

PROGRAMACIÓN DE SISTEMAS DISTRIBUIDOS

Simon Pickin
Alberto Núñez

MPI

Índice

2

- Introducción
- Memoria compartida y distribuida
- Modelo de ejecución (SPMD)
- Funciones de MPI
 - ▣ Funciones de entorno
 - ▣ Funciones punto-a-punto
 - ▣ Funciones de sincronización
 - ▣ Funciones colectivas

Introducción

3

- MPI es un **estándar** para una biblioteca de **paso de mensajes**
- El objetivo es **comunicar procesos** en ordenadores remotos
 - ▣ También funciona con procesos ejecutados en el mismo ordenador
- Actualmente hay varias implementaciones:
 - ▣ Open MPI (<http://www.open-mpi.org/>)
 - ▣ MPICH (<http://www.mpich.org/>)
 - ▣ MVAPICH (<http://mvapich.cse.ohio-state.edu/>)
 - ▣ IBM Platform MPI (<http://www.ibm.com/systems/es/platformcomputing/products/mpi/>)
 - ▣ Previamente “Platform MPI” (IBM adquirió Platform en 2012)
 - ▣ derivado de “Scali MPI” y “HP MPI” (Platform adquirió Scali en 2007 y HP-MPI en 2009)
 - ▣ Intel MPI (<https://software.intel.com/es-es/intel-mpi-library>)

Introducción

4

- MPI 2.0 tiene más de 100 funciones
 - ▣ Utilizaremos sólo **las más importantes**
- La unidad básica de ejecución de MPI son los **procesos**
- Cada uno tiene su **espacio de memoria independiente**
- ¿Cómo se intercambian información entre ellos?
 - ▣ Paso de mensajes
- Cada proceso tiene un identificador único
 - ▣ Rank

Paralelización. La Ley de Amdahl (1967)

5

- Ejemplo: aumento de velocidad con el paralelismo
 - ▣ Se tienen que pintar 10 habitaciones
 - Una habitación es el doble de grande que las demás
 - ▣ Un solo pintor tarda 11 unidades de tiempo
 - 9 habitaciones sencillas + 1 habitación doble
 - ▣ 10 pintores, uno por habitación, tardan 2 unidades de tiempo
 - 9 pintores necesitan 1 unidad de tiempo para terminar
 - 1 pintor necesita 2 unidades de tiempo para terminar
- Ley de Amdahl: $\text{Aceleración} = 1 / ((1-p) + p/n)$
 - ▣ p es la parte del cálculo que se puede paralelizar
 - ▣ n es el número de procesadores
 - ▣ En el caso de los pintores $p = 10/11$, $n = 10$, aumento max. = 5,5
 - ▣ Cuantifica el máximo aumento teórico con adición de más paralelismo

Paralelización. Ejemplo 1

6

- Ejemplo 1: multiplicación de dos matrices cuadradas (tamaño N)
 - ▣ Se envía uno de los matrices a todos los procesos
 - ▣ Se envía un trozo distinto del otro matriz a cada proceso
 - Para simplificar, suponemos que N es divisible por el número de procesos
 - Nos interesa la proporción $P = \text{tiempo de comunicación} / \text{tiempo de cálculo}$, dado:
 - $T_{\text{comm}} = \text{tiempo de una comunicación}$, $T_{\text{calc}} = \text{tiempo de un cálculo}$
- Cálculo de la proporción de interés
 - ▣ Número de cálculos = $O(N^3)$
 - ▣ Número de comunicaciones = $N^2 + P * (N^2/P) + N^2$
 - ▣ $P = 3 * N^2 * T_{\text{comm}} / N^3 * T_{\text{calc}} = (3 * T_{\text{comm}} / T_{\text{calc}}) * (1/N)$
- Conclusiones:
 - ▣ Se gana en rendimiento con la paralelización (P tiende a 0 con aumento de N)

Paralelización. Ejemplo 2

7

- Ejemplo 2: adición de dos matrices cuadradas (tamaño N)
 - ▣ Se envía uno de los matrices a todos los procesos
 - ▣ Se envía un trozo distinto del otro matriz a cada proceso
 - Para simplificar, suponemos que N es divisible por el número de procesos
 - Nos interesa la proporción: $P = \text{tiempo de comunicación} / \text{tiempo de cálculo}$, dado:
 - $T_{\text{comm}} = \text{tiempo de una comunicación}$, $T_{\text{calc}} = \text{tiempo de un cálculo}$
- Cálculo de la proporción de interés
 - ▣ Número de cálculos = N^2
 - ▣ Número de comunicaciones = $N^2 + P * (N^2/P) + N^2$
 - ▣ $P = 3 * N^2 * T_{\text{comm}} / N^2 * T_{\text{calc}} = 3 * T_{\text{comm}} / T_{\text{calc}}$
- Conclusiones:
 - ▣ No se gana en rendimiento con la paralelización (P es fija)

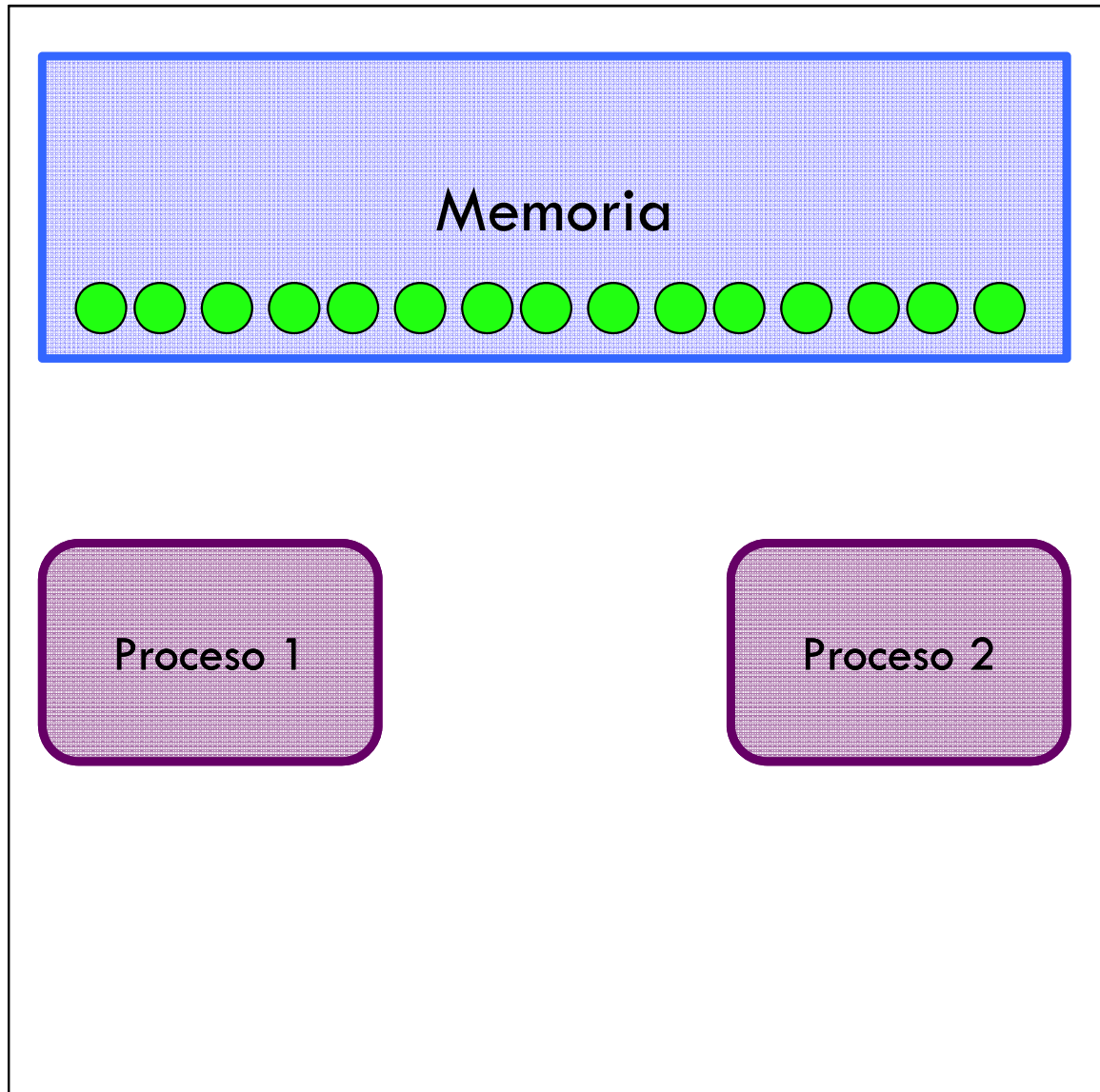
Índice

8

- Introducción
- Memoria compartida y distribuida
- Modelo de ejecución (SPMD)
- Funciones de MPI
 - ▣ Funciones de entorno
 - ▣ Funciones punto-a-punto
 - ▣ Funciones de sincronización
 - ▣ Funciones colectivas

