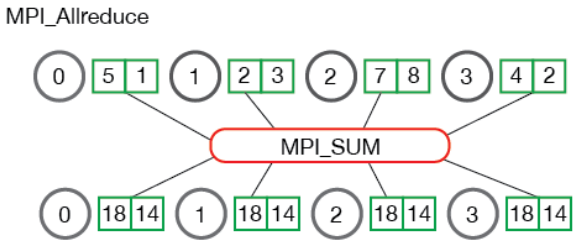
Comunicaciones colectivas:

Reducción **todos-a-todos**

```
int MPI_Allreduce ( *sendbuf,  *recvbuf, count, datatype, op, comm )
```

- **sendbuf:** Variable que contiene la información a comunicar.
- **recvbuf:** Variable que contiene la información a recibir.
- int **count:** Cantidad de elementos contenidos en sendbuf.
- MPI_Datatype **datatype:** Tipo de la variable sendbuf y recvbuf.
- MPI_Op **op:** Operación a ejecutar.
- MPI_Comm **comm:** Comunicador.



MPI_ALLREDUCE **combina los elementos almacenados en sendbuf** de cada proceso definido en el comunicador comm, utilizando la operación op, y regresa el resultado en **recvbuf** de cada proceso. Notar que tanto sendbuf como recvbuf deben tener el mismo número de elementos de tipo datatype; asimismo, todos los procesos involucrados en la operación deben llamar a esta función con el mismo valor de count, datatype, op.

La opción “in place” se especifica con el valor MPI_IN_PLACE en el argumento sendbuf en todos los procesos. En este caso, los datos de entrada se toman del buffer de recepción, recvbuf, donde serán reemplazados por los datos de salida.

Ejemplo 6

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv)  {
   int myrank, numprocs, i, m;
   double *x, *y;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   if (myrank == 0)
{
          printf("Longitud de los vectores: \n");
          scanf("%i",&m);
}
   MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
   x = (double *)malloc(m * sizeof(double));
   y = (double *)malloc(m * sizeof(double));
   for (i=0; i<m; i++) {
         x[i]=(myrank+1) * i;
   y[i]=0.0;  }
   printf("Soy %d. Antes de recibir el valor de x es:  ",myrank);
   for (i=0; i<m; i++) printf("%4.1f ",x[i]);
   printf("\n");
   MPI_Allreduce(x,y,m,MPI_DOUBLE,MPI_SUM, MPI_COMM_WORLD);
   printf("Soy %d. Despues de recibir el valor de y es:",myrank);
   for (i=0; i<m; i++) printf("%4.1f ",y[i]);
  printf("\n");
   free(x);  free(y);
MPI_Finalize();
}
```

De la misma forma que con `MPI_Reduce`, podríamos usar la misma variable x para la entrada-salida con la opción `MPI_IN_PLACE`:

```
MPI_Allreduce(MPI_IN_PLACE,x,m,MPI_DOUBLE,MPI_SUM,
MPI_COMM_WORLD);
```

Salida del Ejemplo 6

```
MPI_Allreduce(x,y,m,MPI_DOUBLE,MPI_SUM, MPI_COMM_WORLD);
```

Después de la ejecución (`mpirun -np 3 ejemplo6`), la salida que produce el *ejemplo6* es:

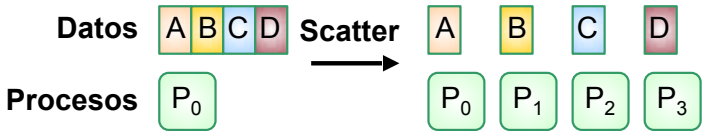
```
Longitud de los vectores:
10
Soy 0. Antes de recibir el valor de x es:   0.0  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0
Soy 1. Antes de recibir el valor de x es:   0.0  2.0  4.0  6.0  8.0 10.0 12.0 14.0 16.0 18.0
Soy 2. Antes de recibir el valor de x es:   0.0  3.0  6.0  9.0 12.0 15.0 18.0 21.0 24.0 27.0
Soy 0. Despues de recibir el valor de y es: 0.0  6.0 12.0 18.0 24.0 30.0 36.0 42.0 48.0 54.0
Soy 1. Despues de recibir el valor de y es: 0.0  6.0 12.0 18.0 24.0 30.0 36.0 42.0 48.0 54.0
Soy 2. Despues de recibir el valor de y es: 0.0  6.0 12.0 18.0 24.0 30.0 36.0 42.0 48.0 54.0
```

Comunicaciones colectivas:

Scatter/Gather

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

- **sendbuf:** Variable que contiene la información a comunicar.
- int **sendcnt:** Tamaño de los segmentos a comunicar.
- MPI_Datatype **sendtype:** Tipo de la variable sendbuf.
- **recvbuf:** Variable que contiene la información a recibir.
- int **recvcnt:** Cantidad de elementos contenidos en recvbuf.
- MPI_Datatype **recvtype:** Tipo de la variable recvbuf.
- int **root:** Número lógico del proceso que hace el envío y desde el cual se espera recibir información.
- MPI_Comm **comm:** Comunicador.



El proceso raíz (root) dispone del mensaje que es **dividido en segmentos de igual tamaño** (sendcnt). El i-ésimo segmento se envía al i-ésimo proceso del grupo (recvbuf). La **cantidad** de datos **enviados** tiene que ser **igual** a la cantidad de datos **recibidos** y debe coincidir en todos los procesos.

La opción “in place” se especifica con el valor MPI_IN_PLACE en el argumento recvbuf del proceso root. En este caso, recvcnt y recvtype se ignoran, y el proceso root no envía nada a sí mismo.

Ejemplo 7

