

Computación de alta performance

Sergio Nesmachnow (sergion@fing.edu.uy)

Universidad de la República, Uruguay

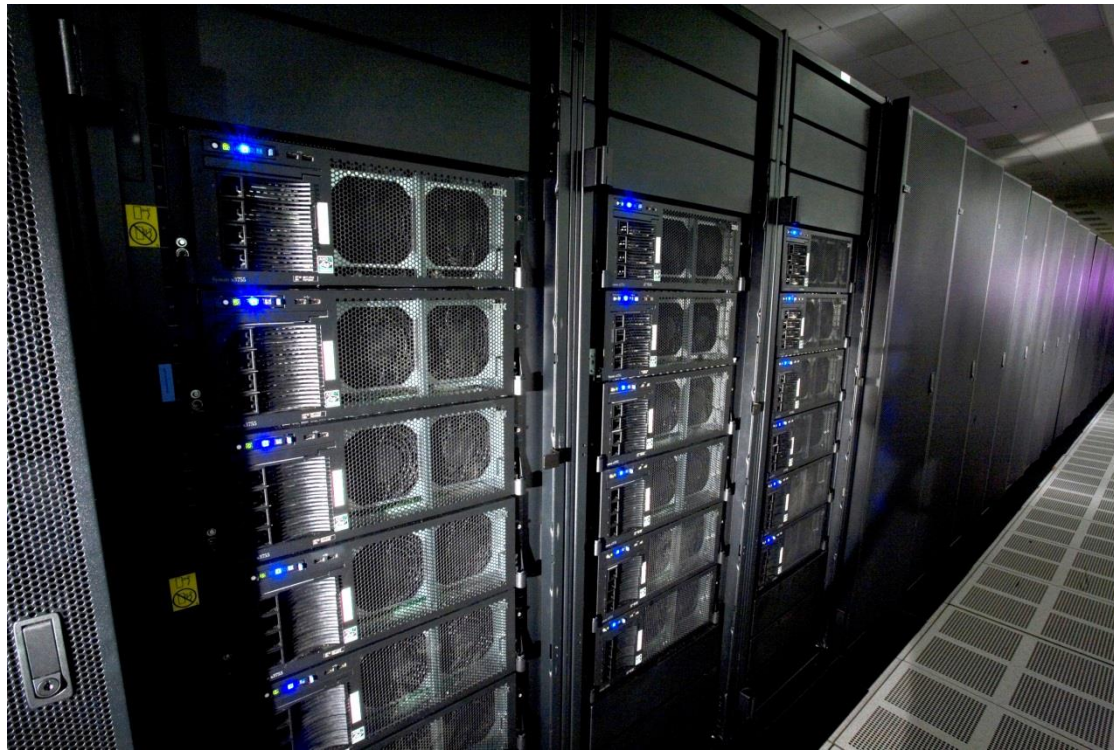


Message Passing Interface (MPI)

Sergio Nesmachnow (sergion@fing.edu.uy)

Universidad de la República, Uruguay





MPI (continuación)



Operaciones colectivas

- Las comunicaciones colectivas permiten la transferencia de datos entre todos los procesos que pertenecen a un grupo específico
- No se usan etiquetas para los mensajes, estas se sustituyen por identificadores de los grupos (comunicadores)

Las operaciones colectivas comprenderán a todos los procesos en el alcance del comunicador

- Por defecto, todos los procesos se incluyen en el comunicador genérico `MPI_COMM_WORLD`
- Las operaciones colectivas son bloqueantes
- Las operaciones colectivas que involucran un subconjunto de procesos deben precederse de un particionamiento de los subconjuntos y relacionar los nuevos grupos con nuevos comunicadores



Operaciones colectivas

- Se clasifican en tres clases:
 1. **Sincronización** (operaciones de barrera): procesos que esperan a que otros miembros del grupo alcancen el punto de sincronización
 2. **Movimiento** (transferencia) de datos: operaciones para difundir, recolectar y esparcir datos entre procesos
 3. **Cálculos colectivos**: operaciones para reducción global, tales como suma, máximo, mínimo o cualquier función definida por el usuario

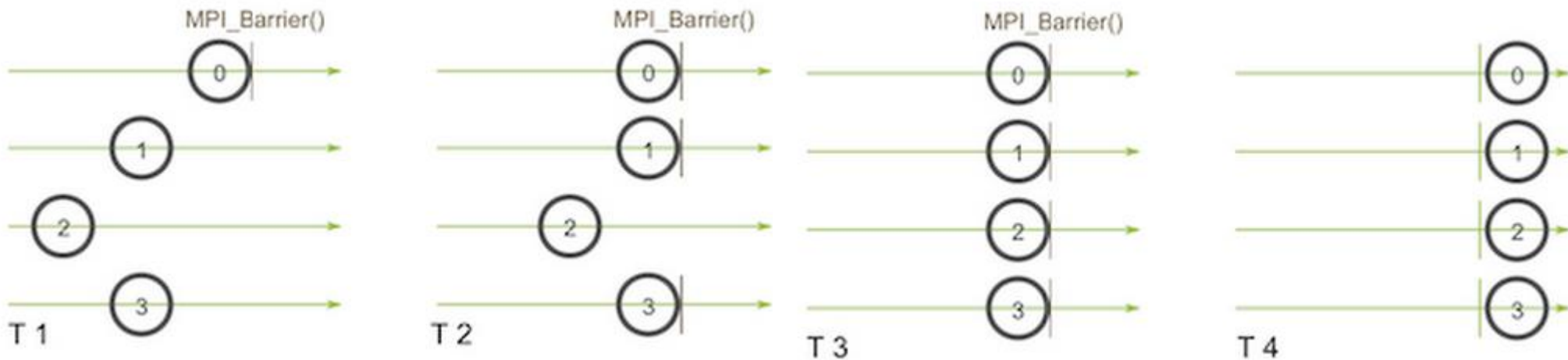


Barrier

- MPI_Barrier (comm)

Crea una barrera de sincronización en un grupo

Al llegar a la invocación a la operación, cada tarea se bloquea hasta que todas las tareas del grupo alcancen la invocación de MPI_Barrier



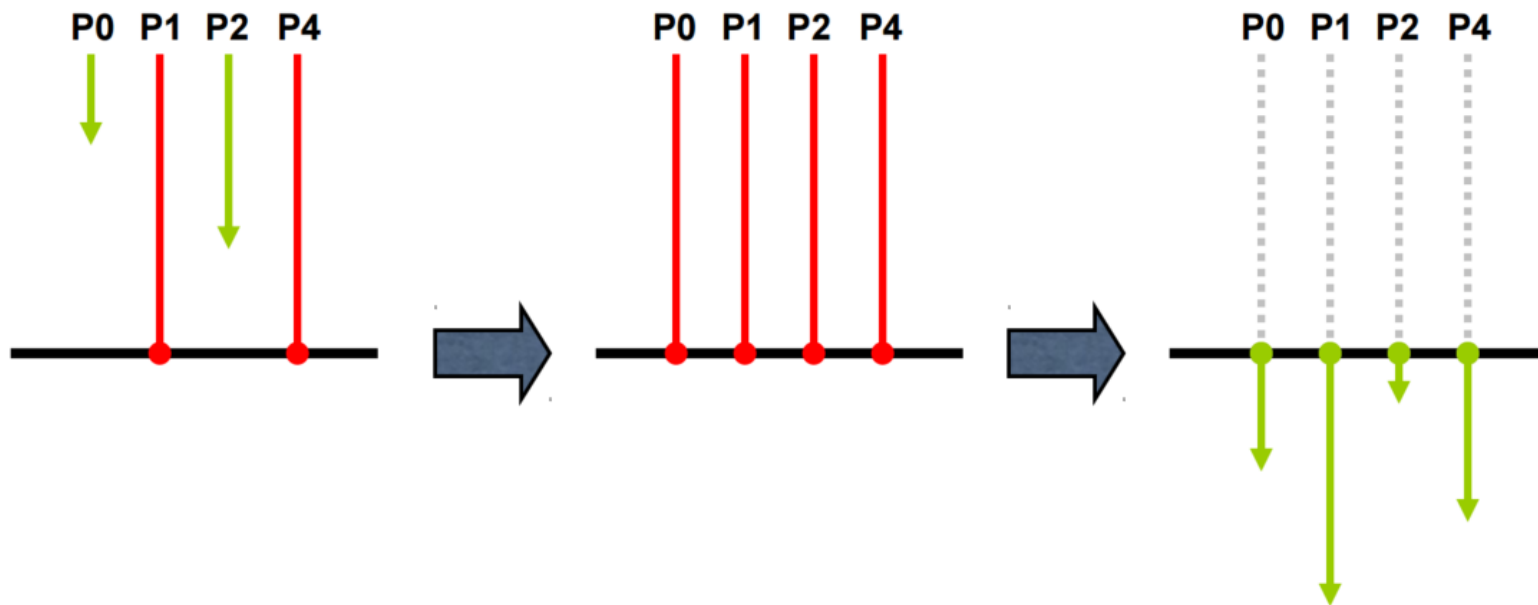
Barrier

- `MPI_Barrier (comm)`

Crea una barrera de sincronización en un grupo

Al llegar a la invocación a la operación, cada tarea se bloquea hasta que todas las tareas del grupo alcancen la invocación de `MPI_Barrier`

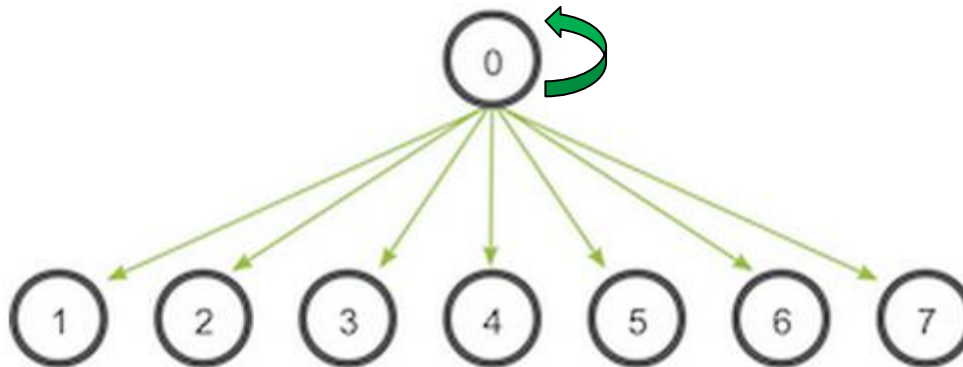
Muy útil en procesamiento asincrónico





Broadcast

- `MPI_Bcast` (`buffer`, `count`, `datatype`, `root`, `comm`)
Envía un mensaje desde el proceso con rango `root` a todos los demás procesos del comunicador

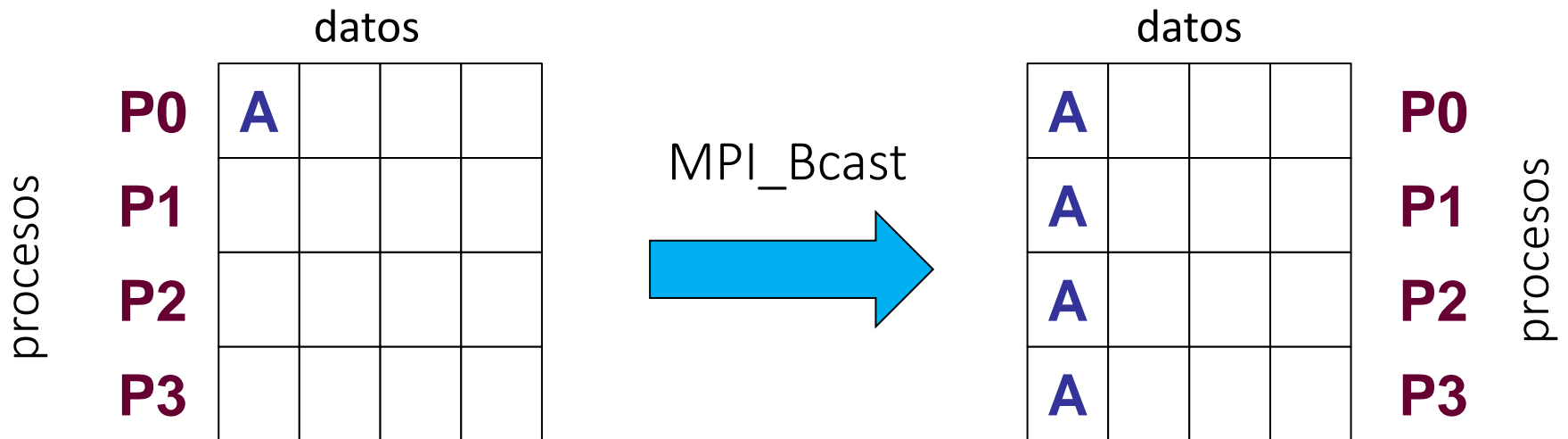


El broadcast también envía el mensaje al proceso root !!



Broadcast

- `MPI_Bcast` (`buffer`, `count`, `datatype`, `root`, `comm`)
Envía un mensaje desde el proceso con rango `root` a todos los demás procesos del comunicador





