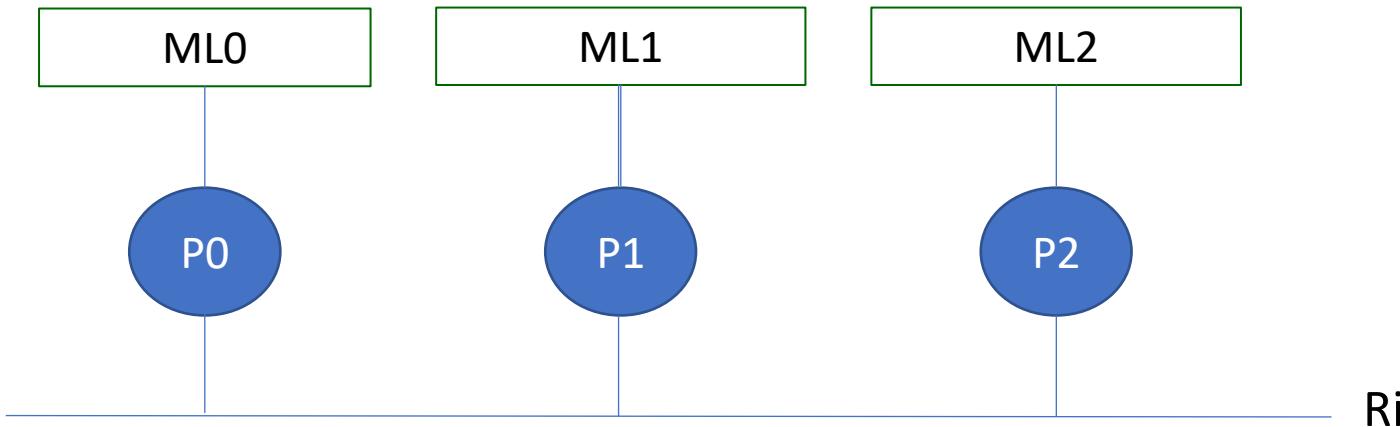


Transparencias MPI

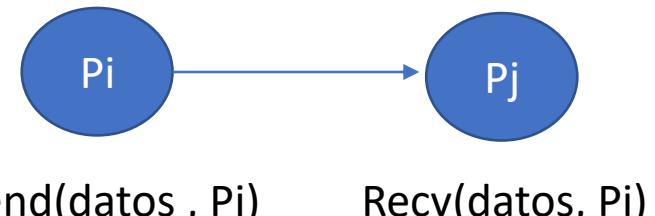
T3-S3

Modelo de Arquitectura Paso de Mensajes

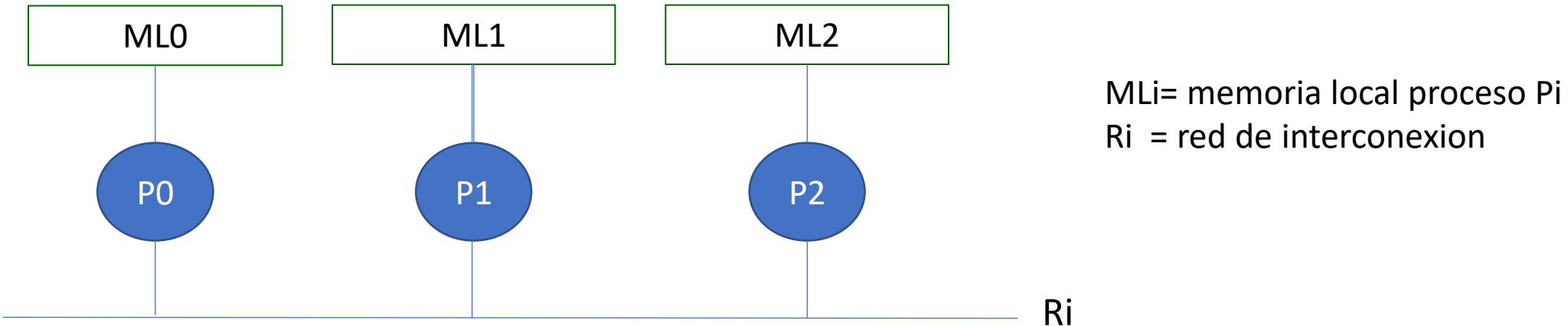


M_{Li}= memoria local proceso P_i
R_i = red de interconexión

- Las memorias M_{Li} tienen el mismo programa
- Cada memoria local M_{Li} tiene los datos locales del proceso P_i
- Para intercambiar los datos entre 2 procesos se usa la red de interconexión para el envío y recepción de datos contenidos en los mensajes
- Para que no hayan bloqueos tienen que estar emparejados el envío (Send) y la recepción (Recv)
- Se trata de una programación laboriosa pues hay que controlar todos los detalles de la paralización



Modelo de Arquitectura Paso de Mensajes



- P_i son procesos más pesados que los hilos; similares a los procesos del sistema operativo.
- Normalmente tendremos un proceso por procesador
- Cada proceso tiene un identificador que lo diferencia del resto
- Creación de procesos:
 - Estática: al inicio del programa
 - En línea de comandos (mpiexec)
 - Existen durante toda la ejecución
 - Es lo más habitual
 - Dinámica: durante la ejecución
 - Primitiva spawn()

Modelo de Paso de Mensajes

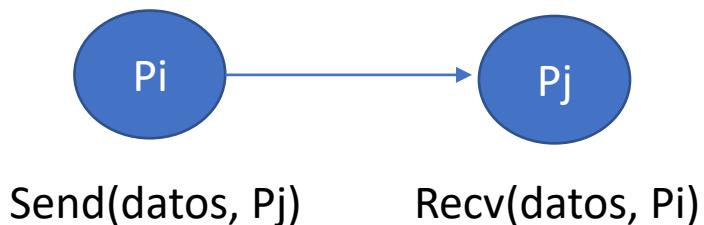
- Intercambio explícito de información mediante envío y recepción de mensajes
- Modelo más usado en computación a gran escala
- Está basada en bibliotecas de funciones (aprendizaje más fácil que un lenguaje nuevo)

Ventajas:

Universalidad
Fácil comprensión
Gran expresividad
Mayor eficiencia

Inconvenientes:

Programación compleja
Control total de las comunicaciones



MPI(Multi Point Interface)

- MPI es una especificación propuesta por un comité de investigadores, usuarios y empresas

<https://www mpi-forum.org>

- Especificaciones:

- MPI-1.0 (1994), última actualización MPI-1.3 (2008)
- MPI-2.0 (1997), última actualización MPI-2.2 (2009)
- MPI-3.0 (2012), última actualización MPI-3.1 (2015)
- MPI-4.0 (2020), última actualización MPI-4.1 (2021)

- Antecedentes:

- Cada fabricante ofrecía su propio entorno (migración costosa)
- PVM (Parallel Virtual Machine) fue un primer intento de estandarización

Características de MPI

- Características principales:
 - Es portable a cualquier plataforma paralela
 - Es simple (con tan sólo 6 funciones se puede implementar cualquier programa)
 - Es potente (más de 300 funciones)
- El estándar especifica interfaz para C y Fortran
- Hay muchas implementaciones disponibles:
 - Propietarias: IBM, Cray, SGI, ...
 - MPICH (www.mpich.org)
 - Open MPI (www.open-mpi.org)
 - MVAPICH (mvapich.cse.ohio-state.edu)

Modelo de programación en MPI

- MPI requiere una inicialización (**MPI_Init**) y una finalización (**MPI_Finalize**)
- Los argumentos **argc** y **argv** utilizados en **MPI_Init** son los mismos que los de la función **main**
- Ejemplo: ejemplo.c

```
#include <mpi.h>

int main(int argc, char* argv[]) {
    .....
    MPI_Init(&argc, &argv);
    .....
    MPI_Finalize();
    return 0;
}
```

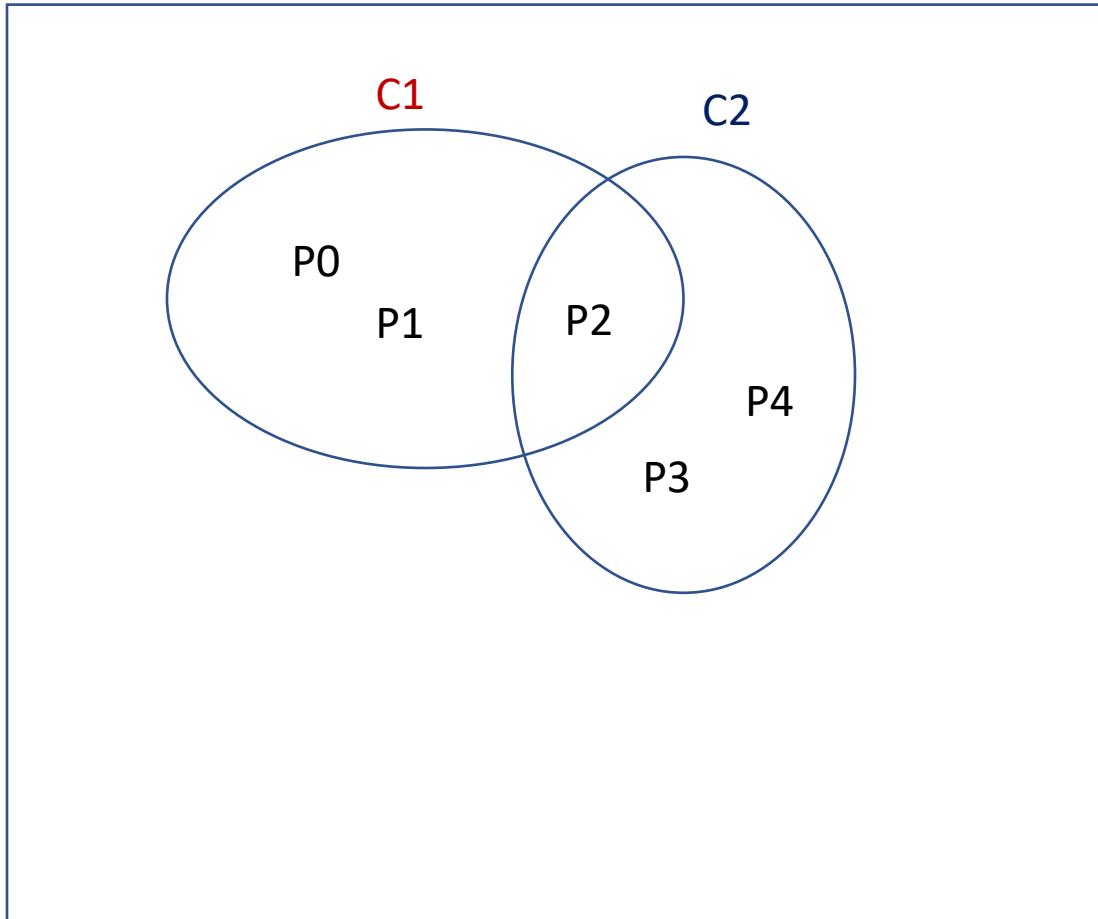
- Compilación: mpicc –o ejemplo ejemplo.c

Modelo de programación en MPI - Operaciones

- Las operaciones se pueden agrupar en:
 - Comunicación punto a punto: Intercambio de información entre dos procesos
 - Comunicación colectiva: Intercambio de información en un conjunto de procesos
 - Gestión de datos: Tipos de datos derivados
 - Operaciones de alto nivel: Grupos, comunicadores, atributos, topologías
 - Operaciones avanzadas (MPI-2, MPI-3, MPI-4): Entrada-salida, creación de procesos, comunicación unilateral
 - Utilidades: Interacción con el entorno del sistema
- La mayoría de las operaciones se realizan sobre comunicadores

Modelo de programación en MPI - Operaciones

Comunicador = *Grupo de procesos + Contexto*



MPI_COMM_WORLD

- Siempre está definido el comunicador universal (MPI_COMM_WORLD), el cual está formado por todos los procesos que se han lanzado
- Se puede definir diferentes comunicadores
- En toda operación de comunicación se debe indicar el comunicador usado: la operación sólo afecta a los procesos del comunicador
- **`MPI_Comm_rank()`**: Permite conocer el identificador del proceso dentro de un comunicador
- **`MPI_Comm_size()`**: Permite conocer el número de procesos en un comunicador

Modelo de programación en MPI

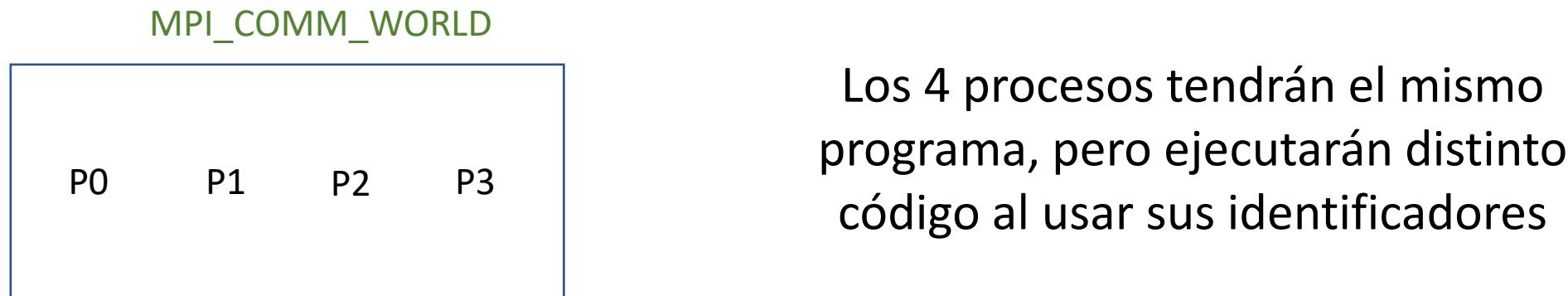
Ejemplo: hola_mpi.c

```
#include<stdio.h>
#include <mpi.h>
int main(int argc, char* argv[]) {
    int id; /* identificador del proceso */
    int p; /* número de procesos */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    printf("Hola mundo, soy el proceso %d de %d\n", id, p);
    MPI_Finalize();
    return 0;
}
```

- Compilación: mpicc –o hola_mpi hola_mpi.c

Modelo de ejecución en MPI

- `mpiexec -n p programa`: Se lanzan **p** procesos para ejecutar el código
- Ejemplo: `mpiexec -n 4 hola_mpi`



Local host: MSI

```
-----  
Hola mundo, soy el proceso 0 of 4 procesos  
Hola mundo, soy el proceso 1 of 4 procesos  
Hola mundo, soy el proceso 2 of 4 procesos  
Hola mundo, soy el proceso 3 of 4 procesos
```

Comunicaciones punto a punto en MPI

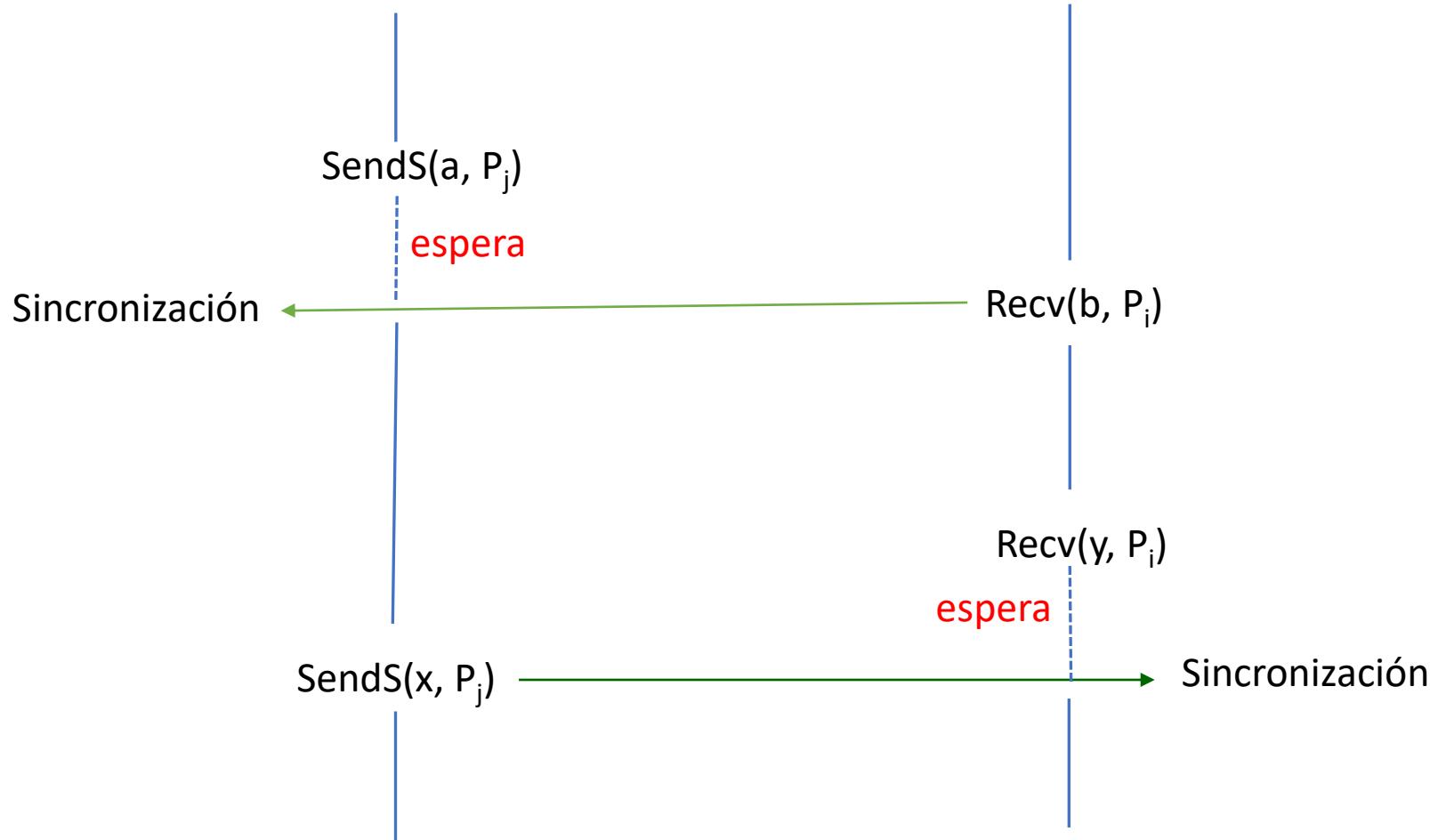
- Tipos de envíos
 - Bloqueante/No bloqueante
 - Síncrono
 - Con Buffer
 - Estándar
- Tipos de recepciones
 - Estándar
 - No bloqueante

Envíos/recepciones bloqueantes

- Al finalizar la llamada a un Send bloqueante es seguro modificar la variable que se envía
- Al finalizar la llamada a un Recv bloqueante se garantiza que la variable contiene el mensaje
- Las no bloqueantes simplemente inician la operación

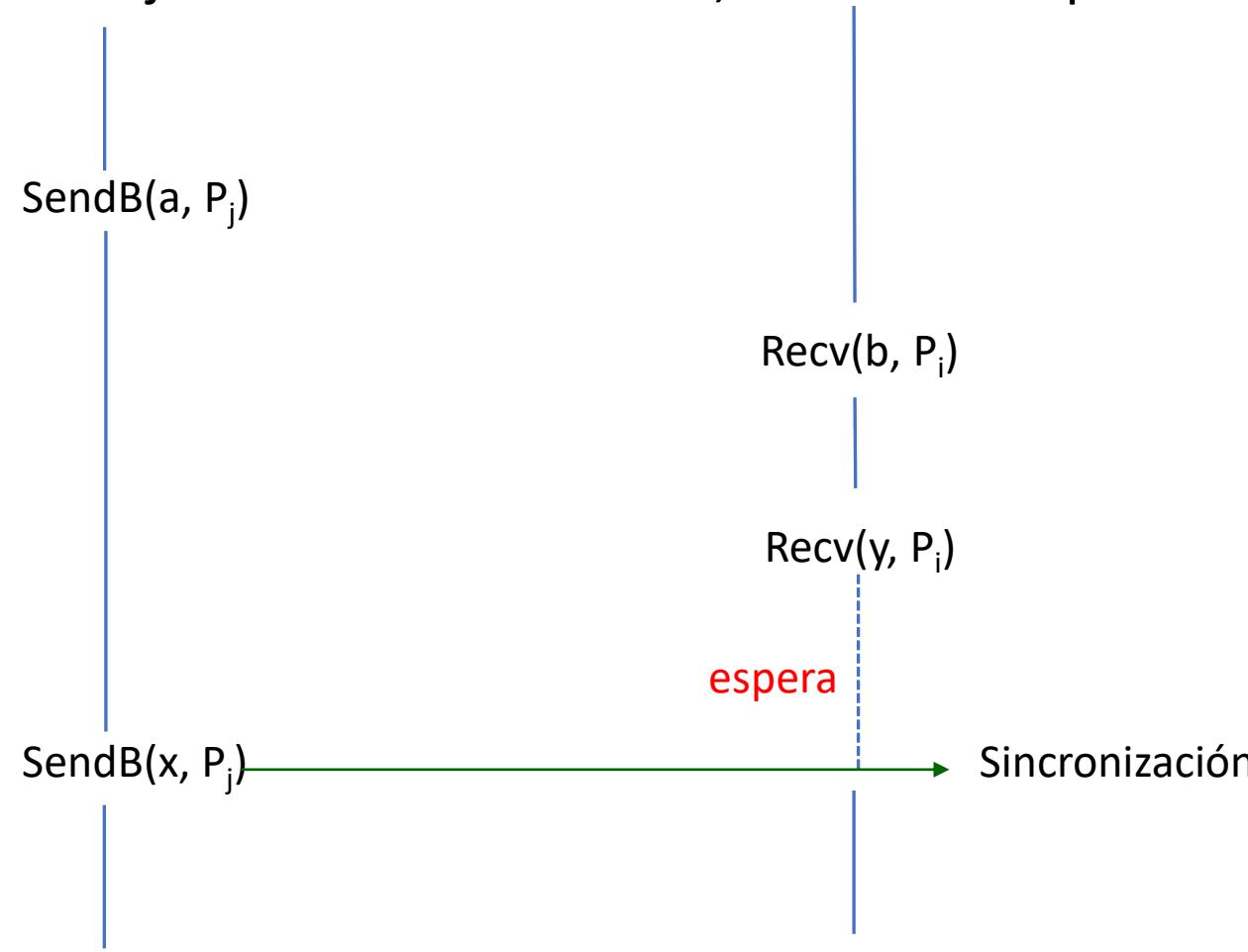
Envío síncrono

- La operación **Send** no acaba hasta que el otro proceso ha efectuado la correspondiente recepción **Recv**



Envío con Buffer

- Al hacer un proceso un **sendB** el mensaje se copia en un buffer y el proceso continua
- Cuando el receptor hace un Recv el mensaje enviado es el que contiene el buffer
- **Problema:** Si el mensaje no cabe en el buffer, entonces se produce un error



Problemas de interbloqueo

- Un mal uso de Send y Recv puede producir interbloqueo
- Caso de comunicación síncrona:

```
/* Proceso 0 */          /* Proceso 1 */  
Send(x,1);              Send(y,0);  
Recv(y,1);              Recv(x,0);
```

Ambos quedan bloqueados en el envío

- Caso de envío con buffer:

El ejemplo anterior no causaría interbloqueo

Puede haber otras situaciones con interbloqueo

```
/* Proceso 0 */          /* Proceso 1 */  
Recv(y,1);              Recv(x,0);  
SendB(x,1);             SendB(y,0);
```

Possible solución: intercambiar el orden de uno de ellos:

```
/* Proceso 0 */          /* Proceso 1 */  
Send(x,1);              Recv(x,0);  
Recv(y,1);              Send(y,0);
```

Envíos en MPI

`MPI_Ssend(síncrono), MPI_Bsend(con buffer), MPI_Send(estándar)`

- Todos tienen los mismos argumentos

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

- Con pequeños mensajes se comporta como `MPI_Bsend`
- Con grandes mensajes se comporta como `MPI_Ssend`

buf: puntero a la variable que contiene el mensaje (dato simple con &, array sin &)

count: número de datos

datatype: tipo de dato (`MPI_INT`, `MPI_DOUBLE`,...)

dest: identificador del proceso que recibirá el mensaje

tag: número entero positivo correspondiente a la etiqueta del mensaje

comm: comunicador. Casi siempre usaremos el comunicador `MPI_COMM_World`

Envío no bloqueante en MPI

`MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`

request: puntero a la variable que almacena la petición no bloqueante

Antes de modificar la variable que contiene al mensaje, es necesario asegurarse que se puede hacer: o bien esperar hasta que se realice la recepción o bien comprobar que se ha realizado la recepción del mensaje:

- Espera:

`MPI_Wait(MPI_Request *request, MPI_Status *status)`

- Comprobación:

`MPI_Test(MPI Request *request, int *flag, MPI_Status *status)`

- **flag:** 0 si no se ha producido la petición de recepción del mensaje, 1 si se ha producido
- **status:** puntero al estado de la comunicación

Recepción estándar en MPI

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

buf: puntero a la variable en donde se almacena el mensaje (dato simple con &, array sin &)

count: número de datos

datatype: tipo de dato (**MPI_INT**, **MPI_DOUBLE**,...)

source: identificador del proceso que recibirá el mensaje

- Si no se quiere especificar, se pone **MPI_ANY_SOURCE**

tag: número entero positivo correspondiente a la etiqueta del mensaje

- Si no se quiere especificar, se pone **MPI_ANY_TAG**

comm: comunicador. Casi siempre usaremos el comunicador **MPI_COMM_WORLD**

status: puntero a una estructura (estado del mensaje)

- Si no se desea conocer el estado del mensaje, se pone **MPI_STATUS_IGNORE**
- stat.MPI_SOURCE: identificador del proceso que realiza el envío
- stat.MPI_TAG: etiqueta del mensaje recibido
- La longitud del mensaje se determina mediante la función:

MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

Recepción no bloqueante en MPI

MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

request: puntero a la variable que almacena la petición no bloqueante

Es necesario asegurarse antes de modificar la variable que va a contener el mensaje esperar o comprobar que se ha realizado el envío del mensaje:

- Espera:

`MPI_Wait(MPI_Request *request, MPI_Status *status)`

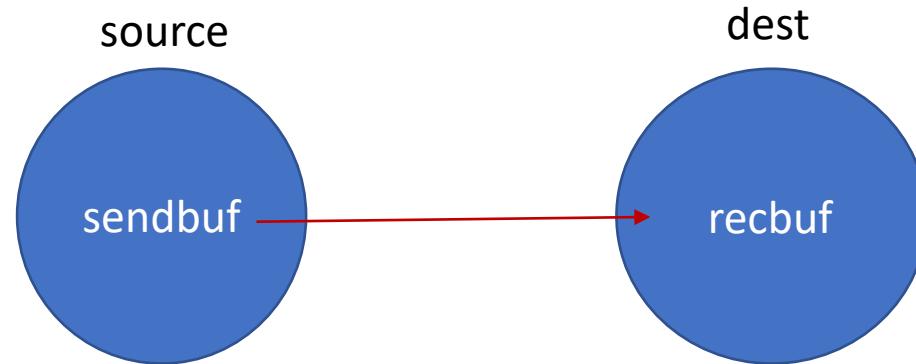
- Comprobación:

`MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`

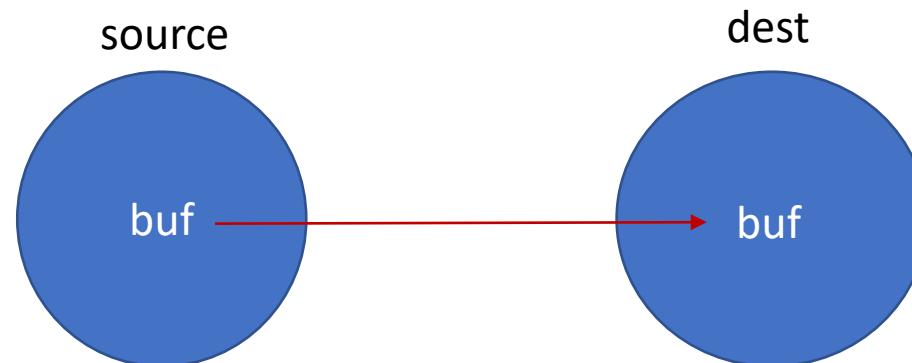
- **flag:** 0 si no se ha producido la petición de recepción del mensaje, 1 si se ha producido
- **status:** puntero al estado de la comunicación

Operaciones combinadas en MPI

```
MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag,  
            void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag,  
            MPI_Comm comm, MPI_Status *status)
```



```
MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype,  
                     int dest, int sendtag,  
                     int source, int recvtag,  
                     MPI_Comm comm, MPI_Status *status)
```



Ejercicio 1

- Completa el siguiente programa de MPI que permita enviar desde P0 a P1 el valor de una variable entera a, usando para ello las funciones MPI_send y MPI_recv.

