Programación Paralela (MPI)

FUNDAMENTOS TECNOLÓGICOS PARA BIG DATA

Isaac Esau Rubio Torres

2018

Contenidos

- Paralelización de aplicaciones con MPI
- Biblioteca de paso de mensajes

Introducción

- Paralelizar una aplicación para que se ejecute en un sistema de memoria compartida SMP “no es muy complejo”.

  El uso de variables compartidas facilita la comunicación entre procesos, aunque implica:

  – analizar detalladamente el tipo de variables.
  – sincronizar correctamente el acceso a las variables compartidas.

- Sin embargo, el número de procesadores de un sistema SMP no suele ser muy grande, por lo que no es fácil conseguir altos niveles de paralelismo.

Introducción

- Es relativamente sencillo conseguir una máquina paralela tipo *cluster* con muchos procesadores, uniendo P máquinas independientes mediante una red de conexión estándar.

- Por ejemplo, nosotros vamos a utilizar una máquina de 32 + 3 nodos (32 + 3x4 = 44 pr.), unidos mediante una red gigabit ethernet.

  No es una máquina de alto rendimiento, pero es “barata” y sencilla de ampliar (en nodos de cálculo y en comunicación).

Introducción

- Sin embargo, programar aplicaciones para sistemas de memoria distribuida es más complejo.

- Recuerda:
  - la memoria de cada procesador es de uso privado, por lo que todas las variables son, por definición, privadas.
  - la comunicación entre procesos debe hacerse a través de paso explícito de mensajes.
  - la red de comunicación juega un papel importante en el rendimiento del sistema.

Introducción

- Diferentes alternativas para programar aplicaciones:
  - utilizar lenguajes diseñados específicamente para sistemas paralelos (OCCAM).
  - ampliar la sintaxis de un lenguaje estándar para gestionar el paso de mensajes (Fortran M).
  - utilizar un lenguaje estándar y una librería de funciones de comunicación.
  -

Introducción

- Necesitamos:
  - un método para crear procesos: estático / dinámico.
  - un método para enviar y recibir mensajes, punto a punto y de manera global.

- El estándar actual de programación de los sistemas de memoria distribuida, mediante paso de mensajes, es MPI (*message-passing interface*).

  PVM → MPI 1.0 (94) → MPI 2.0 (97)

Introducción

- MPI es, básicamente, una librería (grande) de funciones de comunicación para el envío y recepción de mensajes entre procesos.
  Para Fortran y C.
  Se busca: portabilidad, eficiencia...

- El objetivo de MPI es explicitar la comunicación entre procesos, es decir:
  > el movimiento de datos entre procesadores
  > la sincronización de procesos

Introducción

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

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

Recuerda 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.

Introducción

- MPI gestiona los procesos (número y asignación) de manera estática (MPI2 permite gestión dinámica de procesos).

- La comunicación entre procesos puede hacerse de formas muy diferentes.

  Elegiremos una determinada estrategia en función de la longitud de los mensajes, de la estructura del programa...

Introducción

- En todo caso, ten 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, dos implementaciones libres de uso muy extendido: LAM y MPICH.

  Nosotros vamos a usar MPICH.

Funciones Básicas: Init/Finalize

1. Comienzo y final del programa:

```
> 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`, y si un proceso no ejecuta `_Finalize` el programa queda como “colgado”.

Funciones Básicas

- 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.

F. Básicas: Comm_rank/_size

2. 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

- Un ejemplo simple

```
#include <stdio.h> #include
<mpi.h> main (int argc, char
*argv[])
{ int pid, npr, A = 2;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &pid);
   MPI_Comm_size(MPI_COMM_WORLD,  &npr);  A  =  A  +  1;
   printf("Proceso %d de %d activado. A = %d \n", pid, npr, A);
   MPI_Finalize();
}
```

Funciones Básicas

- Otro ejemplo: planificación de un bucle

```
... main (int argc, char
*argv[])
{ ...
   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &pid);

