Transparencias MPI

Tipos Derivados en MPI

MPI_Type_vector

- A partir de un tipo de datos homogéneo (array), se puede crear un nuevo tipo de datos, el cual permite enviar en un solo mensaje count=n bloques de datos no contiguos que tienen la misma longitud (blocklength) y se encuentran separados a la misma distancia medida desde los comienzos de los bloques (stride).

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

count=número de bloques=n
blocklength=longitud de los bloques
stride=Número de elementos entre el inicio de un bloque y el inicio del bloque siguiente
oldtype=tipo de dato original
newtype=nuevo tipo de dato

- El nuevo tipo es ingresado en el sistema para poder ser utilizado mediante MPI_Type_commit:

MPI_Type_commit(MPI_Datatype * newtype)

- Cuando no se va a utilizar o antes de finalizar el programa, se utiliza MPI_Type_free para liberar el tipo de datos creado del sistema:

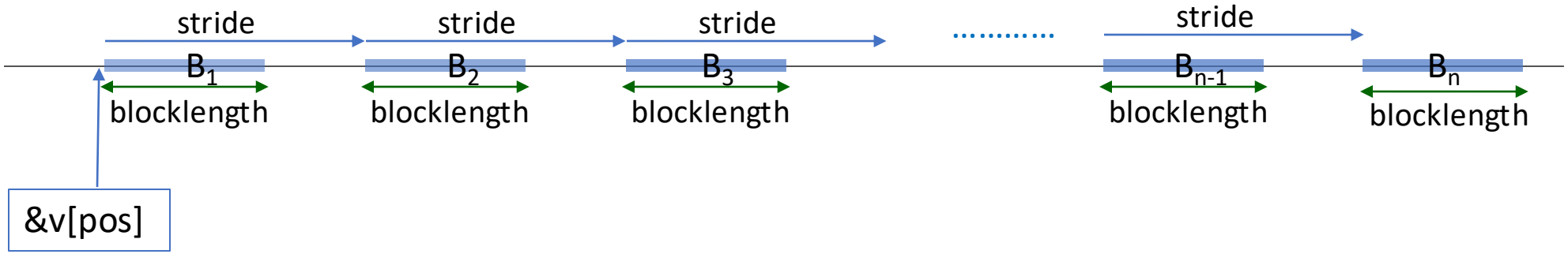
MPI_Type_free MPI Datatype * newtype)

MPI_Type_vector

- Los bloques son enviados mediante:

MPI_Send(&v[pos], 1, newtype,…….)

donde &v[pos] es la dirección del primer dato del primer bloque (B_1) a enviar:

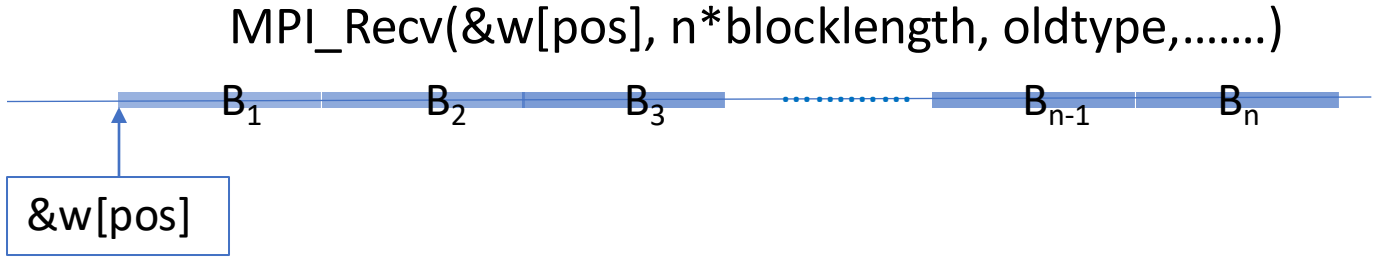


Los datos son “empaquetados” y después enviados mediante MPI_Send

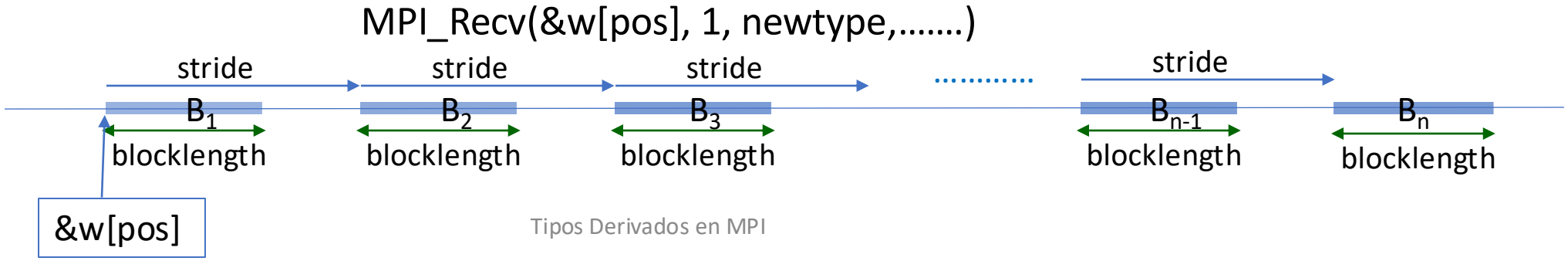
$$t_c = t_s + \mathrm{n}\cdot\mathrm{blocklength}\cdot t_w$$

