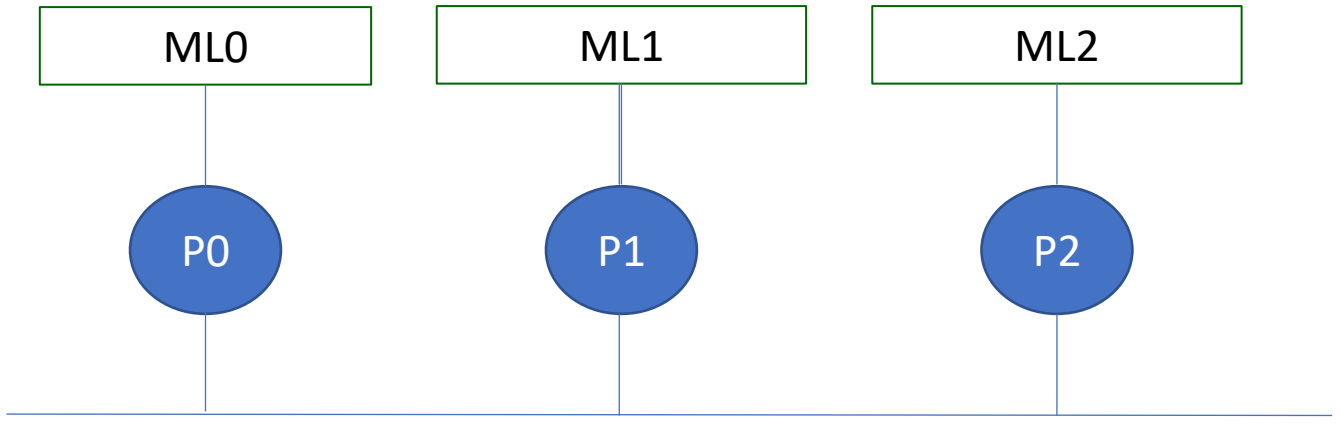
Modelo de arquitectura en memoria distribuida (MPI)



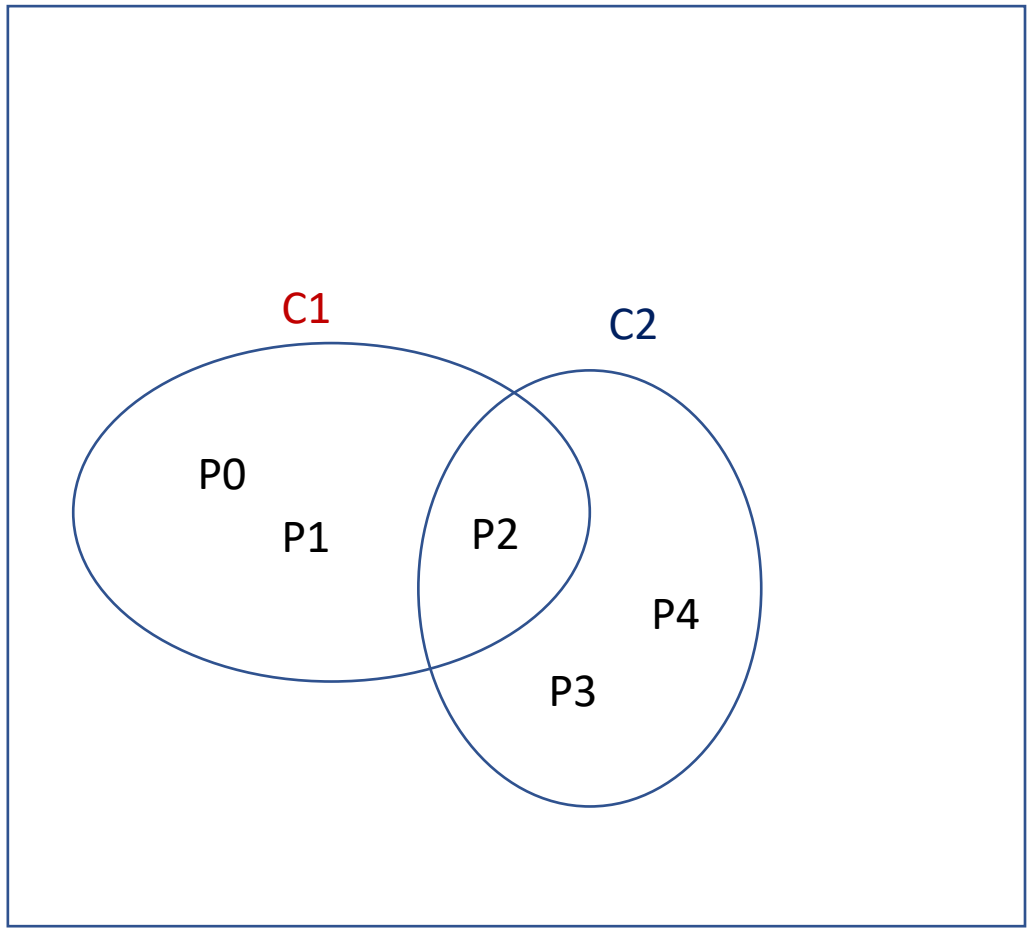
Pi= proceso con identificador i
MLi= memoria local proceso Pi
Ri = red de interconexion

