Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Introducción a MPI / MPI2

Santander, 5 de Noviembre de 2010

Xavier Abellan
BSC Support team

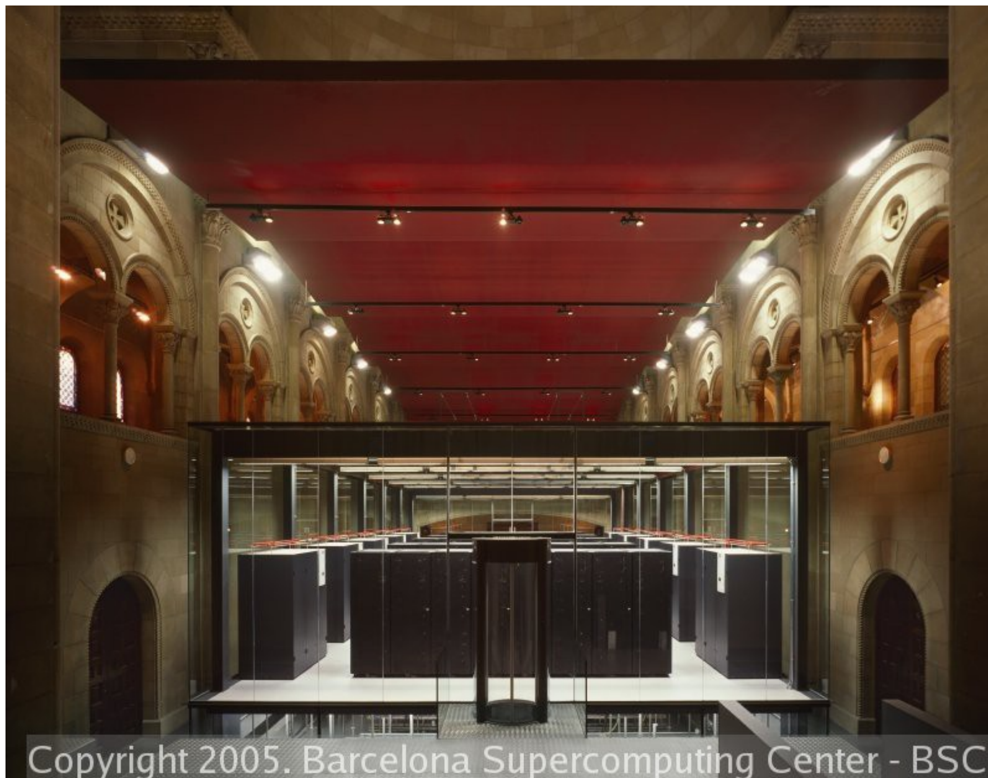
Índice

- Introducción
- Qué es MPI?
  - Conceptos básicos
  - Algunos ejemplos
- Funciones básicas
- Colectivas
- MPI2 y MPI-I/O

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Introducción

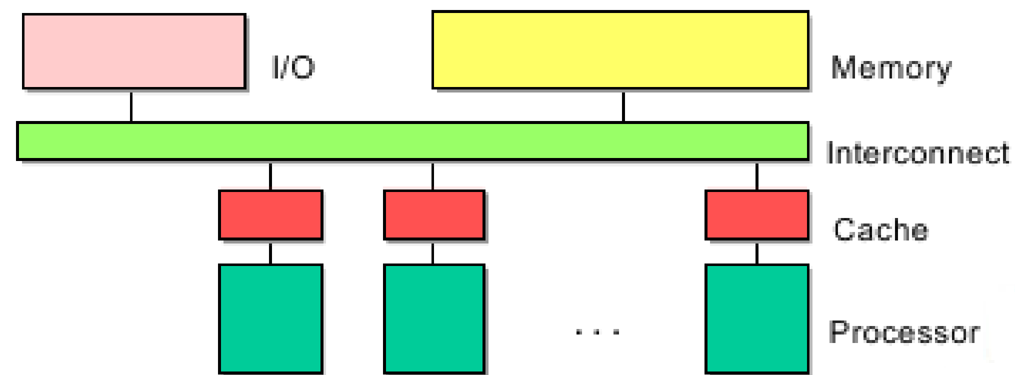
- Supercomputación:
  - Centenares, miles de procesadores
  - MareNostrum: 10240 procesadores