Memoria compartida

9

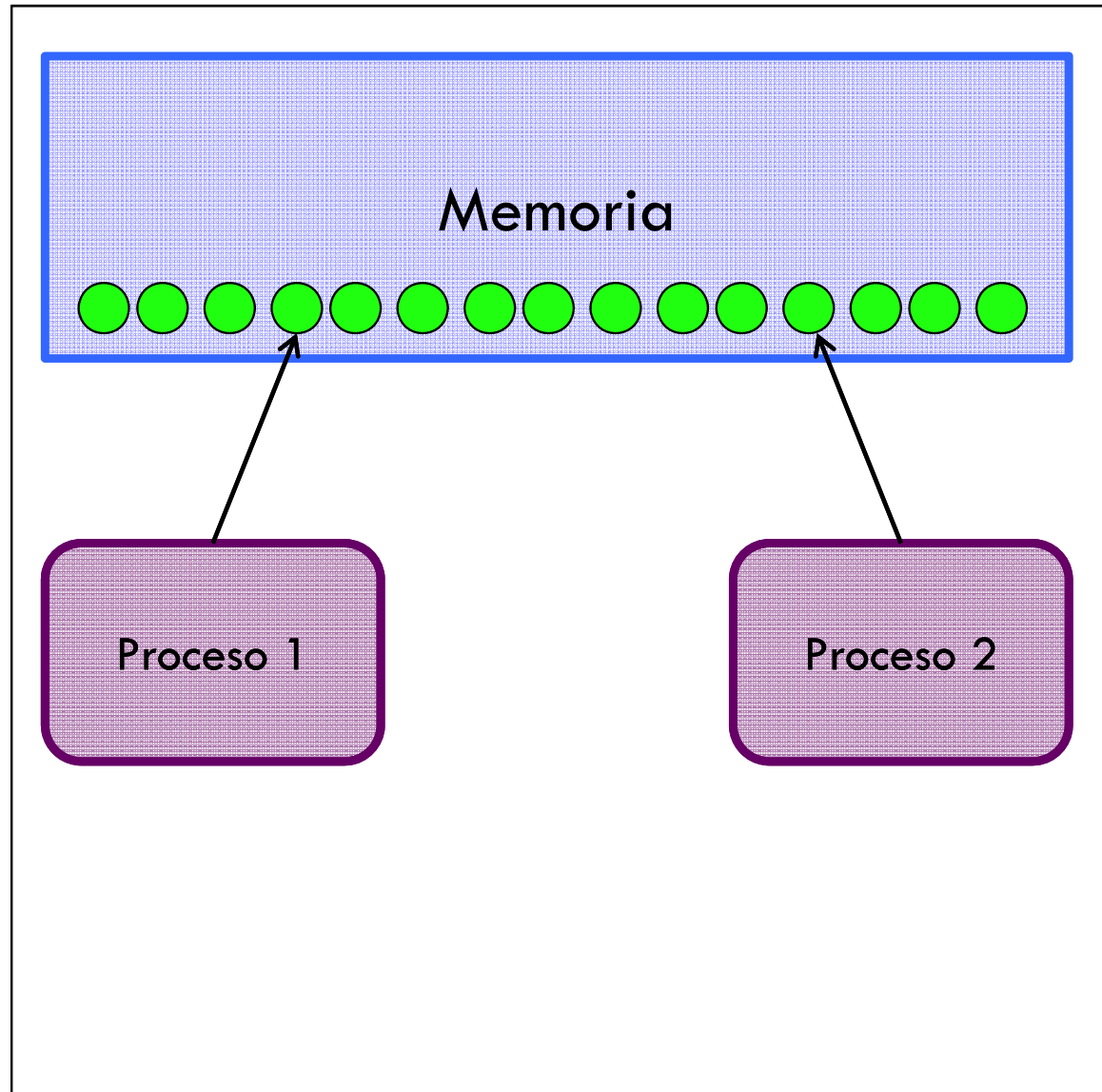


1 nodo
Memoria compartida

● Posición de memoria
→ Lectura
→ Escritura

Memoria compartida

10



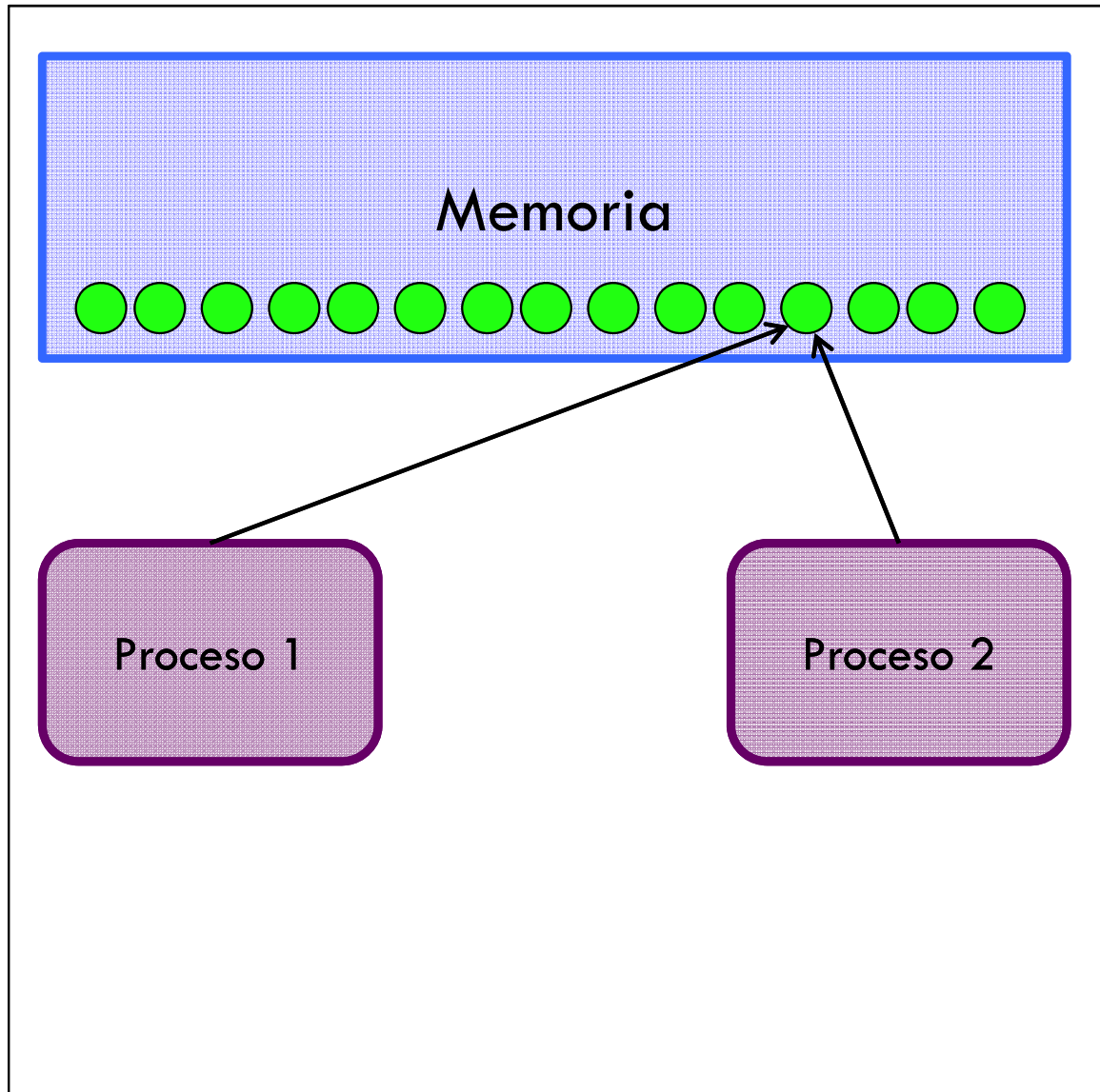
1 nodo
Memoria compartida

● Posición de memoria
→ Lectura
→ Escritura

Lecturas en paralelo

Memoria compartida

11



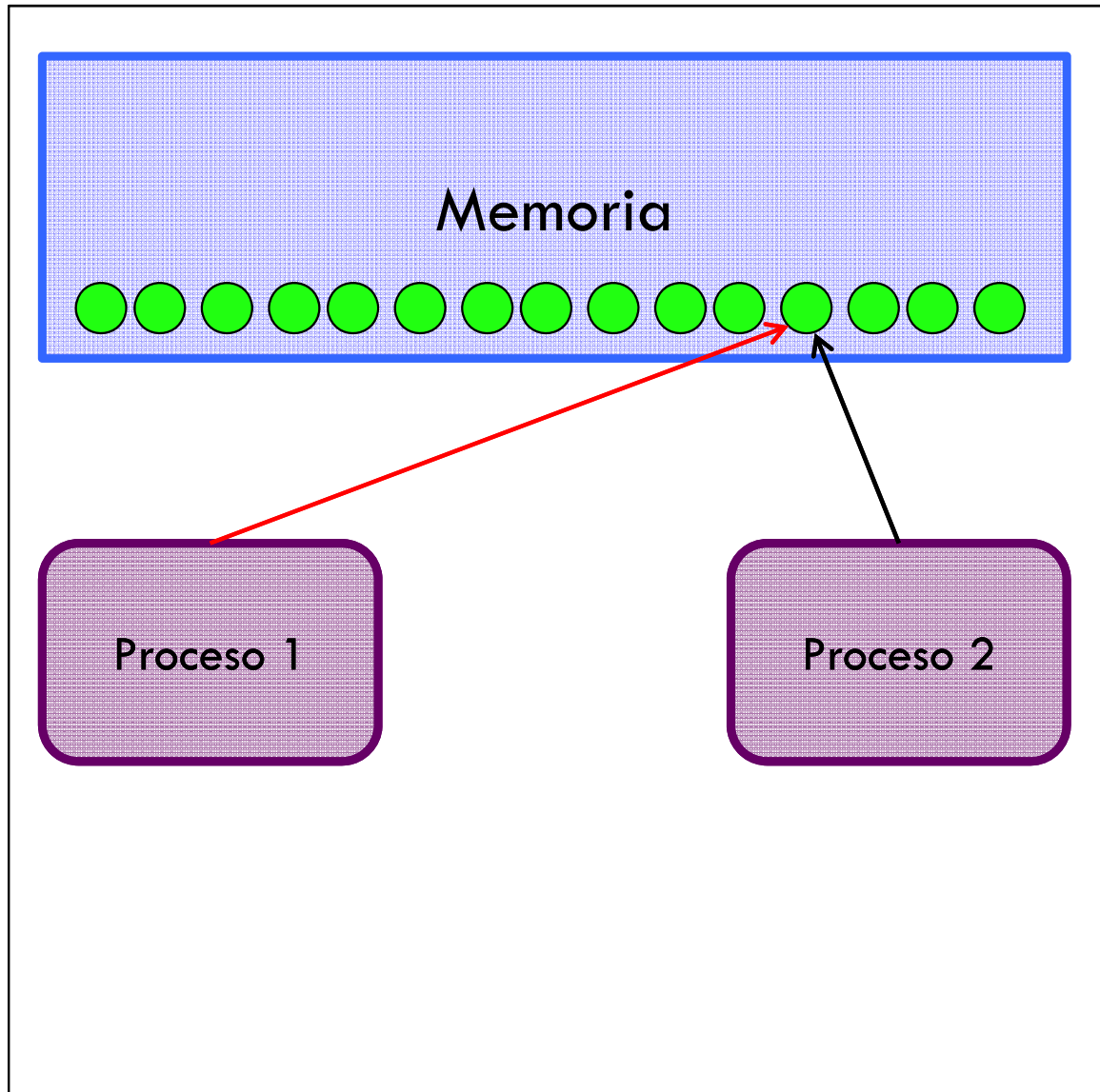
1 nodo
Memoria compartida

● Posición de memoria
→ Lectura
→ Escritura

Lecturas en paralelo

Memoria compartida

12



1 nodo
Memoria compartida

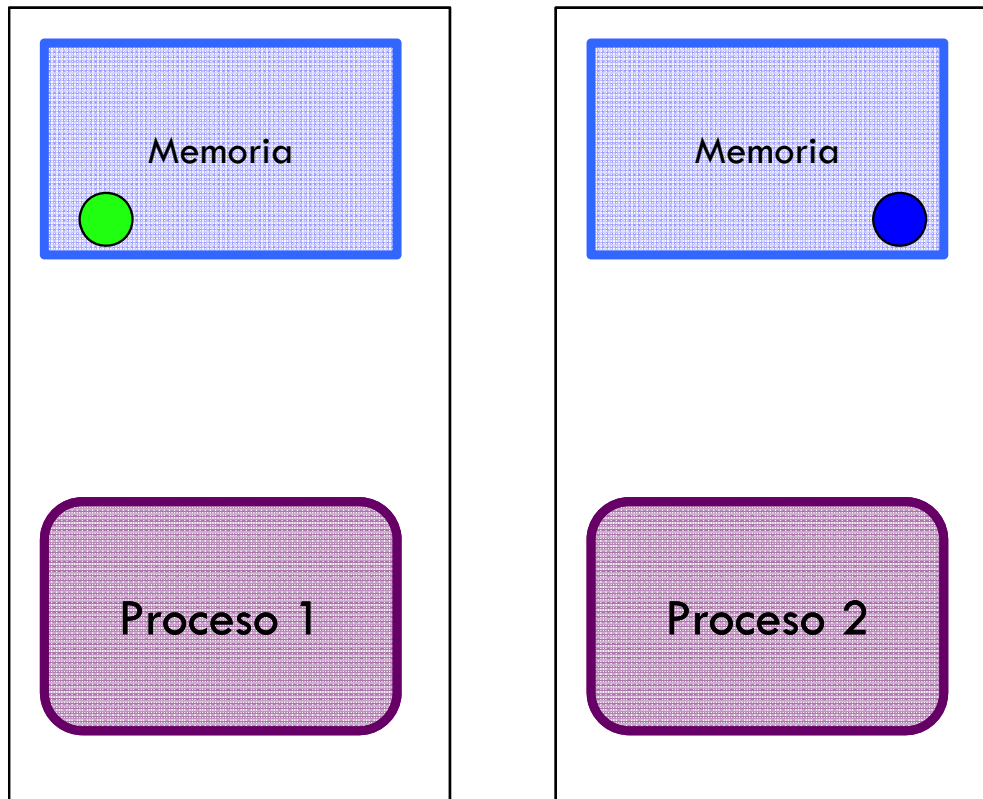
● Posición de memoria
→ Lectura
→ Escritura

Lecturas en paralelo

Escrituras con exclusión mutua
- Mutex

Memoria distribuida

13



n nodos

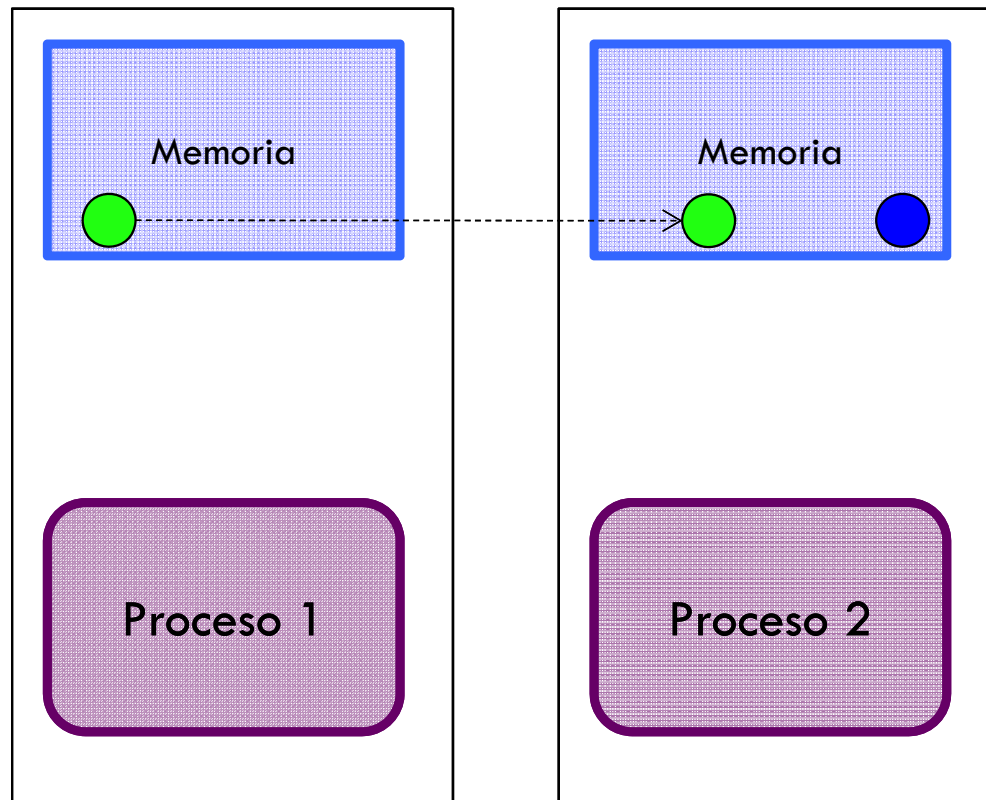
Memoria distribuida

¿Cómo comparten datos
varios procesos?

Un proceso sólo tiene acceso
a su espacio de memoria

Memoria distribuida

14



n nodos

Memoria distribuida

¿Cómo comparten datos
varios procesos?

Un proceso sólo tiene acceso
a su espacio de memoria

El proceso 1 le envía al
proceso 2 datos

Hay que utilizar la red

Índice

15

- Introducción
- Memoria compartida y distribuida
- Modelo de ejecución (SPMD)
- Funciones de MPI
 - ▣ Funciones de entorno
 - ▣ Funciones punto-a-punto
 - ▣ Funciones de sincronización
 - ▣ Funciones colectivas

Flynn's taxonomy

16

- SISD (*Single Instruction, Single Data stream*)
 - ▣ P.ej. MIPS architecture
- SIMD (*Single Instruction, Multiple Data streams*)
 - ▣ P.ej. GPU
- MISD (*Multiple Instruction, Single Data stream*)
 - ▣ Poco común, p.ej. para tolerancia a fallos (Arianne, Space Shuttle,...)
- MIMD (*Multiple Instruction, Multiple Data streams*)
 - ▣ SPMD (*Single Program, Multiple Data*)
 - Arquitectura paralela más común
 - ▣ MPMD (*Multiple Programs, Multiple Data*)
 - P.ej. Sony PS3 (arquitectura “cell” con procesadores SPU/PPU)

Modelo de ejecución (SPMD)

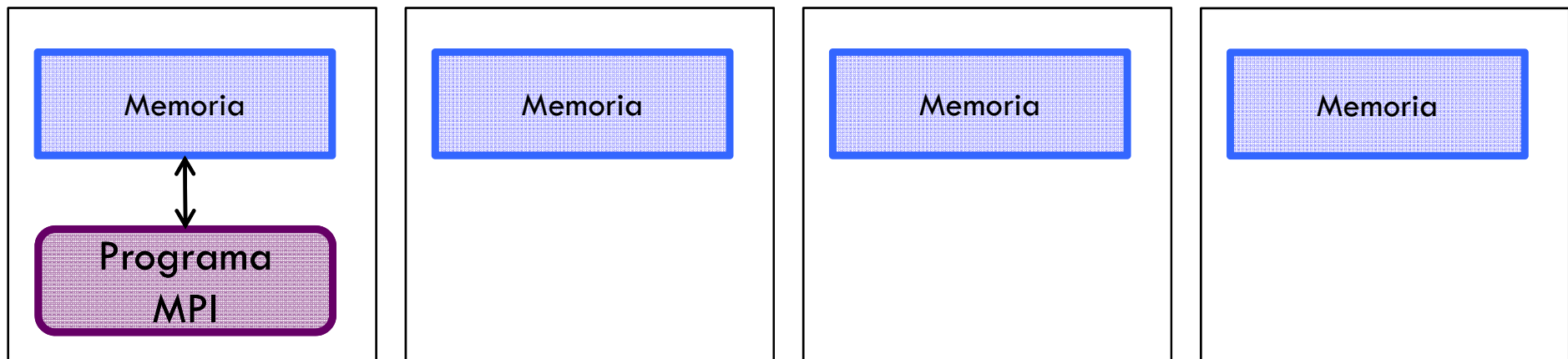
17

- Single Program Multiple Data (SPMD)
 - **1 programa** ejecutado en paralelo
 - El mismo programa **se copia** a todos los nodos
 - Se ejecutan **todos en paralelo**
 - Cada proceso comienza la ejecución **en el mismo punto**.
 - ¿Cómo **diferenciamos el comportamiento** de cada proceso?
 - Cada uno tendrá un **flujo de ejecución** en el programa
 - Sabemos quién es cada proceso por su ID (**Rank**)

Modelo de ejecución (SPMD)

