

Funciones básicas: Comunicación (Blocking Standard Send)

Envío de una columna en una matriz (caso C)

```
double A[100][200];
```

- **Solución 1:** Enviar uno a uno los elementos de la columna.

```
for (i=0; i<100; i++)  
    MPI_SEND(&A[i][0], 1, MPI_DOUBLE, dest, tag, comm);
```

- **Solución 2:** Empaquetar los datos en un vector.

```
double c[100];  
for (i=0; i<100; i++) c[i]=A[i][0];  
MPI_SEND(&c, 100, MPI_DOUBLE, dest, tag, comm);
```

Funciones básicas: Comunicación (Blocking Standard Send)

Envío de una fila de una matriz (caso C)

```
double A[100][200];
```

- **Solución 1:** Enviar uno a uno los elementos de la columna.

```
for (j=0; j<200; j++)
```

```
    MPI_SEND(&A[0][j], 1, MPI_DOUBLE, dest, tag, comm);
```

- **Solución 2:** Empaquetar los datos en un vector.

```
double c[200];
```

```
for (j=0; j<200; j++) c[j]=A[0][j];
```

```
    MPI_SEND(&c, 200, MPI_DOUBLE, dest, tag, comm);
```

- **Solución 3:** Un solo send, por ser posiciones contiguas de memoria

```
    MPI_SEND(&A[0][0], 200, MPI_DOUBLE, dest, tag, comm);
```

Ejemplo 3

(envío de un bloque consecutivo de columnas)

```
#include <stdio.h>
#include <mpi.h>
#define maxm 1000
#define maxn 900
int main(int argc, char **argv)
{
    int myrank,numprocs;
    int i,j,k,indice,columna_ini;
    int m,col_por_proceso;
    double A[maxm][maxn], aux;
    double start_time,end_time,total_time;
    MPI_Status estado;
    int request;
    int MPI_COL;

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

Definimos A
tamaño máximo de la
matriz: 1000 filas
y 900 columnas

Ejemplo 3

(envío de un bloque consecutivo de columnas)

```
if (myrank == 0) {  
    printf("Numero de columnas a cada proceso: \n");  
    scanf("%d", &col_por_proceso);  
    m = col_por_proceso*numprocs;  
    printf("El orden de la matriz es: %d\n", m);  
    for (i=0; i<m; i++) {  
        for (j=0; j<m; j++) {  
            A[i][j]=i+j;  
        }  
    }  
    for (i=1; i<numprocs; i++) {  
        MPI_Send(&m, 1, MPI_INT, i, 0, MPI_COMM_WORLD);  
        MPI_Send(&col_por_proceso, 1, MPI_INT, i, 0, MPI_COMM_WORLD);  
    }  
}  
  
else {  
    MPI_Recv(&m, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &estado);  
    MPI_Recv(&col_por_proceso, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &estado);  
}
```

El proceso 0 pide número de columnas enviadas a cada proceso

El tamaño de la matriz: m x m

El proceso 0 envía al resto el número de columnas asignadas a cada proceso y el orden de la matriz m

Ejemplo 3

(envío de un bloque consecutivo de columnas)

```

if (myrank == 0) {
    start_time = MPI_Wtime();
    columna_ini=col_por_proceso; // OPCION 1 (elemento a elemento)
    for (k=1; k<numprocs; k++) {
        for (i=0; i<m; i++) {
            for (j=columna_ini; j<columna_ini+col_por_proceso; j++) {
                MPI_Send(&A[i][j],1,MPI_DOUBLE,k,9,MPI_COMM_WORLD);
            }
            columna_ini = columna_ini + col_por_proceso;
        }
    }
    end_time = MPI_Wtime();
    total_time=end_time-start_time;
    printf("TIEMPO OPCION 1: %f\n",total_time); }

else {
    for (i=0; i<m; i++)
        for (j=0; j<col_por_proceso; j++)
            MPI_Recv(&A[i][j],1,MPI_DOUBLE,0,9,MPI_COMM_WORLD,&estado);
}

```

El proceso 0 envía al resto el elemento A[i][j].

La columna inicial asociada a cada proceso

Ejemplo 3

(envío de un bloque consecutivo de columnas)