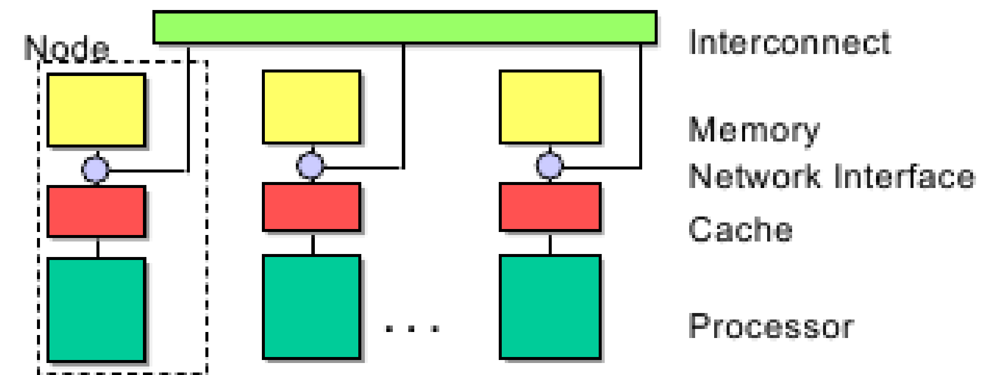


- Los programas secuenciales no sirven…
- **El paralelismo es la única opción!**

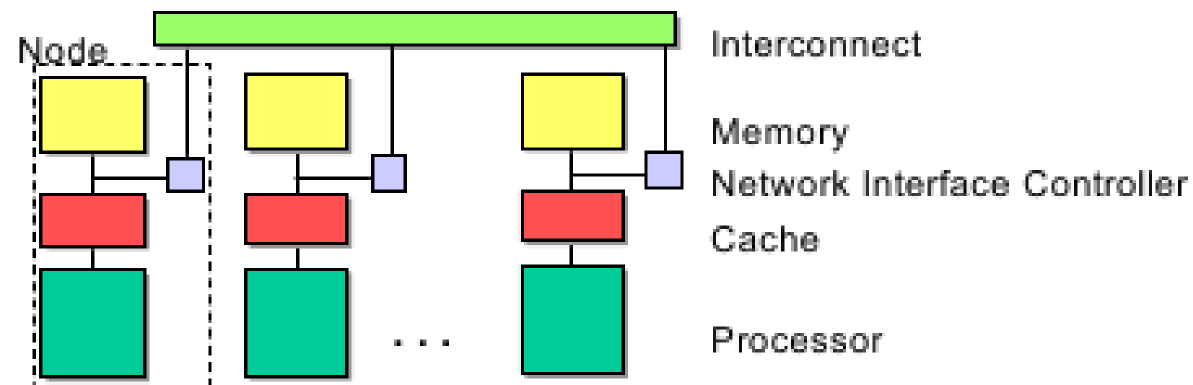
Introducción: Arquitecturas

- Memoria Compartida





- Memoria Distribuida

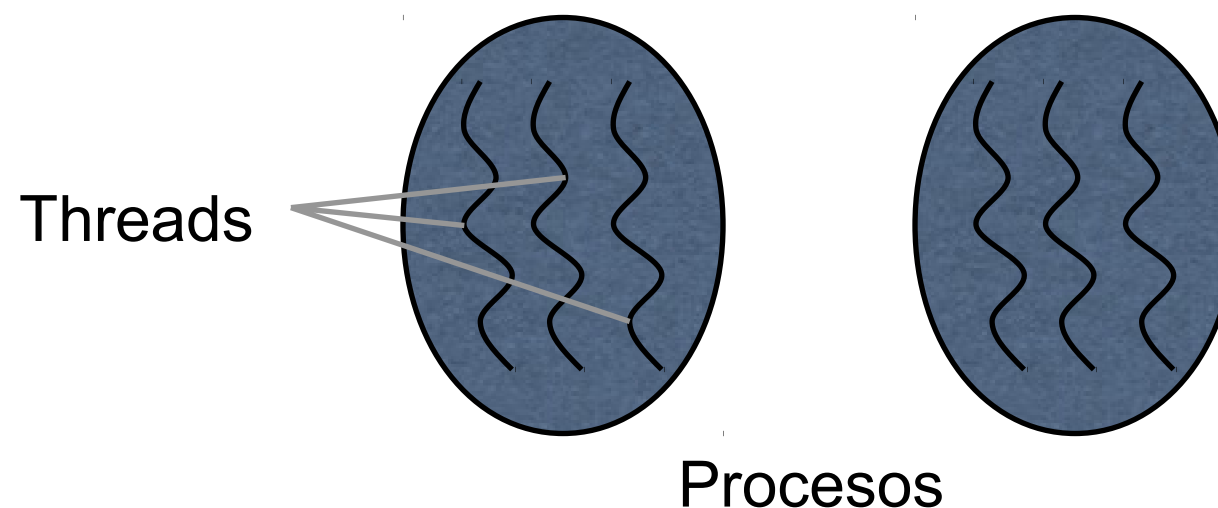


- Solución Híbrida (MareNostrum)

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Introducción: Procesos y threads

- **Proceso**: instancia de un programa en ejecución
  - Contador de programa
  - Espacio de memoria propio
- Un proceso no puede acceder a la memoria de otro proceso
- Un proceso puede ser dividido en **threads** (hilos)
  - Diferentes hilos de ejecución del mismo programa, que pueden ser ejecutados en paralelo
  - Comparten la memoria: todos los threads tienen acceso a toda la memoria del programa.



Introducción: Modelos de programación

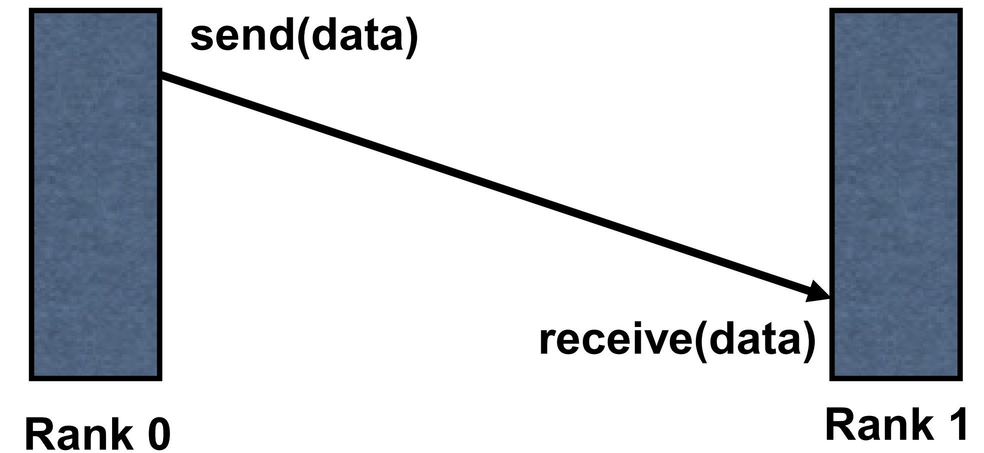
- Memoria compartida
  - Paralelismo de threads
    - Un solo proceso, compartición de datos implícita
    - Pthread
    - OpenMP
- Memoria distribuida
  - Paralelismo de procesos
    - Compartición de datos explícita
    - MPI
- Solución híbrida
  - Procesos + threads
    - MPI + OpenMP

¿Qué es MPI?

- **MPI**: **M**essage **P**assing **I**nterface
- API de programación para comunicación entre procesos
  - Sincronización
  - Movimiento de datos entre los diferentes procesos
- Portable entre diferentes arquitecturas
- Independiente del lenguaje de programación
  - C, C++, Fortran
- Estándar de facto en HPC
  - Especificación del estándar por mpi-forum.org
    - Dos versiones principales:
      - MPI-1 y MPI-2
  - Múltiples implementaciones
    - MPICH, OpenMPI, propietarias de HP, IBM, intel, etc.

¿Qué es MPI?

- **MPI**: **M**essage **P**assing **I**nterface
- La comunicación se realiza mediante el envío y recepción explícita de mensajes



- Los procesos se distribuyen el trabajo
  - Cada proceso trabaja con su parte del problema
- MPI se implementa como una librería
  - Más de 100 funciones diferentes en la versión 1
  - Más de 200 en la versión 2
  - Sólo se suelen utilizar un pequeño subconjunto

Un ejemplo: Hello world!

```
#include <stdio.h>

int main( int argc, char *argv[] )

{

    printf( "Hello, world!\n" );

    return 0;

}
```

Resultado:

```
$>./helloworld

Hello, world!

$>
```


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Un ejemplo: Hello world!