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char **argv){
int a;
MPI_Init(&argc, &argv);
int id; /*Identificador del proceso y número de procesos*/
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Status status;
if (id == 0){
    a=5;
    MPI_Send(&a, 1, MPI_INT, 1, 100, MPI_COMM_WORLD);
}
else if (id==1)
    MPI_Recv(&a, 1, MPI_INT, 0, 100, MPI_COMM_WORLD, &status);
MPI_Finalize();
return 0;
}
```

&a(dato simple) v(array)

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

Receive a message from one process:

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

Elementary Datatypes:

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

Reserved Communicators:

MPI_COMM_WORLD, MPI_COMM_SELF

Ejercicio 1

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

int main (int argc, char **argv){
    int a;
    MPI_Init(&argc, &argv);
    int id, np;
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    if (id == 0){
        a=5;
        MPI_Send(&a,1, MPI_INT, 1, 100, MPI_COMM_WORLD);
    }
    else if (id==1)
        MPI_Recv(&a, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Finalize();
    return 0;
}
```

Alternativamente en MPI_Recv, si nos da igual conocer el origen, la etiqueta y el estado de la recepción del mensaje, podemos usar:

- MPI_ANY_SOURCE en lugar de **0** (source)
- MPI_ANY_TAG en lugar de **100** (tag)
- MPI_STATUS_IGNORE en lugar de **&status**

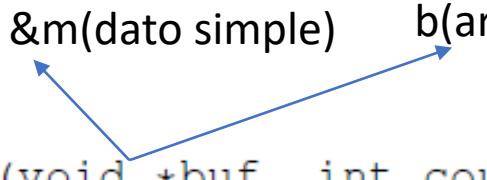
```
int MPI_Send(void *buf, int count,
            MPI_Datatype datatype, int dest, int
            tag, MPI_Comm comm)

Receive a message from one process:
int MPI_Recv(void *buf, int count,
            MPI_Datatype datatype, int source, int
            tag, MPI_Comm comm, MPI_Status *status)
```

Ejercicio 2

Completa el siguiente programa de MPI, de manera que en primer lugar P0 lea un entero m y un vector b mediante la función **void lee**, y a continuación P0 envíe esos datos al proceso P1:

```
#include <mpi.h>
#include <stdio.h>
#define N 1000
void lee(int *m, double b[N]);
int main (int argc, char **argv){
    int m;
    double b[N];
    MPI_Init(&argc, &argv);
    int id; /*identificador de proceso*/
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    if (id == 0){
        leer(&m, b);/*P0 lee m y b*/
        MPI_Send(&m, 1, MPI_INT, 1, 100, MPI_COMM_WORLD);
        MPI_Send(b, N, MPI_DOUBLE, 1, 200, MPI_COMM_WORLD);
    }
    else if (id==1){
        MPI_Recv(&m, 1, MPI_INT, 0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(b, m, MPI_DOUBLE, 0, 200, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    MPI_Finalize();
    return 0;
}
```


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

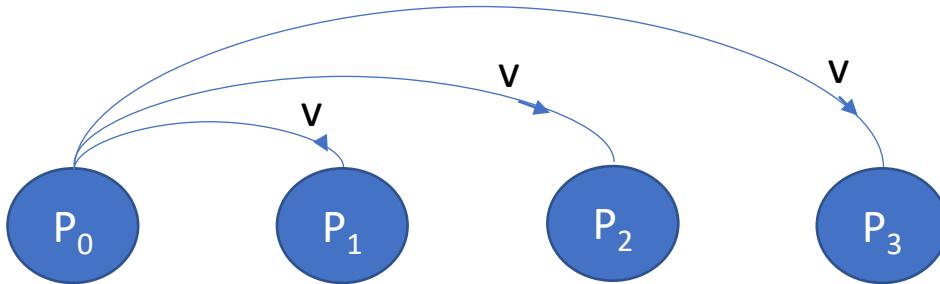
Receive a message from one process:

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

Elementary Datatypes:

MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG,
MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT,
MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_FLOAT,
MPI_DOUBLE, MPI_LONGDOUBLE, MPI_BYTE,
MPI_PACKED

Ejemplo: difusión de un dato



Enviar un dato almacenado en la variable **v** desde el proceso P0 al resto de procesos:

```
double v;  
int p, rank, i;  
MPI_Comm_size(MPI_COMM_WORLD, &p);  
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
if (rank == 0) {  
    read_value(&v); /* valor a difundir */  
    for (i=1; i<p; i++)  
        MPI_Send(&v, 1, MPI_DOUBLE, i, 100,MPI_COMM_WORLD);  
}  
else  
    MPI_Recv(&v, 1, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Ejercicio 3

Completa el siguiente programa de MPI de manera que imprima en pantalla la suma de los n primeros números naturales (n es obtenido al ejecutar el programa). Para ello, usaremos un reparto cíclico de los n números, de manera que cada proceso sume su parte. El proceso P0 recibirá de cada Pi la suma calculada por Pi, calculando el total; los demás procesos enviarán a P0 su suma calculada. Además, P0 deberá imprimir el proceso que le envía la suma y la suma.

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

#include <mpi.h>

int main(int argc,char *argv[]){
    int n, i;
    if (argc==2) n = atoi(argv[1]);
    else n = 8;
    .....
    return 0;
}
```

Ejemplo: Sumar $1+2+3+4+5+6+7+8+9+10+11+12+\dots+24$ ($n=24$)

Cada proceso realiza una suma parcial (reparto cíclico):

- P0: $sl=1+ 5+ 9+ 13+ 17+ 21= 66$
- P1: $sl=2+ 6+ 10+ 14+ 18+ 22= 72$
- P2: $sl=3+ 7+ 11+ 15+ 19+ 23= 78$
- P3: $sl=4+ 8+ 12+ 16+ 20+ 24= 84$

Después:

- P1, P2 y P3 envían su suma parcial a P0
- P0 recoge los valores enviados por P1, P2 y P3 y se los suma a su suma parcial

Ejercicio 3

```
MPI_Init(&argc, &argv);
int id, p;
MPI_Status stat;
MPI_Comm_size(MPI_COMM_WORLD,&p);
MPI_Comm_rank(MPI_COMM_WORLD,&id);
int sl=0, s;
for (i = id + 1; i <= n; i += p) sl += i; ←
if (id==0){
    for (i=1; i<p; i++){
        MPI_Recv(&s, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &stat);
        printf("P%d ha enviado una suma igual a %d\n",stat.SOURCE, s);
        sl+=s;
    }
    printf("Suma total = %d\n", sl);
}
else
    MPI_Send(&sl, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
MPI_Finalize();
```

P0: $sl=1+ 5+ 9+ 13+ 17+ 21= 66$
P1: $sl=2+ 6+ 10+ 14+ 18+ 22= 72$
P2: $sl=3+ 7+ 11+ 15+ 19+ 23= 78$
P3: $sl=4+ 8+ 12+ 16+ 20+ 24= 84$

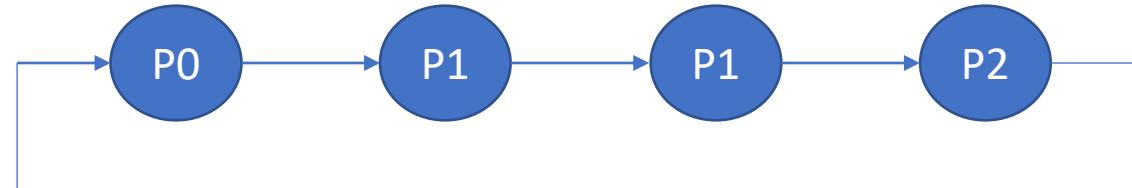
```
int MPI_Send(void *buf, int count,
            MPI_Datatype datatype, int dest, int
            tag, MPI_Comm comm)

Receive a message from one process:
int MPI_Recv(void *buf, int count,
            MPI_Datatype datatype, int source, int
            tag, MPI_Comm comm, MPI_Status *status)
```

Ejercicio 4

El siguiente fragmento de código es incorrecto (puede dar lugar a bloqueos). Indica por qué y propón tres soluciones.

```
int sbuf[N], rbuf[N], rank, size, src, dst;  
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
MPI_Comm_size(MPI_COMM_WORLD, &size);  
src = (rank==0)? size-1: rank-1;  
dst = (rank==size-1)? 0: rank+1;  
MPI_Ssend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD);  
MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



```
int sbuf[N], rbuf[N], rank, size, src, dst;
```

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

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

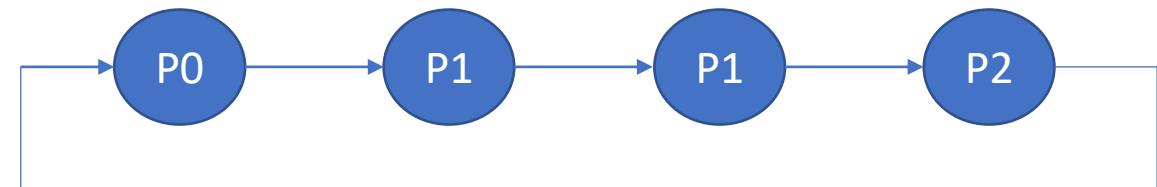
```
src = (rank==0)? size-1: rank-1;
```

```
dst = (rank==size-1)? 0: rank+1;
```

```
MPI_Ssend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD);
```

```
MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Ejercicio 4



Solución protocolo pares/impares:

```
if (rank%2==0) {  
    MPI_Ssend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD);  
    MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
}  
else {  
    MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
    MPI_Ssend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD);  
}
```

```
int sbuf[N], rbuf[N], rank, size, src, dst;
```

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

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

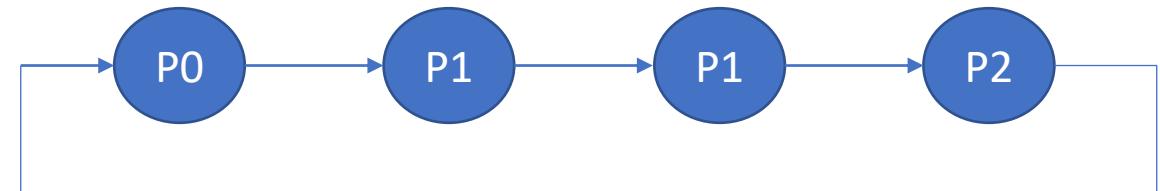
```
src = (rank==0)? size-1: rank-1;
```

```
dst = (rank==size-1)? 0: rank+1;
```

```
MPI_Ssend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD);
```

```
MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Ejercicio 4



Solución operación combinada:

Combined send and receive:

```
int MPI_Sendrecv(void *sendbuf, int  
    sendcount, MPI_Datatype sendtype, int  
    dest, int sendtag, void *recvbuf, int  
    recvcount, MPI_Datatype recvtype, int  
    source, int recvtag, MPI_Comm comm,  
    MPI_Status *status)
```

```
MPI_Sendrecv(sbuf, N, MPI_INT, dst, 111, rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Ejercicio 4

```
int sbuf[N], rbuf[N], rank, size, src, dst;  
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
MPI_Comm_size(MPI_COMM_WORLD, &size);  
src = (rank==0)? size-1: rank-1;  
dst = (rank==size-1)? 0: rank+1;  
MPI_Ssend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD);  
MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Non-Blocking Point-to-Point

Begins a non-blocking send:

```
int MPI_Isend(void *buf, int count,  
              MPI_Datatype dtype, int dest, int tag,  
              MPI_Comm comm, MPI_Request *request)
```

Begin to receive a message:

```
int MPI_Irecv(void *buf, int count,  
              MPI_Datatype dtype, int src, int tag,  
              MPI_Comm comm, MPI_Request *request)
```

Complete a non-blocking operation:

```
int MPI_Wait(MPI_Request *request,  
             MPI_Status *status)
```

Check or complete a non-blocking operation:

```
int MPI_Test(MPI_Request *request, int  
            *flag, MPI_Status *status)
```

Solución operaciones no bloqueantes:

```
MPI_Isend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD, &req);  
MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Ejercicio 5

- Se quiere parallelizar el siguiente código mediante MPI. Suponemos que se disponemos de 3 procesos.

```
double a[N], b[N], c[N], v=0.0, w=0.0;
```

```
T1(a,&v);
```

```
T2(b,&w);
```

```
T3(b,&v);
```

```
T4(c,&w);
```

```
T5(c,&v);
```

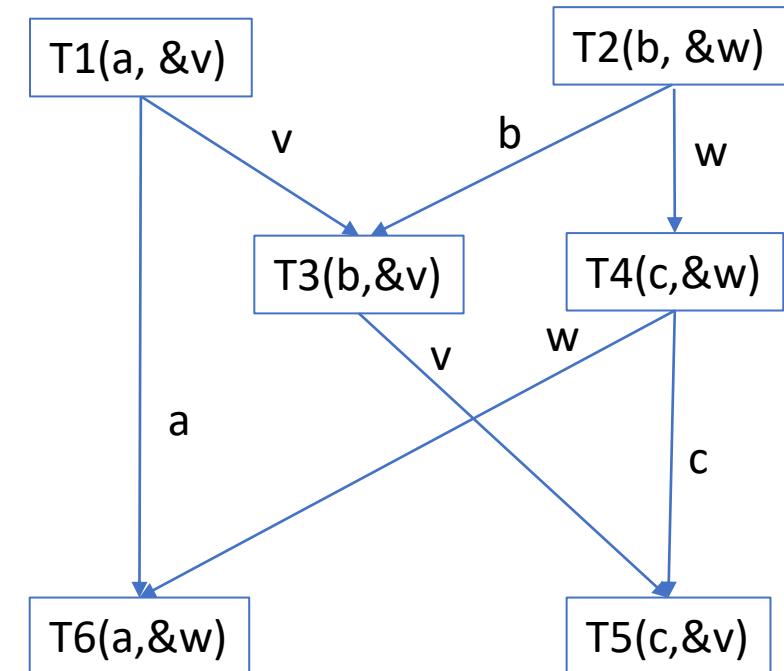
```
T6(a,&w);
```

Todas las funciones leen y modifican sus dos argumentos, incluyendo los vectores. Suponemos que los vectores a, b y c están almacenados en P0, P1 y P2, respectivamente, y son demasiado grandes para poder ser enviados eficientemente de un proceso a otro.

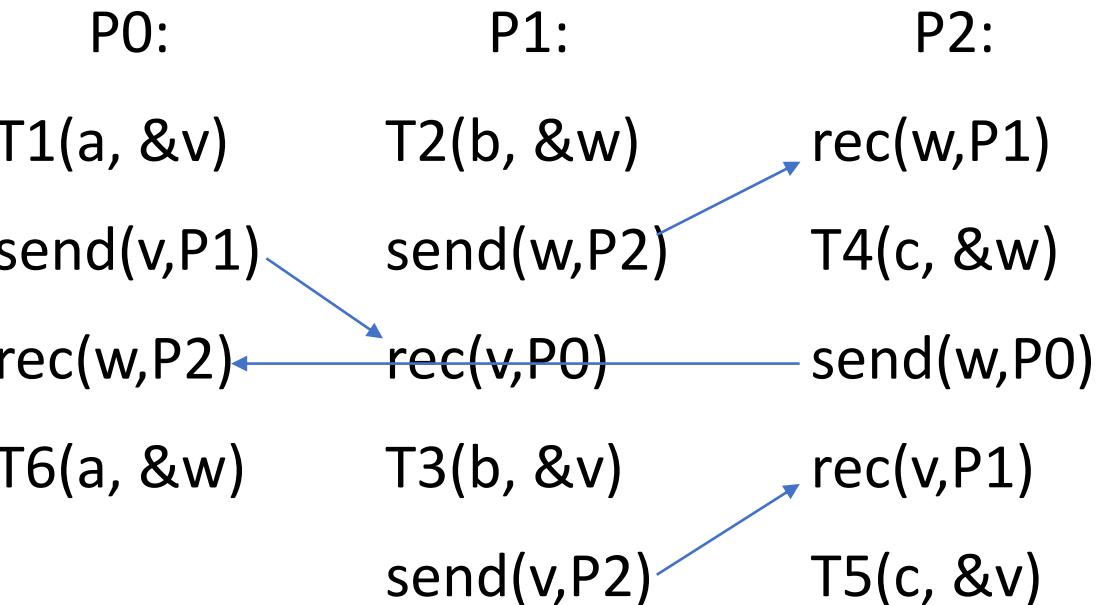
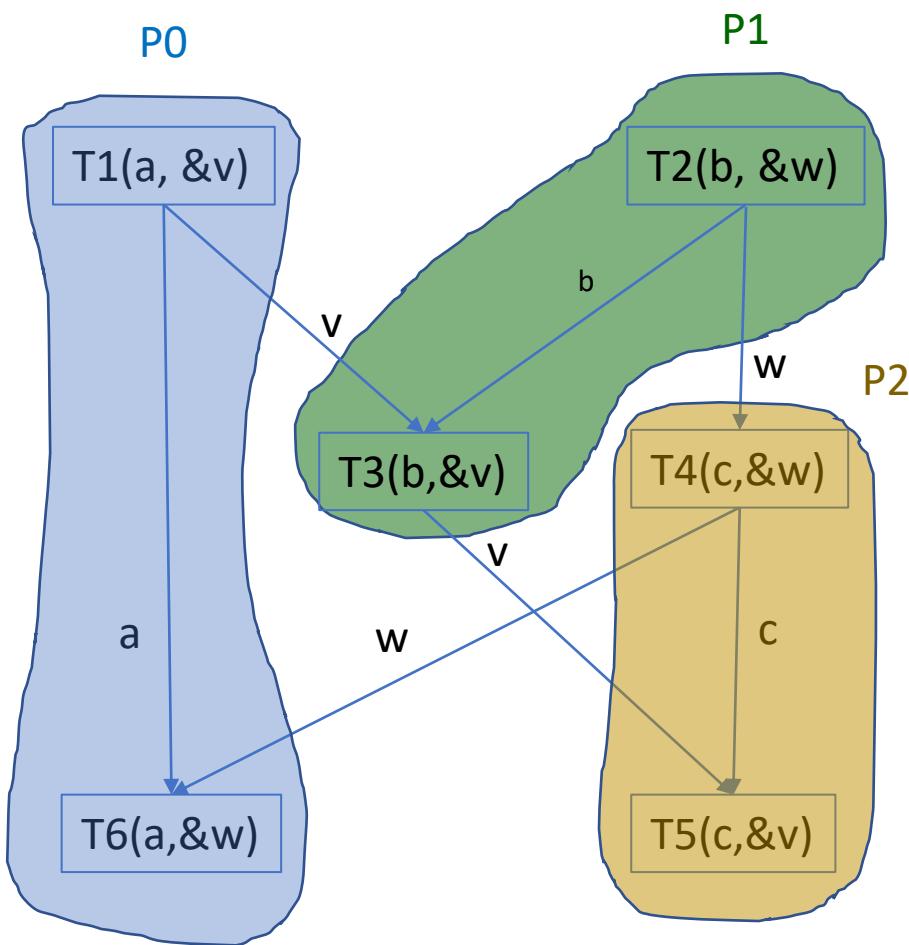
- (a) Dibuja el grafo de dependencias de las diferentes tareas, indicando qué tarea se asigna a cada proceso.

Ejercicio 5

```
double a[N],b[N],c[N],v=0.0,w=0.0;  
T1(a,&v);  
T2(b,&w);  
T3(b,&v);  
T4(c,&w);  
T5(c,&v);  
T6(a,&w);
```



Ejercicio 5

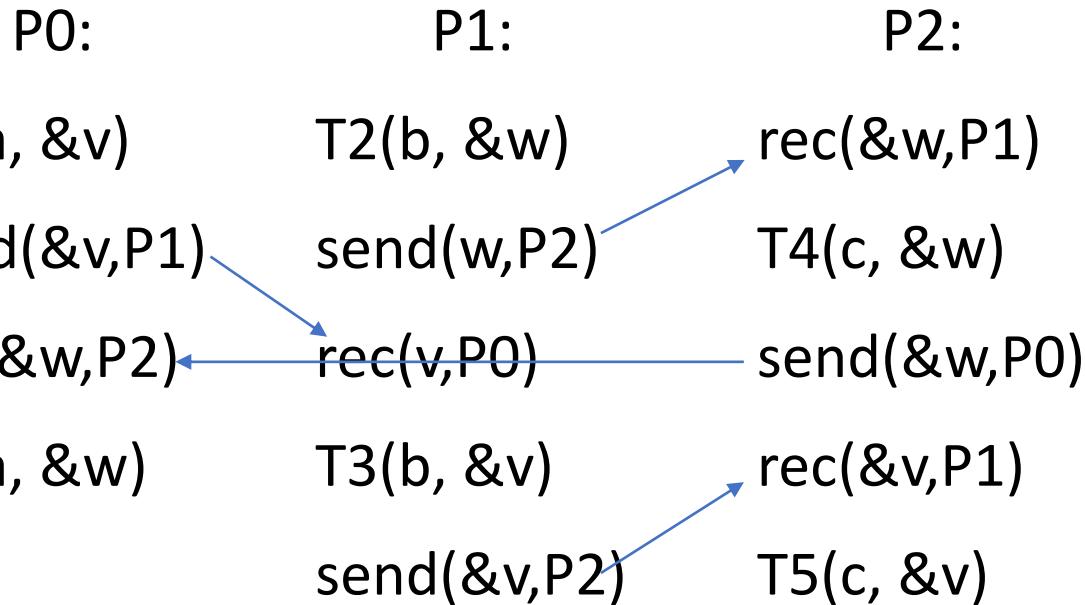


Ejercicio 5

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

Receive a message from one process:

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



```
MPI_Status stat;  
double a[N], b[N], c[N], v=0.0, w=0.0;  
int p, rank;  
MPI_Comm_size(MPI_COMM_WORLD,&p);  
MPI_Comm_rank(MPI_COMM_WORLD,&rank);  
if (rank==0) {  
    T1(a,&v);  
    MPI_Send(&v, 1, MPI_DOUBLE, 1, 111, MPI_COMM_WORLD);  
    MPI_Recv(&w, 1, MPI_DOUBLE, 2, 111, MPI_COMM_WORLD, &stat);  
    T6(a,&w);  
}  
else if (rank==1) {  
    T2(b,&w);  
    MPI_Send(&w, 1, MPI_DOUBLE, 2, 111, MPI_COMM_WORLD);  
    MPI_Recv(&v, 1, MPI_DOUBLE, 0, 111, MPI_COMM_WORLD, &stat);  
    T3(b,&v);  
    MPI_Send(&v, 1, MPI_DOUBLE, 2, 111, MPI_COMM_WORLD);  
}  
else /* rank==2 */  
{  
    MPI_Recv(&w, 1, MPI_DOUBLE, 1, 111, MPI_COMM_WORLD, &stat);  
    T4(c,&w);  
    MPI_Send(&w, 1, MPI_DOUBLE, 0, 111, MPI_COMM_WORLD);  
    MPI_Recv(&v, 1, MPI_DOUBLE, 1, 111, MPI_COMM_WORLD, &stat);  
    T5(c,&v);  
}
```

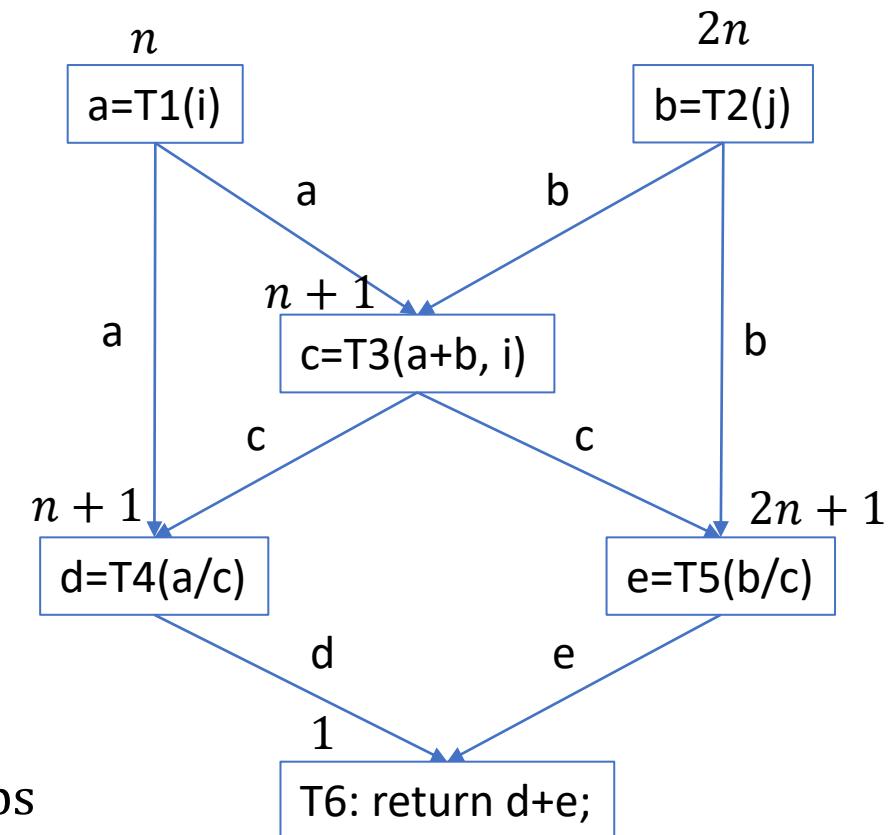
Ejercicio 6

Dada la siguiente función, donde suponemos que las funciones T1, T3 y T4 tienen un coste de n y las funciones T2 y T5 de $2n$, siendo n un valor constante.

```
double ejemplo(int i, int j){  
    double a, b, c, d, e;  
    a = T1(i);  
    b = T2(j);  
    c = T3(a+b, i);  
    d = T4(a/c);  
    e = T5(b/c);  
    return d+e; /* T6 */  
}
```

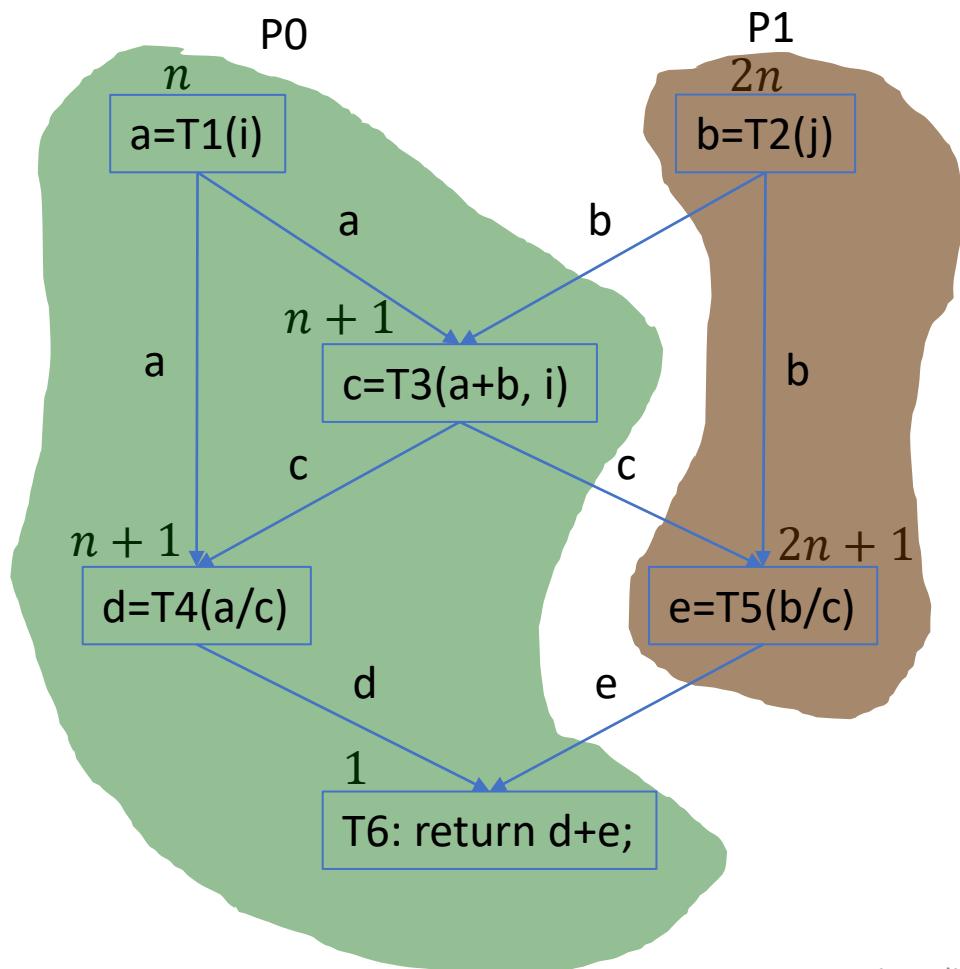
(a) Dibuja el grafo de dependencias y calcula el coste secuencial.

$$t_s = n + 2n + n + 1 + n + 1 + 2n + 1 + 1 = 7n + 4 \text{ flops}$$



Ejercicio 6

Paralelízalo usando MPI dos procesos. Ambos procesos invocan a las funciones con los mismos valores de los argumentos i, j (no es necesario comunicarlos). El valor de retorno de la función debe ser correcto en el proceso 0 (no es necesario que esté en ambos procesos)



P0:

```
a=T1(i)
Recv(b,P1)
c=T3(a+b, i)
Send(c,P1)
d=T4(a/c)
Recv(e,P1)
T6: return d+e;
```

P1:

```
b=T2(j)
Send(b,P0)
Recv(c,P0)
e=T5(b/c)
Send(e,P0)
```

Ejercicio 6

P0:

a=T1(i)

Rec(b,P1) ←

c=T3(a+b, i)

Send(c,P1) → Rec(c,P0)

d=T4(a/c)

Send(e,P0) → Rec(e,P1) ←

T6: return d+e;

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

Receive a message from one process:
int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype, int source, int
             tag, MPI_Comm comm, MPI_Status *status)
```

P1:

b=T2(j)

Send(b,P0) → Rec(c,P0)

e=T5(b/c)

Send(e,P0) →

```
double ejemplo(int i,int j){
double a,b,c,d,e;
int rank;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank==0) {
    a = T1(i);
    MPI_Recv(&b, 1, MPI_DOUBLE, 1, 111, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    c = T3(a+b,i);
    MPI_Send(&c, 1, MPI_DOUBLE, 1, 112, MPI_COMM_WORLD);
    d = T4(a/c);
    MPI_Recv(&e, 1, MPI_DOUBLE, 1, 113, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    return d+e;
}
else {
    b = T2(j);
    MPI_Send(&b, 1, MPI_DOUBLE, 0, 111, MPI_COMM_WORLD);
    MPI_Recv(&c, 1, MPI_DOUBLE, 0, 112, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    e = T5(b/c);
    MPI_Send(&e, 1, MPI_DOUBLE, 0, 113, MPI_COMM_WORLD);
}
}
```

Ejercicio 6

c) Calcula el tiempo de ejecución paralelo (cálculo y comunicaciones) y el speedup con dos procesos

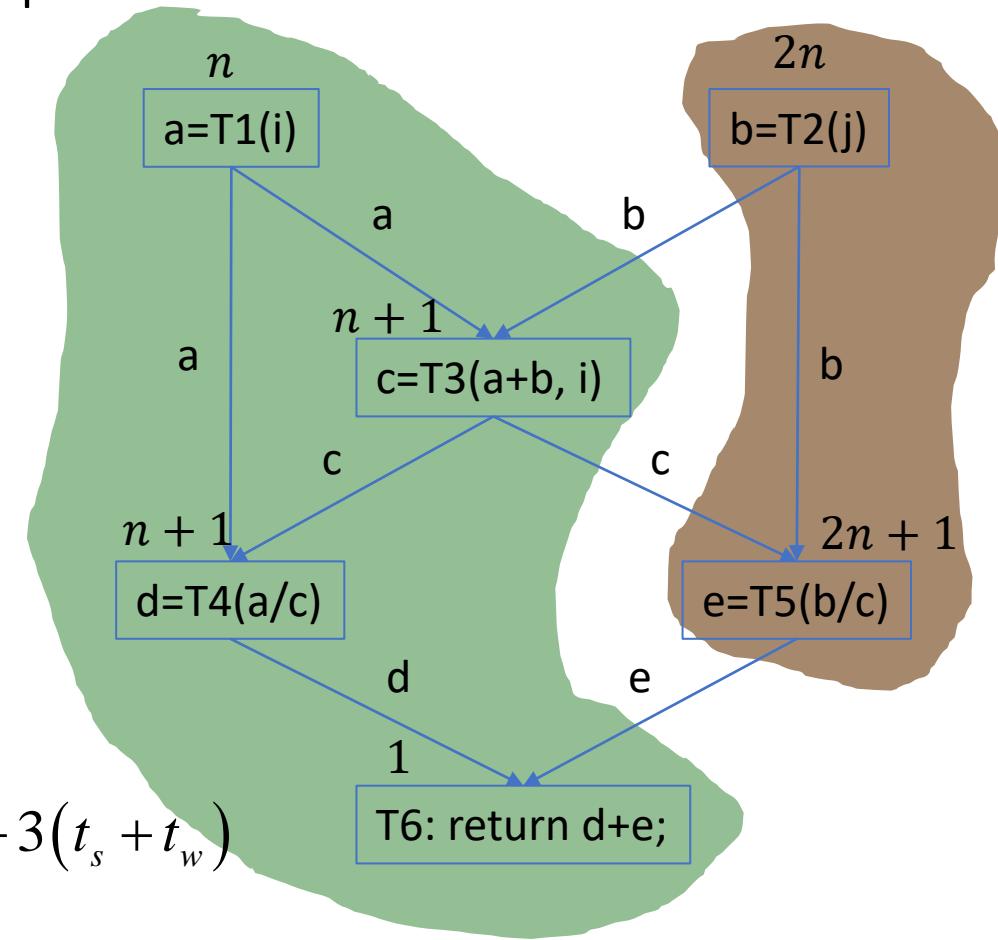
El tiempo paralelo de la implementación propuesta se debe calcular a partir del coste asociado al camino crítico del grafo de dependencias, correspondiente a T2 – T3 – T5 – T6

P0:

```
a=T1(i); /*n flops*/
Rec(b,P1);
c=T3(a+b, i) ;/*n+1 flops*/
Send(c,P1)
d=T4(a/c); /*n+1 flops*/
Rec(e,P1)
T6: return d+e; *1 flop*/
```

P1:

```
b=T2(j) /*2n flops*/
Send(b,P0)
Rec(c,P0)
e=T5(b/c) ;/*2n+1 flops*/
Send(e,P0)
```



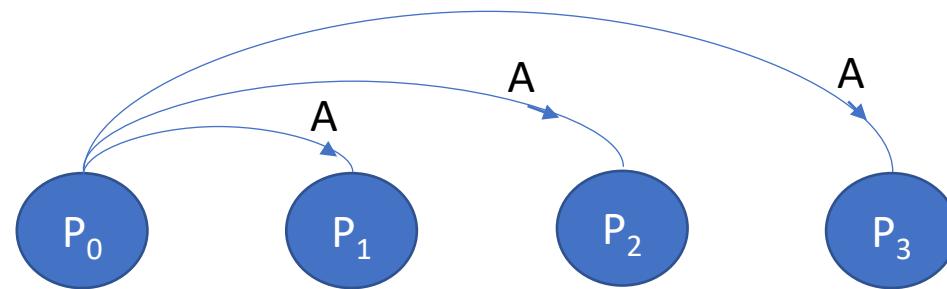
$$\left. \begin{aligned} t_a(n, 2) &= 2n + n + 1 + 2n + 1 + 1 = 5n + 3 \text{ flops} \\ t_c(n, 2) &= 3(t_s + t_w) \end{aligned} \right\} t(n, 2) \approx 5n \text{ flops} + 3(t_s + t_w)$$

$$S(n, 2) = \frac{t(n)}{t(n, 2)} \approx \frac{7n}{5n + 3(t_s + t_w)}$$

Ejercicio 7

Supongamos que en MPI se dispone de p procesos y que el procesador P_0 contiene una matriz A de dimensión $M \times N$ de tipo double que debe ser difundida al resto de procesos ¿Qué función de comunicaciones colectivas deberían ejecutar todos los procesos ¿Cómo podrías sustituir esa comunicación colectiva mediante comunicaciones punto a punto? Calcula el tiempo de comunicaciones.

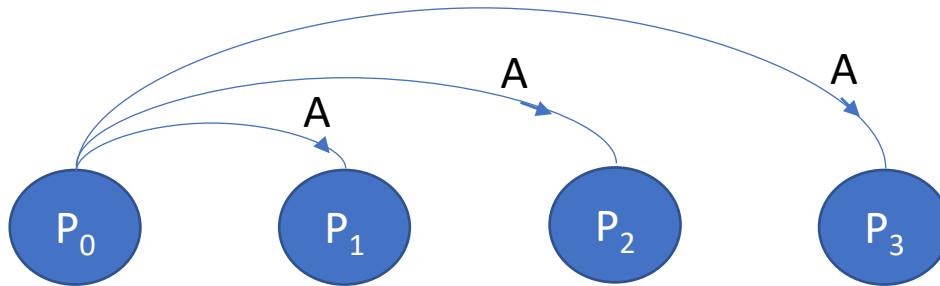
$A(P_0) \xrightarrow[\text{MPI_Bcast}]{\text{Difusión}} A(P_1), A(P_2), A(P_3)$



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

`MPI_Bcast(A, M*N, MPI_DOUBLE, 0, MPI_COMM_WORLD)`

Ejercicio 7



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

Implementación usando comunicaciones punto a punto:

```
double A[M][N];  
MPI_Init(&argc, &argv);  
int id, p;  
MPI_Status stat;  
MPI_Comm_size(MPI_COMM_WORLD, &p);  
MPI_Comm_rank(MPI_COMM_WORLD, &id);  
if (id==0)  
    for(i=1; i<p; i++)  
        MPI_Send(A, M*N, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);  
else  
    MPI_Recv(A, M*N, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, &stat);  
MPI_Finalize();
```

Tiempo de comunicaciones:

$$t_c = (p - 1)(t_s + MNt_w)$$

Ejercicio 8

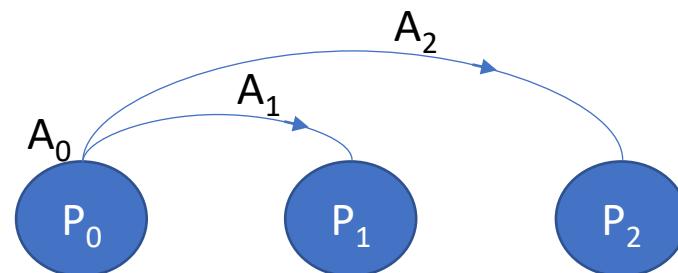
Supongamos que en MPI se dispone de **p** procesos y que el procesador P0 contiene a una matriz **A** de dimensión MxN de tipo double ¿Qué función usariás para repartir la matriz **A** entre todos los procesos? ¿Cómo podrías sustituir esa comunicación colectiva mediante comunicaciones punto a punto, suponiendo que el número de procesos **p** divide a **M**? Calcula el tiempo de comunicaciones.

Nota: Las matrices locales las almacenaremos en las matrices **Al**. Por comodidad, supondremos que esas matrices son también de dimensión MxN.

$$A(P_0) = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \end{bmatrix} k \xrightarrow[\text{MPI_Scatter}]{\text{Reparto}} \begin{bmatrix} Al(P_0) = A_0 \\ Al(P_1) = A_1 \\ Al(P_2) = A_2 \end{bmatrix} k$$

$k = M / p$

($k=n^{\circ}$ de filas de $A_i=n^{\circ}$ de filas de $Al(P_i)$, $p=\text{número de procesos}$)



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

MPI_Scatter(A, k*N, MPI_DOUBLE, Al, k*N, MPI_DOUBLE, 0, MPI_COMM_World)

Ejercicio 8

$$A(P_0) = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \end{bmatrix} k \xrightarrow[\text{MPI_Scatter}]{\text{Reparto}} \begin{bmatrix} Al(P_0) = A_0 \\ Al(P_1) = A_1 \\ Al(P_2) = A_2 \end{bmatrix} k$$

$k = M / p$

($k = \text{nº de filas de } A_i = \text{nº de filas de } Al(P_i)$, $p = \text{número de procesos}$)

Ejemplo: $M=12$, $N=6$, $p=3 \rightarrow k=12/3=4$

$$A(P_0) = \begin{array}{c} A[0][0] \quad x \ x \ x \ x \ x \\ A[1][0] \quad x \ x \ x \ x \ x \\ A[2][0] \quad x \ x \ x \ x \ x \\ A[3][0] \quad x \ x \ x \ x \ x \\ A[4][0] \quad x \ x \ x \ x \ x \\ A[5][0] \quad x \ x \ x \ x \ x \\ A[6][0] \quad x \ x \ x \ x \ x \\ A[7][0] \quad x \ x \ x \ x \ x \\ A[8][0] \quad x \ x \ x \ x \ x \\ A[9][0] \quad x \ x \ x \ x \ x \\ A[10][0] \quad x \ x \ x \ x \ x \\ A[11][0] \quad x \ x \ x \ x \ x \end{array} \xrightarrow{\hspace{1cm}} \begin{array}{c} Al(P_0) = \begin{array}{cccccc} x & x & x & x & x & x \end{array} \\ Al(P_1) = \begin{array}{cccccc} x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \end{array} \\ Al(P_2) = \begin{array}{cccccc} x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \end{array} \end{array}$$

$K=M/p$

P0:

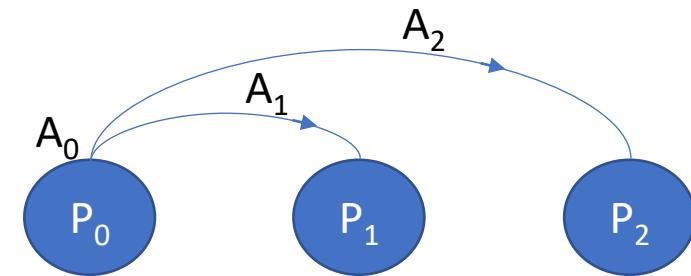
Copiar k filas de A en Al (kN elementos consecutivos)

Para $i=1$ hasta $p-1$

Enviar bloque de kN elementos desde $A[k*i][0]$ a P_i

Resto:

Recibir en Al kN elementos de P_0



K=M/p

P0:

Copiar k filas de A en AI (kN elementos consecutivos)

Para i=1 hasta p-1

Enviar bloque de kN elementos desde A[k*i][0] a Pi

Resto:

Recibir en AI kN elementos de P0



```
MPI_Init(&argc, &argv);
double A[M][N], AI[M][N];
int id, p;
MPI_Status stat;
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
int k=M/p; /* k= nº de filas de los bloques que se envían/reciben*/
if (id==0){
    MPI_Sendrecv(&A[0][0], k*N, MPI_DOUBLE, 0, 1, AI, k*N, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, &stat);
    for(i=1; i<p; i++)
        MPI_Send(&A[k*i][0], k*N, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
}
else
    MPI_Recv(AI, k*N, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, &stat);
MPI_Finalize();
```

Ejercicio 8

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

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

int MPI_Sendrecv(void *sendbuf, int
sendcount, MPI_Datatype sendtype, int
dest, int sendtag, void *recvbuf, int
recvcount, MPI_Datatype recvtype, int
source, int recvtag, MPI_Comm comm,
MPI_Status *status)
```

Tiempo de comunicaciones:

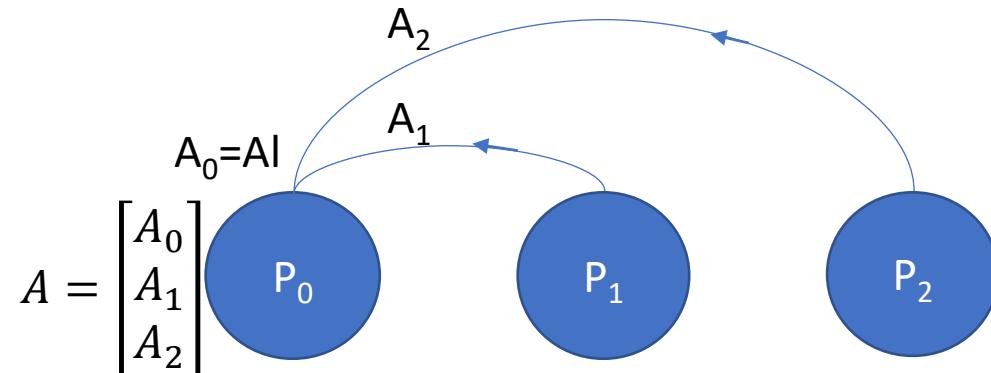
$$t_c = (p - 1) \left(t_s + \frac{MN}{p} t_w \right)$$

Ejercicio 9

Supongamos que en MPI se dispone de **p** procesos y una matriz **A** de dimensión MxN de tipo double se encuentra repartida en matrices locales **Al** también de tamaños MxN ¿Qué función de comunicaciones colectivas deberían ejecutar todos los procesos para que P0 recogiera en **A** todas las matrices? ¿Cómo podrías sustituir esa comunicación colectiva mediante comunicaciones punto a punto, suponiendo que el número de procesos **p** divide a **M**? Calcula el tiempo de comunicaciones.

$$\begin{matrix} N \\ \left[\begin{array}{c} Al(P_0) = A_0 \\ \hline Al(P_1) = A_1 \\ \hline Al(P_2) = A_2 \end{array} \right] k \xrightarrow{\text{Recogida MPI_Gatter}} A(P_0) = \left[\begin{array}{c} A_0 \\ A_1 \\ A_2 \end{array} \right] k \end{matrix}$$

$$M = kp \quad (p = \text{nº procesos})$$



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

MPI_Gather(Al, k*N, MPI_DOUBLE, A, k*N, MPI_DOUBLE, 0, MPI_COMM_WORLD)

N

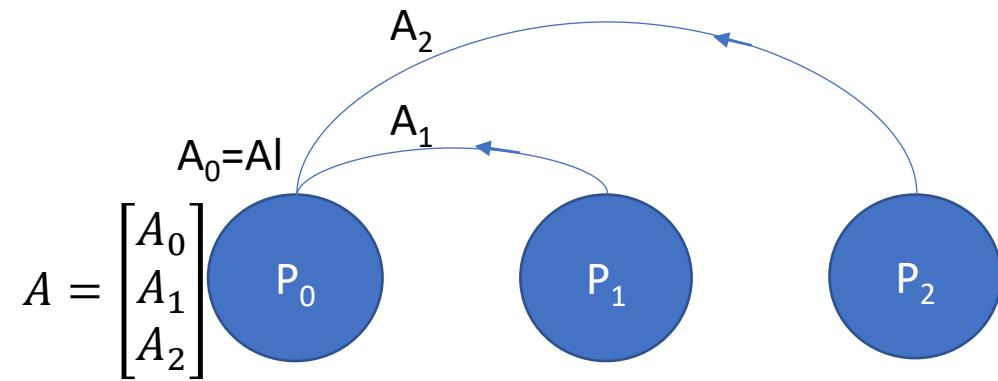
Ejercicio 9

$$\left[\begin{array}{c} Al(P_0) = A_0 \\ Al(P_1) = A_1 \\ Al(P_2) = A_2 \end{array} \right] k \xrightarrow[\text{MPI_Gatter}]{\text{Recogida}} A(P_0) = \left[\begin{array}{c} A_0 \\ A_1 \\ A_2 \end{array} \right] k$$

$$M = kp \quad (p = \text{nº procesos})$$

Ejemplo: $M=12$, $N=6$, $p=3 \rightarrow k=12/3=4$

$Al(P_0) =$ <table border="0"> <tbody> <tr><td>x x x x x x</td></tr> </tbody> </table>	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	$Al(P_1) =$ <table border="0"> <tbody> <tr><td>x x x x x x</td></tr> </tbody> </table>	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	$Al(P_2) =$ <table border="0"> <tbody> <tr><td>x x x x x x</td></tr> </tbody> </table>	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x		<table border="0"> <thead> <tr><th>$A[0][0]$</th><th>x x x x x</th></tr> <tr><th>$A[1][0]$</th><th>x x x x x</th></tr> <tr><th>$A[2][0]$</th><th>x x x x x</th></tr> <tr><th>$A[3][0]$</th><th>x x x x x</th></tr> <tr><th>$A[4][0]$</th><th>x x x x x</th></tr> <tr><th>$A[5][0]$</th><th>x x x x x</th></tr> <tr><th>$A[6][0]$</th><th>x x x x x</th></tr> <tr><th>$A[7][0]$</th><th>x x x x x</th></tr> <tr><th>$A[8][0]$</th><th>x x x x x</th></tr> <tr><th>$A[9][0]$</th><th>x x x x x</th></tr> <tr><th>$A[10][0]$</th><th>x x x x x</th></tr> <tr><th>$A[11][0]$</th><th>x x x x x</th></tr> </thead> </table>	$A[0][0]$	x x x x x	$A[1][0]$	x x x x x	$A[2][0]$	x x x x x	$A[3][0]$	x x x x x	$A[4][0]$	x x x x x	$A[5][0]$	x x x x x	$A[6][0]$	x x x x x	$A[7][0]$	x x x x x	$A[8][0]$	x x x x x	$A[9][0]$	x x x x x	$A[10][0]$	x x x x x	$A[11][0]$	x x x x x
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
x x x x x x																																														
$A[0][0]$	x x x x x																																													
$A[1][0]$	x x x x x																																													
$A[2][0]$	x x x x x																																													
$A[3][0]$	x x x x x																																													
$A[4][0]$	x x x x x																																													
$A[5][0]$	x x x x x																																													
$A[6][0]$	x x x x x																																													
$A[7][0]$	x x x x x																																													
$A[8][0]$	x x x x x																																													
$A[9][0]$	x x x x x																																													
$A[10][0]$	x x x x x																																													
$A[11][0]$	x x x x x																																													



P0:

Copiar Al en la primera posición de A
 Para $i=1$ hasta $p-1$
 Recibir en $A[k*i][0]$ bloque de kN elementos de P_i
 Fin Para

Resto:

Enviar los kN elementos de Al a P0

P0:

Copiar Al a partir de la primera posición de A

Para i=1 hasta p-1

Recibir en A[k*i][0] bloque de kN elementos de Pi

Resto:

Enviar los kN elementos de Al a P0

```
MPI_Init(&argc, &argv);
double A[M][N], Al[M][N];
int id, p;
MPI_Status stat;
MPI_Comm_size(MPI_COMM_WORLD,&p);
MPI_Comm_rank(MPI_COMM_WORLD,&id);
int k=M/p; /* k= nº de filas de los bloques que se envían/reciben
if (id==0){
    MPI_Sendrecv(Al, k*N, MPI_DOUBLE, 0, 1, A, k*N, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, &stat);
    for(i=1; i<p; i++)
        MPI_Recv(&A[i*k][0], k*N, MPI_DOUBLE, i, 1, MPI_COMM_WORLD, &stat);
}
else
    MPI_Send(Al, k*N, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);

MPI_Finalize();
```

Ejercicio 9

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

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

int MPI_Sendrecv(void *sendbuf, int
                 sendcount, MPI_Datatype sendtype, int
                 dest, int sendtag, void *recvbuf, int
                 recvcount, MPI_Datatype recvtype, int
                 source, int recvtag, MPI_Comm comm,
                 MPI_Status *status)
```

Tiempo de comunicaciones:

$$t_c = (p - 1) \left(t_s + \frac{MN}{p} t_w \right)$$

Ejercicio 9

¿Qué cambiarías en el código que usa comunicaciones colectivas para que la matriz A la tuvieran todos los procesos?

$$\begin{array}{c} N \\ \left[\begin{array}{c} Al(P_0) = A_0 \\ Al(P_1) = A_1 \\ Al(P_2) = A_2 \end{array} \right] k \xrightarrow[\text{MPI_Allgather}]{\text{Recogida}} A(P_0, P_1, P_2) = \left[\begin{array}{c} A_0 \\ A_1 \\ A_2 \end{array} \right] k \\ M = kp \quad (p = \text{nº procesos}) \end{array}$$

MPI_Gather(Al, k*N, MPI_DOUBLE, A, k*N, MPI_DOUBLE, 0, MPI_COMM_WORLD)



MPI_Allgather(Al, k*N, MPI_DOUBLE, A, k*N, MPI_DOUBLE, MPI_COMM_WORLD)

Ejercicio 10 (Suma elementos de un vector)

Realiza una implementación paralela mediante MPI de la función **suma_vec**, la cual tiene como argumento de entrada un vector **v** de dimensión **N** y devuelve la suma de los elementos del vector v. Supondremos que inicialmente **v** está almacenada en P0 y que **N** es divisible entre el número de procesos **p**. Calcula el tiempo paralelo.

Solución:

```
double suma_vec (double v[N]) {
```

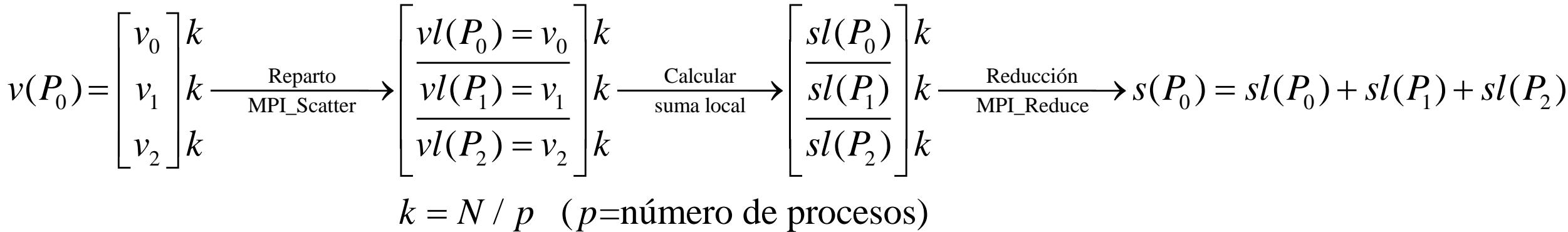
```
.....
```

```
    return 0;
```

```
}
```

$$v(P_0) = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} k \xrightarrow[\text{MPI_Scatter}]{\text{Reparto}} \begin{bmatrix} vl(P_0) = v_0 \\ vl(P_1) = v_1 \\ vl(P_2) = v_2 \end{bmatrix} k \xrightarrow[\text{sumar local}]{\text{Calcular}} \begin{bmatrix} sl(P_0) \\ sl(P_1) \\ sl(P_2) \end{bmatrix} k \xrightarrow[\text{MPI_Reduce}]{\text{Reducción}} s(P_0) = sl(P_0) + sl(P_1) + sl(P_2)$$

$k = N / p$ (p=número de procesos)



```

double suma_vecp(double v[N]){
    int p;
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    int i, k; /*k=nº de filas de los v_i*/
    /*En algunas ocasiones por simplicidad declaramos: double vl[N];*/
    double *vl = (double*) malloc(sizeof(double)*k);
    double v[N], sl=0, s; /* sl=suma local, s= suma total*/
    k=N/p;
    MPI_Scatter(v, k, MPI_DOUBLE, vl, k, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    for (i=0; i<k; i++) /*calcular suma local*/
        sl+=vl[i];
    MPI_Reduce(&sl, &s, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    return s;
}

```

```

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

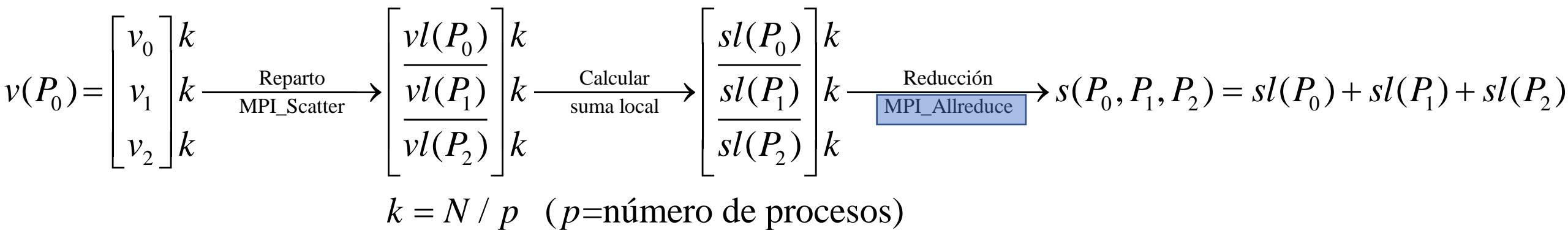
int MPI_Reduce(void *sendbuf, void
*recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, int root, MPI_Comm
comm)

```

$$\begin{aligned}
t_{\text{Scatter}} &= (p - 1) \left(t_s + \frac{N}{p} t_w \right) \\
t_{\text{Reduce}} &= (p - 1)(t_s + t_w) \\
t_c(N, p) &= (p - 1) \left(2t_s + \left(1 + \frac{N}{p} \right) t_w \right)
\end{aligned}$$

Ejercicio 10 (Suma elementos de un vector)

¿Qué modificaría en la implementación anterior para que todos recibiesen la suma?



```
MPI_Reduce(&sl, &s, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

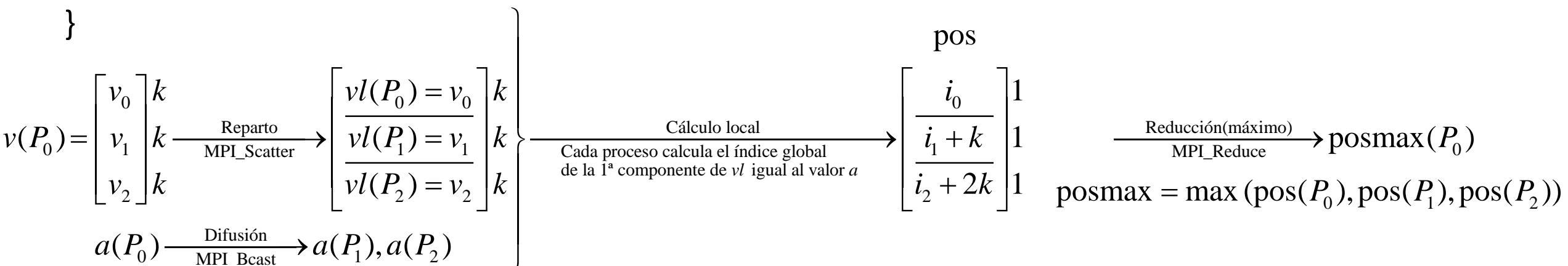


```
MPI_Allreduce(&sl, &s, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

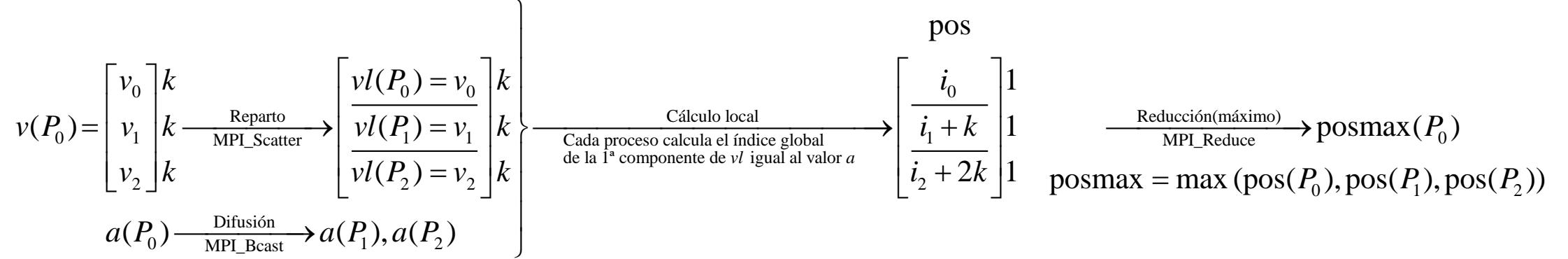
Ejercicio 10 (Búsqueda lineal)

La siguiente función devuelve la última posición del vector **v** coincidente con **a** o **-1** si no la hubiese. Realiza una implementación MPI, suponiendo que inicialmente **v** y **a** están inicialmente almacenados en la memoria local de P0. Calcular el coste de las comunicaciones.

```
int pos(int v[N], int a){
    int pos=-1, i;
    for(i=N-1; i>=0; i--)
        if (v[i]==a){
            pos=i;
            break;
    }
    return pos;
}
```



$$k = N / p \quad (p = \text{número de procesos})$$



$$k = N / p \quad (p = \text{número de procesos})$$

```

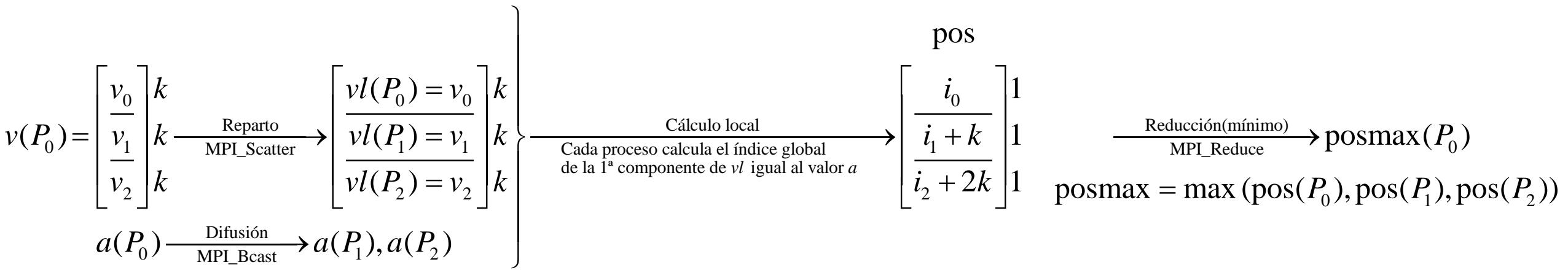
int posp(int v[N], int a){
    int p, id;
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&id);
    int k=N/p; /* Número de componentes de vl que le corresponden a cada proceso*/
    int pos, posmax, i;
    int *vl = (int*) malloc(sizeof(int)*k);
    MPI_Scatter(v, k, MPI_INT, vl, k, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&a, 1, MPI_INT, 0, MPI_COMM_WORLD);
    int pos=-1;
    for(i=k-1; i>=0; i--)
        if (vl[i]==a){
            pos=i+id*k; /*posición global=posición local+k*id*/
            break;
        }
    MPI_Reduce(&pos, &posmax, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
    return posmax;
}

```

```

int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

```



$k = N / p$ (p = número de procesos)

$$\left. \begin{array}{l}
 \text{MPI_Scatter(vector de dimensión } N): t_{c_s} = (p-1) \left(t_s + \frac{N}{p} t_w \right) \\
 \text{MPI_Gatter(dato simple)}: t_{c_g} = (p-1)(t_s + t_w) \\
 \text{MPI_Reduce(dato simple)}: t_{c_r} = (p-1)(t_s + t_w)
 \end{array} \right\} t_c = (p-1) \left(3t_s + \left(\frac{N}{p} + 1 \right) t_w \right)$$

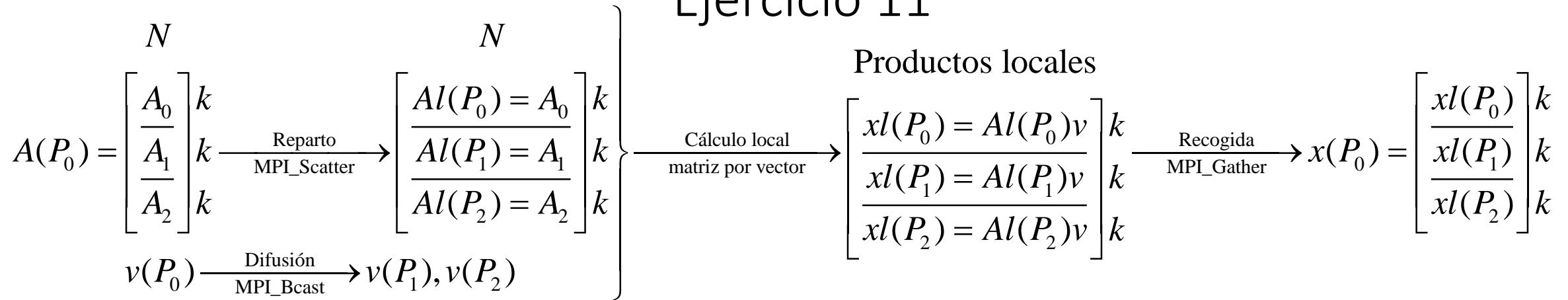
Ejercicio 11

La siguiente función calcula el producto de una matriz cuadrada por un vector, ambos de dimensión N:

```
void prod_mat_vec(double A[N][N], double v[N], double x[N]){
    int i, j;
    for (i=0;i<N;i++) {
        x[i]=0;
        for (j=0; j<N; j++)
            x[i] += A[i][j]*v[j];
    }
}
```

Paraleliza la función anterior mediante MPI, teniendo en cuenta que el proceso P0 contiene a la matriz A y al vector v, y que el vector x debe quedar almacenado en P0. Calcula el coste paralelo.

Ejercicio 11



```

void prod_mat_vec(double A[N][N], double v[N], double x[N]){
    int i, j, k, p;
    double Al[N][N], xl[N];
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    k=N/p;

    MPI_Scatter(A, k*N, MPI_DOUBLE, Al, k*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(v, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    for (i=0; i<k; i++) {
        xl[i]=0;
        for (j=0;j<N;j++)
            xl[i] += Al[i][j]*xl[j];
    }
    MPI_Gather(xl, k, MPI_DOUBLE, x, k, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

```

```

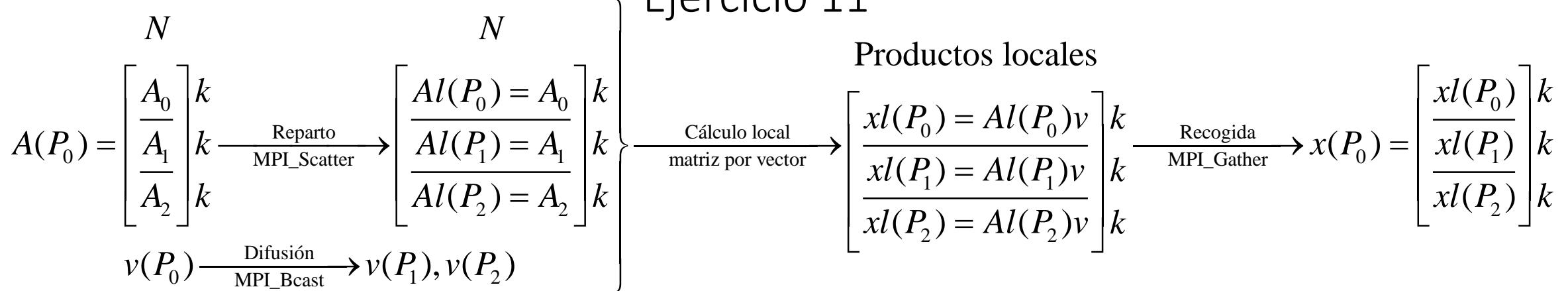
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

