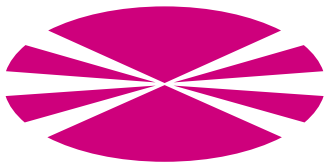
T3: Principios de Programación Paralela

T3.2: Modelo de Paso de Mensajes

Departamento de Ingeniería de Computadores

Primavera 2023



Índice

Conceptos Básicos

Modelo de Paso de Mensajes

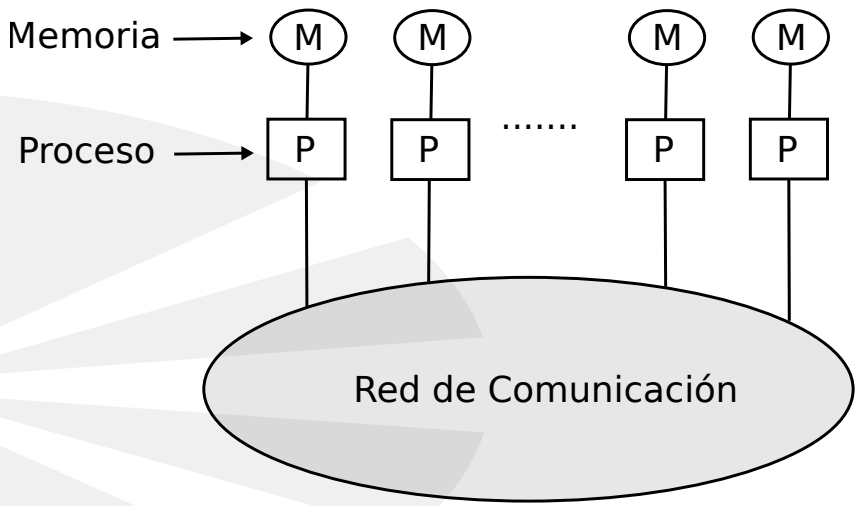
- Paradigma muy extendido en programación paralela
- MPI (Message Passing Interface) es la solución más popular (desde MPI1, 1992)
- Mínimos requerimientos al HW para su implementación
- Soporta un gran número de entornos paralelos, especialmente de memoria distribuida
- En este modelo uno o más procesos se comunican llamando a rutinas de una biblioteca para recibir y enviar mensajes entre procesos
- Control del paralelismo por el programador, que ha de evitar dependencias de datos, interbloqueos y *race conditions*
- Llamadas a MPI (u otra librería) desde programas C o Fortran
- Implementaciones de MPI: MPICH2, OpenMPI, Intel MPI, ...

Conceptos Básicos

Modelo de ejecución de un programa en paso de mensajes

- Programa paralelo compuesto de múltiples procesos/tareas que utilizan su propia memoria local durante la computación
- Generalmente un proceso/tarea por elemento de procesado (e.g., CPU core)
- Comunicación entre procesos mediante envío y recepción de mensajes *two-sided*, un envío se corresponde con una recepción

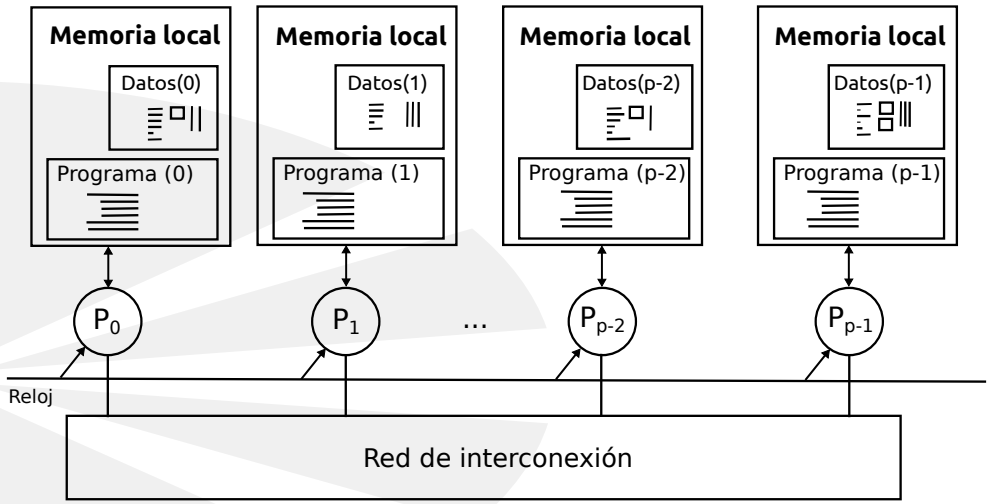
Arquitectura del modelo de paso de mensajes:



Conceptos Básicos

Estructura de un programa en paso de mensajes

- **MPMD** (Multiple Program Multiple Data): cada proceso/tarea tiene su propio programa con comunicaciones asíncronas entre ellos (máxima flexibilidad y complejidad)
- **SPMD** (Single Program Multiple Data): todos los procesos/tareas comparten un mismo programa/binario aunque en su lógica interna las tareas se pueden ejecutar de forma condicional dependiendo del proceso. Se suele hacer uso de comunicaciones síncronas con lo que suele resultar más sencillo programar pero con menor escalabilidad.



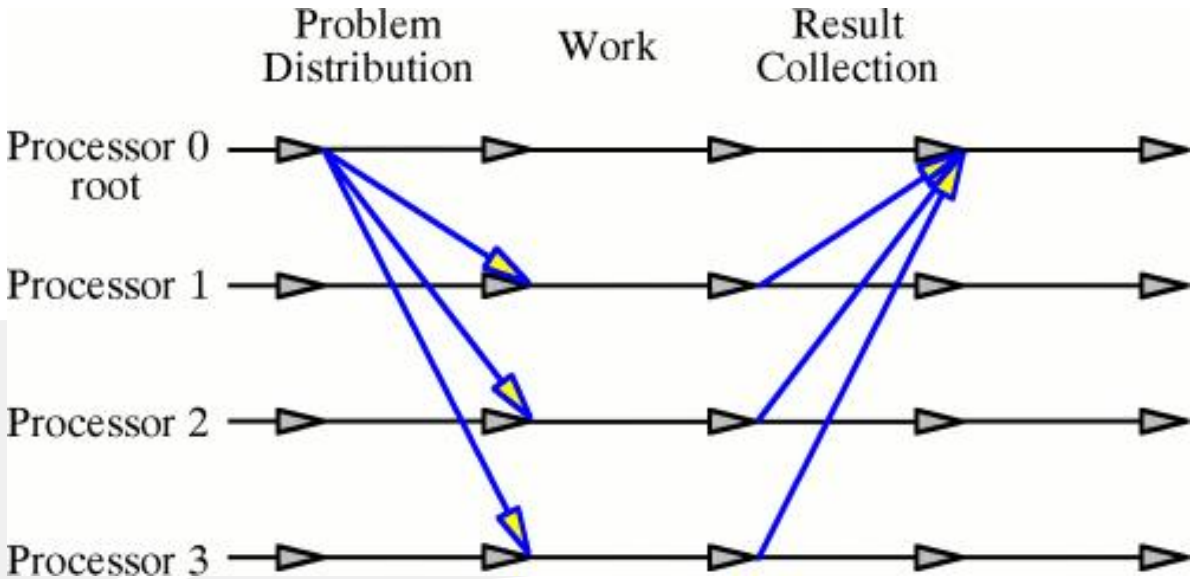
Conceptos Básicos

Características de un programa MPI C

- Incluye la librería de MPI (mpi.h)
  - Las funciones MPI tienen la forma `MPI_Nombre(parámetros)`
  - Devuelven un valor de éxito (`MPI_SUCCESS`) o error (`MPI_ERR_{*}`). Consultar man.
- Los procesos son independientes hasta que se inicializa MPI (`MPI_Init`), pudiendo colaborar intercambiando datos, sincronizándose tras ese punto
- Clave que los procesos conozcan el número de procesos (*numprocs*, obtenido con `MPI_Comm_size`) que se han puesto en marcha así como su identificador (entre 0 y $numprocs - 1$, obtenible con `MPI_Comm_rank`)
- `MPI_Finalize` se llama cuando ya no es necesario que los procesos colaboren entre sí. Libera todos los recursos reservados por MPI
- `MPI_COMM_WORLD`: comunicador global, incluye a todos los procesos