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

Resultado:

```
$> mpirun -np 2 ./helloworld
Hello, world!
Hello, world!
$>
```

Un ejemplo: Hello world!

```
program main
include 'mpif.h'
integer ierr

call MPI_INIT( ierr )
print *, 'Hello, world!'
call MPI_FINALIZE( ierr )
end
```

Resultado:

```
$> mpirun -np 2 ./helloworld
Hello, world!
Hello, world!
$>
```

MPI en C y Fortran

- Bindings en C y Fortran muy parecidos
- In C:
  - Se debe hacer el #include mpi.h
  - Las funciones MPI devuelven MPI_SUCCESS o un código de error
- In Fortran:
  - Se debe hacer el include mpif.h, o usar el MPI module (sólo MPI-2)
  - Todas las llamadas MPI calls se hacen como subrutinas, con el último parámetro para el código de error
- Por defecto, un error hace abortar todos los procesos

BSC Barcelona Supercomputing Center Centro Nacional de Supercomputación

Ejecutar un programa MPI

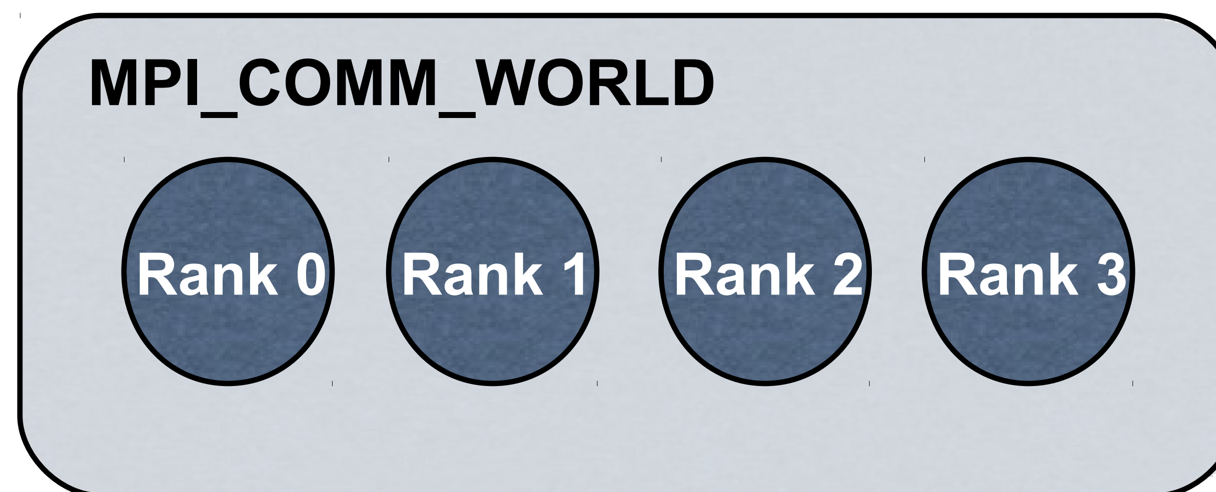
- El estándar mpi no especifica como hay que ejecutar un programa MPI
  - Lo más habitual: mpirun o mpiexec.
  - En la RES: srun

```
#!/bin/bash
# @ job_name        = test_parallel
# @ initialdir      = .
# @ output          = mpi_%j.out
# @ error           = mpi_%j.err
# @ total_tasks     = 4
# @ wall_clock_limit = 00:02:00

srun ./test_mpi
```


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

¿Quién es quién?

- ¿ Cómo sabe cada proceso…
  - cuántos procesos hay en total?
    - **MPI_Comm_size:** número total de procesos en la ejecución
  - cuál de ellos es?
    - **MPI_Comm_rank:** índice del proceso entre 0 y size-1
- Comunicadores
  - Definen un conjunto de tareas dentro de la ejecución
  - MPI_COMM_WORLD contiene todos los procesos



Un ejemplo: Hello world!

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
   int rank, size;

   MPI_Init( &argc, &argv );

   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );

   printf( "I am %d of %d\n", rank, size );

   MPI_Finalize();
   return 0;
}
```

Resultado:

```
$> mpirun -np 4 ./helloworld
I am 0 of 4
I am 3 of 4
I am 1 of 4
I am 2 of 4
$>
```

Datatypes

- Datos a enviar definidos como:
  - Memory address
  - Count
  - Datatype
- Un datatype se define recursivamente como:
  - Predefinido (MPI_INT,MPI_DOUBLE…)
  - Array (contiguo) de datatypes o bloques de datatypes
  - Bloques no contiguos de datatypes
  - Cualquier estructura arbitraria de datatypes
- Existen funciones para crear datatypes personalizados
  - Ej: una fila de una matriz, un array de pares de float e int...

BSC Barcelona Supercomputing Center Centro Nacional de Supercomputación

Funciones básicas

- Muchos programas MPI se pueden escribir con un subconjunto de 6 funciones:
  - MPI_INIT
  - MPI_FINALIZE
  - MPI_COMM_SIZE
  - MPI_COMM_RANK
  - MPI_SEND
  - MPI_RECV

Envío: MPI_Send

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

- El buffer de envío se define con los parámetros buf, count y datatype.
- Dest indica el rango del proceso dentro del comunicador comm a quien va dirigido el mensaje
- Tag es una etiqueta del mensaje para facilitar la identificación del mensaje en el receptor.
  - MPI_ANY_TAG: etiqueta por defecto
- **Blocking send**: retorna cuando se ha completado el envío de la información y se pueden reusar los buffers. El receptor puede no haber completado la recepción del mensaje.

Recepción: MPI_Recv

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

- Espera hasta que recibe un mensaje que coincida con source y tag.
- Source indica el rango del proceso dentro del comunicador comm de quien envía el mensaje.
  - MPI_ANY_SOURCE: acepta cualquier emisor
- Tag es una etiqueta del mensaje para facilitar la identificación del mensaje en el receptor.
  - MPI_ANY_TAG: acepta cualquier tag
- Se pueden recibir menos datos que count, pero no más.
- Status contiene más información sobre la operación
  - Estado de la recepción, datos leídos, etc…

Más sobre sends y recvs…

- Todo send debe tener su recv correspondiente.
  - Si se hace recv de un mensaje que no se llega a enviar, el programa entra en “deadlock”.
  - Si no hay suficiente espacio en el receptor, el send debe esperar a que se produzca el recv
- Código problemático:

- Soluciones:
  - Reordenar los envíos y recepciones
  - Usar MPI_Sendrecv
  - Non-blocking operations

Más sobre sends y recvs…

- Modos de comunicación
  - Standard
  - Buffered: el usuario proporciona un buffer al sistema para hacer el envío
  - Synchronous: el send no termina hasta que empieza el recv correspondiente
  - Ready: el usuario garantiza que en el momento del send ya hay un recv esperando el dato
    - Permite optimizaciones en el envío
    - Comportamiento indeterminado si el recv no se ha ejecutado
- Sincronización local
  - Blocking (estándar)
  - Non-blocking (inmediate)

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Más sobre sends y recvs…

- Modos de comunicación
  - Standard
  - Buffered: el usuario proporciona un buffer al sistema para hacer el envío
  - Synchronous: el send no termina hasta que empieza el recv correspondiente
  - Ready: el usuario garantiza que en el momento del send ya hay un recv esperando el dato
    - Permite optimizaciones en el envío
    - Comportamiento indeterminado si el recv no se ha ejecutado
- Sincronización local
  - Blocking (estándar)
  - Non-blocking (inmediate)

BSC Barcelona Supercomputing Center Centro Nacional de Supercomputación

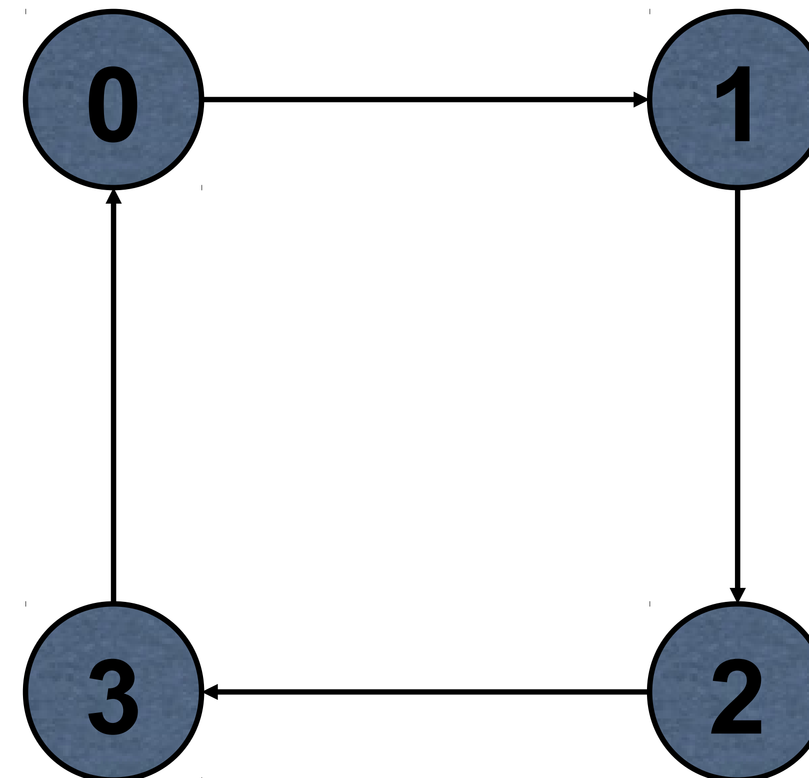
Más sobre sends y recvs…