18

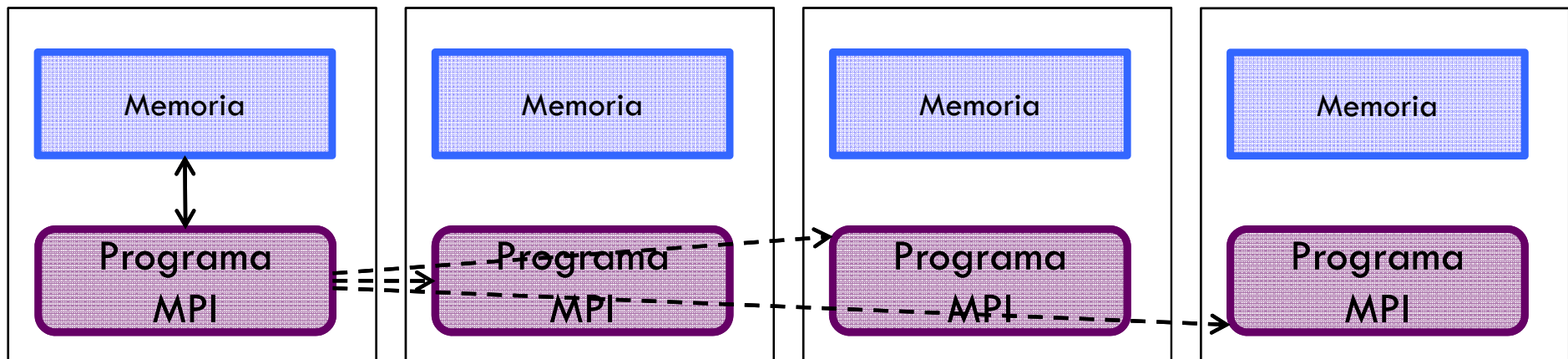
- Single Program Multiple Data (SPMD)
 - La ejecución se lanza desde un nodo
 - El programa se copia a todos los nodos involucrados
 - Se ejecuta cada programa en su nodo



Modelo de ejecución (SPMD)

19

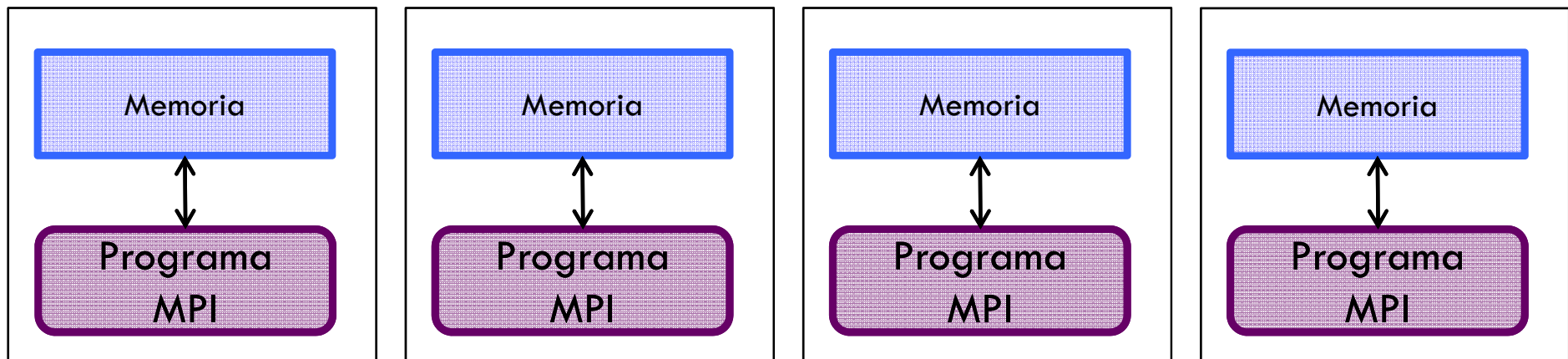
- Single Program Multiple Data (SPMD)
 - La ejecución se lanza desde un nodo
 - El programa se copia a todos los nodos involucrados
 - Se ejecuta cada programa en su nodo



Modelo de ejecución (SPMD)

20

- Single Program Multiple Data (SPMD)
 - ▣ La ejecución se lanza desde un nodo
 - ▣ El programa se copia a todos los nodos involucrados
 - ▣ Se ejecuta cada programa en su nodo



Hello World!

21

- Antes de empezar con MPI...
 - ▣ El Hello World! Ejecutado en n ordenadores a la vez!

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

int main(int argc, char *argv[]) {
    int myrank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Hola soy el proceso %d de %d\n", myrank, size);

    MPI_Finalize();
    exit(0);
}
```

Índice

22

- Introducción
- Memoria compartida y distribuida
- Modelo de ejecución (SPMD)
- Funciones de MPI
 - ▣ Funciones de entorno
 - ▣ Funciones punto-a-punto
 - ▣ Funciones de sincronización
 - ▣ Funciones colectivas

Funciones de MPI

23

- Funciones de entorno
 - ▣ MPI_INIT
 - ▣ MPI_COMM_SIZE
 - ▣ MPI_COMM_RANK
 - ▣ MPI_FINALIZE
 - ▣ MPI ABORT
- Comunicación punto a punto
- Funciones de sincronización
- Comunicaciones colectivas

Índice

24

- Introducción
- Memoria compartida y distribuida
- Modelo de ejecución (SPMD)
- Funciones de MPI
 - ▣ Funciones de entorno
 - ▣ Funciones punto-a-punto
 - ▣ Funciones de sincronización
 - ▣ Funciones colectivas

Funciones de MPI

Entorno

25

- MPI agrupa los procesos en “comunicadores”
- Los procesos que intercambian mensajes comparten comunicador
- El comunicador **MPI_COMM_WORLD** agrupa todos los procesos
- Generalmente se utiliza este comunicador

MPI_INIT

26

- **int MPI_Init(int *argc, char **argv);**
 - Establece un entorno de MPI
 - Se tiene que invocar **antes de cualquier otra** llamada a MPI
 - Sólo se puede invocar **una vez por proceso**

MPI_FINALIZE

27

- **int MPI_Finalize(void);**
 - ▣ Termina el procesamiento de MPI
 - ▣ Tiene que ser la **última llamada** MPI invocada
 - ▣ **Libera** los recursos utilizados por MPI

MPI_COMM_SIZE

28

- **int MPI_Comm_size(MPI_Comm comm, int* size);**
 - ▣ Devuelve el número de procesos relacionados con el comunicador
 - comm: comunicador (Parámetro de entrada)
 - size: número de procesos en este comunicador (parámetro de salida)

MPI_COMM_RANK

29

- **int MPI_Comm_rank(MPI_Comm comm, int* rank);**
 - ▣ Devuelve el **ID (rank) del proceso** asociado a un comunicador
 - ▣ Este ID toma valores del 0 a (size-1)
 - comm: comunicador (Parámetro de entrada)
 - rank: rank del proceso llamante en el comunicador (parámetro de salida)

MPI_ABORT

30

- **int MPI_Abort(MPI_Comm comm, int errorcode);**
 - ▣ Fuerza la finalización de todos los procesos MPI

Esquema de programas MPI

31

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

. . .

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Algún tipo de algoritmo. . .

    MPI_Finalize();
}
```

Índice

32

- Introducción
- Memoria compartida y distribuida
- Modelo de ejecución (SPMD)
- Funciones de MPI
 - ▣ Funciones de entorno
 - ▣ Funciones punto-a-punto
 - ▣ Funciones de sincronización
 - ▣ Funciones colectivas

Funciones de MPI

33

- Funciones de entorno
- Comunicación punto a punto
 - ▣ MPI_SEND
 - ▣ MPI_RECV
 - ▣ MPI_ISEND
 - ▣ MPI_Irecv
- Funciones de sincronización
- Comunicaciones colectivas

Funciones de MPI

34

- Comunicación punto a punto
 - ▣ Hay **un proceso emisor** y **un proceso receptor** del mensaje
- Hay dos tipos de comunicación punto a punto:
 - ▣ **Síncrona:** El proceso emisor **espera** a que se realice el envío del mensaje.
 - Comunicación bloqueante
 - MPI_Send, MPI_Recv
 - ▣ **Asíncrona:** El proceso emisor envía el mensaje y continúa su ejecución **sin asegurarse** de que el proceso receptor haya solicitado el mensaje.
 - Comunicación no bloqueante
 - MPI_Isend, MPI_Irecv

Funciones de MPI

35

Proceso A



Proceso B



- Proceso A se ejecuta
- Proceso B se ejecuta

Funciones de MPI

36

Proceso A



MPI_Send

Proceso B



- Proceso A se ejecuta
- Proceso B se ejecuta
- Proceso A envía un mensaje a B
- Proceso B continúa se ejecución

Funciones de MPI

37

Proceso A



MPI_Send

Proceso B

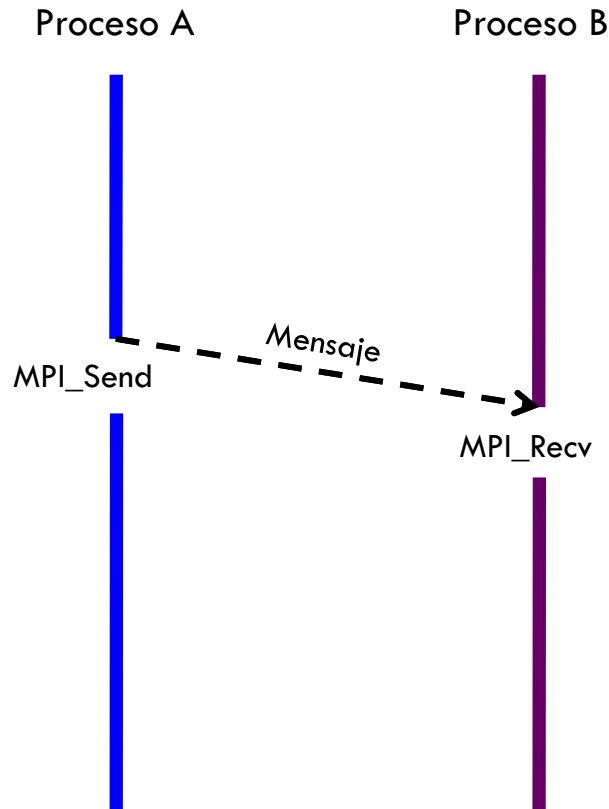


MPI_Recv

- Proceso A se ejecuta
- Proceso B se ejecuta
- Proceso A envía un mensaje a B
- Proceso B continúa se ejecución
- Proceso B espera el mensaje de A

Funciones de MPI

38



- Proceso A se ejecuta
- Proceso B se ejecuta
- Proceso A envía un mensaje a B
- Proceso B continúa se ejecución
- Proceso B espera el mensaje de A
- Una vez sincronizados, cada proceso continúa su ejecución.

Tipos de datos

39

□ Tipos de datos MPI_Datatype:

- MPI_CHAR
- MPI_SHORT
- MPI_INT
- MPI_LONG
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED_SHORT
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG
- MPI_FLOAT
- MPI_DOUBLE
- MPI_LONG_DOUBLE
- MPI_BYTE
- MPI_PACKED

MPI_Send

40

```
int MPI_Send( void* buf,           /* input */
              int count,          /* input */
              MPI_Datatype datatype, /* input */
              int destination,    /* input */
              int tag,            /* input */
              MPI_Comm comm );    /* input */
```

También *MPI_Bsend*:
bloquea hasta que los
datos han sido copiados
desde el bufer de la
aplicación hasta el bufer
de salida del sistema

- Envía un mensaje de forma síncrona
- Este mensaje lo puede recibir un proceso mediante *MPI_Recv* o *MPI_IRecv*
 - ▣ buf: puntero a los datos que se envían (mensaje)
 - ▣ count: número de elementos en el mensaje
 - ▣ datatype: tipo de los datos enviados
 - ▣ destination: *rank* del proceso al que se le envía el mensaje
 - ▣ tag: etiqueta que puede servir de ID único del mensaje (en particular, para ligar envío y recepción)
 - ▣ comm: comunicador

MPI_Recv

41

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

- Recibe un mensaje de forma síncrona
- Este mensaje lo puede enviar un proceso mediante MPI_Send o MPI_Isend
 - ▣ buf: puntero donde se van a almacenar los datos recibidos (mensaje)
 - ▣ count: maximum número de elementos en el mensaje
 - ▣ datatype: tipo de los datos enviados
 - ▣ source: *rank* del proceso del que se espera recibir el mensaje
 - MPI_ANY_SOURCE indica que se puede recibir un mensaje de cualquier proceso
 - ▣ tag: etiqueta que puede servir de ID único del mensaje (en particular, para ligar envío y recepción)
 - ▣ comm: Comunicador
 - ▣ MPI_Status: contiene source y tag (puede haber usado MPI_ANY_SOURCE y MPI_ANY_TAG) y el count (el parámetro solo contiene el máximo posible) y, posiblemente otras informaciones)

Ejemplo

42

```
#include "mpi.h"

int rank, nproc;

int main (int argc, char* argv[] ) {
    int isbuf, irbuf;
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);

    if (rank == 0){
        isbuf = 9;
        MPI_Send(&isbuf, 1, MPI_INTEGER, 1, 1, MPI_COMM_WORLD);
    }
    else if(rank == 1) {
        MPI_Recv( &irbuf, 1, MPI_INTEGER, 0, 1, MPI_COMM_WORLD, &status);
        printf( "%d\n", irbuf );
    }
    MPI_Finalize();
}
```

Ejemplo

43

```
#include "mpi.h"
```

```
int rank, nproc;
```

```
int main (int argc, char* argv[] ) {  
    int isbuf, irbuf;  
    MPI_Status status;
```

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

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

```
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);
```

```
    if (rank == 0){
```

```
        isbuf = 9;
```

```
        MPI_Send(&isbuf, 1, MPI_INTEGER, 1, 1, MPI_COMM_WORLD);
```

```
    }
```

```
    else if(rank == 1) {
```

```
        MPI_Recv( &irbuf, 1, MPI_INTEGER, 0, 1, MPI_COMM_WORLD, &status);
```

```
        printf( "%d\n", irbuf );
```

```
    }
```

```
    MPI_Finalize();
```

```
}
```

Mensaje

Número de
elementos

Tipo

Receptor

Etiqueta

Comunicador

Ejemplo

44

```
#include "mpi.h"
```

```
int rank, nproc;
```

```
int main (int argc, char* argv[] ) {  
    int isbuf, irbuf;  
    MPI_Status status;
```

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

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

```
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);
```

```
    if (rank == 0){
```

```
        isbuf = 9;
```

```
        MPI_Send(&isbuf, 1, MPI_INTEGER, 1, 1, MPI_COMM_WORLD);
```

```
    }
```

```
    else if(rank == 1){
```

```
        MPI_Recv( &irbuf, 1, MPI_INTEGER, 0, 1, MPI_COMM_WORLD, &status);
```

```
        printf( "%d\n", irbuf );
```

```
    }
```

```
    MPI_Finalize();
```

```
}
```

