

Universidad de Los Andes  
Facultad de Ingeniería  
Escuela de Sistemas

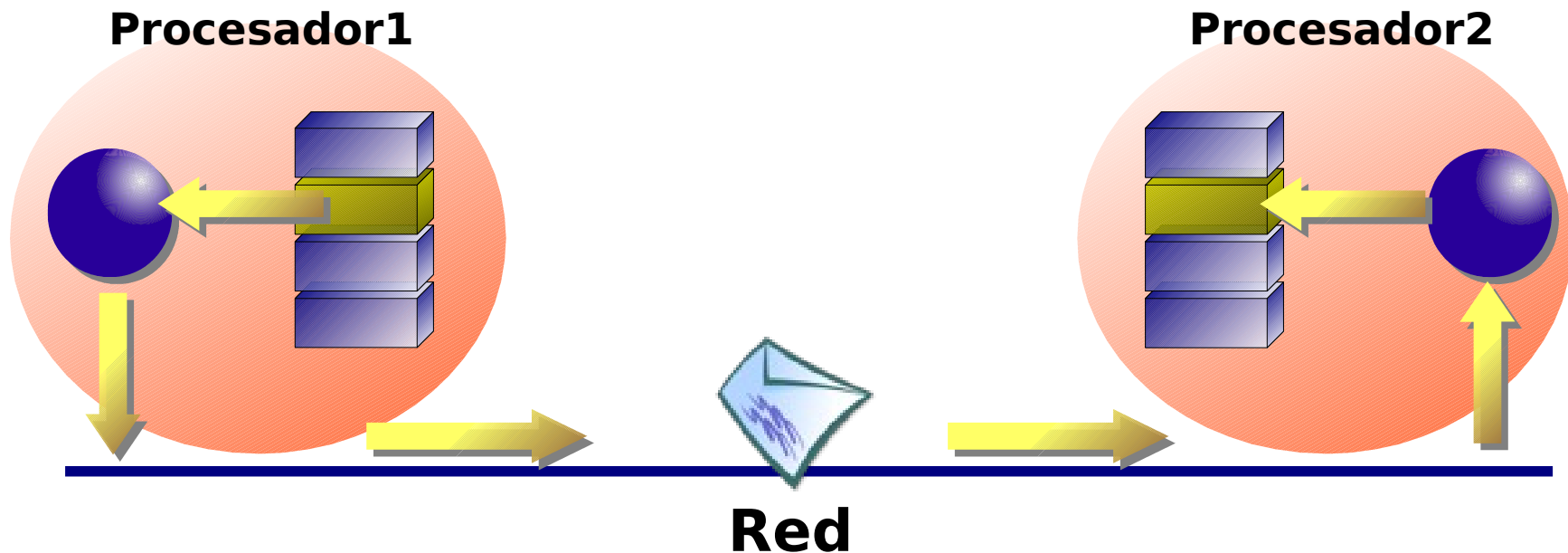
# Message Passing Interface MPI

**Prof. Gilberto Díaz**  
**gilberto@ula.ve**

*Departamento de Computación, Escuela de Sistemas, Facultad de Ingeniería  
Universidad de Los Andes, Mérida 5101 Venezuela*

**MPI** es un estándar para la implementación de sistemas de pase de mensajes diseñado por un grupo de investigadores de la industria y la academia para funcionar en una amplia variedad de computadores paralelos y de forma tal que los códigos sean portables.

Su diseño esta inspirado en máquinas con una arquitectura de **memoria distribuida** en donde cada procesador es propietario de cierta memoria y la única forma de intercambiar información es a través de mensajes.



Sin embargo, hoy en día también encontramos implementaciones de MPI en máquinas de memoria compartida.