MPI_Type_vector

- En el proceso receptor del mensaje, los datos pueden ser almacenados como:
  - Bloques consecutivos. En este caso, se recibe como un vector de dimensión n*blocklength, debiendo indicar la posición desde la cual se van a almacenar los datos, el nº total de datos y el tipo básico de partida.



  - Bloques separados con la misma estructura que tenían los datos en el envío (desempaquetado). Al igual que en el caso anterior, hay que indicar la posición desde la cual se van a almacenar los datos, un número datos igual a 1 y el tipo derivado usado.



MPI_Type_vector

n
=
MPI_Type_vector(int count,

int blocklength,

int stride,

MPI_Datatype oldtype,

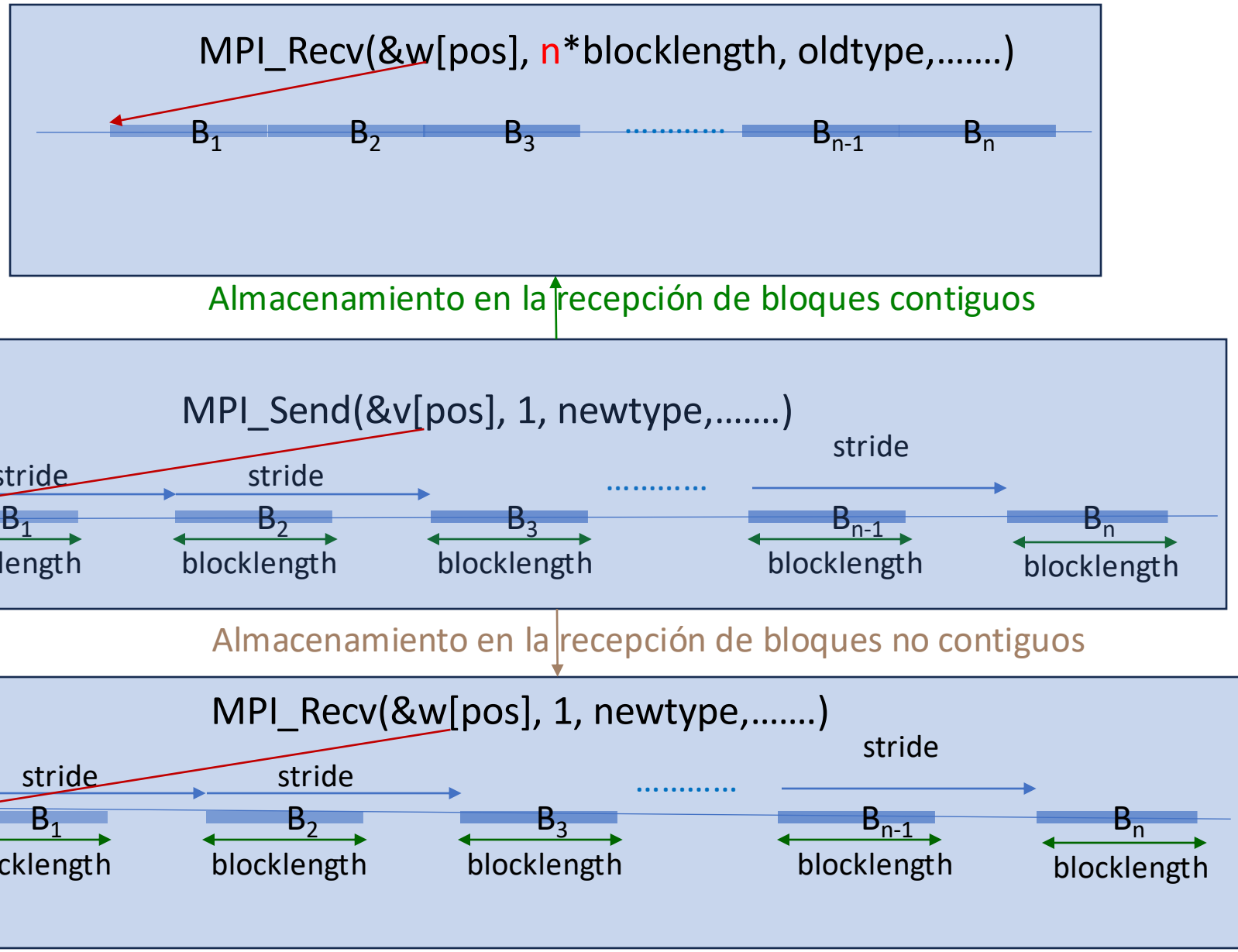
MPI_Datatype *newtype)