- Las memorias MLi tienen el mismo programa
- Los procesos pueden ejecutar diferentes códigos, según sus identificadores
- Cada memoria local MLi tiene los datos locales del proceso Pi
- Para intercambiar los datos entre 2 procesos se usa la red de interconexión para el envío y recepción de datos contenidos en los mensajes

Comunicadores en MPI

Comunicador = $Grupo\ de\ procesos + Contexto$

MPI_COMM_WORLD (comunicador universal)



- Siempre está definido el comunicador universal (MPI_COMM_WORLD), el cual está formado por todos los procesos que se han lanzado
- Se puede definir diferentes comunicadores
- En toda operación de comunicación se debe indicar el **comunicador** usado: la operación sólo afecta a los procesos de ese comunicador
- **MPI_Comm_rank**: Permite conocer el identificador del proceso dentro de un comunicador
- **MPI_Comm_size**: Permite conocer el número de procesos en un comunicador
- Ejemplo:

```
int id, p;
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Comm_size(MPI_COMM_WORLD, &p);
```

Funciones de comunicación bloqueantes

MPI_Send(void *senbuf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

MPI_Recv(void *recbuf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

senbuf/recbuf puntero al dato que se va a envíar/recibir

- Dato simple lleva delante &; array no tiene delante &

count: número de datos a enviar/recibir

datatype : tipo de dato (MPI_DOUBLE, MPI_INT, MPI_CHAR, etc.)

- dato simple: 1, array: número de elementos del array

source/dest: Identificador del proceso que va a envíar/recibir el dato

tag: etiqueta del mensaje (entero positivo)

comm: comunicador, normalmente se usa el comunicador universal **MPI_COMM_WORLD**

status: información del mensaje:

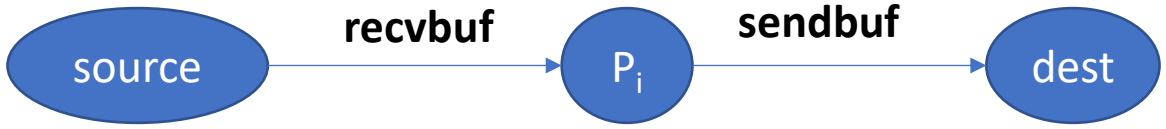
- status.MPI_SOURCE: proceso que ha realizado el envío
- status.MPI_TAG: etiqueta del mensaje recibido

En MPI_Recv se pueden usar las constantes:

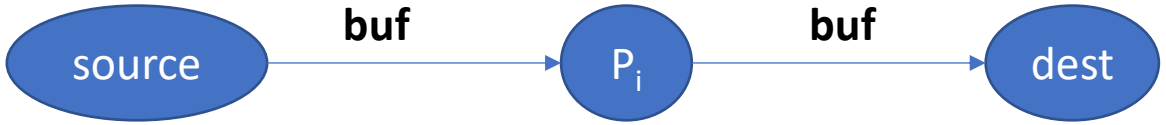
MPI_ANY_SOURCE (source), MPI_ANY_TAG(tag), MPI_STATUS_IGNORE(status)

Operaciones combinadas en MPI

MPI_Sendrecv(void ***sendbuf**, int sendcount, MPI_Datatype sendtype, int **dest**, int sendtag,
void ***recvbuf**, int recvcount, MPI_Datatype recvtype, int **source**, int recvtag, MPI_Comm comm, MPI_Status *status)



MPI_Sendrecv_replace(void ***buf**, int count, MPI_Datatype datatype, int **dest**, int sendtag, int **source**, int recvtag, MPI Comm_comm,
MPI_Status *status)



- P_i es el proceso que invoca a la función
- Ambas operaciones combinan en una sola llamada el envío de un mensaje a un destino y la recepción de otro mensaje, procedente de otro proceso.
- Origen y destino pueden ser el propio proceso P_i.
- Esta operación es útil para ejecutar una operación de desplazamiento a través de una cadena de procesos.
- La diferencia entre las dos operaciones es que en MPI_Sendrecv se utilizan diferentes variables para el envío y recepción y en MPI_Sendrecv_replace la misma variable.

Funciones de comunicación no boqueantes en MPI

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

- Los primeros 6 argumentos coinciden con los argumentos que aparecen en los envíos bloqueantes

MPI Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, **MPI Request *request**)

- Los primeros 6 argumentos coinciden con los argumentos que aparecen en las recepciones bloqueantes

request: puntero de tipo MPI_Request a la variable que almacena la petición no bloqueante

Funciones necesarias en las comunicaciones no boqueantes:

- Espera:

MPI_Wait(MPI_Request *request, MPI_Status *status)

- Comprobación:

MPI_Test(MPI Request *request, int *flag, MPI_Status *status)

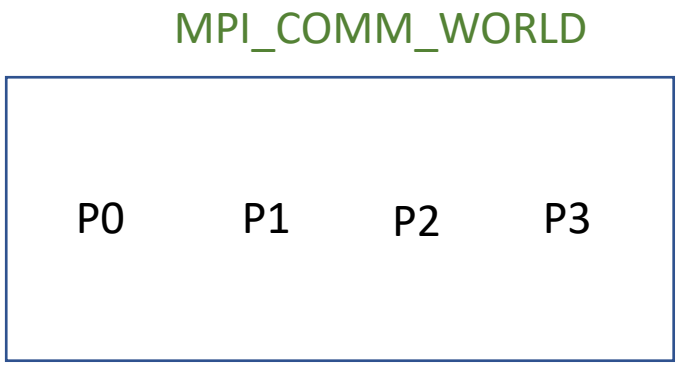
  - **flag**: 0 si no se ha producido la petición de recepción del mensaje, 1 si se ha producido
  - **status**: puntero al estado de la comunicación

Modelo de programación en MPI

Ejemplo: hola_mpi.c

```
#include<stdio.h>
#include <mpi.h>
int main(int argc, char* argv[]) {
   int id; /* identificador del proceso */
   int p; /* número de procesos */
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &id);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   printf("Hola mundo, soy el proceso %d de %d\n", id, p);
   MPI_Finalize();
   return 0;
}
```

- Compilación: mpicc –o hola_mpi hola_mpi.c
- Ejecución: mpiexec -n 4 ./hola_mpi




```
 Local host: MSI
----------------------------------------
Hola mundo, soy el proceso 0 of 4 procesos
Hola mundo, soy el proceso 1 of 4 procesos
Hola mundo, soy el proceso 2 of 4 procesos
Hola mundo, soy el proceso 3 of 4 procesos
```