Conceptos Básicos

- Programación paralela basada en procesos que se comunican a través de mensajes
- Debemos centrarnos en las fases más costosas computacionalmente



Conceptos Básicos

Estructura básica de un programa MPI C

```
#include <mpi.h>   // 1. include the library

int main(int argc, char *argv[]) {
  int numprocs, rank;

  MPI_Init(&argc, &argv); // 2. initialize MPI environment
                          //    and remove extra arguments

  MPI_Comm_size(MPI_COMM_WORLD, &numprocs); // 3. get number of processes
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);     // 4. get self rank

  // ... do stuff

  MPI_Finalize(); // 5. wait for all processes
                  //    and finalize MPI environment
}
```

Conceptos Básicos

Hello World MPI C

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

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

  printf("Process %d on %s out of %d\n", rank, processor_name, numprocs);

  MPI_Finalize();
}
```

Conceptos Básicos

Compilación

```
mpicc mpi-hello.c -o mpi-hello
```

Ejecución

```
mpirun -np 4 ./mpi-hello
```

Output

```
Process 0 on localhost out of 4
Process 1 on localhost out of 4
Process 3 on localhost out of 4
Process 2 on localhost out of 4
```

Mensajes MPI

Tipos de operaciones

- **Punto a punto**: De un proceso origen a un proceso destino
  - MPI_Send, MPI_Recv, ...
  - Tienen argumento `dest` o `source`
- **Colectivas**: Involucran a todos los procesos de un comunicador
  - MPI_Barrier, MPI_Bcast, MPI_Reduce, ...
  - Suelen tener argumento `root`
- **Bloqueantes**: Los procesos esperan a que el mensaje se reciba
  - Ojo, bloqueo no necesariamente implica sincronización
- **No bloqueantes**: Los procesos continuan independientemente del receptor
  - MPI_Isend, MPI_Irecv, ...
  - Mismas funciones pero con prefijo `I`
  - `MPI_Wait` garantiza que la operación se ha completado

Mensajes MPI

Conceptos clave

- **MPI_Datatype**: Tipos de datos usados por MPI

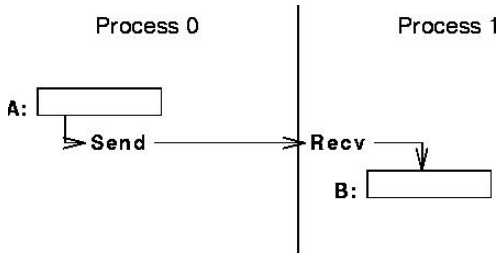
MPI	C	MPI	C
MPI_INT	int	MPI_LONG	long
MPI_FLOAT	float	MPI_DOUBLE	double
MPI_CHAR	char	MPI_SHORT	short

  - Se pueden crear tipos derivados y complejos
- **Tag**: Número `arbitrario` para identificar mensajes punto a punto
  - El receptor puede usar la constante `MPI_ANY_TAG`
- **Status**: Estructura con información sobre la operación
  - `count`: número de elementos recibidos
  - `MPI_SOURCE`: fuente del envío
  - `MPI_TAG`: tag del envío
  - Puede usarse `MPI_STATUS_IGNORE`

Operaciones Punto a Punto

Punto a punto MPI

- **Bloqueantes**: MPI_Send y MPI_Recv
  - Por cada **envío** por parte de un proceso debe haber una **recepción**
  - ... y viceversa!




```
assert(numprocs % 2 == 0);

if (my_id % 2 == 0)
  partner = my_id+1;
else
  partner = my_id-1;

MPI_Send(..., partner, ...);
MPI_Recv(..., partner, ...);
```

Este código se bloquea...