MPI Type_commit(

MPI_Datatype * newtype)

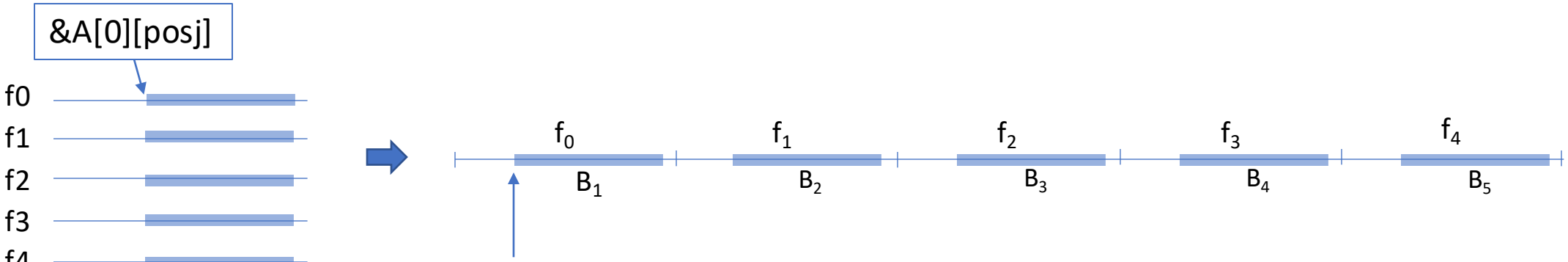
MPI Type free

MPI Datatype * newtype)



MPI_Type_vector

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

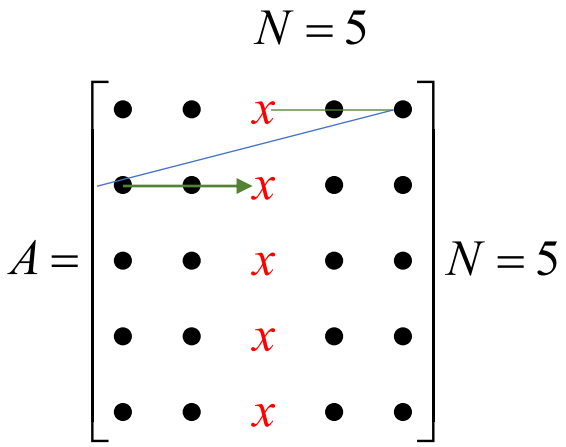


&A[0][posj] (en este caso suponemos que el primer bloque se encuentra en la fila 0, aunque podría comenzar en otra fila)

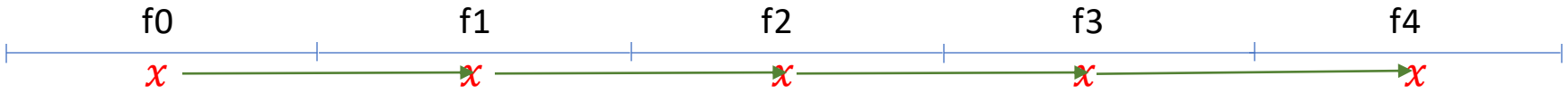
MPI_Send(&A[0][posj], newtype, 1,…….)