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

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

```

Ejercicio 11



```
for (i=0;i<k;i++) {
    xl[i]=0;
    for (j=0;j<N;j++)
        xl[i] += Al[i][j]*x[j];
}
```

Tiempo aritmético:

$$t_a = \sum_{i=0}^{k-1} \sum_{j=0}^N 2 = 2kN = \frac{2N^2}{p} \text{ flops}$$

Tiempo de comunicaciones:

$$\left. \begin{array}{l} \text{MPI_Scatter : } t_{c_s} = (p-1) \left(t_s + \frac{N^2}{p} t_w \right) \\ \text{MPI_Bcast: } t_{c_b} = (p-1) \left(t_s + N t_w \right) \\ \text{MPI_Gatter : } t_{c_g} = (p-1) \left(t_s + \frac{N}{p} t_w \right) \end{array} \right\} t_c = (p-1) \left(3t_s + \left(\frac{N^2}{p} + N + \frac{N}{p} \right) t_w \right)$$

Trasparencias adicionales T3-S3

Tiempo paralelo:

$$t_p = t_a + t_c$$

Ejercicio 12

Realiza una implementación paralela mediante MPI de la función **suma_mat**, la cual tiene como argumento de entrada una matriz A de dimensión MxN y devuelve la suma de los elementos de la matriz A. Supondremos que inicialmente A está almacenada en P0 y que **M** es divisible entre el número de procesos **p**. Calcula el tiempo paralelo.

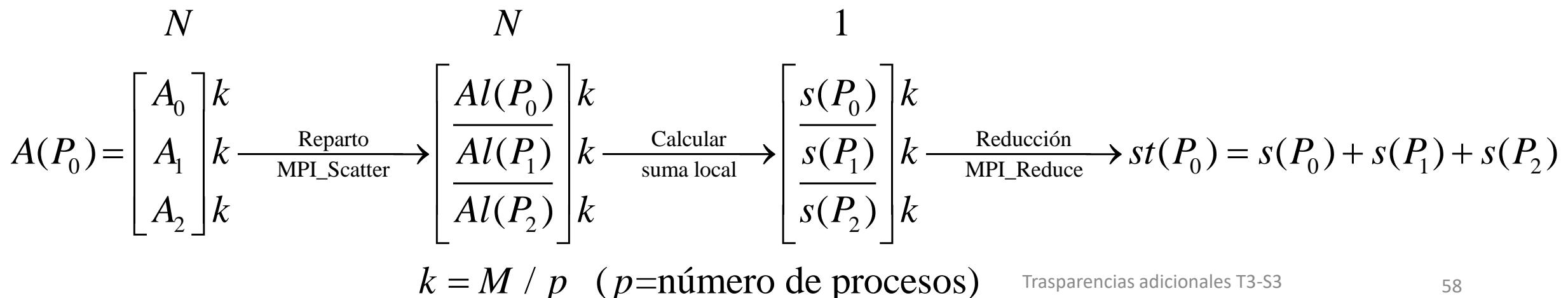
Solución:

```
double suma_mat (double A[M][N]) {
```

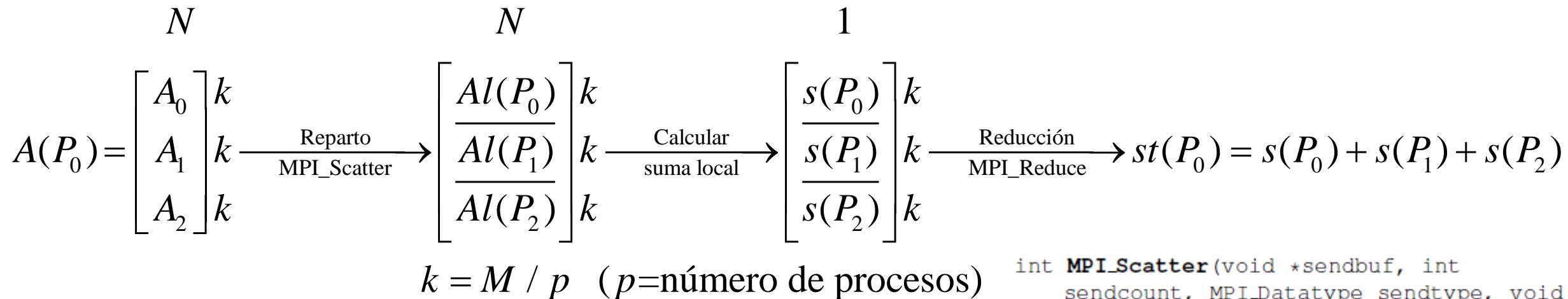
```
.....
```

```
    return 0;
```

```
}
```



Ejercicio 12



```

double suma_mat_p(double A[M][N]){
    int id, p;
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    int i, j, k= M/p; /*k=nº de filas de Ai = nº de filas de Al(Pi)*/
    double Al[M][N], s=0, st; /* s=suma local, st= suma total*/
    MPI_Scatter(A, k*N, MPI_DOUBLE, Al, k*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    for (i=0; i<k; i++) /*calcular suma local*/
        for (j=0; j<N; j++)
            s+=Al[i][j];
    MPI_Reduce(&s, &st, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    return st;
}

```

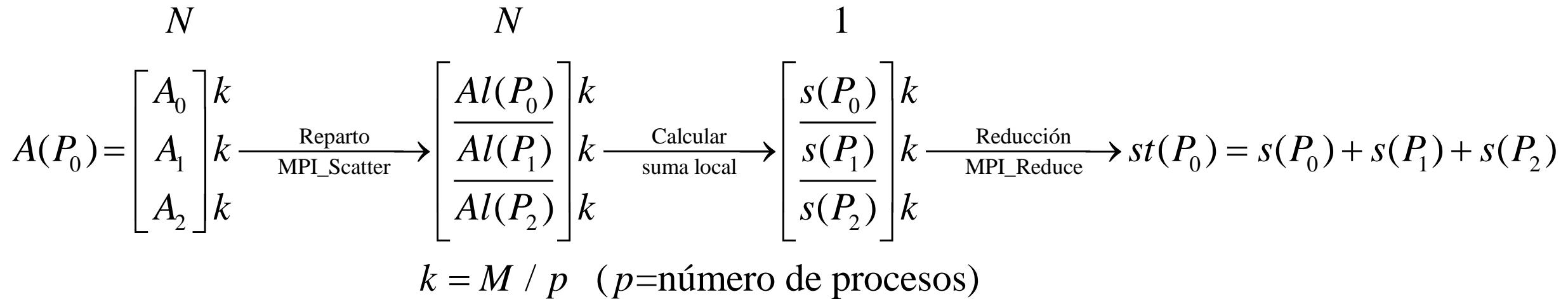
```

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

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

```

Ejercicio 12



$$t_{\text{Scatter}} = (p - 1) \left(t_s + \frac{kN}{p} t_w \right)$$

$$t_{\text{Reduce}} = (p - 1)(t_s + t_w)$$

$$t_c(N, p) = (p - 1) \left(2t_s + \left(1 + \frac{kN}{p} \right) t_w \right)$$

Ejemplo de la función MPI_Scan

Dado un vector v de longitud N , distribuido entre los procesos, donde cada proceso tiene n_{local} elementos consecutivos del vector, se quiere obtener la posición inicial del subvector local

$$v(\text{distribuido}) = \begin{bmatrix} v(P_0) \\ \hline \frac{v(P_1)}{v(P_0)} \\ \hline \frac{v(P_2)}{v(P_1)} \end{bmatrix} \begin{array}{l} \xrightarrow{n_0} \text{global}(v(P_0)) : 0 \\ \xrightarrow{n_1} \text{global}(v(P_1)) : n_0 \\ \xrightarrow{n_2} \text{global}(v(P_2)) : n_0 + n_1 \end{array}$$
$$n_i = \text{nlocal}(P_i)$$

Ejemplo de la función MPI_Scan

```
int global, nlocal, N;
calcula_nlocal(N,&nlocal); /* por ejemplo, nlocal=N/p */
MPI_Scan(&nlocal,&global,1,MPI_INT,MPI_SUM,comm);
global -= nlocal;
```

$$\begin{aligned} \text{global}(v(P_0)) &= n_0 \\ \text{global}(v(P_1)) &: n_0 + n_1 \\ \text{global}(v(P_2)) &: n_0 + n_1 + n_2 \end{aligned}$$

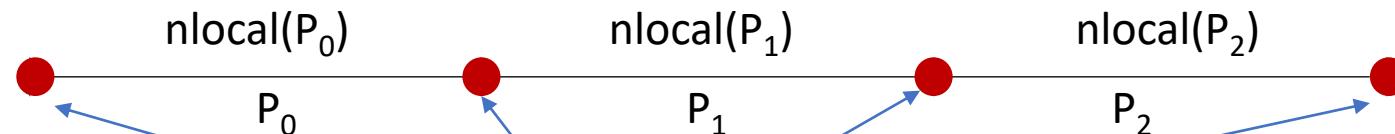
MPI_Scan

	Estado Inicial	Estado Final
P ₀	A	A
P ₁	B	A+B
P ₂	C	A+B+C
P ₃	D	A+B+C+D
P ₄	E	A+B+C+D+E
P ₅	F	A+B+C+D+E+F

$$v(\text{distribuido}) = \left[\frac{v(P_0)}{n_0} \quad \frac{v(P_1)}{n_1} \quad \frac{v(P_2)}{n_2} \right]$$

$$n_i = \text{nlocal}(P_i)$$

$$\begin{aligned} \text{global}(v(P_0)) &= n_0 - n_0 = 0 \\ \text{global}(v(P_1)) &= n_0 + n_1 - n_1 = n_0 \\ \text{global}(v(P_2)) &: n_0 + n_1 + n_2 - n_2 = n_0 + n_1 \end{aligned}$$

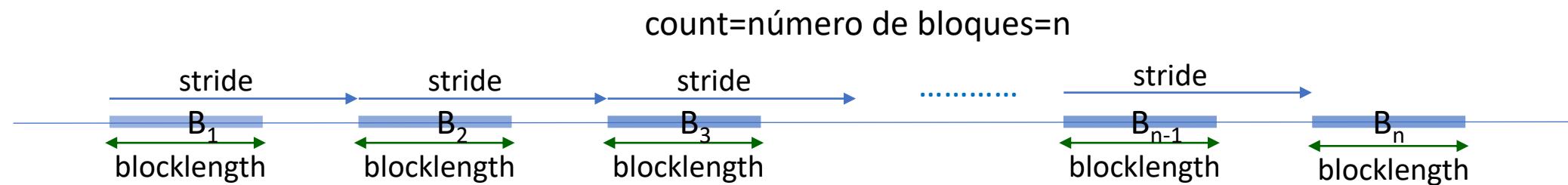


Conocer las posiciones iniciales del vector global

MPI_Type_vector

- A partir de un array, crea un nuevo tipo de datos de tipo homogéneo que permite enviar en un solo mensaje bloques de datos no contiguos que tienen la misma longitud y se encuentran separados a la misma distancia.

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



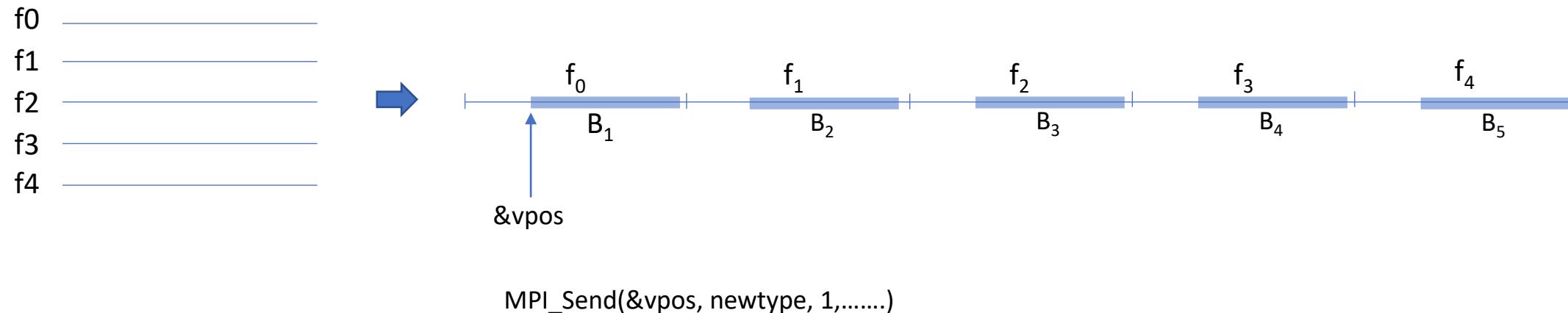
- La forma de enviar en un solo mensaje estos bloques sería:

```
MPI_Send(&vpos, newtype, 1,.....)
```

donde &pos es la dirección del primer dato de B_1

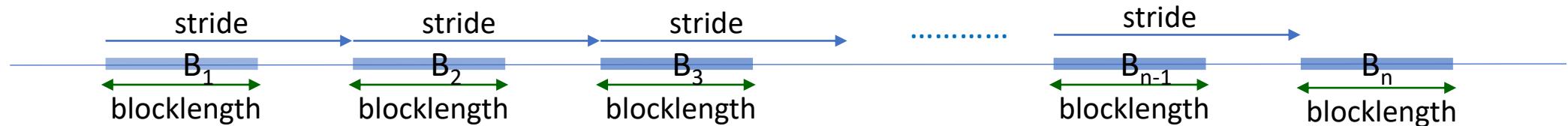
MPI_Type_vector

- Habitualmente trabajaremos con matrices de manera que nos interesará hacer envíos de bloques no consecutivos de la matriz.
- Para ello, consideraremos que las filas de la matriz se encuentran almacenadas en posiciones consecutivas de memoria



MPI_Type_vector

- Bloque de datos en origen para ser enviados:



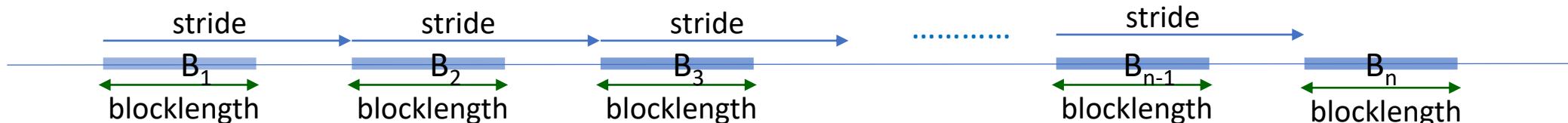
- Los bloques son enviados juntos en el mensaje:



- En el proceso receptor del mensaje, los datos recibidos los podemos almacenar:
 - Como bloques consecutivos. En este caso hay que indicar el tipo básico y el nº total de datos.



- Como bloques separados con la misma estructura que tenían los datos en el envío. En este caso, hay que indicar el tipo derivado usado y el nº datos igual a 1.

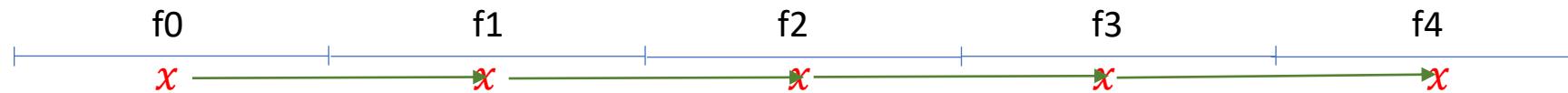


Ejercicio 13

Crea un tipo de datos en MPI que permita realizar envíos eficientes de columnas de una matriz de tipo double de orden NxN en un solo mensaje.

$$N = 5$$
$$A = \begin{bmatrix} \bullet & \bullet & x & \bullet & \bullet \\ \bullet & \bullet & x & \bullet & \bullet \\ \bullet & \bullet & x & \bullet & \bullet \\ \bullet & \bullet & x & \bullet & \bullet \\ \bullet & \bullet & x & \bullet & \bullet \end{bmatrix} N = 5$$

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



count=N
blocklength=1
stride=N



```
MPI_Datatype col;  
MPI_Type_vector(N, 1, N, MPI_DOUBLE, &col);  
MPI_Type_commit(&col);  
.....  
MPI_Type_free (&col);
```

Ejercicio 14

Sea A una matriz cuadrada de dimensión NxN de tipo double almacenada en el procesador P0.

a) Usando el tipo de datos del ejercicio anterior ¿cómo se podría realizar un envío eficiente de la segunda columna de la matriz A desde P0 a P1?, quedando esta columna almacenada en un vector v de dimensión N? ¿Cuál sería el tiempo de comunicaciones?

```
int p, id;
MPI_Status stat;
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Datatype col;
MPI_Type_vector(N, 1, N, MPI_DOUBLE, &col);
MPI_Type_commit(&col);
if (id==0)
    MPI_Send(&A[0][1], 1, col, 1, 100, MPI_COMM_WORLD);
else if (id==1)
    MPI_Recv(v, N, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD, &stat);
MPI_Type_free (&col);
```

$$A(P_0) = \begin{bmatrix} \bullet & \textcolor{blue}{x} & \bullet & \bullet & \bullet \\ \bullet & \textcolor{red}{x} & \bullet & \bullet & \bullet \\ \bullet & \textcolor{red}{x} & \bullet & \bullet & \bullet \\ \bullet & \textcolor{red}{x} & \bullet & \bullet & \bullet \\ \bullet & \textcolor{red}{x} & \bullet & \bullet & \bullet \end{bmatrix} \quad N=5$$

$v(P_1) = [\textcolor{red}{x} \ x \ x \ x \ x]$

$\xrightarrow{\textcolor{red}{xxxxx}}$

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

Receive a message from one process:

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

Tiempo de comunicaciones usando el tipo col:

$$t_c = t_s + Nt_w$$

Ejercicio 14

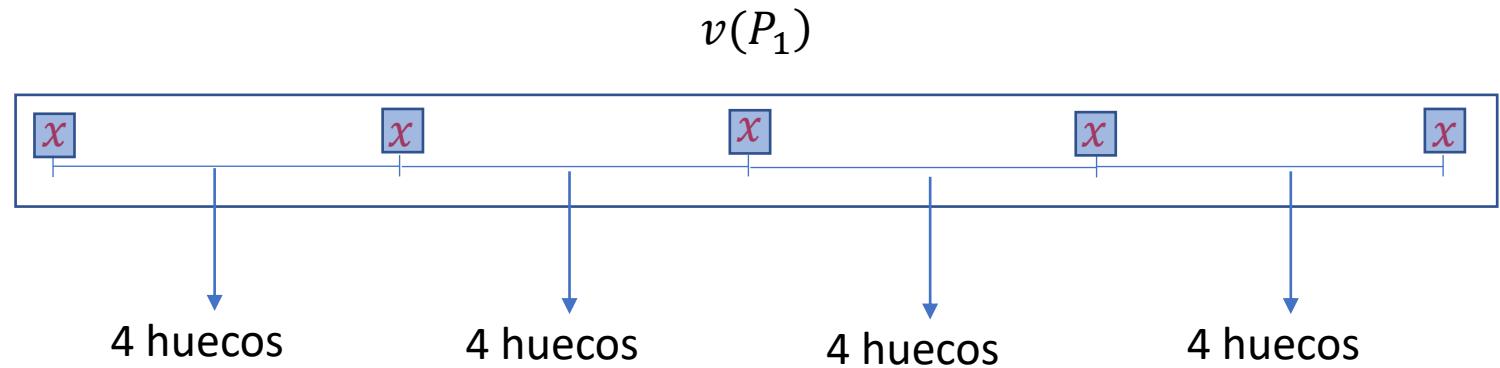
Nota: Si en la recepción hubiésemos usado:

```
MPI_Recv(v, 1, col, 0, 100, MPI_COMM_WORLD, &stat);
```

habríamos cometido un error:

- Los elementos de v no estarían almacenados consecutivamente
- La memoria reservada para v quedaría sobrepasada

$$N = 5$$
$$A = \begin{bmatrix} x & \bullet & \bullet & \bullet \\ x & \bullet & \bullet & \bullet \end{bmatrix} N = 5$$



Ejercicio 14

b) ¿Y si el envío hubiese sido sin utilizar ese tipo de datos?

```
int p, id, i;
MPI_Status stat;
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
if (id==0)
    for(i=0; i<N; i++)
        MPI_Send(&A[i][1], MPI_DOUBLE, col, 100, MPI_COMM_WORLD);
else if (id==1)
    for(i=0; i<N; i++)
        MPI_Recv(&v[i], N, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD, &stat);
```

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

Receive a message from one process:

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

Tiempo de comunicaciones sin usar el tipo **col**:

$$t_c = N(t_s + t_w)$$

Ejercicio 14

c) Usa el tipo de datos **col** para que la primera columna de la matriz A de P0 sea recibida en la tercera columna de la matriz A de P1

```

int p, id, i;
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Status stat;
MPI_Datatype col;

MPI_Type_vector(N, 1, N, MPI_DOUBLE, &col);
MPI_Type_commit(&col);
if (id==0)
    MPI_Send(&A[0][0], 1, col, 1, 100, MPI_COMM_WORLD);
else if (id==1)
    MPI_Recv(&A[0][2], 1, col, 0, 100, MPI_COMM_WORLD, &stat);
MPI_Type_free (&col);

```

$$N = 5$$

$$A(P_0) = \begin{bmatrix} x & \bullet & \bullet & \bullet & \bullet \\ x & \bullet & \bullet & \bullet & \bullet \\ x & \bullet & \bullet & \bullet & \bullet \\ x & \bullet & \bullet & \bullet & \bullet \\ x & \bullet & \bullet & \bullet & \bullet \end{bmatrix} N = 5 \xrightarrow{x \ x \ x \ x} A(P_1) = \begin{bmatrix} \bullet & \bullet & x & \bullet & \bullet \\ \bullet & \bullet & x & \bullet & \bullet \\ \bullet & \bullet & x & \bullet & \bullet \\ \bullet & \bullet & x & \bullet & \bullet \\ \bullet & \bullet & x & \bullet & \bullet \end{bmatrix} N = 5$$

```

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

```

Receive a message from one process:

```

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

```

Tiempo de comunicaciones sin usar el tipo **col**:

$$t_c = N(t_s + t_w)$$

Ejercicio 15

Crea un tipo de datos en MPI que permita realizar envíos eficientes de la diagonal de una matriz de orden N de un proceso a otro.

$$A = N \begin{bmatrix} x & x & \cdots & x & x \\ x & x & \cdots & x & x \\ \vdots & \vdots & \ddots & x & x \\ x & x & \cdots & x & x \\ x & x & \cdots & x & x \end{bmatrix}$$

count=N
blocklength=1
Stride=N+1

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

MPI_Datatype diag;
MPI_Type_vector(N, 1, N+1, MPI_DOUBLE, &diag);
MPI_Type_commit(&diag);
.....
MPI_Type_free (&diag);
```

Ejercicio 16

Crea un tipo de datos en MPI que permita realizar envíos eficientes de la antidiagonal de una matriz de orden N de un proceso a otro.

$$A = N \begin{bmatrix} x & x & \cdots & x & \textcolor{red}{x} \\ x & x & \cdots & \textcolor{red}{x} & x \\ \vdots & \vdots & \ddots & x & x \\ x & \textcolor{red}{x} & \cdots & x & x \\ \textcolor{red}{x} & x & \cdots & x & x \end{bmatrix}$$

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

count=N
blocklength=1
Stride=N-1



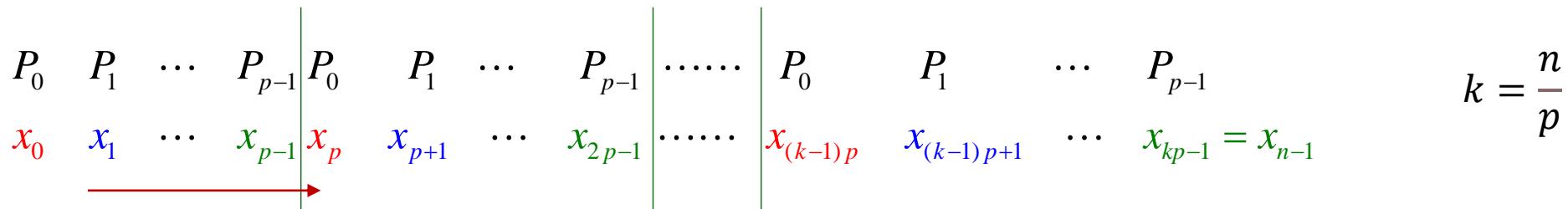
```
MPI_Datatype antidiag;  
MPI_Type_vector(N, 1, N-1, MPI_DOUBLE, &antidiag);  
MPI_Type_commit(&antidiag);  
.....  
MPI_Type_free (&antidiag);
```

Ejercicio 17

Se quiere implementar una función en MPI que permita repartir de forma cíclica un vector **x** de dimensión **n** entre **p** procesos, con la siguiente cabecera:

```
void reparto_ciclico(double x[], double xl[], int n)
```

siendo **xl** la parte local una vez se ha repartido **x**. Impleméntala definiendo para ello un tipo derivado que permita realizar comunicaciones eficientes (suponer que **n** es divisible entre el número de procesos **p**). Calcula el tiempo de comunicaciones. ¿Cuál sería el tiempo de comunicaciones si no se hubiera usado el tipo derivado?

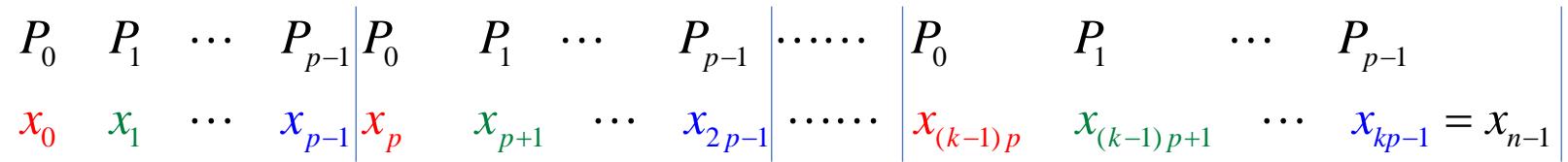


count=k
blocklength=1
stride=p

```
MPI_Datatype ciclico ;  
MPI_Type_vector(k, 1, p, MPI_DOUBLE, &ciclico);  
MPI_Type_commit(&ciclico);
```

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

Ejercicio 17



count=k
blocklength=1
Stride=p

```

void reparto_ciclico(double x[], double xl[], int n){
int p, id, k, i;
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
k=n/p;
MPI_Datatype ciclico;
MPI_Status stat;
MPI_Type_vector(k, 1, p, MPI_DOUBLE, &ciclico);
MPI_Type_commit(&ciclico);
if (id==0) {
    for (i=1; i<p; i++)
        MPI_Send(&x[i], 1, ciclico, i, 0, MPI_COMM_WORLD);
    MPI_Sendrecv(&x[0], 1, ciclico, 0, 1, xl, k, MPI_DOUBLE, 0,1, MPI_COMM_WORLD, &stat);
}
else
    MPI_Recv(&xl[0], k, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &stat);
MPI_Type_free(&ciclico);
}

```

Combined send and receive:

```

int MPI_Sendrecv(void *sendbuf, int
sendcount, MPI_Datatype sendtype, int
dest, int sendtag, void *recvbuf, int
recvcount, MPI_Datatype recvtype, int
source, int recvtag, MPI_Comm comm,
MPI_Status *status)

```

p-1 mensajes de k=n/p datos:

$$t_c = (p - 1) \left(t_s + \frac{n}{p} t_w \right)$$

Sin tipos derivados:

$$t_c = \left(n - \frac{n}{p} + 1 \right) (t_s + t_w)$$

Ejercicio 18

Se desea distribuir entre 4 procesos una matriz cuadrada de orden $2N$ almacenada en P_0 ($2N$ filas por $2N$ columnas), definida a bloques como

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$$

donde cada bloque A_{ij} corresponde a una matriz cuadrada de orden N , de manera que se quiere que el proceso P_0 almacene localmente la matriz A_{00} , P_1 la matriz A_{01} , P_2 la matriz A_{10} y P_3 la matriz A_{11} . Por ejemplo, la siguiente matriz de dimensión $N = 2$ quedaría distribuida como se muestra:

$$A = \left(\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array} \right) \quad \begin{array}{ll} \text{En } P_0: \left(\begin{array}{cc} 1 & 2 \\ 5 & 6 \end{array} \right) & \text{En } P_1: \left(\begin{array}{cc} 3 & 4 \\ 7 & 8 \end{array} \right) \\ \text{En } P_2: \left(\begin{array}{cc} 9 & 10 \\ 13 & 14 \end{array} \right) & \text{En } P_3: \left(\begin{array}{cc} 11 & 12 \\ 15 & 16 \end{array} \right) \end{array}$$

- a) Implementa una función que realice la distribución mencionada, definiendo para ello el tipo de datos MPI necesario. La cabecera de la función sería:

```
void comunica(double A[2*N][2*N], double B[N][N], int rank)
```

donde **A** es la matriz inicial, almacenada en el proceso 0, **B** es la matriz local donde cada proceso debe guardar el bloque que le corresponda de A y **rank** el índice de proceso.

Nota: se puede asumir que el número de procesos del comunicador es 4.