   MPI_Comm_size(MPI_COMM_WORLD, &npr);

   for (i=pid; i<N; i=i+npr) func(i);

   MPI_Finalize();
}
```

Funciones Básicas: comunicación

3. Envío y recepción de mensajes

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

3. Envío y recepción de mensajes entre dos procesos

   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

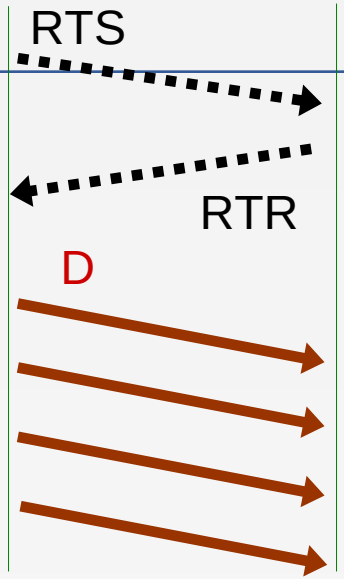
- MPI ofrece diferentes modo de comunicación. Veamos un resumen.

- Modos de comunicación (1)

Funciones Básicas: comunicación

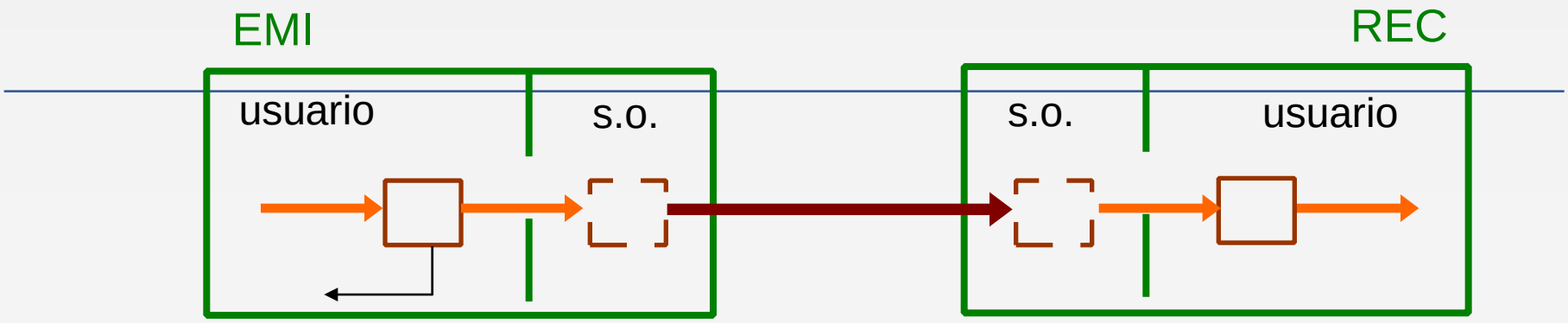
- síncrona: la comunicación no se produce hasta que emisor y receptor se ponen de acuerdo (sin búfer intermedio).

  - petición de transmisión (espera)
  - aceptación de transmisión - envío de datos (de usuario a usuario)



Modos de comunicación (1)

- con búfer (*buffered*): el emisor deja el mensaje en un búfer y retorna. La comunicación se produce cuando el receptor está dispuesto a ello. El búfer no se puede reutilizar hasta que se vacíe.



¡Ojo con el tamaño del búfer!

Funciones Básicas: comunicación

- Modos de comunicación (2)
  - bloqueante

    Se espera a que la “comunicación” se produzca, antes de continuar con la ejecución del programa.

La comunicación síncrona es bloqueante. La comunicación con búfer también, si el mensaje no cabe en el búfer.

- no bloqueante

  Se retorna “inmediatamente” de la función de comunicación, y se continúa con la ejecución. Se comprueba más tarde si la comunicación se ha efectuado.

Funciones Básicas: comunicación

- Cada estrategia tiene sus ventajas e inconvenientes:

> síncrona: es más rápida si el receptor está dispuesto a recibir; nos ahorramos la copia en el búfer. Además del intercambio de datos, sirve para sincronizar los procesos.
Ojo: al ser bloqueante es posible un *deadlock*!

> con búfer: el emisor no se bloquea si el receptor no está disponible, pero hay que hacer copia(s) del mensaje (más lento).

Funciones Básicas: Send y Recv

- 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 (que se puede “procesar”).

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

Funciones Básicas: Send y Recv

- Función estándar para enviar un mensaje:

```
> MPI_Send(&mess,count,type,dest, tag,comm);
```

- mensaje a enviar: [`mess` (@comienzo), `count` (tamaño), `type`]
- receptor: [`dest` (@destino), `comm` (comunicador)]
- `tag`: 0..32767 (clase de mensajes, orden...)

Tipos : `MPI_CHAR, INT, LONG, FLOAT, DOUBLE,BYTE...`

`Send` utiliza la capacidad de *buffering* del sistema; es decir, retorna una vez copiado en el búfer el mensaje a enviar… ¡siempre que quepa!

Funciones Básicas: Send y Recv

- Función básica para recibir un mensaje:

```
> MPI_Recv(&mess,count,type,source, tag,comm, &status);
```

- mensaje a recibir: [`mess`, `count`, `type`]
- emisor: [`source`, `comm`]
- `tag`: clase de mensaje
- `status`: devuelve información sobre el mensaje recibido

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

Funciones Básicas: Send y Recv

- Algunas precisiones:
  - `source`, `dest`, `count` y `tag` son enteros (`int`); `comm` y `status` son de tipo `MPI_Comm` y `MPI_Status`.

- para que la comunicación se efectúe tienen que coincidir las direcciones de emisor y receptor, y el `tag` del mensaje.
- el tamaño del mensaje (`count`) definido en la función `Recv` debe ser igual o mayor al definido en `Send`.
- el origen de un mensaje en la función `Recv` puede ser `MPI_ANY_SOURCE`, y el tipo de mensaje puede ser `MPI_ANY_TAG`.

Funciones Básicas: Send y Recv

- Algunas precisiones:
  - `status` es un *struct* con tres campos, en el que se devuelve información sobre el mensaje recibido:

`status.MPI_SOURCE`: indica el emisor del mensaje

`status.MPI_TAG`: devuelve el `tag` del mensaje recibido

`status.MPI_ERROR`: devuelve un código de error

(aunque lo más habitual es abortar en caso de error)

- también puede obtenerse el tamaño del mensaje recibido ejecutando:

```
> MPI_Get_count(&status,type,&count);
```

Funciones Básicas: Send y Recv

- Algunas precisiones:
  - si un proceso tiene varios mensajes pendientes de recibir, no se reciben en el orden en que se enviaron sino en el que se indica en la recepción mediante los parámetros de `origen` y `tag` del mensaje.

- si el `tag` del mensaje que se recibe puede ser cualquiera, los mensajes que provienen del mismo origen se reciben en el orden en que se enviaron.

Ejemplo

```
...
#define      N 10
int main (int argc, char **argv)
{ int pid, npr, orig, dest, ndat,
 tag;
 int i, VA[N];
 MPI_Status info;

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

 for (i=0;i<N;i++) VA[i] = 0;

 if (pid == 0)
 { for (i=0;i<N;i++) VA[i] =
   i; dest = 1; tag = 0;
   MPI_Send(&VA[0],N,MPI_INT,dest,
 tag, MPI_COMM_WORLD); }
```

```
else if (pid == 1)
{ for (i=0;i<N;i++)
   printf("%4d",VA[i]);

   orig = 0; tag = 0;
   MPI_Recv(&VA[0], N, MPI_INT, orig,
            tag, MPI_COMM_WORLD, &info);