```
if (myrank == 0) {
    start_time = MPI_Wtime();
    columna_ini=col_por_proceso;    // OPCION 2 (fila a fila)
    for (k=1; k<numprocs; k++) {
        for (i=0; i<m; i++)
            MPI_Send(&A[i][columna_ini],col_por_proceso,MPI_DOUBLE,k,9,
                    MPI_COMM_WORLD);
        columna_ini = columna_ini + col_por_proceso;
    }
    end_time = MPI_Wtime();
    total_time=end_time-start_time;
    printf("TIEMPO OPCION 2: %f\n",total_time);
}
else {
    for (i=0; i<m; i++)
        MPI_Recv(&A[i][0],col_por_proceso,MPI_DOUBLE,0,9,MPI_COMM_WORLD,&estado);
}
MPI_Finalize();
}
```

Ejemplo 3

(envío de un bloque consecutivo de columnas)

Compilación: `mpicc -o ejemplo3 ejemplo3.c`

Ejecución: `mpirun -np 3 ./ejemplo3`

Salida:

Soy el proceso 0 de un total de 3.

Soy el proceso 1 de un total de 3.

Soy el proceso 2 de un total de 3.

Numero de columnas a cada proceso:

100

El orden de la matriz es: 300

columna_ini 200

columna_ini 300

TIEMPO OPCION 1: 0.045204

TIEMPO OPCION 2: 0.001056

Ejemplo envioarray.c

(envío de los elementos de un array)

```
if (myrank == 0) {
    for (i=0; i<n; i++) {
        x[i] = i;
        y[i] = (double)1/(i+1);
    }
    j = n/numprocs;
    ln = j + n%numprocs;
    indice = ln;
    for (i=1; i<numprocs; i++) {
        MPI_Send( &x[indice], j, MPI_DOUBLE, i, i, MPI_COMM_WORLD);
        MPI_Send( &y[indice], j, MPI_DOUBLE, i, i, MPI_COMM_WORLD);
        indice = indice + j;
    }
}
else {
    ln = n/numprocs;
    MPI_Recv ( x, ln, MPI_DOUBLE, 0, myrank, MPI_COMM_WORLD, &status);
    MPI_Recv ( &y[0], ln, MPI_DOUBLE, 0, myrank, MPI_COMM_WORLD, &status);
    {
```


Ejemplo envioarray.c

(envío de los elementos de un array)

```
MPI_Send( &x[indice], j, MPI_DOUBLE, i, i, MPI_COMM_WORLD);
MPI_Send( &y[indice], j, MPI_DOUBLE, i, i, MPI_COMM_WORLD);
```

```
mpicc -o envioarray envioarray.c
```

```
mpirun -np 2 envioarray
```

Numero de elementos en los vectores originales:

10

Se apunta al indice que le corresponde a cada proceso

Soy 0. Antes del send/rcv el valor de x es:	1.00	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00	10.00
Soy 0. Antes del send/rcv el valor de y es:	1.00	0.50	0.33	0.25	0.20	0.17	0.14	0.12	0.11	0.10
Soy 1. Antes del send/rcv el valor de x es:	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Soy 1. Antes del send/rcv el valor de y es:	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	-539586854931141752752174417313792.00	0.00								

Las componentes no inicializadas toman valores indefinidos, no necesariamente 0!!!

Soy 0. Despues del send/rcv el valor de x es:	1.00	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00	10.00
Soy 0. Despues del send/rcv el valor de y es:	1.00	0.50	0.33	0.25	0.20	0.17	0.14	0.12	0.11	0.10
Soy 1. Despues del send/rcv el valor de x es:	6.00	7.00	8.00	9.00	10.00	0.00	0.00	0.00	0.00	0.00
Soy 1. Despues del send/rcv el valor de y es:	0.17	0.14	0.12	0.11	0.10	0.00	0.00	0.00	0.00	0.00
	-1724388504617060276179535712584734345497678989484920151089377101349912576.00	0.00								

En este caso se empieza a recibir en el indice 0

```
MPI_Recv ( x, ln, MPI_DOUBLE, 0, myrank, MPI_COMM_WORLD, &status);
MPI_Recv ( &y[0], ln, MPI_DOUBLE, 0, myrank, MPI_COMM_WORLD, &status);
```

Ejemplo memdinamica.c

(envío de los elementos de un array Asignación dinámica de memoria)