Ejercicio 1

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




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



count=N
blocklength=1
stride=N

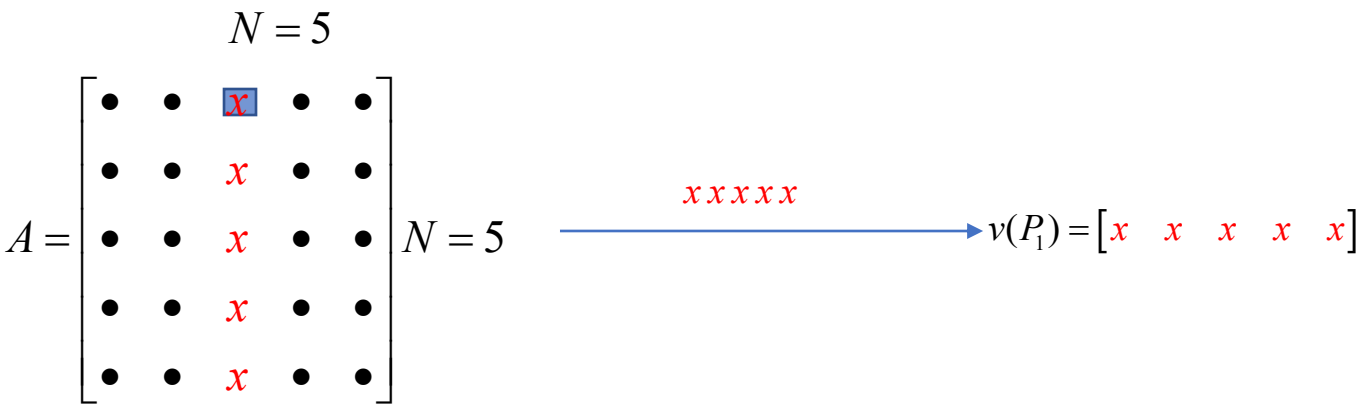
```
MPI_Datatype col;
MPI_Type_vector(N, 1, N, MPI_DOUBLE, &col);
MPI_Type_commit(&col);
……..
MPI_Type_free (&col);
```

Ejercicio 2

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

a) Usando el tipo de datos del ejercicio anterior ¿cómo se podría realizar un envío eficiente de la tercera 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][2], 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);
```




```
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**:

$$t_c = t_s + Nt_w$$

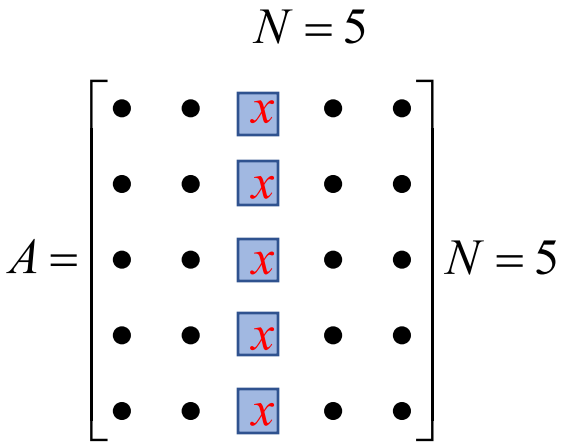
Ejercicio 2

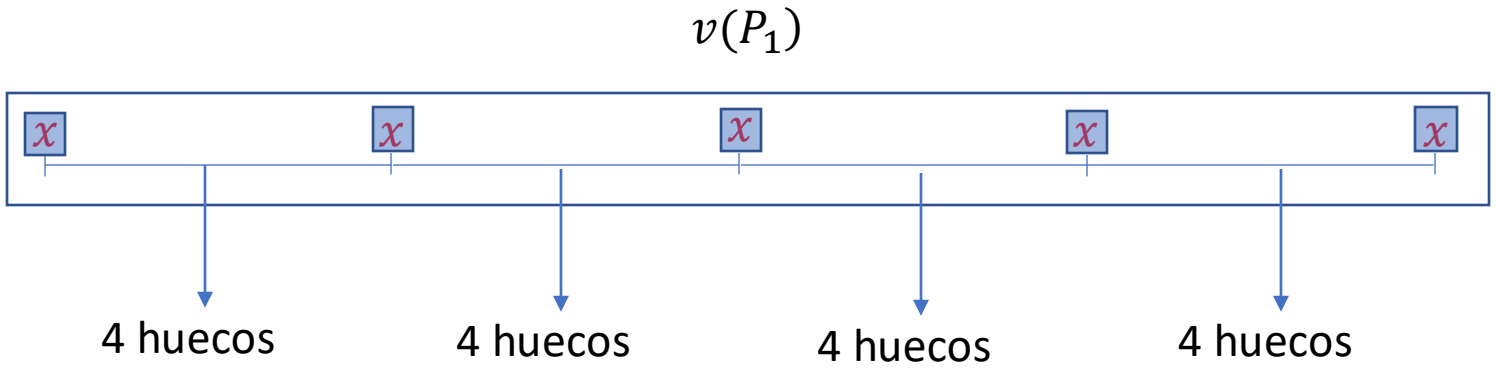
Nota: Si en la recepción hubiésemos usado:

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 podría quedar sobrepasada





Ejercicio 2

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);
for(i=0; i<N; i++){
  if (id==0)
    MPI_Send(&A[i][2], 1,MPI_DOUBLE, 1, 100, MPI_COMM_WORLD);
  else if (id==1)
    MPI_Recv(&v[i], 1, 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)
```

El tiempo de comunicaciones sin usar el tipo **col**:

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

El tiempo de comunicación usando el tipo **col** es:

$$t_c = t_s + Nt_w$$

¡El tiempo de comunicaciones ha resultado ser mayor!

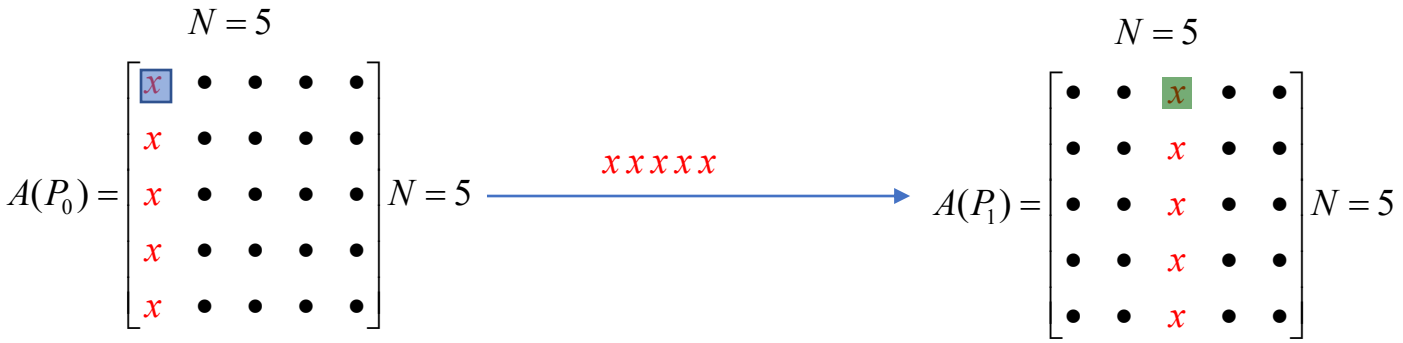
$$N = 5$$

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

Ejercicio 2

c) Usa el tipo de datos **col** para que la primera columna de la matriz A de P_0 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);
```




```
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 3

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.

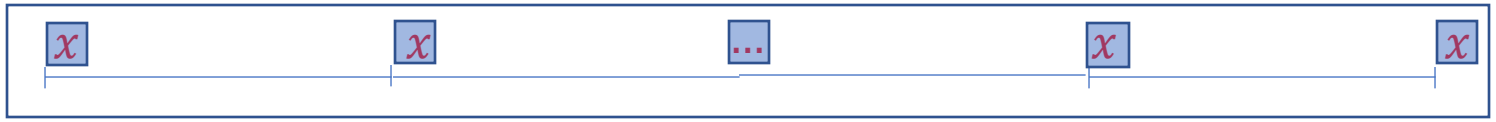
$$A = \begin{bmatrix} x & x & \text{L} & x & x \\ x & x & \text{L} & x & x \\ \text{M} & \text{M} & \text{O} & x & x \\ x & x & \text{L} & x & x \\ x & x & \text{L} & x & x \end{bmatrix} \begin{matrix} \\ \\ \\ \\ N \end{matrix} \qquad N$$

$$N = 5$$

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

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

$v(P_1)$



count=N
blocklength=1
stride=N+1

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

Ejercicio 4

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.

$$A = N \begin{bmatrix} x & x & \text{L} & x & x \\ x & x & \text{L} & x & x \\ \text{M} & \text{M} & \text{O} & x & x \\ x & x & \text{L} & x & x \\ x & x & \text{L} & x & x \end{bmatrix}$$

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

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

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 5 (Parcial 2021)

Se tiene un programa MPI en el que dos procesos, $\boldsymbol{P_0}$ y $\boldsymbol{P_1}$, deben comunicarse determinados elementos de una matriz de números reales de doble precisión, representada por una matriz cuadrada **A** en el proceso emisor y **B** en el receptor. El algoritmo matricial requiere que se envíen los elementos de la diagonal principal (excepto el primero) junto con los de la primera subdiagonal, marcados como *d* y *s*, respectivamente, en la matriz **A** de la figura, de forma que el proceso receptor debe almacenar estos elementos desplazados una fila hacia arriba, es decir, con los valores *s* ocupando la diagonal principal y los valores *d* ocupando la primera superdiagonal, como se indica en la matriz **B** de la figura. Supondremos que **A** es solo conocido por P_0.

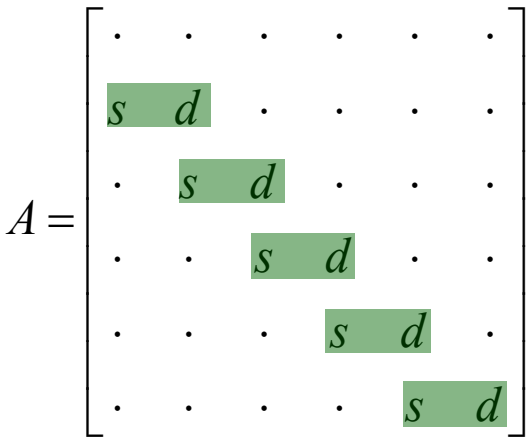
$$A=\begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ s & d & \cdot & \cdot & \cdot & \cdot \\ \cdot & s & d & \cdot & \cdot & \cdot \\ \cdot & \cdot & s & d & \cdot & \cdot \\ \cdot & \cdot & \cdot & s & d & \cdot \\ \cdot & \cdot & \cdot & \cdot & s & d \end{bmatrix} \qquad B=\begin{bmatrix} s & d & \cdot & \cdot & \cdot & \cdot \\ \cdot & s & d & \cdot & \cdot & \cdot \\ \cdot & \cdot & s & d & \cdot & \cdot \\ \cdot & \cdot & \cdot & s & d & \cdot \\ \cdot & \cdot & \cdot & \cdot & s & d \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

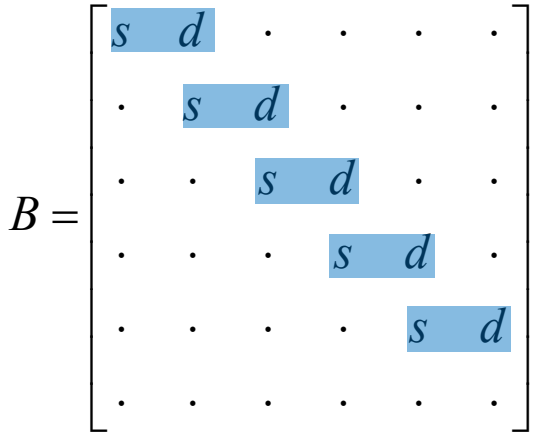
(a) Escribe el código necesario para realizar la comunicación (envío desde *P*0 y recepción en *P*1) utilizando un único mensaje. Es obligatorio el uso de tipos derivados. Se deberá usar la siguiente cabecera de función:

void comunica (double A[N][N], double B[N][N])

(b) Indica cuál es el coste de la comunicación.

Ejercicio 5 (Parcial 2021)






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

count=N-1
blocklength=2
stride=N+1

(a) Hay que definir un nuevo tipo de datos que nos servirá tanto para el envío como la recepción en un solo mensaje de todos los datos:

```
void comunica (double A[N][N], double B[N][N]){
    int rank;
    MPI_Datatype diags;
    MPI_Type_vector(N-1, 2, N+1, MPI_DOUBLE, &diags);
    MPI_Type_commit(&diags);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank==0)
        MPI_Send(&A[1][0], 1, diags, 1, 0, MPI_COMM_WORLD);
    else if (rank==1)
        MPI_Recv(&B[0][0], 1, diags, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    MPI_Type_free(&diags);
}
```

(b) Indica cuál es el coste de la comunicación:

$$t_c = t_s + 2(N-1)t_w$$

Ejercicio 6

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:

$$A(P_0) = \left(\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right) \Rightarrow \begin{cases} B(P_0) = \begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix}, B(P_1) = \begin{pmatrix} 3 & 4 \\ 7 & 8 \end{pmatrix} \\ B(P_2) = \begin{pmatrix} 9 & 10 \\ 13 & 14 \end{pmatrix}, B(P_3) = \begin{pmatrix} 11 & 12 \\ 15 & 16 \end{pmatrix} \end{cases}$$

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.

Ejercicio 6

$$A(P_0) = \begin{array}{c} N \\ N \end{array} \left[\begin{array}{ccccc|ccccc} x & x & L & x & x & x & x & L & x & x \\ x & x & L & x & x & x & x & L & x & x \\ M & M & O & M & M & M & M & O & M & M \\ x & x & L & x & x & x & x & L & x & x \\ x & x & L & x & x & x & x & L & x & x \\ \hline x & x & L & x & x & x & x & L & x & x \\ x & x & L & x & x & x & x & L & x & x \\ M & M & O & M & M & M & M & O & M & M \\ x & x & L & x & x & x & x & L & x & x \\ x & x & L & x & x & x & x & L & x & x \end{array}\right] \Rightarrow$$

$$B(P_0) = A_{00} = \begin{bmatrix} x & x & L & x & x \\ x & x & L & x & x \\ M & M & O & M & M \\ x & x & L & x & x \\ x & x & L & x & x \end{bmatrix} \quad B(P_1) = A_{01} = \begin{bmatrix} x & x & L & x & x \\ x & x & L & x & x \\ M & M & O & M & M \\ x & x & L & x & x \\ x & x & L & x & x \end{bmatrix}$$

$$B(P_2) = A_{10} = \begin{bmatrix} x & x & L & x & x \\ x & x & L & x & x \\ M & M & O & M & M \\ x & x & L & x & x \\ x & x & L & x & x \end{bmatrix} \quad B(P_3) = A_{11} = \begin{bmatrix} x & x & L & x & x \\ x & x & L & x & x \\ M & M & O & x & x \\ x & x & L & x & x \\ x & x & L & x & x \end{bmatrix}$$

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

Queremos en un solo envío, empaquetar y enviar los elementos del mismo color. El proceso receptor los almacenará en posiciones consecutivas de memoria

count=N
blocklength=N
stride=2N

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

Ejercicio 6

$$A(P_0) = \begin{array}{c} \\ N \\ N \end{array} \begin{array}{c} \begin{array}{cc} N & N \end{array} \\ \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] \end{array} \Rightarrow \begin{array}{ll} B(P_0) = A_{00} & B(P_1) = A_{01} \\ B(P_2) = A_{10} & B(P_3) = A_{11} \end{array}$$

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)
```