```
MPI_Send (buf, count, datatype, dest, tag, comm)
MPI_Bsend(buf, count, datatype, dest, tag, comm)
MPI_Rsend(buf, count, datatype, dest, tag, comm)
MPI_Ssend(buf, count, datatype, dest, tag, comm)
MPI_Isend (buf, count, datatype, dest, tag, comm, request)
MPI_Ibsend(buf, count, datatype, dest, tag, comm, request)
MPI_Issend(buf, count, datatype, dest, tag, comm, request)
MPI_Irsend(buf, count, datatype, dest, tag, comm, request)
```

- Request: parámetro que gestiona la comunicación
  - MPI_Test/Wait(request) para comprobar/esperar si se ha completado y liberar buffers

Ejemplo: Anillo

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

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

  MPI_Status status;
  int rank, size, next, from, tag, num, howmany, whoAmI;

  num=tag=0;

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  next = (rank + 1) % size;
  from = (rank + size - 1) % size;

  if (rank == 0) {

    printf("Hello! I am process %d\n",rank);

    MPI_Send(&num,1,MPI_INT,next,tag,MPI_COMM_WORLD);
    MPI_Recv(&num,1,MPI_INT,from,tag,MPI_COMM_WORLD,&status);

  } else {

    MPI_Recv(&num,1,MPI_INT,from,tag,MPI_COMM_WORLD,&status);

    printf("Hello! I am process %d\n",rank);

    MPI_Send(&num,1,MPI_INT,next,tag,MPI_COMM_WORLD);
 }

 MPI_Finalize();
 return 0;

}
```



Resultado:

```
$> mpirun -np 4 ./helloworld
Hello! I am process 0
Hello! I am process 1
Hello! I am process 2
Hello! I am process 3
$>
```

Colectivas

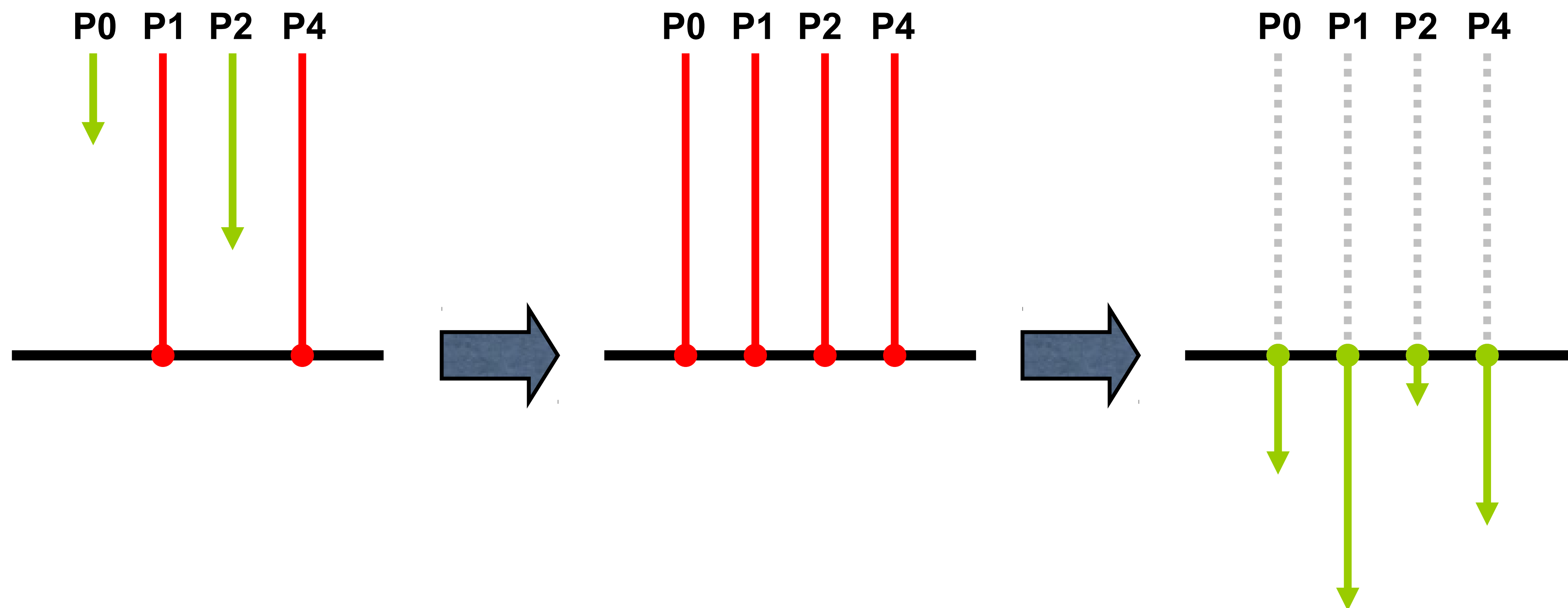
- Comunicación y cálculo coordinadas en un grupo de procesos (comunicador).
- Todos los procesos del comunicador deben llamar la función.
- Reemplazan multiples llamadas de send/recv
- Facilitan la entendibilidad del código
- La implementación puede optimizar la comunicación.
- Todas las colectivas son blocking.
- Tres clases de colectivas:
  - Sincronización
  - Movimiento de datos
  - Cálculo colectivo