```
#include <stdlib.h> // la utilizamos para asignar memoria dinamicamente
...
double *x, *y; //creamos variables puntero de tipo doble a las cuales posteriormente asignaremos memoria dinamicamente
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
if (myrank == 0) {
    printf("Numero de elementos en los vectores: \n");
    scanf("%d", &n);
    for (i=1;i<numprocs;i++)
        MPI_Send( &n, 1, MPI_INT, i, 0, MPI_COMM_WORLD); }
else
    MPI_Recv ( &n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
if (myrank == 0) {
    x = malloc(n * sizeof(double)); // asignamos memoria dinamicamente con la funcion malloc perteneciente a stdlib.h
    y = malloc(n * sizeof(double));
    for (i=0; i<n; i++) {
        x[i] = i+1;
        y[i] = (double)1/(i+1);}
    j=n/numprocs;
    ln = j + n%numprocs;
    indice = ln;
    for (i=1; i<numprocs; i++) {
        MPI_Send( &x[indice], j, MPI_DOUBLE, i, i, MPI_COMM_WORLD);
        MPI_Send( &y[indice], j, MPI_DOUBLE, i, i, MPI_COMM_WORLD);
        indice = indice + j;
    }
}
```

Ejemplo memdinamica.c

(envío de los elementos de un array
Asignación dinámica de memoria)

```
else {  
    ln = n/numprocs;  
    x = malloc(ln * sizeof(double));  
    y = malloc(ln * sizeof(double));  
    MPI_Recv ( x, ln, MPI_DOUBLE, 0, myrank, MPI_COMM_WORLD, &status);  
    MPI_Recv ( &y[0], ln, MPI_DOUBLE, 0, myrank, MPI_COMM_WORLD, &status);  
}  
if (myrank == 0) {  
    printf("Soy %d. Despues del send/recv el valor de x es:",myrank);  
    for (i=0; i<n; i++) printf("%5.2f ",x[i]);  
    printf("\n");  
    printf("Soy %d. Despues del send/recv el valor de y es:",myrank);  
    for (i=0; i<n; i++) printf("%5.2f ",y[i]);  
    printf("\n"); }  
else  
{  
    printf("Soy %d. Despues del send/recv el valor de x es:",myrank);  
    for (i=0; i<ln; i++) printf("%5.2f ",x[i]);  
    printf("\n");  
    printf("Soy %d. Despues del send/recv el valor de y es:",myrank);  
    for (i=0; i<ln; i++) printf("%5.2f ",y[i]);  
    printf("\n");}  
    free(x); // liberar la memoria una vez deja de utilizarse  
    free(y);  
    MPI_Finalize();  
}
```

Notar que todos los procesos han de Reservar memoria

Ejemplo memdinamica.c

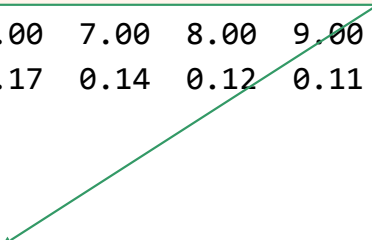
(envío de los elementos de un array Asignación dinámica de memoria)

```
mpicc -o memdinamica memdinamica.c
mpirun -np 2 memdinamica
```

Numero de elementos en los vectores:

10

Soy 0. Antes del send/recv el valor de x es:	1.00	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00	10.00
Soy 0. Antes del send/recv el valor de y es:	1.00	0.50	0.33	0.25	0.20	0.17	0.14	0.12	0.11	0.10
Soy 0. Despues del send/recv el valor de x es:	1.00	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00	10.00
Soy 0. Despues del send/recv el valor de y es:	1.00	0.50	0.33	0.25	0.20	0.17	0.14	0.12	0.11	0.10
Soy 1. Antes del send/recv el valor de x es:	0.00	0.00	0.00	0.00	0.00					
Soy 1. Antes del send/recv el valor de y es:	0.00	0.00	0.00	0.00	0.00					
Soy 1. Despues del send/recv el valor de x es:	6.00	7.00	8.00	9.00	10.00					
Soy 1. Despues del send/recv el valor de y es:	0.17	0.14	0.12	0.11	0.10					



Ejemplo array2d.c

(Asignación dinámica de memoria de forma continua array 2d)

```
#include <stdio.h>
#include <stdlib.h> // Para usar malloc
/* Programa ejemplo de como definir un array 2d en memoria dinamica
   almacenando los elementos de forma continua */

/* Allocate 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;
}

/* 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);
}
```

```
int main( int argc, char **argv )
{
    /* Declaraciones */
    double **a;
    int nrow_a, ncol_a;
    int i,j;

    printf("Numero de filas de A:\n");
    scanf("%d",&nrow_a);
    printf("Numero de columnas de A:\n");
    scanf("%d",&ncol_a);

    /* Allocate arrays */
    a = allocate_array(nrow_a, ncol_a);

    /* Initialize arrays */
    for(i=0; i<nrow_a; i++)
        for(j=0; j<ncol_a; j++)
            a[i][j]=(i+j);

    vermatriz(a, nrow_a, ncol_a, "a");