```
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
   int myrank, numprocs, i, lm, m, root;
   double *x, *y;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   printf("Soy el proceso %d de un total de %d\n",myrank,numprocs);
   root = 0;
   if (myrank == root)
   {
        printf("Longitud de los vectores en cada proceso: \n");
            scanf("%i",&lm);
            m = lm * numprocs;
        printf("Longitud total de los vectores : %i\n",m);
        x = (double *)malloc(m * sizeof(double));
        y = (double *)malloc(m * sizeof(double));
        for (i=0; i<m; i++)
        {
            x[i]= i+1;
            y[i]=2*(i+1);
        }
   }
```

Ejemplo 7

```
  MPI_Bcast(&lm  // Referencia al vector donde se almacena/envia
        ,1      // numero de elementos maximo a recibir
        ,MPI_INT  // Tipo de dato
        ,0      // numero del proceso root
        ,MPI_COMM_WORLD); // Comunicador por el que se recibe
    if (myrank != root)
    {
x = (double *)malloc(lm * sizeof(double));
y = (double *)malloc(lm * sizeof(double));

    }
if (myrank != root) {
    MPI_Scatter(x,lm,MPI_DOUBLE,
                      x,lm,MPI_DOUBLE,
                      root,MPI_COMM_WORLD);
    MPI_Scatter(y,lm,MPI_DOUBLE,
                      y,lm,MPI_DOUBLE,
                      root,MPI_COMM_WORLD);
    }
    else {
    MPI_Scatter(x,lm,MPI_DOUBLE,
                      MPI_IN_PLACE,lm,MPI_DOUBLE,
                      root,MPI_COMM_WORLD);
    MPI_Scatter(y,lm,MPI_DOUBLE,
                      MPI_IN_PLACE,lm,MPI_DOUBLE,
                      root,MPI_COMM_WORLD);
    }
```

```
    printf("Soy %d. Despues de scatter el valor de x es:",myrank);
    for (i=0; i<10; i++) printf("%4.1f ",x[i]);
    printf("\n");
    printf("Soy %d. Despues de scatter el valor de y es:",myrank);
    for (i=0; i<10; i++) printf("%4.1f ",y[i]);
    printf("\n");