   MPI_Get_count(&info, MPI_INT, &ndat);

   printf("Datos desde pr %d; tag = %d,
            ndat = %d \n", info.MPI_SOURCE,
            info.MPI_TAG, ndat);

   for (i=0;i<ndat;i++)
            printf("%4d",VA[i]);
 }

MPI_Finalize();
}
```

F. Básicas: temporización

- Un par de funciones MPI para obtener tiempos de ejecución:

```
doubleMPI_Wtime();
```

tiempo (s) transcurrido desde algún instante anterior

```
double MPI_Wtick();
```

devuelve la precisión de la medida de tiempo

```
t1 = MPI_Wtime(); ... t2 =
MPI_Wtime();  printf("T  =
%f\n", t2-t1);
```

Comunicación en grupo

- Muchas aplicaciones requieren de operaciones de comunicación en las que participan muchos procesos.
- La comunicación es en grupo o colectiva si participan en ella todos los procesos del comunicador.

- Ejemplo: un *broadcast*, envío de datos desde un proceso a todos los demás.

  En general, podría ejecutarse mediante un bucle de funciones tipo send/receive, pero no sería muy eficiente.

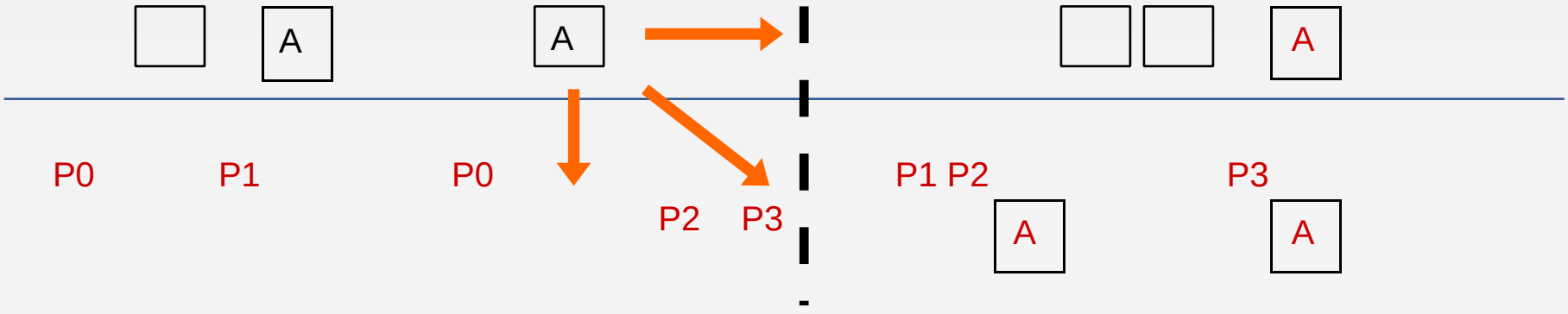
Comunicación en grupo

- Todos los procesos del comunicador deben ejecutar la función. Las funciones de comunicación en grupo son bloqueantes en el mismo sentido que la función `Send`.

- Tres tipos: 1 Movimiento de datos
  2 Cálculo en grupo
  3 Sincronización
- Las principales funciones de comunicación en grupo que ofrece MPI son las siguientes:

CG: mov. De datos, broadcast

- 1aBROADCAST: envío de datos desde un proceso (`root`) a todos los demás.

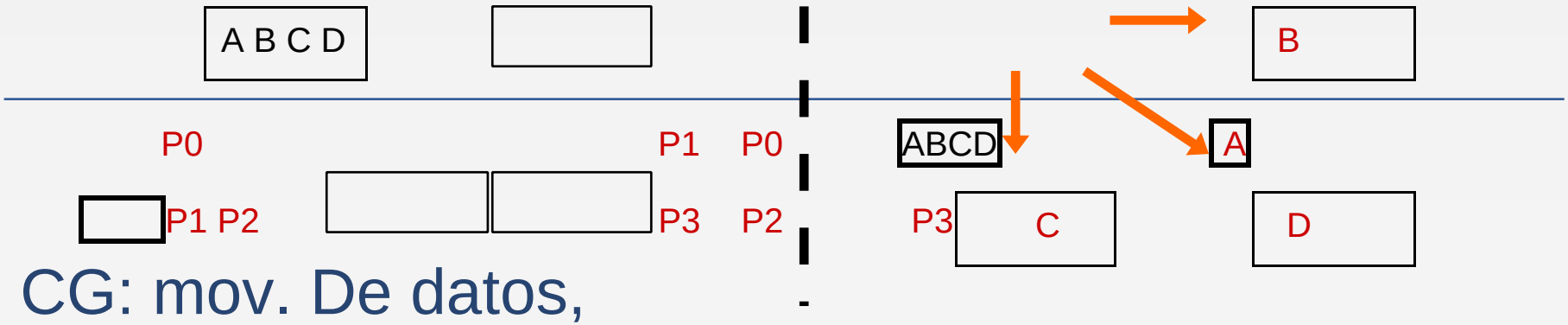