    /* Deallocate arrays */
    deallocate_array(a, nrow_a);
}
```

Comunicaciones colectivas:

Barreras

```
int MPI_Barrier( comm )
```

- MPI_Comm **comm**: comunicador.
-

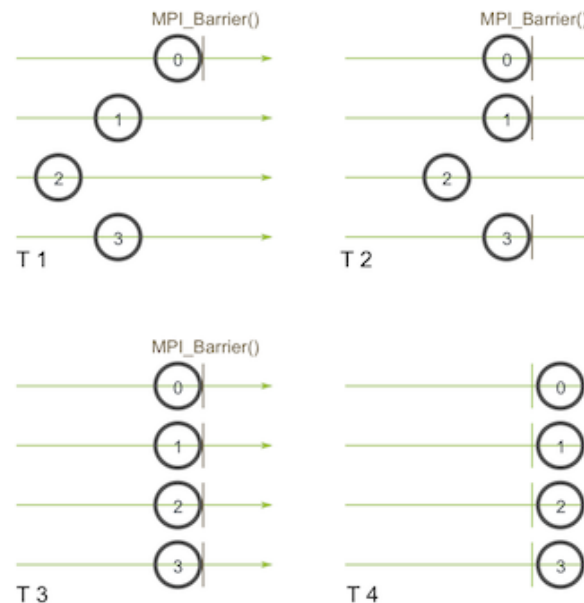
Los procesos incluidos en comm detienen su progreso al llamar a esta función y sólo continuarán hasta que todos los procesos hayan llamado a esta función.

Comunicaciones colectivas:

Barreras

MPI_Barrier(comm)

Imagina que el eje horizontal representa la ejecución del programa y los círculos representan diferentes procesos:

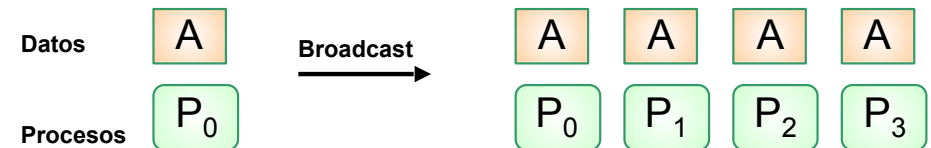


Comunicaciones colectivas:

Broadcast uno-a-todos

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

- **buf:** Variable que contiene la información a comunicar.
- int **count:** Cantidad de elementos contenidos en buf.
- MPI_Datatype **datatype:** Tipo de la variable buf.
- int **root:** Número lógico del proceso que hace el envío y desde el cual se espera recibir información.
- MPI_Comm **comm:** Comunicador.



Uno de los procesadores (el *root*) envía un mensaje a todos los procesadores incluidos en *comm*. *count* y *datatype* deben coincidir en todas las llamadas a esta función con objeto de que la cantidad de datos enviados y recibidos sea la misma.

Ejemplo 4

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv)
{
    int myrank, numprocs;
    float dato, res;
    dato = 7.0;
    res = 0.0;
    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);
    if (myrank == 1)    res = dato * dato;

    printf("Antes de recibir, el valor de res es %f\n",res);
    MPI_Bcast(&res, 1, MPI_FLOAT, 1, MPI_COMM_WORLD);
    printf("Despues de recibir, el valor de es %f\n",res);

    MPI_Finalize();
}
```

Variable que contiene la información a comunicar

Cantidad de elementos

Tipo variable

procesador que hace el envío y desde el cual se espera recibir información

Salida del Ejemplo 4

- Compilamos: `mpicc -o ejemplo4 ejemplo4.c`
- Después de la ejecución (`mpirun -np 3 ejemplo4`), la salida que produce el *ejemplo4* es:

```
Soy el proceso 0 de un total de 3
Antes de recibir, el valor de res es 0.000000
Despues de recibir, el valor de es 49.000000
```

```
Soy el proceso 2 de un total de 3
Antes de recibir, el valor de res es 0.000000
Despues de recibir, el valor de es 49.000000
```

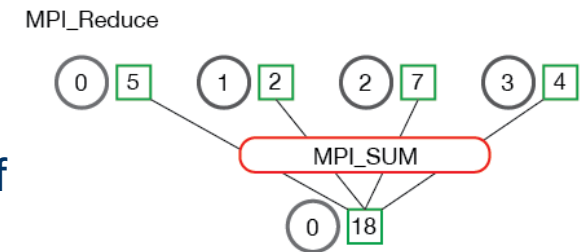
```
Soy el proceso 1 de un total de 3
Antes de recibir, el valor de res es 49.000000
Despues de recibir, el valor de es 49.000000
```