    free(x);
    free(y);
    MPI_Finalize();
}
```

Salida del Ejemplo 7

- Después de la ejecución (`mpirun -np 2 ejemplo7`), la salida que produce el *ejemplo7* es:

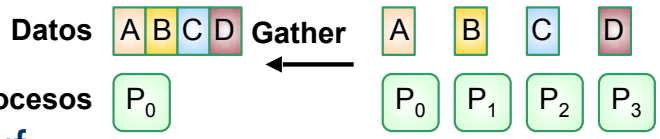
```
Soy el proceso 0 de un total de 2
Longitud de los vectores en cada proceso:
Soy el proceso 1 de un total de 2
5
Longitud total de los vectores : 10
Soy 0. Despues de scatter el valor de x es: 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
Soy 0. Despues de scatter el valor de y es: 2.0  4.0  6.0  8.0 10.0 12.0 14.0 16.0 18.0 20.0
Soy 1. Despues de scatter el valor de x es: 6.0  7.0  8.0  9.0 10.0  nan  0.0  0.0  0.0  0.0
Soy 1. Despues de scatter el valor de y es: 12.0 14.0 16.0 18.0 20.0  0.0  0.0  0.0  0.0  0.0
```

Comunicaciones colectivas:

Scatter/Gather

int MPI_Gather(*sendbuf, sendcnt, sendtype, *recvbuf, recvcnt, recvtype, root, comm)

- **sendbuf:** Variable que contiene la información a comunicar.
- int **sendcnt:** Tamaño de los segmentos a comunicar.
- MPI_Datatype **sendtype:** Tipo de la variable sendbuf.
- **recvbuf:** Variable que contiene la información a recibir.
- int **recvcnt:** Cantidad de elementos contenidos en recvbuf.
- MPI_Datatype **recvtype:** Tipo de la variable recvbuf.
- int **root:** Número lógico del proceso que espera recibir la información.
- MPI_Comm **comm:** Comunicador.



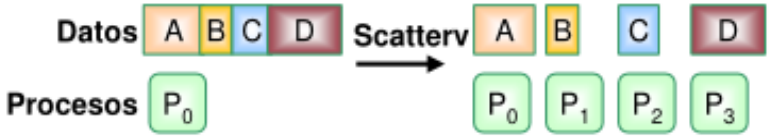
Cada proceso (incluido el root) envía el contenido de su buffer de envío al proceso root. El proceso raíz recibe los mensajes y los **almacena por orden del número de proceso**. El buffer de recepción (recvbuf) es ignorado en todos los procesos distintos del root. La cantidad de datos enviados tiene que ser igual a la cantidad de datos recibidos y debe coincidir en todos los procesos.

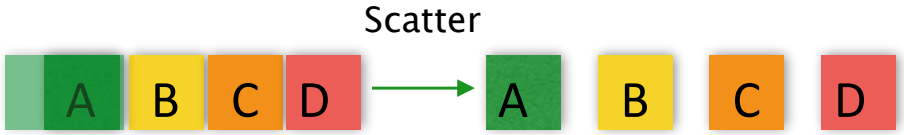
La opción “in place” se especifica con el valor MPI_IN_PLACE en el argumento sendbuf del proceso root. En este caso, sendcnt y sendtype se ignoran, y la contribución del proceso root al vector de salida se asume que está en su lugar correcto del buffer de recepción recvbuf.

Comunicaciones colectivas:

Scatter/Gather

- Con MPI_SCATTER un proceso raíz **trocea un mensaje en partes iguales** y los envía individualmente al resto de procesos y a sí mismo.
- MPI_SCATTERV extiende la funcionalidad de MPI_SCATTER, permitiendo **variar el tamaño de los datos** que se mandan a cada proceso.
- No obstante nos las podemos arreglar para utilizar MPI_SCATTER cuando los trozos del mensaje son todos iguales salvo uno de ellos.

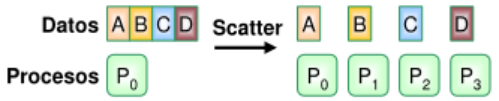




Podemos enviar sólo una parte de A si el proceso ya dispone del resto de A apuntando a la posición de A adecuada.

Salida del Ejemplo 8

- Sabemos que con MPI_SCATTER un proceso raíz **trocea un mensaje en partes iguales** y los envía individualmente al resto de procesos y a sí mismo.



- En el ejemplo 8 el proceso 0 genera una matriz 7x10 y efectúa un scatter. Si ejecutamos el ejemplo con 2 procesos vemos que la matriz se reparte de forma correcta a pesar de que hay 7 filas y 2 procesos (notar que no podemos dividir 7 filas en dos chunks de filas iguales). Revisa el código y analiza porque sucede esto.
- Notar que un comportamiento similar se observa con MPI_GATHER. Al ejecutar el ejemplo8 con dos procesos, el proceso 0 genera una matriz 4x10, el proceso 1 una matriz 3x10, y a pesar de ser matrices de tamaño distinto al efectuar un gather se obtiene de forma correcta una matriz 7x10.

Ejemplo 8