Ejercicio 7

Sea **A** un array bidimensional de números reales de doble precisión, de dimensión N×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:

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

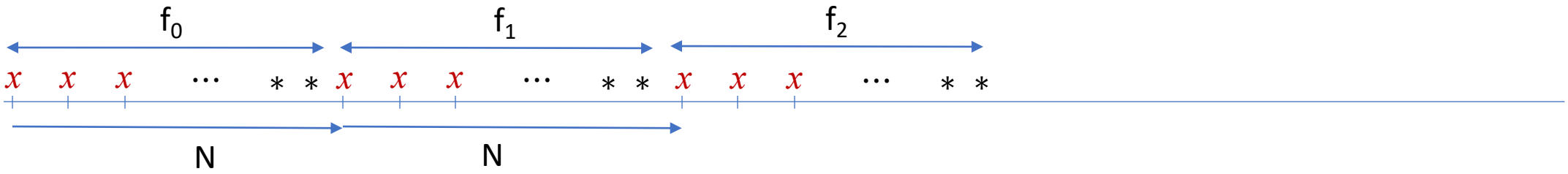
a) Realiza las correspondientes llamadas para el envío desde P_0 y la recepción en P_1 de la submatriz de la figura anterior.

b) 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 7

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

```
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

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

```
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, 100, MPI_COMM_WORLD);
}
else if (rank==1) {
   MPI_Recv(&A[0][0], 1, newtype, 0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
MPI_Type_free(&newtype);
```

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);
```

Relación entre índices globales e índices locales: Reparto por bloques de un conjunto de índices (Tr 50 de T3)

Indice global (i)	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)

$$\begin{array}{cc} i & \lfloor k \\ i_l & i_p \end{array} \Leftrightarrow \boxed{\begin{array}{l} i_p = i / k \\ i_l = i\%k \end{array}} \Leftrightarrow \boxed{i = ki_p + i_l}$$

Ejemplo: calcular el proceso que va a manejar el índice global 7 y cuál será el correspondiente índice local

$$i = 7, k = 4$$

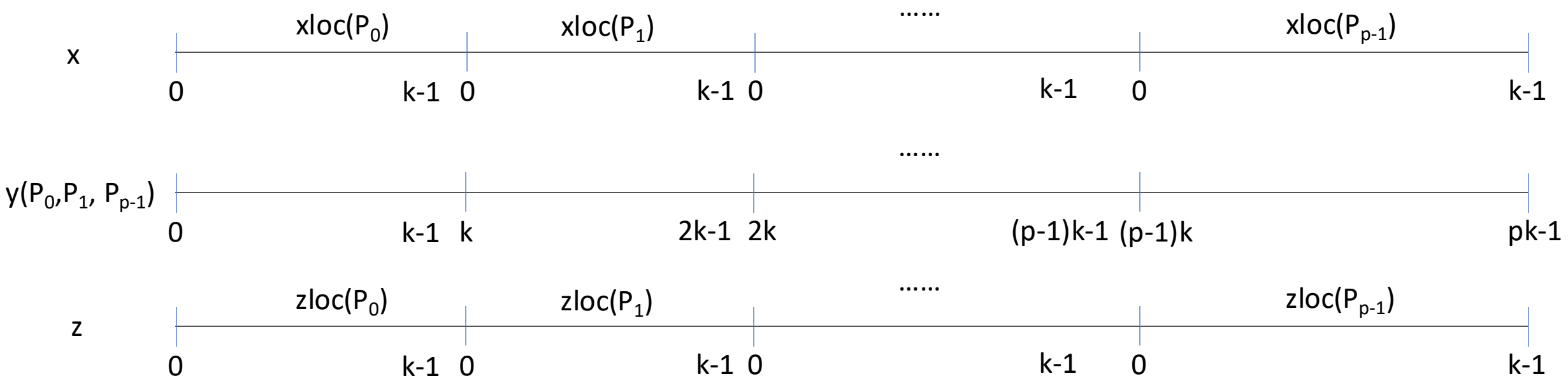
$$\begin{array}{cc} 7 & \lfloor 4 \\ 3 & 1 \end{array} \rightarrow \boxed{\begin{array}{l} i_p = 1 \\ i_l = 3 \end{array}}$$