Mensaje

Número de
elementos

Tipo

Emisor

Etiqueta

Comunicador

Estado

Otro ejemplo

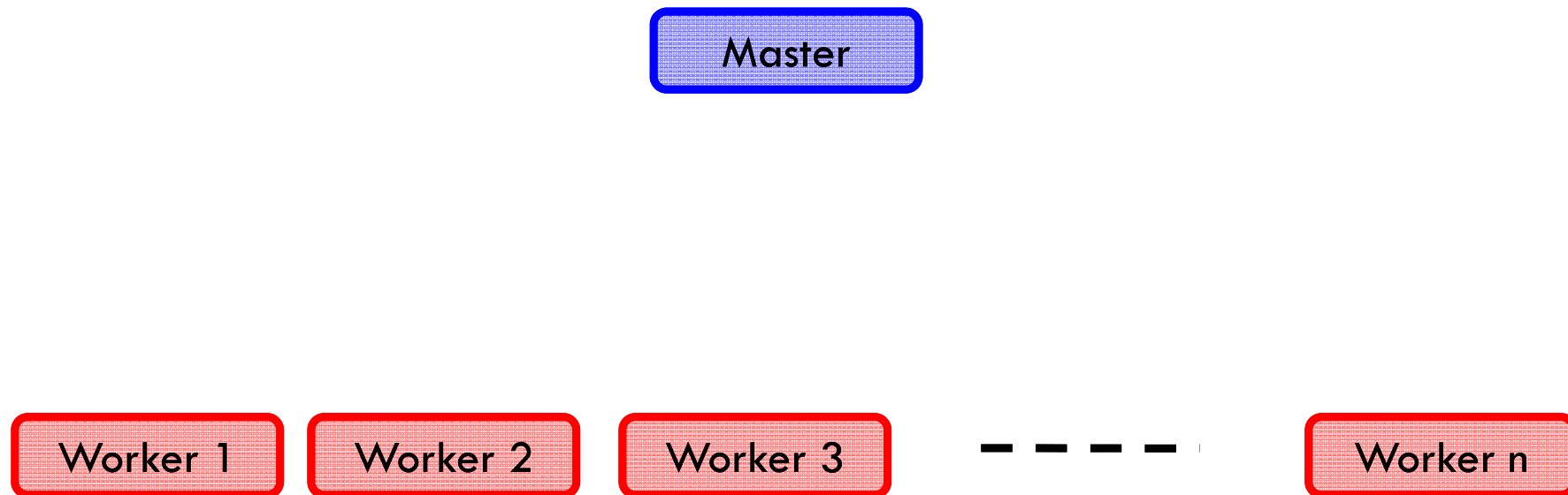
45

- Un proceso (**master**) envía un número al resto de los procesos (**worker**)
 - Pero el master también puede hacer su parte del cálculo
- Cada proceso secundario eleva al cuadrado el número recibido
 - Devuelven el resultado al proceso maestro
- El proceso maestro suma todos los números recibidos

Otro ejemplo

46

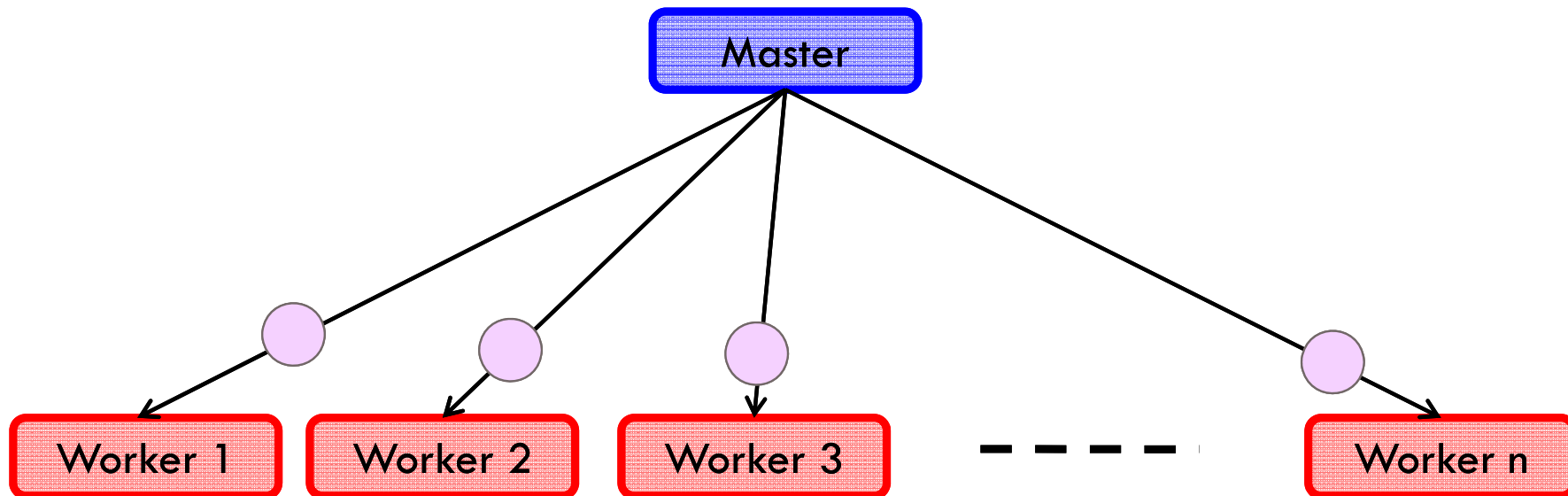
- Un proceso (master) envía un número al resto de los procesos (worker)
- Cada proceso secundario eleva al cuadrado el número recibido
 - ▣ Devuelven el resultado al proceso maestro
- El proceso maestro suma todos los números recibidos



Otro ejemplo

47

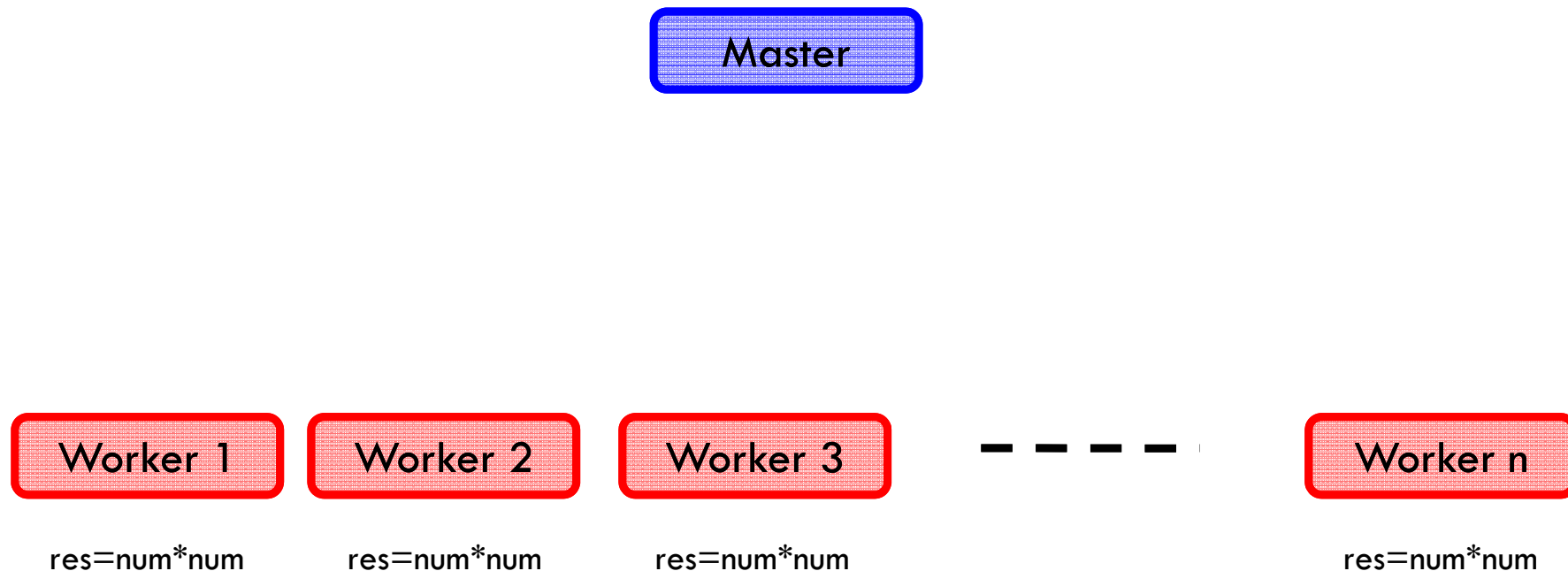
- Un proceso (**master**) envía un número al resto de los procesos (**worker**)
- Cada proceso secundario eleva al cuadrado el número recibido
 - ▣ Devuelven el resultado al proceso maestro
- El proceso maestro suma todos los números recibidos



Otro ejemplo

48

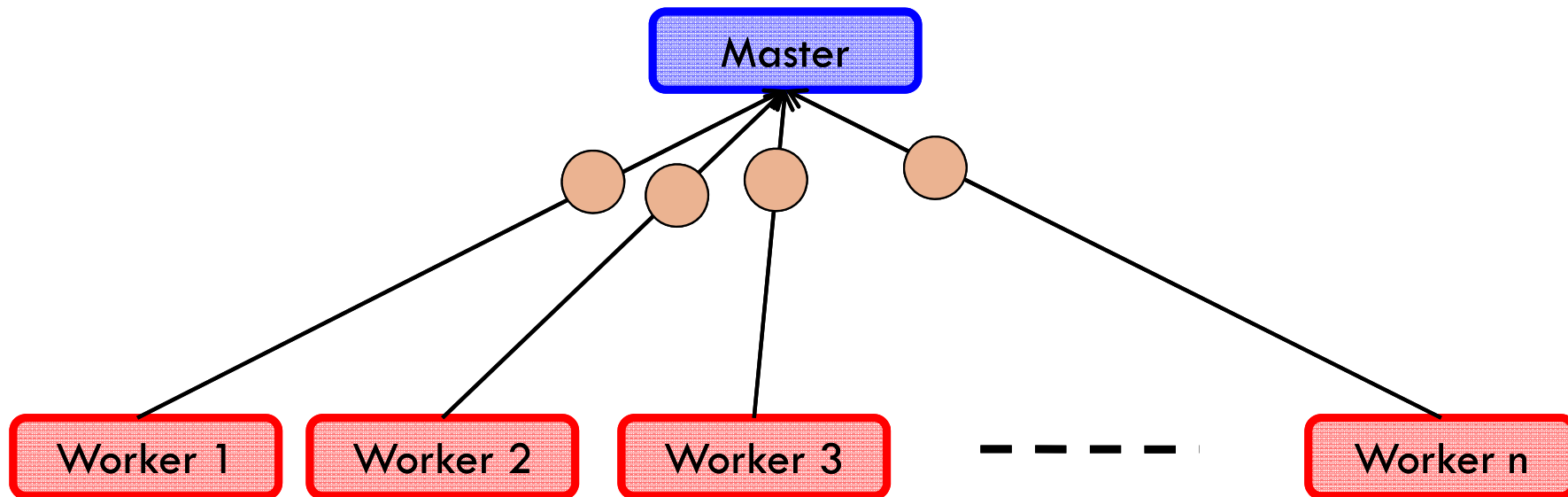
- Un proceso (master) envía un número al resto de los procesos (worker)
- Cada proceso secundario eleva al cuadrado el número recibido
 - ▣ Devuelven el resultado al proceso maestro
- El proceso maestro suma todos los números recibidos



Otro ejemplo

49

- Un proceso (master) envía un número al resto de los procesos (worker)
- Cada proceso secundario eleva al cuadrado el número recibido
 - ▣ Devuelven el resultado al proceso maestro
- El proceso maestro suma todos los números recibidos



Otro ejemplo

50

- Un proceso (master) envía un número al resto de los procesos (worker)
- Cada proceso secundario eleva al cuadrado el número recibido
 - ▣ Devuelven el resultado al proceso maestro
- El proceso maestro suma todos los números recibidos

Master

$$\sum_{i=1}^n num[worker_i]$$

Worker 1

Worker 2

Worker 3

— — — —

Worker n

Otro ejemplo

51

```
#include "mpi.h"

int rank, numProcesos, resultado, num;

int main (int argc, char* argv[] ) {
    int *numbers, *square;
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numProcesos );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    // Inicializamos los arrays numbers y square
    ...
}
```

Otro ejemplo

52

```
if (rank == 0){
    for (i=1; i<numProcesos; i++)
        MPI_Send(&(numbers[i-1]), 1, MPI_INTEGER, i, 1, MPI_COMM_WORLD);

    for (i=1; i<numProcesos; i++)
        MPI_Recv(&(square[i-1]), 1, MPI_INTEGER, MPI_ANY_SOURCE, 1,
MPI_COMM_WORLD);

    for (i=1; i<numProcesos; i++)
        resultado += square[i-1];

    printf ("El resultado es:%d\n", resultado);
}

else{
    MPI_Recv( &num, 1, MPI_INTEGER, 0, 1, MPI_COMM_WORLD, &status);
    num = num*num;
    MPI_Send(&num, 1, MPI_INTEGER, 0, 1, MPI_COMM_WORLD);
}

MPI_Finalize();
}
```

MPI_Isend

53

```
int MPI_Isend( void* buf,           /* input */
               int count,           /* input */
               MPI_Datatype datatype, /* input */
               int destination,      /* input */
               int tag,              /* input */
               MPI_Comm comm,        /* input */
               MPI_Request *request); /* output */
```

- Envía un mensaje de forma asíncrona
- Los parámetros son los mismos que MPI_Send añadiendo MPI_Request
 - ▣ Identificador que se puede usar posteriormente para saber si la recepción ha terminado mediante llamadas a:
 - MPI_Test(): Devuelve un *flag* que indica si la operación se ha completado
 - También: MPI_Testany, MPI_Testall, MPI_Testsome
 - MPI_Wait(): Devuelve el control de la ejecución si la operación se ha completado, espera a que finalice en caso contrario.
 - También: MPI_Waitany, MPI_Waitall, MPI_Waitsome

MPI_IRecv

54

```
int MPI_IRecv( void* buf,                /* output */
               int count,                /* input */
               MPI_Datatype datatype,    /* input */
               int destination,          /* input */
               int tag,                  /* input */
               MPI_Comm comm,            /* input */
               MPI_Request *request);    /* output */
```

- Recibe un mensaje de forma asíncrona
- Los parámetros son los mismos que `MPI_Recv` añadiendo `MPI_Request`
 - ▣ Identificador que se puede usar posteriormente para saber si la recepción ha terminado mediante llamadas a:
 - `MPI_Test()`: Devuelve un *flag* que indica si la operación se ha completado
 - También: `MPI_Testany`, `MPI_Testall`, `MPI_Testsome`
 - `MPI_Wait()`: Devuelve el control de la ejecución si la operación se ha completado, espera a que finalice en caso contrario.
 - También: `MPI_Waitany`, `MPI_Waitall`, `MPI_Waitsome`

Índice

55

□ Funciones de MPI

- ▣ Funciones de entorno

- ▣ Funciones punto-a-punto

- ▣ Funciones de sincronización

 - MPI_BARRIER

- ▣ Funciones colectivas

MPI_Barrier

56

- `int MPI_Barrier(MPI_Comm comm);` `/* input */`
 - ▣ Se utiliza para sincronizar a TODOS los procesos asociados a un determinado comunicador.
 - ▣ Hasta que TODOS los procesos no ejecuten `MPI_Barrier`, no se continúa la ejecución