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Sincronización: MPI_Barrier

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

- Bloqueo hasta que todos los procesos del comunicador llaman a la función

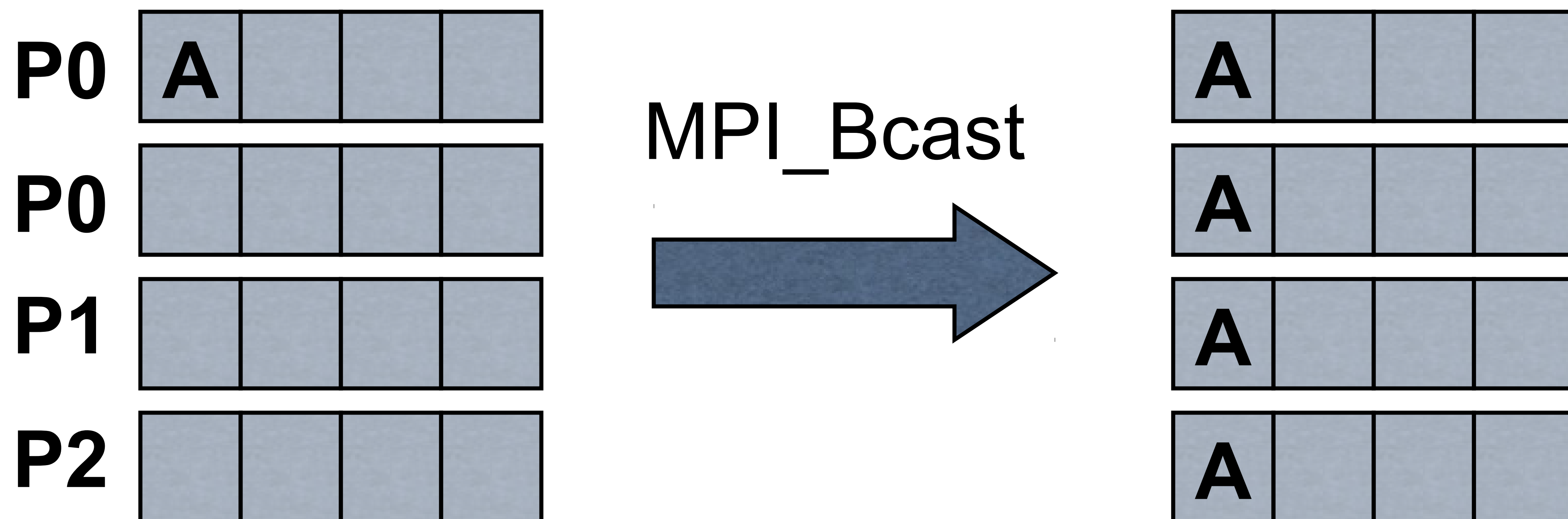


BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Movimiento de datos: Broadcast

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

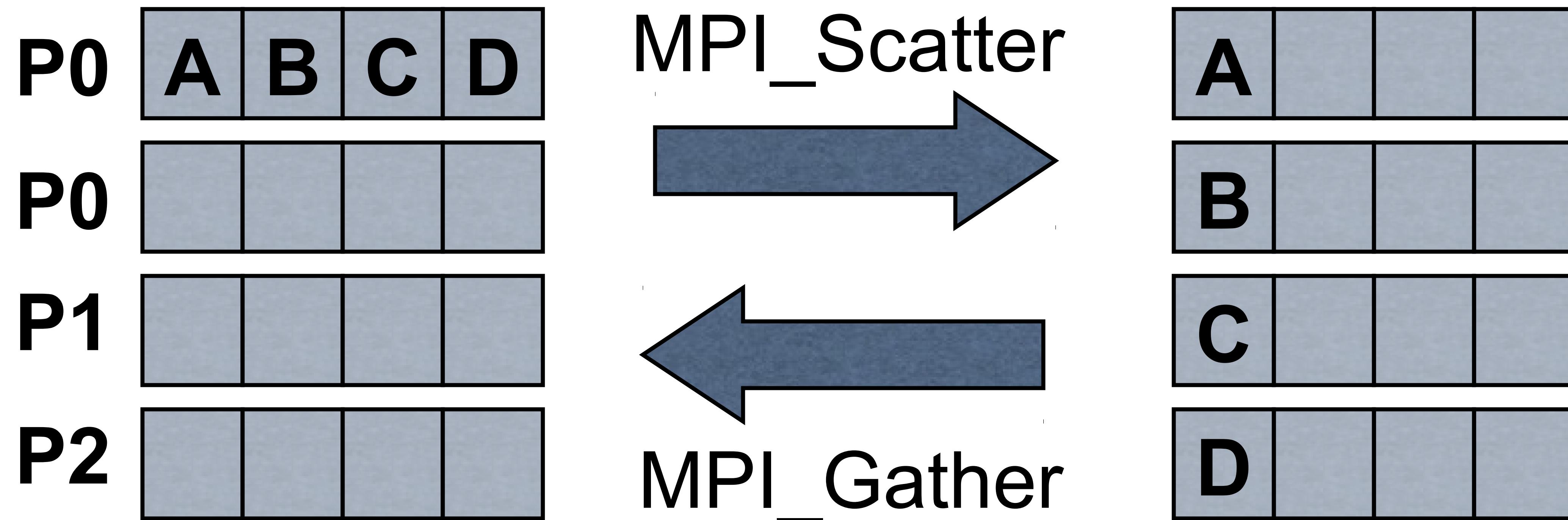
- Envío de información des del proceso root hacia todos los procesos del comunicador



BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Movimiento de datos: Gather / Scatter