Broadcast

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

int main(int argc, char *argv[]){
    int i,myid,numprocs,root,count;
    int buffer[4];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    root=0; count=4;
    if(myid == root){
        for(i=0;i<count;i++)
            buffer[i]=i;
    }
    MPI_Bcast(buffer,count,MPI_INT,root,MPI_COMM_WORLD);
    for(i=0;i<count;i++){printf("%d ",buffer[i]);}
    printf("\n");
    MPI_Finalize();
}
```

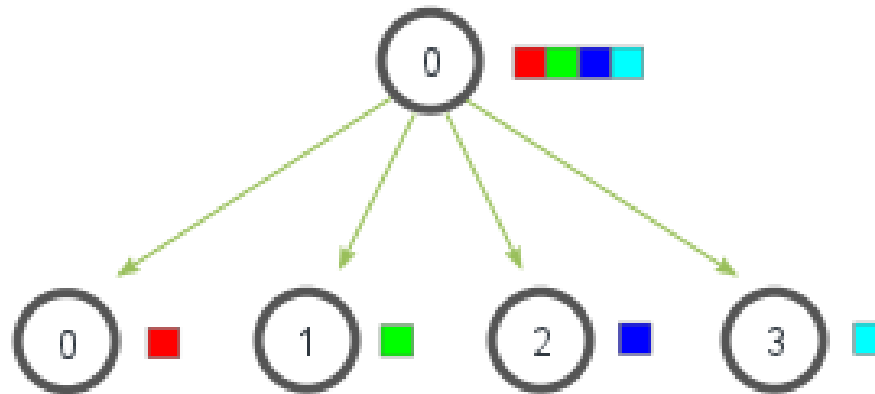
MPI_Bcast funciona “como send y receive” para el proceso root y como “receive” para el resto de los procesos



Scatter: esparcir/distribuir

- Un proceso distinguido (root) rompe un arreglo de datos y envía las partes a cada proceso

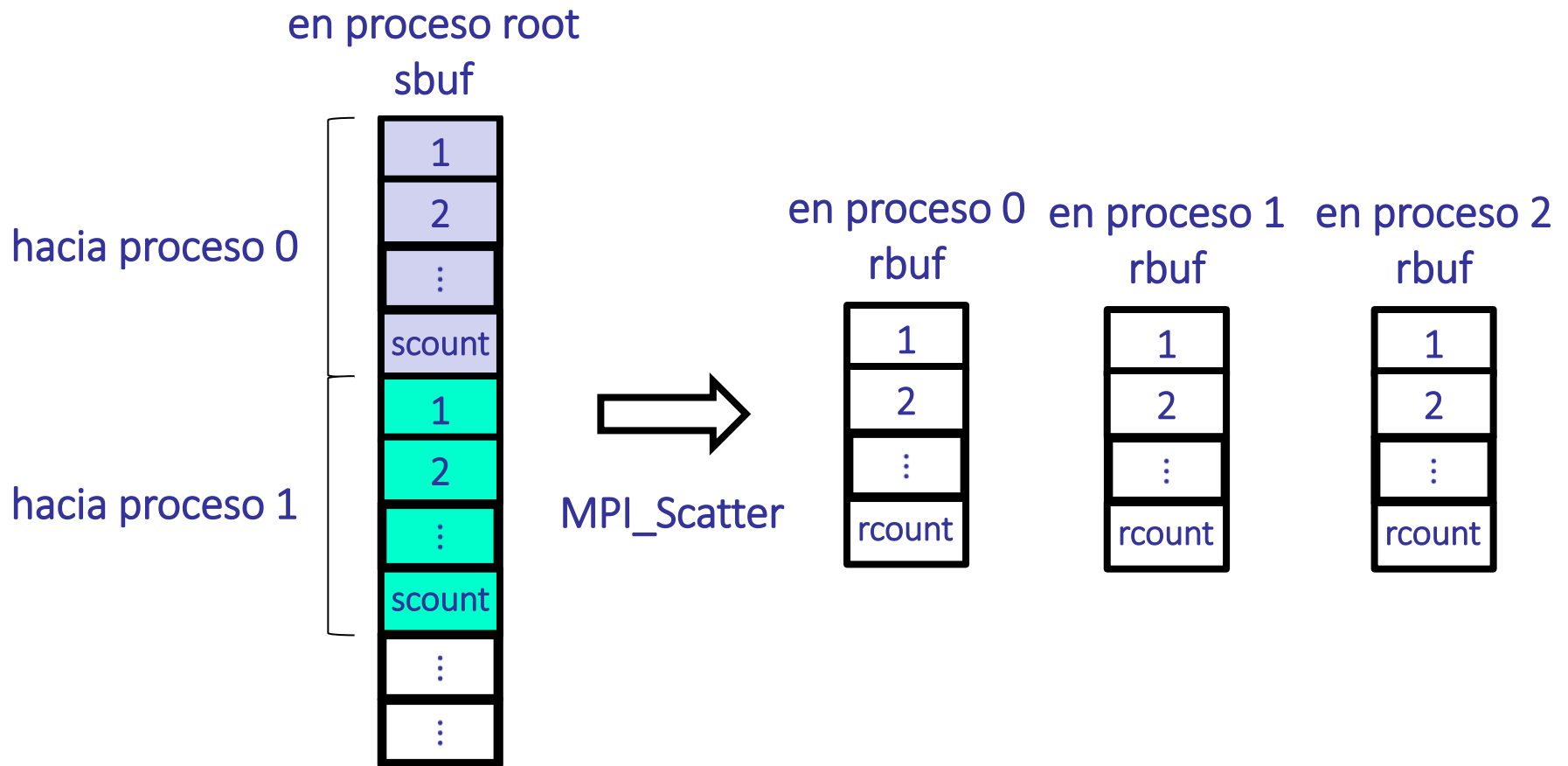
`MPI_Scatter (void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype recv_datatype, int root, MPI_Comm communicator)`





Scatter: esparcir/distribuir

- Un proceso distinguido (root) rompe un arreglo de datos y envía las partes a cada proceso

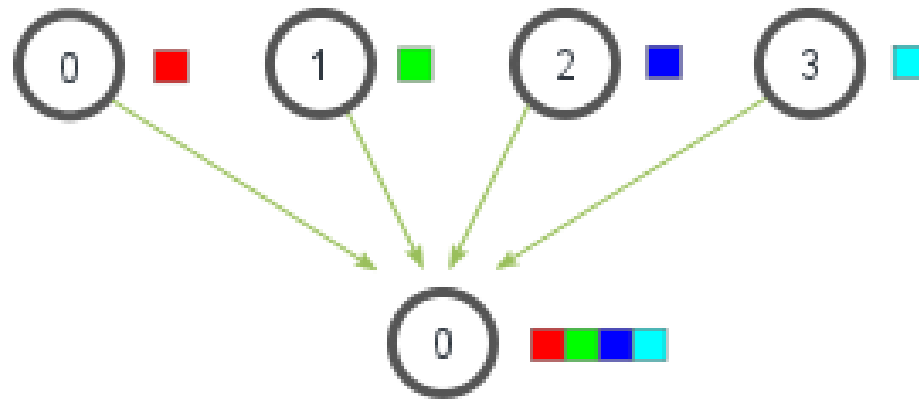




Gather: recolectar

- Cada proceso envía diferentes datos al proceso distinguido (root) y este los agrupa en un arreglo

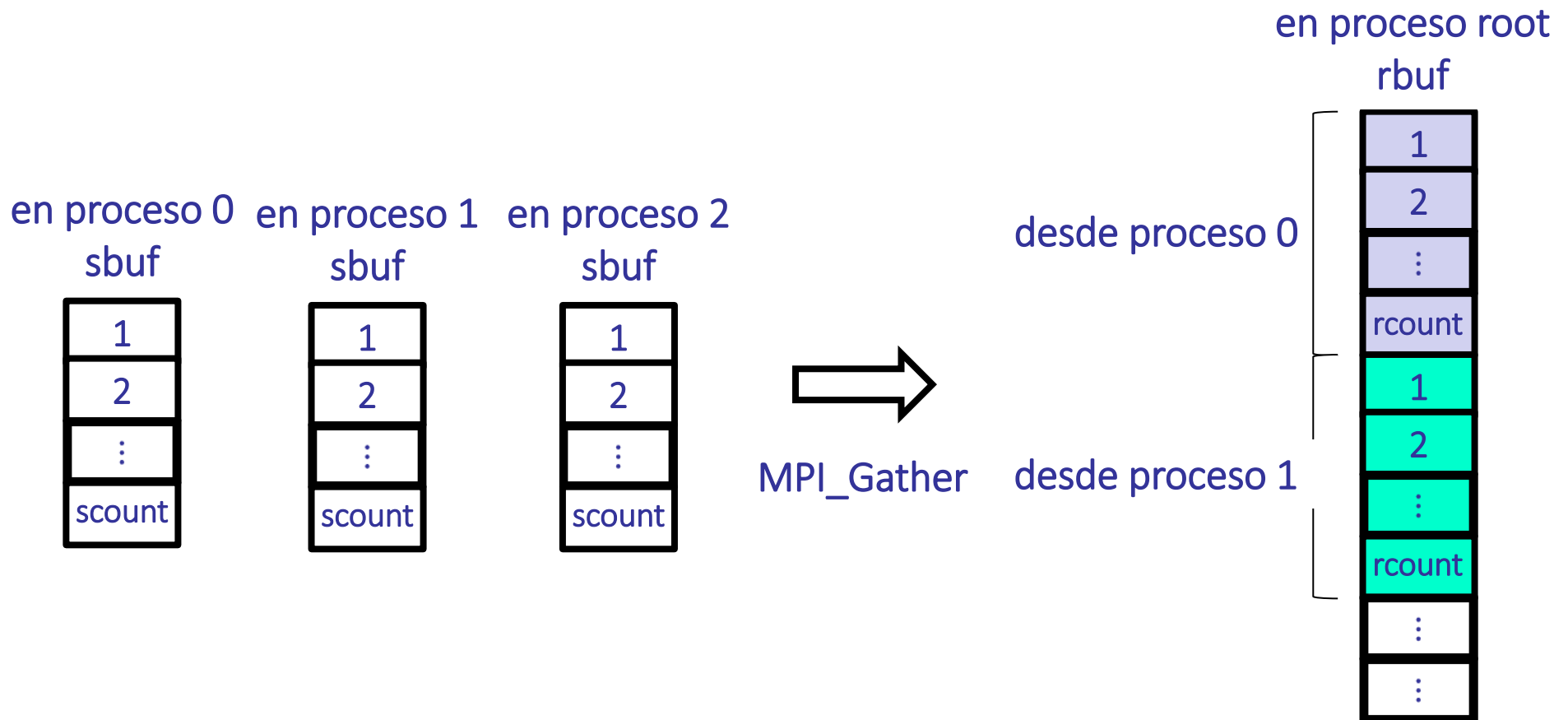
`MPI_Gather` (`void*` send_data, `int` send_count, `MPI_Datatype` send_datatype, `void*` recv_data, `int` recv_count, `MPI_Datatype` recv_datatype, `int` root, `MPI_Comm` communicator)





Gather: recolectar

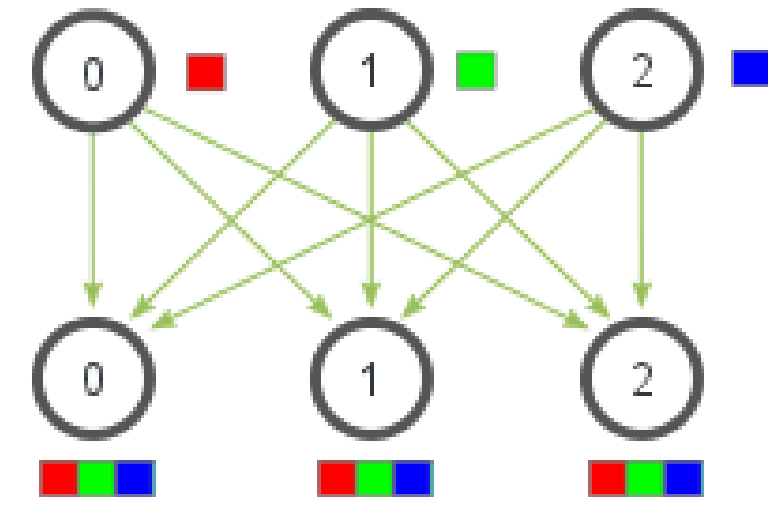
- Cada proceso envía diferentes datos al proceso distinguido (root) y este los agrupa en un arreglo





Allgather: recolectar y difundir

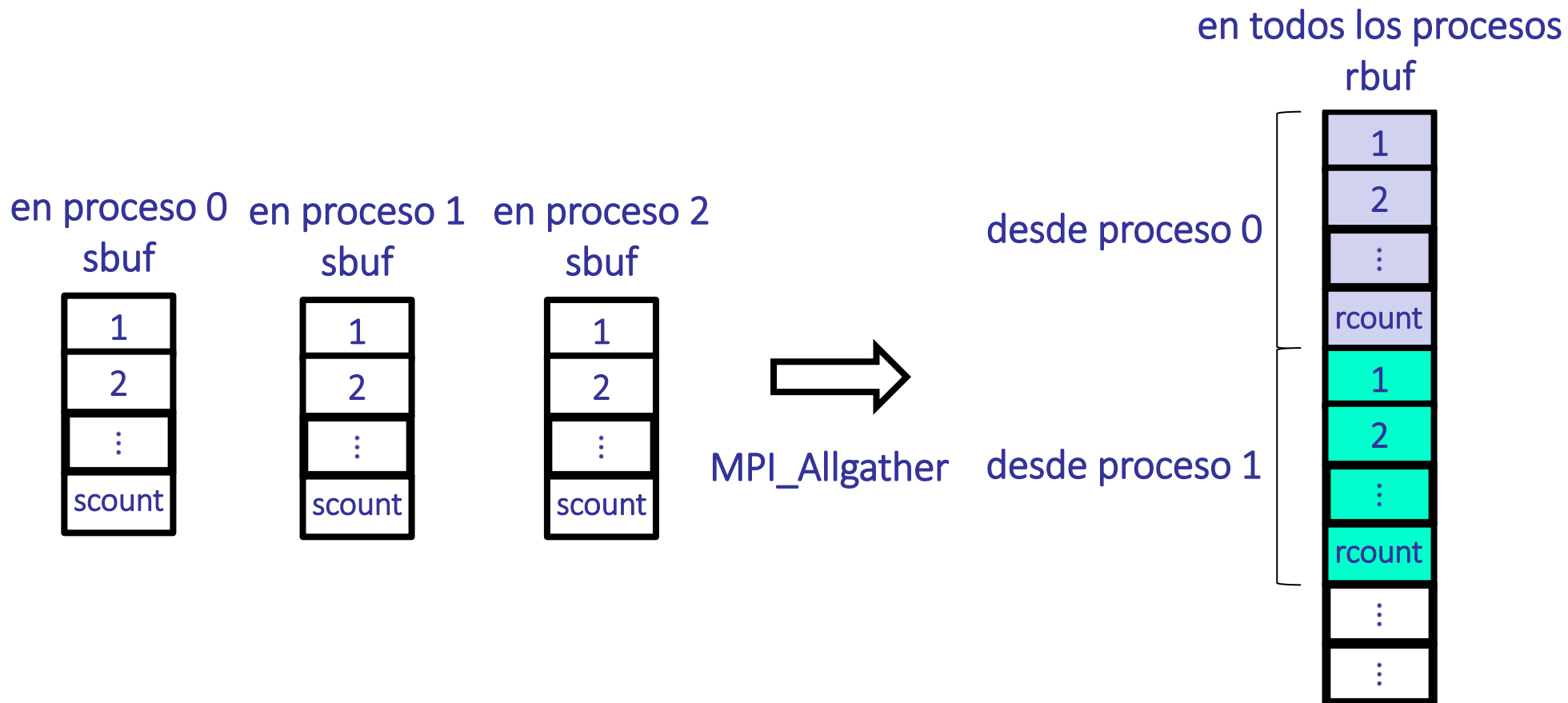
- Funciona como gather, pero TODOS los procesos reciben el resultado
MPI_Allgather (void* send_data, int send_count, MPI_Datatype
send_datatype, void* recv_data, int recv_count,
MPI_Datatype recv_datatype, MPI_Comm
communicator)





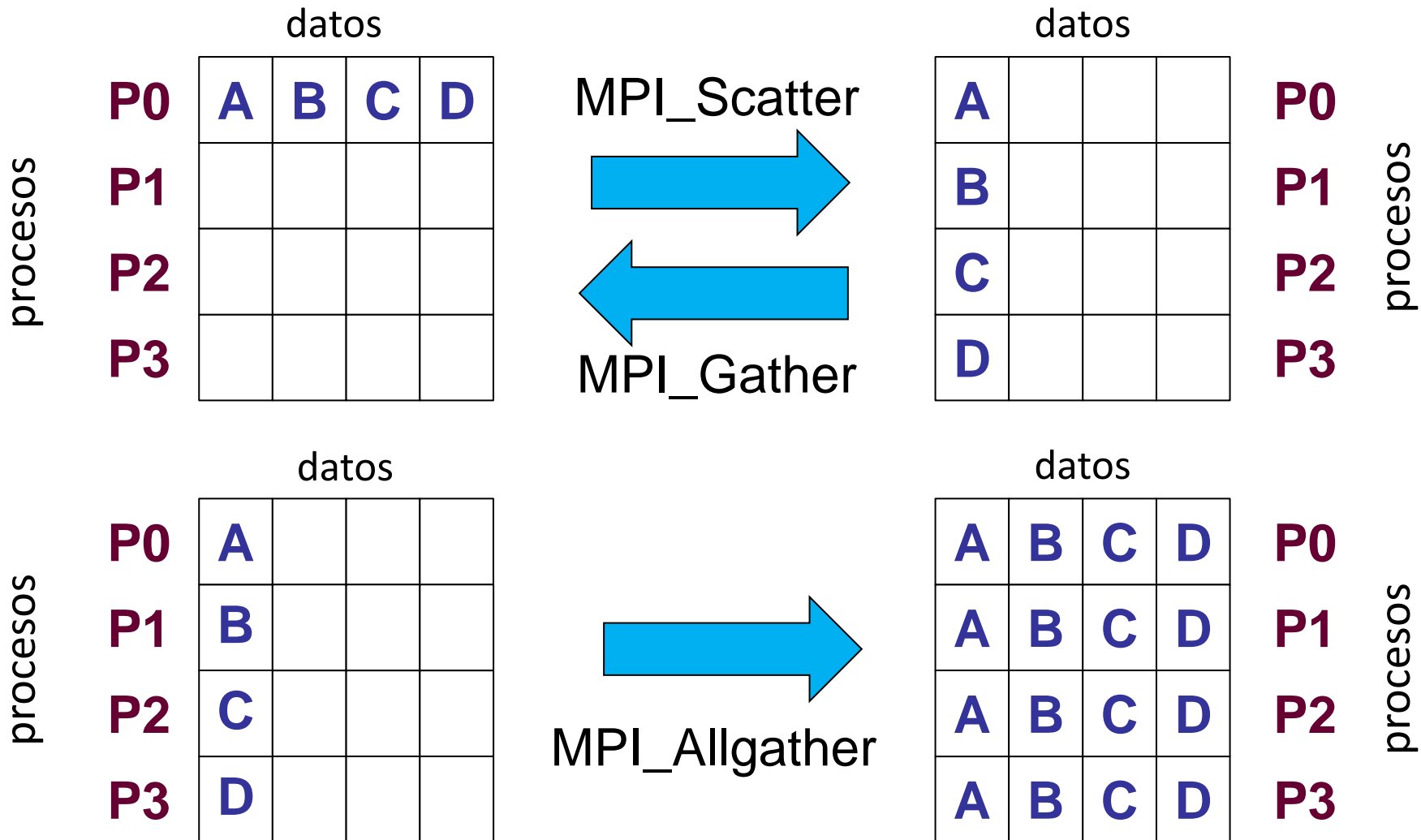
Allgather: recolectar y difundir

- Funciona como gather, pero todos los procesos reciben el resultado





Difusión y recolección de datos





Ejemplo: operaciones colectivas

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(int argc, char *argv[]) {
    int numtasks, rank, sendcount, recvcount, source;
    float sendbuf[SIZE][SIZE] = {{1.0, 2.0, 3.0, 4.0}, {5.0, 6.0,
7.0,8.0}, {9.0, 10.0, 11.0, 12.0}, {13.0, 14.0, 15.0, 16.0}};
    float recvbuf[SIZE];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    if (numtasks == SIZE) {
        source = 1; sendcount = SIZE; recvcount = SIZE;
        MPI_Scatter(sendbuf, sendcount, MPI_FLOAT, recvbuf, recvcount,
            MPI_FLOAT, source, MPI_COMM_WORLD);
        printf("rank= %d  Results: %f %f %f %f\n", rank, recvbuf[0],
            recvbuf[1], recvbuf[2], recvbuf[3]);
    } else printf("Must specify %d processors. Terminating.\n", SIZE);
    MPI_Finalize();
}
```



Ejemplo: operaciones colectivas

- Scatter en las filas de una matriz
- Salida del programa

```
rank= 0  Results: 1.00000  2.00000  3.00000  4.00000
rank= 1  Results: 5.00000  6.00000  7.00000  8.00000
rank= 2  Results: 9.00000 10.00000 11.00000 12.00000
rank= 3  Results: 13.00000 14.00000 15.00000 16.00000
```



Ejemplo: MPI_Scatter y MPI_Gather

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int numnodes,myid,mpi_err; /* variables globales */
#define root 0

int main(int argc,char *argv[]){
    int *myray,*send_ray,*back_ray, int count,size,mysize,i,k,j,total;

    MPI_Init(argc,argv);
    MPI_Comm_size( MPI_COMM_WORLD, &numnodes );
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    count=4; /* cada proceso recibe count elementos del root */

    myray=(int*)malloc(count*sizeof(int));
    if(myid == root){
        /* crear los datos a ser enviados por el root */
        size=count*numnodes;
        send_ray=(int*)malloc(size*sizeof(int));
        back_ray=(int*)malloc(numnodes*sizeof(int));
        for(i=0;i<size;i++)
            send_ray[i]=i;
    }
}
```



Ejemplo: MPI_Scatter y MPI_Gather

```
/* enviar los datos a cada proceso */
mpi_err=MPI_Scatter(send_ray,count,MPI_INT,myray,count,MPI_INT,root,MPI_COMM_WOR
LD);

/* cada proceso realiza una suma local */
total=0;
for(i=0;i<count;i++)
    total=total+myray[i];
printf("myid= %d total= %d\n ",myid,total);

/* enviar las sumas locales al proceso root*/
mpi_err = MPI_Gather(&total,1,MPI_INT,back_ray,1,MPI_INT,root,MPI_COMM_WORLD);

/* el root imprime la suma global */
if(myid == root){
    total=0;
    for(i=0;i<numnodes;i++)
        total=total+back_ray[i];
    printf("results from all processors= %d \n ",total);
}
MPI_Finalize();
}
```

Fuente: http://geco.mines.edu/workshop/class2/examples/mpi/c_ex05.c



Cálculos colectivos

- El usuario puede combinar cálculos parciales de todos los procesos
- Los resultados quedan disponibles en un proceso particular o en todos los procesos
- Permite la ejecución sincronizada
- Combinación de resultados parciales:
 - El proceso principal (root) recibe los cálculos parciales y los combina usando la operación indicada

```
ierr = MPI_Reduce(sbuf,rbuf,cont,datatype,op, root, comm)
```

- Todos los procesos pueden almacenar el resultado con:
 - `MPI_Allreduce`
 - En este caso la rutina no requiere el parámetro root

Operaciones de reducción

Reducción	Operación	Tipos de datos
MPI_MAX	Máximo	integer, float
MPI_MIN	Mínimo	integer, float
MPI_SUM	Suma	integer, float
MPI_PROD	Producto	integer, float
MPI_LAND	And lógico	integer
MPI_BAND	And bit a bit	integer, MPI_BYTE
MPI_LOR	Or lógico	integer
MPI_BOR	Or bit a bit	integer, MPI_BYTE
MPI_LXOR	Xor lógico	integer
MPI_BXOR	Xor bit a bit	integer, MPI_BYTE
MPI_MAXLOC	Máx y loc	Par de integer
MPI_MINLOC	Mín y loc	Par de integer

Definición de operaciones

- El usuario puede definir sus propias operaciones
 - Por ejemplo, en C

```
void function(void *invec,void *inout,int *len,  
MPI_Datatype *datatype)  
    { Cuerpo de la función }  
  
...  
int commute;  
MPI_Op op;  
  
...  
ierr = MPI_Op_create(function,commute,&op);  
  
...  
ierr=MPI_Op_free(&op);    {op = MPI_OP_NULL}
```




Ejemplo de reducción

- El ejemplo previo de scatter/gather y cálculo colectivo puede implementarse con una operación de reducción

- En lugar de

```
/* enviar las sumas locales al proceso root */  
mpi_err = MPI_Gather(&total,1,MPI_INT,back_ray,1,MPI_INT,root,MPI_COMM_WORLD);  
/* el root imprime la suma global */  
if(myid == root){  
    total=0;  
    for(i=0;i<numnodes;i++){total=total+back_ray[i];}  
    printf("results from all processors= %d \n ",total);  
}
```

- Simplemente sería

```
/* enviar las sumas locales al proceso root y reducir */  
mpi_err = MPI_Reduce(&total,&gttotal,1,MPI_INT,MPI_SUM,root,MPI_COMM_WORLD);  
/* el root imprime la suma global */  
if(myid == root){  
    printf("results from all processors= %d \n ",gttotal);  
}
```



Manejo de grupos

- Organización de tareas en grupos
 - Permite comunicaciones y operaciones colectivas sobre un conjunto de tareas relacionadas
 - Provee la base para implementar topologías de procesos
- MPI provee rutinas para la manipulación de grupos:
 - Crear un grupo de procesos a partir de otros grupos de procesos
 - Crear un comunicador para el nuevo grupo
 - Realizar operaciones colectivas entre los procesos de un grupo
 - Organizar los grupos de procesos en topologías virtuales
- Todas las tareas que pertenezcan al grupo deben participar en la invocación de la creación del grupo



Manejo de grupos

- Definen un colección ordenada de procesos
- Cada proceso que pertenece a un grupo tiene un único identificador asociado (rank) en ese grupo
- Los procesos pueden pertenecer a más de un grupo a la vez
- El rank es siempre relativo a un grupo
- Tareas que no pertenezcan a un grupo no podrán participar en operaciones o comunicaciones colectivas que especifiquen ese grupo
- Un grupo se asocia con un comunicador, por defecto todas las tareas están en el grupo asociado con `MPI_COMM_WORLD`
- Existencia del grupo vacío: `MPI_GROUP_EMPTY`
- Los grupos son creados a partir de otro grupo
- El grupo base es el asociado al comunicador inicial `MPI_COMM_WORLD`



Manejo de grupos

- `MPI_Group_difference (group1,group2,*newgroup)`
Crea un grupo diferencia de dos grupos
- `MPI_Group_incl (group,n,*ranks,*newgroup)`
Crea un grupo con las tareas listadas de un grupo existente
- `MPI_Group_excl (group,n,*ranks,*newgroup)`
Crea un grupo con las tareas no listadas de un grupo existente
- `MPI_Group_intersection (group1,group2,*newgroup)`
- `MPI_Group_union (group1,group2,*newgroup)`
- `MPI_Group_compare (group1,group2,*result)`
Compara dos grupos y retorna:
 - `MPI_IDENT` si las tareas y su orden coinciden.
 - `MPI_SIMILAR` si solo las tareas son iguales.
 - `MPI_UNEQUAL` en otro caso.



Manejo de grupos

- `MPI_Group_rank (group, *rank)`
Retorna el rango del proceso en el grupo (o `MPI_UNDEFINED` si el proceso no es miembro del grupo)
- `MPI_Group_size (group, *size)`
Retorna el tamaño (#procesos) del grupo
- `MPI_Group_free (group)`
Elimina un grupo
- `MPI_Comm_group (comm, *group)`
Devuelve el grupo asociado a un comunicador
- `MPI_Comm_create (comm, group, *newcomm)`
Crea un comunicador para un grupo a partir de un comunicador existente
- `MPI_Comm_dup (comm, *newcomm)`
Duplica un comunicador existente (con toda su información)



Ejemplo: manejo de grupos

```
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8
#define MASTER 0
#define MSGSIZE 7

int main(int argc, char *argv[]) {
    int rank, new_rank;
    int ranks1[4] = {0,1,2,3};
    int ranks2[4] = {4,5,6,7};
    char *msg;
    MPI_Group orig_group, new_group;
    MPI_Comm new_comm;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    /* Extraer la referencia del grupo original */
    MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
```



Ejemplo: manejo de grupos