MPI_Barrier

57

- `int MPI_Barrier(MPI_Comm comm);` `/* input */`
 - ▣ Se utiliza para sincronizar a TODOS los procesos asociados a un determinado comunicador.
 - ▣ Hasta que TODOS los procesos no ejecuten `MPI_Barrier`, no se continúa la ejecución

proceso 1



proceso 2



proceso 3



`MPI_Barrier`

proceso 4

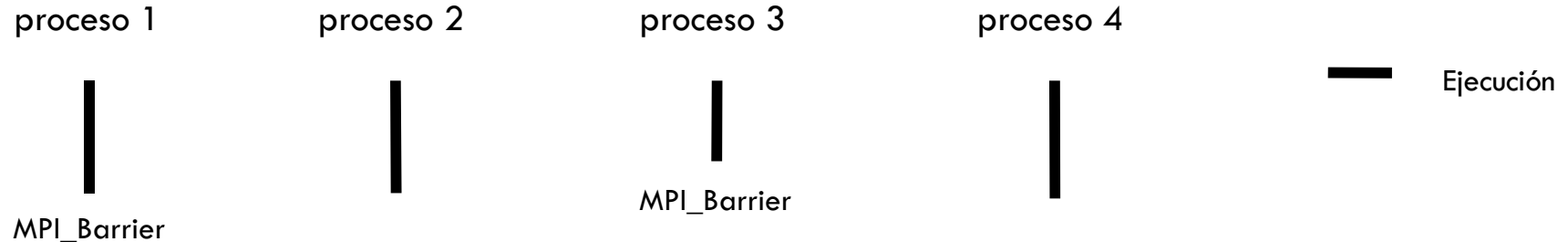


Ejecución

MPI_Barrier

58

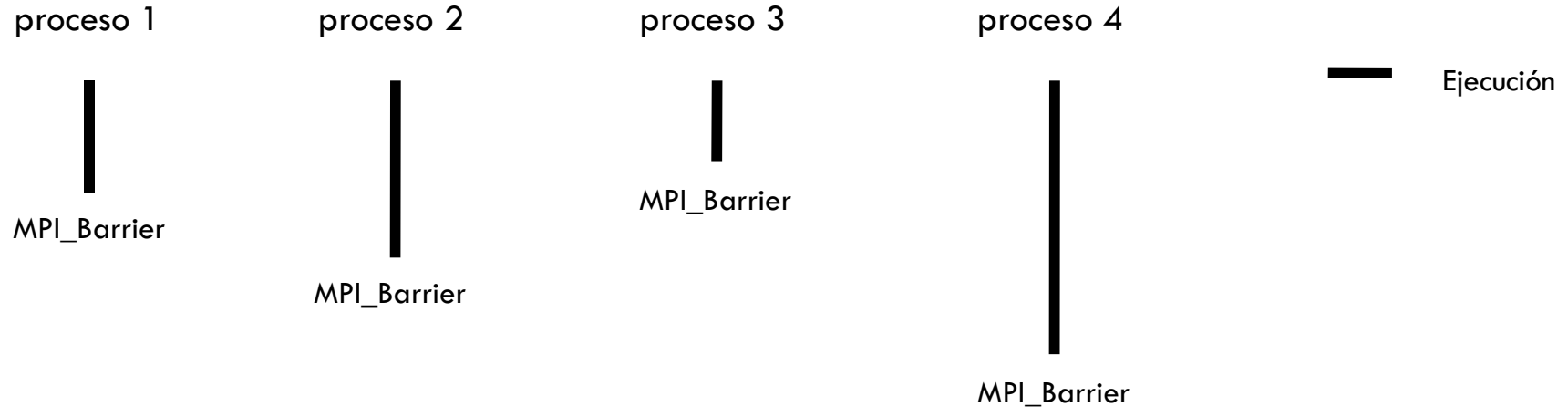
- `int MPI_Barrier(MPI_Comm comm);` `/* input */`
 - ▣ Se utiliza para sincronizar a TODOS los procesos asociados a un determinado comunicador.
 - ▣ Hasta que TODOS los procesos no ejecuten `MPI_Barrier`, no se continúa la ejecución



MPI_Barrier

59

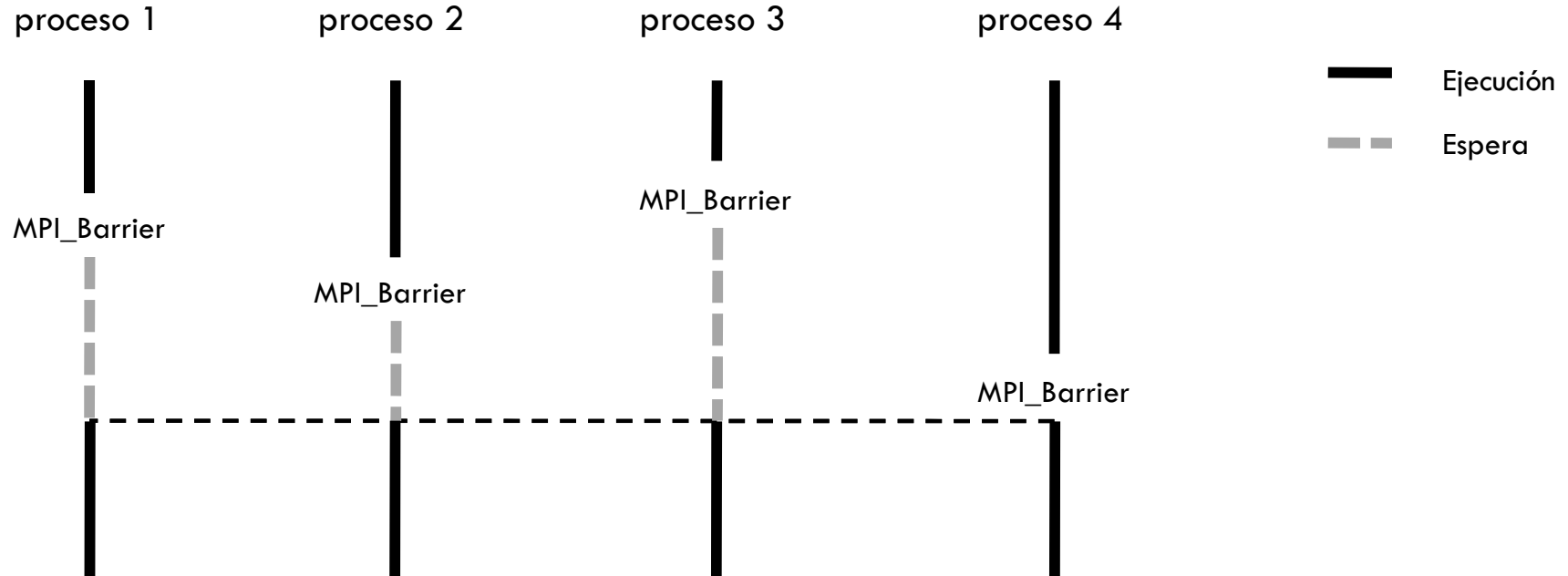
- `int MPI_Barrier(MPI_Comm comm);` `/* input */`
 - ▣ Se utiliza para sincronizar a TODOS los procesos asociados a un determinado comunicador.
 - ▣ Hasta que TODOS los procesos no ejecuten `MPI_Barrier`, no se continúa la ejecución



MPI_Barrier

60

- `int MPI_Barrier(MPI_Comm comm);` `/* input */`
 - ▣ Se utiliza para sincronizar a TODOS los procesos asociados a un determinado comunicador.
 - ▣ Hasta que TODOS los procesos no ejecuten `MPI_Barrier`, no se continúa la ejecución



Índice

61

□ Funciones de MPI

▣ Funciones de entorno

▣ Funciones punto-a-punto

▣ Funciones de sincronización

▣ Funciones colectivas

■ MPI_BCAST

■ MPI_SCATTER

■ MPI_GATHER

■ MPI_ALLGATHER

■ MPI_REDUCE

MPI_Bcast

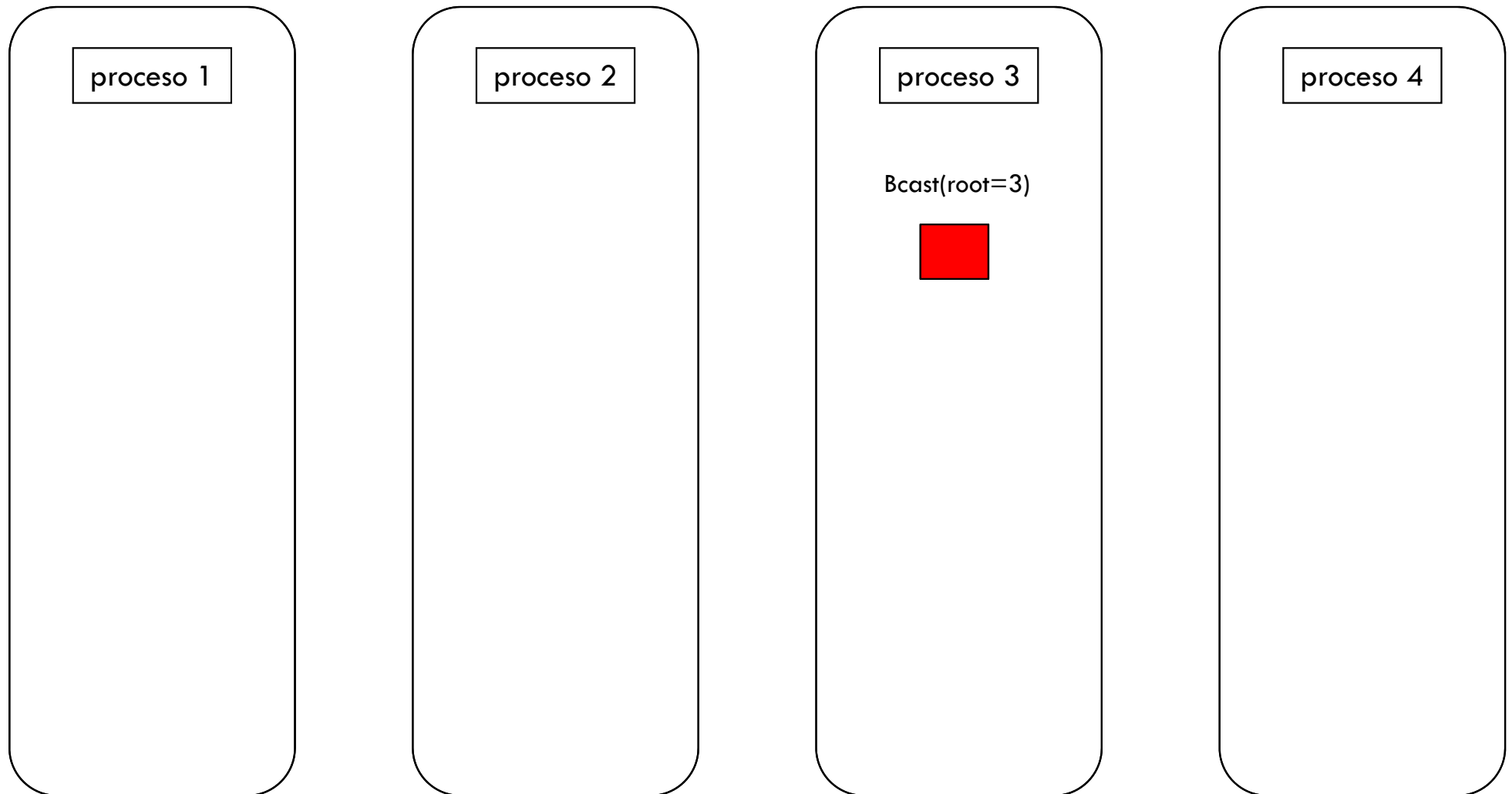
62

```
int MPI_Bcast( void* buffer,                /* input (root) / output (los demás) */
               int count,                  /* input */
               MPI_Datatype datatype,      /* input */
               int root,                   /* input */
               MPI_Comm comm);             /* input */
```

- ▣ buffer:
 - ▣ caso de root: datos del mensaje enviado al resto de procesos
 - ▣ caso de los demás procesos: datos del mensaje recibido de root
- ▣ count: número de elementos en el mensaje
- ▣ datatype: tipo de datos del mensaje
- ▣ root: *rank* del proceso emisor
- ▣ comm: comunicador (grupo de procesos participantes)

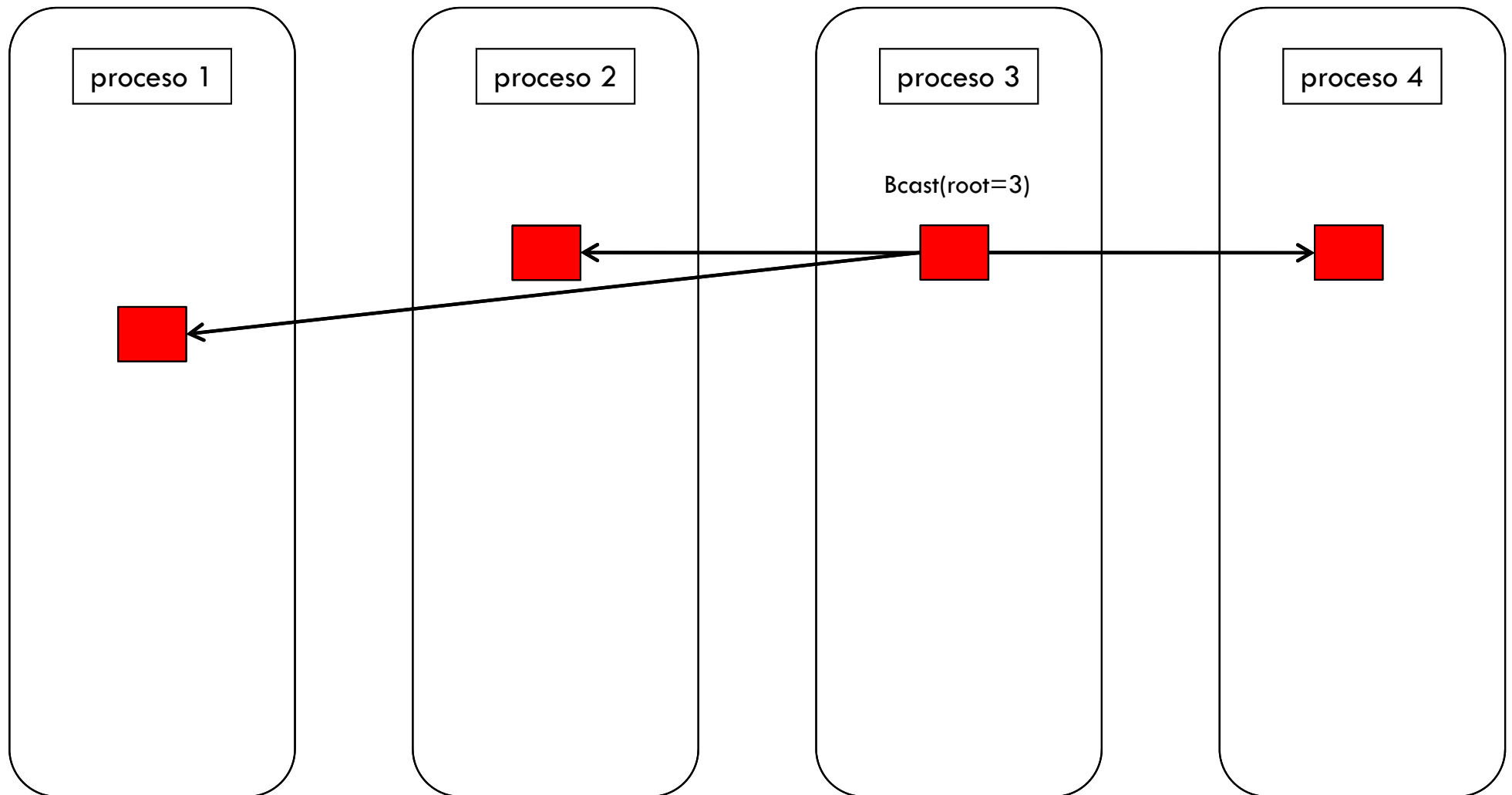
MPI_Bcast

63



MPI_Bcast

64



MPI_Bcast

65

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

int main( int argc, char **argv){

int rank, valor;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);