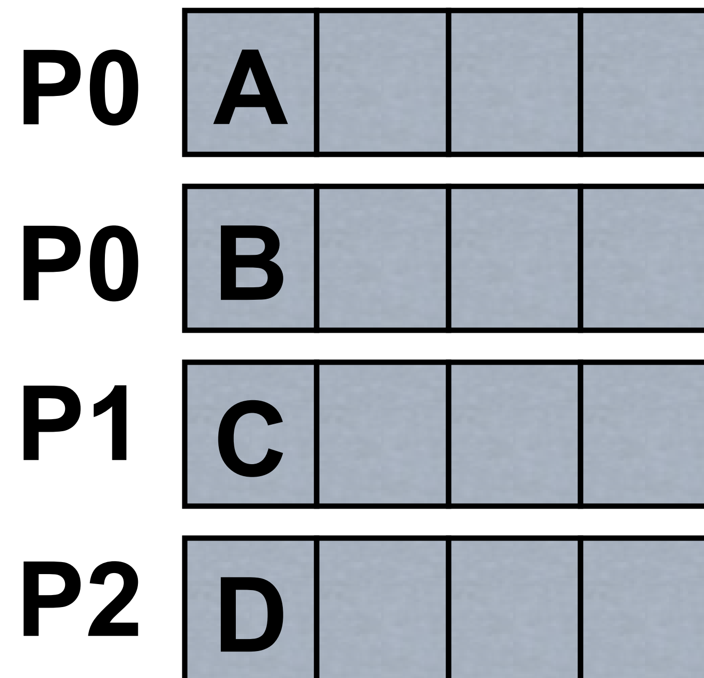
```
int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void
  *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm )

int MPI_Scatter ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void
  *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm )
```



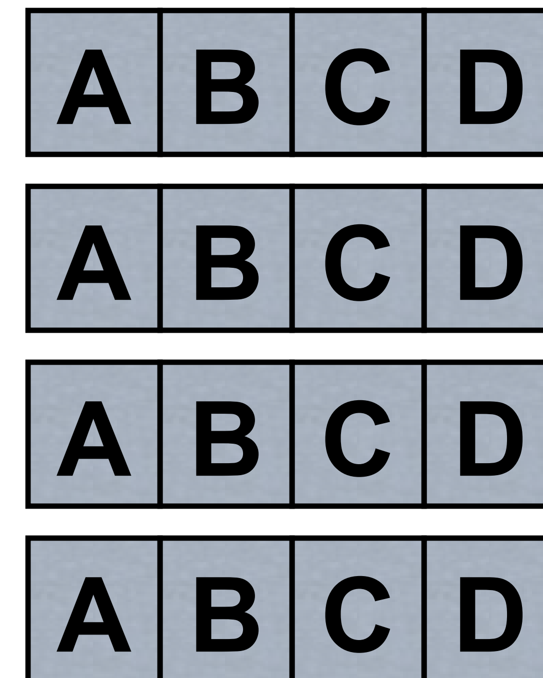
BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Movimiento de datos: Allgather

```
int MPI_Allgather(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount,
  MPI_Datatype rdtype,                  MPI_Comm comm)
```



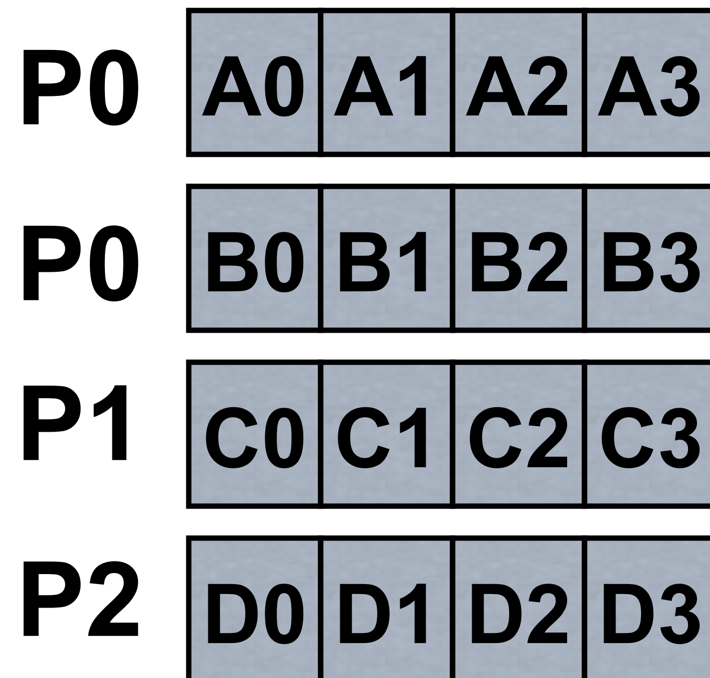
MPI_Allgather



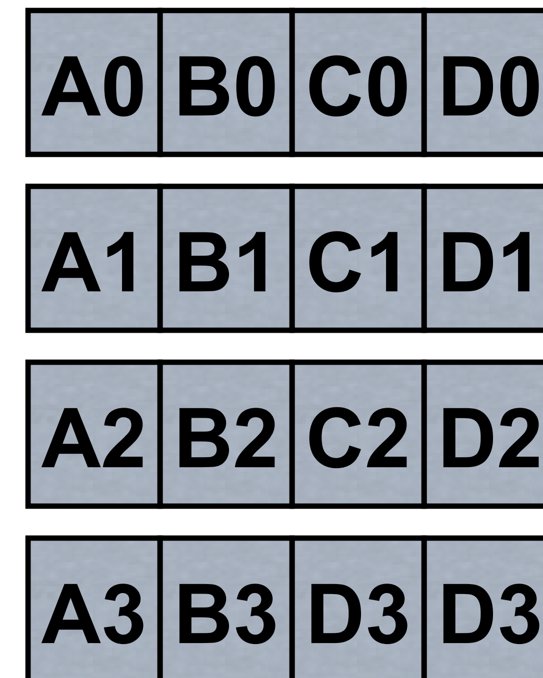
BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Movimiento de datos: Alltoall