```
/* Dividir las tareas en dos grupos. Crear un nuevo grupo y un nuevo
comunicador. Luego, hallar el nuevo rango en el nuevo comunicador y
prepar el proceso maestro para la comunicación colectiva */
if (rank < NPROCS/2) {
    MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
    MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
    MPI_Group_rank (new_group, &new_rank);
    if (new_rank == MASTER) msg="Group 1";
} else {
    MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
    MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
    MPI_Group_rank (new_group, &new_rank);
    if (new_rank == MASTER) msg="Group 2";
}
MPI_Bcast(&msg,MSGSIZE,MPI_CHAR,MASTER,new_comm);
printf("rank= %d newrank= %d msg= %s\n",rank,new_rank,msg);
MPI_Finalize();
}
```



Ejemplo: manejo de grupos

Salida del ejemplo:

```
rank= 0 newrank= 0 msg= Group 1  
rank= 1 newrank= 1 msg= Group 1  
rank= 2 newrank= 2 msg= Group 1  
rank= 3 newrank= 3 msg= Group 1  
rank= 4 newrank= 0 msg= Group 2  
rank= 5 newrank= 1 msg= Group 2  
rank= 6 newrank= 2 msg= Group 2  
rank= 7 newrank= 3 msg= Group 2
```


Topologías virtuales

- Definen un orden de los procesos MPI en una estructura geométrica.
- Las topologías son virtuales, no tienen ningún vínculo con la estructura física del multicomputador subyacente
- Se construyen sobre los comunicadores y los grupos de procesos
- Son aplicables para aplicaciones con patrones de comunicación específicos, con “vecinos” determinados

Topología cartesiana (grid)

```
MPI_Cart_create(comm, ndims, dims,  
periods, reorder, &cartcomm);
```

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Topologías virtuales

- Definen un orden de los procesos MPI en una estructura geométrica.
- Las topologías son virtuales, no tienen ningún vínculo con la estructura física del multicomputador subyacente
- Se construyen sobre los comunicadores y los grupos de procesos
- Son aplicables para aplicaciones con patrones de comunicación específicos, con “vecinos” determinados

Topología cartesiana (grid)

```
MPI_Cart_create(comm, ndims, dims,  
periods, reorder, &cartcomm);
```

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Topologías virtuales

```
MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],  
                const int periods[], int reorder, MPI_Comm *comm_cart)
```

- `comm_old`: comunicador base
- `ndims`: número de dimensiones de la grilla
- `dims`: dimensiones de la grilla
- `periods`: array de tamaño `ndims`, indica si la grilla es periódica (`true`) o no (`false`) en cada dimensión
- `reorder`: booleano, indica si los rangos pueden reordenarse o no
- `comm_cart`: comunicador de la grilla

```
MPI_Cart_create(com, 2, [4,4], [0,0], 0, &cartcom);
```

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Topologías virtuales

`MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)`

Retorna las coordenadas en una grilla cartesiana, el inverso es `MPI_Cart_rank`

- `comm`: comunicador de la grilla
- `rank`: rango del proceso
- `maxdims`: largo del vector `coords`
- `coords`: arreglo con las coordenadas del proceso en la grilla

`MPI_Cart_shift(MPI_Comm comm, int dir, int disp, int *source, int *dest)`

Obtiene los dos vecinos de un proceso en una dimensión

- `comm`: comunicador de la grilla
- `dir`: dimensión del desplazamiento (0: columna, 1: fila)
- `disp`: desplazamiento (> 0: arriba/derecha, < 0: abajo/izquierda)
- `source`: rango del vecino1
- `dest`: rango del vecino2



Ejemplo: topologías virtuales

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 16
#define UP    0           // vecinos // vecinos
#define DOWN  1
#define LEFT  2
#define RIGHT 3

main(int argc, char *argv[]) {
    int numtasks, rank, source, dest, outbuf, i, tag=1,
    inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,},
    nbrs[4], dims[2]={4,4}, periods[2]={0,0}, reorder=0, coords[2];

    MPI_Request reqs[8];
    MPI_Status stats[8];
    MPI_Comm cartcomm;    // comunicador para la grilla
```



Ejemplo: topologías virtuales

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
if (numtasks == SIZE) {
    // crear topología virtual cartesiana
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);

    // obtener rango, coordenadas, rango de vecinos
    MPI_Comm_rank(cartcomm, &rank);
    MPI_Cart_coords(cartcomm, rank, 2, coords);
    MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
    MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);
    printf("rank: %d coords: %d %d neighbors(u,d,l,r): %d %d %d %d\n",
           rank, coords[0], coords[1], nbrs[UP], nbrs[DOWN], nbrs[LEFT], nbrs[RIGHT]);
    outbuf = rank;
}
```



Ejemplo: topologías virtuales

```
// intercambiar datos (rango) con los cuatro vecinos
for (i=0; i<4; i++) {
    dest = nbrs[i];
    source = nbrs[i];
    MPI_Isend(&outbuf, 1, MPI_INT, dest, tag, MPI_COMM_WORLD, &reqs[i]);
    MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag, MPI_COMM_WORLD, &reqs[i+4]);
}
MPI_Waitall(8, reqs, stats);
printf("rank= %d, inbuf(u,d,l,r)= %d %d %d %d\n",
       rank, inbuf[UP], inbuf[DOWN], inbuf[LEFT], inbuf[RIGHT]);
} else
    printf("Deben especificarse %d procesos.\n", SIZE);
MPI_Finalize();
}
```



Ejemplo: topologías virtuales

```
rank= 0 coords= 0 0 neighbors(u,d,l,r)= -1 4 -1 1
rank= 0 inbuf(u,d,l,r)= -1 4 -1 1
rank= 8 coords= 2 0 neighbors(u,d,l,r)= 4 12 -1 9
rank= 8 inbuf(u,d,l,r)= 4 12 -1 9
rank= 1 coords= 0 1 neighbors(u,d,l,r)= -1 5 0 2
rank= 1 inbuf(u,d,l,r)= -1 5 0 2
rank= 13 coords= 3 1 neighbors(u,d,l,r)= 9 -1 12 14
rank= 13 inbuf(u,d,l,r)= 9 -1 12 14
...
...
rank= 3 coords= 0 3 neighbors(u,d,l,r)= -1 7 2 -1
rank= 3 inbuf(u,d,l,r)= -1 7 2 -1
rank= 11 coords= 2 3 neighbors(u,d,l,r)= 7 15 10 -1
rank= 11 inbuf(u,d,l,r)= 7 15 10 -1
rank= 10 coords= 2 2 neighbors(u,d,l,r)= 6 14 9 11
rank= 10 inbuf(u,d,l,r)= 6 14 9 11
rank= 9 coords= 2 1 neighbors(u,d,l,r)= 5 13 8 10
rank= 9 inbuf(u,d,l,r)= 5 13 8 10
```


Definición de tipos de datos en MPI

- MPI permite al usuario construir nuevos tipos de datos en tiempo de ejecución
- Primitivas:
 - `MPI_Type_contiguous`
 - `MPI_Type_vector`
 - `MPI_Type_indexed`
 - `MPI_Type_struct`
- Tipos de datos predefinidos (para C):

<code>MPI_CHAR, MPI_WCHAR</code>	<code>MPI_UNSIGNED_LONG</code>
<code>MPI_SHORT</code>	<code>MPI_FLOAT</code>
<code>MPI_INT</code>	<code>MPI_DOUBLE</code>
<code>MPI_LONG, MPI_LONG_LONG</code>	<code>MPI_LONG_DOUBLE</code>
<code>MPI_UNSIGNED_CHAR</code>	<code>MPI_BYTE</code>
<code>MPI_UNSIGNED_SHORT</code>	<code>MPI_PACKED</code>
<code>MPI_UNSIGNED_INT</code>	

Definición de tipos de datos en MPI

- `MPI_Type_contiguous (count,oldtype,*newtype)`

Produce un nuevo tipo de dato, formado por copias contiguas de un tipo existente

- `MPI_Type_vector (count,blocklength,stride,oldtype,*newtype)`

Similar al anterior, permite “saltos regulares” en los desplazamientos

- `MPI_Type_indexed (count,blocklens[],offsets[],old_type,*newtype)`

Crea un array “indexado” por los desplazamientos provistos

- `MPI_Type_hindexed`

Idéntica a `MPI_Type_indexed`, pero los offsets se especifican en bytes

- `MPI_Type_commit`

Finaliza la definición de tipo de datos, reserva memoria, etc.

- `MPI_Type_free`

Elimina el tipo de dato derivado, libera la memoria.

Definición de tipos de datos en MPI

`MPI_Type_contiguous(count, oldtype, *newtype)`

- Tipo de dato fila: `count = 4`, `oldtype = MPI_FLOAT`, `newtype = rowtype`
- `MPI_Type_contiguous(4, MPI_FLOAT, & rowtype)`

`a[4][4]`

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

`MPI_send(&a[2][0], 1,
rowtype, dest, tag, comm)`

9.0	10.0	11.0	12.0
-----	------	------	------

se envía un elemento
de tipo `rowtype`

Definición de tipos de datos en MPI

`MPI_Type_vector(count, blocklength, stride, oldtype, *newtype)`

- Tipo de dato columna: `count = 4`, `blocklength=1`, `stride=4`
- `MPI_Type_vector(4, 1, 4, MPI_FLOAT, & coltype)`

`a[4][4]`

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

`MPI_send(&a[0][1], 1,
coltype, dest, tag, comm)`

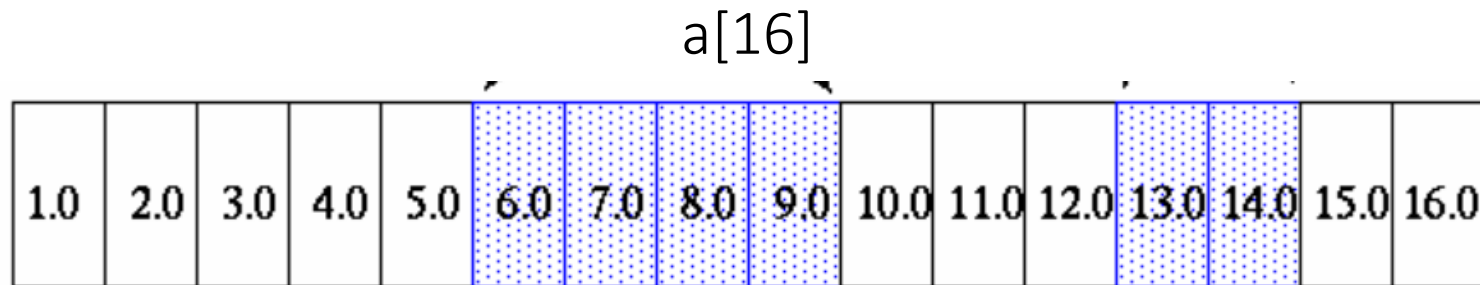
2.0	6.0	10.0	14.0
-----	-----	------	------

se envía un elemento
de tipo `coltype`

Definición de tipos de datos en MPI

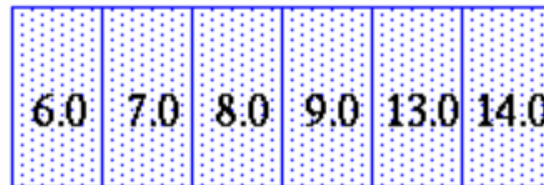
`MPI_Type_indexed(count, blocklens[], offsets[], oldtype, *newtype)`

- Tipo de dato de largo variable: `count = 2`, `blocklens=[4,2]`, `offsets=[5,12]`
- `MPI_Type_indexed(2, blocklens, offsets, MPI_FLOAT, &indextype)`



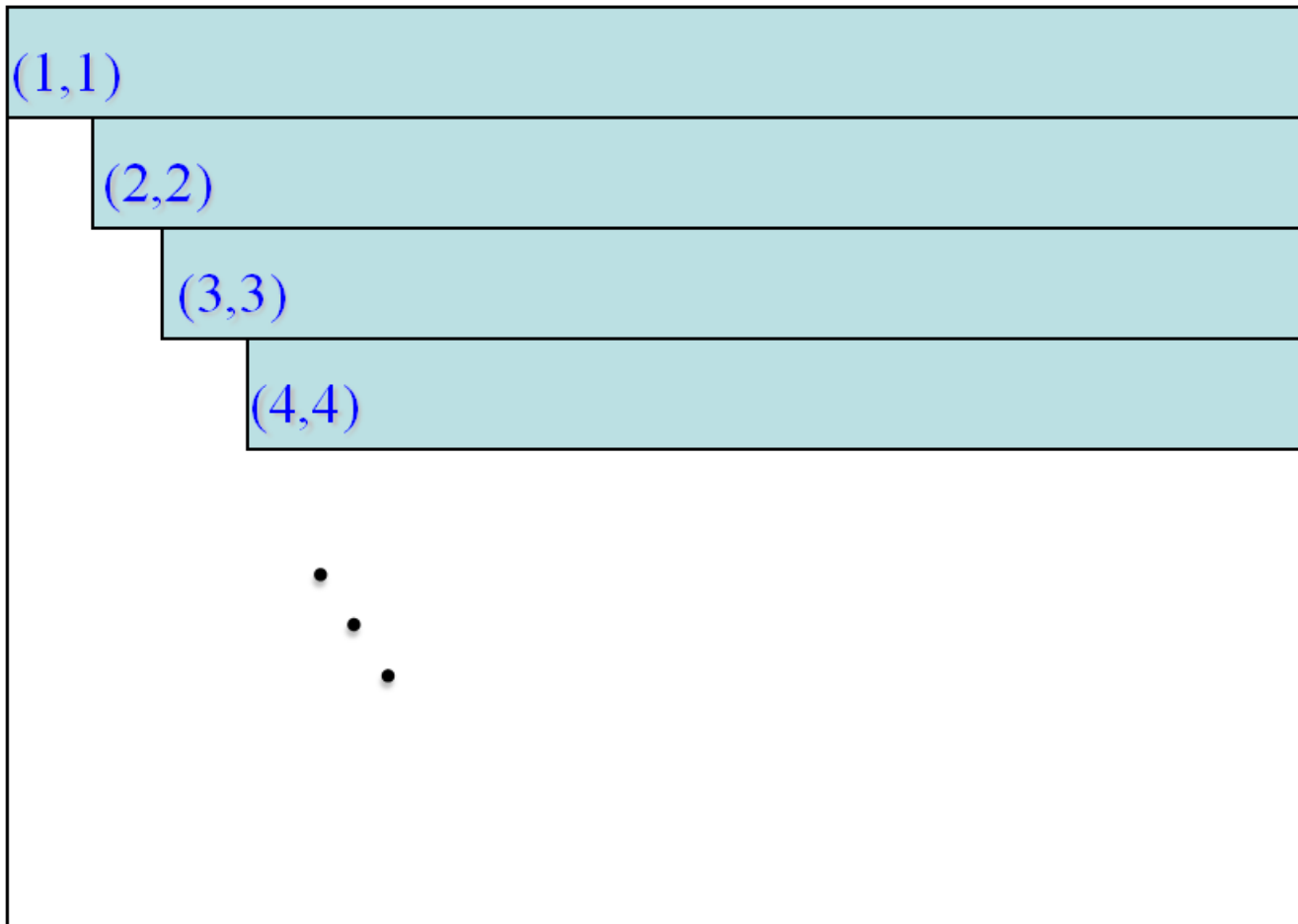
`MPI_send(&a, 1, indextype, dest, tag, comm)`

se envía un elemento
de tipo `indextype`



Ejemplo: definición de tipos de datos en MPI

- Matriz A (100x100):



Ejemplo: definición de tipos de datos en MPI

```
double a[100][100];
int separacion[100],longbloqs[100],i;
MPI_Datatype superior;
....
/* Cálculo del inicio y tamaño de cada fila */
for (i=0;i<100;i++) {
    separacion[i] = 100*i + i;
    longbloqs[i] = 100 - i ;
}
/* Crear el tipo de dato para la parte triangular superior de la matriz*/
MPI_Type_indexed(100,longbloqs,separacion,MPI_DOUBLE,&superior);
MPI_Type_commit(&superior);

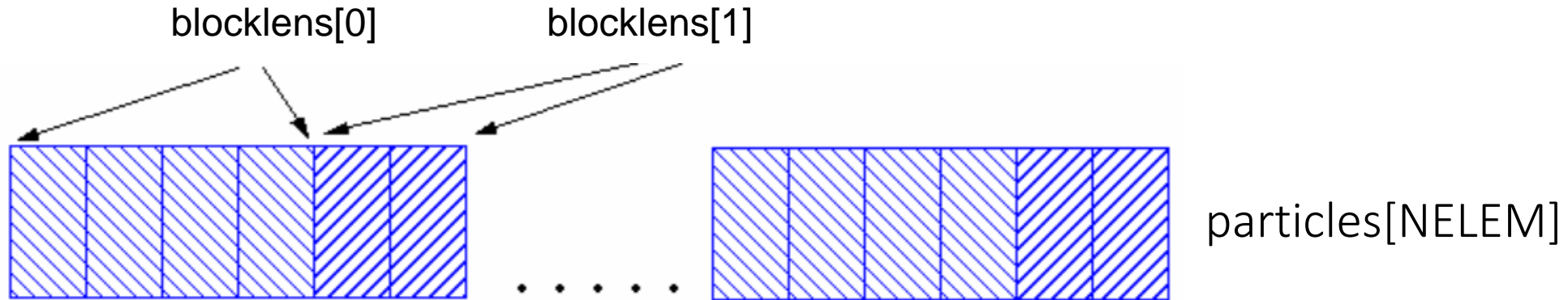
/* ... Enviar la parte superior ... */
MPI_Send(a,1,superior,dest,tag,MPI_COMM_WORLD);
```

Definición de tipos de datos en MPI

`MPI_Type_struct(count, blocklens[], offsets[], oldtypes, *newtype)`

```
typedef struct {float x,y,z,velocity; int type, cat;} particle
particle particles[NELEM]
```

- Tipo de dato estructurado: `count = 2`, `blocklens=[4,2]`, `offsets=[0,4*extent]`
- `MPI_Type_struct(2, blocklens, offsets, {MPI_FLOAT,MPI_INT}, &structtype)`



`MPI_send(particles, NELEM, particletype, dest, tag, comm)` envía un elemento de tipo `particletype` (array de `NELEM` elementos, cada uno formado por cuatro floats y dos enteros)

Definición de tipos de datos en MPI

```
MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

```
MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,  
MPI_Aint *extent) (en MPI-3)
```

- Retorna el extent (tamaño) de un tipo de datos
- En el ejemplo previo: `MPI_Type_extent(MPI_FLOAT, &extent)`
- Proporciona una función que encapsula el uso de `sizeof()`

Ejemplo: definición de tipos de datos en MPI

- Considérense los tipos de datos

```
struct A {                                struct B {  
    int f;                                struct A a;  
    short p; };                          int pp, vp; };
```

- Construcción del tipo de datos derivado en MPI:

```
static const int blocklen[] = {1, 1, 1, 1};  
static const MPI_Aint disp[] = { offsetof(struct B, a) +  
offsetof(struct A, f), offsetof(struct B, a) + offsetof(struct A, p),  
offsetof(struct B, pp), offsetof(struct B, vp) };
```

```
static MPI_Datatype type[] = {MPI_INT, MPI_SHORT, MPI_INT, MPI_INT};
```

```
MPI_Datatype newtype;  
MPI_Type_create_struct(sizeof(type) / sizeof(*type), blocklen, disp,  
type, &newtype);  
MPI_Type_commit(&newtype);
```

Definición de tipos de datos en MPI

- Otras funciones para crear tipos de datos derivados:
- `MPI_Type_hvector`: Idéntica a `MPI_Type_vector`, pero el stride se especifica en bytes
- `MPI_Type_hindexed`: Idéntica a `MPI_Type_indexed`, pero los offsets se especifican en bytes
- `MPI_Type_create_subarray`: crea un array (n dimensional) que es parte de otro array (n dimensional)
- `MPI_Type_create_darray`: crea un array multidimensional distribuido