```
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
#define m 7
#define n 10
int main(int argc, char **argv)
{
     void vermatriz(int max, double a[][max], int ,
int , char []);
   int myrank, numprocs, i, j, lm, root, resto, slice;
   double A[m][n], B[m][n];
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   root = 0;
   if (myrank == root) {
       resto = m % numprocs;
       for (i=0; i<m; i++)
          for (j=0; j<n; j++)
             A[i][j] = i-j;
       vermatriz(n,A,m,n,"A");
}
   else {
       resto = 0;
   }
   slice = m / numprocs;
   lm = slice + resto;
   if (myrank != root) {
      MPI_Scatter(&A[resto][0],slice*n,MPI_DOUBLE,
               &A[resto][0],slice*n,MPI_DOUBLE,
               root,MPI_COMM_WORLD); }
   else {
      MPI_Scatter(&A[resto][0],slice*n,MPI_DOUBLE,
               MPI_IN_PLACE,slice*n,MPI_DOUBLE,
               root,MPI_COMM_WORLD); }
   printf("Proceso %d\n",myrank);
   vermatriz(n,A,lm,n,"A");
   for (i=0; i<lm; i++)
       for (j=0; j<n; j++)
           B[i][j] = 2*A[i][j];
   if (myrank != root) {
      MPI_Gather(&B[resto][0],slice*n,MPI_DOUBLE,
               &B[resto][0],slice*n,MPI_DOUBLE,
               0, MPI_COMM_WORLD); }
   else {
      MPI_Gather(MPI_IN_PLACE,slice*n,MPI_DOUBLE,
               &B[resto][0],slice*n,MPI_DOUBLE,
               0, MPI_COMM_WORLD); }
   if (myrank == 0){
      vermatriz(n,B,m,n,"B"); }
   MPI_Finalize();
}
```

El proceso 0 define la matriz

El proceso 0 esparce la matriz

Esta parte la hacen todos. Cada proceso genera su parte de B.

Con el Gather se juntan los diversos trozos de B en el proceso 0

Salida del Ejemplo 8

- Después de la ejecución (`mpirun -np 2 ejemplo8`), la salida que produce el *ejemplo8* es:

```
  A=     0       1       2       3       4       5       6       7       8       9
      0:  0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000  -8.000  -9.000
      1:  1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000  -8.000
      2:  2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000
      3:  3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000
      4:  4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000
      5:  5.000   4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000
      6:  6.000   5.000   4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000

Proceso 0
    A=     0       1       2       3       4       5       6       7       8       9
       0:  0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000  -8.000  -9.000
       1:  1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000  -8.000
       2:  2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000
       3:  3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000

Proceso 1
    A=     0       1       2       3       4       5       6       7       8       9
       0:  4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000
       1:  5.000   4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000
       2:  6.000   5.000   4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000

    B=     0       1       2       3       4       5       6       7       8       9
       0:  0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000 -16.000 -18.000
       1:  2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000 -16.000
       2:  4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000
       3:  6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000
       4:  8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000
       5: 10.000   8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000
       6: 12.000  10.000   8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000
```

Salida del Ejemplo 8

- Después de la ejecución (`mpirun -np 3 ejemplo8`), la salida que produce el *ejemplo8* es:

```
Proceso 0
   A=       0       1       2       3       4       5       6       7       8       9
      0:  0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000  -8.000  -9.000
      1:  1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000  -8.000
      2:  2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000
   B=       0       1       2       3       4       5       6       7       8       9
      0:  0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000 -16.000 -18.000
      1:  2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000 -16.000
      2:  4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000
      3:  6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000
      4:  8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000
      5: 10.000   8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000
      6: 12.000  10.000   8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000
Proceso 1
   A=       0       1       2       3       4       5       6       7       8       9
      0:  3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000
      1:  4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000
Proceso 2
   A=       0       1       2       3       4       5       6       7       8       9
      0:  5.000   4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000
      1:  6.000   5.000   4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000
```

Salida del Ejemplo 8

```
Soy el proceso 0 de un total de 2
    A=     0       1       2       3       4       5       6       7       8       9
       0:  0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000  -8.000  -9.000
       1:  1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000  -8.000
       2:  2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000
       3:  3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000
       4:  4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000
       5:  5.000   4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000
       6:  6.000   5.000   4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000
```

scatter

```
Proceso 0
    A=     0       1       2       3       4       5       6       7       8       9
       0:  0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000  -8.000  -9.000
       1:  1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000  -8.000
       2:  2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000
       3:  3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000
```