```
assert(numprocs % 2 == 0);

if (my_id % 2 == 0)
  partner = my_id+1;
else
  partner = my_id-1;

if (my_id % 2 == 0) {
  /* procesos pares */
  MPI_Send(..., partner, ...);
  MPI_Recv(..., partner, ...);
}
else
{
  /* procesos impares */
  MPI_Recv(..., partner, ...);
  MPI_Send(..., partner, ...);
}
```

Operaciones Punto a Punto

MPI_Send

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

- Envía un mensaje al proceso `dest` en el comunicador `comm`
- El mensaje está almacenado en `buff` y consta de **al menos** `count` items del tipo `datatype`
- El mensaje está etiquetado con un `tag`
- La llamada a MPI_Send finaliza cuando *buff* puede ser reusado (generalmente cuando el mensaje ha sido recibido en el destino)

Operaciones Punto a Punto

MPI_Recv

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

- Recibe un mensaje del proceso `source` del comunicador `comm` con la etiqueta `tag`
  - También se puede recibir de cualquier proceso del comunicador con `MPI_ANY_SOURCE`
  - También se puede recibir mensajes con cualquier etiqueta con `MPI_ANY_TAG`
- En los dos casos anteriores se recupera el *source* o *tag* recibidos accediendo a `status.MPI_SOURCE` y/o a `status.MPI_TAG`
- El mensaje se recibe en `buff` y consta de un **máximo** de `count` items del tipo `datatype`
- La llamada a *MPI_Recv* finaliza cuando se ha recibido el mensaje en *buff*

Operaciones Punto a Punto

Variantes bloqueantes de send

- MPI_Ssend: síncrona.
- MPI_Bsend: con buffer.

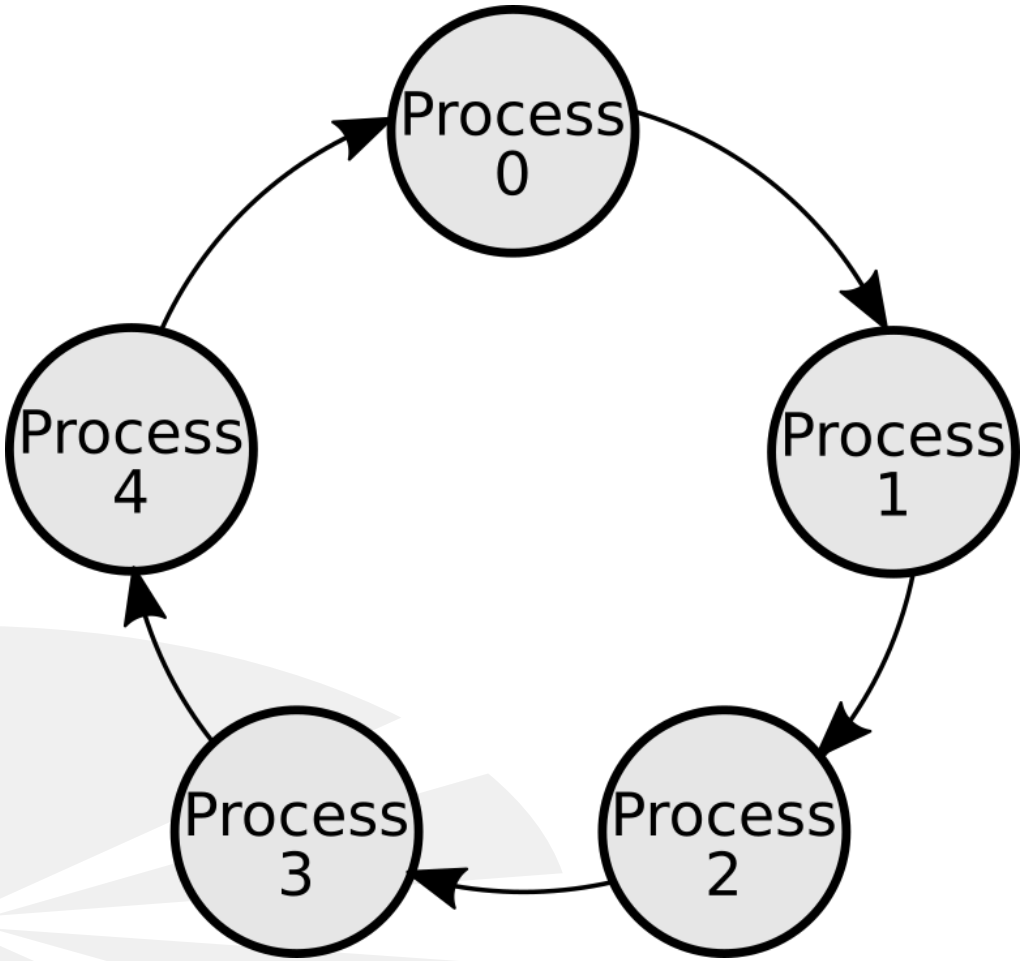
Comunicaciones P2P no bloqueantes