    // Master process?
    if (rank == 0){
        printf ("Enter a value\n");
        scanf("%d", &valor);
    }

    // Broadcast del mensaje
    MPI_Bcast(&valor, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf( "Proceso %d recibe valor [%d]\n", rank, valor);

    MPI_Finalize( );
    return 0;
}
```

Nótese: todos los participantes llaman a Bcast (uno envía, los otros reciben)

MPI_Bcast

66

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

int main( int argc, char **argv){

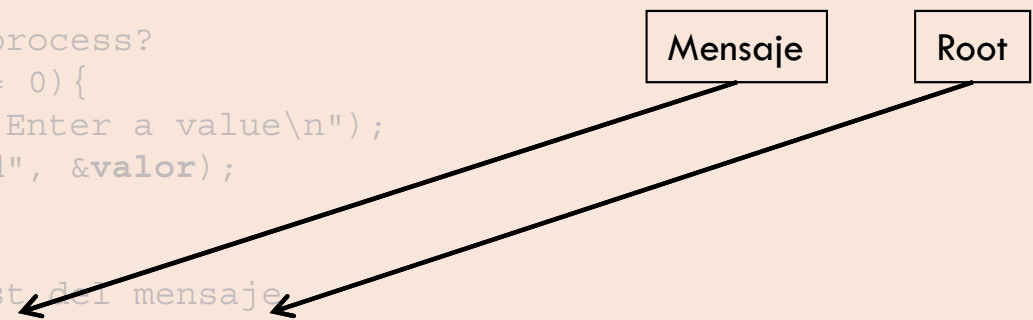
    int rank, valor;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);

    // Master process?
    if (rank == 0){
        printf ("Enter a value\n");
        scanf("%d", &valor);
    }

    // Broadcast del mensaje
    MPI_Bcast(&valor, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf( "Proceso %d recibe valor [%d]\n", rank, valor);

    MPI_Finalize( );
    return 0;
}
```



MPI_Scatter

67

```
int MPI_Scatter(void* sendbuf,  
               int sendcount,  
               MPI_Datatype sendtype,  
               void* recvbuf,  
               int recvcount,  
               MPI_Datatype recvtype,  
               int root,  
               MPI_Comm comm);
```

- Distribuye un mensaje
- Desde *root* a todos los procesos
- ¿Diferencia con BCast?
 - BCast = Mismo mensaje
 - Scatter = Porciones del mensaje
- Si ***sendcount*** no es divisible por ***size***, error. Ver función ***MPI_Scatterv***

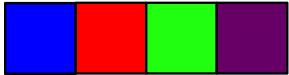
- *sendbuf*: datos para enviar, un trozo a cada proceso (solo *root* lo usa)
- *sendcount*: número de elementos que *root* envía a cada destinatario
- *sendtype*: tipo de datos del mensaje a enviar
- *recvbuf*: datos del mensaje recibido de *root*
- *recvcount*: número de elementos a recibir
- *recvtype*: tipo de datos del mensaje a recibir
- *root*: rank del proceso que envía los datos a todos los procesos
- *comm*: comunicador (procesos participantes)

MPI_Scatter

68

proceso 1

Scatter(root=1)



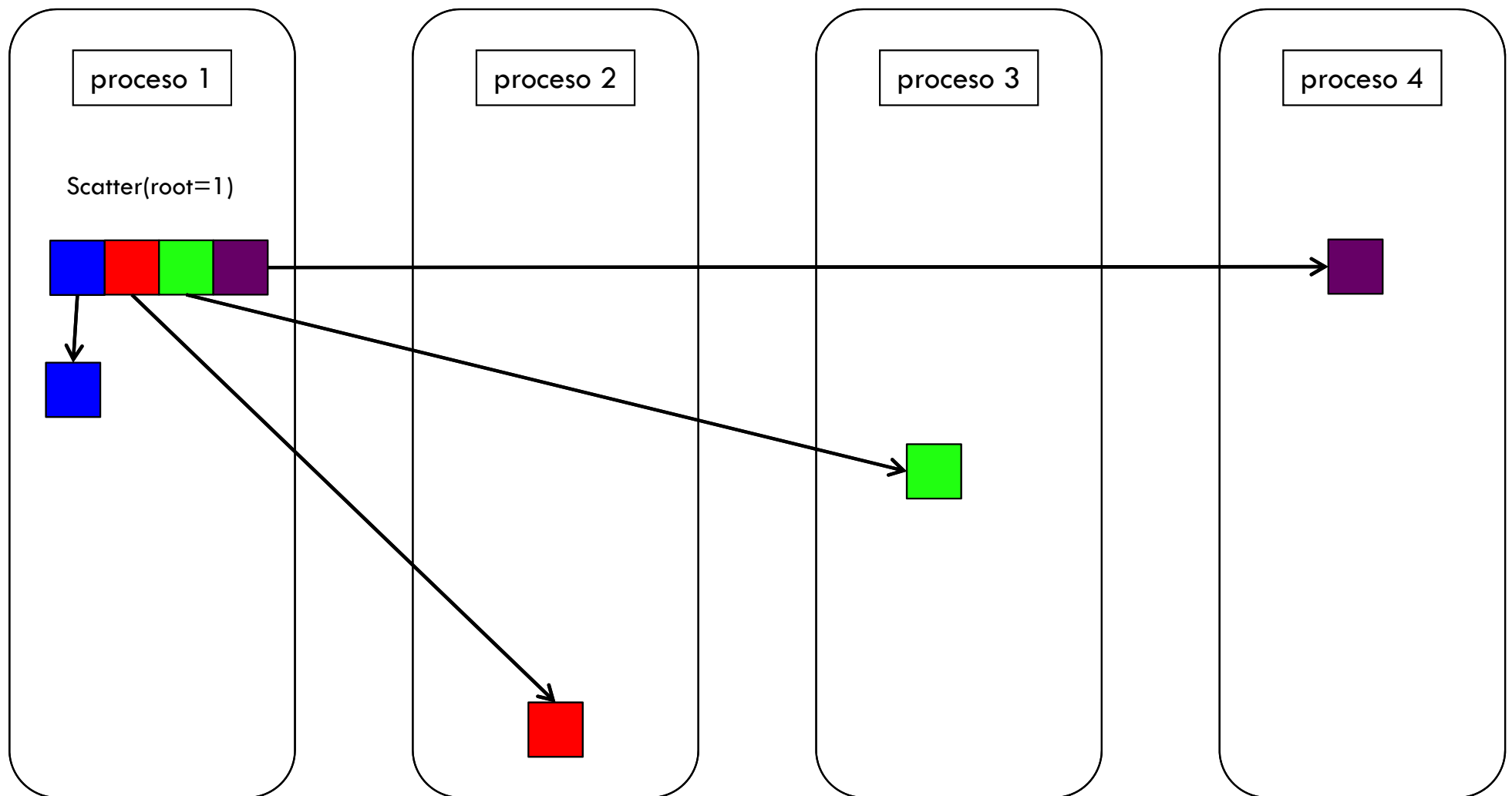
proceso 2

proceso 3

proceso 4

MPI_Scatter

69



MPI_Scatter

70

```
#include "mpi.h"
#include <stdio.h>
#define BUFFER_SIZE 3

int main(int argc, char **argv){

    // Total procesos y rank
    int size, rank;

    // Número de elementos para enviar/recibir y emisor
    int sendcount, recvcnt, source;

    // Buffer de envío
    float sendbuf[BUFFER_SIZE][BUFFER_SIZE] = {
        {6.5, 1.6, 9.4},
        {4.6, 3.5, 3.555},
        {23.6, 42.3, 235545.5}};

    // Buffer de recepción
    float recvbuf[BUFFER_SIZE];

    // Init...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

MPI_Scatter

71

```
// Comprobar el número de procesos
if (size == BUFFER_SIZE) {
    source = 1;
    sendcount = BUFFER_SIZE;
    recvcount = BUFFER_SIZE;

    // Repartimos el mensaje entre los procesos...
    MPI_Scatter(sendbuf,
        sendcount,
        MPI_FLOAT,
        recvbuf,
        recvcount,
        MPI_FLOAT,
        source,
        MPI_COMM_WORLD);

    printf("rank= %d  Results: %f %f %f\n", rank, recvbuf[0],
        recvbuf[1], recvbuf[2]);
}

else
    printf("Esta prueba funciona para %d procesos.\n", BUFFER_SIZE);