Ejercicio 8

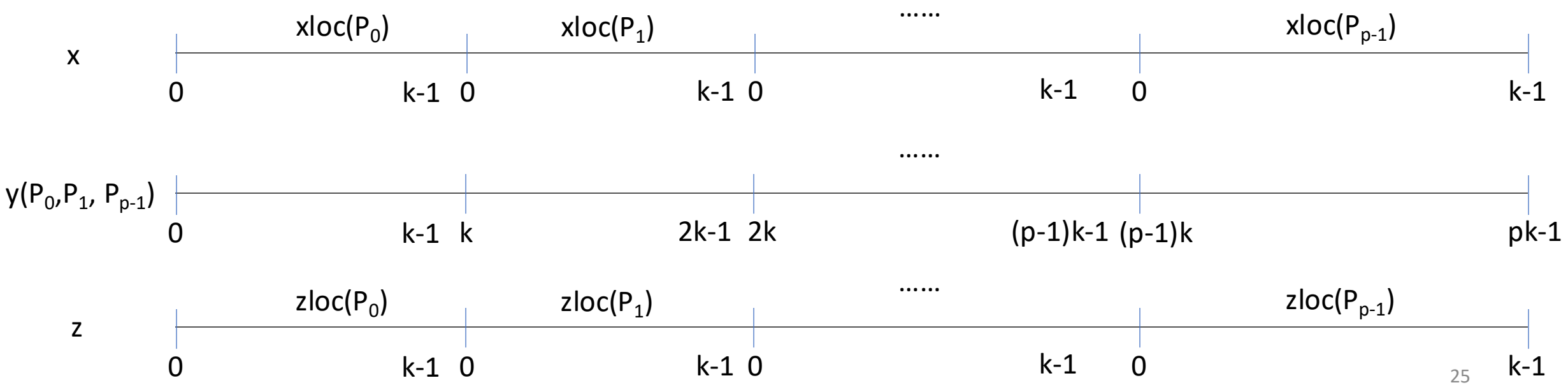
- Supongamos en un programa que un vector **x** de dimensión **N** se encuentra repartido por bloques entre todos los procesos mediante vectores locales **xloc** y un vector **y** replicado en todos los procesos.
- Implementa la función :

  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 9



25

$$i = ki_p + i_l$$

```
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];
}
```

Relación entre índices globales e índices locales: Reparto cíclico de un conjunto de índices

Indice global (i)	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)

$$\begin{array}{c|c} i & p \\ \hline i_p & i_l \end{array} \Leftrightarrow \boxed{\begin{array}{l} i_p = i\%p \\ i_l = i/p \end{array}} \Leftrightarrow \boxed{i = i_l p + i_p}$$

Ejemplo: calcular el proceso que va a manejar el índice global 7 y cuál será el correspondiente índice local

$$i = 7, k = 4$$

$$\begin{array}{c|c} 7 & 3 \\ \hline 1 & 2 \end{array} \rightarrow \boxed{\begin{array}{l} i_p = 1 \\ i_l = 2 \end{array}}$$

Ejercicio 10

- Supongamos en un programa que un vector **x** de dimensión **N** se encuentra **repartido cíclicamente** 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**.

……

$$i = pi_l + i_p$$

```
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 11

- 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**, **p** el número de procesos (**p** divide a **n**) e **ip** el índice de un proceso. Partiendo de la siguiente versión secuencial, implementa el código paralelo, en los siguientes casos:

a) El vector x es conocido por todos los procesos, la matriz **A** está repartida por bloques de filas y queremos que el vector **y** esté repartido por bloques de componentes

b) El vector x es conocido por todos los procesos, la matriz **A** tiene un reparto cíclico de filas y queremos que el vector **y** tenga un reparto cíclico de sus componentes

```
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];
    }
  }
}
```

Ejercicio 11

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 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:

$$i = pi_l + i_p$$

$$i_p = i\%p$$

$$i_l = i / p$$