El estándar define la sintaxis y semántica de un conjunto de rutinas útiles

Dentro de las implantaciones más populares de MPI se encuentran:

- MPICH
- LAM
- OpenMPI

Trabajaremos con OpenMPI pues es auspiciada por un grupo importante de instituciones



Los principales objetivos de MPI:

- Proporcionar código portátil
- Proporcionar implantaciones eficientes
- Soportar arquitecturas paralelas heterogeneas

El proceso de instalación es bastante sencillo. Siga los siguientes pasos para llevar a cabo este proceso:

- Descargue los fuentes

```
cd /usr/local/src/  
wget http://www.open-mpi.org/software/ompi/v1.3/  
downloads/ openmpi-1.3.2.tar.bz2
```

- Desempaquete los fuentes

```
tar xvjf openmpi-1.3.2.tar.bz2
```



El proceso de instalación es bastante sencillo. Siga los siguientes pasos para llevar a cabo este proceso:

- Configure el software

```
cd openmpi-1.3.2
```

```
./configure --prefix=/opt/openmpi/gnu
```

- Compile el software e instalelo

```
make && make install
```

Para poder ejecutar un programa paralelo es necesario configurar el ambiente apropiado:

- Un sistema de archivos común donde se encuentren todos los ficheros involucrados en la ejecución
- Un sistema de autenticación de usuarios que permita el acceso sin contraseña a los nodos participantes

## Las implantaciones de MPI son bibliotecas

- Todas las operaciones son ejecutadas realizando invocaciones a funciones
- Debemos incluir los siguientes archivos cabecera (headers)
  - `mpi.h` en C
  - `mpif.h` para Fortran77 y Fortran90

Podemos categorizar las funciones de MPI de la siguiente manera:

- Funciones de gestión
  - Inicializar y terminar el ambiente paralelo
- Funciones para la comunicación entre pares de procesos (comunicación punto a punto)
- Funciones para la comunicación entre grupos de procesos (comunicación colectiva)
- Funciones para crear tipos de datos