```
int MPI_Isend( void *buff, ..., MPI_Request *request);
int MPI_Irecv( void *buff, ..., MPI_Request *request);
```

- Inician el proceso de envío/recepción, pero no garantizan su compleción: no es seguro modificar el buffer tras la llamada.
- El objeto MPI_Request contiene la información necesaria sobre la operación, permitiendo invocar posteriormente a MPI_Wait/MPI_Waitall para garantizar la finalización
- Son compatibles con los variantes bloqueantes, e.g., es posible enviar un mensaje con MPI_Send y recibirlo con MPI_Irecv

Operaciones Punto a Punto

Ejemplo MPI: **Ring**



Operaciones Punto a Punto

Ring MPI C

```
// ... MPI initialization stuff
int passed_num = 0;

printf("  Process %d/%d: passed_num = %d (before)\n",
       rank, numprocs, passed_num);

if (my_id == 0) {
  passed_num = 1;

  MPI_Send(&passed_num,1,MPI_INT,1,0,MPI_COMM_WORLD);

  MPI_Recv(&passed_num,1,MPI_INT,numprocs-1,0,MPI_COMM_WORLD,&status);

}
else
{
  MPI_Recv(&passed_num,1,MPI_INT,my_id-1,0,MPI_COMM_WORLD,&status);

  passed_num++;

  MPI_Send(&passed_num,1,MPI_INT,(my_id+1)%numprocs,0,MPI_COMM_WORLD);
}

printf("  Process %d/%d: passed_num = %d (after)\n",
       rank, numprocs, passed_num);

// ...
```

Operaciones Punto a Punto

Ring MPI C (Salida)