```
Proceso 1
    A=     0       1       2       3       4       5       6       7       8       9
       0:  4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000
       1:  5.000   4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000
       2:  6.000   5.000   4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000
```

```
Proceso 0

        0       1       2       3       4       5       6       7       8       9
    0:  0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000 -16.000 -18.000
    1:  2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000 -16.000
    2:  4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000
    3:  6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000

Proceso 1

        0       1       2       3       4       5       6       7       8       9
    4:  8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000
    5: 10.000   8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000
    6: 12.000  10.000   8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000
```

gather

```
B=     0       1       2       3       4       5       6       7       8       9
   0:  0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000 -16.000 -18.000
   1:  2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000 -16.000
   2:  4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000
   3:  6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000
   4:  8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000
   5: 10.000   8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000
   6: 12.000  10.000   8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000
```

Comunicaciones colectivas:

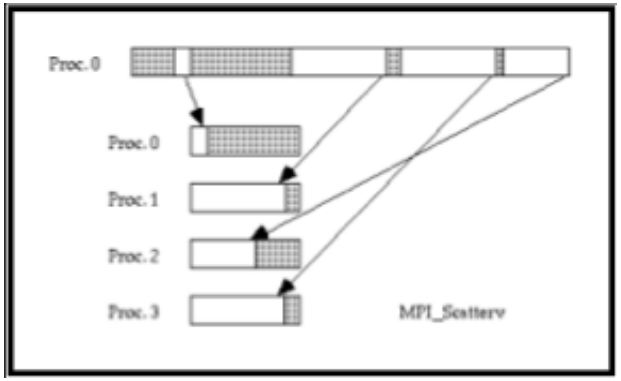
- Otras versiones de estas funciones son:
  - **MPI_Scatterv**(…); la información que se distribuye es de tamaño variable.
  - **MPI_Gatherv**(…); la información que se recolecta es de tamaño variable.
  - **MPI_Allgatherv**(…); “suma” de las dos anteriores.

Comunicaciones colectivas:

Scatter/Gather

MPI_Scatterv :

○ Es similar a MPI_Scatter excepto que permite gaps entre los datos a transferir y longitud variable de datos



Comunicaciones colectivas:

Scatter/Gather

```
int MPI_Scatterv( const void *sendbuf, const int *sendcounts, const int
*displs, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm)
```

- sendbuf: Variable que contiene la información a comunicar.
- sendcounts: Array de tipo entero (de longitud igual al tamaño del grupo) indicando el número de elementos a enviar a cada proceso.
- displs: Array de tipo entero (de longitud igual al tamaño del grupo). La entrada i indica el desplazamiento (relativo a sendbuf) desde el que se debe tomar los datos para enviar al proceso i.
- sendtype: Tipo de la variable sendbuf.
- recvbuf: Variable que contiene la información a recibir.
- recvcount: Cantidad de elementos contenidos en recvbuf.
- recvtype: Tipo de la variable recvbuf.
- root: Número lógico del procesador que envía la información.
- comm: Comunicador.

MPI_Scatterv extiende la funcionalidad de MPI_SCATTER, permitiendo variar el tamaño de los datos que se mandan a cada proceso (sendcounts). Permite mayor flexibilidad para determinar de dónde se toman los datos desde el proceso root (displs).

Comunicaciones colectivas:

Scatter/Gather

Ejemplo ejscatterv.c

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

#define m 7
#define n 10

int main(int argc, char **argv)
{
    void vermatriz(int max, double a[][max], int , int , char []);

    int myrank, numprocs, i, j, lm, root, resto, slice;
    double A[m][n], D[m][n];
    int *a_chunk_sizes;
    int *a_despla;


    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    a_chunk_sizes=(int *)malloc(numprocs*sizeof(int));
    a_despla=(int *)malloc(numprocs*sizeof(int));


    root = 0;

    if (myrank == root)
    {
       for (i=0; i<m; i++)
          for (j=0; j<n; j++)
             A[i][j] = i-j;
       vermatriz(n,A,m,n,"A");
    }
    MPI_Barrier(MPI_COMM_WORLD);

    slice = m/numprocs;
    resto = m % numprocs;

    for(i=1; i<=numprocs-1; i++) {
        a_chunk_sizes[i]=slice*n;
        a_despla[i] = resto*n + i*slice*n;
    }

    a_chunk_sizes[0]=(slice+resto)*n;
    a_despla[0] = 0;

    printf("Proceso %d, a_chunk_sizes %d , a_despla %d \n", myrank, a_chunk_sizes[myrank], a_despla[myrank]);

    MPI_Scatterv(&A[0][0],a_chunk_sizes,&a_despla[0],MPI_DOUBLE,
                 &D[0][0],a_chunk_sizes[myrank],MPI_DOUBLE,
                          root,MPI_COMM_WORLD);

    lm = a_chunk_sizes[myrank]/n;
    printf("\n");
    vermatriz(n,D,lm,n,"D");
    printf("\n");

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Finalize();
}
void vermatriz(int max, double a[][max], int fil, int col, char
nombre[]){
int i,j;
printf("%5s=",nombre);
printf("%6d ",0);
for (j=1;j<col;j++){
   printf("%7d ",j);
}
printf("\n");
for (i=0;i<fil;i++){
   printf("%8d:",i);
   for (j=0;j<col;j++){
      printf("%7.3f ",a[i][j]);
   }
   printf("\n");
}
}
```