- Inicializar el ambiente de ejecución paralelo

`MPI_Init(&argc, &argv)`

- Identificación dentro del ambiente paralelo

`MPI_Comm_rank(MPI_COMM_WORLD, &my_id)`

- Obtener el número de procesos que realmente se pudo iniciar

`MPI_Comm_size(MPI_COMM_WORLD, &nproc)`

- Finalizar el ambiente de ejecución paralelo

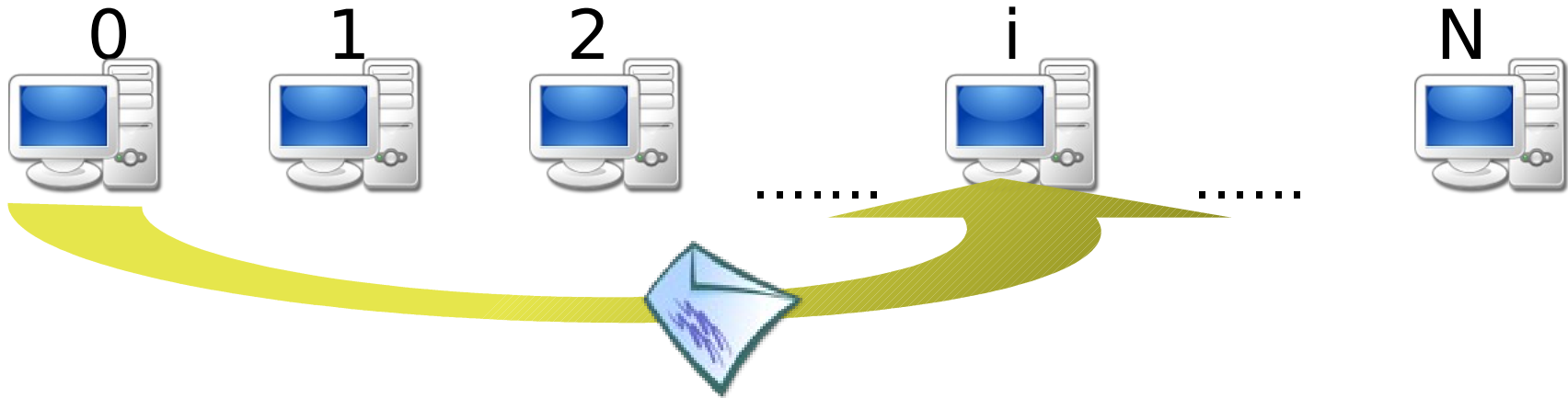
`MPI_Finalize()`

**Fin del programa**

**Inicio del programa**

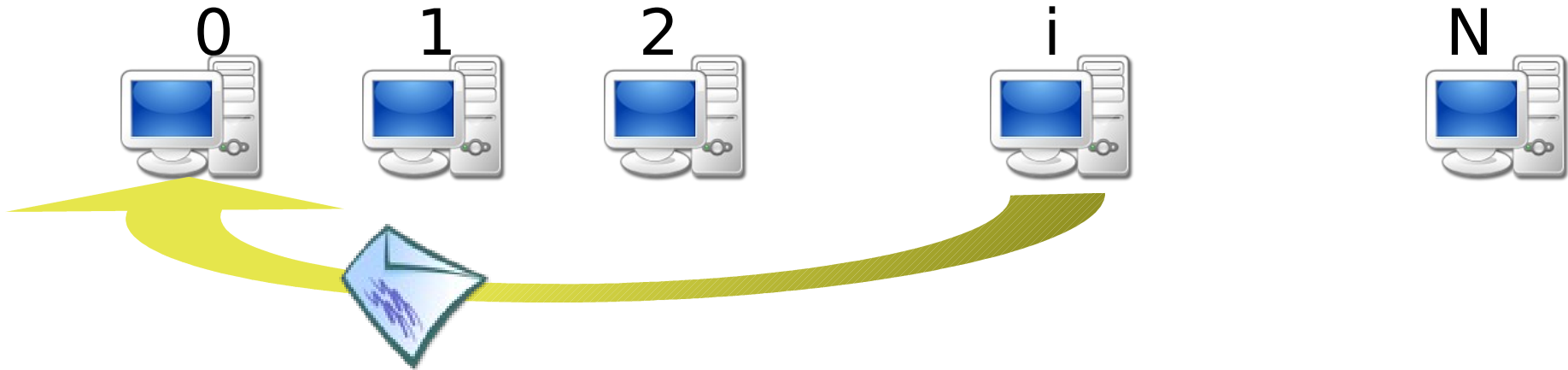
- Función para enviar mensajes

`MPI_Send(&a,size,MPI_INT,i>tag,COMM)`



- Función para recibir mensajes

`MPI_Recv(&r,1,MPI_INT,i,tag,comm,&stat)`



```
#include <stdio.h>
#include <mpi.h>
int my_id, nproc, tag = 99, source;
MPI_Status status;

int main (int argc, char** argv){
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Send(&my_id, 1, MPI_INT, (my_id+1)%nproc, tag,
MPI_COMM_WORLD);
    MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,tag,
MPI_COMM_WORLD, &status);
    printf("%d recibio mensaje de %d\n",my_id,source);
    MPI_Finalize();
}
```



El proceso de compilación difiere entre las distintas implantaciones de MPI.

Generalmente, el comando para compilar **incluye** todas las banderas del compilador necesarias para incluir tanto los **archivos cabecera** como las **bibliotecas**.

Ejemplos:

```
mpicc programa.c -o prog.exe
```

```
mpif77 programa.f -o prog.exe
```

```
pif90 programa.f90 -o prog.exe
```

El proceso de ejecución en la mayoría de las implantaciones se realiza de la siguiente manera:

```
mpirun -machinefile machines -np 4 p.exe
```

En general, ejecutar un programa en MPI es independiente de la implantación y puede necesitar varios **scripts, argumentos, y variables de ambiente**