```
user@localhost:~/ $ mpirun -n 8 ./ring

  Process 1/8: passed_num = 0 (before)
  Process 3/8: passed_num = 0 (before)
  Process 4/8: passed_num = 0 (before)
  Process 2/8: passed_num = 0 (before)
  Process 6/8: passed_num = 0 (before)
  Process 7/8: passed_num = 0 (before)
  Process 5/8: passed_num = 0 (before)
  Process 0/8: passed_num = 0 (before)
  Process 1/8: passed_num = 2 (after)
  Process 2/8: passed_num = 3 (after)
  Process 3/8: passed_num = 4 (after)
  Process 4/8: passed_num = 5 (after)
  Process 5/8: passed_num = 6 (after)
  Process 6/8: passed_num = 7 (after)
  Process 7/8: passed_num = 8 (after)
  Process 0/8: passed_num = 8 (after)
```

Observación: los procesos hacen el segundo *printf* de forma ordenada. ¿Por qué?

Operaciones Colectivas

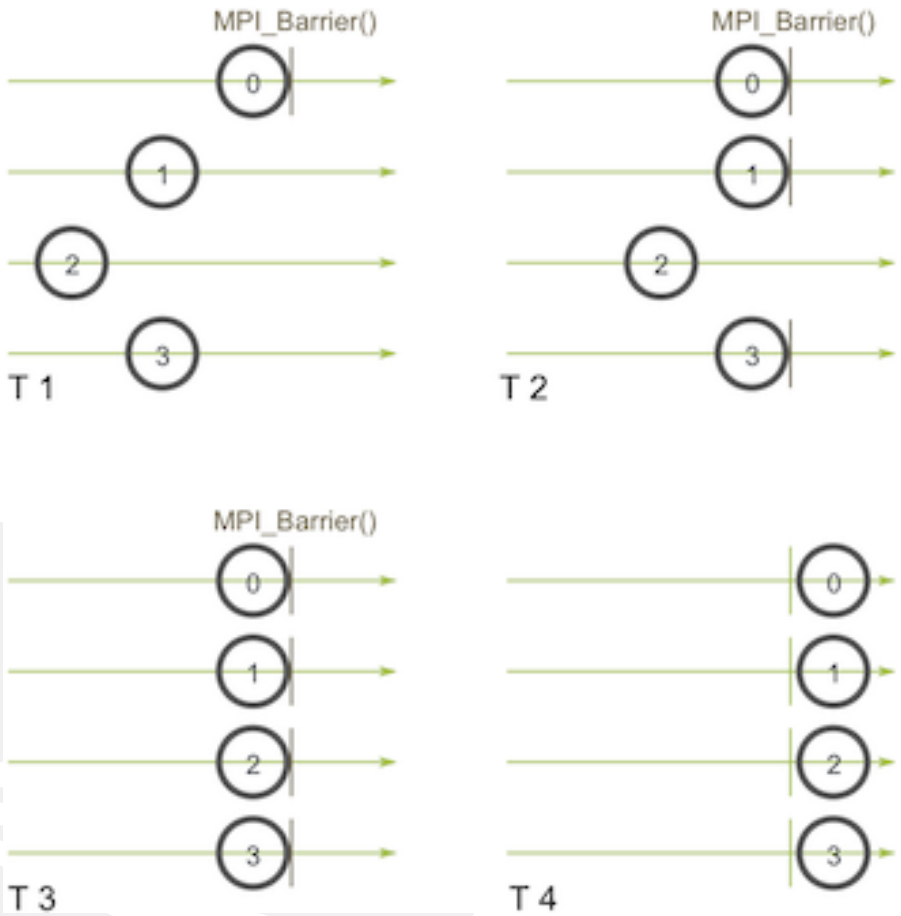
Colectivas MPI

- Operaciones típicas en las que intervienen todos los procesos de un comunicador
  - Barrier o barrera
  - Broadcast o difusión