N

N

Ejercicio 18

$A(P_0) =$	$\begin{array}{ c c c c c } \hline x & x & \cdots & x & x \\ \hline x & x & \cdots & x & x \\ \hline \vdots & \vdots & \ddots & \vdots & \vdots \\ \hline x & x & \cdots & x & x \\ \hline x & x & \cdots & x & x \\ \hline \hline x & x & \cdots & x & x \\ \hline x & x & \cdots & x & x \\ \hline \vdots & \vdots & \ddots & \vdots & \vdots \\ \hline x & x & \cdots & x & x \\ \hline x & x & \cdots & x & x \\ \hline \end{array}$	$\begin{array}{ c c c c c } \hline x & x & \cdots & x & x \\ \hline x & x & \cdots & x & x \\ \hline \vdots & \vdots & \ddots & \vdots & \vdots \\ \hline x & x & \cdots & x & x \\ \hline x & x & \cdots & x & x \\ \hline \hline x & x & \cdots & x & x \\ \hline x & x & \cdots & x & x \\ \hline \vdots & \vdots & \ddots & \vdots & \vdots \\ \hline x & x & \cdots & x & x \\ \hline x & x & \cdots & x & x \\ \hline \end{array}$
N		\Rightarrow

$$B(P_0) = A_{00} = \begin{bmatrix} x & x & \cdots & x & x \\ x & x & \cdots & x & x \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x & x & \cdots & x & x \\ x & x & \cdots & x & x \end{bmatrix}$$

$$B(P_2) = A_{10} = \begin{bmatrix} x & x & \cdots & x & x \\ x & x & \cdots & x & x \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x & x & \cdots & x & x \\ x & x & \cdots & x & x \end{bmatrix}$$

$$B(P_1) = A_{01} = \begin{bmatrix} x & x & \cdots & x & x \\ x & x & \cdots & x & x \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x & x & \cdots & x & x \\ x & x & \cdots & x & x \end{bmatrix}$$

$$B(P_3) = A_{11} = \begin{bmatrix} x & x & \cdots & x & x \\ x & x & \cdots & x & x \\ \vdots & \vdots & \ddots & x & x \\ x & x & \cdots & x & x \\ x & x & \cdots & x & x \end{bmatrix}$$

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

count=N
blocklength=N
stride=2N



MPI_Datatype bloques22;
MPI_Type_vector(N, N, 2*N, MPI_DOUBLE, &bloques22);
MPI_Type_commit(&bloques22);

Ejercicio 18

$N \quad N$

$$A(P_0) = \begin{bmatrix} A_{00} & | & A_{01} \\ \hline A_{10} & | & A_{11} \end{bmatrix} \Rightarrow \begin{array}{ll} B(P_0) = A_{00} & B(P_1) = A_{01} \\ B(P_2) = A_{10} & B(P_3) = A_{11} \end{array}$$

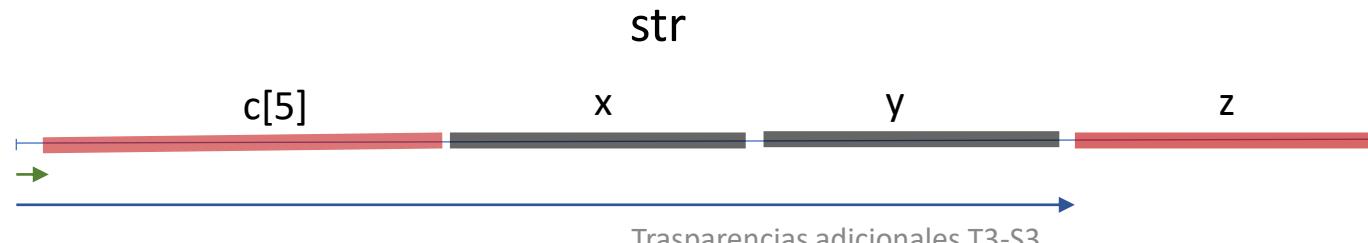
count=N
blocklength=N
Stride=2N

```
void reparto_bloques22(double A[2N][2N], B[N][N], ){  
int p, id, k, i;  
MPI_Comm_size(MPI_COMM_WORLD, &p);  
MPI_Comm_rank(MPI_COMM_WORLD, &id);  
MPI_Status stat;  
MPI_Datatype bloques22;  
MPI_Type_vector(N, N, 2N, MPI_DOUBLE, &bloques22);  
MPI_Type_commit(&bloques22);  
if (id==0) {  
    MPI_Send(&A[0][N], 1, bloques22, 1, 100, MPI_COMM_WORLD);  
    MPI_Send(&A[N][0], 1, bloques22, 2, 100, MPI_COMM_WORLD);  
    MPI_Send(&A[N][N], 1, bloques22, 3, 100, MPI_COMM_WORLD);  
    MPI_Sendrecv(A, 1, bloques22, 0, 100, B, N*N, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD, &stat);  
}  
else  
    MPI_Recv(B, N*N, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD, &stat);  
MPI_Type_free(& bloques22);  
}
```

```
int MPI_Send(void *buf, int count,  
            MPI_Datatype datatype, int dest, int  
            tag, MPI_Comm comm)  
  
int MPI_Recv(void *buf, int count,  
            MPI_Datatype datatype, int source, int  
            tag, MPI_Comm comm, MPI_Status *status)  
  
int MPI_Sendrecv(void *sendbuf, int  
                 sendcount, MPI_Datatype sendtype, int  
                 dest, int sendtag, void *recvbuf, int  
                 recvcount, MPI_Datatype recvtype, int  
                 source, int recvtag, MPI_Comm comm,  
                 MPI_Status *status)
```

Tipos de datos derivados heterogéneos

```
MPI_Type_create_struct(int count, int lens[], MPI_Aint displs[], MPI_Datatype old_types[], MPI_Datatype *new_type)  
Ejemplo: Queremos a partir de la siguiente estructura, enviar envío de los datos c[5] y z. Crea un nuevo tipo de dato.  
struct {  
    char c[5];  
    double x, y, z;  
} str;  
  
MPI_Datatype types[2] = {MPI_CHAR, MPI_DOUBLE}, newtype;  
int lengths[2] = { 5, 1 }; /* solo queremos enviar c y z */  
MPI_Aint ad1, ad2, ad3, displs[2]; /* MPI_Aint se usa cuando se quiere tener variables que contengan direcciones de memoria*/  
MPI_Get_address(& str, &ad1);  
MPI_Get_address(&str.c[0], &ad2);  
MPI_Get_address(& str.z, &ad3);  
displs[0] = ad2 - ad1;  
displs[1] = ad3 - ad1;  
MPI_Type_struct(2, lengths, displs, types, &newtype);  
MPI_Type_commit(&newtype);
```



Ejercicio 19 (ejemplo maestro-trabajadores)

Se quiere implementar el cálculo de la ∞ -norma de una matriz cuadrada A de dimensión n , que se obtiene como el máximo de las sumas de los valores absolutos de los elementos de cada fila:

$$\|A\|_{\infty} = \max_{i=0, \dots, n-1} \left\{ \sum_{j=0}^{n-1} |a_{ij}| \right\}$$

Para ello, se propone un esquema maestro-trabajadores. A continuación, se muestra la función correspondiente al maestro (el proceso con identificador 0). La matriz se almacena por filas en un array unidimensional, y suponemos que es muy dispersa (tiene muchos ceros), por lo que el maestro envía únicamente los elementos no nulos, usando para ello la función `comprime`. Implementa mediante una función el código MPI de los procesos trabajadores.

Ejercicio 19

```
int comprime(double *A,int n,int i,double *buf){  
int j,k = 0;  
for (j=0;j<n;j++)  
    if (A[i*n+j]!=0.0) { buf[k] = A[i*n+j]; k++; }  
    return k;  
}  
double maestro(double *A,int n){  
double buf[n];  
double norma=0.0,valor;  
int fila, completos=0, p, i, k;  
MPI_Status status;  
MPI_Comm_p(MPI_COMM_WORLD,&p);  
for (fila=0; fila<p-1; fila++) {  
    if (fila<n) {  
        k = comprime(A, n, fila, buf);  
        MPI_Send(buf, k, MPI_DOUBLE, fila+1, TAG_FILA, MPI_COMM_WORLD);  
    }  
    else  
        MPI_Send(buf, 0, MPI_DOUBLE, fila+1, TAG_END, MPI_COMM_WORLD);  
}  
while (completos<n) {  
    MPI_Recv(&valor, 1, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_RESU, MPI_COMM_WORLD, &status);  
    if (valor>norma) norma=valor;  
    completos++;  
    if (fila<n) {  
        k = comprime(A, n, fila, buf);  
        fila++;  
        MPI_Send(buf, k, MPI_DOUBLE, status.MPI_SOURCE, TAG_FILA,MPI_COMM_WORLD);  
    }  
    else  
        MPI_Send(buf, 0, MPI_DOUBLE, status.MPI_SOURCE, TAG_END, MPI_COMM_WORLD);  
}  
return norma;  
}
```

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

Receive a message from one process:

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

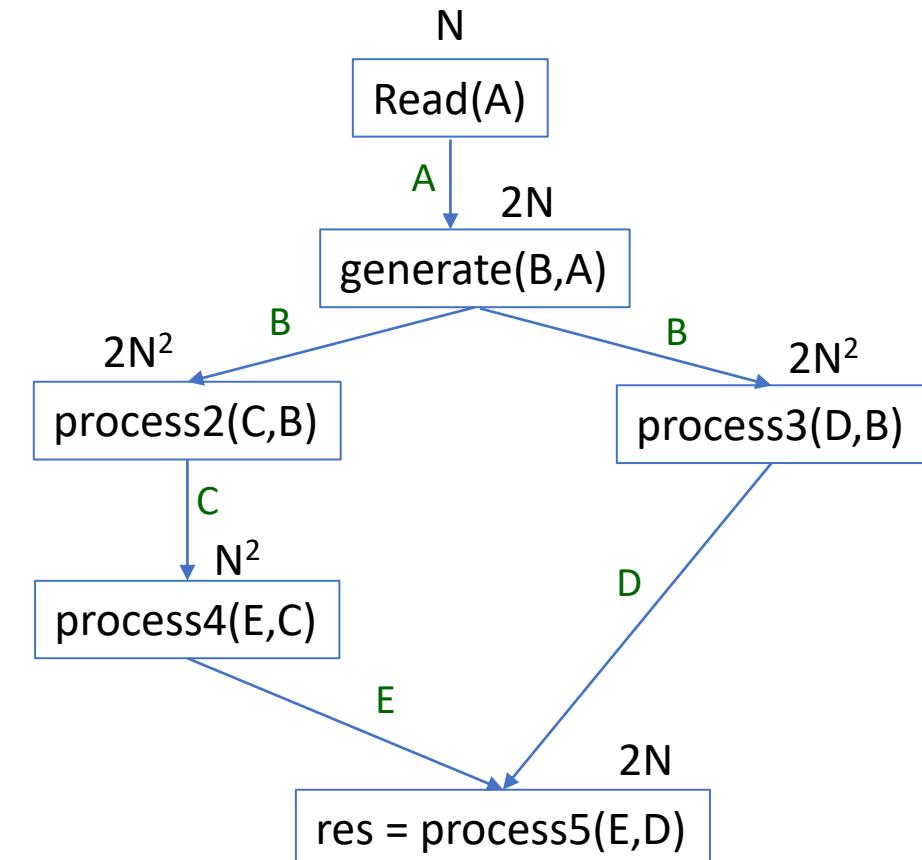
```
void trabajador(int n){  
    double s, buf[n];  
    int i, k;  
    MPI_Status status;  
    MPI_Recv(buf, n, MPI_DOUBLE, 0, MPI_ANY_TAG,  
             MPI_COMM_WORLD, &status);  
    while (status.MPI_TAG==TAG_FILA) {  
        MPI_Get_count(&status, MPI_DOUBLE, &k);  
        s=0.0;  
        for (i=0; i<k; i++)  
            s+=fabs(buf[i]);  
        MPI_Send(&s, 1, MPI_DOUBLE, 0, TAG_RESU,  
                MPI_COMM_WORLD);  
        MPI_Recv(buf, n, MPI_DOUBLE, 0, MPI_ANY_TAG,  
                 MPI_COMM_WORLD, &status);  
    }  
}
```

Ejercicio 20

En el siguiente programa secuencial, en el que indicamos con comentarios el coste computacional de cada función, todas las funciones invocadas modifican únicamente el primer argumento. Observa que A, D y E son vectores, mientras que B y C son matrices.

```
int main (int argc, char *argv[]) {  
    double A[N], B[N][N], C[N][N], D[N], E[N], res;  
    read(A); // T0, cost N  
    generate(B, A); // T1, cost 2N  
    process2(C, B); // T2, cost 2N^2  
    process3(D, B); // T3, cost 2N^2  
    process4(E, C); // T4, cost N^2  
    res = process5(E, D); // T5, cost 2N  
    printf("Result: %f\n", res);  
    return 0;  
}
```

a) Obtén el grafo de dependencias.



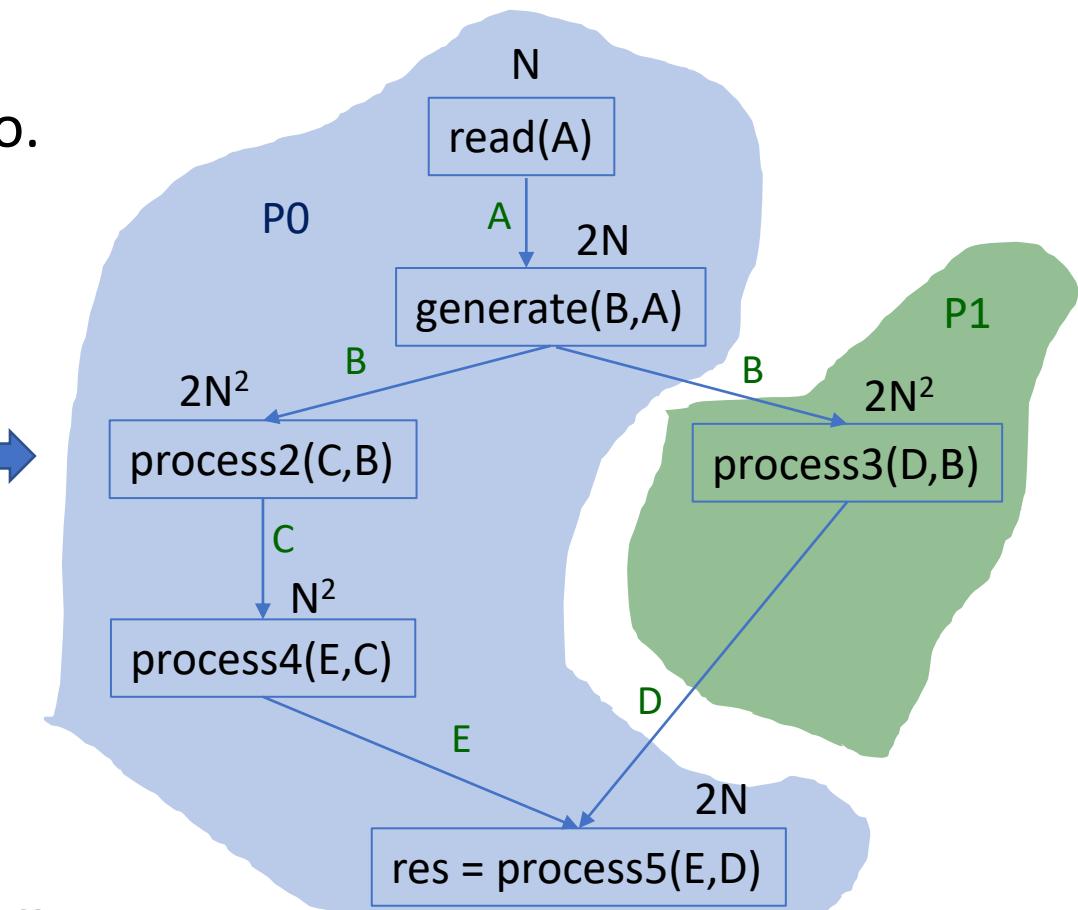
Ejercicio 20

Implementa una versión paralela con MPI, teniendo en cuenta los siguientes aspectos:

- Utiliza el número más apropiado de procesos paralelos para que la ejecución sea lo más rápida posible. Solo el proceso P0 debe realizar las operaciones read y printf.
- Presta atención al tamaño de los mensajes y utiliza las técnicas de agrupamiento y replicación si fuera conveniente.
- Realiza la implementación del programa completo.
`double A[N], B[N][N], C[N][N], D[N], E[N], res;`

Se utilizarán dos procesos
Posible distribución de tareas:

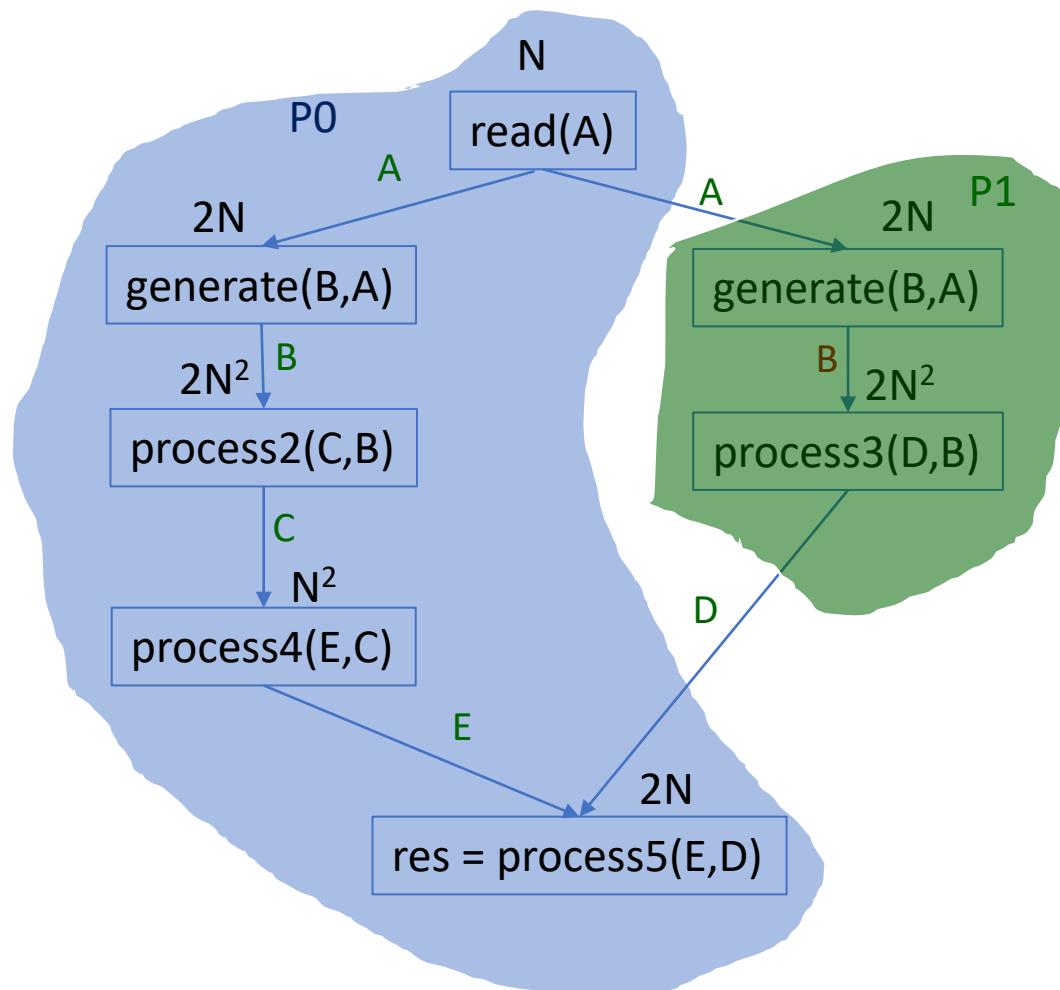
Problema: el coste de envío de la matriz **B** es superior al del envío del vector **D**, y la tarea **generate** no puede hacerse en paralelo con ninguna otra



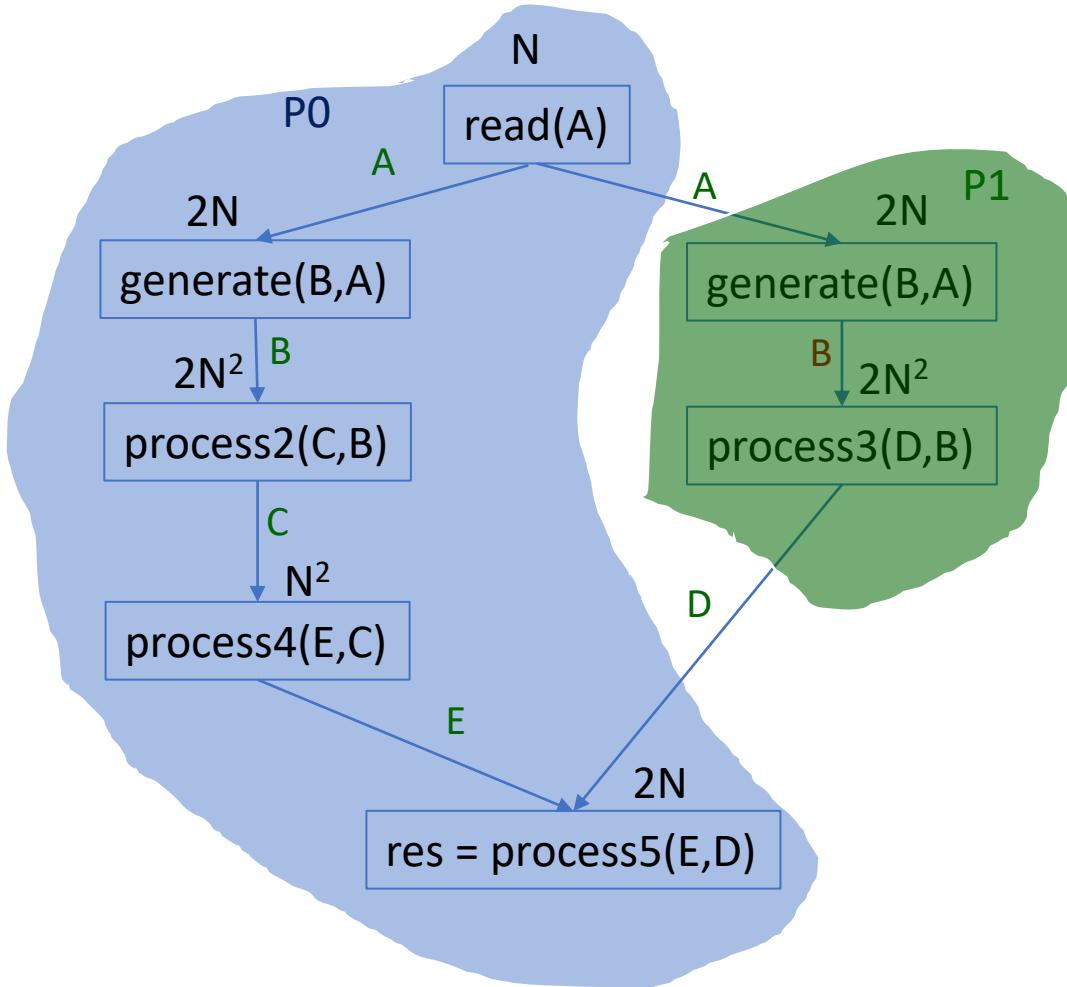
Ejercicio 20

Solución: replicar la tarea **generate**

```
double A[N], B[N][N], C[N][N], D[N], E[N], res;
```



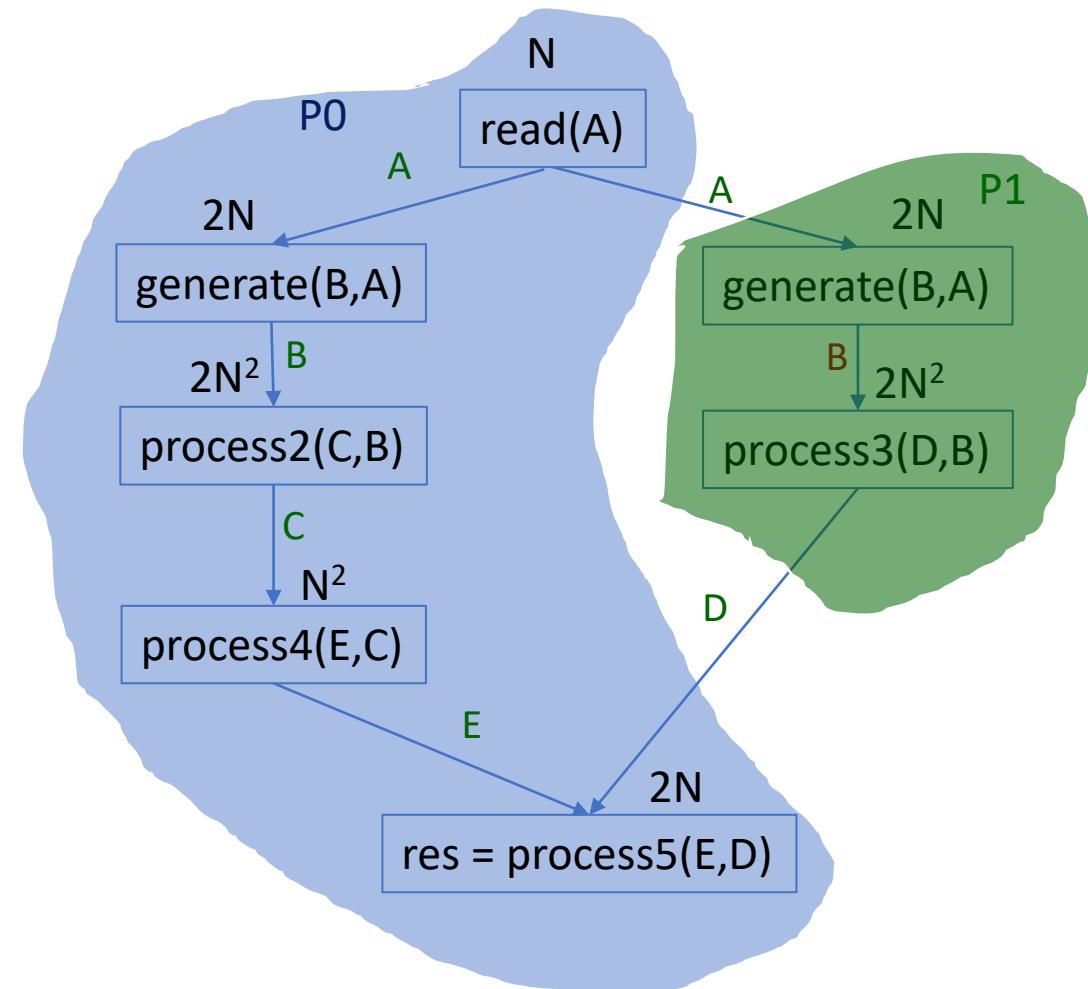
Ejercicio 20



```
double A[N], B[N][N], C[N][N], D[N], E[N], res;  
int rank, p;  
MPI_Status status;  
MPI_Init(&argc, &argv);  
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
MPI_Comm_size(MPI_COMM_WORLD, &p);  
if (rank==0) {  
    read(A); // T0, cost N  
    MPI_Send(A, N, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);  
    generate(B, A); // T1, cost 2N  
    process2(C, B); // T2, cost 2N^2  
    process4(E, C); // T4, cost N^2  
    MPI_Recv(D, N, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status);  
    res = process5(E, D); // T5, cost 2N  
    printf("Result: %f\n", res);  
}  
else {  
    MPI_Recv(A, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);  
    generate(B, A); // T1, cost 2N  
    process3(D, B); // T3, cost 2N^2  
    MPI_Send(D, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);  
}  
MPI_Finalize();
```

Ejercicio 20

c) Calcula el coste secuencial, coste paralelo, speed-up y eficiencia con 2 procesos.



$$t(n) = N + 2N + 2N^2 + N^2 + 2N + 2N^2 = 5N^2 + 5N \approx 5N^2 \text{ flops}$$

$$t_a(n, p) = N + 2N + 2N^2 + N^2 + 2N = 3N^2 + 5N \approx 3N^2 \text{ flops}$$

$$t_c(n, p) = 2(t_s + Nt_w)$$

$$t(n, p) = t_a(n, p) + t_c(n, p) = 3N^2 \text{ flops} + 2(t_s + Nt_w)$$

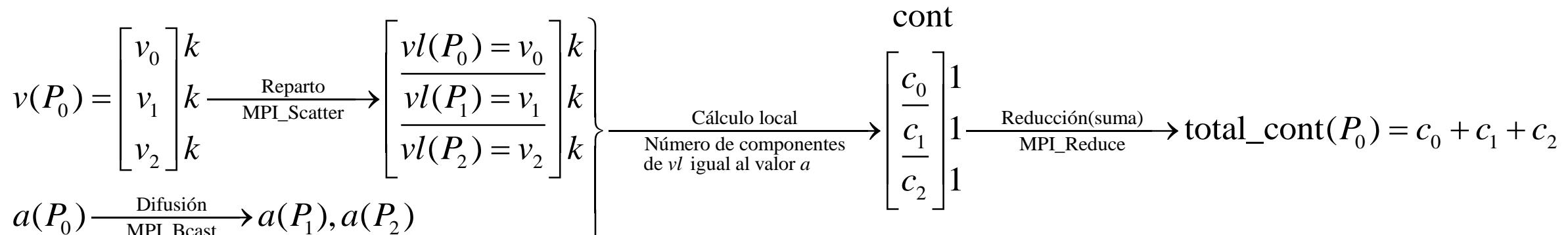
$$S(n, 2) = \frac{5N^2 \text{ flops}}{3N^2 \text{ flops} + 2(t_s + Nt_w)}$$

$$E(n, 2) = \frac{S(n, 2)}{2} = \frac{5N^2 \text{ flops}}{6N^2 \text{ flops} + 4(t_s + Nt_w)}$$

Ejercicio 21

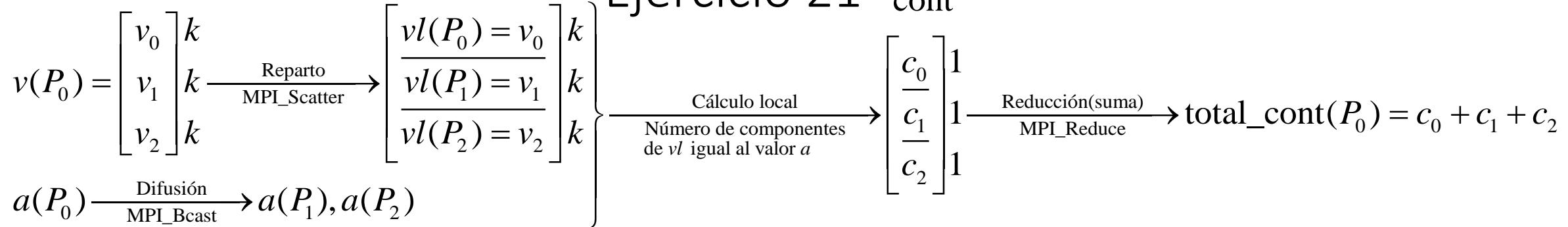
La siguiente función devuelve el número de componentes de un vector \mathbf{v} coincidentes con un entero a . Realiza una implementación MPI, suponiendo que inicialmente \mathbf{v} y a están inicialmente almacenados en la memoria local de P_0 . Calcular el coste de las comunicaciones.

```
int cont(int v[N], int a){  
    int cont=0, i;  
    for(i=0; i<N; i++)  
        if (v[i]==a)  
            cont++;  
    return cont;  
}
```



$$k = N / p \quad (p = \text{número de procesos})$$

Ejercicio 21 cont



int contp(int v[N], int a){ $k = N / p$ (p = número de procesos)

```

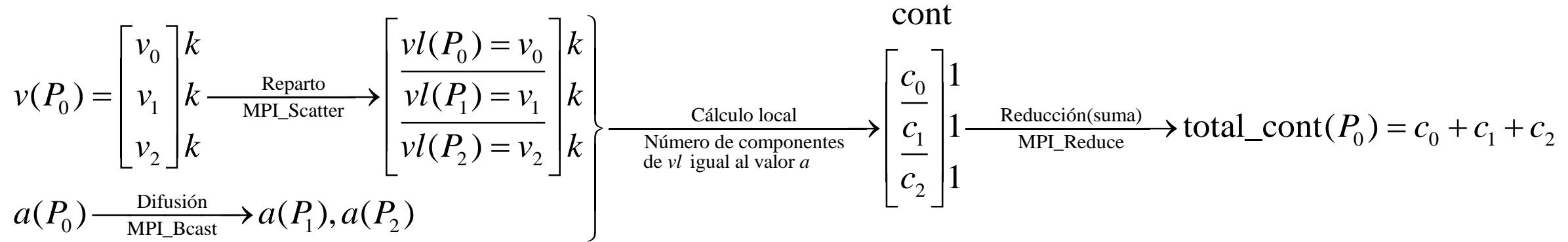
int p, id;
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
int k=N/p;
int cont=0, total_cont, i;
int *vl = (int*) malloc(sizeof(int)*k);
MPI_Scatter(v, k, MPI_INT, vl, k, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&a, 1, MPI_INT, 0, MPI_COMM_WORLD);
for(i=0; i<k; i++)
    if (vl[i]==a)
        cont++;
MPI_Reduce(&cont, &total_cont, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
return total_cont;
}
```

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

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

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

Ejercicio 21



$$k = N / p \quad (p = \text{número de procesos})$$

$$\left. \begin{array}{l} \text{MPI_Scatter(vector de dimensión } N\text{)} : t_{c_s} = (p-1) \left(t_s + \frac{N}{p} t_w \right) \\ \text{MPI_Bcast(dato simple)} : t_{c_g} = (p-1) (t_s + t_w) \\ \text{MPI_Reduce(dato simple)}: t_{c_r} = (p-1) (t_s + t_w) \end{array} \right\} t_c = (p-1) \left(3t_s + \left(\frac{N}{p} + 2 \right) t_w \right)$$

Ejercicio 22

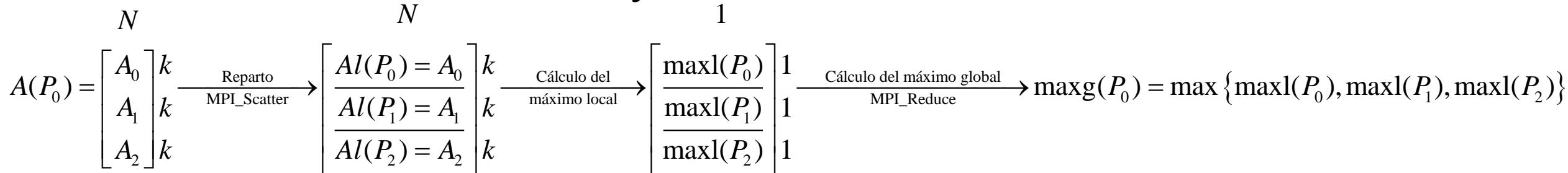
La siguiente función devuelve el valor máximo de una matriz de cuadrada de orden N.

```
int max_mat(int A[N][N]){
    int i, j;
    double maxi=A[0][0];
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            if (A[i][j]>maxi)
                maxi=A[i][j];
    return maxi;
}
```

Realiza una implementación MPI sabiendo que inicialmente el único proceso que tiene almacenada la matriz A es P0, en los siguientes casos:

- a) El valor máximo quede almacenado en P0.
- b) El valor máximo quede almacenado en todos los procesos.
- c) Calcula el tiempo de comunicaciones en los dos casos anteriores.