En el estándar MPI-2 existe una forma distinta de ejecutar un programa paralelo

`mpiexec` argumentos

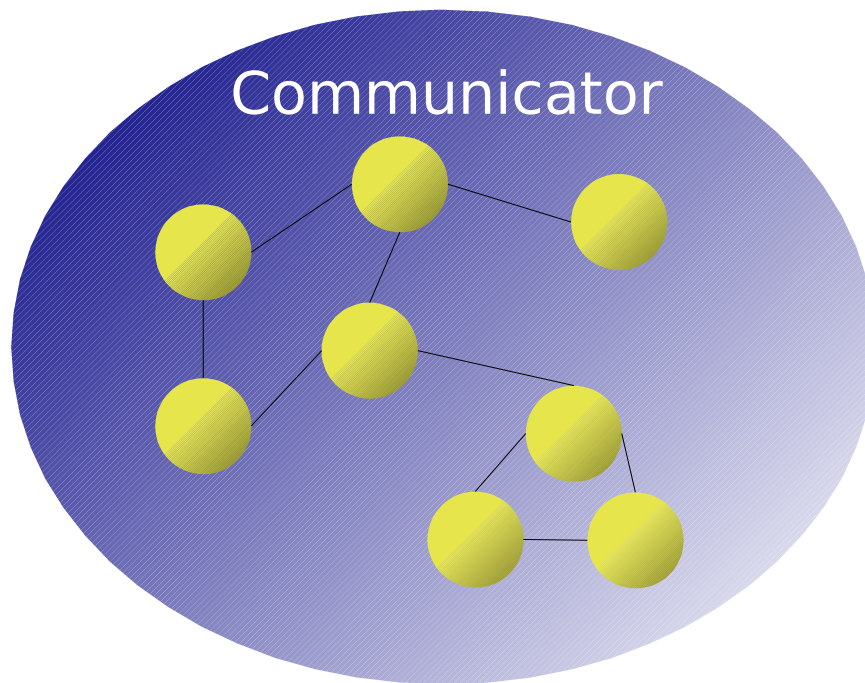
Encontramos los siguientes elementos:

- Los archivos cabecera (headers)
- El entorno de comunicación (MPI Communicator)
- Formato de las funciones de MPI
- Tamaño del entorno de comunicación
- El identificador de cada proceso dentro del entorno de comunicación
- Inicialización y finalización del programa

Los archivos cabecera contienen

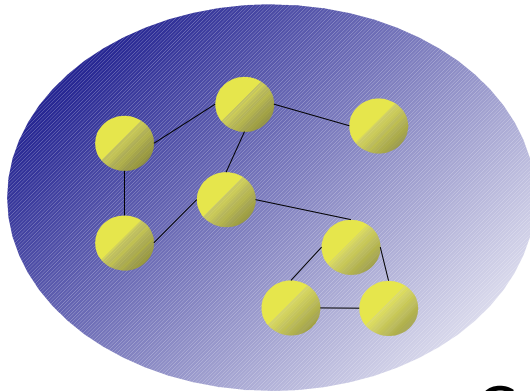
- Definiciones de los tipos de datos
- Los prototipos de las funciones
  - `mpi.h` en C
  - `mpif.h` para Fortran77 y Fortran90

El entorno de comunicación (**MPI Communicator**) es definido como una variable mediante la cual se puede hacer referencia a un grupo de procesos que se comunican entre si.