Comunicaciones colectivas:

Reducción todos-a-uno

```
int MPI_Reduce( *sendbuf, *recvbuf, count, datatype, op, dest, 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.
- int **dest**: Número lógico del proceso al cual se ha transferido información.
- MPI_Comm **comm**: Comunicador.



MPI_REDUCE Combina los elementos almacenados en sendbuf de cada proceso definido en el comunicador comm, utilizando la operación op, y regresa el **resultado en recvbuf** del proceso **dest**. 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 y dest.

La opción "in place" puede ser especificada con el valor **MPI_IN_PLACE** en el argumento **sendbuf** en el proceso root. En este caso, los datos de entrada en el proceso root, se toman del buffer de recepción recvbuf, donde serán reemplazados con los datos de salida.

Comunicaciones colectivas:

Reducción todos-a-uno

```
int MPI_Reduce( *sendbuf, *recvbuf, count, datatype, op, dest, 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.
- int **dest**: Número lógico del proceso al cual se ha transferido el resultado.
- MPI_Comm **comm**: Comunicador.

Operación	Significado
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Suma
MPI_PROD	Producto
...	...

MPI_REDUCE Combina los elementos almacenados en sendbuf de cada proceso definido en el comunicador comm, utilizando la operación op, y regresa el resultado en recvbuf del proceso dest. 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 y dest.

La opción "in place" puede ser especificada con el valor MPI_IN_PLACE en el argumento sendbuf en el proceso root. En este caso, los datos de entrada en el proceso root, se toman del buffer de recepción recvbuf, donde serán reemplazados con los datos de salida.

Ejemplo 5

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv)
{
    int myrank, numprocs, i;
    MPI_Status estado;
    double x[10], y[10];
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    for (i=0; i<10; 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<10; i++) printf("%4.1f ",x[i]);
    printf("\n");
    MPI_Reduce(&x,&y,10,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
    printf("Soy %d. Despues de recibir el valor de y es:",myrank);
    for (i=0; i<10; i++) printf("%4.1f ",y[i]);
    printf("\n");
    MPI_Finalize();
}
```

Salida del Ejemplo 5

```
MPI_Reduce(&x,&y,10,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
```

- Después de la ejecución (`mpirun -np 2 ejemplo5`), la salida que produce el *ejemplo5* es:

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 0. Despues de recibir el valor de y es:	0.0	3.0	6.0	9.0	12.0	15.0	18.0	21.0	24.0	27.0
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 1. Despues de recibir el valor de y es:	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Comunicaciones colectivas: MPI_IN_PLACE

Si quisiéramos usar la misma variable x para la entrada-salida:

```
MPI_Reduce(&x,&x,10,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
```

Provocaría un error:

Buffers must not be aliased

SOLUCIÓN: Usar MPI_IN_PLACE en el root

```
if (myrank != 0)
    MPI_Reduce(&x,&x,10,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
else
    MPI_Reduce(MPI_IN_PLACE,&x,10,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
```

Comunicaciones colectivas: MPI_IN_PLACE

Las operaciones colectivas pueden ejecutarse con la opción “in place”, cuando el buffer de salida es el mismo que el buffer de entrada. La forma de especificar esta situación es a través de un valor especial en uno de sus argumentos, **MPI_IN_PLACE**, en lugar del buffer de salida o del buffer de entrada, según el caso.

La operación “in place” reduce movimientos de memoria no necesarios tanto por la implementación de MPI como por el usuario.

Con la opción “in place”, el buffer de recepción se transforma en un buffer de envío y recepción, en muchas comunicaciones colectivas.

Ejemplo 5

```

#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv)
{
    int myrank, numprocs, i;
    MPI_Status estado;
    double x[10], y[10];
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    for (i=0; i<10; 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<10; i++) printf("%4.1f ",x[i]);
    printf("\n");
    MPI_Reduce(&x,&y,10,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
    printf("Soy %d. Despues de recibir el valor de y es:",myrank);
    for (i=0; i<10; i++) printf("%4.1f ",y[i]);
    printf("\n");
    MPI_Finalize();
}

```

Si quisiéramos usar la misma variable x para la entrada-salida:

```
MPI_Reduce(&x,&x,10,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
```

Provocaría un error:
Buffers must not be aliased

SOLUCIÓN: Usar MPI_IN_PLACE en el root

```

if (myrank != 0)
    MPI_Reduce(&x,&x,10,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
else
    MPI_Reduce(MPI_IN_PLACE,&x,10,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

```