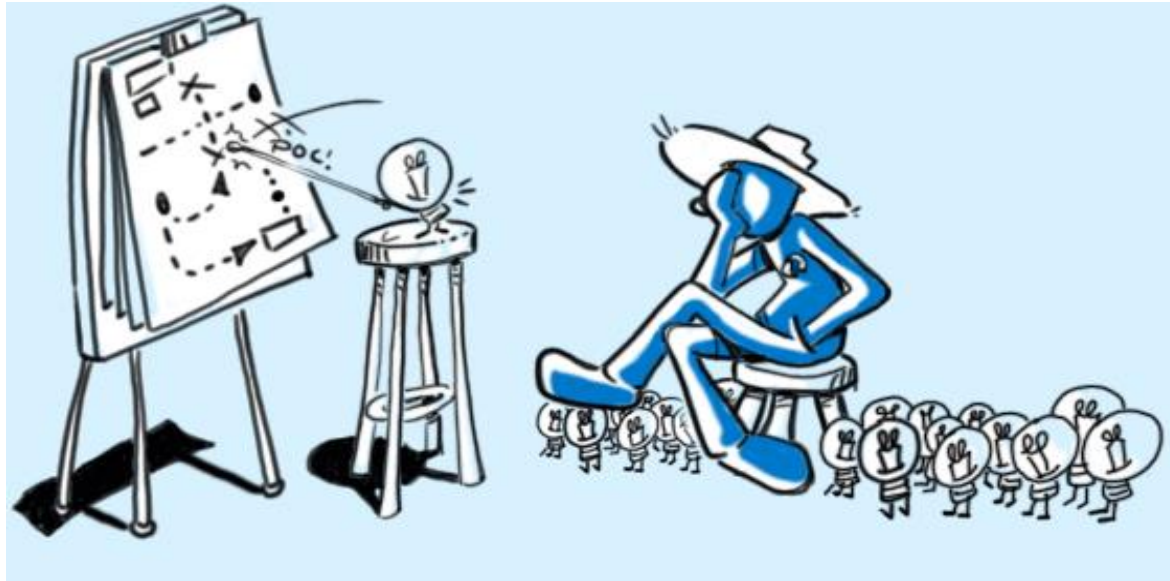
Ejemplo: tipos de datos derivados

- Intercambio de una estructura de partículas (dos floats, un int)

```
int nblocks = 2, blocklen[] = {2, 1}; oldtypes[] = {MPI_FLOAT, MPI_INT};
MPI_Aint displ[] = {0, 8};           // Seteo manual, no muy recomendado
Particle p;

...                                 // Inicializar MPI
MPI_Get_address(&(p.x), &displ[0]); // Obtener dirección y desplazamiento
MPI_Get_address(&(p.type), &displ[1]);
displ[1] -= displ[0];
displ[0] -= displ[0];
MPI_Type_create_struct(nblocks, blocklen, displ, oldtypes, &MPI_Particle);
MPI_Type_commit(&MPI_Particle);

p.x = ... // Inicializar partícula
int dst = 0, src = 1;
if (rank == src)
    MPI_Send(&p, 1, MPI_Particle, dst, 10, MPI_COMM_WORLD);
else
    MPI_Recv(&p, 1, MPI_Particle, src, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



CASO DE ESTUDIO

Método de Jacobi paralelo



Caso de estudio

- Método de Jacobi para resolver sistemas de ecuaciones lineales: $Ax = b$.
- Bajo ciertas condiciones, un sistema lineal puede resolverse mediante el método iterativo de Jacobi:

$$a_{ii}x_i^{(k+1)} = -\sum_{\substack{j=1 \\ j \neq i}}^n a_{ij}x_j^{(k)} + b_i \quad x_i^{(k+1)} = \frac{-\sum_{\substack{j=1 \\ j \neq i}}^n a_{ij}x_j^{(k)} + b_i}{a_{ii}}$$

- Para calcular el paso $k+1$ solo se precisan datos del paso anterior k .
- El cálculo puede paralelizarse realizando una división de dominio, con un modelo maestro-esclavo.



Caso de estudio

- Asumiendo el sistema de 3×3:

$5x_1 - 2x_2 + 3x_3 = -1$	→ P0
$-3x_1 + 9x_2 + x_3 = 2$	→ P1
$2x_1 - x_2 - 7x_3 = 3$	→ P2

- Se aplica una división del dominio en 3 procesos
- Cada proceso toma la fila i , y calcula el x_i correspondiente
- Cada proceso necesita la fila i de la matriz A , el b_i y la solución actual completa
- El algoritmo general tendrá el siguiente esquema:
 - Distribuir datos iniciales
 - Iterar hasta cierta condición esperada
 - Calcular x_i
 - Distribuir x_i a todos los procesos



Caso de estudio

- Fase de distribución inicial:
 - Se necesita transmitir una fila a cada proceso: se puede utilizar un scatter para la matriz A
 - La misma situación ocurre para el vector b
 - En cambio, es necesario transmitir la solución inicial a todos los procesos (broadcast)
- Fase de iteración:
 - Cada proceso calcula su parte del vector x
 - Para distribuir los valores calculados se puede utilizar un broadcast, o mejor un Allgather
- Ejemplo
 - Disponible en <http://www.tezu.ernet.in/dcompsc/facility/HPCC/hypack/mpi-1x-hypack-2013/matrix-comp-solver-codes/jacobi-mpi-code-clang.c>



Caso de estudio: Jacobi con MPI

```
#include <stdio.h>
#include <assert.h>
#include <mpi.h>
#define MAX_ITERATIONS 100
double Distance(double *X_Old, double *X_New, int n_size);

main(int argc, char** argv) {
    /* ..... Inicialización de variables .....*/
    MPI_Status status;
    int n_size, NoofRows_Bloc, NoofRows, NoofCols;
    int Numprocs, MyRank, Root=0;
    int irow, jrow, icol, index, Iteration, GlobalRowNo;
    double **Matrix_A, *Input_A, *Input_B, *ARecv, *BRecv;
    double *X_New, *X_Old, *Bloc_X, tmp; FILE *fp;
    /* ..... Inicialización de MPI .....*/
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
    MPI_Comm_size(MPI_COMM_WORLD, &Numprocs);
```



Caso de estudio: Jacobi con MPI

```
/* .....Leer archivo de entrada .....*/
if(MyRank == Root) {
    if ((fp = fopen (".matrix-data-jacobi.inp", "r")) == NULL) {
        printf("Can't open input matrix file"); exit(-1);
    }
    fscanf(fp, "%d %d", &NoofRows,&NoofCols);
    n_size=NoofRows;
/* ... Reservar memoria y leer datos ....*/
    Matrix_A = (double **) malloc(n_size*sizeof(double *));
    for(irow = 0; irow < n_size; irow++){
        Matrix_A[irow] = (double *) malloc(n_size * sizeof(double));
        for(icol = 0; icol < n_size; icol++)
            fscanf(fp, "%lf", &Matrix_A[irow][icol]);
    }
    fclose(fp);
    if ((fp = fopen (".vector-data-jacobi.inp", "r")) == NULL){
        printf("Can't open input vector file"); exit(-1);
    }
    fscanf(fp, "%d", &NoofRows);
```



Caso de estudio: Jacobi con MPI

```
n_size=NoofRows;
Input_B = (double *)malloc(n_size*sizeof(double));
for (irow = 0; irow<n_size; irow++)
    fscanf(fp, "%lf",&Input_B[irow]);
fclose(fp);
/* ...Convertir Matrix_A en un array 1D Input_A .....*/
Input_A = (double *)malloc(n_size*n_size*sizeof(double));
index = 0;
for(irow=0; irow<n_size; irow++)
    for(icol=0; icol<n_size; icol++)
        Input_A[index++] = Matrix_A[irow][icol];
}
MPI_Bcast(&NoofRows, 1, MPI_INT, Root, MPI_COMM_WORLD);
MPI_Bcast(&NoofCols, 1, MPI_INT, Root, MPI_COMM_WORLD);
if(NoofRows != NoofCols) {
    MPI_Finalize();
    if(MyRank == 0) printf("Input Matrix Should Be Square.\n");
    exit(-1);
}
```



Caso de estudio: Jacobi con MPI

```
/* .... Broadcast del tamaño de la matriz ...*/
MPI_Bcast(&n_size, 1, MPI_INT, Root, MPI_COMM_WORLD);
if(n_size % Numprocs != 0) {
    MPI_Finalize();
    if(MyRank == 0) printf("Matrix Can not be Striped Evenly ..... \n");
    exit(-1);
}
NoofRows_Bloc = n_size/Numprocs;
/*..... Memoria para matriz de entrada y vector en cada proceso ....*/
ARecv = (double *) malloc (NoofRows_Bloc * n_size* sizeof(double));
BRecv = (double *) malloc (NoofRows_Bloc * sizeof(double));

/*..... Scatter de los datos de entrada a todos los procesos ....*/
MPI_Scatter (Input_A, NoofRows_Bloc * n_size, MPI_DOUBLE, ARecv, NoofRows_Bloc *
n_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter (Input_B, NoofRows_Bloc, MPI_DOUBLE, BRecv, NoofRows_Bloc,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
```



Caso de estudio: Jacobi con MPI

```
X_New = (double *) malloc (n_size * sizeof(double));
X_Old = (double *) malloc (n_size * sizeof(double));
Bloc_X = (double *) malloc (NoofRows_Bloc * sizeof(double));
/* Inicializar X[i] = B[i] */
for(irow=0; irow<NoofRows_Bloc; irow++)
    Bloc_X[irow] = BRecv[irow];
MPI_Allgather(Bloc_X, NoofRows_Bloc, MPI_DOUBLE, X_New, NoofRows_Bloc,
              MPI_DOUBLE, MPI_COMM_WORLD);

Iteration = 0;
do {
    for(irow=0; irow<n_size; irow++)
        X_Old[irow] = X_New[irow];
    for(irow=0; irow<NoofRows_Bloc; irow++){
        GlobalRowNo = (MyRank * NoofRows_Bloc) + irow;
        Bloc_X[irow] = BRecv[irow];
        index = irow * n_size;
        for(icol=0; icol<GlobalRowNo; icol++){
            Bloc_X[irow] -= X_Old[icol] * ARecv[index + icol];
        }
    }
}
```



Caso de estudio: Jacobi con MPI

```
    for(icol=GlobalRowNo+1; icol<n_size; icol++){
        Bloc_X[irow] -= X_Old[icol] * ARecv[index + icol];
    }
    Bloc_X[irow] = Bloc_X[irow] / ARecv[irow*n_size + GlobalRowNo];
}
MPI_Allgather(Bloc_X, NoofRows_Bloc, MPI_DOUBLE, X_New, NoofRows_Bloc,
              MPI_DOUBLE, MPI_COMM_WORLD);

Iteration++;
} while((Iteration < MAX_ITERATIONS)&&(Distance(X_Old,X_New,n_size)>=1.0E-24));

/* ..... Imprimir vector resultado .....*/
if (MyRank == 0) {
    printf("Results of Jacobi Method on processor %d: \n", MyRank);
    printf("Matrix Input_A \n");
    for (irow = 0; irow < n_size; irow++) {
        for (icol = 0; icol < n_size; icol++)
            printf("%.3lf  ", Matrix_A[irow][icol]);
        printf("\n");
    }
}
```



Caso de estudio: Jacobi con MPI

```
printf("Matrix Input_B \n");
for (irow = 0; irow < n_size; irow++) {
    printf("%.3lf\n", Input_B[irow]);
}
printf("Solution vector \n");
printf("Number of iterations = %d\n",Iteration);
for(irow = 0; irow < n_size; irow++)
    printf("%.12lf\n",X_New[irow]);
}
MPI_Finalize();
}

double Distance(double *X_Old, double *X_New, int n_size){
    int index;
    double Sum = 0.0;
    for(index=0; index<n_size; index++)
        Sum += (X_New[index]-X_Old[index])*(X_New[index]-X_Old[index]);
    return(Sum);
}
```

Caso de estudio

- Otros códigos de ejemplo para la resolución de sistemas de ecuaciones lineales con MPI
 - Disponibles en https://www.cdac.in/index.aspx?id=print_page&print=ev_hpc_hypack_matrix-comp-solver-mpi
 - Eliminación gaussiana
 - Gradiente conjugado
 - Método de Gauss Seidel



UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY



MPI-2 y MPI-3

Motivación

- MPI-1 fue liberado en 1994 como resultado de un trabajo en conjunto entre varios vendedores
- Se desarrollaron una gran cantidad de implementaciones propietarias y libres
- A su vez, se estaban desarrollando otros modelos de programación paralela que proveían otras funcionalidades
- Las funcionalidades más notables que no proveía MPI-1:
 - Creación dinámica de procesos
 - Entrada/salida paralela
 - Único modelo de comunicación: two-sided
- De forma de cubrir esas necesidades se elaboró un nuevo foro de discusión y una posterior nueva versión del estándar

Evolución del estándar

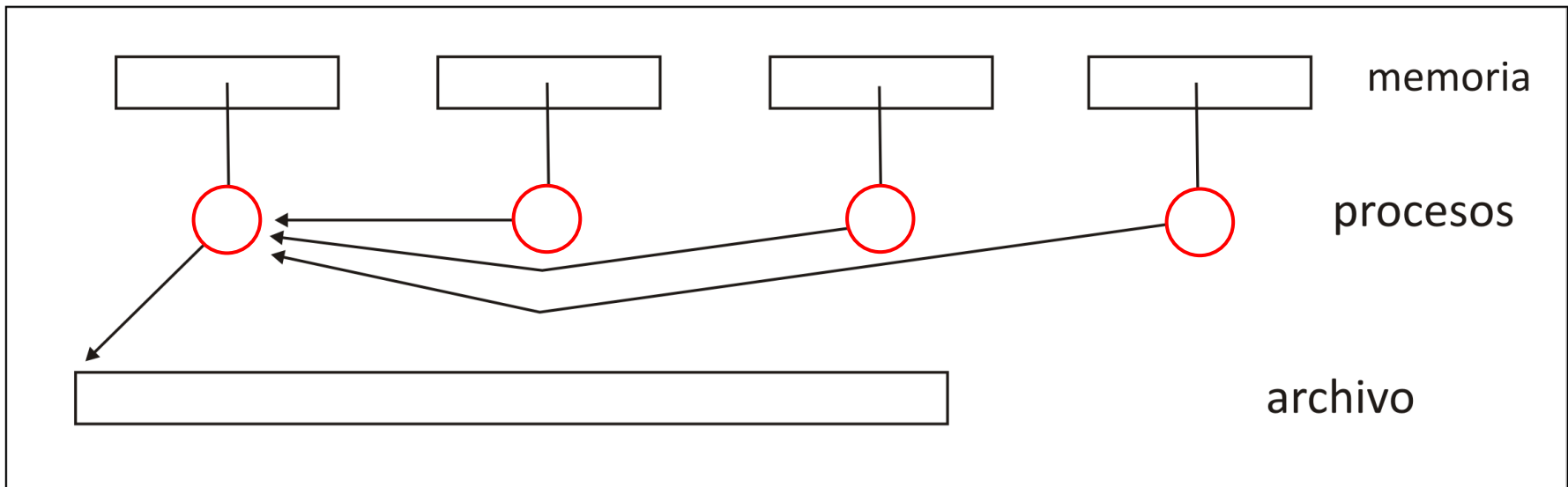
- MPI-1.0: Mayo, 1994
- MPI-1.1: Junio, 1995
- MPI-1.2: Julio, 1997
 - MPICH 1.2.7p1 (Noviembre, 2005)
- MPI-2.0: Julio, 1997
- MPI-1.3: Mayo, 2008
- MPI-2.1: Junio, 2008
- MPI-2.2: Setiembre, 2009
 - MPICH2 2.4.1 (Setiembre, 2011)
- MPI-3.0: Setiembre, 2012

Entrada/salida paralela

- MPI-1 no proporcionó ningún soporte para operaciones de entrada/salida
- Los programadores se vieron obligados a diseñar sus propios mecanismos de E/S
- Las dos configuraciones más utilizadas son:
 - E/S secuencial: Un único proceso MPI centraliza la lectura/escritura en un archivo único
 - E/S Paralela con múltiples archivos: Cada proceso lee/escribe de un archivo distinto

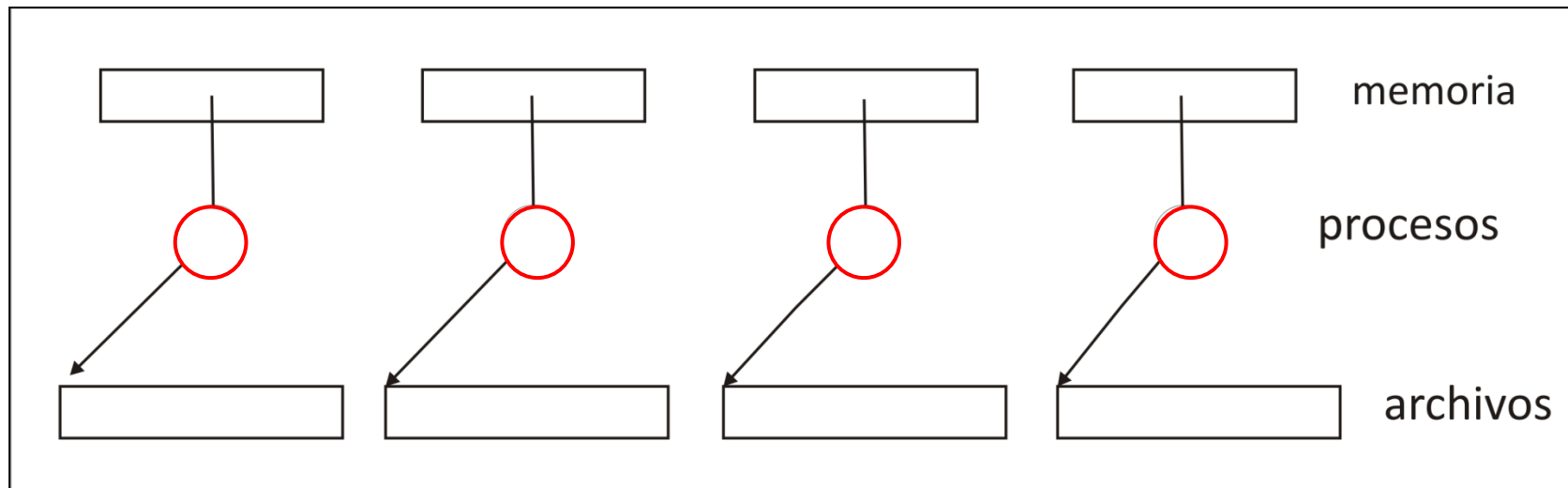
Entrada/salida secuencial

- Un proceso es dedicado a concentrar todas las lecturas/escrituras
- Los demás procesos envían/reciben los mensajes a través de intercambio de mensajes
- Bajo desempeño



Entrada/salida paralela a múltiples archivos

- Cada proceso MPI lee/escribe de un archivo distinto
- No es necesario la sincronización por parte de los procesos
- El archivo debe ser unido posteriormente por fuera de MPI
- Mayor esfuerzo de gestión y overhead

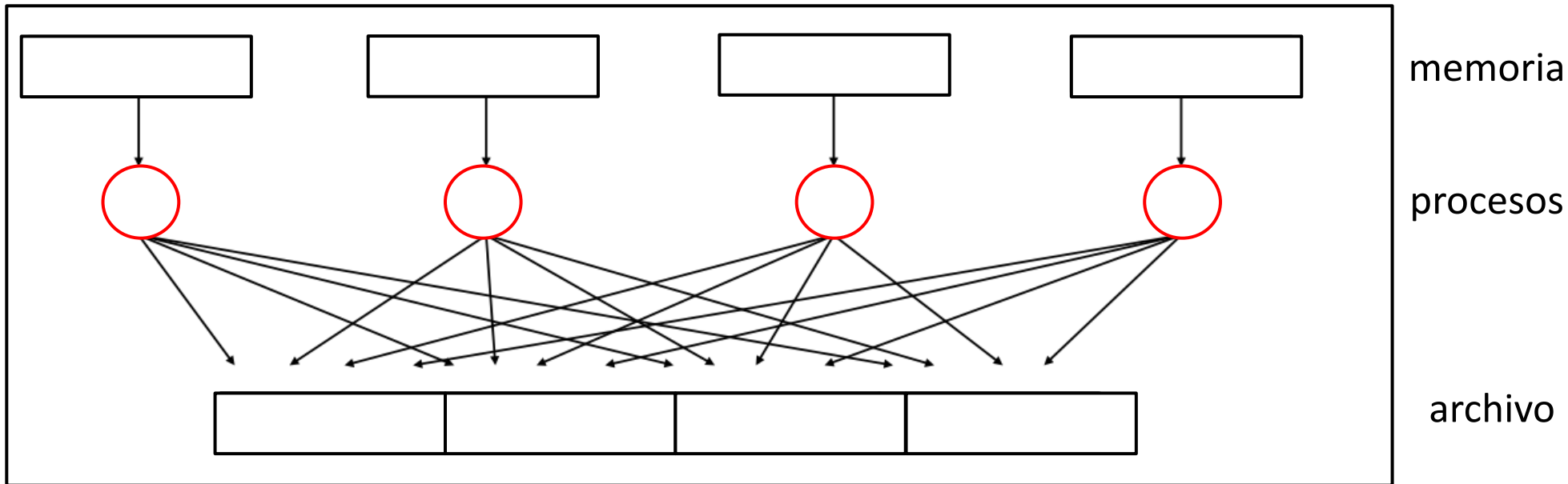


Entrada/salida en MPI-2

- El estandar MPI-2 introdujo un conjunto de primitivas específicas para el manejo de E/S
- Ventajas sobre E/S estándar (POSIX):
 - Performance/desempeño
 - Permite implementar E/S con archivo único (no se requiere un archivo por proceso)
- Los procesos MPI pueden acceder a una porción de un archivo común
- Se proponen primitivas similares a las ya establecidas para manipulación de archivos: open, close, read, write, seek, etc.
- Se proveen operaciones para acceso contiguo y no contiguo
- Se definen vistas del archivo (file view) para cada proceso de forma de lograr el acceso no contiguo

Entrada/salida paralela en MPI

- Cada proceso MPI lee/escribe de un único archivo
- Cada proceso tiene porciones/vistas del archivo
- Escribir es análogo a enviar (send) y leer es análogo a recibir (receive)
- Mayor desempeño, la gestión de vistas la realiza MPI



Entrada/salida en MPI-2

- Escribir es análogo a enviar (send) y leer es análogo a recibir (receive)
- El sistema de E/S paralela requiere:
 - Operaciones colectivas
 - Tipos de datos definidos por el usuario para especificar los datos a almacenar en memoria y en el archivo
 - Comunicaciones para separar el pasaje de mensajes de los procesos (nivel de aplicación) de los mensajes utilizados para E/S (nivel del sistema)
 - Operaciones no bloqueantes
- Al nivel de aplicación:
 - Lecturas y escrituras concurrentes de múltiples procesos a un único archivo
- Al nivel del sistema:
 - Un sistema de archivos paralelo y hardware que soporte el acceso concurrente

Abrir y cerrar archivos

- MPI-2 define el tipo **MPI_File** que proporciona punteros (*handlers*) a los archivos
- Las operaciones de apertura (`MPI_File_open`) y cierre (`MPI_File_close`) de archivo son operaciones colectivas

`MPI_File_open(comm, filename, mode, info, fpointer)`

- **comm**: Comunicador asociado a la operación colectiva
- **mode**: modo de apertura: `MPI_MODE_RDONLY`, `MPI_MODE_WRONLY`, `MPI_MODE_CREATE`, etc.
- **info**: sugerencias para optimizar la implementación
- **fpointer**: *handler* para acceso

`MPI_File_close(fpointer)`

Lectura/escritura

- Los archivos abiertos por un proceso tienen asociado un *file pointer* para acceso secuencial sobre una vista (*file view*) del archivo
- La vista por defecto es todo el archivo, pero se pueden definir porciones específicas según un formato dado a través de tipo de datos estándar o estructurados MPI
- Las operaciones no son colectivas
- Las operaciones son relativas al *file pointer* del archivo referenciado
- Las funciones `MPI_File_read` y `MPI_File_write` no son *thread-safe*

- La operación `MPI_File_read` utiliza el file pointer para acceder al archivo
`MPI_File_read(MPI_File fp, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
 - `fp`: handler para acceso
 - `count`: número de elementos a escribir
 - `datatype`: tipo del dato a leer
 - `buf`: dirección del buffer de salida
 - `status`: información adicional