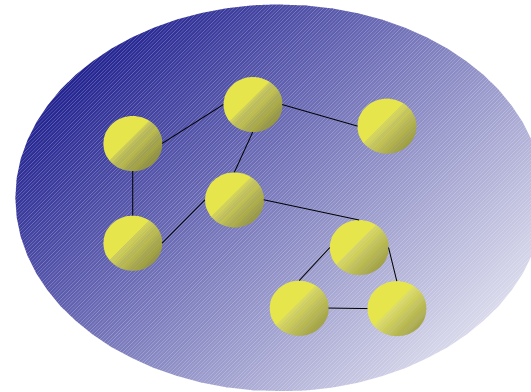


El programador puede definir múltiples entornos de comunicación

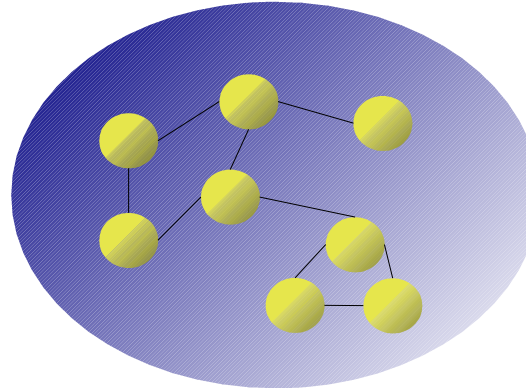
Communicator 1



Communicator 2

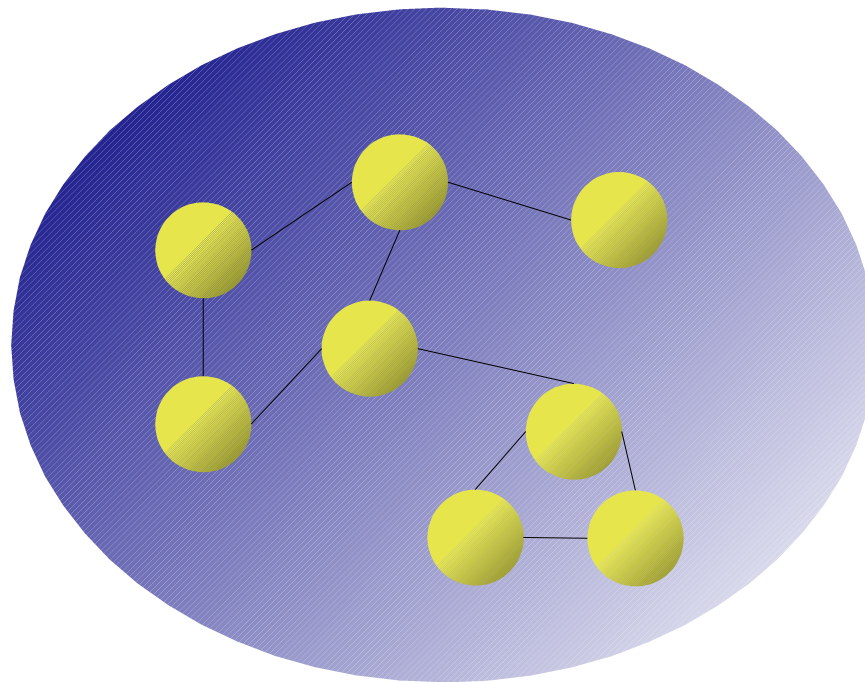


Communicator 3



Existe un entorno de comunicación **predefinido**

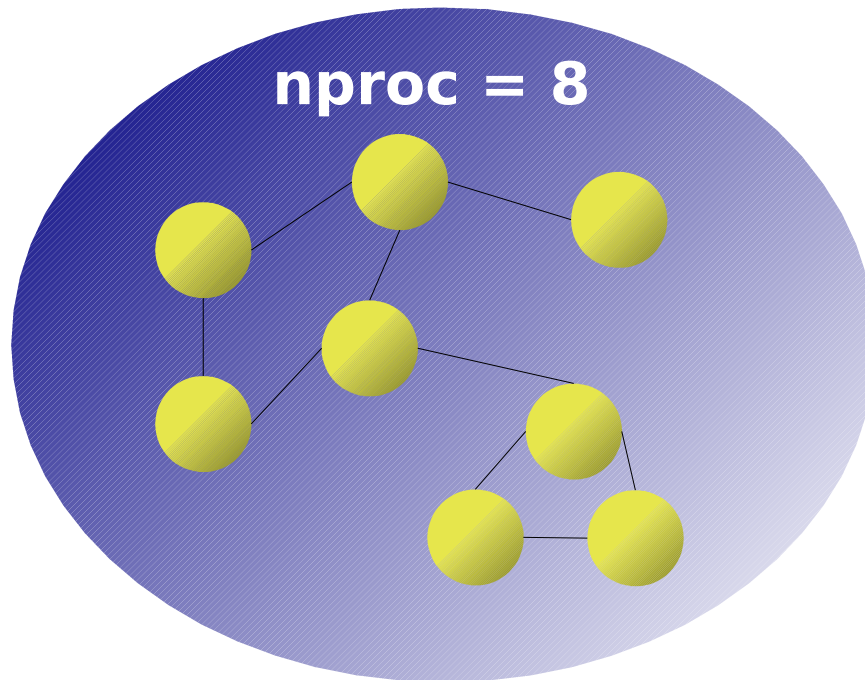
`MPI_COMM_WORLD`





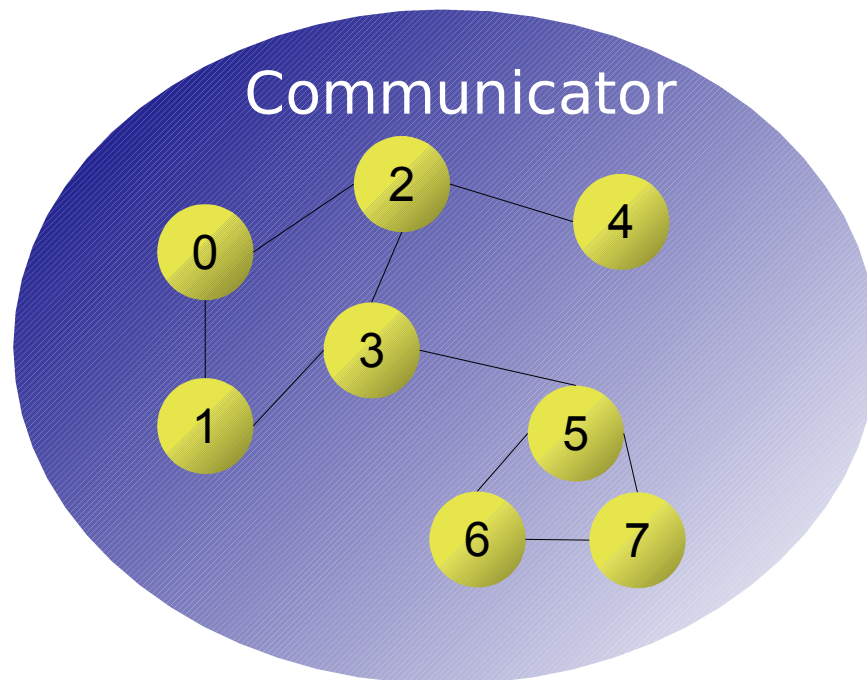
El tamaño del entorno de comunicación se refiere al número de procesos que se lograron arrancar al momento de la inicialización.

```
MPI_Comm_size(MPI_COMM_WORLD, &nproc)
```



El identificador distingue cada proceso dentro del entorno de comunicación:

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_id)
```



- Los procesos pueden ser reunidos en un grupo
- Los mensajes se transmiten dentro de un contexto determinado
- Un grupo y un contexto conforman el entorno de comunicación

- Cada proceso es identificado dentro del grupo que pertenece al entorno de comunicación a través de su ID
- El entorno predefinido contiene todos los procesos iniciales **MPI\_COMM\_WORLD**

Las funciones de envío y recepción gestionan los datos utilizando los tres primeros argumentos:

- Dirección de memoria donde están los datos
- Tamaño de la memoria
- Tipo de dato

`MPI_Send(&buffer, size, MPI_INT,.....`  
`MPI_Recv(&buffer, size, MPI_INT,.....`

Dentro de los tipos de datos que podemos encontrar en MPI tenemos:

MPI\_BYTE  
MPI\_INT  
MPI\_CHAR  
MPI\_SHORT  
MPI\_FLOAT  
MPI\_DOUBLE  
MPI\_LONG\_DOUBLE  
MPI\_UNSIGNED.....