Ejemplo ejscatterv2.c

```c
/* Desasignamos un puntero a un array 2d*/
void deallocate_array(double **array, int row_dim)
{
  int i;
  for(i=1; i<row_dim; i++)
    array[i]=NULL;
  free(array[0]);
  free(array);
}

/* Asignamos un puntero a un array 2d*/
double **allocate_array(int row_dim, int col_dim)
{
  double **result;
  int i;
  /* Necesitamos ir con cuidado: el array debe ser asignado a un trozo contiguo de memoria, para que MPI pueda distribuirlo correctamente. */

  result=(double **)malloc(row_dim*sizeof(double *));
  result[0]=(double *)malloc(row_dim*col_dim*sizeof(double));
  for(i=1; i<row_dim; i++)
    result[i]=result[i-1]+col_dim;
  return result;
}
```

```c
a_chunk_sizes=(int *)malloc(numprocs*sizeof(int));
a_despla=(int *)malloc(numprocs*sizeof(int));

// El padre pide por pantalla el tamaño de la matriz
…
// Se envian los tamaños de la matriz a todos los hijos
    MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);


    A = allocate_array(m, n);

// Se inicializa la matriz
    inicializarEstructuras(m, n, A);


/* Se calculan los trozos correspondientes a cada nodo
    dividiendo la carga equitativamente. Si la division no
    es entera, el padre se queda con el resto de la carga */
    slice = m/numprocs;
    resto = m % numprocs;
/* Se calcula para cada nodo el numero de elementos de cada
    trozo, y el correspondiente desplazamiento que tiene que
    tener en cuenta la operacion Scatterv.*/
    for(i=1; i < numprocs; i++) {
        a_chunk_sizes[i] = slice*n;
        a_despla[i] = resto*n + i*slice*n;
      }

    a_chunk_sizes[0]=(slice+resto)*n;
    a_despla[0] = 0;
```

Ejemplo ejscatterv2.c

```
if(pid != 0) {
     resto = 0;
   }

   D = allocate_array(slice + resto, n);

   // Se realiza el troceo de la matriz A. Cada nodo recibe su trozo correspondiente en la matriz D.
   MPI_Scatterv(&A[0][0],a_chunk_sizes,&a_despla[0],MPI_DOUBLE,
                &D[0][0],a_chunk_sizes[pid],MPI_DOUBLE,root,MPI_COMM_WORLD);

   lm = a_chunk_sizes[pid]/n;


   printf("Soy el proceso %d.\n", pid);
   vermatriz(D,lm,n,"A");
   printf("\n");

   deallocate_array(A, m);

   if(pid == root) {
           printf("\n");
                       deallocate_array(D, slice + resto);
                       }
   else{
     deallocate_array(D, slice);
   }
  MPI_Finalize();
```

Comunicaciones colectivas:

Scatter/Gather

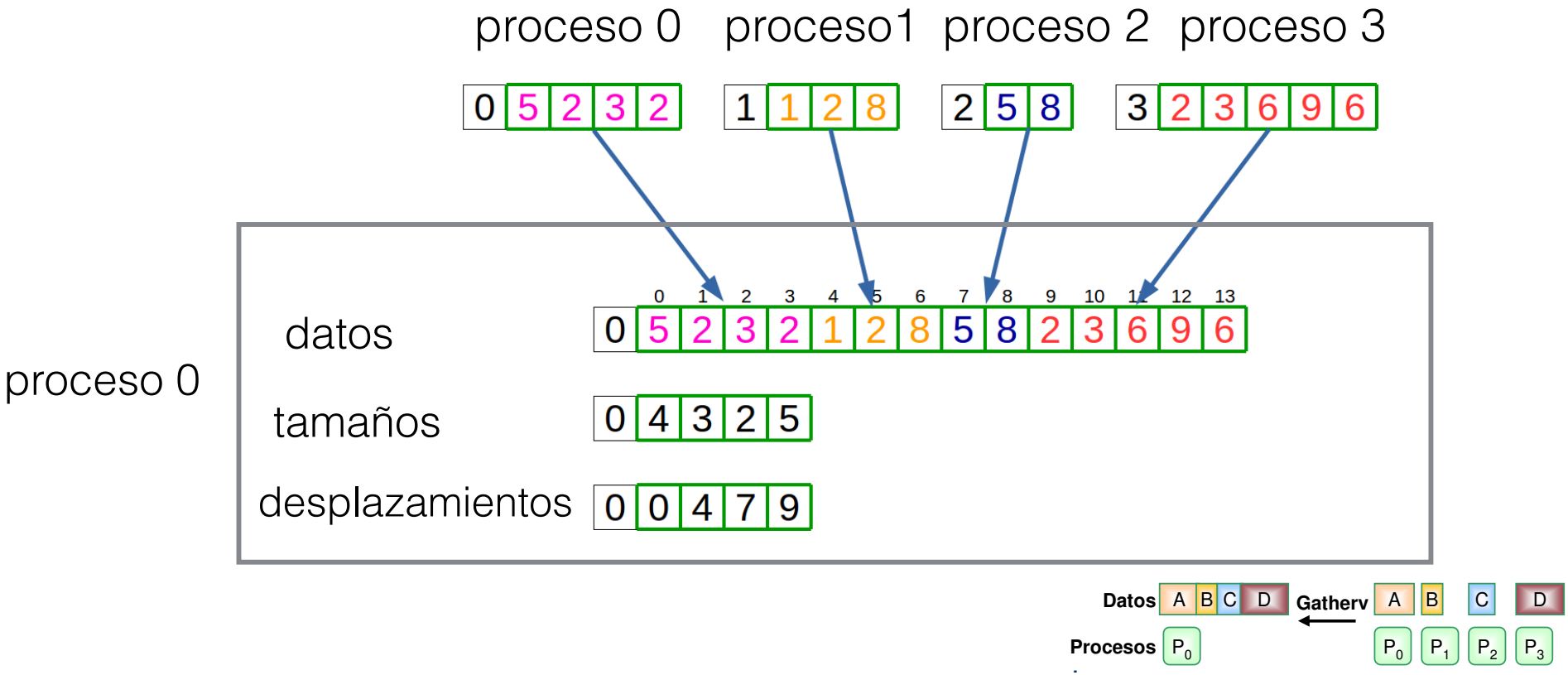
```
int MPI_Gatherv (void *sendbuf, int *sendcounts, MPI_Datatype sendtype,
void *recvbuf, int recvcounts, int *displs, MPI_Datatype recvtype, int
root, MPI_Comm comm)
```

- **sendbuf:** Variable que contiene la información a comunicar.
- **sendcount:** Cantidad de elementos contenidos en sendbuf. Tener en cuenta que puede variar en cada proceso.
- **sendtype:** Tipo de la variable sendbuf.
- **recvbuf:** Variable que contiene la información a recibir.
- **recvcounts:** Array de tipo entero (de longitud igual al tamaño del grupo) que contiene el número de elementos recibidos desde cada proceso.
- **displs: A**rray de tipo entero (de longitud igual al tamaño del grupo). La entrada i indica el desplazamiento (relativo a recvbuf) a partir del cual se colocarán los datos recibidos desde el proceso i.
- **recvtype:** Tipo de la variable recvbuf.
- **root:** Número lógico del procesador que recibe la información.
- **comm:** Comunicador.

MPI_Gatherv recoge mensajes individuales desde cada uno de los procesos en comm y distribuye el mensaje resultante al proceso con identificador root. Los mensajes individuales pueden tener diferentes tamaños (recvcounts) y desplazamientos (displs).