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



MPI_Alltoall

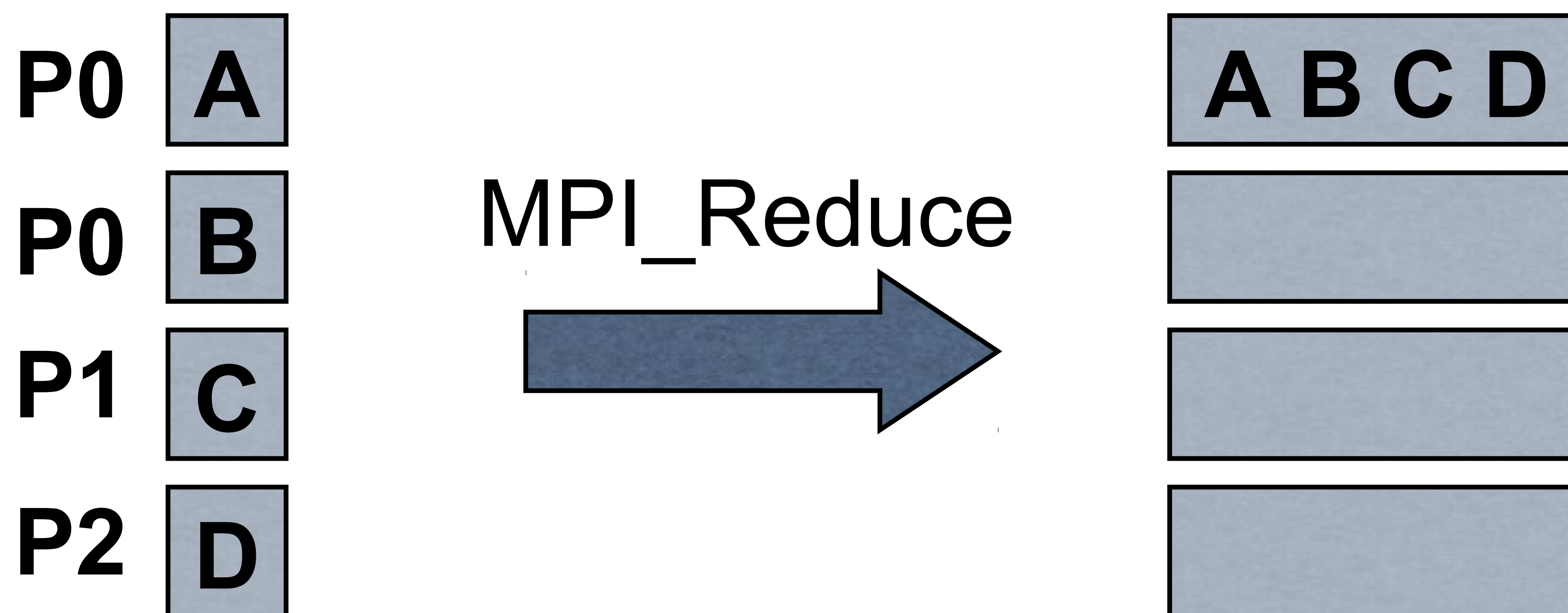


BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Calculo colectivo: Reduce

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

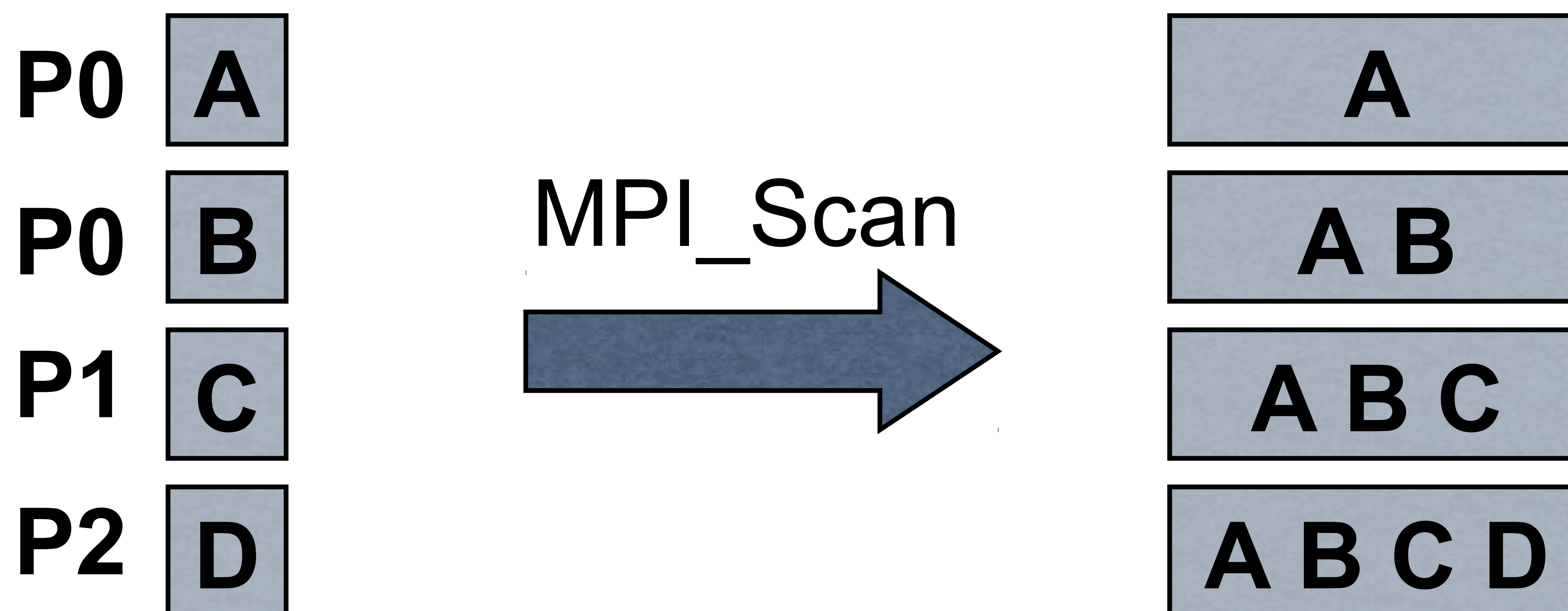
- Permiten hacer cálculos sencillos y comunicación en una sola llamada



Calculo colectivo: Scan

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

- Permiten hacer cálculos sencillos y comunicación en una sola llamada



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Calculo colectivo: Funciones

`MPI_Max`	Maximum
`MPI_Min`	Minimum
`MPI_Prod`	Product
`MPI_Sum`	Sum
`MPI_Land`	Logical and
`MPI_Lor`	Logical or
`MPI_Lxor`	Logical exclusive or
`MPI_Band`	Binary and
`MPI_Bor`	Binary or
`MPI_Bxor`	Binary exclusive or
`MPI_Maxloc`	Maximum and location
`MPI_Minloc`	Minimum and location

Más sobre colectivas

- MPI proporciona más funciones colectivas
- Muchas funciones derivadas de otras existentes:
  - All: proporcionan los resultados a todos los procesos
    - MPI_Allgather, MPI_Allreduce, etc.
  - V: permiten que las partes a enviar sean de distinto tamaño
    - MPI_Gatherv, MPI_Alltoallv, etc.
- Se pueden definir funciones de reducción personalizadas
  - MPI_Op_create, MPI_Op_free
- Las operaciones de reducción pueden no siempre dar el mismo resultado exacto.
  - Redondeo de floats, etc.