```
> MPI_Bcast(&mess,count,type,root,comm);
```

(La implementación suele ser en árbol)

CG: mov. De datos, scatter

- 1bSCATTER: reparto de datos desde un proceso al resto de procesos del comunicador.



CG: mov. De datos, scatter

```
> MPI_Scatter(&send_data,send_count,send_type,
              &recv_data,recv_count,recv_type,
              root,comm);
```

- el proceso `root` distribuye `send_data` en P trozos, uno por procesador, de tamaño `send_count`.
- los datos se reciben en `recv_data` (también en `root`).

- lo lógico es que el tamaño y tipo de los datos que se envían y se reciben sean iguales.

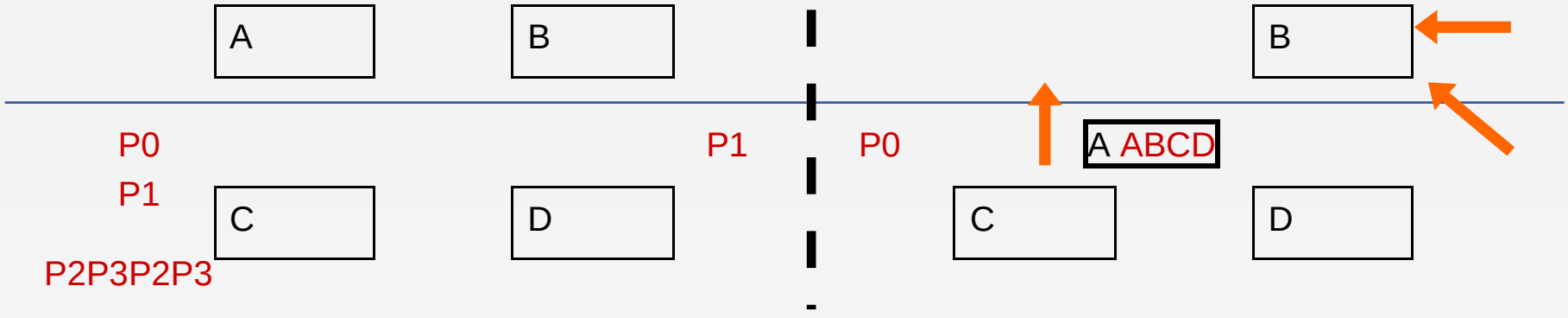
Ejemplo: `A` = (0, 1, 2, 3, 4, 5, 6, 7) en P0

```
MPI_Scatter(A,2,MPI_INT,B,2,MPI_INT,0,comm);
```

(P0) `B` = 0, 1 (P1) `B` = 2, 3 (P2) `B` = 4, 5 (P3) `B` = 6, 7

CG: mov. De datos, gather

- 1c GATHER: recolección de datos de todos los procesos en uno de ellos (orden estricto de `pid`).



CG: mov. De datos, gather