Comunicaciones colectivas:

Scatter/Gather



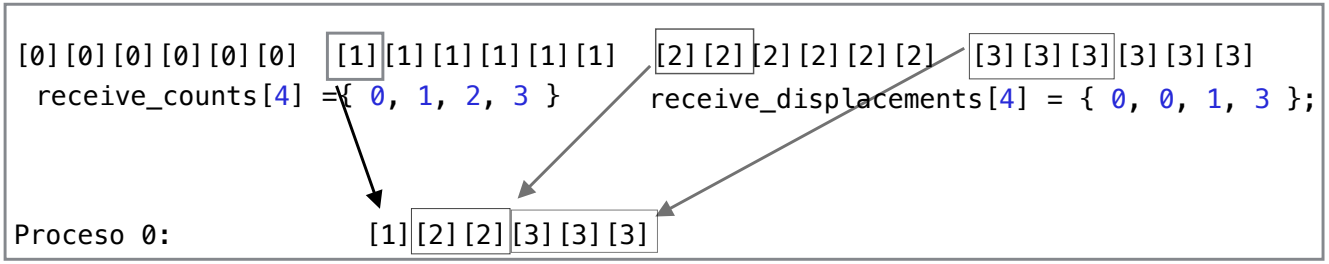
MPI_Gatherv recoge mensajes individuales desde cada uno de los procesos en comm y distribuye el mensaje resultante al proceso con identificador root. Los mensajes individuales pueden tener diferentes tamaños (recvcounts) y desplazamientos (displs).

Comunicaciones colectivas:

Scatter/Gather

ejgatherv.c

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

int main(int argc, char *argv[])
{
    int buffer[6];
    int rank, size, i;
    int receive_counts[4] = { 0, 1, 2, 3 };
    int receive_displacements[4] = { 0, 0, 1, 3 };

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (size != 4)
    {
        if (rank == 0)
        {
            printf("Please run with 4 processes\n");
        }
        MPI_Finalize();
        return 0;
    }
    for (i=0; i<6; i++)
    {
        buffer[i] = rank;
    }
    printf("Proceso %d :", rank);
        for (i=0; i<6; i++)
        {
            printf("[%d]", buffer[i]);
        }
        printf("\n");
    MPI_Gatherv(buffer, receive_counts[rank], MPI_INT, buffer, receive_counts, receive_displacements, MPI_INT, 0, MPI_COMM_WORLD);
    if (rank == 0)
    {
        printf("Proceso %d :", rank);
        for (i=0; i<6; i++)
        {
            printf("[%d]", buffer[i]);
        }
        printf("\n");
    }
    MPI_Finalize();
    return 0;
}
```



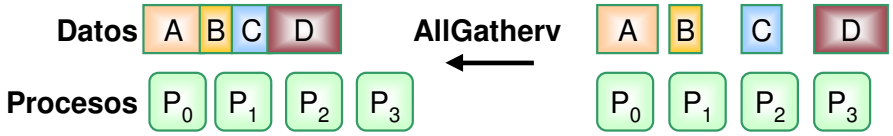

```
mpirun -np 4 ./ejgatherv

Proceso 0 :[0][0][0][0][0][0]
Proceso 1 :[1][1][1][1][1][1]
Proceso 2 :[2][2][2][2][2][2]
Proceso 3 :[3][3][3][3][3][3]

Proceso 0 :[1][2][2][3][3][3]
```

Comunicaciones colectivas:

Scatter/Gather

```
int MPI_Allgatherv(const void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                   void *recvbuf, const int *recvconts, const int *displs,
                   MPI_Datatype recvtype, MPI_Comm comm)
```

- **sendbuf:** Variable que contiene la información a comunicar.
- **sendcnt:** Cantidad de elementos contenidos en sendbuf.
- **sendtype:** Tipo de la variable sendbuf.
- **recvbuf:** Variable que contiene la información a recibir.
- **recvcnts:** Array de tipo entero (de longitud igual al tamaño del grupo) que contiene el número de elementos recibidos desde cada proceso.
- **displs:** Array de tipo entero (de longitud igual al tamaño del grupo). La entrada i indica el desplazamiento (relativo a recvbuf) a partir del cual se colocarán los datos recibidos desde el proceso i.
- **recvtype:** Tipo de la variable recvbuf.
- **comm:** Comunicador.



MPI_Allgatherv recoge mensajes individuales desde cada uno de los procesos en comm y distribuye el mensaje resultante a todos los procesos. Los mensajes individuales pueden tener diferentes tamaños (recvcnts) y desplazamientos (displs).