Existen funciones dentro de MPI para construir tipos de datos propios del usuario

En MPI los mensajes contienen una etiqueta (**TAG**) que ayuda al proceso receptor a identificar el mensaje.

Cada mensaje puede ser categorizados en la recepción por una etiqueta específica o simplemente indicarse que puede tener cualquier categoría **MPI\_ANY\_TAG**

- Función para enviar mensajes

`MPI_Send(&a,n,MPI_INT,dest,tag,COMM)`

Donde

*a* son los datos a enviar

*n* es el número de elementos en *a*

*MPI\_INT* el tipo de datos

*dest* identificador del proceso al que se le enviará los datos

*tag* es la etiqueta del mensaje

*COMM* es el entorno de comunicación



- Función para recibir mensajes

`MPI_Recv(&a,n,MPI_INT,source,tag,comm,&stat)`

Donde

*a* variable donde se reciben los datos

*n* es el número de elementos en *a*

*MPI\_INT* el tipo de datos

*source* identificador del proceso que envió los datos

*tag* es la etiqueta del mensaje

*comm* es el entorno de comunicación

*stat* arreglo de enteros con información sobre el mensaje en caso de error

La **completación** del proceso de comunicación significa que la memoria utilizada en la transferencia del mensaje se puede reutilizar.

- **En el emisor:** la variable enviada se puede reutilizar
- **En el receptor:** la variable recibida se puede leer

El proceso de comunicación en MPI tiene dos formas o **modos** de llevarse a cabo

- **Bloqueante**: las rutinas de envío y recepción retornan cuando se ha completado la comunicación.