```
bufsize = FILESIZE/nprocs;  
nints = bufsize/sizeof(int);  
MPI_File_open(MPI_COMM_WORLD, "archivo.txt",  
              MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);  
MPI_File_read(fh, buf, nints, MPI_INT, &status);
```

Lectura con posicionamiento previo

- Para ubicarse en una posición dentro del archivo se utiliza la función `MPI_File_seek`, que actualiza el file pointer del proceso

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int update)
```

- Update tiene tres valores
 - `MPI_SEEK_SET`: el puntero se setea al offset
 - `MPI_SEEK_CUR`: el puntero se setea a la posición actual más el offset
 - `MPI_SEEK_END`: el puntero se setea al final del archivo más el offset

```
bufsize = FILESIZE/nprocs;  
nints = bufsize/sizeof(int);  
  
MPI_File_open(MPI_COMM_WORLD, "archivo.txt",  
              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);
```

Escritura

- MPI_File_write también utiliza el file pointer para el acceso

```
MPI_File_write(MPI_File fp, void *buf, int count,  
MPI_Datatype datatype, MPI_Status *status)
```

- fp: handler para acceso
- count: número de elementos a escribir
- datatype: tipo del dato a leer
- buf: dirección del buffer de la aplicación
- status: información adicional

```
MPI_File_open(MPI_COMM_WORLD, "archivo.txt", MPI_MODE_CREATE  
| MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);  
MPI_File_write(fh, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);
```

- Pueden utilizarse tipos estructurados MPI
- Puede utilizarse MPI_File_seek para posicionamiento.
- Puede definirse una vista para escribir en una porción del archivo

Escritura: ejemplo

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[]){
MPI_File fh;
int buf[1000], rank;

MPI_Init(argc, argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_File_open(MPI_COMM_WORLD, "test.out", MPI_MODE_CREATE |
              MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
if (rank == 0)
    MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
MPI_File_close(&fh);
MPI_Finalize();
return 0;
}
```

Lectura/escritura en MPI

- **MPI_File_open** es colectiva sobre el comunicador
 - Debe ser colectiva para proporcionar soporte a E/S colectiva, para mejorar el desempeño
 - Los modos de apertura son similares a los de Unix/Linux:
MPI_MODE_WRONLY, MPI_MODE_RDWR, MPI_MODE_CREATE
 - Info prové información adicional para mejorar el desempeño
- **MPI_File_write** es independiente paa cada proceso (por este motive se chequea el rango del proceso que escribe)
- **MPI_File_close** es colectiva, su análogo es MPI_Comm_free
 - Libera los recursos utilizados para la gestión del archivo compartido

Lectura/escritura con posición

- Además de las operaciones básicas se proveen funciones para leer/escribir con posición explícita:
 - `MPI_File_read_at`
 - `MPI_File_write_at`
- Estas funciones son thread-safe y no modifican el file pointer
- Se utiliza un direccionamiento (posicionamiento explícito) que no modifica el estado interno (fp)
- Es útil cuando la sincronización de las operaciones colectivas no es natural para el esquema de paralelismo, o cuando el overhead al utilizar operaciones colectivas conspira contra el desempeño
 - Por ejemplo: E/S de pocos datos durante una lectura de inicialización

Lectura con posición: ejemplo

```
#include "mpi.h"
MPI_Status status;
MPI_File fh;
MPI_Offset offset;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", MPI_MODE_RDONLY,
              MPI_INFO_NULL, &fh)
nints = FILESIZE / (nprocs*INTSIZE);
offset = rank * nints * INTSIZE;

MPI_File_read_at(fh, offset, buf, nints, MPI_INT, &status);
MPI_Get_count(&status, MPI_INT, &count);
printf("process %d read %d ints\n", rank, count);
MPI_File_close(&fh);
```

- Acceso no contiguo
 - Cuando un proceso abre un archivo tiene una visión total de su contenido
 - MPI proporciona **vistas** para describir la parte de un archivo que es responsabilidad de cada proceso
 - Las vistas se definen por un offset y un tipo de datos (MPI_Datatype)
 - Proveen una manera eficiente de implementar el acceso no contiguo
 - Luego de invocada/definida la vista, solo la parte definida (porción visible) es accesible al proceso MPI para E/S

Vistas

- MPI_File_set_view define una vista de un archivo

```
int MPI_File_set_view(MPI_File fp, MPI_Offset disp, MPI_Datatype  
etype, MPI_Datatype filetype, char *datarep, MPI_Info info)
```

- **fp**: handler para acceso
- **disp**: offset de inicio de la vista (relativo al inicio del archivo)
- **etype**: (elementary datatype) tipo de dato básico (o derivado) de la vista
- **filetype**: tipo de dato, etype o derivado (a partir de etype)
- **datarep**: representación de datos “native”, “internal” o “external32”
- **info**: información adicional para mejorar el desempeño (MPI_INFO_NULL: sin información)

```
MPI_File_open(MPI_COMM_WORLD, "archivo.txt", MPI_MODE_CREATE  
             | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);  
MPI_File_set_view(fh, myrank * BUFSIZE * sizeof(int), MPI_INT,  
                 MPI_INT, "native", MPI_INFO_NULL);  
MPI_File_write(fh, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);
```

Vistas

- Se espera que se invoque `MPI_File_set_view` inmediatamente después de `MPI_File_open`
- El acceso a los datos se realiza en etype unidades, para leer o escribir items (datos) de tipo etype. Los offsets se expresan en etypes y los file pointers apuntan al inicio de etypes.
- Inicialmente, todos los procesos ven al archivo como un arreglo lineal de bytes (byte stream), etype y filetype son ambos `MPI_BYTE`.
- Si el archivo se abrió en el modo `MPI_MODE_SEQUENTIAL`, el desplazamiento especial `MPI_DISPLACEMENT_CURRENT` debe especificarse en `disp`, para setear el desplazamiento a la posición actual del file pointer compartido. `MPI_DISPLACEMENT_CURRENT` es inválido en otro caso.

Vistas

 etype: MPI_DOUBLE

 filetype: patrón (uno de cada cuatro doubles)

Inicio del archivo
desplazamiento

Vista: cada tarea repite el patrón, con diferentes desplazamientos



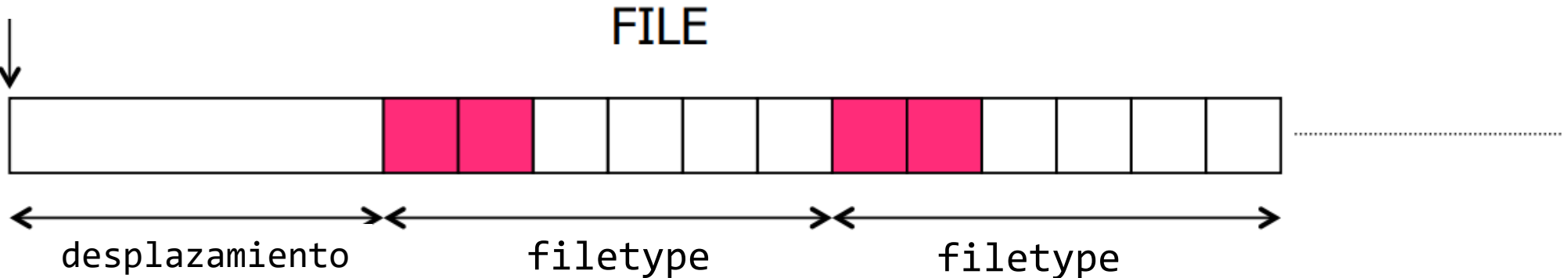
Vistas

 etype: MPI_INT



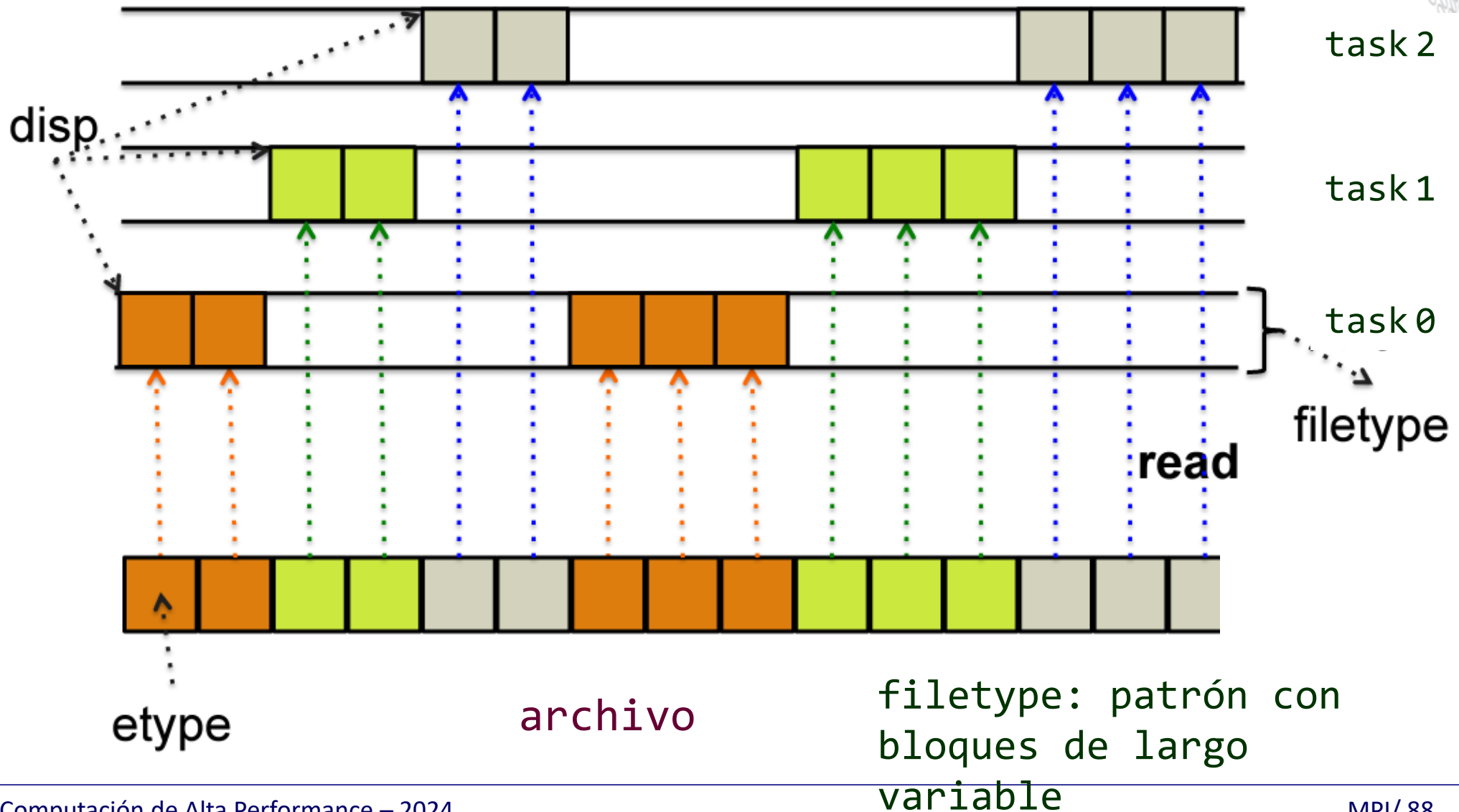
filetype: patrón (dos enteros seguidos de un gap de cuatro enteros)

Inicio del archivo



Vista: cada tarea repite el patrón, con diferentes desplazamientos

Vistas



Escritura sobre vistas

- Simplemente se invoca `MPI_File_write` con el file handler retornado en la creación de la vista

```
MPI_File_open(MPI_COMM_WORLD, "archivo.txt", MPI_MODE_CREATE  
             | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);  
MPI_File_set_view(fh,myrank * BUFSIZE * sizeof(int), MPI_INT,  
                 MPI_INT, "native", MPI_INFO_NULL);  
MPI_File_write(fh,buf,BUFSIZE,MPI_INT,MPI_STATUS_IGNORE);
```

- En el ejemplo el tipo de información del archivo es `MPI_INT`
- Se pueden utilizar tipos derivados de MPI

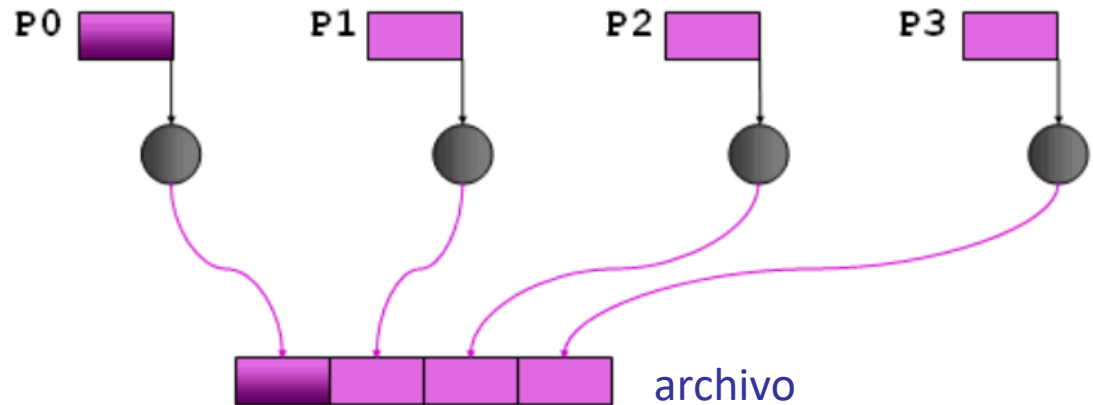
Vistas: ejemplo 1

- Escribir datos en bloques definidos por vistas

```
#define N 100
MPI_Datatype arraytype;
MPI_Offset disp;

disp = rank * sizeof(int) * N;
etype = MPI_INT;
MPI_Type_contiguous(N, MPI_INT,
MPI_Type_commit(&arraytype);
```

```
MPI_File_open(MPI_COMM_WORLD, "data1", MPI_MODE_CREATE | MPI_MODE_RDWR,
MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, arraytype, "native", MPI_INFO_NULL);
MPI_File_write(fh, buf, N, etype, MPI_STATUS_IGNORE);
```



Vistas: ejemplo 2

```
int buf[NW*2];  
  
MPI_File_open(MPI_COMM_WORLD, "data2", MPI_MODE_RDWR, MPI_INFO_NULL, &fh);  
/* archivo: dos bloques de NW enteros, separados NW*nprocs */  
MPI_Type_vector(2, NW, NW*nprocs, MPI_INT, &fileblk);  
MPI_Type_commit(&fileblk);  
  
disp = (MPI_Offset)rank*NW*sizeof(int);  
MPI_File_set_view(fh, disp, MPI_INT, fileblk, "native", MPI_INFO_NULL);  
/* escribir dos 'ablk', cada uno con NW enteros */  
MPI_Type_contiguous(NW, MPI_INT, &ablk);  
MPI_Type_commit(&ablk);  
MPI_File_write(fh, (void *)buf, 2, ablk, &status);
```

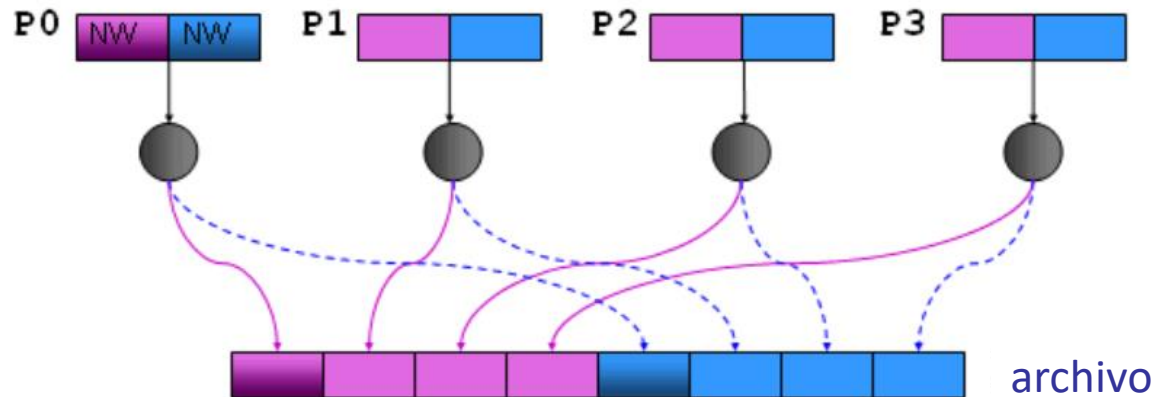
- El tipo de dato en memoria es un arreglo (contíguo) de NW enteros.
- El tipo de la vista es fileblk, desplazado $\text{rank} * \text{NW} * \text{sizeof}(\text{int})$.
- Dos unidades del tipo de dato se escriben en el archivo sobre el que se definió la vista

Vistas: ejemplo 2

```

int buf[NW*2];
MPI_File_open(MPI_COMM_WORLD, "data2", MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
/* archivo: dos bloques de NW enteros, separados NW*nprocs */
MPI_Type_vector(2, NW, NW*nprocs, MPI_INT, &fileblk);
MPI_Type_commit(&fileblk);

disp = (MPI_Offset)rank*NW*sizeof(int);
MPI_File_set_view(fh, disp, MPI_INT, fileblk, "native", MPI_INFO_NULL);
/* escribir dos 'ablk', cada uno con NW enteros */
MPI_Type_contiguous(NW, MPI_INT, &ablk);
MPI_Type_commit(&ablk);
MPI_File_write(fh, (void *)buf, 2, ablk, &status);
    