Ejercicio 22



a)

```

double contp(double A[N][N]){
    int p, id, i, j, k;
    double Al[N][N];
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&id);
    k=N/p;
    MPI_Scatter(A, k*N, MPI_DOUBLE, Al, k*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    int maxl=Al[0][0], maxg;
    for(i=0; i<k; i++)
        for(j=0; j<N; j++)
            if (Al[i][j]>maxl)
                maxl=Al[i][j];
    MPI_Reduce(&maxl, &maxg, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    return maxg;
}

```

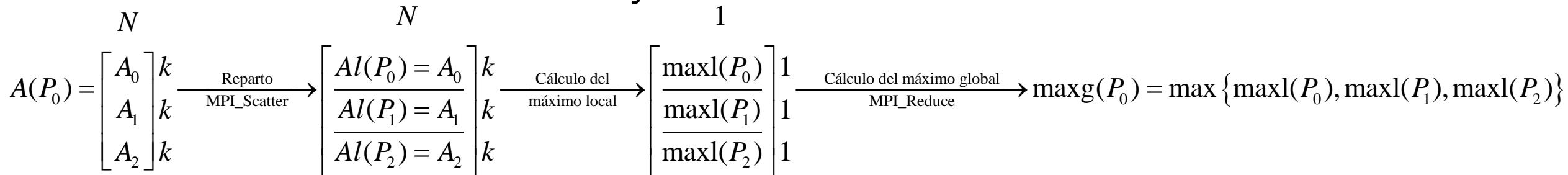
```

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

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

```

Ejercicio 22



b)

```

double contp(double A[N][N]){
    int p, id, i, j, k;
    double Al[N][N];
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    k=N/p;
    MPI_Scatter(A, k*N, MPI_DOUBLE, Al, k*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    int maxl=A[0][0], maxg;
    for(i=0; i<k; i++)
        for(j=0; j<N; j++)
            if (Al[i][j]>maxl)
                maxl=Al[i][j];
    MPI_Allreduce(&maxi, &maxg, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
    return maxg;
}

```

$$k = N / p \quad (p = \text{número de procesos})$$

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

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

MPI_Allreduce: mismos componentes que MPI_Reduce, salvo que no tiene el argumento `int root`

Ejercicio 22

$$A(P_0) = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \end{bmatrix} k \xrightarrow[\text{MPI_Scatter}]{\text{Reparto}} \begin{bmatrix} Al(P_0) = A_0 \\ Al(P_1) = A_1 \\ Al(P_2) = A_2 \end{bmatrix} k \xrightarrow[\text{máximo local}]{\text{Cálculo del}} \begin{bmatrix} \maxl(P_0) \\ \maxl(P_1) \\ \maxl(P_2) \end{bmatrix} 1 \xrightarrow[\text{MPI_Reduce}]{\text{Cálculo del máximo global}} \maxg(P_0) = \max \{ \maxl(P_0), \maxl(P_1), \maxl(P_2) \}$$

c)

$$k = N / p \quad (p = \text{número de procesos})$$

$$\left. \begin{aligned} \text{MPI_Scatter(matriz de dimensión } kN \text{)}: t_{c_s} &= (p-1) \left(t_s + \frac{N^2}{p} t_w \right) \\ \text{MPI_Reduce(dato simple)}: t_{c_r} &= (p-1)(t_s + t_w) \end{aligned} \right\} t_c = (p-1) \left(2t_s + \left(\frac{N^2}{p} + 1 \right) t_w \right)$$

Allreduce se puede realizar mediante una operación **reduce** sobre el proceso 0, seguida de un **broadcast** del resultado. Para el **broadcast**, suponemos que el proceso 0 envía un mensaje de un elemento a cada uno de los demás procesos:

$$\left. \begin{aligned} \text{MPI_Scatter(vector de dimensión } N \text{)}: t_{c_s} &= (p-1) \left(t_s + \frac{N^2}{p} t_w \right) \\ \text{MPI_Allreduce(dato simple)}: t_{c_r} &= 2(p-1)(t_s + t_w) \end{aligned} \right\} t_c = (p-1) \left(3t_s + \left(\frac{N^2}{p} + 2 \right) t_w \right)$$

Ejercicio 23

Sea **A** un array bidimensional de números reales de doble precisión, de dimensión $N \times N$. Define un tipo de datos derivado MPI que permita enviar una submatriz de tamaño 3×3 . Por ejemplo, la submatriz que empieza en $A[0][0]$ tendría los elementos marcados con x :

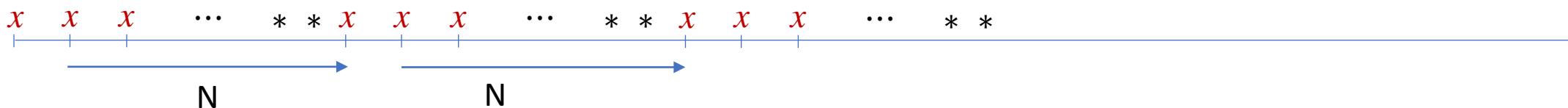
$$A = \begin{bmatrix} x & x & x & \cdot & \cdot & \cdot \\ x & x & x & \cdot & \cdot & \cdot \\ x & x & x & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

- Realiza las correspondientes llamadas para el envío desde P_0 y la recepción en P_1 del bloque de la figura anterior.
- Indica qué habría que modificar en el código anterior para que el bloque enviado por P_0 sea el que empieza en la posición $(0,3)$, y que se reciba en P_1 sobre el bloque que empieza en la posición $(3,0)$.

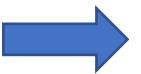
Ejercicio 23

$$A = \begin{bmatrix} x & x & x & \cdots & * & * \\ x & x & x & \cdots & * & * \\ x & x & x & \cdots & * & * \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ * & * & * & \cdots & * & * \\ * & * & * & \cdots & * & * \end{bmatrix}$$

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



count=3
blocklength=3
stride=N



```
MPI_Datatype newtype;  
MPI_Type_vector(3, 3, N, MPI_DOUBLE, &newtype);  
MPI_Type_commit(&newtype);
```

a) Realiza las correspondientes llamadas para el envío desde P0 y la recepción en P1 del bloque de la figura anterior

```
double A[N][N];
int rank;
MPI_Datatype newtype;
... /* la matriz es rellenada*/
MPI_Type_vector(3, 3, N, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) {
    MPI_Send(&A[0][0], 1, newtype, 1, 0, MPI_COMM_WORLD);
}
else if (rank==1) {
    MPI_Recv(&A[0][0], 1, newtype, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
MPI_Type_free(&newtype);
```

$$A = \begin{bmatrix} x & x & x & \cdot & \cdot & \cdot \\ x & x & x & \cdot & \cdot & \cdot \\ x & x & x & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

b) Indica qué habría que modificar en el código anterior para que el bloque enviado por P0 sea el que empieza en la posición (0,3), y que se reciba en P1 sobre el bloque que empieza en la posición (3,0).

```
double A[N][N];
int rank;
MPI_Datatype newtype;
... /* la matriz es rellenada*/
MPI_Type_vector(3, 3, N, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) {
    MPI_Send(&A[0][3], 1, newtype, 1, 0, MPI_COMM_WORLD);
}
else if (rank==1) {
    MPI_Recv(&A[3][0], 1, newtype, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
MPI_Type_free(&newtype);
```

Ejercicio 24

Dado el siguiente fragmento de un programa MPI:

```
struct Tdatos {  
    int x;  
    int y[n];  
    double a[n];  
};  
void distribuye(struct Tdatos *datos, int n, MPI_Comm comm) {  
    int p, id, i;  
    MPI_Status status;  
    MPI_Comm_size(comm, &p);  
    MPI_Comm_rank(comm, &id);  
    if (id==0) {  
        for (i=1; i<p; i++) {  
            MPI_Send(&(datos->x), 1, MPI_INT, i, 0, comm);  
            MPI_Send(&(datos->y[0]), n, MPI_INT, i, 0, comm);  
            MPI_Send(&(datos->a[0]), n, MPI_DOUBLE, i, 0, comm);  
        }  
    }  
    else {  
        MPI_Recv(&(datos->x), 1, MPI_INT, 0, 0, comm, &status);  
        MPI_Recv(&(datos->y[0]), n, MPI_INT, 0, 0, comm, &status);  
        MPI_Recv(&(datos->a[0]), n, MPI_DOUBLE, 0, 0, comm, &status);  
    }  
}
```

Modificar la función distribuye_datos para optimizar las comunicaciones.

```

void distribuye(struct Tdatos *datos, int n, MPI_Comm comm) {
    int p, id, i;
    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &id);
    MPI_Status status;
    int count=3;
    int longitudes[]={1,n,n};
    MPI_Aint despls[3];
    MPI_Aint dir, dirx, diry, dira;
    /* Calculo de los desplazamientos de cada componente */
    MPI_Get_address(datos, &dir);
    MPI_Get_address(&(datos->x), &dirx);
    MPI_Get_address(&(datos->y[0]), &diry);
    MPI_Get_address(&(datos->a[0]), &dira);
    despls[0]=dirx-dir;
    despls[1]=diry-dir;
    despls[2]=dira-dir;
    MPI_Datatype tipos[]={MPI_INT, MPI_INT, MPI_DOUBLE};
    MPI_Type_create_struct(count, longitudes, despls, tipos, &Tnuevo);
    MPI_Type_commit(&Tnuevo);
    If (id==0)
        for (i=1; i<p; i++)
            MPI_Send(datos, 1, Tnuevo, i, 0, comm);
    else MPI_Recv(datos, 1, Tnuevo, 0, 0, comm, &status);
    MPI_Type_free(&Tnuevo);
}

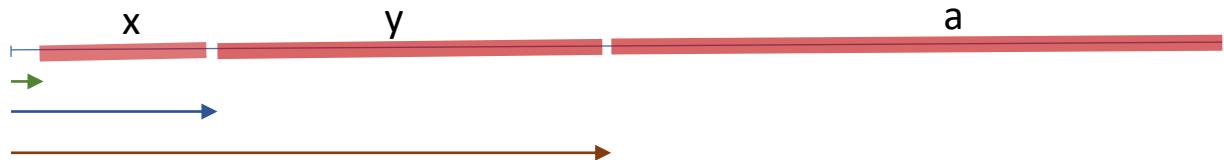
```

```

int ^MPI_Type_create_struct(int count,
                            int blocklens[], MPI_Aint indices[],
                            MPI_Datatype old_types[], MPI_Datatype
                            *newtype)

struct Tdatos {
    int x;
    int y[n];
    double a[n];
};

```



Reparto por bloques de un conjunto de índices

	k=12/3=4				k=12/3=4				k=12/3=4			
Indice global (i_g)	0	1	2	3	4	5	6	7	8	9	10	11
Indice proceso (i_p)	0	0	0	0	1	1	1	1	2	2	2	2
Indice local (i_l)	0	1	2	3	0	1	2	3	0	1	2	3

N= número de índices (12)

p= número de procesos (3)

k=N/p= nº de índices que le corresponden a cada proceso (4)

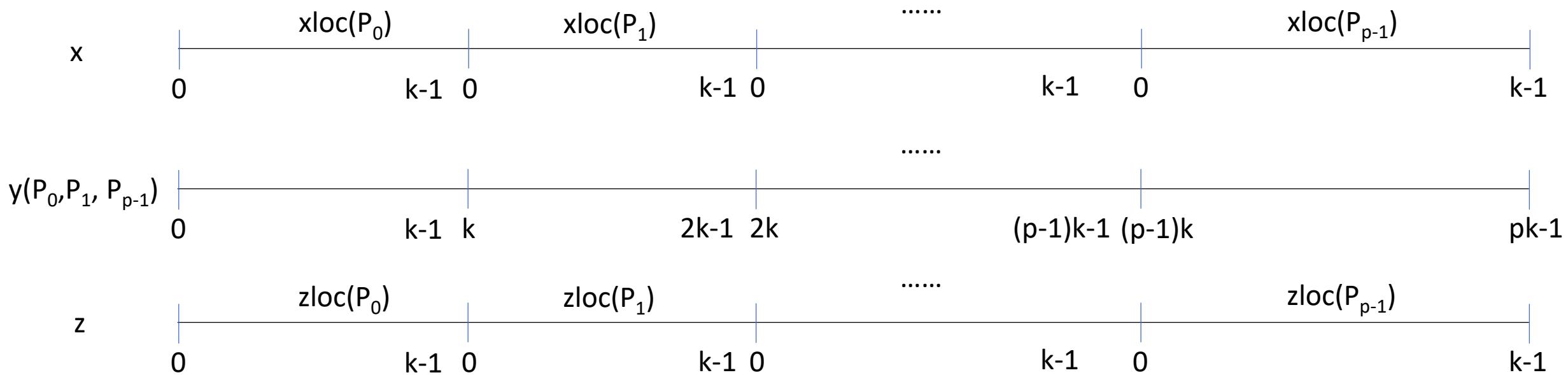
$$\begin{array}{ccc} i_g & \lfloor k & \leftrightarrow \\ & \quad \quad \quad & \\ i_l & i_p & \end{array}$$
$$\begin{aligned} i_g &= ki_p + i_l \\ i_p &= i / k \\ i_l &= i \% k \end{aligned}$$

Ejercicio 25

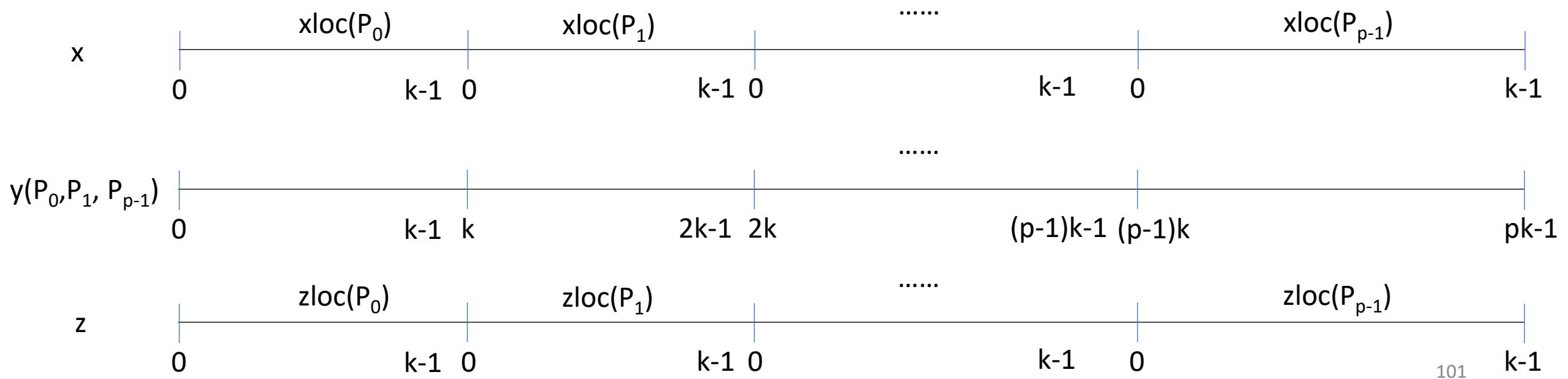
- Supongamos en un programa que un vector **x** de dimensión **N** se encuentra repartido por bloques entre todos los procesos mediante vectores locales **xloc** y un vector **y** replicado en todos los procesos.
- Implementa la función :

```
void suma(double xloc[], double y[], double zloc[], int n, int p, int ip)
```

la cual debe obtener un vector $z = x + y$ repartido en vectores locales **zloc**
- Supongamos que **N** es divisible entre **p**.



Ejercicio 25



$$i_g = k i_p + i_l$$

```
void suma(double xloc[], double y[], double zloc[], int N, int p, int ip) {  
    /*N= nº de componentes de los vectores globales, p=nº de procesos, ip: índice del proceso*/  
    int il, k=N/p;  
    for (il=0; il<k; il++)  
        zloc[il]=xloc[il]+y[k*ip+il];  
}
```

Reparto cíclico de un conjunto de índices

Indice global (i_g)	0	1	2	3	4	5	6	7	8	9	10	11
Indice proceso (i_p)	0	1	2	0	1	2	0	1	2	0	1	2
Indice local (i_l)	0	0	0	1	1	1	2	2	2	3	3	3

N= número de índices (12)

p= número de procesos (3)

k= N/p=nº de índices que le corresponden a cada proceso (4)

$$\begin{array}{ccc} i_g & \boxed{p} & \leftrightarrow \\ i_p & i_l & \end{array}$$
$$i_g = pi_l + i_p$$
$$i_p = i \% p$$
$$i_l = i / p$$

Ejercicio 25

- Supongamos en un programa que un vector **x** de dimensión **N** se encuentra **repartido cícicamente** entre todos los procesos mediante vectores locales **xloc** y un vector **y** replicado en todos los procesos.

- Implementa la función :

```
void suma(double xloc[], double y[], double zloc[], int n, int p, int ip)
```

la cual debe obtener un vector $z=x+y$ con un **reparto cíclico** en vectores locales **zloc**

- Supongamos que **N** es divisible entre **p**.

.....

$$i_g = pi_l + i_p$$

```
void suma(double xloc[], double y[], double zloc[], int N, int p, int ip) {  
    /* n= nº de componentes de los vectores globales, p=nº de procesos, ip= índice del proceso*/  
    int il, k=N/p;  
    for (il=0; il<k; il++)  
        zloc[il]=xloc[il]+y[p*il+ip];  
}
```

Ejercicio 26

- Sea la función :

```
void prodmv(double **A, double *x, double *y, int n, int p, int ip)
```

la cual debe obtener el vector $y = Ax$, siendo **n** la dimensión de A, x e y, **p** el número de procesos (**p** divide a **n**) e **ip** el índice de proceso. Implementa dicha función en los siguientes casos:

- a) La matriz A tiene un reparto por bloques de filas e y debe estar repartido por bloques
- b) La matriz A tiene un reparto cíclico de filas e y debe tener un reparto cíclico de filas

Implementación secuencial:

```
void prodmv(double **A, double *x, double *y, int n){  
    int i, j;  
    for(i=0; i<n; i++) {  
        y[i]=0.0;  
        for(j=0; i<n; j++)  
            y[i]+=A[i][j]*x[j];  
    }  
}
```

Implementación a)

```
void prodmv(double **A, double *x, double *y, int n, int p, int ip){  
    int i, j;  
    k=n/p;  
    for(i=0; i<n; i++) {  
        if(ip==i/k){/* ip tiene la fila i/k de A*/}  
            y[i%k]=0.0;  
        for(j=0; i<n; j++)  
            y[i%k]+=A[i%k][j]*x[j];  
    }  
}
```

Reparto por bloques:

$$i = ki_p + i_l$$

$$i_p = i / k$$

$$i_l = i \% k$$

Implementación secuencial:

```
void prodmv(double **A, double *x, double *y, int n){  
    int i, j;  
    for(i=0; i<n; i++) {  
        y[i]=0.0;  
        for(j=0; i<n; j++)  
            y[i]+=A[i][j]*x[j];  
    }  
}
```

Implementación b)

```
void prodmv(double **A, double *x, double *y, int n, int p, int ip){  
    int i, j;  
    k=n/p;  
    for(i=0; i<n; i++) {  
        if(ip==i%p){/* ip tiene la fila i/k de A*/}  
            y[i/p]=0.0;  
        for(j=0; i<n; j++)  
            y[i/p]+=A[i/p][j]*x[j];  
    }  
}
```

Reparto cíclico:

$$i = pi_l + i_p$$

$$i_p = i \% p$$

$$i_l = i / p$$

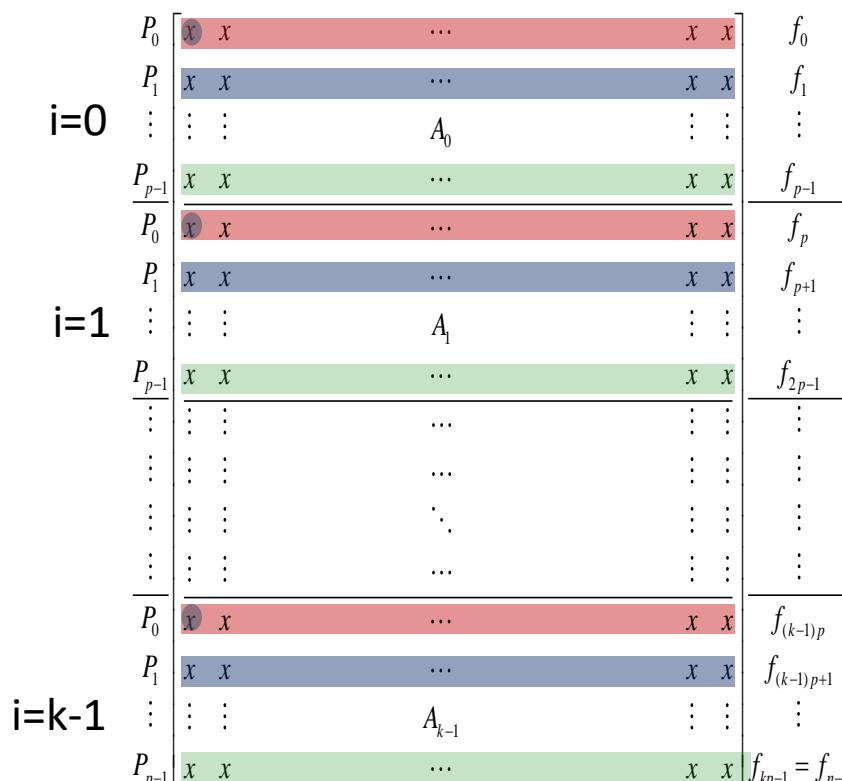
Ejercicio 27

Dada una matriz cuadrada \mathbf{A} de dimensión n almacenada en el proceso P0, implementa el código necesario para que \mathbf{A} sea repartida cíclicamente por filas entre todos los procesos, almacenándolas en las matrices locales \mathbf{A}_l

- a) usando MPI_Scatter (como en la sesión 4 de la práctica 3)
- b) usando MPI_Type_vector

calculando en ambos casos el tiempo de comunicaciones ¿Cuál tendrá menor tiempo paralelo?

$$\mathbf{A}(P_0) \in \mathbb{R}^{n \times n}$$



Ejercicio 27-a)

Para i=0 hasta k-1

Reparto de A_i (MPI_Scatter): una fila para cada matriz A_i de cada uno de los procesos

Fin para

$$A(P_0) \in \mathbb{R}^{n \times n}$$

$i=0$	P_0	$\begin{bmatrix} x & x & \cdots & x & x \end{bmatrix}$	f_0
	P_1	$\begin{bmatrix} x & x & \cdots & x & x \end{bmatrix}$	f_1
	\vdots	A_0	\vdots
	P_{p-1}	$\begin{bmatrix} x & x & \cdots & x & x \end{bmatrix}$	f_{p-1}
	P_0	$\begin{bmatrix} x & x & \cdots & x & x \end{bmatrix}$	f_p
$i=1$	P_1	$\begin{bmatrix} x & x & \cdots & x & x \end{bmatrix}$	f_{p+1}
	\vdots	A_1	\vdots
	P_{p-1}	$\begin{bmatrix} x & x & \cdots & x & x \end{bmatrix}$	f_{2p-1}
	\vdots	\ddots	\vdots
	\vdots	\ddots	\vdots
	\vdots	\ddots	\vdots
	P_0	$\begin{bmatrix} x & x & \cdots & x & x \end{bmatrix}$	$f_{(k-1)p}$
$i=k-1$	P_1	$\begin{bmatrix} x & x & \cdots & x & x \end{bmatrix}$	$f_{(k-1)p+1}$
	\vdots	A_{k-1}	\vdots
	P_{p-1}	$\begin{bmatrix} x & x & \cdots & x & x \end{bmatrix}$	$f_{kp-1} = f_{n-1}$



$$Al(P_0) = \begin{bmatrix} x & x & \cdots & x \\ x & x & \cdots & x \\ \vdots & \vdots & \ddots & \vdots \\ x & x & \cdots & x \end{bmatrix} \quad \bar{f}_0 = f_0 \\ \bar{f}_1 = f_p \\ \vdots \\ \bar{f}_{k-1} = f_{(k-1)p}$$

$$Al(P_1) = \begin{bmatrix} x & x & \cdots & x \\ x & x & \cdots & x \\ \vdots & \vdots & \ddots & \vdots \\ x & x & \cdots & x \end{bmatrix} \quad \bar{f}_0 = f_1 \\ \bar{f}_1 = f_{p+1} \\ \vdots \\ \bar{f}_{k-1} = f_{(k-1)p+1}$$

$$Al(P_{p-1}) = \begin{bmatrix} x & x & \cdots & x \\ x & x & \cdots & x \\ \vdots & \vdots & \ddots & \vdots \\ x & x & \cdots & x \end{bmatrix} \quad \bar{f}_0 = f_{p-1} \\ \bar{f}_1 = f_{2p-1} \\ \vdots \\ \bar{f}_{k-1} = f_{kp-1}$$

Ejercicio 27-a)

$$A \in \mathbb{R}^{n \times n}$$

$i=0$

$$\begin{array}{c|ccccc|c}
P_0 & x & x & \cdots & & x & x & f_0 \\
P_1 & x & x & \cdots & & x & x & f_1 \\
\vdots & \vdots & \vdots & & A_0 & & \vdots & \vdots \\
P_{p-1} & x & x & \cdots & & x & x & f_{p-1} \\
\hline P_0 & x & x & \cdots & & x & x & f_p \\
P_1 & x & x & \cdots & & x & x & f_{p+1} \\
\vdots & \vdots & \vdots & & A_1 & & \vdots & \vdots \\
P_{p-1} & x & x & \cdots & & x & x & f_{2p-1} \\
\vdots & \vdots & \vdots & & \cdots & & \vdots & \vdots \\
\vdots & \vdots & \vdots & & \ddots & & \vdots & \vdots \\
\vdots & \vdots & \vdots & & \cdots & & \vdots & \vdots \\
\hline P_0 & x & x & \cdots & & x & x & f_{(k-1)p} \\
P_1 & x & x & \cdots & & x & x & f_{(k-1)p+1} \\
\vdots & \vdots & \vdots & & A_{k-1} & & \vdots & \vdots \\
P_{p-1} & x & x & \cdots & & x & x & f_{kp-1} = f_{n-1}
\end{array}$$

$Al(P_0) = \begin{bmatrix} x & x & \cdots & x \\ x & x & \cdots & x \\ \vdots & \vdots & \ddots & \vdots \\ x & x & \cdots & x \end{bmatrix}$ $\bar{f}_0 = f_0$
 $\bar{f}_1 = f_p$
 \vdots
 $\bar{f}_{k-1} = f_{(k-1)p}$

$Al(P_1) = \begin{bmatrix} x & x & \cdots & x \\ x & x & \cdots & x \\ \vdots & \vdots & \ddots & \vdots \\ x & x & \cdots & x \end{bmatrix}$ $\bar{f}_0 = f_1$
 $\bar{f}_1 = f_{p+1}$
 \vdots
 $\bar{f}_{k-1} = f_{(k-1)p+1}$

\vdots

$Al(P_{p-1}) = \begin{bmatrix} x & x & \cdots & x \\ x & x & \cdots & x \\ \vdots & \vdots & \ddots & \vdots \\ x & x & \cdots & x \end{bmatrix}$ $\bar{f}_0 = f_{p-1}$
 $\bar{f}_1 = f_{2p-1}$
 \vdots
 $\bar{f}_{k-1} = f_{kp-1}$

Para $i=0$ hasta $k-1$
 Reparto de A_i
 Fin para

i	Primer elemento de A	Primer elemento de Al
0	$A[0][0]$	$Al[0][0]$
1	$A[p][0]$	$Al[1][0]$
2	$A[2*p][0]$	$Al[2][0]$
...
$k-1$	$A[(k-1)*p][0]$	$Al[k-1][0]$

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

Ejercicio 27-a)

$$A \in \mathbb{R}^{n \times n}$$

P_0		\dots	$x \quad x$	$f_0 \rightarrow \bar{f}_0$
P_1		\dots	$x \quad x$	f_1
\vdots	$\vdots \vdots$	A_0	$\vdots \vdots$	\vdots
P_{p-1}		\dots	$x \quad x$	$f_{p-1} \rightarrow \bar{f}_1$
P_0		\dots	$x \quad x$	$f_p \rightarrow \bar{f}_1$
P_1		\dots	$x \quad x$	f_{p+1}
\vdots	$\vdots \vdots$	A_1	$\vdots \vdots$	\vdots
P_{p-1}		\dots	$x \quad x$	$f_{2p-1} \rightarrow \bar{f}_1$
\vdots	$\vdots \vdots$	\cdots	$\vdots \vdots$	\vdots
\vdots	$\vdots \vdots$	\cdots	$\vdots \vdots$	\vdots
\vdots	$\vdots \vdots$	\ddots	$\vdots \vdots$	\vdots
\vdots	$\vdots \vdots$	\cdots	$\vdots \vdots$	\vdots
P_0		\dots	$x \quad x$	$f_{(k-1)p} \rightarrow \bar{f}_{k-1}$
P_1		\dots	$x \quad x$	$f_{(k-1)p+1}$
\vdots	$\vdots \vdots$	A_{k-1}	$\vdots \vdots$	\vdots
P_{p-1}		\dots	$x \quad x$	$f_{kp-1} = f_{n-1}$

f_i fila i-ésima de la matriz A

\bar{f}_i fila i-ésima de la matriz Al

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

i	Primer elemento de A	Primer elemento de Al
0	$A[0][0]$	$Al[0][0]$
1	$A[p][0]$	$Al[1][0]$
2	$A[2*p][0]$	$Al[2][0]$
...
$k-1$	$A[(k-1)*p][0]$	$Al[k-1][0]$

```
double A[n][n], Al[n][n];
Int i, p, k;
MPI_Comm_size(MPI_COMM_WORLD, &p);
k=n/p;
for(i=0; i<k; i++)
    MPI_Scatter(&A[i*p][0], n, MPI_DOUBLE, &Al[i][0], n,
                MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

$$t_c = k(p-1) \left(t_s + nt_w \right) = \frac{n(p-1)}{p} \left(t_s + nt_w \right)$$

Ejercicio 27-b)

count=k=n/p blocklength=n stride=p*n

P_0		f_0
P_1		f_1
\vdots	$\vdots \vdots$	$\vdots \vdots$
P_{p-1}		f_{p-1}
$\frac{P_{p-1}}{P_0}$		f_p
P_1		f_{p+1}
\vdots	$\vdots \vdots$	$\vdots \vdots$
P_{p-1}		f_{2p-1}
\vdots	$\vdots \vdots$	$\vdots \vdots$
\vdots	$\vdots \vdots$	$\vdots \vdots$
\vdots	$\vdots \vdots$	$\vdots \vdots$
\vdots	$\vdots \vdots$	$\vdots \vdots$
P_0		$f_{(k-1)p}$
P_1		$f_{(k-1)p+1}$
\vdots	$\vdots \vdots$	$\vdots \vdots$
P_{p-1}		$f_{kp-1} = f_{n-1}$

Al crear el nuevo tipo de datos, en un solo mensaje
se envían los datos del mismo color

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