MPI_Finalize();
}
```

MPI_Gather

72

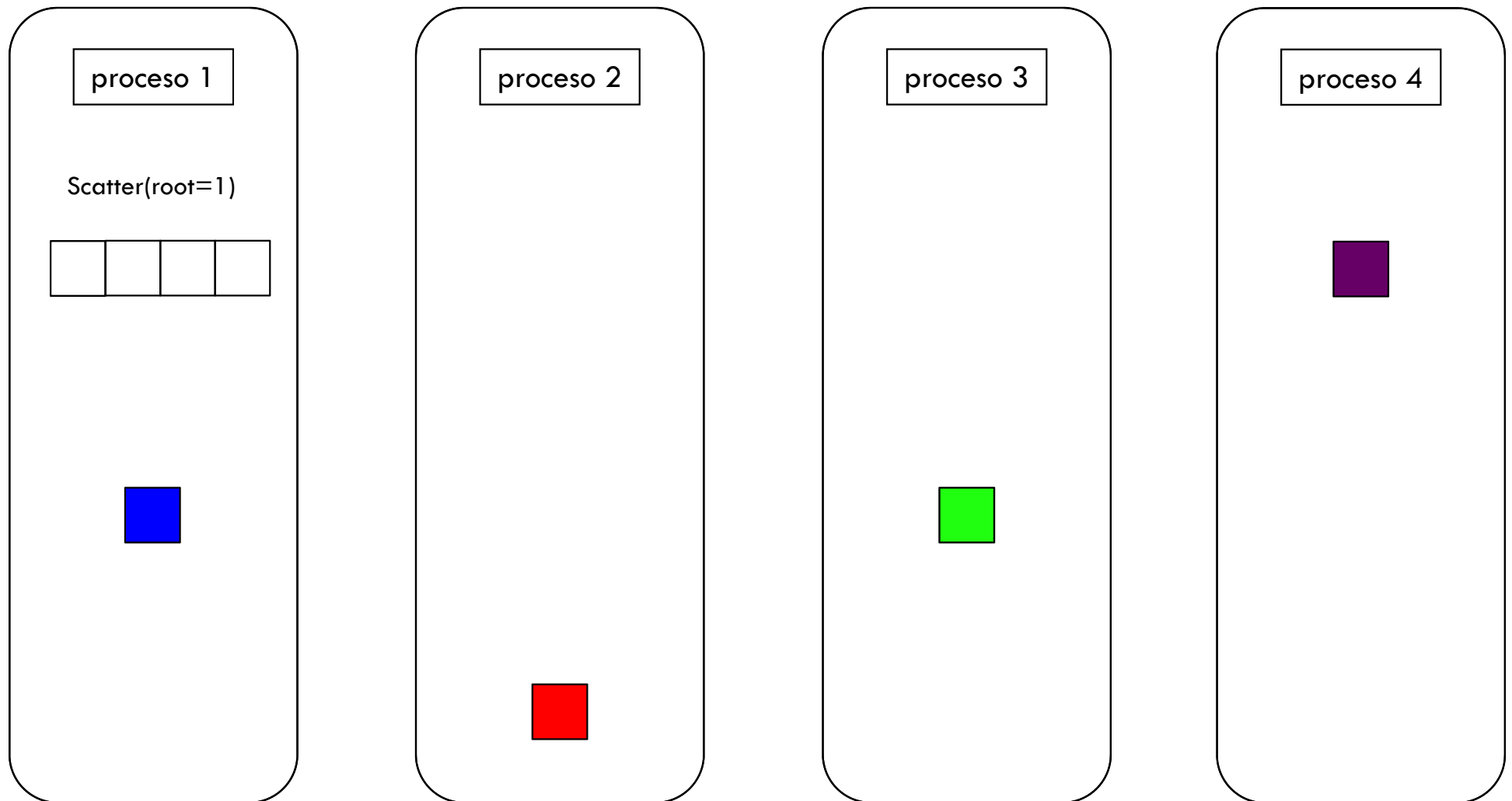
```
int MPI_Gather(void* sendbuf,  
              int sendcount,  
              MPI_Datatype sendtype,  
              void* recvbuf,  
              int recvcount,  
              MPI_Datatype recvtype,  
              int root,  
              MPI_Comm comm);
```

- ❑ sendbuf: datos para enviar a root
- ❑ sendcount: número de elementos a enviar a root
- ❑ sendtype: tipo de datos a enviar
- ❑ recvbuf: datos recibidos, un trozo de cada proceso (solo root lo usa)
- ❑ recvcount: número de elementos que *root* recibe de cada emisor
- ❑ recvtype: tipo de datos que root recibe
- ❑ root: rank del proceso que recibe los datos de todos los procesos
- ❑ comm: comunicador (procesos participantes)

- ❑ Recolecta un mensaje
- ❑ Desde todos los procesos a root
- ❑ Al final del proceso, **root** tiene una copia de cada bloque, cada uno enviado por un proceso distinto
- ❑ Es la función contraria a **MPI_Scatter**
- ❑ Se almacena en orden estricto de **rank**
- ❑ Si **sendcount** no es divisible por **size**, error. Ver función **MPI_Gatherv**

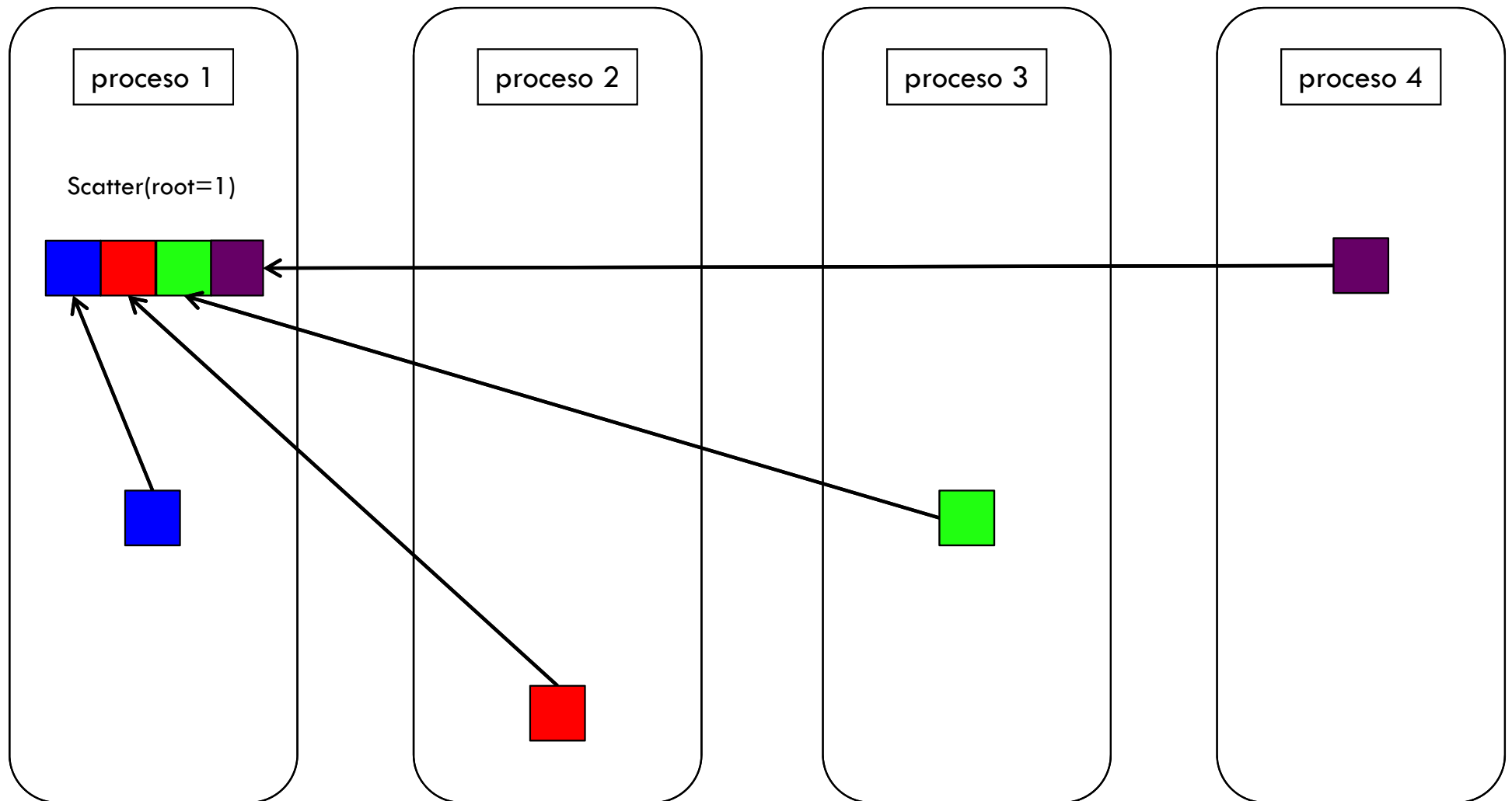
MPI_Gather

73



MPI_Gather

74



MPI_Allgather

75

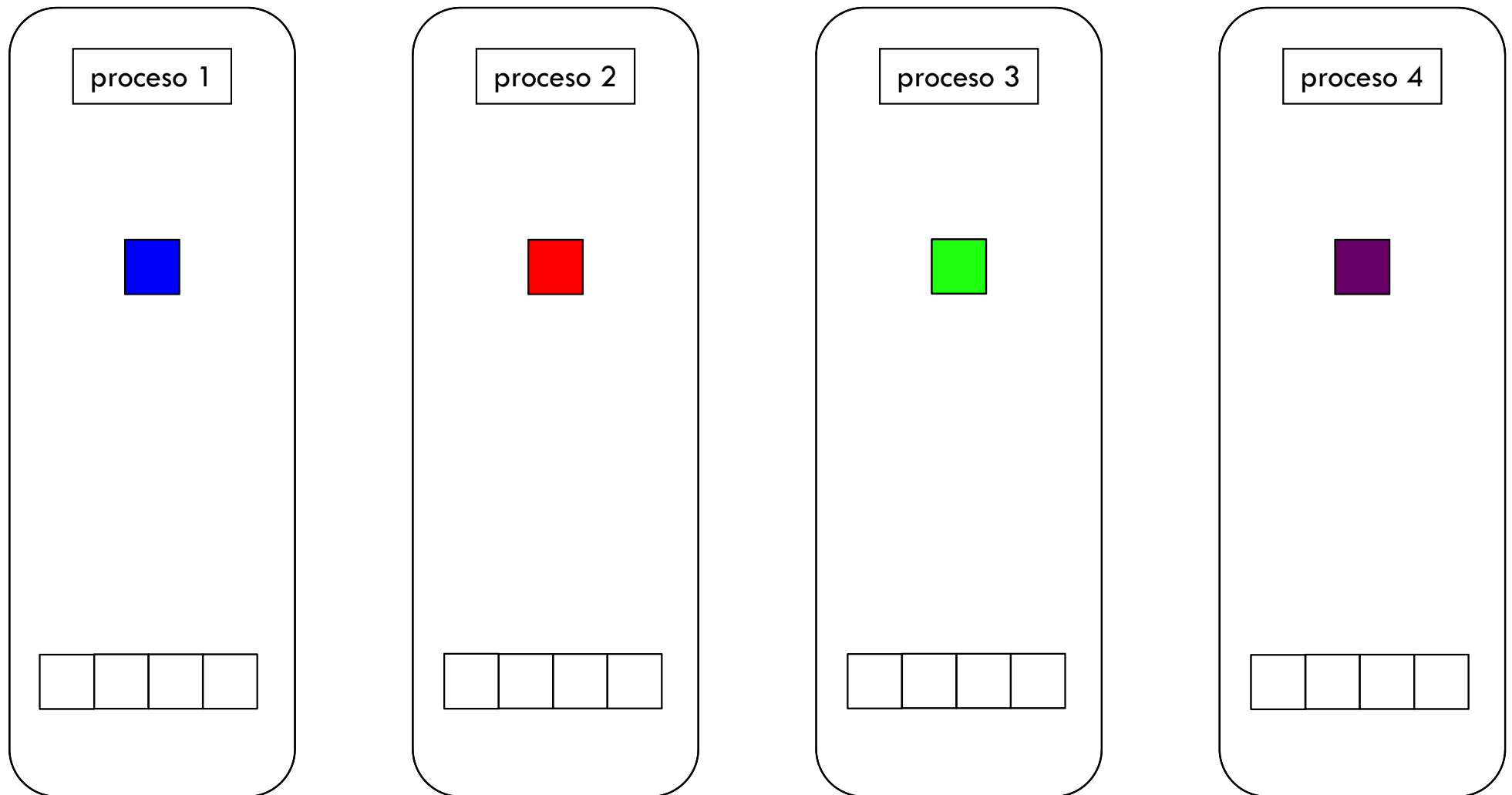
```
int MPI_Allgather(void* sendbuf,  
                 int sendcount,  
                 MPI_Datatype sendtype,  
                 void* recvbuf,  
                 int recvcount,  
                 MPI_Datatype recvtype,  
                 MPI_Comm comm);
```

- Parecido a MPI_Gather
- Al finalizar, **todos los procesos** tienen una copia de cada elemento, cada uno enviado por un proceso distinto

- sendbuf: datos para enviar a todos los procesos
- sendcount: número de elementos a enviar a cada destinatario
- sendtype: tipo de datos del mensaje a enviar
- recvbuf: datos recibidos, un trozo de cada proceso
- recvcount: número de elementos a recibir de cada emisor
- recvtype: tipo de datos del mensaje a recibir
- comm: Comunicador

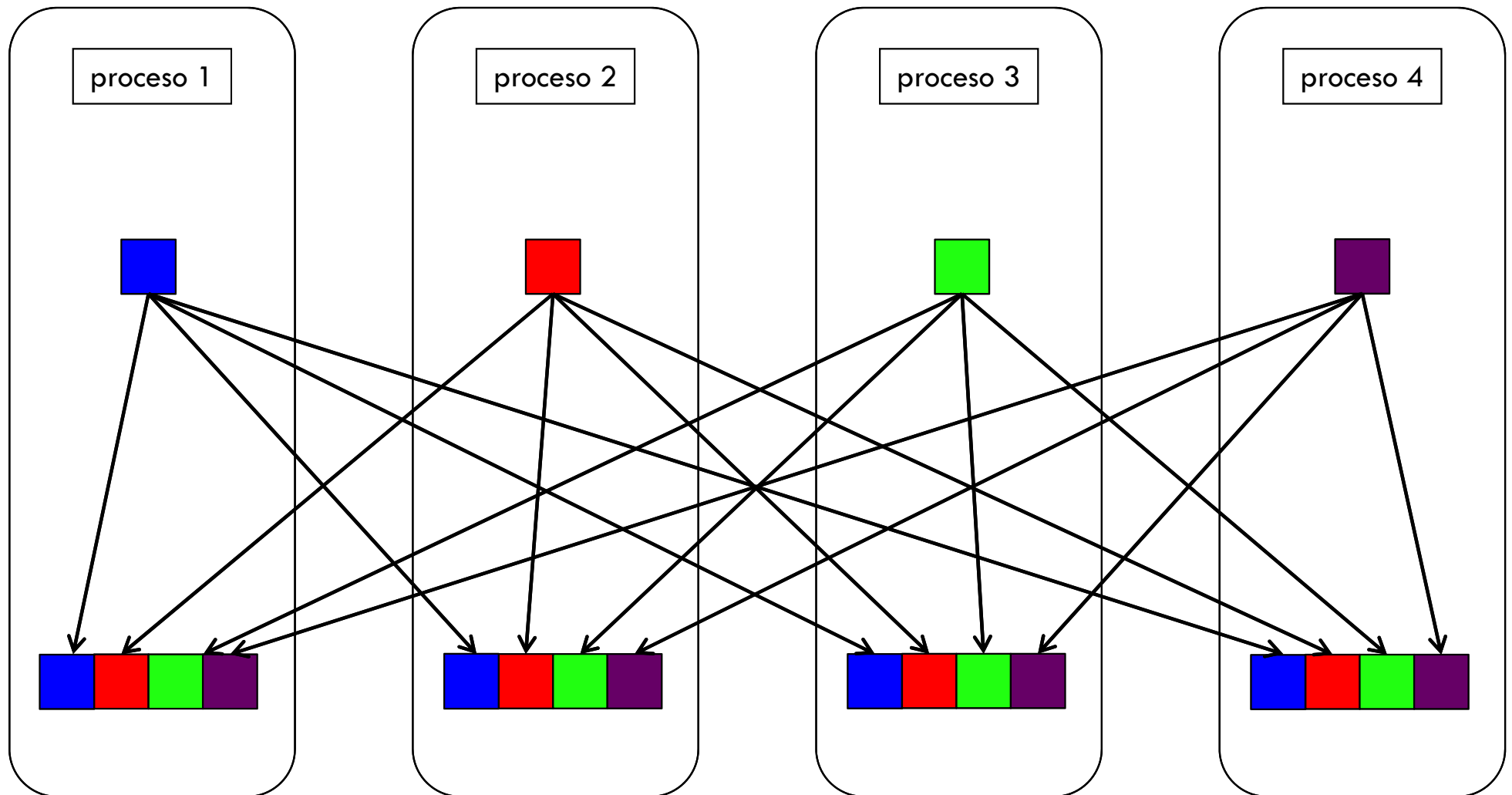
MPI_Allgather

76



MPI_Allgather

77



MPI_Alltoall

78

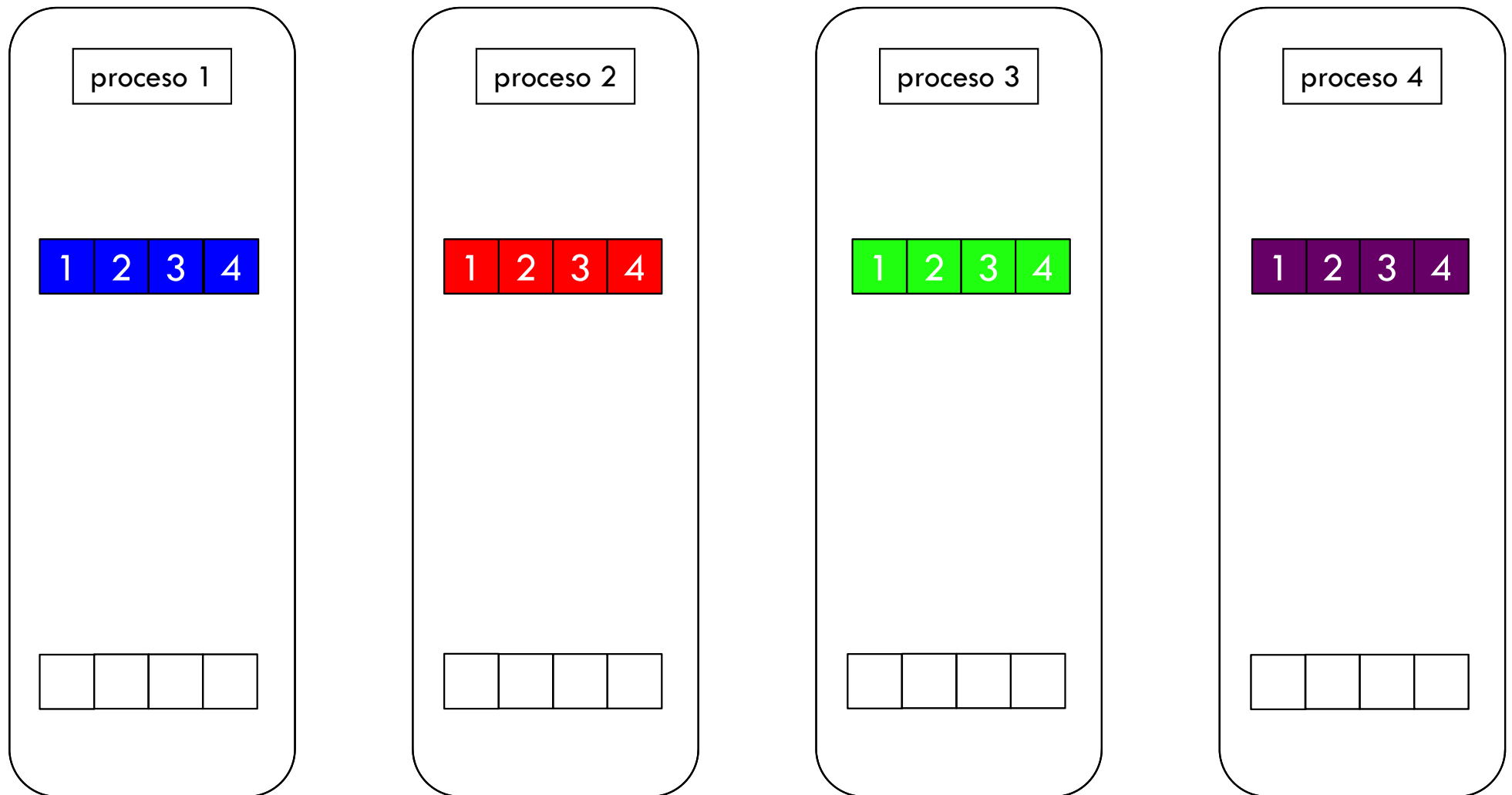
```
int MPI_Alltoall(void* sendbuf,  
                int sendcount,  
                MPI_Datatype sendtype,  
                void* recvbuf,  
                int recvcount,  
                MPI_Datatype recvtype,  
                MPI_Comm comm);
```

- Parecido a MPI_Gather
- Al finalizar, **cada proceso** tiene los bloques por los que fue destinatario

- sendbuf: datos para enviar, un trozo a cada proceso
- sendcount: número de elementos a enviar a cada destinatario
- sendtype: tipo de datos del mensaje a enviar
- recvbuf: datos recibidos, un trozo de cada proceso
- recvcount: número de elementos a recibir de cada emisor
- recvtype: tipo de datos del mensaje a recibir
- comm: comunicador

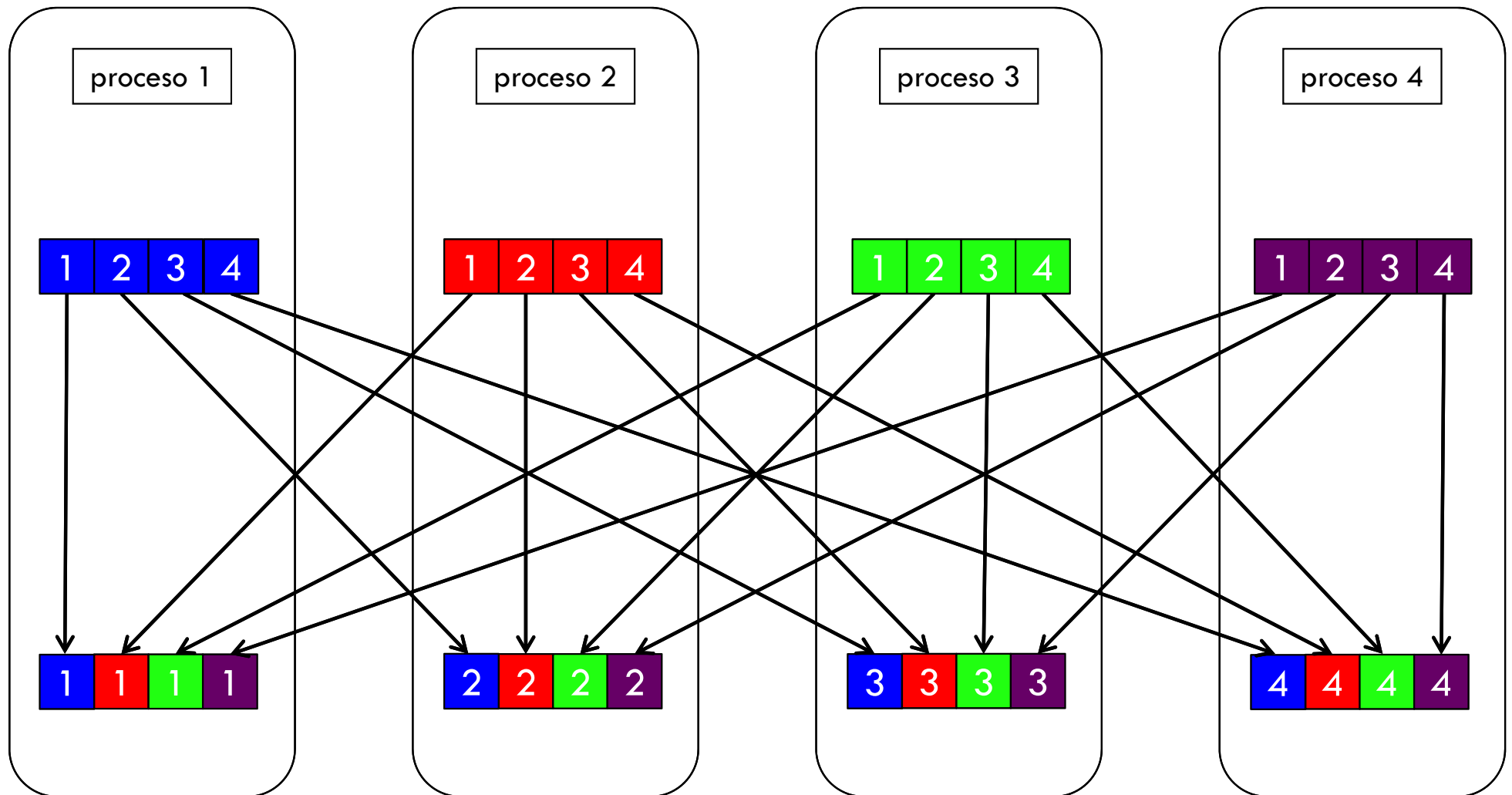
MPI_Alltoall

79



MPI_Alltoall

80



MPI_Reduce

81

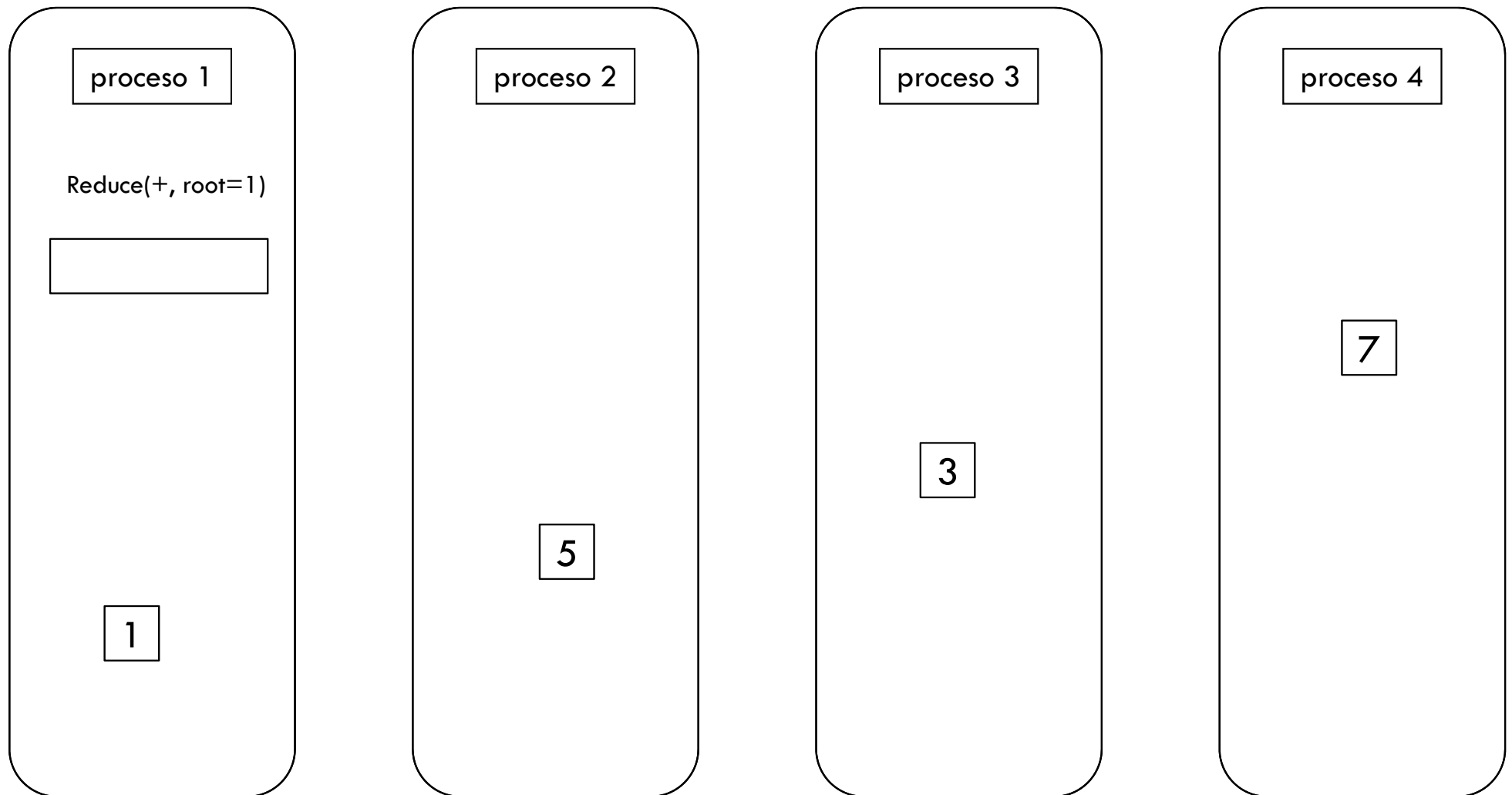
```
int MPI_Reduce(void* sendbuf,  
              void* recvbuf,  
              int count,  
              MPI_Datatype datatype,  
              MPI_Op op,  
              int root,  
              MPI_Comm comm);
```

- Operación de reducción
- Aplica una operación sobre los datos enviados
 - Utiliza todos los procesos
- El resultado se almacena en **recvbuf** de root
- Ambos buffers tienen
 - Mismo tipo de elementos

- sendbuf: datos para enviar a root
- recvbuf: resultado, calculado con los datos enviados por todos los procesos (solo root lo usa)
- count: número de elementos que cada proceso envió a root
- datatype: tipo de datos del mensaje
- root: rank del proceso que almacena el resultado
- comm: comunicador

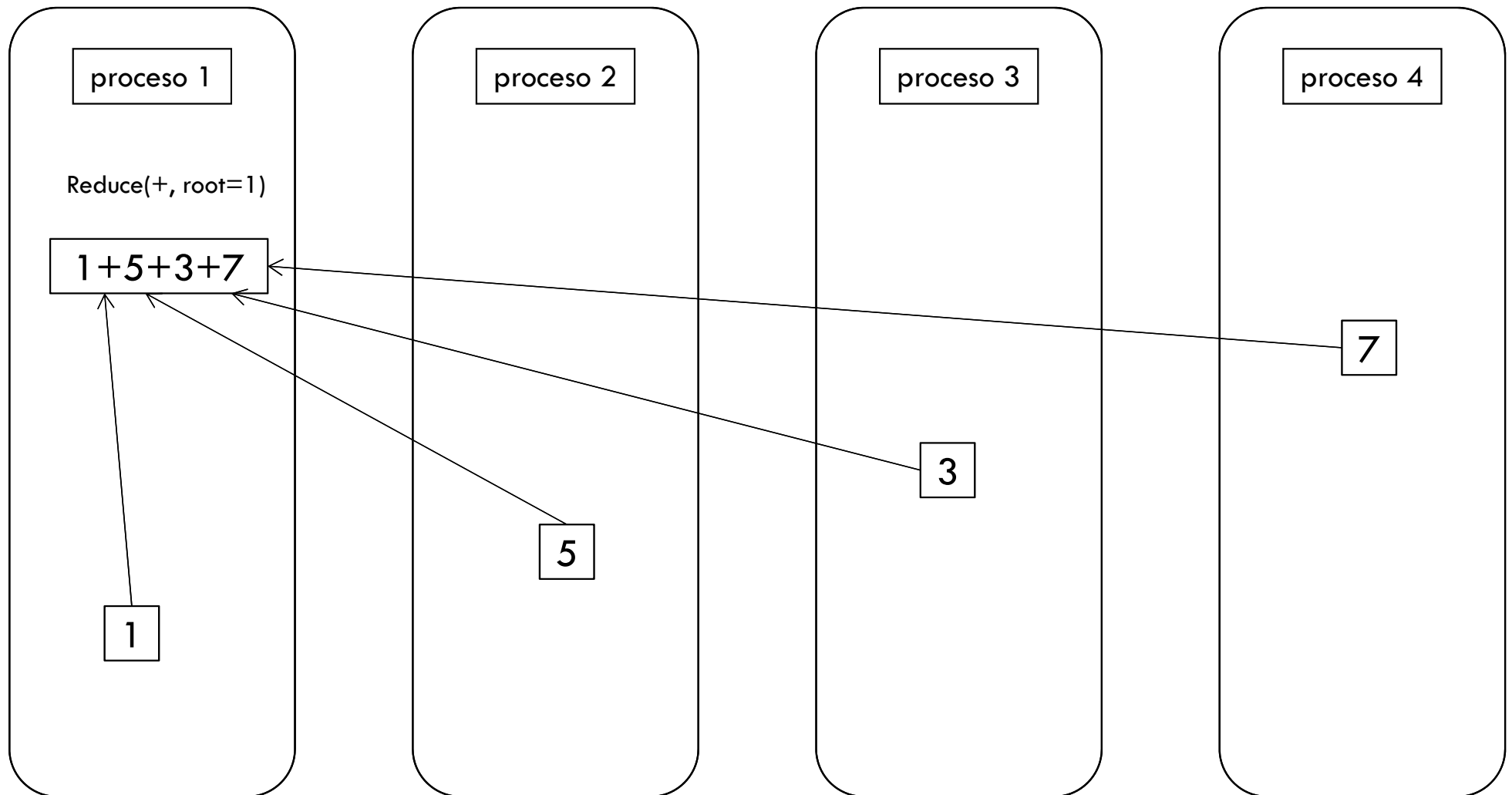
MPI_Reduce

82



MPI_Reduce

83



MPI_Reduce

84

Operaciones predefinidas:

- [MPI_MAX] Valor máximo
- [MPI_MIN] Valor mínimo
- [MPI_SUM] Suma
- [MPI_PROD] Producto
- [MPI LAND] AND lógico
- [MPI_BAND] AND (nivel de bit)
- [MPI_LOR] OR lógico
- [MPI BOR] OR (nivel de bit)
- [MPI_LXOR] XOR lógico
- [MPI_BXOR] XOR (nivel de bit)

MPI_Reduce

85

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

main(int argc, char **argv){

    int rank,size;
    int num, resultado;

    // Init
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Número asociado a cada proceso
    num = (rank+10)*2;
    printf ("Proceso [%d] con número: %d\n", rank, num);

    // Aplicamos la operación de reducción
    MPI_Reduce(&num, &resultado, 1, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);

    // Master process?
    if (rank==0)
        printf("Proceso [%d] tiene el resultado: %d\n",rank, resultado);

    MPI_Finalize();
}
```