```

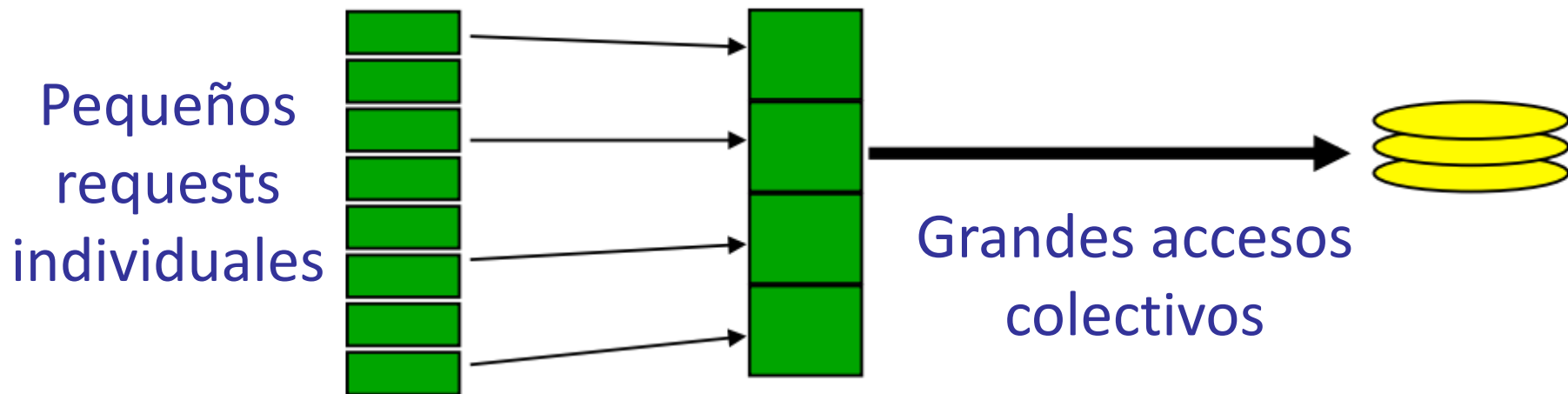


Vistas: acceso no contiguo

- El acceso no contiguo es fundamental en el paralelismo
- Ejemplo: arrays distribuidos almacenados en archivos
- Una de las principales ventajas de la E/S en MPI sobre la de Lixux es la capacidad de especificar accesos no contiguous en memoria y en archivos en una única invocación a función, usando tipos derivados.
- Permite implementaciones optimizadas para mejorar el desempeño
- Los accesos no contiguous combinados con operaciones de E/S colectivas permiten alcanzar los mejores niveles de desempeño

E/S colectiva

- Una optimización crítica para implementar E/S paralela
- Todos los procesos en un grupo (comunicador) deben invocar la función de E/S colectiva
- Permite al file system tener una vision global de las transferencias de datos e implementarlas en dos fases: 1) comunicación y 2) E/S
- Comunicaciones preliminares usan MPI para agregar datos
- Idea: construir grandes bloques de datos para lectura/escritura eficiente



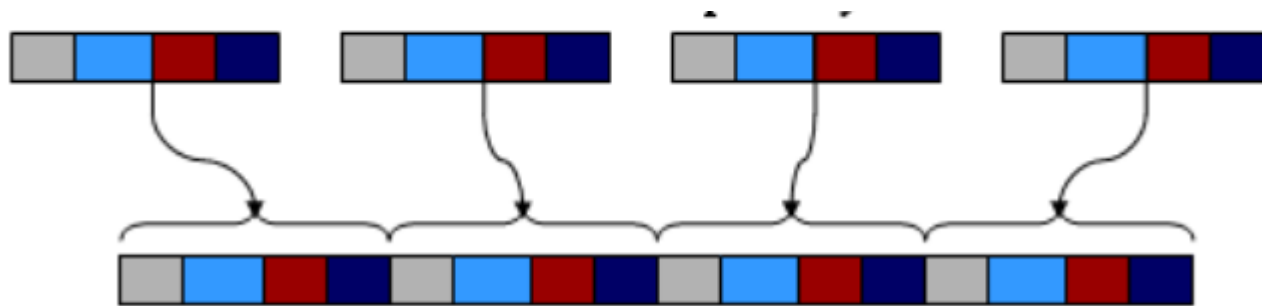
E/S colectiva

- Idea: construir grandes bloques de datos para lectura/escritura eficiente
- Requests de diferentes procesos pueden agruparse
- Particularmente efectivo cuando los accesos de diferentes procesos son no-contiguos e intercalados



Estructuras de memoria en cuatro procesos

MPI recolecta en buffers intermedios



MPI escribe en archivo con el formato especificado

E/S colectiva

- `MPI_File_write_at_all` escritura colectiva para procesos en un grupo
 - El grupo es el del comunicador indicado en `MPI_File_open`
- ```
int MPI_File_write_at_all (MPI_File fp, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
```
- `fp`: handler para acceso
  - `offset`: offset en el archivo (relativo al inicio del archivo)
  - `buffer`: dirección del buffer a escribir
  - `count`: número de elementos a escribir
  - `datatype`: tipo de datos MPI (estándar o derivado)
  - `status`: información adicional
- `MPI_File_read_at_all` tiene el mismo header
  - `MPI_File_write_at_all` y `MPI_File_read_at_all` indican la posición en el archivo, no necesitan modificar el file pointer y por ello son thread-safe, a diferencia de una invocación separada a `MPI_File_seek`



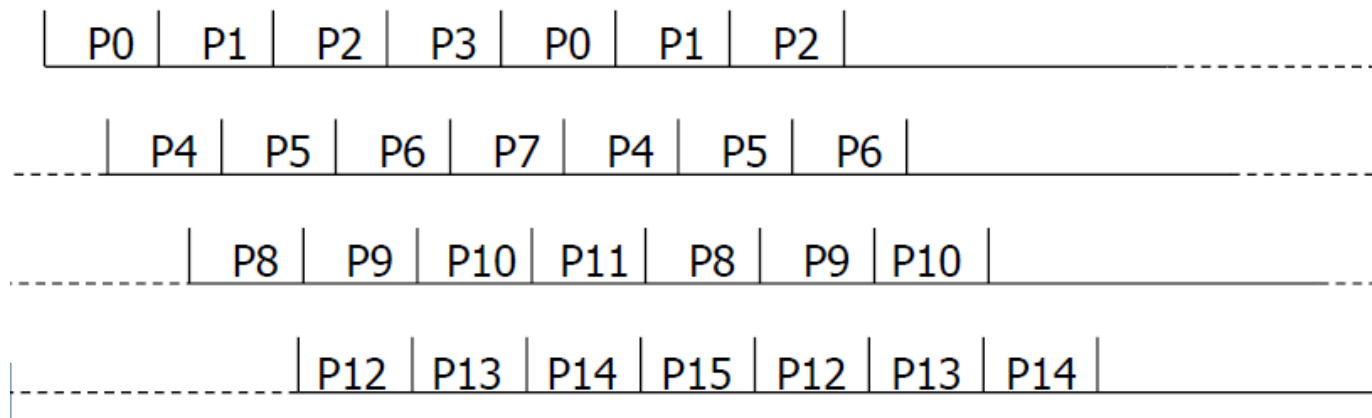
# E/S: niveles de acceso

- Ejemplo: acceso a una gran estructura de datos [array] distribuida entre 16 procesos

|     |     |     |     |
|-----|-----|-----|-----|
| P0  | P1  | P2  | P3  |
| P4  | P5  | P6  | P7  |
| P8  | P9  | P10 | P11 |
| P12 | P13 | P14 | P15 |

Cada cuadrado representa un subarray en la memoria de un único proceso

- Patrón de accesos en el archivo



# E/S: niveles de acceso

- Acceso de nivel 0: cada proceso realiza lecturas independientes para cada fila en el array (como en Linux/Unix)

```
MPI_File_open(..., file, ..., &fh);
for (i=0; i<n_local_rows; i++) {
 MPI_File_seek(fh, ...);
 MPI_File_read(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

# E/S: niveles de acceso

- Acceso de nivel 1: similar al acceso de nivel 0, pero cada proceso utiliza funciones de E/S colectiva

```
MPI_File_open(MPI_COMM_WORLD, file, ..., &fh);
for (i=0; i<n_local_rows; i++) {
 MPI_File_seek(fh, ...);
 MPI_File_read_all(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

# E/S: niveles de acceso

- Acceso de nivel 2: cada proceso crea un tipo de datos derivado para describir el patrón de acceso no contiguo, define una vista y realiza lecturas independientes

```
MPI_Type_create_subarray(..., &subarray, ...);
MPI_Type_commit(&subarray);
MPI_File_open(..., file, ..., &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read(fh, A, ...);
MPI_File_close(&fh);
```

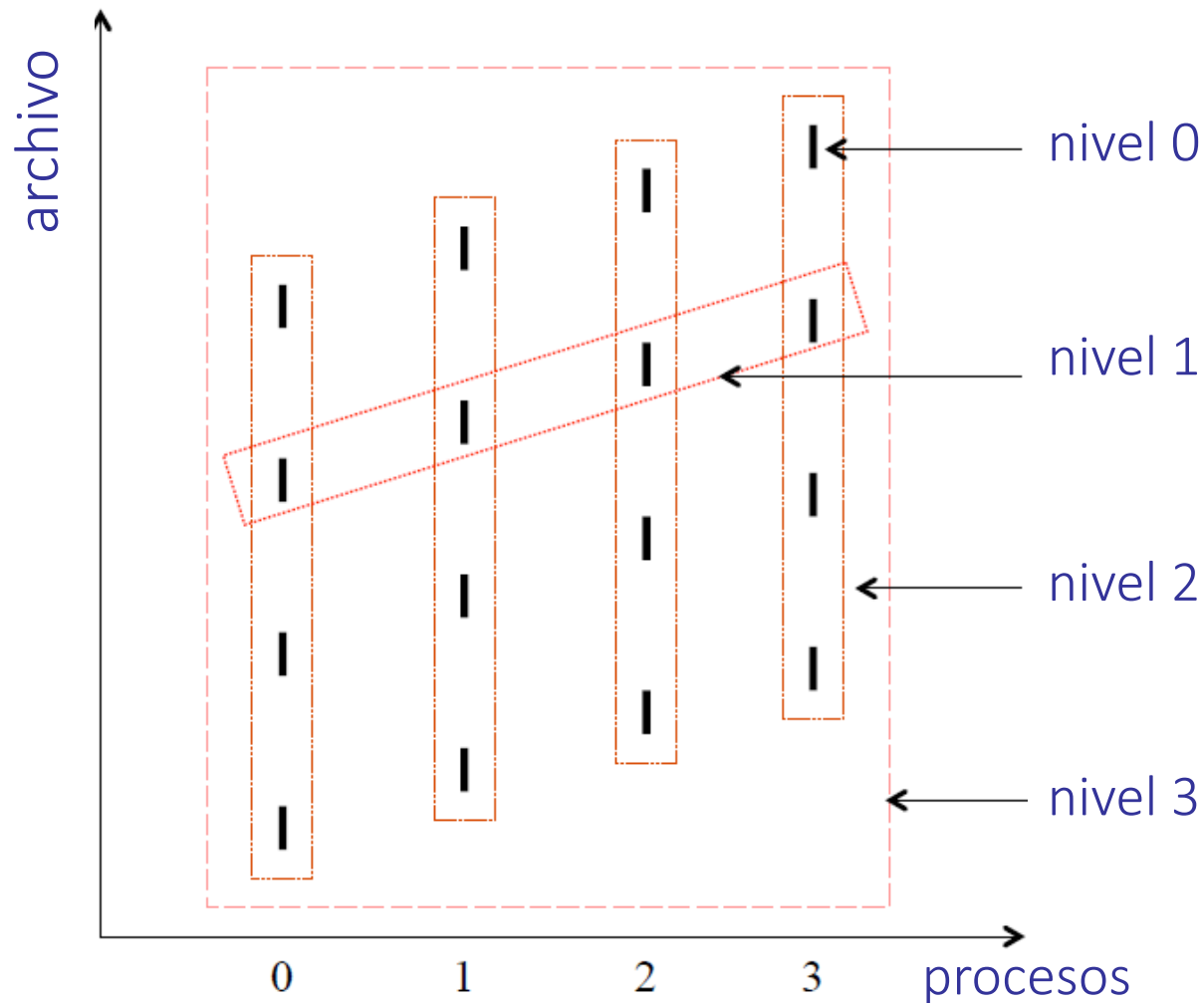
# E/S: niveles de acceso

- Acceso de nivel 3: similar al acceso de nivel 2, excepto que cada proceso utiliza funciones de E/S colectiva

