest, tag, comm)

- **buf:** Variable que contiene la información a comunicar.
- int **count:** C
- MPI_Dataty
- int **dest:** Nú
- int **tag:** Ide
  se ha de cor
- MPI_Comm

Un send/rece
hasta que el
usar. En el ca
que tienen qu
(no necesaria

MPI	C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long_double

Funciones básicas:

Comunicación (Blocking Standard Receive)

```
int MPI_Recv( *buf, count, datatype, source, tag, comm, *status)
```

- **buf:** Variable que contiene la información a comunicar.
- int **count:** Cantidad de elementos contenidos en buf.
- MPI_Datatype **datatype:** Tipo de la variable buf.
- int **source:** Número lógico del proceso **desde el cual se espera recibir** información. MPI permite no especificar explícitamente el proceso desde el que se espera recibir; en este caso se indicará *source* como **MPI_ANY_SOURCE**.
- int **tag:** Identifica el envío. Generalmente es cero y sólo cambia cuando se ha de comunicar más de un envío. **MPI_ANY_TAG** permite aceptar mensajes con **cualquier identificador**.
- MPI_Comm **comm:** Comunicador.
- MPI_Status **status:** Estructura interna que contiene los campos MPI_SOURCE, MPI_TAG y count.

`Recv` se **bloquea** hasta que se efectúa la recepción.

Funciones básicas:

Comunicación (Blocking Standard Receive)

```
int MPI_Recv( *buf, count, datatype, source, tag, comm, *status)
```

- **buf:** Vari
- int **coun**
- MPI_Dat
- int **sour** informaci que se MPI_ANY
- int **tag:** l ha de c mensajes
- MPI_Co
- MPI_Status **status:** Estructura interna que contiene los campos MPI_SOURCE, MPI_TAG y count.

Sobre el tipo de dato MPI_Status

Se utiliza para guardar **información** sobre operaciones de **recepción** de mensajes (como por ejemplo MPI_Recv). Este tipo de dato guarda una **estructura interna** con los siguientes **campos**:

- **MPI_SOURCE**: Indica el rango del proceso **origen** en el comunicador en el que se realizó la transmisión (int).
- **MPI_TAG**: Valor de la **etiqueta** que tenía el mensaje (int).

Funciones básicas:

Comunicación (Blocking Standard Receive)

El argumento status en MPI_RECV

Para una operación de recepción, este argumento se utiliza para guardar información referente a la operación de recepción de mensaje realizada: entre esta información está la fuente del mensaje y la etiqueta del mensaje.

- En C, este argumento es un puntero a una estructura predefinida MPI_Status. Por ejemplo:
  - status.MPI_SOURCE
  - status.MPI_TAG.
- Además, el número real de bytes recibidos se puede obtener desde status mediante la rutina MPI_Get_count.

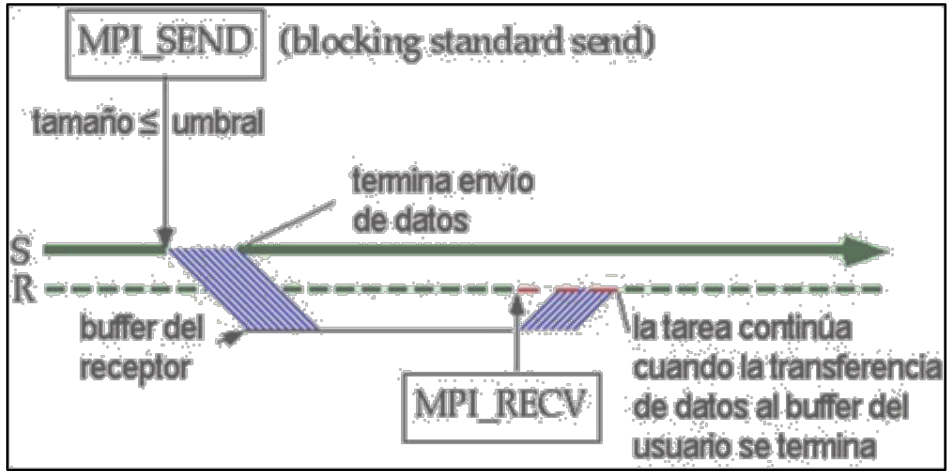
Declaración de la variable status:

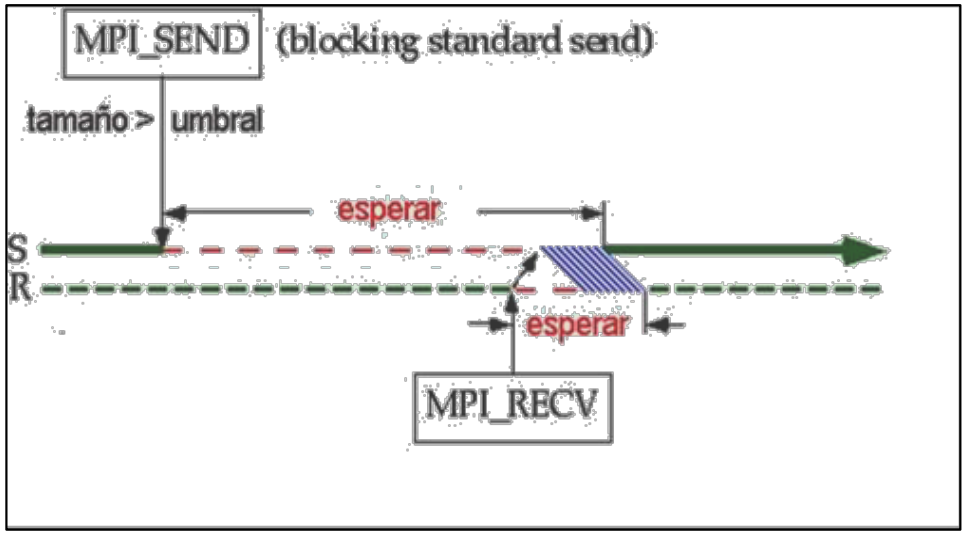
MPI_Status status;

Funciones básicas:

Comunicación (Blocking Standard Send/Recv)

- En el blocking standard send (MPI_Send (...)) se **copia el mensaje** sobre la red **en el buffer** del sistema del nodo que recibe, entonces **la tarea que ejecuta el send continúa con su ejecución**.
- El buffer del sistema se crea cuando comienza el programa (el usuario no necesita utilizarlo de ninguna manera). Hay un buffer del sistema por cada tarea que se encargará de manejar múltiples mensajes. El mensaje será copiado del buffer del sistema a la tarea que invoca el receive cuando se ejecute el receive.
- En ocasiones el buffer proporcionado por el sistema puede ser insuficiente pudiendo ocasionar problemas de interbloqueo.





ejemplo2.c

```
   Int main(int argc, char **argv) {
   int myrank, numprocs, i;
   MPI_Status estado;
   float dato = 7.0, res = 0.0;
MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   printf("Soy el proceso %d de un total de %d.\n",myrank,numprocs);
   if (myrank > 0)
 {
         res = dato * myrank;
         MPI_Send(&res,1, MPI_FLOAT, 0, 8, MPI_COMM_WORLD); }
   else {
          printf("Antes de recibir, el valor de res es %f\n",res);
          for ( i=1 ; i<numprocs ; i++ ) {
           MPI_Recv(&res, 1, MPI_FLOAT, i, 8, MPI_COMM_WORLD, &estado);
                      printf("Despues de recibir de %d, el valor de res es %f\n",
          i, res);
           printf("estado.MPI_SOURCE = %d, estado.MPI_TAG = %d \n",
           estado.MPI_SOURCE, estado.MPI_TAG);
        }
  }
MPI_Finalize();
}
```

destino

tamaño

En este caso podríamos haber utilizado MPI_ANY_SOURCE

tag

Salida del Ejemplo 2

- Después de la ejecución (mpirun –np 3 ejemplo2), la salida que produce el *ejemplo2* es:

```
Soy el proceso 0 de un total de 3.
Antes de recibir, el valor de res es 0.000000

Despues de recibir de 1, el valor de res es 7.000000
estado.MPI_SOURCE = 1, estado.MPI_TAG = 8

Despues de recibir de 2, el valor de res es 14.000000
estado.MPI_SOURCE = 2, estado.MPI_TAG = 8

Soy el proceso 1 de un total de 3.
Soy el proceso 2 de un total de 3.
```