MPI 2: qué hay de nuevo

- Dynamic process management
  - Arrancar procesos MPI en runtime
  - Conectarse a un proceso MPI arrancado separadamente
- One-sided communications
  - Acceso directo a la memoria de otro proceso (window)
- MPI I/O (Paralelo)
- Mejor soporte para threads
  - SINGLE, FUNNELED, SERIALIZED, MULTIPLE
- Mejoras en el soporte para Fortran 9x y C++
- Mejoras en los tipos de datos definidos por el usuario
  - Necesarios para usos avanzados de MPI I/O

Estrategias de I/O para MPI

- Estrategias de I/O en programas MPI:
  - El proceso 0 escribe/lee el fichero (llamadas estándar) y los demás se comunican con éste
  - Todos leen /escriben en ficheros diferentes (llamadas estándar), que después se tienen que combinar de alguna forma
  - Todos escriben en el mismo fichero en paralelo (llamadas MPI I/O)
    - Escalable
    - Estándar

BSC Barcelona Supercomputing Center Centro Nacional de Supercomputación

MPI I/O: Open / Close

- Todos los procesos de un comunicador abren un fichero:
  - `int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh);`
    - comm: comunicador sobre el que se va a hacer el I/O
    - amode
      - **MPI_MODE_RDONLY, MPI_MODE_WRONLY, MPI_MODE_CREATE**, etc.
      - Pueden combinarse (+ en Fortran, | en C/C++)
    - info:
      - ‘Hints’ a la implementación para mejorar el rendimiento
      - **MPI_INFO_NULL** (por defecto)
    - fh: parallel file handler
  - **MPI_File_close(fh)**

MPI I/O: Read

- El fichero debe haber sido abierto con **MPI_MODE_RDONLY/RDWR**
- Leer fichero :
  - `int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
  - Actualiza el puntero de fichero después de leer
  - no es thread safe
- Leer fichero en un punto concreto:
  - `int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
  - Thread safe
- Cambiar el puntero del fichero:
  - `MPI_File_seek(MPI_File *fh, MPI_Offset offset, int whence);`
    - offset: offset en bytes (con default file view)
    - whence: MPI_SEEK_SET, MPI_SEEK_CUR, MPI_SEEK_END, etc
- Bytes leidos con **MPI_Get_count**

BSC Barcelona Supercomputing Center Centro Nacional de Supercomputación

MPI I/O: Read

```
MPI_File fh;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);

MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);

MPI_File_read(fh, buf, nints, MPI_INT, &status);

MPI_File_close(&fh);
```


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MPI I/O: Write

- Similar a la lectura
- El fichero se abre con **MPI_MODE_WRONLY/CREATE/RDWR**

  Escribir en el fichero:
  - ```
    int MPI_File_write(MPI_File fh, void *buf, int count,
                       MPI_Datatype datatype, MPI_Status
    *status)
    ```
  - Actualiza el puntero de fichero después de escribir
  - no es thread safe
- Escribir fichero en un punto concreto:
  - ```
    int MPI_File_write_at(MPI_File fh, MPI_Offset offset,
    void *buf, int count, MPI_Datatype datatype, MPI_Status
    *status)
    ```
  - Thread safe

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MPI I/O: contiguous vs non-contiguous access

- Contiguous access
  - Cada proceso accede a una parte contigua de los datos
  - Muy parecido a un read/write de UNIX
- Non-contiguous access
  - Cada proceso accede a diferentes partes de los datos en diferentes puntos de un fichero
  - Puede ser muy costoso implementado con reads/writes separados
  - Se usarán File Views para implementar acceso no contiguo

MPI I/O: File views

- Define qué parte de un fichero es visible a un proceso
  - Partes no contiguas definidas con User defined datatypes
  - Definidos como una secuencia de datatypes nativos y una secuencia de desplazamientos
- Define el tipo de datos al que se accede
  - Portabilidad
- Vista por defecto
  - Todo el fichero es visible
  - Los offsets son en bytes

MPI I/O: File views

- `int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info)`
  - disp: Offset des del inicio del fichero (en bytes)
  - etype:
    - MPI type o user defined type
    - Unidad básica de acceso (Offsets en unidades de este tipo)
  - filetype:
    - El mismo tipo que etype o un nuevo tipo derivado formado por etypes
    - Especifica qué parte del fichero es visible
  - datarep
    - Data representation (para portabilidad)
    - “native”: lo guarda en el mismo formato que en memoria
  - Info:
    - “Hints” para la implementación
    - **MPI_INFO_NULL**: No hints

MPI I/O: File views
contiguous example

```
MPI_File thefile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;

MPI_File_open(MPI_COMM_WORLD, "testfile",
     MPI_MODE_CREATE | MPI_MODE_WRONLY,
     MPI_INFO_NULL, &thefile);

MPI_File_set_view(thefile, myrank * BUFSIZE *
 sizeof(int),
         MPI_INT, MPI_INT, "native",  MPI_INFO_NULL);

MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
      MPI_STATUS_IGNORE);

MPI_File_close(&thefile);
```

MPI I/O: File views
non-contiguous example

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

lb = 0; extent = 6 * sizeof(int);

MPI_Type_contiguous(2, MPI_INT, &contig);

MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);

disp = 5 * sizeof(int); etype = MPI_INT;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
     MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);

MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);

MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

MPI I/O: Colectivas

- Colectivas para lectura y escritura
  - `MPI_File_read_all`
  - `MPI_File_write_all`
- Mismos parametros que en las equivalentes individuales
  - `MPI_File_read`
  - `MPI_File_write`
- Todos los procesos en el comunicador deben llamar la función
- Mejoras en el rendimiento comparado con las llamadas individuales.
- Siempre que sea posible, utilizar colectivas

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MPI I/O: Shared pointers

- En ciertas operaciones, el valor del puntero es compartido entre todos los procesos del comunicador
  - Si alguno lee/escribe, se actualiza para todos
- Operaciones:
  - Bloqueantes: MPI_File_seek/write/read_shared
  - No bloqueantes: MPI_File_iwrite/iread_shared
  - Colectivas: MPI_File_read/write_ordered
- Siguen siendo paralelas
- Útiles p.e. para llevar un log de la aplicación

MPI I/O: Hints

- ROMIO: Implementación MPI I/O de MPICH2
- Optimizaciones:
  - Data sieving
    - ind_rd_buffer_size, ind_wr_buffer_size, romio_ds_read, romio_ds_write.
  - Two-phase I/O
    - cb_config_list, cb_nodes, cb_buffer_size, romio_cb_read, romio_cb_write
- Hints específicas para filesystems
  - PFS, XFS, PVFS, Lustre, PanFS
  - No hay para GPFS
- Se pueden definir de forma global con el fichero /etc/romio-hints o la variable ROMIO_HINTS

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MPICH2-MX @ RES

- Última versión instalada en RES:
  - 1.0.8p1..3
- Capaz de usar el PMI de SLURM (libpmi.so)
  - Arranca los procesos con srun
- No implementa:
  - Dynamic process management (connect/accept, comm_spawn[_multiple])
  - One-sided operations
  - MPI_THREAD_MULTIPLE support

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Referencias

- MPI standard
  - http://www.mcs.anl.gov/research/projects/mpi/
- MPI Forum
  - http://www.mpi-forum.org/

Gracias por vuestra atención