```
>     MPI_Gather(&send_data,send_count,send_type,
                 &recv_data,recv_count,recv_type,
                 root, comm);
```

- el proceso `root` recolecta en `recv_data` los datos enviados en `send_data` por cada proceso del comunicador.
- los datos se guardan en el orden marcado por el `pid`.

- `recv_count` indica el tamaño de los datos recibidos de cada proceso, no el total; lo lógico es que tamaño y tipo de los datos que se envían y se reciben sean iguales.

Ejemplo: (P0) `B` = 0, 1 (P1) `B` = 2, 3 (P2) `B` = 4, 5 (P3) `B` = 6, 7

```
MPI_Gather(B,2,MPI_INT,C,2,MPI_INT,0,comm);
```

→ en P0: `C` = 0, 1, 2, 3, 4, 5, 6, 7

CG: movimiento de datos

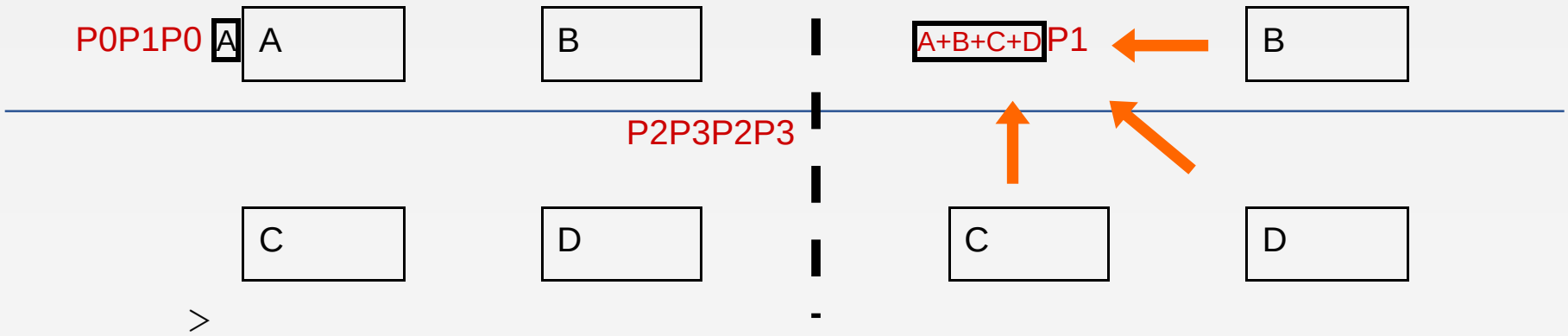
- Otras versiones de estas funciones
  - `MPI_Allgather(…);` al final, todos los procesos disponen de todos los datos.
  - `MPI_Gatherv(…);` la información que se recolecta es de tamaño variable.

> `MPI_Allgatherv(…);` “suma” de las dos anteriores.

> `MPI_Alltoall(…);` todos los procesos distribuyen datos a todos los procesos.
> `MPI_Scatterv(…);` la información que se distribuye es de tamaño variable.
> `MPI_Alltoallv(…);` “suma” de las dos anteriores.

CG: cálculo en grupo, reduce

- REDUCE: una operación de reducción con los datos de cada procesador, dejando el resultado en uno de ellos (`root`).




```
>
-
MPI_Reduce(&operand,&result,count,datatype, operator,root,comm);
```

CG: cálculo en grupo, reduce

- Algunos comentarios
  - Operación: `result` = `result`*`operator`*`operand`
- `result` es una variable del proceso destino, de nombre diferente a `operand` (no *aliasing*).
  - Funciones típicas de reducción: `MPI_MAX`, `_MIN`, `_SUM`, `_PROD`, `_BAND`, `_BOR`, `_BXOR`, `_LAND`, `_LOR`, `_LXOR`, `_MAXLOC`, `_MINLOC`
  - Pueden definirse otras operaciones de reducción:

```
MPI_Op_create(…); MPI_Op_free(…);
```

CG: sincronización

3 BARRIER: sincronización global entre los procesos del comunicador.

```
> MPI_Barrier(comm);
```

La función se bloquea hasta que todos los procesos del comunicador la ejecutan.