  - Scatter o reparto
  - Gather o recolección
  - Reduce o reducción
  - Otras (e.g., Scan)
  - Combinaciones de las previas (e.g., Allreduce o Allgather)
- Uso recomendable al incrementar productividad:
  - Mayor rendimiento (optimizadas para cada librería, sistema, etc...)
  - Reducción de errores
  - Codificación a más alto nivel

Operaciones Colectivas

MPI_Barrier: Establece una barrera que bloquea el programa hasta que todos los procesos han alcanzado esta rutina.

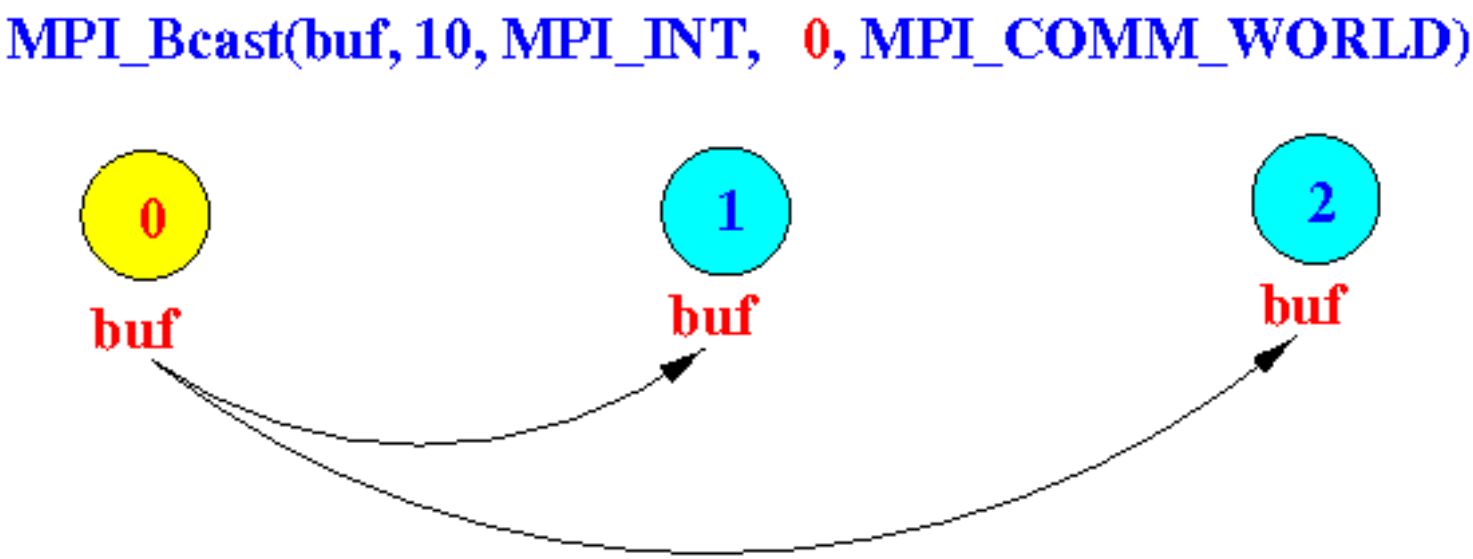
```
int MPI_Barrier(MPI_Comm comm);
```



Operaciones Colectivas

MPI_Bcast: comunicación uno a todos de *count* datos del tipo *datatype* desde el proceso raíz (*root*) al resto de procesos del comunicador *comm*.

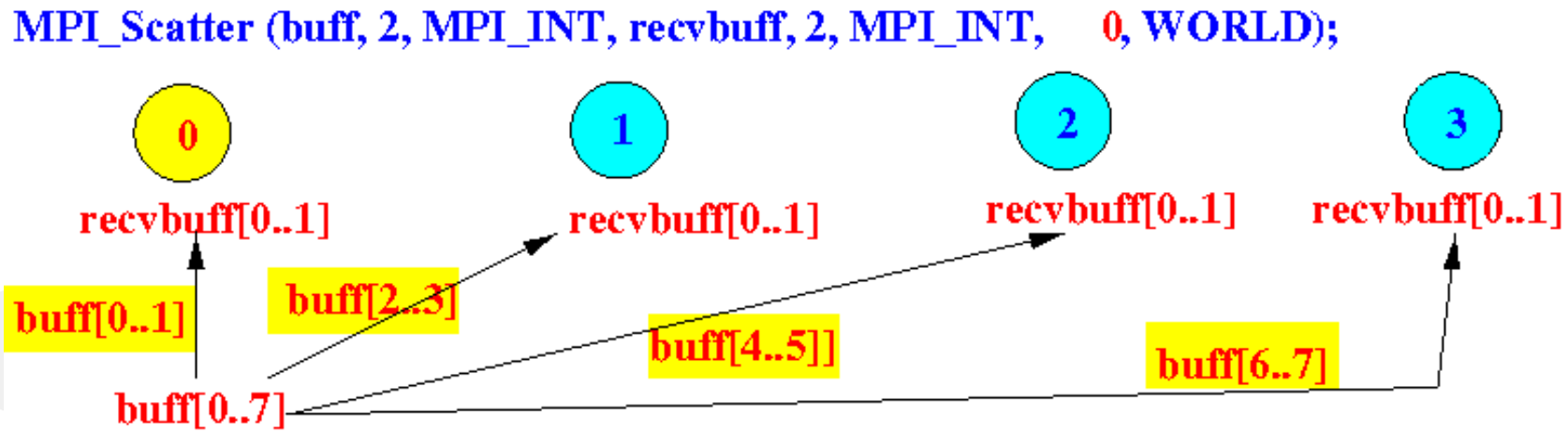
```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm);
```



Operaciones Colectivas

MPI_Scatter: distribuye *sendcnt* elementos de *buff* de tipo *sendtype* desde el proceso *root* a todos los procesos del comunicador *comm*.