- **No bloqueante**: las funciones retornan inmediatamente. El usuario debe chequear si se ha **completado la comunicación** antes de utilizar el contenido de una variable.

Las funciones `MPI_Send` y `MPI_Recv` son bloqueantes:

Cuando `MPI_Send` retorna entonces se puede seguir utilizando el contenido del buffer (datos enviados) y modificarlos si es necesario.

No se tiene conocimiento si el nodo destino recibió el mensaje.

Las funciones `MPI_Send` y `MPI_Recv` son bloqueantes:

`MPI_Recv` retornará sólo cuando se hayan recibido todos los datos en el buffer de recepción.

Las implementaciones de MPI proporcionan también funciones **no bloqueantes** para el envío y recepción de mensajes.

`MPI_Isend(...MPI_Request *req)`

`MPI_Irecv(...MPI_Request *req)`

Donde:

*req* es el manejador de la solicitud de la comunicación

MPI\_Isend y MPI\_Irecv permiten que la ejecución del programa continúe y se pueda realizar otras tareas.

Sin embargo, es necesario que el proceso de comunicación sea **completado** para poder utilizar el contenido de la variable que se envió o se recibió.

El usuario debe chequear si se ha **completado la comunicación** antes de utilizar el contenido de una variable.

Las implantaciones de MPI proporcionan también una forma de **comunicación sincrónica**.

Esto quiere decir que la comunicación no se completará hasta que el mensaje sea recibido por el nodo destino.

<b>MPI_Ssend</b>	bloqueante
<b>MPI_Issend</b>	no bloqueante



Cuando se utilizan las funciones no bloqueantes el usuario debe agregar código para verificar si la comunicación se ha completado.