```
MPI_Type_create_subarray(..., &subarray, ...);
MPI_Type_commit(&subarray);
MPI_File_open(MPI_COMM_WORLD, file,..., &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read_all(fh, A, ...);
MPI_File_close(&fh);
```

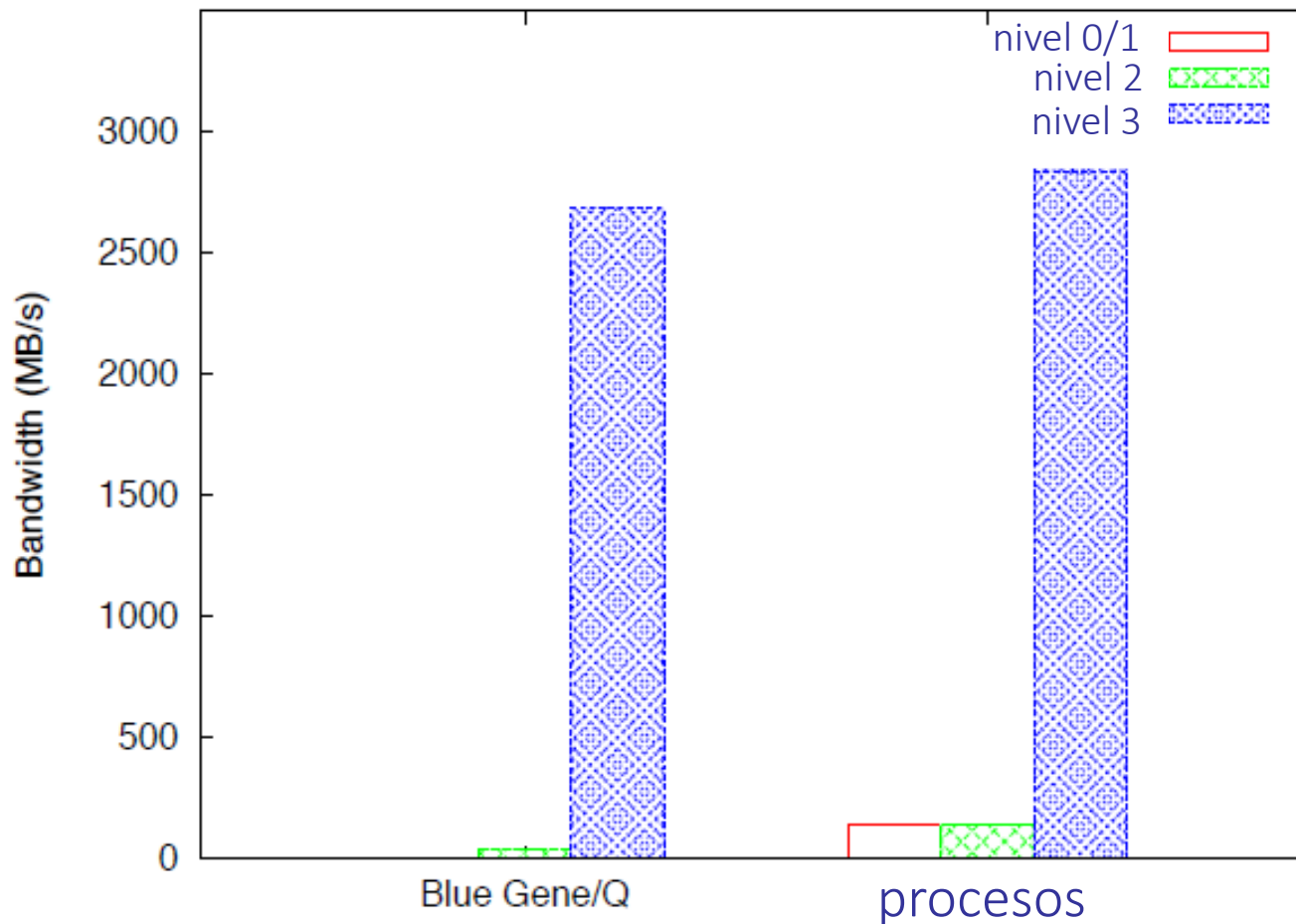
# E/S: niveles de acceso

- Los cuatro niveles de acceso



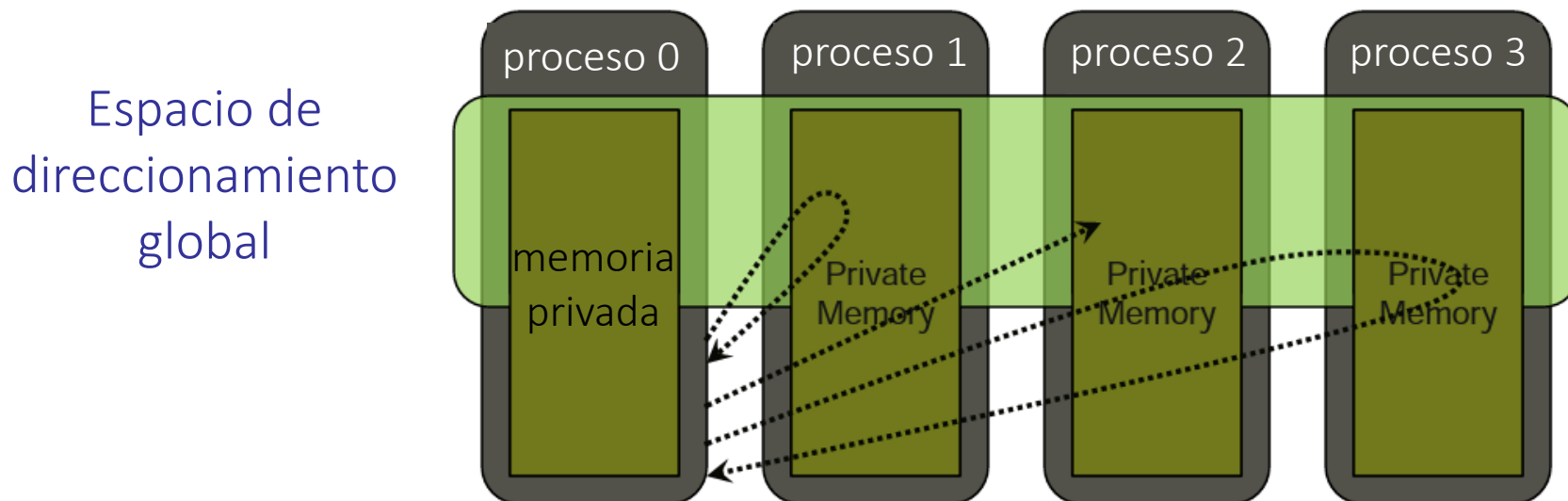
# E/S: niveles de acceso y desempeño

- Desempeño de la escritura de un array 3D usando 256 procesos, en dos supercomputadoras



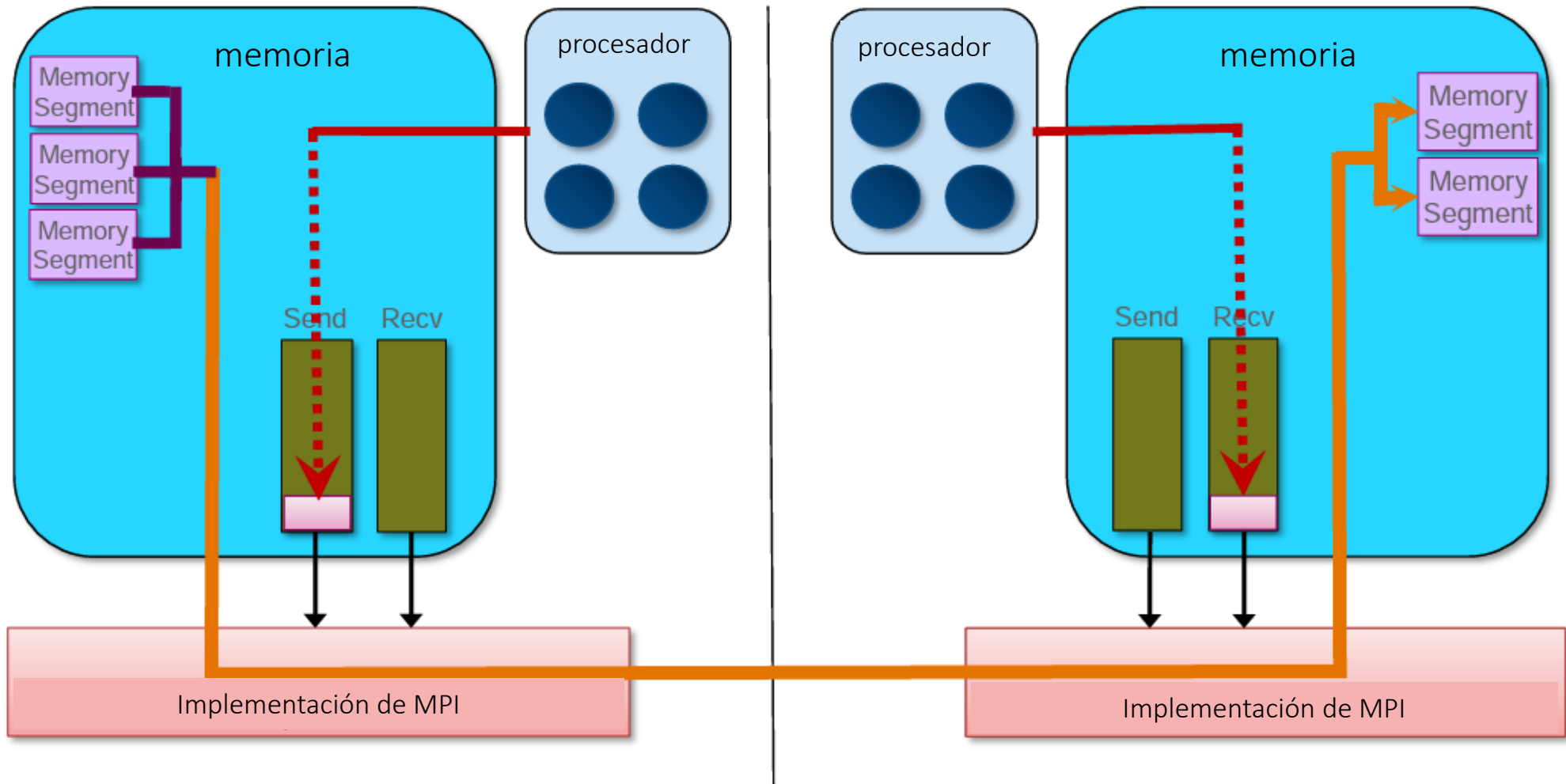
# Comunicaciones de una vía

- La idea es desacoplar el movimiento/comunicación de datos con la necesidad de sincronización
- El proceso local puede copiar datos sin necesidad de que el proceso remoto sincronice
- Un proceso expone una parte de la memoria para acceso (lectura/escritura) por parte de otros procesos

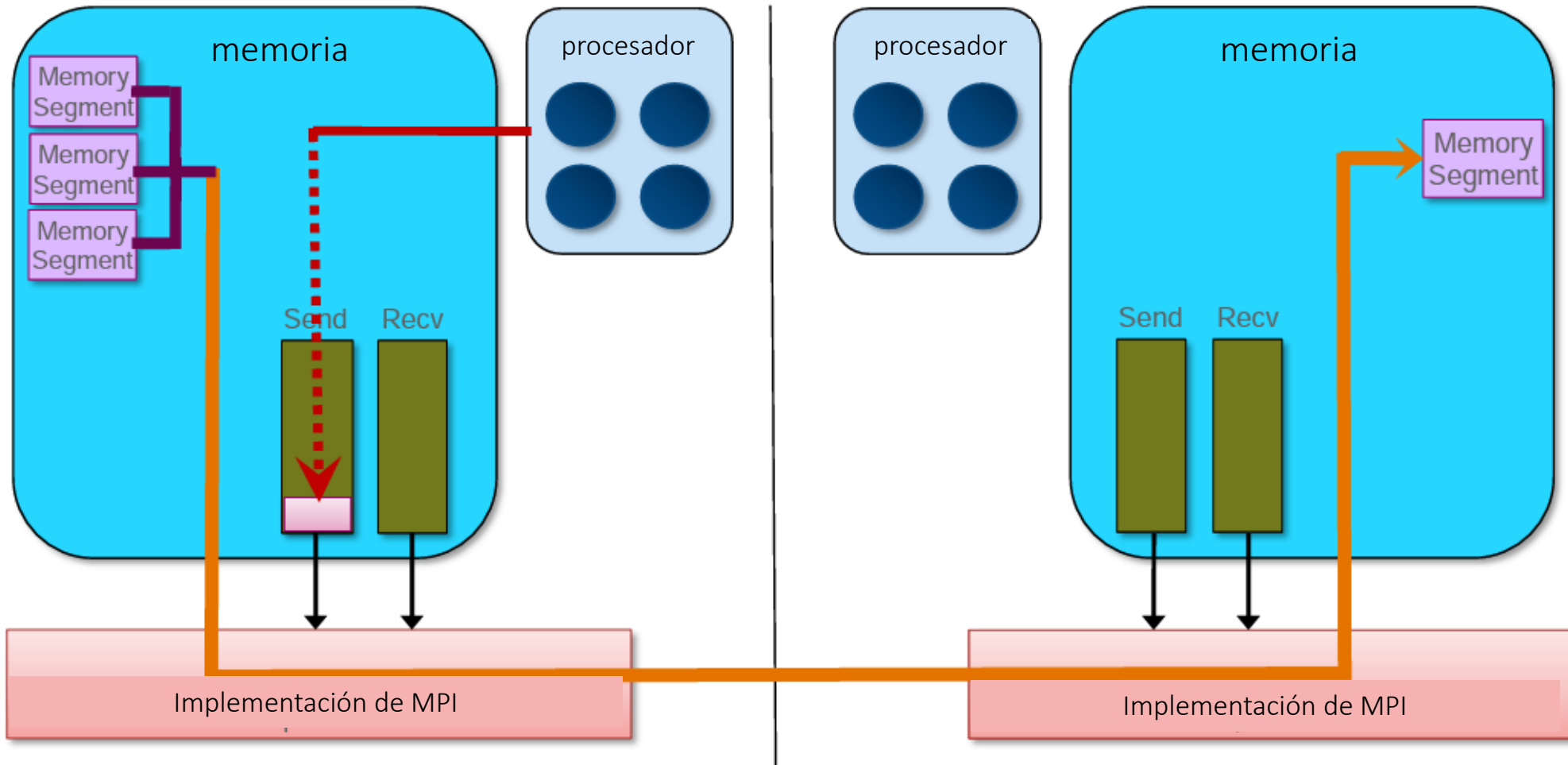




# Comunicaciones de dos vías



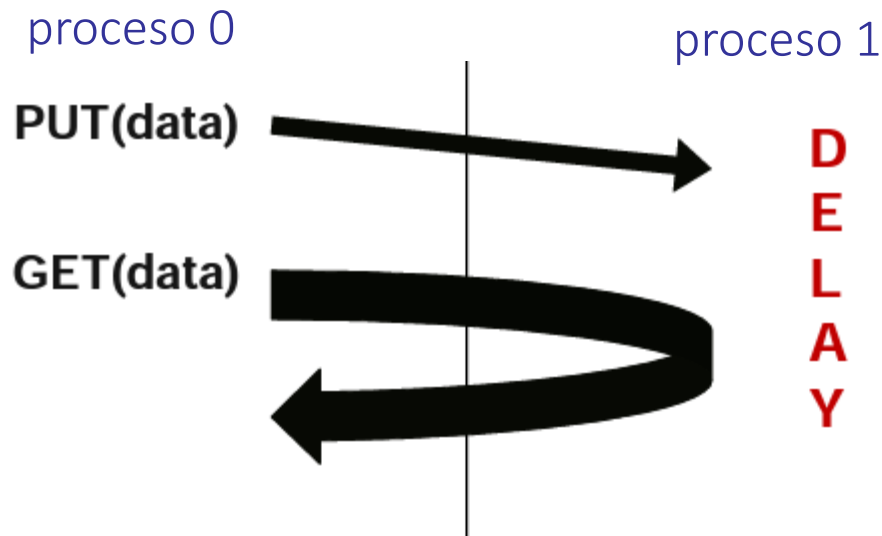
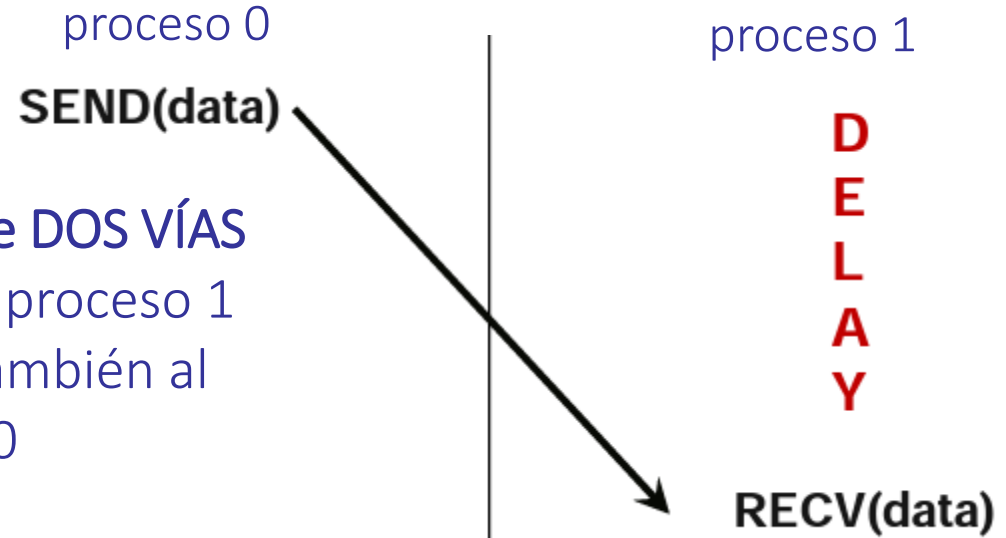
# Comunicaciones de una vía



# Comunicaciones de una y dos vías

- Desacopla al proceso emisor del proceso receptor

**COMUNICACIÓN de DOS VÍAS**  
Una demora en el proceso 1 puede detener también al proceso 0



**COMUNICACIÓN de UNA VÍA**  
Una demora en el proceso 1 no afecta al proceso 0

# Comunicaciones de una vía/RMA

- Existe un modelo formal para implementar las comunicaciones de una vía en MPI-3: el acceso a memoria remota (RMA)
- Conceptos y acciones implementadas:
  - Creación
  - Lectura y escritura
  - Sincronización de datos
  - Modelos de memoria
- La memoria usada por un proceso solo es accesible localmente
- Luego de alocar la memoria, el proceso debe realizar una invocación explícita a una rutina MPI para declarar una variable o region de memoria como accesible remotamente.
- Luego de declarada la memoria remota, todos los procesos participantes pueden leer y escribir datos sin sincronizarse explícitamente

# RMA en MPI

- Para MPI RMA es “Un grupo de procesos que colectivamente crean una ventana de memoria”
- Cuatro modelos:
  - **MPI\_WIN\_ALLOCATE**: crear un espacio de memoria (buffer) y hacerlo accesible remotamente.
  - **MPI\_WIN\_CREATE**: hacer accesible un espacio de memoria (buffer) ya creado
  - **MPI\_WIN\_CREATE\_DYNAMIC**: el espacio de memoria (buffer) a compartir aún no está creado, pero lo será en el futuro. Permite agregar/remover dinámicamente buffers a/de una ventana.
  - **MPI\_WIN\_ALLOCATE\_SHARED**: múltiples procesos en el mismo nodo de cómputo comparten un espacio de memoria (buffer).

# Creación de ventana

- `MPI_Win_allocate` crea una región de memoria remotamente accesible en una ventana RMA. Solo los datos expuestos en la ventana son accesibles mediante operaciones RMA.

```
MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info
info, MPI_Comm comm, void *baseptr, MPI_Win * win)
```

- **size**: tamaño de la ventana, en bytes
- **disp\_unit**: unidad para desplazamientos en la ventana, en bytes
- **info**: información adicional
- **comm**: comunicador asociado al grupo de procesos de la ventana
- **baseptr**: dirección base de la ventana en memoria local
- **win**: handler de la ventana definida (puntero)

# MPI\_Win\_allocate: ejemplo

```
int main(int argc, char ** argv) {
int* a;
MPI_Win win;

MPI_Init(&argc, &argv);
/* creación colectiva de una ventana para acceso remoto */
MPI_Win_allocate(1000*sizeof(int), sizeof(int), MPI_INFO_NULL,
 MPI_COMM_WORLD, &a, &win);
/* El vector 'a' es accesible a todos los procesos
 en MPI_COMM_WORLD */
MPI_Win_free(&win);
MPI_Finalize();
return 0;
}
```

# Creación de ventana

- `MPI_Win_create` crea un objeto ventana para comunicaciones de una vía mediante operaciones RMA

```
MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
 MPI_Info info, MPI_Comm comm, MPI_Win * win)
```

- **base**: dirección inicial de la ventana
  - **size**: tamaño de la ventana, en bytes
  - **disp\_unit**: unidad para desplazamientos en la ventana, en bytes
  - **info**: información adicional
  - **comm**: comunicador asociado al grupo de procesos de la ventana
  - **win**: handler de la ventana definida (puntero)
- La diferencia con `MPI_Win_allocate` es que `MPI_Win_create` recibe un buffer o variable de memoria ya alocada



# MPI\_Win\_create: ejemplo

```
Int main(int argc, char **argv) {
int* a;
MPI_Win win;
MPI_Init(&argc, &argv);
/* crear memoria privada y utilizarla */
MPI_Alloc_mem(1000*sizeof(int), MPI_INFO_NULL, &a);
/* creación colectiva de una ventana para acceso remoto */
MPI_Win_create(a, 1000*sizeof(int), sizeof(int),
MPI_INFO_NULL, MPI_COMM_WORLD, &win);
/* El vector 'a' es accesible a todos los procesos
en MPI_COMM_WORLD */
MPI_Win_free(&win);
MPI_Free_mem(a);
MPI_Finalize();
return 0;
}
```

# Creación de ventana dinámica

- `MPI_Win_create_dynamic` crea una ventana para comunicaciones de una vía mediante operaciones RMA luego agregar para datos

```
MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm,
 MPI_Win * win)
```

- **info**: información adicional
  - **comm**: comunicador asociado al grupo de procesos de la ventana
  - **win**: handler de la ventana definida (puntero)
- Inicialmente la ventana está vacía.
  - Los procesos pueden acoplarse/desacoplarse y acceder a la memoria expuesta en la ventana usando `MPI_Win_attach` y `MPI_Win_detach`
  - El origen de la ventana es `MPI_BOTTOM`
  - Todos los desplazamientos son relativos a `MPI_BOTTOM`
  - Un proceso debe comunicar el desplazamiento luego del attach

# MPI\_Win\_create\_dynamic: ejemplo

```
Int main(int argc, char **argv) {
int* a; MPI_Win win;

MPI_Init(&argc, &argv);
MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &win);
/* crear memoria privada y utilizarla */
a = (int*) malloc(1000 * sizeof(int)); a[0] = 1; a[1] = 2;
/* Acoplarse a la ventana y exponer el vector a */
MPI_Win_attach(win, a, 1000*sizeof(int));
/* El vector 'a' es accesible a los procesos en MPI_COMM_WORLD */
/* Desacoplarse de la ventana, el vector a ya no es compartido*/
MPI_Win_detach(win, a);
free(a);
MPI_Win_free(&win);
MPI_Free_mem(a);
MPI_Finalize();
return 0; }
```

# Memoria remota: movimiento de datos

- MPI provee funciones para que un proceso local pueda leer, escribir y modificar atómicamente datos en ventanas de memoria remota
- `MPI_Put`: escribir (local → remota)
- `MPI_Get`: leer (remota → local)
- `MPI_Accumulate`: acumular (local → remota, atómica)
- `MPI_Get_accumulate`: leer y acumular (remota → remota, atómica)
- `MPI_Compare_and_swap`: comparar e intercambiar, remota → remota)
- `MPI_Fetch_and_op`: similar a `MPI_Get_accumulate`, remota → remota atómica - atómica)
- Las operaciones son **no bloqueantes**
- Las operaciones se completan (en origen y destino) invocando una operación de sincronización sobre la ventana

# Memoria remota: movimiento de datos

- Las operaciones de movimiento de datos son **no bloqueantes**
- El buffer de la aplicación (local) no debe modificarse hasta que la operación se haya completado (mediante la invocación a la función de sincronización).
- No debe accederse concurrentemente a la misma dirección de memoria en una ventana (conflicto): si una ubicación se actualiza mediante MPI\_Put o MPI\_Accumulate, la ubicación no puede ser accedida por un MPI\_Get o por otra operación de RMA mientras la operación de actualización no se haya completado.
- Una ventana no puede actualizarse mediante MPI\_Put o MPI\_Accumulate simultáneamente con una operación de modificación local, aunque las actualizaciones se realicen sobre ubicaciones diferentes de la ventana.
- El proceso invocante puede ser también destino: un proceso puede usar operaciones RMA para mover datos en su propia memoria.

# Memoria remota: movimiento de datos

- MPI\_Put permite escribir datos en una ventana

```
MPI_Put(void *origin_addr, int origin_count, MPI_Datatype
origin_dtype, int target_rank, MPI_Aint target_disp, int
target_count, MPI_Datatype target_dtype, MPI_Win win)
```

- origin\_addr: dirección inicial del buffer de origen
  - origin\_count: número de entradas en el buffer de origen
  - origin\_dtype: tipo de datos de las entradas en el buffer de origen
  - target\_rank: rango del proceso destino
  - target\_disp: desplazamiento desde el inicio de la ventana del buffer destino
  - target\_count: número de entradas en el buffer de destino
  - target\_dtype: tipo de datos de las entradas en el buffer de destino
  - win: ventana para la comunicación de una vía
- Copia origin\_count elementos de tipo origin\_datatype, comenzando en origin\_addr del proceso origen a la ventana win del proceso target\_rank

# Memoria remota: movimiento de datos

- `MPI_Put` copia `origin_count` elementos de tipo `origin_datatype`, comenzando en `origin_addr` del proceso origen a la ventana `win` del proceso `target_rank`
- Los datos se escriben en el buffer de destino en la dirección,  $\text{win\_base} + \text{target\_disp} \times \text{disp\_unit}$ , siendo `win_base` la dirección base de la ventana y `disp_unit` el desplazamiento unitario, ambos especificados en la inicialización de la ventana, por el proceso destino
- La transferencia de datos es similar a un envío (`MPI_Send`) de `origin_count` elementos de tipo `origin_datatype`, ubicados a partir de `origin_addr` desde el proceso origen al proceso `target_rank`, que los recibe con `MPI_Recv` en `target_addr` (buffer de destino), sobre un comunicador que englobe a ambos procesos.
- Los datos a copiar deben poder contenerse (sin truncamiento) en el buffer de destino y éste debe poder contenerse en la ventana de destino.

- MPI\_Get permite leer datos de una ventana

```
MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
origin_dtype, int target_rank, MPI_Aint target_disp, int
target_count, MPI_Datatype target_dtype, MPI_Win win)
```

- origin\_addr: dirección inicial del buffer de origen
  - origin\_count: número de entradas en el buffer de origen
  - origin\_dtype: tipo de datos de las entradas en el buffer de origen
  - target\_rank: rango del proceso destino
  - target\_disp: desplazamiento desde el inicio de la ventana del buffer destino
  - target\_count: número de entradas en el buffer de destino
  - target\_dtype: tipo de datos de las entradas en el buffer de destino
  - win: ventana para la comunicación de una vía
- Lee datos desde el proceso destino (target\_rank) al proceso origen
  - Similar a MPI\_Put, con direcciones invertidas



# Memoria remota: movimiento de datos

- MPI\_Accumulate aplica operaciones sobre datos de una ventana

```
MPI_Accumulate(void *origin_addr, int origin_count, MPI_Datatype
origin_dtype, int target_rank, MPI_Aint target_disp, int
target_count, MPI_Datatype target_dtype, MPI_Op op; MPI_Win win)
```

- **origin\_addr**: dirección inicial del buffer de origen
  - **origin\_count**: número de entradas en el buffer de origen
  - **origin\_dtype**: tipo de datos de las entradas en el buffer de origen
  - **target\_rank**: rango del proceso destino
  - **target\_disp**: desplazamiento desde el inicio de la ventana del buffer destino
  - **target\_count**: número de entradas en el buffer de destino
  - **target\_dtype**: tipo de datos de las entradas en el buffer de destino
  - **op**: operación de reducción
  - **win**: ventana para la comunicación de una vía
- Reduce datos de origen y destino en el buffer de destino aplicando la operación indicada (op) como recombinao

# Memoria remota: movimiento de datos

- `MPI_Accumulate` reduce datos de origen y destino en el buffer de destino aplicando la operación indicada (`op`) como recombinaor
- Operaciones predefinidas: `MPI_SUM`, `MPI_PROD`, `MPI_OR`, `MPI_REPLACE`, `MPI_NO_OP`
- No admite operaciones de finidas por el usuario
- Tripletas de descripción de datos diferentes para origen y destino
- Los tipos de datos de los operandos deben concordar con los de la operación `op`
- `MPI_REPLACE` implementa  $f(a,b) = b$  , con un `MPI_Put` atómico implícito

- MPI\_Get\_Accumulate implementa una combinación de leer-modificar-escribir de manera atómica

```
MPI_Get_Accumulate(void *origin_addr, int origin_count,
MPI_Datatype origin_dtype, void *result_addr, int
result_count, MPI_Datatype result_dtype, int target_rank,
MPI_Aint target_disp, int target_count, MPI_Datatype
target_dtype, MPI_Op op; MPI_Win win)
```

- origin\_addr: dirección inicial del buffer de origen
- origin\_count: número de entradas en el buffer de origen
- origin\_dtype: tipo de datos de las entradas en el buffer de origen
- result\_addr: dirección inicial del buffer resultado
- result\_count: número de entradas en el buffer resultado
- result\_dtype: tipo de datos de las entradas en el buffer resultado
- ...