```
double A[n][n], AI[n][n];
Int i, p, ip, k;
MPI_Status stat;
MPI_Comm_size(MPI_COMM_WORLD,&p);
MPI_Comm_rank(MPI_COMM_WORLD,&ip);
k=n/p;
MPI_Datatype ntype;
MPI_Type_vector(k, n, p*n, MPI_DOUBLE, &ntype);
MPI_Type_commit(&ntype);
if (ip==0)
    for(i=1; i<p; i++)
        MPI_Send(&A[i][0], 1, ntype, i, 100, MPI_COMM_WORLD);
    MPI_Sendrecv(A, 1, ntype, 0, 100, AI, k*n, MPI_DOUBLE, 0, 100,
                MPI_COMM_WORLD, &stat);
else
    MPI_Recv(AI, k*n, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD, &stat);
}
MPI_Type_free(&newtype);
```

$$t_c = (p-1) \left(t_s + \frac{n^2}{p} t_w \right)$$

Ejercicio 27-b)

$$\text{MPI_Scatter: } t_c = \frac{n(p-1)}{p} (t_s + nt_w) = \frac{n(p-1)}{p} t_s + \frac{n^2(p-1)}{p} t_w$$

$$\text{Type_Vector: } t_c = (p-1) \left(t_s + \frac{n^2}{p} t_w \right) = (p-1)t_s + \frac{n^2(p-1)}{p} t_w$$

- Las dos distribuciones tienen el mismo coeficiente de t_w
- Veamos qué ocurre con los coeficientes de t_s

$$\frac{n(p-1)}{p} > (p-1) \Leftrightarrow n > p$$

- Luego si $n > p$, entonces el tiempo paralelo es mayor para el reparto mediante MPI_Scatter

Ejercicio 28

- En un programa paralelo se dispone de un vector distribuido por bloques entre los procesos, de manera que cada proceso guarda su bloque en el array **vloc**.
- Implementa una función que desplace los elementos del vector una posición a la derecha, haciendo además que el último elemento pase a ocupar la primera posición. Por ejemplo, si tenemos 3 procesos, dado el estado inicial:

	P_0	P_1	P_2
vloc	[2 5 3]	[7 1 0]	[6 4 9]

El estado final sería:

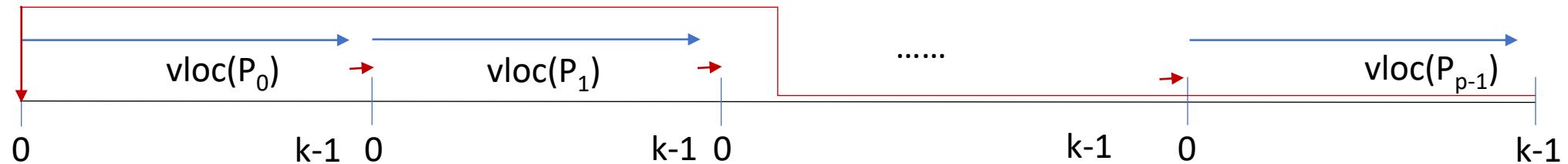
	P_0	P_1	P_2
vloc	[9 2 5]	[3 7 1]	[0 6 4]

La función deberá evitar los posibles interbloqueos. La cabecera de la función será:

```
void desplazar(double vloc[], int k)
```

donde **k** es el número de elementos de **vloc** (supondremos que $k > 1$).

Ejercicio 28

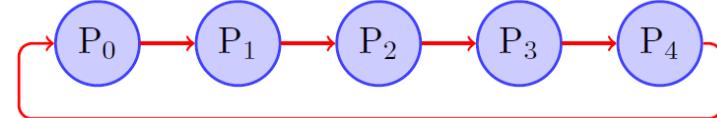


```
void desplazar(double vloc[], int mb){  
    int prev, sig, p, rank, i;  
    double elem;  
    MPI_Status stat;  
    MPI_Comm_size(MPI_COMM_WORLD, &p);  
    MPI_Comm_rank(MPI_COMM_WORLD, &ip);  
    if (ip==p-1) sig = 0;  
    else sig = ip+1;  
    if (ip==0) prev = p-1;  
    else prev = ip-1;  
    /* Desplazamiento local */  
    elem = vloc[k-1];  
    for (i=k-1; i>0; i--)  
        vloc[i] = vloc[i-1];  
    /* Comunicación con los vecinos */  
    MPI_Sendrecv(&elem, 1, MPI_DOUBLE, sig, 0, &vloc[0], 1, MPI_DOUBLE, prev, 0, MPI_COMM_WORLD, &stat);  
}
```

Ver transparencia 27 de S3:

Ejemplo – Desplazamiento en Anillo

En el caso del anillo, todos los procesos han de enviar y recibir



Se puede producir bloqueo en el caso de un envío de tipo síncrono

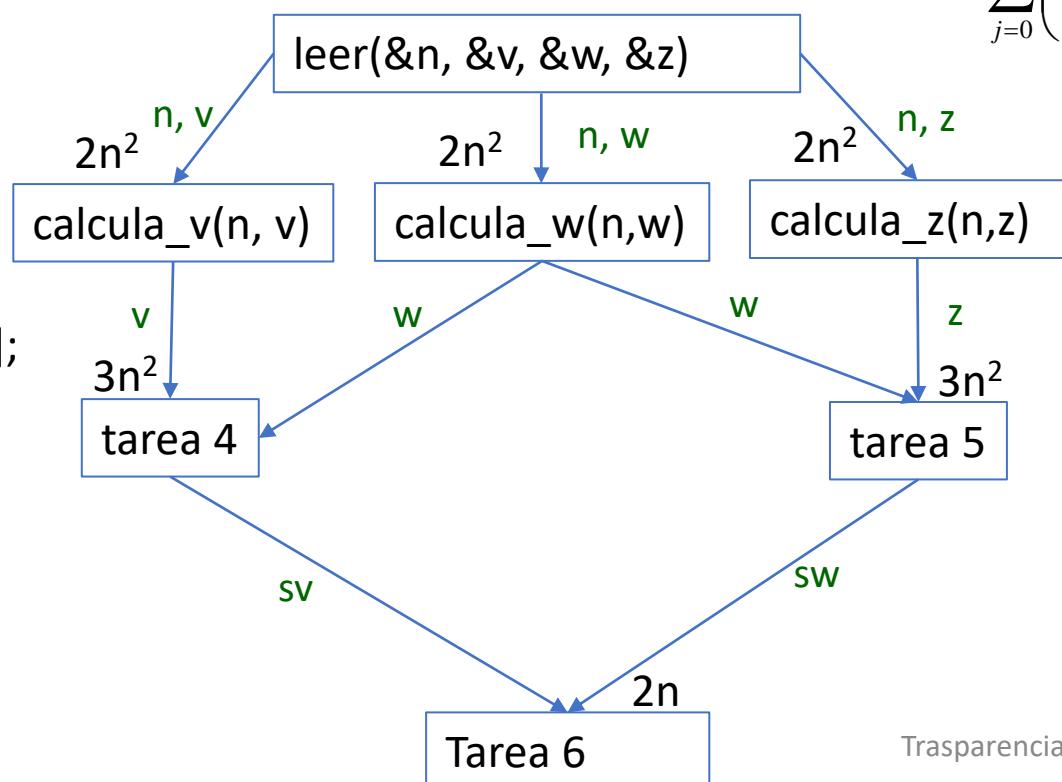
Ejercicio 29

Sea la siguiente función:

```
double funcion(){  
    int i, n, j;  
    double *v, *w, *z, sv, sw, x, res=0.0;  
    leer(&n, &v, &w, &z);  
    calcula_v(n, v); /* tarea 1 */  
    calcula_w(n, w); /* tarea 2 */  
    calcula_z(n,z); /* tarea 3 */  
    for (j=0; j<n; j++) /* tarea 4 */  
        sv = 0;  
        for (i=0; i<n; i++) sv = sv + v[i]*w[i];  
        for (i=0; i<n; i++) v[i]=sv*v[i];  
    }  
    for (j=0; j<n; j++) /* tarea 5 */  
        sw = 0;  
        for (i=0; i<n; i++) sw = sw + w[i]*z[i];  
        for (i=0; i<n; i++) z[i]=sw*z[i];  
    }  
    /* tarea 6 */  
    x = sv+sw;  
    for (i=0; i<n; i++) res = res+x*z[i];  
return res;  
}
```

Las funciones `calcula_X` tienen como entrada los datos que reciben como argumentos y con ellos modifican el vector X indicado. Por ejemplo, `calcula_v(n,v)` toma como datos de entrada los valores de n y v y modifica el vector v .

- a) Dibuja el grafo de dependencias de las diferentes tareas, incluyendo en el mismo el coste de cada una de las tareas y el volumen de las comunicaciones. Suponer que las funciones `calcula_X` tienen un coste de $2n^2$.



Costes tareas T4 y T5:

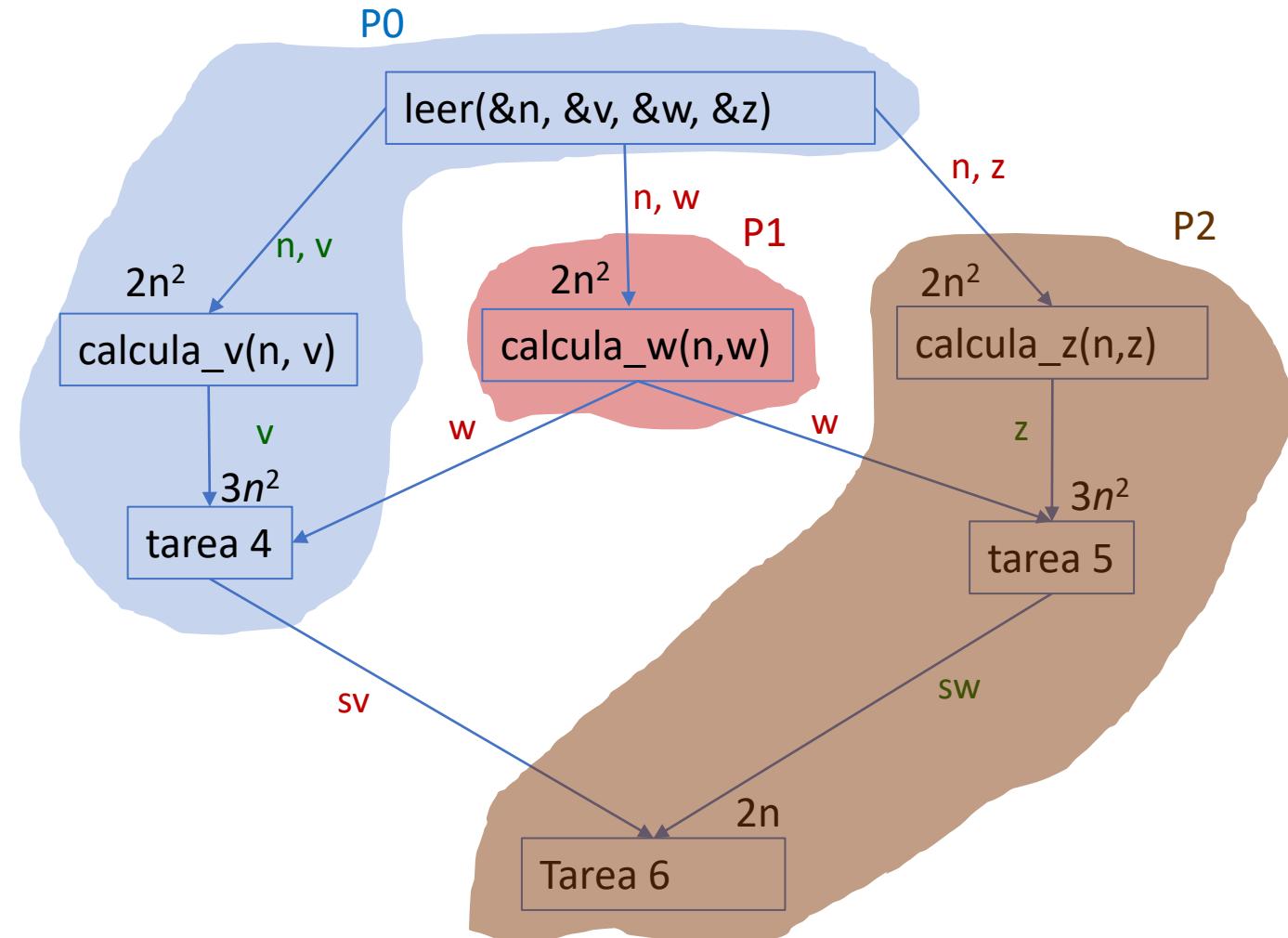
$$\sum_{j=0}^{n-1} \left(\sum_{i=0}^{n-1} 2 + \sum_{i=0}^{n-1} 1 \right) = \sum_{j=0}^{n-1} (2n + n) = 3n^2 \text{ flops}$$

Costes tarea T6:

$$1 + \sum_{i=0}^{n-1} 2 \approx 2n \text{ flops}$$

Ejercicio 29

Paralelízalo usando MPI, de forma que los procesos MPI disponibles ejecutan las diferentes tareas. Se puede suponer que hay al menos 3 procesos. Supondremos que solo un proceso devuelve el resultado correcto.



P0:
 leer(&n, &v, &w, &z)
 Send(n, P1)
 Send(n, P2)
 Send(w, P1)
 Send(z, P2)
 calcula_v(n, v)
 Recv(w, P1)
 tarea 4
 Send(sv, P2)

P1:
 Recv(n, P0)
 Recv(w, P0)
 calcula_w(n, w)
 Send(w, P0)
 Send(w, P2)

P2:
 Recv(n, P0)
 Recv(z, P0)
 calcula_z(n, z)
 Recv(w, P1)
 Tarea 5
 Recv(sv, P0)
 tarea 6

Ejercicio 29

P0:

```
leer(&n, &v, &w, &z)
Send(n, P1)
Send(n, P2)
Send(w, P1)
Send(z, P2)
calcula_v(n, v)
Recv(w,P1)
tarea 4
Send(sv, P2)
```

```
double funcion(){
    int i,n,j;
    double *v,*w,*z,sv,sw,x,res=0.0;
    int p,rank;
    MPI_Status st;
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if (rank==0) {
        leer(&n, &v, &w, &z);
        MPI_Send(&n, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Send(&n, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
        MPI_Send(w, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
        MPI_Send(z, n, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD);
        calcula_v(n,v); /* tarea 1 */
        MPI_Recv(w, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &st);
        /* tarea 4 (mismo codigo del caso secuencial) */
        ....
        MPI_Send(&sv, 1, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD);
    }
}
```

Ejercicio 29

P1:

```
Recv(n,P0)
Recv(w,P0)
calcula_w(n,w)
Send(w, P0)
Send(w, P2)
```

```
else if (rank==1) {
    MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &st);
    MPI_Recv(w, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &st);
    calcula_w(n,w); /* tarea 2 */
    MPI_Send(w, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    MPI_Send(w, n, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD);
}
```

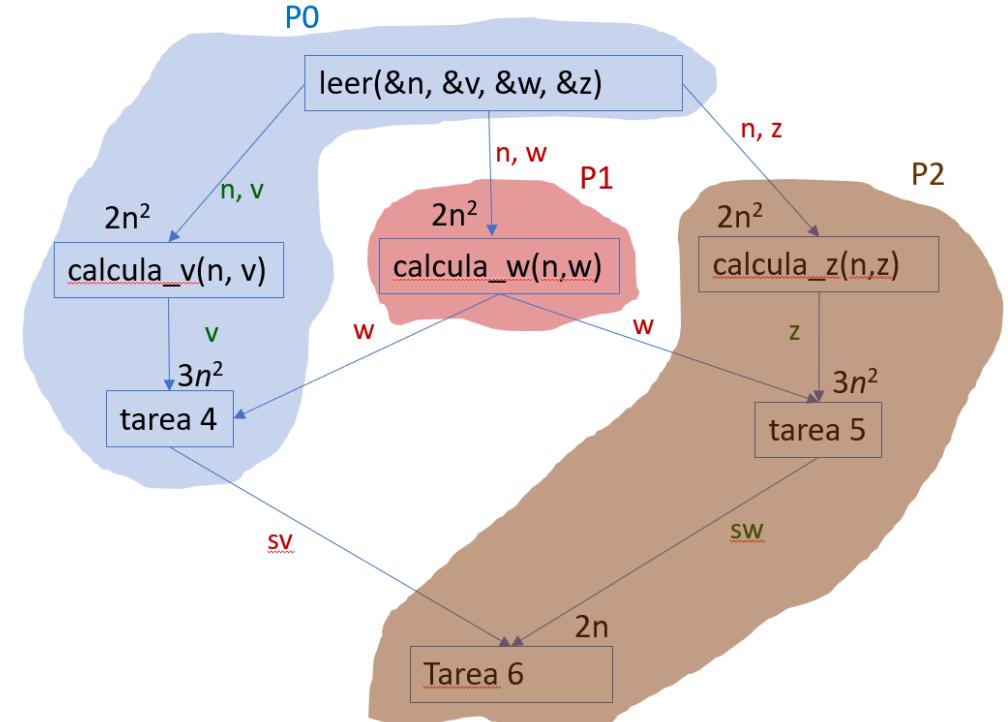
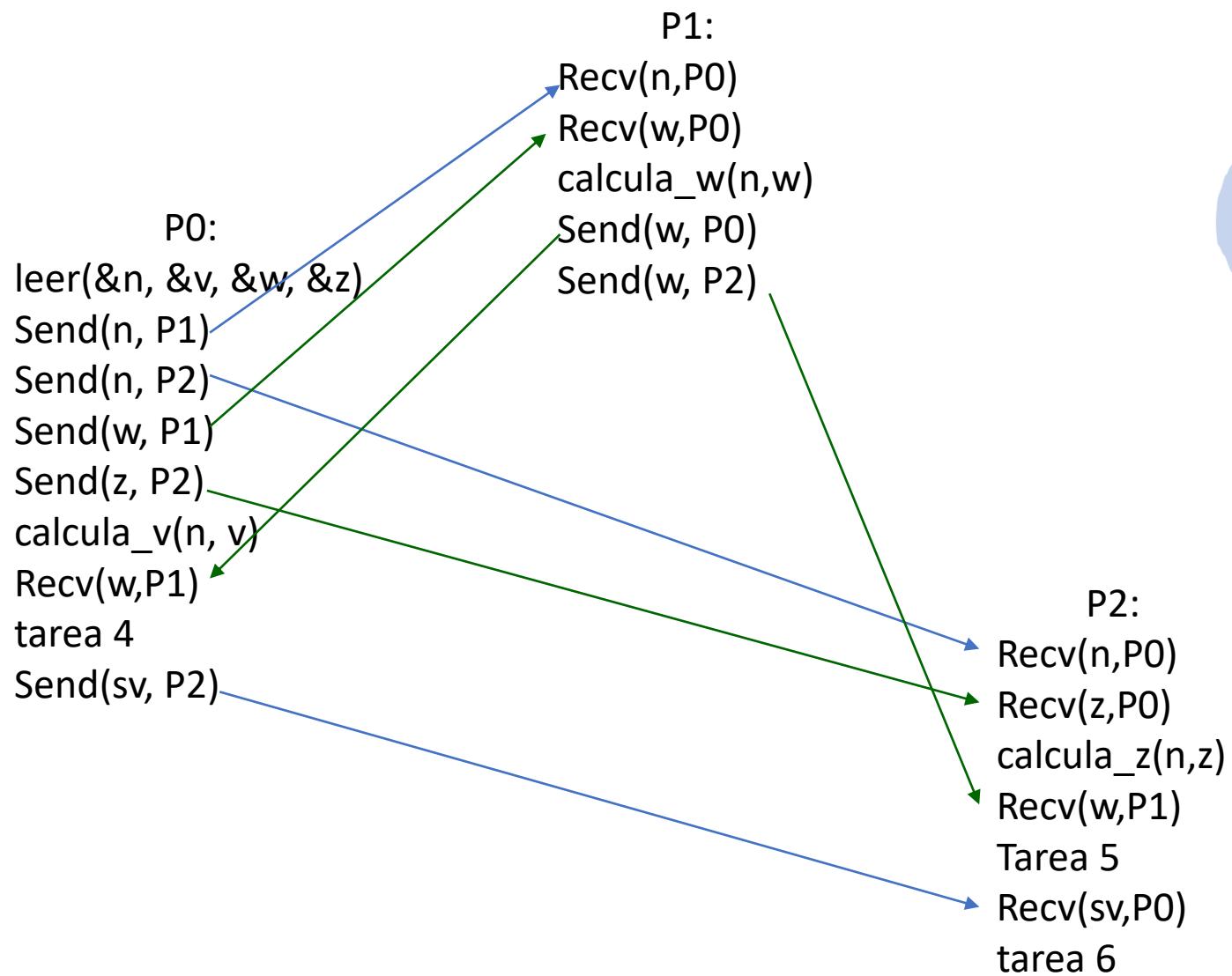
Ejercicio 29

P2:

Recv(n,P0)
Recv(z,P0)
calcula_z(n,z)
Recv(w,P1)
Tarea 5
Recv(sv,P0)
tarea 6

```
else if (rank==2) {  
    MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &st);  
    MPI_Recv(z, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &st);  
    calcula_z(n,z); /* tarea 3 */  
    MPI_Recv(w, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &st);  
    /* tarea 5 (mismo codigo del caso secuencial) */  
    ....  
    MPI_Recv(&sv, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,&st);  
    /* tarea 6 (mismo codigo del caso secuencial) */  
    ....  
}  
return res;  
}
```

Ejercicio 29



$$t(n) \approx 3 \cdot 2n^2 + 2 \cdot 3n^2 + 2n \approx 12n^2 \text{ flops}$$

$$t_a(n, 3) \approx 2n^2 + 3n^2 + 2n \approx 5n^2 \text{ flops}$$

$$t_c(n, 3) = 3(t_s + t_w) + 3(t_s + nt_w) \approx 6t_s + 3nt_w$$

$$t(n, 3) = t_a(n, 3) + t_c(n, 3) \approx 5n^2 \text{ flops} + 6t_s + 3nt_w$$

$$S(n, 3) = \frac{t(n)}{t(n, 3)} = \frac{12n^2 \text{ flops}}{5n^2 \text{ flops} + 6t_s + 3nt_w}$$

Ejercicio 30

Dada la siguiente función

```
void func(double A[M][N], int sup[M]) {  
    int i, j;  
    double s = 0, m;  
    for (i=0; i<M; i++)  
        for (j=0; j<N; j++)  
            s += A[i][j];  
    m = s/(M*N);  
    for (i=0; i<M; i++) {  
        sup [i] = 0;  
        for (j=0; j<N; j++)  
            if (A[i][j]>m) sup[i]++;  
    }  
}
```

- a) ¿Qué hace la función?
- b) Escribe un programa MPI suponiendo que **A** se encuentra inicialmente en el proceso 0, y que al finalizar la función el vector **sup** debe estar también en el proceso 0. Se puede suponer que el número de filas de la matriz es divisible entre el número de procesos.
- c) Calcula el tiempo paralelo

Solución:

- a) Dada la matriz, obtiene un vector **sup** de manera que su componente i -ésima contiene el nº de elementos de la fila i -ésima de la matriz **A** que son mayores que la media de los elementos de **A**

```
void func(double A[M][N], int sup[M]) {  
    int i, j;  
    double s = 0, m;  
    for (i=0; i<M; i++)  
        for (j=0; j<N; j++)  
            s += A[i][j];  
    m = s/(M*N);  
    for (i=0; i<M; i++) {  
        sup[i] = 0;  
        for (j=0; j<N; j++)  
            if (A[i][j]>m) sup[i]++;  
    }  
}
```

} N

$$A(P_0) = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \end{bmatrix} k \xrightarrow[\text{MPI_Scatter}]{\text{Reparto de A}} \begin{bmatrix} Al(P_0) = A_0 \\ Al(P_1) = A_1 \\ Al(P_2) = A_2 \end{bmatrix} k \xrightarrow[\text{Cálculo de la suma local}]{\quad} \begin{bmatrix} s(P_0) \\ s(P_1) \\ s(P_2) \end{bmatrix} 1 \xrightarrow[\text{MPI_Allreduce}]{\text{Suma global}} m(P_0, P_1, P_2) = s(P_0) + s(P_1) + s(P_2)$$

$$\longrightarrow m(P_0, P_1, P_2) = \frac{m(P_0, P_1, P_2)}{MN} \xrightarrow[\text{de suppl}]{\text{Cálculo local}} \begin{bmatrix} suppl(P_0) \\ suppl(P_1) \\ suppl(P_2) \end{bmatrix} k \xrightarrow[\text{MPI_Gatter}]{\text{Recogida vectores locales suppl}} suppl(P_0) = \begin{bmatrix} suppl(P_0) \\ suppl(P_1) \\ suppl(P_2) \end{bmatrix} k$$

$$k = M / p \text{ (} p = \text{número de procesos) }$$

```

void funcp(double A[M][N], int sup[M]) {
    double Al[M][N];
    int supl[M];
    double s=0;
    int i, j, p;
    MPI_Comm_size(MPI_COMM_WORLD, &p);
}

```

```

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

```

`MPI_Scatter(A, M*N/p, MPI_DOUBLE, Al, M*N/p, MPI_DOUBLE, 0, MPI_COMM_WORLD)`

```

for (i=0; i<M/p; i++)
    for (j=0; j<N; j++)
        s += Al [i][j];

```

Almacenar en s la suma
de los elementos de Al

$$\begin{bmatrix} \frac{s(P_0)}{1} \\ \frac{s(P_1)}{1} \\ \frac{s(P_2)}{1} \end{bmatrix}$$

`MPI_Allreduce(&s, &m, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);`

$$m = m/(M*N)$$

$$m(P_0, P_1, P_2) = \frac{m(P_0, P_1, P_2)}{MN}$$

```

for (i=0; i<M/p; i++) {
    supl[i] = 0;
    for (j=0; j<N; j++)
        if (Al[i][j]>m) supl [i]++;
}

```

Para cada componente,
 $supl$ contiene el nº de
elementos de la fila
mayores que la media

$$\begin{bmatrix} \frac{supl(P_0)}{k} \\ \frac{supl(P_1)}{k} \\ \frac{supl(P_2)}{k} \end{bmatrix}$$

`MPI_Gather(supl, M/p, MPI_INT, sup, M/p, MPI_INT, 0, MPI_COMM_WORLD);`

$$A(P_0) = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \end{bmatrix} M / p \xrightarrow[\text{Reparto de } A \text{ con } MPI_Scatter}]{} \begin{bmatrix} Al(P_0) = A_0 \\ Al(P_1) = A_1 \\ Al(P_2) = A_2 \end{bmatrix} M / p$$

$$\begin{bmatrix} \frac{s(P_0)}{1} \\ \frac{s(P_1)}{1} \\ \frac{s(P_2)}{1} \end{bmatrix}$$

```

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

```

↓ Reducción (suma) para todos
 $m(P_0, P_1, P_2) = s(P_0) + s(P_1) + s(P_2)$

```

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

```

$$\begin{bmatrix} \frac{supl(P_0)}{1} \\ \frac{supl(P_1)}{1} \\ \frac{supl(P_2)}{1} \end{bmatrix} \xrightarrow[\text{Recogida de } supl \text{ en } P_0]{} sup(P_0) = \begin{bmatrix} \frac{supl(P_0)}{k} \\ \frac{supl(P_1)}{k} \\ \frac{supl(P_2)}{k} \end{bmatrix} k$$

```

void funcp(double A[M][N], int sup[M]) {
    double Al[M][N];
    int supl[M];
    double s=0;
    int i, j, p;
    MPI_Comm_size(MPI_COMM_WORLD, &p);

```

```
MPI_Scatter(A, M*N/p, MPI_DOUBLE, Al, M*N/p, MPI_DOUBLE, 0, MPI_COMM_WORLD)
```

```

for (i=0; i<M/p; i++)
    for (j=0; j<N; j++)
        s += Al [i][j];

```

```
MPI_Allreduce(&s, &m, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

```
m = m/(M*N)
```

```

for (i=0; i<M/p; i++) {
    supl[i] = 0;
    for (j=0; j<N; j++)
        if (Al[i][j]>m) supl [i]++;
}

```

```
MPI_Gather(supl, M/p, MPI_INT, sup, M/p, MPI_INT, 0, MPI_COMM_WORLD);
```

$$t_a(M, N, p) = \sum_{i=0}^{M/p} \sum_{j=0}^N 1 + p - 1 + 2 \approx \frac{MN}{p} \text{ flops}$$

MPI_Scatter(matriz de dimensión MN): $t_{c_s} = (p-1) \left(t_s + \frac{MN}{p} t_w \right)$

MPI_Allreduce(dato simple): $t_{c_r} = 2(p-1)(t_s + t_w)$

MPI_Gather: $t_{c_g} = (p-1) \left(t_s + \frac{M}{p} t_w \right)$

$$t(M, N, p) = \frac{MN}{p} \text{ flops} + t_{c_s} + t_{c_r} + t_{c_g}$$

Ejercicio 31

- Se quiere optimizar el envío y recepción de matrices de dimensión $3n \times 3n$, donde n es un número natural mayor o igual a 1, cuya estructura se presenta a continuación:

$$A = \begin{pmatrix} A_0 & 0_{n \times n} & 0_{n \times n} \\ 0_{n \times n} & A_1 & 0_{n \times n} \\ 0_{n \times n} & 0_{n \times n} & A_2 \end{pmatrix} \in \mathbb{R}^{3n \times 3n}, \quad A_i \in \mathbb{R}^{n \times n}, \quad 0_{n \times n} = \text{matriz nula}$$

- Suponiendo que se tienen 3 procesos, implementa un programa paralelo MPI que cumpla las siguientes condiciones:
 - P₀ utiliza la función leeMat para obtener la matriz **A**: A=leeMat(N).
 - P₀ debe repartir dicha matriz en matrices locales **Al** entre tres procesos de manera que P₀ se quede con A₀, P₁ se quede con A₁ y P₂ se quede con A₂, usando para ello tipos derivados
- Calcula el tiempo de comunicaciones
- Ejemplo n=2:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 6 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 8 & 9 \\ 0 & 0 & 0 & 0 & 10 & 11 \end{pmatrix} \Rightarrow Al(P_0) = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \quad Al(P_1) = \begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix} \quad Al(P_2) = \begin{pmatrix} 8 & 9 \\ 10 & 11 \end{pmatrix}$$

Ejercicio 31

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

$$A = \begin{array}{|c|c|c|} \hline & A_0 & 0_{n \times n} & 0_{n \times n} \\ \hline A_0 & \begin{matrix} x & x & \cdots & x \\ x & x & \cdots & x \\ \vdots & \vdots & \ddots & \vdots \\ x & x & \cdots & x \end{matrix} & \begin{matrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{matrix} & \begin{matrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{matrix} \\ \hline & 0_{n \times n} & A_1 & 0_{n \times n} \\ \hline 0_{n \times n} & \begin{matrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{matrix} & \begin{matrix} x & x & \cdots & x \\ x & x & \cdots & x \\ \vdots & \vdots & \ddots & \vdots \\ x & x & \cdots & x \end{matrix} & \begin{matrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{matrix} \\ \hline & 0_{n \times n} & 0_{n \times n} & A_2 \\ \hline 0_{n \times n} & \begin{matrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{matrix} & \begin{matrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{matrix} & \begin{matrix} x & x & \cdots & x \\ x & x & \cdots & x \\ \vdots & \vdots & \ddots & \vdots \\ x & x & \cdots & x \end{matrix} \\ \hline \end{array}$$