Problemas de interbloqueo

- Un mal uso de las comunicaciones síncronas Send y Recv puede producir un interbloqueo

```
/* Proceso 0 */          /* Proceso 1 */
Send(x, 1);              Send(y, 0);
Recv(y, 1);              Recv(x, 0);
```

Ambos procesos pueden quedar bloqueados al intentar enviar al mismo tiempo

- Caso de envío con buffer: El ejemplo anterior no causaría interbloqueo

- Puede haber otras situaciones con interbloqueo con el envío con Buffer

```
/* Proceso 0 */          /* Proceso 1 */
Recv(y, 1);              Recv(x, 0);
SendB(x, 1);             SendB(y, 0);
```

- Posible solución en ambos casos:

```
/* Proceso 0 */          /* Proceso 1 */
Send(x, 1);              Recv(x, 0);
Recv(y, 1);              Send(y, 0);
```

Intercambiar el orden de uno de ellos

Ejemplo

Programa de MPI en e que P0 le un entero positivo **m** y un vector **b** mediante una función **void lee**, y a continuación P0 envía al proceso P1 **m** **y** las primeras **m** componentes del vector **b**:

```
#include <mpi.h>
#include <stdio.h>
#define N 1000
void lee(int *m, double b[N]);
int main (int argc, char **argv){
        int   m;
        double b[N];
        MPI_Init(&argc, &argv);
        int   id; /*identificador de proceso*/
        MPI_Comm_rank(MPI_COMM_WORLD, &id);
        if (id == 0){
                leer(&m, b);/*P0 lee m y b*/
                MPI_Send(&m, 1,  MPI_INT, 1, 100, MPI_COMM_WORLD);
                MPI_Send(b, m, MPI_ DOUBLE, 1, 200, MPI_COMM_WORLD);
        }
        else if (id==1){
                MPI_Recv(&m, 1, MPI_INT, 0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                MPI_Recv(b, m, MPI_DOUBLE, 0, 200, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        MPI_Finalize();
        return 0;
}
```

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

Receive a message from one process:

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

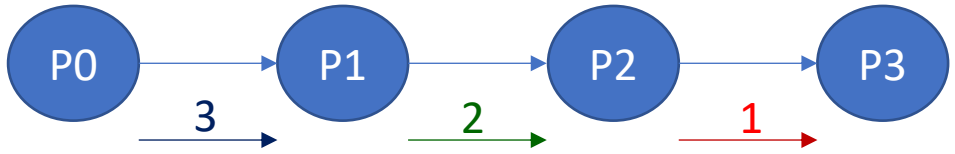
Nota: En el caso del envío/recepción de un vector, por ejemplo, el envío del vector **b**, el primer argumento puede ponerse como **b** o como &**b**[0]

Desplazamiento en Malla 1-D

Cada proceso envía el dato almacenado en **val** al vecino derecho y recibe en **val** el dato enviado por su vecino izquierdo

```
if (rank == 0)
    prev = MPI_PROC_NULL;
else
    prev = rank-1;
if (rank == p-1)
    next = MPI_PROC_NULL;
else
    next = rank+1;
MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
MPI_Recv(&val, 1, MPI_DOUBLE, prev, 0, comm, &status);
```

```
P0:
  Send(val, P1)
  Rec(val, NULL)
P1:
  Send(val,P2)
  Rec(val, P0)
P2:
  Send(val,P3)
  Rec(val, P1)
P3:
  Rec(val, P2)
  Send(val,NULL)
```



Primero se realiza la comunicación 1, después la 2 y finalmente la 3

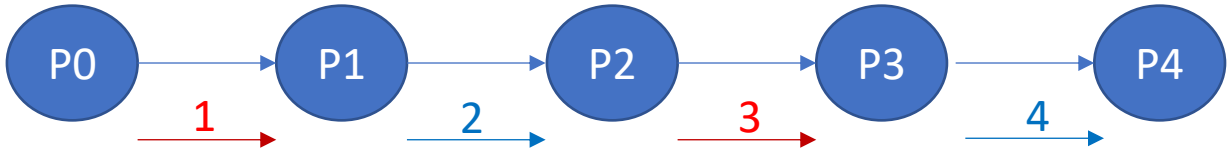
las comunicaciones se realizan secuencialmente, sin concurrencia