MPI_Reduce

86

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

main(int argc, char **argv){

    int rank,size;
    int num, resultado;

    // Init
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Número asociado a cada proceso
    num = (rank+10)*2;
    printf ("Proceso [%d] con número: %d\n", rank, num);

    // Aplicamos la operación de reducción
    MPI_Reduce(&num, &resultado, 1, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);

    // Master process?
    if (rank==0)
        printf("Proceso [%d] tiene el resultado: %d\n",rank, resultado);

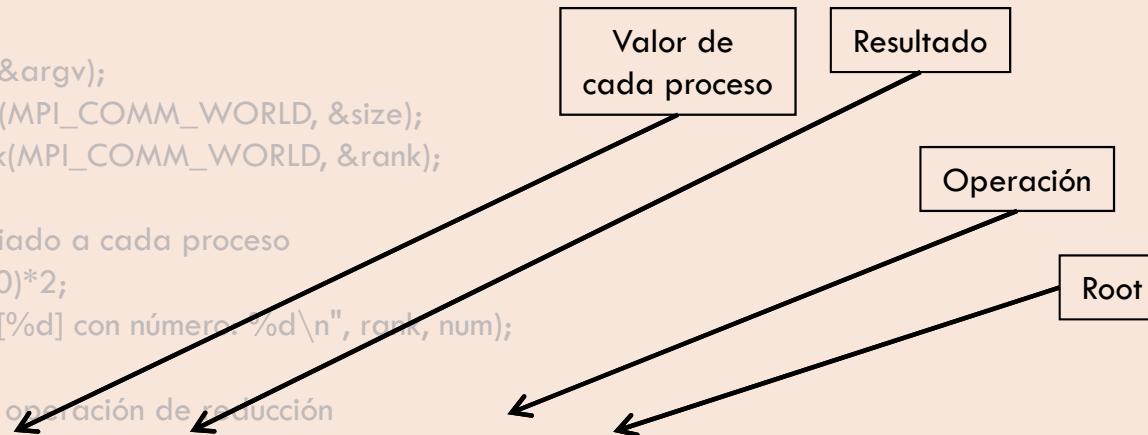
    MPI_Finalize();
}
```

Valor de
cada proceso

Resultado

Operación

Root



MPI_Allreduce

87

- Cada proceso del comunicador termina con una copia del resultado
- Equivalente a `MPI_Reduce` + `MPI_Bcast`