Definir un nuevo tipo para enviar eficientemente A_i



count=n
blocklength=n
stride=3n



```
MPI_Datatype newtype;
MPI_Type_vector(n, n, 3*n, MPI_DOUBLE, &newtype);
```

Ejercicio 31

P_0 reparte la matriz \mathbf{A} en matrices locales \mathbf{Al} entre los tres procesos, de manera que P_0 se queda con A_0 , P_1 se queda con A_1 y P_2 se queda con A_2 , usando el tipo derivado anterior:

```
double max_mat(double A[3*n][3*n], Al[n][n], int p, int ip){  
    MPI_Status stat;  
    MPI_Datatype ntype;  
    MPI_Type_vector(n, n, 3*n, MPI_DOUBLE, &ntype);  
    MPI_Type_commit(&ntype);  
    if ( ip == 0 ){  
        MPI_Send(&A[n][0],1, ntype, 1, 0, MPI_COMM_WORLD);  
        MPI_Send(&A[2*n][0],1, ntype, 2, 0, MPI_COMM_WORLD);  
        MPI_Sendrecv(A, 1, ntype, 0, 0, Al, n*n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &stat);  
    }  
    else  
        MPI_Recv(Al, n*n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &stat);  
    MPI_Type_free (&ntype);  
}
```

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 6 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 8 & 9 \\ 0 & 0 & 0 & 0 & 10 & 11 \end{pmatrix} \Rightarrow \begin{aligned} Al(P_0) &= \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \\ Al(P_1) &= \begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix} \\ Al(P_2) &= \begin{pmatrix} 8 & 9 \\ 10 & 11 \end{pmatrix} \end{aligned}$$

$$t_c = 2(t_s + n^2 t_w)$$

Ejercicio 32 (Parcial 2020)

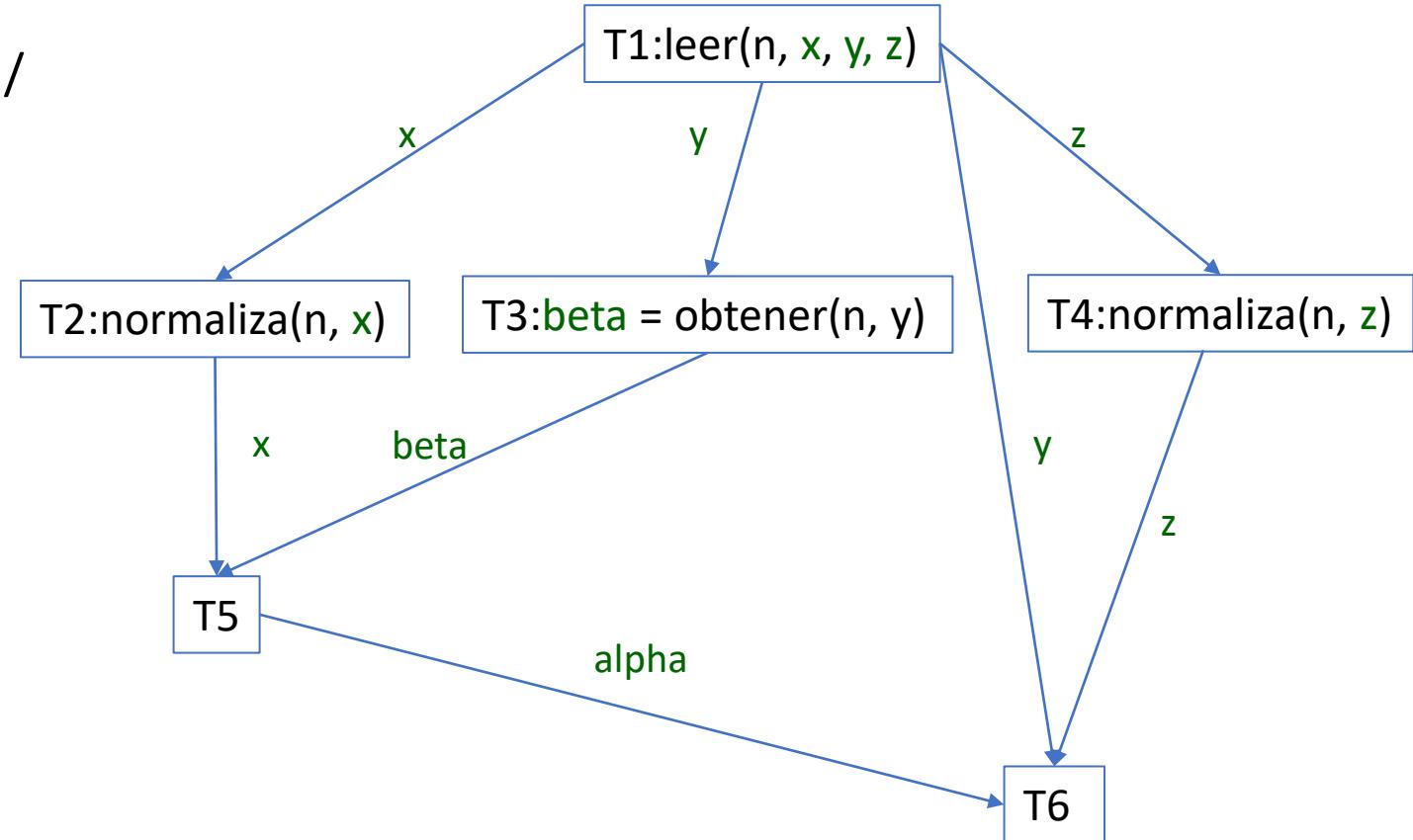
Se quiere parallelizar el siguiente código mediante MPI:

```
int calcular(int n, double x[], double y[], double z[]) {  
    int i;  
    double alpha, beta;  
    /* Leer los vectores x, y, z, de dimensión n */  
    leer(n, x, y, z); /* tarea 1 */  
    normaliza(n, x); /* tarea 2 */  
    beta = obtener(n, y); /* tarea 3 */  
    normaliza(n, z); /* tarea 4 */  
    /* tarea 5 */  
    alpha = 0.0;  
    for (i=0; i<n; i++)  
        if (x[i] > 0.0) alpha = alpha + beta*x[i];  
        else alpha = alpha + x[i]*x[i];  
    /* tarea 6 */  
    for (i=0; i<n; i++) z[i] = z[i] + alpha*y[i];  
}
```

- Suponemos que disponemos de **3 procesos**, de los cuales solo uno ha de llamar a la función **leer**. Se puede asumir que el valor de **n** es conocido por todos los procesos.
 - El resultado final (**z**) puede quedar almacenado en cualquiera de los 3 procesos.
 - La función **leer** modifica los tres vectores, la función **normaliza** modifica su segundo argumento y la función **obtener** no modifica ninguno de sus argumentos de entrada.
- a) Dibuja el grafo de dependencias de las diferentes tareas.

Ejercicio 32 (Parcial 2020)

```
int calcular(int n, double x[], double y[], double z[]) {  
    int i;  
    double alpha, beta;  
    /* Leer los vectores x, y, z, de dimensión n */  
    leer(n, x, y, z); /* tarea 1 */  
    normaliza(n, x); /* tarea 2 */  
    beta = obtener(n, y); /* tarea 3 */  
    normaliza(n, z); /* tarea 4 */  
    /* tarea 5 */  
    alpha = 0.0;  
    for (i=0; i<n; i++)  
        if (x[i] > 0.0) alpha = alpha + beta*x[i];  
        else alpha = alpha + x[i]*x[i];  
    /* tarea 6 */  
    for (i=0; i<n; i++) z[i] = z[i] + alpha*y[i];  
}
```



Ejercicio 32 (Parcial 2020)

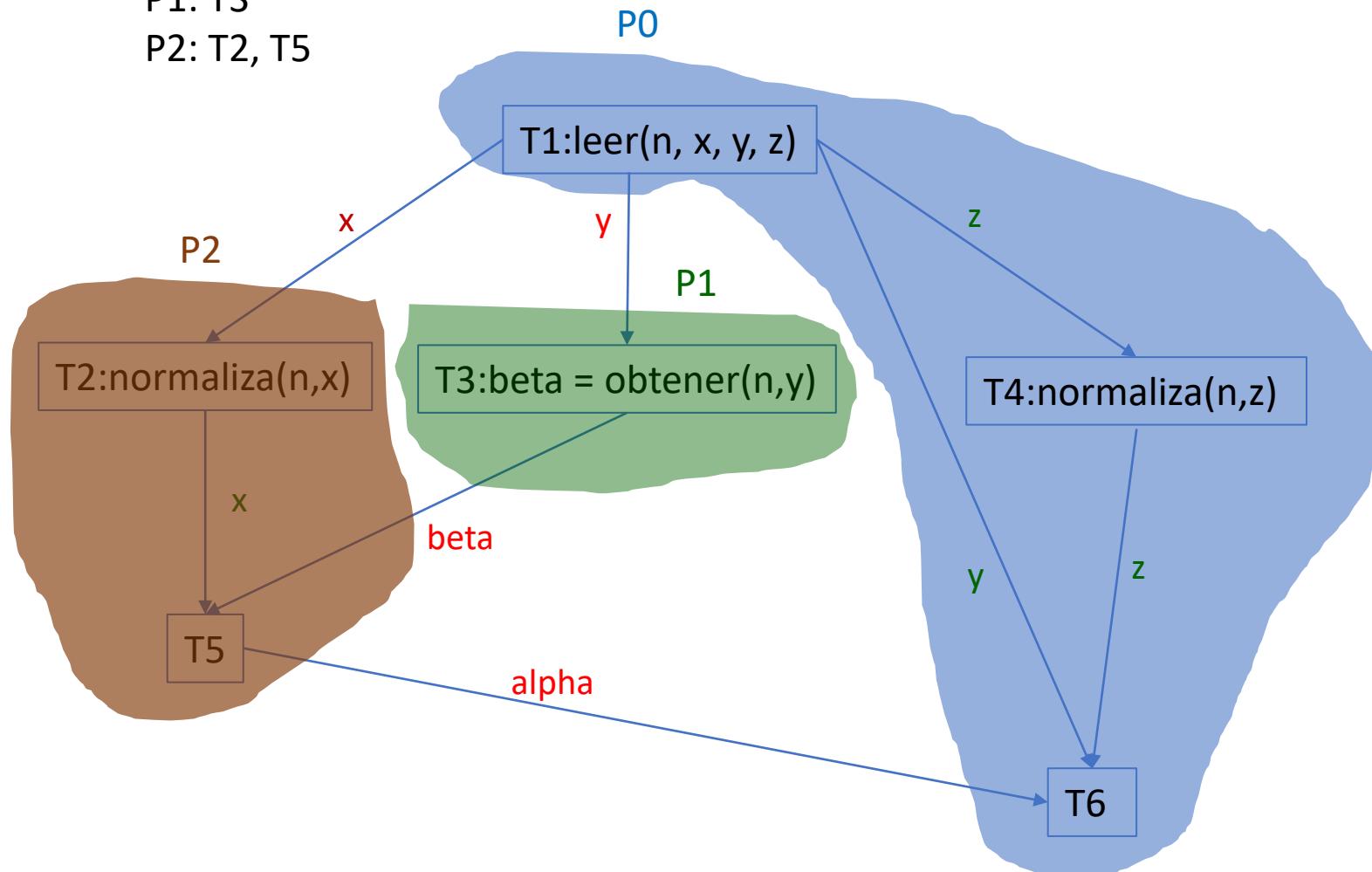
b) Escribe el código MPI que resuelve el problema utilizando una asignación que maximice el paralelismo y minimice el coste de comunicaciones.

Reparto:

P0: T1, T4, T6

P1: T3

P2: T2, T5



P0:
leer(n, x, y, z)
Send(y, P1)
Send(x, P2)
normaliza(n, z)
Recv(alpha, P2)
T6

P1:
Recv(y, P0)
beta=obtener(n, y)
Send(beta, P2)

P2:
Recv(x, P0)
normaliza(n, x)
Recv(beta, P1)
T5
Send(alpha, P0)

P0:

leer(n, x, y, z)
Send(x, P2)
Send(y, P1)
normaliza(n, z)
Recv(alpha, P2)
T6

P1:

Recv(y,P0)
beta=obtener(n, y)
Send(beta, P2)

P2:

Recv(x,P0)
normaliza(n, x)
Recv(beta, P1)
T5
Send(alpha, P0)

```
int calcular_mpi(int n, double x[], double y[], double z[]) {  
    int i, rank;  
    double alpha, beta;  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    if (rank == 0) { /* Leer los vectores x, y, z, de dimension n */  
        leer( n, x, y, z ); /* tarea 1 */  
        MPI_Send(x, n, MPI_DOUBLE, 2, 123, MPI_COMM_WORLD);  
        MPI_Send(y, n, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD);  
        normaliza(n,z);  
        MPI_Recv(&alpha, 1, MPI_DOUBLE, 2, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
        for (i=0; i<n; i++) z[i] = z[i] + alpha*y[i]; /* tarea 6 */  
    }  
    else if (rank == 1) {  
        MPI_Recv(y, n, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
        beta = obtener(n,y); /* tarea 3 */  
        MPI_Send(&beta, 1, MPI_DOUBLE, 2, 123, MPI_COMM_WORLD);  
    }  
    else if (rank == 2) {  
        MPI_Recv(x, n, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
        normaliza(n,x); /* tarea 2 */  
        MPI_Recv(&beta, 1, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
        /* tarea 5 */  
        alpha = 0.0;  
        for (i=0; i<n; i++)  
            if (x[i] > 0.0) alpha = alpha + beta*x[i];  
            else { alpha = alpha + x[i]*x[i]; }  
        MPI_Send(&alpha, 1, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD );  
    }  
}
```

Ejercicio 33 (Parcial 2020)

Se quiere implementar en MPI el envío desde el proceso 0 al resto de procesos de la diagonal principal y antidiagonal de una matriz cuadrada A de dimensión N , empleando para ello tipos de datos derivados (uno para cada tipo de diagonal), con la menor cantidad posible de mensajes. Supondremos que:

- N es una constante conocida.
- Los elementos de la diagonal principal son: $A_{00}, A_{11}, A_{22}, \dots, A_{N-1,N-1}$.
- Los elementos de la antidiagonal son: $A_{0,N-1}, A_{1,N-2}, A_{2,N-3}, \dots, A_{N-1,0}$.
- Solo el proceso 0 posee la matriz A y enviará la totalidad de dichas diagonales al resto de procesos.

Un ejemplo para una matriz de tamaño $N = 5$ sería:

$$A = \begin{pmatrix} * & & & & * \\ & * & & * & \\ & & * & & \\ & * & & * & \\ * & & & & * \end{pmatrix}$$

- (a) Completa la siguiente función, donde los procesos del 1 en adelante almacenarán sobre la matriz A las diagonales recibidas:

```
void sendrecv_diagonals(double A[N][N])
```

Trasparencias adicionales T3-S3

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

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

(a) **Solución:** Definiremos un tipo de datos para envíos de la diagonal principal y otro para envíos de la antidiagonal. A continuación realizaríamos difusiones con ambos tipos de datos:

```
void sendrecv_diagonals(double A[N][N]) {
    MPI_Datatype principal,antidiag;
    MPI_Type_vector(N, 1, N+1, MPI_DOUBLE, &principal);
    MPI_Type_commit(& principal);
    MPI_Type_vector(N, 1, N-1, MPI_DOUBLE, &antidiag);
    MPI_Type_commit(&antidiag);
    MPI_Bcast(A, 1, principal, 0, MPI_COMM_WORLD);
    MPI_Bcast(&A[0][N-1], 1, antidiag, 0, MPI_COMM_WORLD);
    MPI_Type_free(&principal);
    MPI_Type_free(&antidiag);
}
```

$$A = \begin{pmatrix} * & & & * \\ & * & * & * \\ & & * & * \\ * & & * & * \end{pmatrix}$$

Diagonal principal:
 count=N
 blocklength=1
 stride=N+1

Antidiagonal:
 count=N
 blocklength=1
 stride=N-1

Ejercicio 33 (Parcial 2020)

(b) Completa esta otra función, variante de la anterior, donde todos los procesos (incluido el proceso 0) almacenarán sobre los vectores **prin** y **anti** las correspondientes diagonales:

```
void sendrecv_diagonals(double A[N][N], double prin[N], double anti[N]) {  
    int id;  
    MPI_Datatype principal, antidiag;  
    MPI_Comm_rank(MPI_COMM_WORLD, &id);  
    if (id==0) {  
        MPI_Type_vector(N, 1, N+1, MPI_DOUBLE, &principal);  
        MPI_Type_commit(& principal);  
        MPI_Type_vector(N, 1, N-1, MPI_DOUBLE, &antidiag);  
        MPI_Type_commit(&antidiag);  
        MPI_Sendrecv(A, 1, principal, 0, 10, prin, N , MPI_DOUBLE, 0, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
        MPI_Sendrecv(&A[0][N-1], 1, antidiag, 0, 20, anti, N, MPI_DOUBLE, 0, 20, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
        MPI_Type_free(&principal);  
        MPI_Type_free(&antidiag);  
    }  
    MPI_Bcast(prin, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
    MPI_Bcast(anti, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
}
```

```
int MPI_Sendrecv(void *sendbuf, int  
                 sendcount, MPI_Datatype sendtype, int  
                 dest, int sendtag, void *recvbuf, int  
                 recvcount, MPI_Datatype recvtype, int  
                 source, int recvtag, MPI_Comm comm,  
                 MPI_Status *status)  
  
int MPI_Bcast(void *buf, int count,  
              MPI_Datatype datatype, int root,  
              MPI_Comm comm)
```

Ejercicio 34 (Parcial 2020)

Observa la siguiente función, la cual cuenta el número de apariciones de un número en una matriz e indica también la primera fila en la que aparece:

```
void search(double A[M][N], double x) {  
    int i, j, first, count;  
    first = M ; count = 0;  
    for (i=0; i<M; i++)  
        for (j=0; j<N; j++)  
            if (A[i][j] == x) {  
                count++;  
                if (i < first) first = i;  
            }  
    printf("%g está %d veces, la primera vez en la fila %d.\n", x, count, first);  
}
```

Ejercicio 34 (Parcial 2020)

Parallelízala mediante MPI repartiendo la matriz **A** entre todos los procesos disponibles. Tanto la matriz como el valor a buscar están inicialmente disponibles únicamente en el proceso **owner**. Asumimos que el número de filas y columnas de la matriz es un múltiplo exacto del número de procesos. El **printf** que muestra el resultado por pantalla debe hacerlo únicamente un proceso. Utiliza operaciones de comunicación colectiva allí donde sea posible. Para ello, completa esta función:

```
void par_search(double A[M][N], double x, int owner) {  
    double Al[M][N];
```

Nota: por simplicidad, en el siguiente esquema supondremos que el propietario (**owner**) es el proceso 0.

Reparto y Difusión inicial

$$A(P_0) = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \end{bmatrix} k \xrightarrow[\text{MPI_Scatter}]{\text{Reparto}} \begin{bmatrix} Al(P_0) = A_0 \\ Al(P_1) = A_1 \\ Al(P_2) = A_2 \end{bmatrix} k$$
$$x(P_0) \xrightarrow[\text{MPI_Bcast}]{\text{Difusión}} x(P_1), x(P_2)$$
$$k = M / p \quad (p = \text{número de procesos})$$

Cálculo local

```
firstl = M ; count = 0;  
for (i=0; i<k; i++)  
    for (j=0; j<N; j++)  
        if (Al[i][j] == x) {  
            count++;  
            if (i < firstl) firstl = i;  
        }  
firstl=k*id+firstl /*índice global*/
```

$$\begin{matrix} i_g & | & k \\ i_l & & i_p \end{matrix} \xrightarrow{} i_g = k i_p + i_l$$

Obtención de resultados

```
first = min {firstl( $P_0$ ), firstl( $P_1$ ), firstl( $P_2$ )}  
cont = contl( $P_0$ ) + contl( $P_1$ ) + contl( $P_2$ )
```

$$A(P_0) = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \end{bmatrix} k \xrightarrow[\text{MPI_Scatter}]{\text{Reparto}} \begin{bmatrix} Al(P_0) = A_0 \\ Al(P_1) = A_1 \\ Al(P_2) = A_2 \end{bmatrix} k \xrightarrow{\text{Cálculo local}} \begin{cases} \text{firstl} \\ \text{contl} \end{cases} \rightarrow \begin{array}{l} \text{firstl} = k * \text{id} + \text{firstl} \\ (\text{índice global}) \end{array} \rightarrow \begin{array}{l} \text{first}(P_0) = \min \{ \text{firstl}(P_0), \text{firstl}(P_1), \text{firstl}(P_2) \} \\ \text{cont}(P_0) = \text{contl}(P_0) + \text{contl}(P_1) + \text{contl}(P_2) \end{array}$$

$x(P_0) \xrightarrow[\text{MPI_Bcast}]{\text{Difusión}} x(P_1), x(P_2) \quad (k = M / p, p = \text{número de procesos})$

```
void par_search(double A[M][N], double x, int owner) {
    double Al[M][N];
    int i, j, first, count, firstl, contl, id, p;
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    k=M/p;
    MPI_Scatter(A, k*N, MPI_DOUBLE, Al, k*N, MPI_DOUBLE, owner, MPI_COMM_WORLD);
    MPI_Bcast(&x, 1 ,MPI_DOUBLE, owner, MPI_COMM_WORLD);
    firstl = M ; contl = 0;
    for (i=0; i<k; i++)
        for (j=0; j<N; j++)
            if (Al[i][j] == x) {
                contl++;
                if (i < firstl) firstl = i;
            }
    firstl = k * id + firstl;
    MPI_Reduce(&firstl, &first, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);
    MPI_Reduce(&contl, &count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (id == 0) printf("%g está %d veces, la primera vez en la fila %d.\n", x, count, first);
}
```

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

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

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

Ejercicio 35 (Final 2020)

(a) Escribe la llamada a la primitiva MPI de comunicación colectiva equivalente al siguiente código:

```
int sz, rank;
double val, res, aux;
MPI_Comm comm=MPI_COMM_WORLD;
MPI_Status stat;
val = ...
MPI_Comm_size(comm, &sz);
if (sz==1) res = val;
else {
    MPI_Comm_rank(comm, &rank);
    if (rank==0) {
        MPI_Recv(&aux, 1, MPI_DOUBLE, rank+1, 999, comm, &stat);
        res = aux + val;
    }
    else if (rank==sz-1)
        MPI_Send(&val, 1, MPI_DOUBLE, rank-1, 999, comm);
    else {
        MPI_Recv(&aux, 1, MPI_DOUBLE, rank+1, 999, comm, &stat);
        aux = aux + val;
        MPI_Send(&aux, 1, MPI_DOUBLE, rank-1, 999, comm);
    }
}
```

```

double val, res, aux;
MPI_Comm comm=MPI_COMM_WORLD;
MPI_Status stat;
val = ...
MPI_Comm_size(comm, &sz);
if (sz==1) res = val;
else {
    MPI_Comm_rank(comm, &rank);
    if (rank==0) {
        MPI_Recv(&aux, 1, MPI_DOUBLE, rank+1, 999, comm, &stat);
        res = aux + val;
    }
    else if (rank==sz-1)
        MPI_Send(&val, 1, MPI_DOUBLE, rank-1, 999, comm);
    else {
        MPI_Recv(&aux, 1, MPI_DOUBLE, rank+1, 999, comm, &stat);
        aux = aux + val;
        MPI_Send(&aux, 1, MPI_DOUBLE, rank-1, 999, comm);
    }
}

```

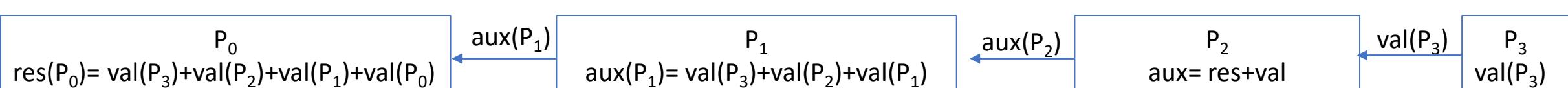
Ejercicio 35 (Final 2020)

```

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

```

Solución: `MPI_Reduce(&val, &res, 1, MPI_DOUBLE, MPI_SUM, 0, comm);`



Ejercicio 35 (Final 2020)

(b) Dada la siguiente llamada a una primitiva de comunicación colectiva:

```
double val=...;  
MPI_Bcast(&val, 1, MPI_DOUBLE, 0, comm);
```

Escribe un fragmento de código equivalente (debe realizar la misma comunicación) pero utilizando únicamente primitivas de comunicación punto a punto.

Solución:

```
double val=...;  
int i, sz, rank;  
MPI_Status stat;  
...  
MPI_Comm_size(comm, &sz);  
MPI_Comm_rank(comm, &rank);  
if (rank==0) {  
    for (i=1; i<sz; i++)  
        MPI_Send(&val, 1, MPI_DOUBLE, i, 100, comm);  
}  
else  
    MPI_Recv(&val, 1, MPI_DOUBLE, 0, 100, comm, &stat);
```

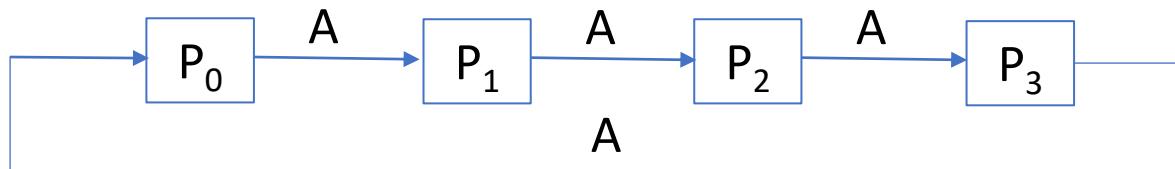
Ejercicio 36 (Final 2020)

Desarrolla un programa paralelo con MPI en el que el proceso 0 lea una matriz de $M \times N$ números reales de disco (con la función `read_mat`) y esta matriz se vaya pasando de un proceso a otro hasta llegar al último, el cual se la devolverá al proceso 0. El programa deberá medir el tiempo total de ejecución, sin contar la lectura de disco, y mostrarlo por pantalla. Utiliza esta cabecera para la función principal:

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

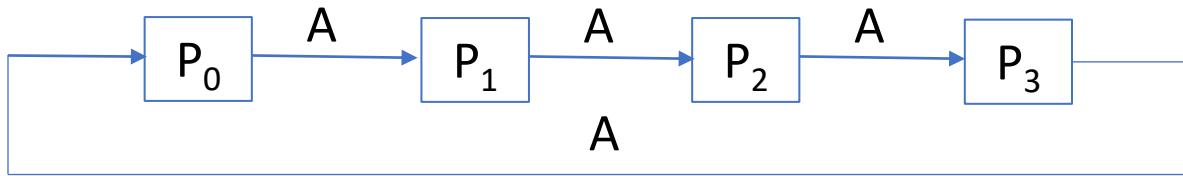
teniendo en cuenta que la función de lectura de la matriz tiene esta cabecera:

```
void read_mat(double A[M][N]);
```



Ejercicio 36 (Final 2020)

```
#define M ...
#define N ...
int main(int argc, char *argv[]) {
    int id, np;
    double A[M][N], t1, t2;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    if (id==0) read_mat(A);
    t1=MPI_Wtime();
    if (id==0) {
        MPI_Send(A, M*N, MPI_DOUBLE, 1, 1234, MPI_COMM_WORLD);
        MPI_Recv(A, M*N, MPI_DOUBLE, p-1, 1234, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else {
        MPI_Recv(A, M*N, MPI_DOUBLE, id-1, 1234, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(A, M*N, MPI_DOUBLE, (id+1)%p, 1234, MPI_COMM_WORLD);
    }
    t2=MPI_Wtime();
    if (id==0) printf("Tiempo: %.2f segundos.\n",t2-t1);
    MPI_Finalize();
    return 0;
}
```



(b) Indica el coste teórico total de las comunicaciones:

$$t_c = p(t_s + MNt_w)$$

Ejercicio 37 (Final 2020)

Queremos repartir una matriz de M filas y N columnas que se encuentra en el proceso 0 entre 4 procesos mediante un reparto por columnas cíclico. Como ejemplo, se muestra el caso de la siguiente matriz de 6 filas y 8 columnas:

$$A(P_0) = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \end{bmatrix}$$

$$Al(P_0) = \begin{bmatrix} 1 & 5 \\ 9 & 13 \\ 17 & 21 \\ 25 & 29 \\ 33 & 37 \\ 41 & 45 \end{bmatrix} \quad Al(P_1) = \begin{bmatrix} 2 & 6 \\ 10 & 16 \\ 18 & 26 \\ 26 & 36 \\ 34 & 46 \\ 42 & 56 \end{bmatrix} \quad Al(P_2) = \begin{bmatrix} 3 & 7 \\ 11 & 15 \\ 19 & 23 \\ 27 & 31 \\ 35 & 39 \\ 43 & 47 \end{bmatrix} \quad Al(P_3) = \begin{bmatrix} 4 & 8 \\ 12 & 16 \\ 20 & 24 \\ 28 & 32 \\ 36 & 40 \\ 44 & 48 \end{bmatrix}$$

Ejercicio 37 (Final 2020)

Implementa una función en MPI que realice, mediante primitivas punto a punto y de la forma más eficiente posible, el envío y recepción de dicha **matriz**. Nota: La recepción de la matriz deberá hacerse en una matriz compacta (en **Imat**), como muestra el ejemplo anterior. **Nota:** El número de columnas se asume que es un múltiplo de 4 y se reparte siempre entre 4 procesos. Para la implementación se recomienda utilizar la siguiente cabecera:

```
int MPI_Reparte_col_cic(float mat[M][N], float Imat[M][N/4])
```

Solución:

Se trataría de definir un nuevo tipo de datos derivado, de manera que se envíen en un solo mensaje los elementos que le corresponden a cada proceso.

Ejercicio 37 (Final 2020)

Por ejemplo, a P1 se le enviarían los elementos coloreados en rojo:

$$A(P_0) = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \end{bmatrix}$$

Como las filas de la matriz se almacenan en posiciones consecutivas de memoria, se trata de un reparto cíclico de elementos almacenados en posiciones consecutivas.



Se trataría de enviar en un solo mensaje $M*N/4$ bloques de 1 elemento, con separación entre los primeros elementos de bloques consecutivos igual a 4.

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

count= $M*N/4$
blocklength=1
Stride=4

Una vez se define este nuevo tipo de dato, se realizarán envíos desde las posiciones 0, 1, 2, o 3 de ese nuevo tipo de dato. En recepción, los datos se almacenarán en matrices de tamaño M filas y N/4 columnas; es decir, se recibirían como datos almacenados en posiciones consecutivas.

Ejercicio 37 (Final 2020)

```
int MPI_Reparte_col_cic(float mat[M][N], float lmat[M][N/4]){
    int id, i;
    MPI_Datatype col;
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Type_vector(M*N/4, 1, 4, MPI_FLOAT ,&col);
    MPI_Type_commit(&col);
    if (id==0) {
        for (i=1;i<4;i++)
            MPI_Send(&mat[0][i], 1, col, i, 10 ,MPI_COMM_WORLD);
        MPI_Sendrecv(mat, 1, col, 0, 10, lmat, M*N/4, MPI_FLOAT, 0, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else
        MPI_Recv(lmat, M*N/4, MPI_FLOAT, 0, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Type_free(&col);
    return 0;
}
```

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

int MPI_Sendrecv(void *sendbuf, int
                 sendcount, MPI_Datatype sendtype, int
                 dest, int sendtag, void *recvbuf, int
                 recvcount, MPI_Datatype recvtype, int
                 source, int recvtag, MPI_Comm comm,
                 MPI_Status *status)
```