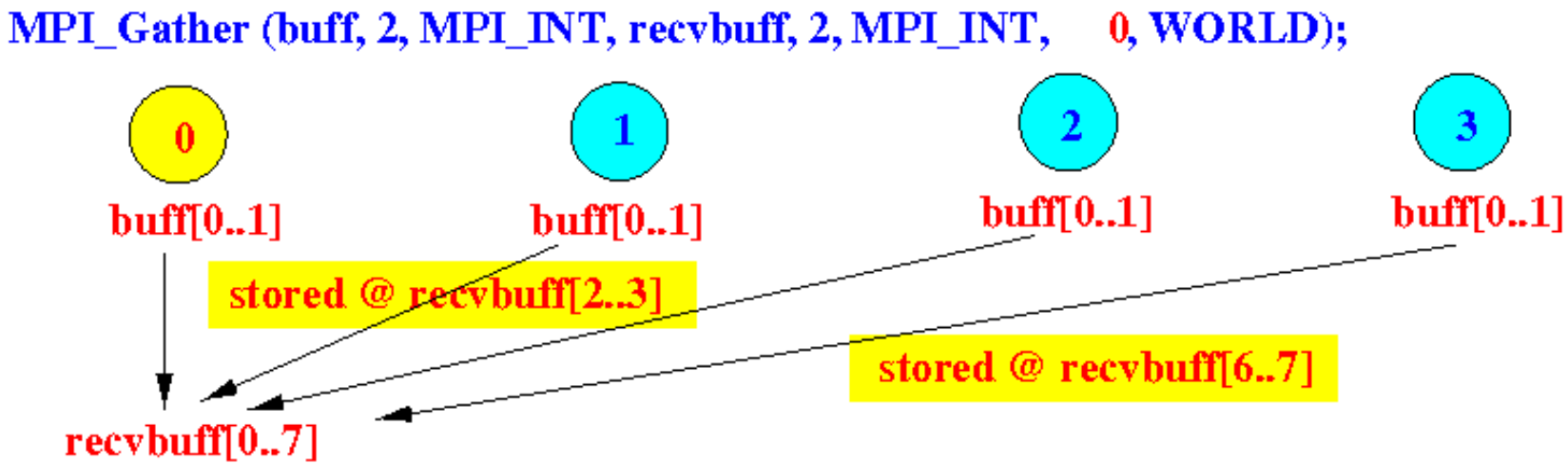
```
MPI_Scatter(void *buff, int sendcnt, MPI_Datatype sendtype,
  void *recvbuff, int recvcnt, MPI_Datatype recvtype, int root,
  MPI_Comm comm);
```



Operaciones Colectivas

MPI_Gather: recibe en el proceso *root*, en *recvbuff* , *recvcnt* elementos de tipo *recvtype* desde todos los procesos del comunicador *comm*.

```
MPI_Gather(void *buff, int sendcnt, MPI_Datatype sendtype,
  void *recvbuff, int recvcnt, MPI_Datatype recvtype, int root,
  MPI_Comm comm);
```



Operaciones Colectivas

MPI_Reduce: realiza una reducción todos a uno, reduciendo los datos de *buff*, *count* elementos de tipo *datatype*, y guardando el resultado en *recvbuff* del proceso *root*. Operaciones *op* disponibles: MPI_{MAX,MIN,SUM,PROD}, MPI_{LAND,LOR,LEXOR}, MPI_{BAND,BOR,BXOR} o MPI_{MAXLOC,MINLOC}

```
int MPI_Reduce(void *buff, void *recvbuff, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```