```
MPI_Get_Accumulate(void *origin_addr, int origin_count,
MPI_Datatype origin_dtype, void *result_addr, int
result_count, MPI_Datatype result_dtype, int target_rank,
MPI_Aint target_disp, int target_count, MPI_Datatype
target_dtype, MPI_Op op; MPI_Win win)
```

- ...
- **target\_rank**: rango del proceso destino
- **target\_disp**: desplazamiento desde el inicio de la ventana del buffer destino
- **target\_count**: número de entradas en el buffer de destino
- **target\_dtype**: tipo de datos de las entradas en el buffer de destino
- **op**: operación de reducción
- **win**: ventana para la comunicación de una vía
- Retorna en el buffer resultado el contenido del buffer destino antes de aplicar la operación de acumulación

# Memoria remota: movimiento de datos

- `MPI_Fetch_and_op` es una versión simple de `MPI_Get_Accumulate`, implementa una combinación de leer-modificar-escribir de manera atómica para un elemento de un tipo de dato predefinido

```
MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
 MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,
 MPI_Op op, MPI_Win win)
```

  - `origin_addr`: dirección inicial del buffer de origen
  - `result_addr`: dirección inicial del buffer resultado
  - `datatype`: tipo de datos del elemento a modificar
  - `target_rank`: rango del proceso destino
  - `target_disp`: desplazamiento desde el inicio de la ventana del buffer destino
  - `op`: operación de reducción
  - `win`: ventana para la comunicación de una vía
- Todos los buffers comparten el tipo de dato predefinido. No se incluye un argumento `count` (que es 1)

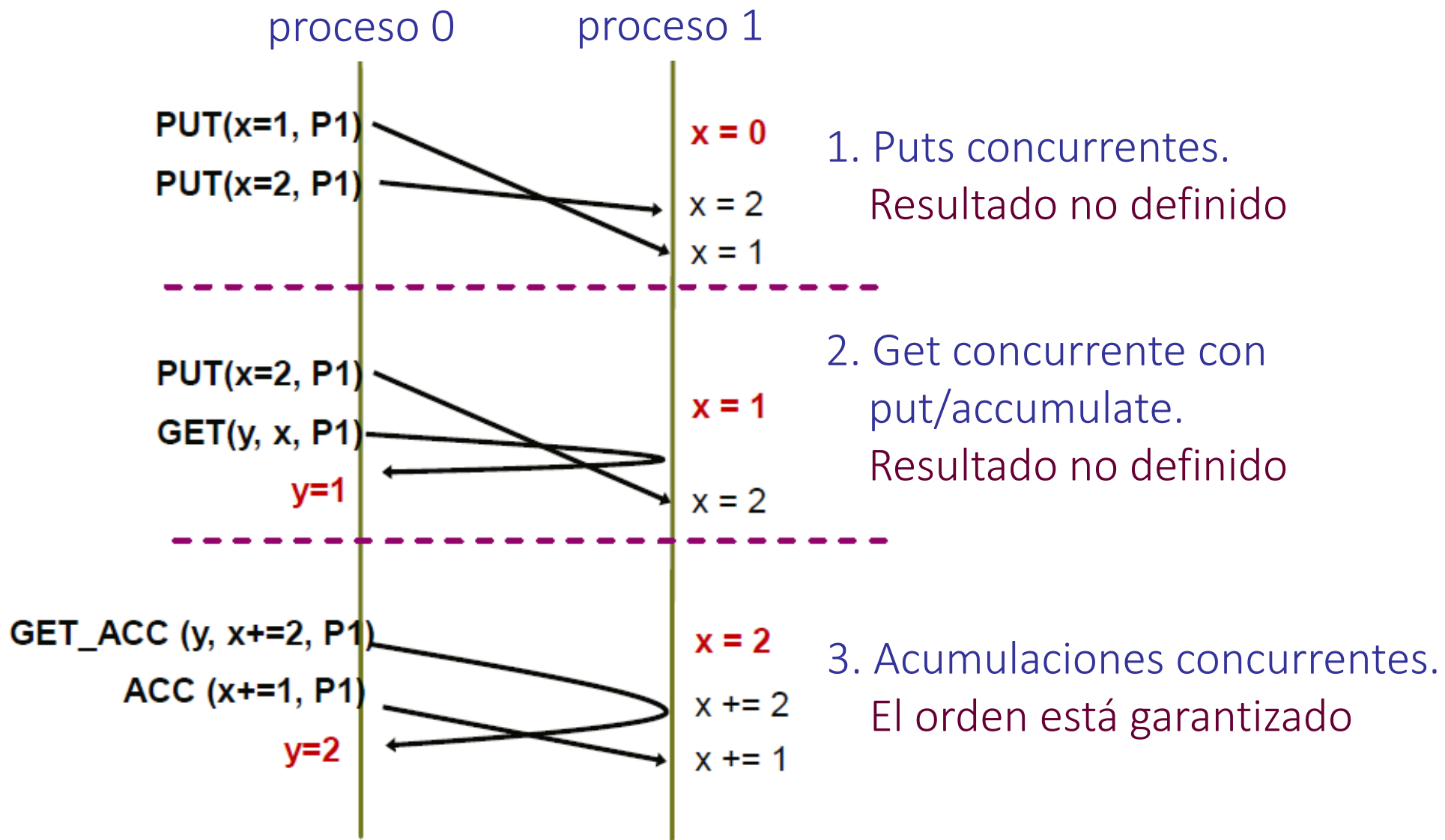
# Memoria remota: movimiento de datos

- `MPI_Compare_and_swap` compara elementos y los intercambia  
`MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr, void *result_addr, MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Win win)`
  - `origin_addr`: dirección inicial del buffer de origen
  - `compare_addr`: dirección inicial del buffer de comparación
  - `datatype`: tipo de datos del elemento a comparar
  - `target_rank`: rango del proceso destino
  - `target_disp`: desplazamiento desde el inicio de la ventana del buffer destino
  - `win`: ventana para la comunicación de una vía
- Compara un elemento de tipo `datatype` en el buffer de comparación (`compare_addr`) con el elemento en `target_disp` en la ventana de destino (especificada por `target_rank` y `win`) y reemplaza el valor en el destino con el valor en el origen (`origin_addr`) si los elementos comparados son idénticos. El valor original en el destino se retorna en `result_addr`.

# Acceso concurrente con RMA

- RMA no garantiza un correcto ordenamiento de las operaciones de lectura y escritura
- El resultado de puts concurrentes a la misma ubicación no está definido
- El resultado de un get concurrente con un put/accumulate no está definido
- Puede ser un resultado incorrecto o basura en ambos casos
- El resultado de operaciones de acumulación concurrentes sobre la misma ubicación se definen según el orden en que ocurrieron
  - MPI\_Put atómico: MPI\_Accumulate con op = MPI\_REPLACE
  - MPI\_Get atómico: MPI\_Get\_accumulate con op = MPI\_NO\_OP
- Las operaciones de acumulación ejecutadas por un proceso están ordenadas por defecto.
- Se puede especificar que no se siga el orden, e indicar un modelo de acceso a memoria: RAW (read-after-write), WAR, RAR, or WAW

# Acceso concurrente con RMA





# RMA: modelo de acceso de datos

- Modelo de acceso de datos
  - ¿Cuándo se permite a un proceso leer/escribir memoria remota?
  - ¿Cuándo están disponibles los datos escritos por el proceso X para que los lea el proceso Y?
- El **modelo de sincronización** define la semántica.
- RMA en MPI provee tres modelos de sincronización
  - Fence (comunicación de destino activo)
  - Post-start-complete-wait (comunicación de destino activo generalizado)
  - Lock/Unlock (comunicación de destino pasivo)

# RMA: modelo de acceso de datos

- **Comunicación de destino activo:** los datos se mueven de la memoria de un proceso a la memoria de otro, y **ambos procesos están explícitamente involucrados en la comunicación.**
- Patrón similar al pasaje de mensajes, excepto que los parámetros de transferencia de datos son indicados por el proceso origen y el proceso destino solo participa en la sincronización.
- **Comunicación de destino pasivo :** los datos se mueven de la memoria de un proceso a la memoria de otro, y solo el proceso origen está explícitamente involucrado en la transferencia.
- Dos procesos de origen pueden comunicarse accediendo a la misma ubicación en una ventana de destino de un proceso diferente a ambos (el proceso destino no participa explícitamente en la comunicación).
- Paradigma similar a memoria compartida: todos los procesos pueden acceder a los datos compartidos, independientemente de su ubicación.

# RMA: modelo de acceso de datos

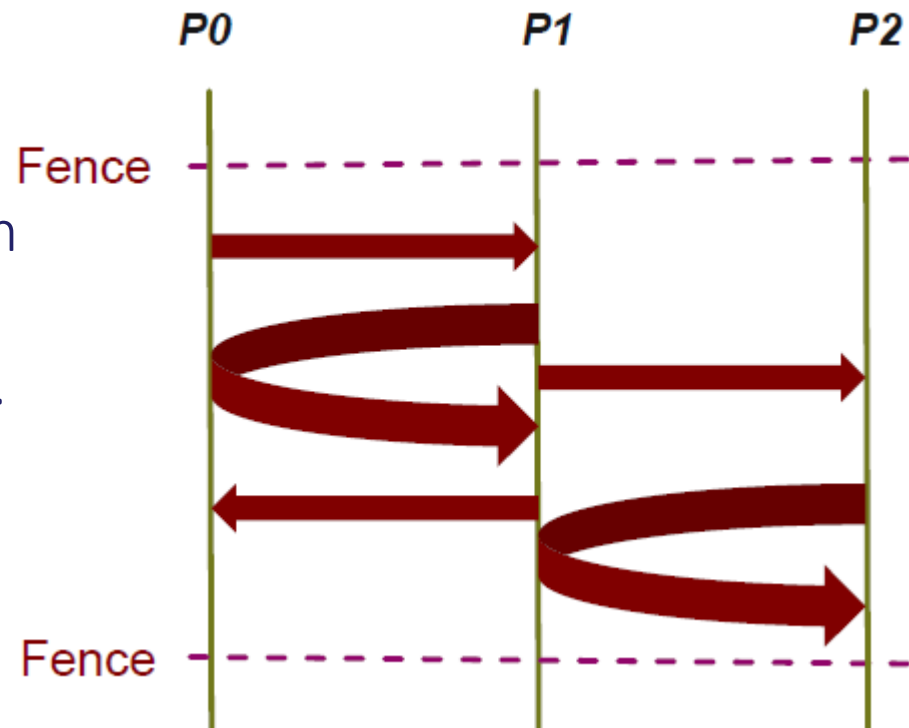
- El acceso a los datos ocurre en “épocas”
- Épocas de acceso: contienen un conjunto de operaciones realizadas por un proceso origen
- Épocas de exposición: permite que los procesos remotos actualicen la ventana de un proceso destino
- Las épocas definen la semántica de ordenación y finalización.
- Los modelos de sincronización proporcionan mecanismos para establecer épocas, por ejemplo, épocas de inicio, finalización y sincronización.

# RMA: sincronización

- `MPI_Win_fence` inicia y finaliza una época de exposición de datos en todos los procesos de una ventana.  
`MPI_Win_fence(int assert, MPI_Win win)`
  - **assert**: condición sobre el contexto de la invocación, `assert = 0` para un caso general
  - **win**: ventana para la comunicación de una vía
- Modelo de sincronización colectiva
  - Todos los procesos en el grupo de `win` deben ejecutar `MPI_Win_fence` para iniciar una época
  - Todos los procesos pueden realizar operaciones de `get/put` para leer y escribir datos en la ventana
  - Todos los procesos en el grupo de `win` deben ejecutar `MPI_Win_fence` para finalizar la época
  - Todas las operaciones se completan con la invocación de finalización

# RMA: sincronización

- `MPI_Win_fence` provee soporte para un modelo de sincronización simple, (loosely-synchronous model) comúnmente utilizado en cómputos paralelos en los que una fase de cómputo global se alterna con fases de comunicaciones globales.
- El mecanismo es útil para algoritmos poco sincronizados, donde la topología del grafo de comunicaciones cambia con mucha frecuencia o en los que cada proceso se comunica con muchos otros.
- Las invocaciones a `MPI_Win_fence` deben preceder y seguir a todas las operaciones de get, put y accumulate que se sincronizan con ese fence



# RMA: sincronización general activa

- `MPI_Win_start/post` inicia una época de accesos RMA para una ventana.  
`MPI_Win_start(MPI_Group group, int assert, MPI_Win win)`  
`MPI_Win_post(MPI_Group group, int assert, MPI_Win win)`
  - `group`: grupo de procesos destino
  - `assert`: condición sobre el contexto, `assert = 0` para un caso general
  - `win`: ventana para la comunicación de una vía
- Invocaciones a funciones RMA sobre `win` durante la época solo pueden acceder a ventanas en procesos de `group`.
- Cada proceso destino en `group` debe invocar a `MPI_Win_post`.
- Accesos RMA a cada ventana de destino serán diferidos, si es necesario, hasta que el proceso destino invoque el `MPI_Win_post` correspondiente.
- `MPI_Win_complete/wait` finaliza la época de accesos para la ventana  
`MPI_Win_complete(MPI_Win win)` en procesos origen  
`MPI_Win_wait(MPI_Win win)` en procesos destino

# RMA: sincronización pasiva

- Comunicaciones asíncronas de una vía, sin participación del proceso destino (similar a un modelo de memoria compartida)
- `MPI_Win_lock` inicia una época de accesos RMA para una ventana en modo pasivo.  
`MPI_Win_lock(int locktype, int rank, int assert, MPI_Win win)`
  - `locktype`: tipo de acceso/lock
  - `rank`: rango del proceso destino (el que tiene la ventana)
  - `assert`: condición sobre el contexto, `assert = 0` para un caso general
  - `win`: ventana para la comunicación de una vía
- La invocación solo la realiza el proceso origen (no el proceso destino)
- Se pueden iniciar múltiples épocas de acceso pasivo para diferentes procesos
- No se pueden definir épocas de acceso concurrentes para el mismo proceso (afecta a los threads)

# RMA: sincronización pasiva

```
MPI_Win_lock(int locktype, int rank, int assert, MPI_Win win)
```

- Tipos de acceso (locktype):
  - **SHARED**: otros procesos que indiquen el modo shared pueden acceder concurrentemente
  - **EXCLUSIVE**: ningún otro proceso puede acceder concurrentemente
- `MPI_Win_unlock` finaliza la época de accesos RMA para una ventana en modo pasivo

```
MPI_Win_unlock(int rank, MPI_Win win)
```

  - **rank**: rango del proceso destino
  - **win**: ventana para la comunicación de una vía
- `MPI_Win_flush/flush_local` complete las operaciones RMA al destino

```
MPI_Win_flush/flush_local(int rank, MPI_Win win)
```

  - **rank**: rango del proceso destino
  - **win**: ventana para la comunicación de una vía



# RMA: sincronización pasiva

- Comunicaciones asíncronas de una vía, sin participación del proceso destino (similar a un modelo de memoria compartida)
- `MPI_Win_start/post` inicia una época de accesos RMA para una ventana.  
`MPI_Win_start/post(MPI_Group group, int assert, MPI_Win win)`
  - `group`: grupo de procesos destino
  - `assert`: condición sobre el contexto, `assert = 0` para un caso general
  - `win`: ventana para la comunicación de una vía
- Invocaciones a funciones RMA sobre `win` durante la época solo pueden acceder a ventanas en procesos de `group`.
- Cada proceso destino en `group` debe invocar a `MPI_Win_start/post`.
- Accesos RMA a cada ventana de destino serán diferidos, si es necesario, hasta que el proceso destino invoque el `MPI_Win_post` correspondiente.

# RMA: sincronización pasiva

- Comunicaciones asíncronas de una vía, sin participación del proceso destino (similar a un modelo de memoria compartida)
- `MPI_Win_complete/wait` finaliza la época de accesos para la ventana  
`MPI_Win_complete(MPI_Win win)` en procesos origen  
`MPI_Win_wait(MPI_Win win)` en procesos destino

# Acceso a memoria remota: ejemplo

```
MPI_Win win;
if (rank == 0) {
 MPI_Win_create(NULL,0,1, MPI_INFO_NULL,MPI_COMM_WORLD,&win);
 MPI_Win_lock(MPI_LOCK_SHARED,1,0,win); // lock del proceso 1
 MPI_Put(buf,1,MPI_INT,1,0,1,MPI_INT,win); // copia datos (de manera
 protegida)

 MPI_Win_unlock(1,win);
 MPI_Win_free(&win);
} else {
 MPI_Win_create(buf,2*sizeof(int),sizeof(int),MPI_INFO_NULL,
 MPI_COMM_WORLD,&win);
}
}
```

# Operaciones colectivas

- Operaciones colectivas no bloqueantes: permiten solapar cómputo y comunicaciones e implementar diversos mecanismos de paralelismo (por ejemplo, software pipelines)

```
MPI_Ibcast(buf, count, type, root, comm, &request);
... // cómputo
MPI_Wait(&request, &status);
```

- Operaciones colectivas para topologías de procesos

```
// crear una topología 3D
MPI_Cart_create(comm, 3, {2,2,2}, {1,1,1}, 1, &newcomm);
// leer datos siguiendo el orden de procesos en newcomm
while(!converged) {
 // comunicaciones entre vecinos
 MPI_Ineighbor_alltoall(..., &newcomm, &req);
 ... // computar internamente en la grilla
 MPI_Wait(&req, MPI_STATUS_IGNORE);
 ... // realizar cálculos externos
}
```

# Procesos dinámicos

- A partir de MPI-2 se proporciona la creación de nuevos procesos a través de la operación colectiva: `MPI_Comm_spawn`
- MPI-2 usa los intercomunicadores (a diferencia de intracomunicadores)
- Los puntos importantes de esta operación son:
  - Es una operación colectiva (invocada por los padres) y también es colectiva en los procesos nuevos `MPI_Init`
  - Retorna un intercomunicador en el cual los procesos padres forman un grupo local y los procesos remotos son los procesos creados
  - Los procesos nuevos tienen su propio `MPI_COMM_WORLD`
  - La función `MPI_Comm_parent`, invocada por los procesos nuevos, retorna un intercomunicador conteniendo a los hijos como grupo local y a los padres como grupo remoto



UNIVERSIDAD  
DE LA REPÚBLICA  
URUGUAY



# Ejercicios



# Ejercicio 0: hello world master-slave

- Código correspondiente a la slide 76
- Utilizando send y receive, en sus diferentes versiones
- Un proceso (maestro) recibe mensajes de otros procesos (esclavos) y es el encargado de imprimir los mensajes de forma centralizada
- Envío y recepción bloqueante y no bloqueante, impresión sincrónica y asincrónica de mensajes recibidos.



# Ejercicio 1: envío y procesamiento

- Código correspondiente a la slide 74
1. Modificar el programa que realiza el envío de un vector para que el proceso P1 devuelva a P0 la suma de los elementos del vector recibido.
    - P0 deberá imprimir el resultado final
  2. Modificar el programa para que un proceso (maestro) envíe a k procesos esclavos



# Ejercicio 2: operaciones colectivas

- Utilizar operaciones colectivas para calcular cuántos números primos hay entre 1 y un determinado número tope  $N$
- Analizar el desempeño y la escalabilidad para diferentes valores de  $N$  y diferente número de procesos en ejecución

# Ejercicio 3: sistema de I/O master-slave

- Sistema de I/O que permita
  1. ordenar entradas y salidas de programas, incluyendo:
    - Salida ordenada (proceso 2 luego de proceso 1)
    - Eliminar duplicados (imprimir un solo "Hello world" en lugar de uno por proceso)
  2. indicar entradas a todos los procesos desde una única terminal
- Idea: separar procesos en MPI\_COMM\_WORLD en clases (masters, encargados de I/O, y slaves, que realizan cálculos [por ejemplo, Jacobi] y hacen su I/O contactando al master), cada clase con su comunicador
- El master debe aceptar mensajes de los slave (tipo MPI\_CHAR, largo máximo 256) e imprimirlos en orden. Al menos dos mensajes deben ser enviados por los esclavos ("hello" y "goodbye", con su rank correspondiente)
- Sugerencias: usar las rutinas MPI\_Comm\_split, MPI\_Send y MPI\_Recv

# Ejercicio 4: broadcasting y procesamiento

- Implementar una aplicación que utilice un proceso maestro que lee desde un archivo de texto y comunica a todos los procesos en ejecución el número de datos leídos y los datos en sí
- El archivo de texto de entrada cuenta con un número entero entre 0 y 9 por línea
- Los procesos se encargan de contar cuantas ocurrencias de su rango hay en el archivo de entrada, y reportan los datos al proceso con rango 0
- El proceso de rango 0 imprime las estadísticas al final



# Ejercicio 5: el pequeño hacker

- Se desea descifrar un texto secreto. Se conoce el valor del texto cifrado, el largo máximo del texto secreto, y el método que se utilizó para cifrarlo. Un ejemplo de código C idéntico al utilizado en el cifrado de la clave secreta.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <crypt.h>
int main(void){
 char *p = crypt("texto secreto", "13");
 printf("%s\n", p);
}
```

- Sabiendo que el texto secreto no tiene más de 6 caracteres alfabéticos en minúscula [a-z], el salt del método crypt() utilizado durante el cifrado fue 13, y que el cifrado del texto secreto es 13rfeUmpQl2s6, implemente un algoritmo paralelo que utilice la fuerza bruta para descubrir el texto utilizando MPI.
- Para compilar el ejemplo es necesario agregar -lcrypt como argumento del comando gcc.

# MPI: bibliografía

- Peter S. Pacheco. 1996. Parallel Programming with MPI. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. 1998. Mpi-The Complete Reference, Volume 1: The MPI Core (2nd. (Revised) ed.). MIT Press, Cambridge, MA, USA.
- William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. Using MPI (2nd Ed.): Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA, USA.