```
MPI_Test(MPI_Request *req, int *flag, MPI_Status *stat)
```

Donde:

*req* es el manejador de la comunicación

*flag* es verdadera si se completó la comunic.

*stat* es el estado del objeto

Si no se puede realizar otra tarea en el programa, pues se necesita el dato del mensaje, entonces se debe esperar por este.

```
MPI_Wait(MPI_Request *req, MPI_Status *stat)
```

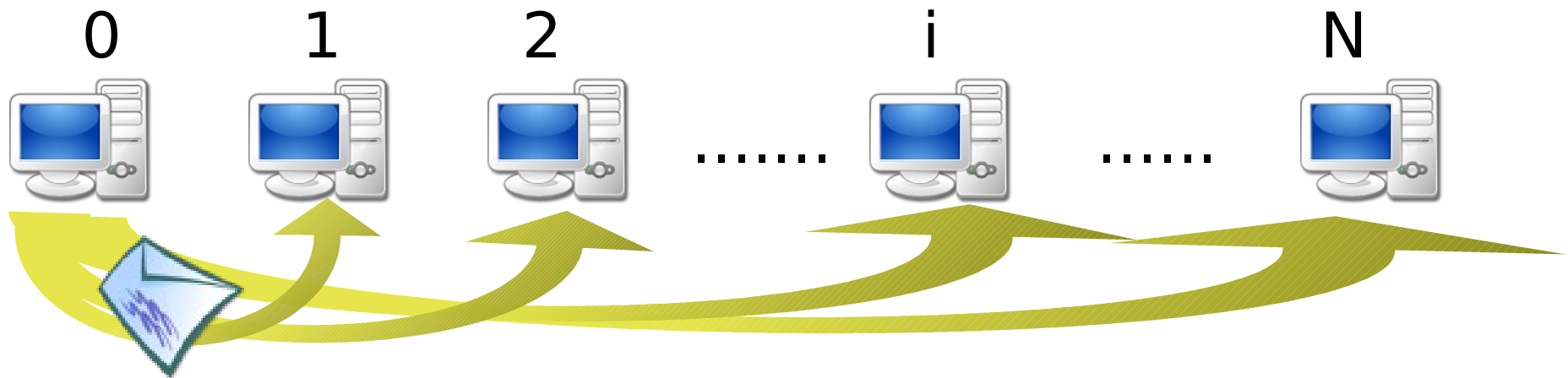
Donde:

*req* es el manejador de la comunicación  
*stat* es el estado del objeto

En ciertas aplicaciones se requiere enviar un conjunto de datos a múltiples procesos o recibir en un único proceso datos de varios remitentes.

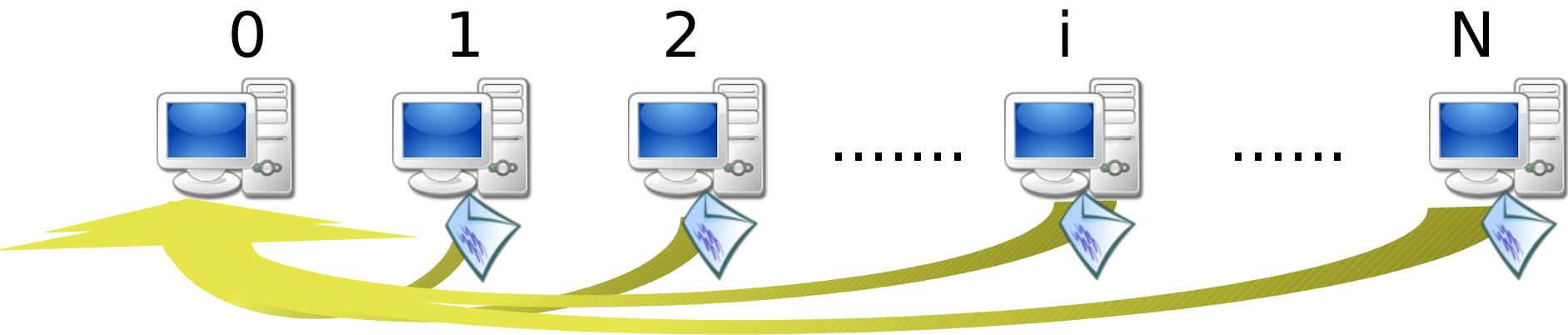
Las comunicaciones colectivas permiten la transferencia de datos entre todos los procesos que pertenecen a un grupo específico.

Por ejemplo: enviar un mismo mensaje al resto de los procesos



`MPI_Bcast(buffer, count, datatype, src, comm)`

Por ejemplo: recibir mensajes del resto de los procesos



`MPI_Reduce(*sendbuf, *recvbuf, count, type, MPI_Op op, src, comm)`

**MPI\_Op** especifica el tipo de operación que se va a ejecutar con los datos que se reciben de cada nodo.

MPI tiene operaciones predefinidas como:

- Suma
- Máximo
- AND lógico, etc

El usuario tiene la posibilidad de definir sus propias operaciones

Las operaciones colectivas de MPI pueden ser clasificadas en tres clases:

- **Sincronización:** Barreras para sincronizar.
- **Movimiento:** (transferencia) de datos. Operaciones para difundir, recolectar y esparcir.
- **Cálculos colectivos:** Operaciones para reducción global, tales como suma, máximo, mínimo o cualquier función definida por el usuario.

**MPI\_Barrier**(MPI\_Comm comm)

**MPI\_Gather**(void \*sendbuf, int sendcount, MPI\_Datatype sendtype, void \*recvbuf, int recvcount, MPI\_Datatype recvtype, int root, MPI\_Comm comm)

**MPI\_Scatter**(void \*sendbuf, int sendcount, MPI\_Datatype sendtype, void \*recvbuf, int recvcount, MPI\_Datatype recvtype, int root, MPI\_Comm comm)



Las implantaciones de MPI permiten definir tipos de datos por parte del usuario.

Esto es útil a la hora de trabajar con tipos de datos compuestos como estructuras o registros

Así mismo, las implantaciones de MPI permiten crear nuevas operaciones para las comunicaciones colectivas.

Las rutinas de comunicación colectiva proporcionan un alto nivel para organizar el programa paralelo.

Cada proceso ejecuta las mismas operaciones de comunicación.

MPI proporciona un conjunto importante de funciones para la comunicación colectiva.