Desplazamiento en Malla 1-D

Protocolo Pares-Impares (soluciona el problema de la serialización)

```
if (rank == 0) prev = MPI_PROC_NULL;
else    prev = rank-1;
if (rank == p-1) next = MPI_PROC_NULL;
else    next = rank+1;
if (rank%2 == 0) {
    MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
    MPI_Recv(&val, 1, MPI_DOUBLE, prev, 0, comm, &status);
} else {
    MPI_Recv(&tmp, 1, MPI_DOUBLE, prev, 0, comm, &status);
    MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
    val = tmp;
}
```

```
P0:
  Send(val, P1)
  Rec(val, NULL)
P1:
  Rec(tmp, P0)
  Send(val,P2)
P2:
  Send(val,P3)
  Rec(val, P1)
P3:
  Rec(tmp, P2)
  Send(val,43)
P4:
  Send(val, NULL)
  Rec(val, P3)
```



Las comunicaciones 1 y 3 se pueden realizar concurrentemente, al igual que las comunicaciones 2 y 4

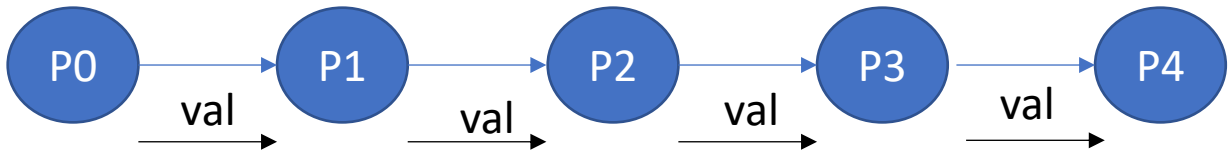
Desplazamiento en Malla 1-D

Usar operaciones combinadas (soluciona el problema de la serialización)

```
if (rank == 0) prev = MPI_PROC_NULL;
else prev = rank-1;
if (rank == p-1) next = MPI_PROC_NULL;
else next = rank+1;
MPI_Sendrecv_replace( &val, 1, MPI_DOUBLE, next, 100, prev, 100, comm, &status);
```

Combined send and receive (using the same buffer):

```
int MPI_Sendrecv_replace(void *buf, int
    count, MPI_Datatype datatype, int
    dest, int sendtag, int source, int
    recvtag, MPI_Comm comm, MPI_Status
    *status)
```



Desplazamiento en Malla 1-D

Usar operaciones no bloqueantes (resuelve el problema de la serialización)

Non-Blocking Point-to-Point

Begins a non-blocking send:

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

Begin to receive a message:

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

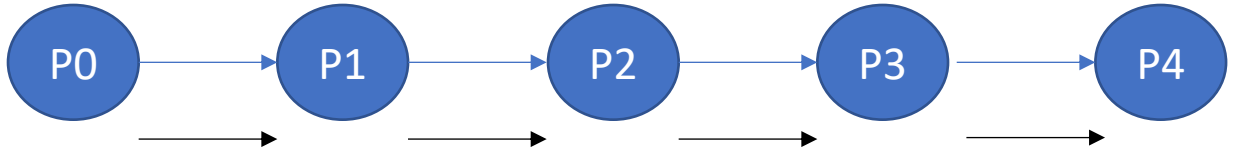
Complete a non-blocking operation:

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

Check or complete a non-blocking operation:

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

```
if (rank == 0) prev = MPI_PROC_NULL;
else prev = rank-1;
if (rank == p-1) next = MPI_PROC_NULL;
else next = rank+1;
```




```
MPI_Isend(&val, 1, MPI_INT, next, 111, MPI_COMM_WORLD, &req);
MPI_Recv(&aux, N, MPI_INT, prev, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Wait(&req, MPI_STATUS_IGNORE);
val=aux;
```