

Deadlocks

- Un deadlock ocurre cuando un proceso queda esperando un mensaje que nunca recibirá.

Modos de Comunicación

- Forma en la que se realiza y completa un envío.
- MPI define 4 modos de envío:
 - ☐ básico(basic)
 - ☐ con buffer (buffered)
 - ☐ síncrono (synchronous)
 - ☐ listo (ready).



Envío con buffer

Cuando se hace un envío **con buffer** se guarda inmediatamente una copia del mensaje en un buffer del emisor. La operación se da por completa en cuanto se ha efectuado esta copia. (No implica que se recibió bien)

Si no hay espacio en el buffer, el envío fracasa.



Síncrono

Si se hace un envío **síncrono**, la operación se da por terminada sólo cuando el mensaje ha sido recibido en destino.

Bloquea el proceso hasta que el receive correspondiente fue realizado en el proceso destino.



Básico

El modo de envío **básico** no especifica la forma en la que se completa la operación: es algo dependiente de la implementación.

Normalmente equivale a un envío con buffer para mensajes cortos y a un envío síncrono para mensajes largos.

Listo

En cuanto al envío en modo **listo**, sólo se puede hacer si antes el otro extremo está preparado para una recepción inmediata. No hay copias adicionales del mensaje (como en el caso del modo con buffer), y tampoco podemos confiar en bloquearnos hasta que el receptor esté preparado.

Características

- Modo síncrono → seguro
- Modo listo → menor overhead en el sistema
- Modo Buffer → separa el emisor del receptor
- Modo estándar → no hay garantía que se inicie la recepción (comprometido)

Ejemplo

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
int main(int argc, char* argv[]) {
    int x, y, np, rank;
    int tag = 42;
    MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &np);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (np != 2) {
        if (rank == 0) {
            printf("Se deben usar dos procesos ");
            MPI_Finalize();
            exit(0);
        }
    }
```



```
x = 12345; y = rank;
if (rank == 0) {
    printf("Proceso %d envía a proceso 1\n", rank);
    MPI_Send(&x, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
    printf("Proceso %d recibe de proceso 1\n", rank);
    MPI_Recv (&y, 1, MPI_INT, 1, tag, MPI_COMM_WORLD,
&status);
    printf ("Process %d received value %d\n", rank, y); }
else {
    MPI_Recv (&y, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
    MPI_Send (&y, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
}
MPI_Finalize();
exit(0);
}
```

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

int main(int argc, char* argv[]) {
    const int K = 1024;
    const int msgsize = 256*K;
    int *x, *y, np, rank;
    int tag = 42;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (np != 2) {
        if (rank == 0) {
            printf("Se deben usar dos procesos ");
        }
        MPI_Finalize();
        exit(0); }
}
```

Ejemplo

```
X = (int *) malloc(msgsize*sizeof(int));
Y = (int *) malloc(msgsize*sizeof(int));
for (i=0; i<msgsize; i++) {  x[i] = 12345; y[i] = rank; }

if (rank == 0) {
printf("Tamaño del mensaje es %d bytes\n", msgsize*sizeof(int));
    printf("Proceso %d envía a proceso 1\n", rank);
    MPI_Send(x, msgsize, MPI_INT, 1, tag, MPI_COMM_WORLD);
    printf("Proceso %d recibe de proceso 1\n", rank);
    MPI_Recv (y, msgsize, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
    printf ("y [0]tiene el valor de %d ", y[0]); }
else {
    MPI_Recv (y, msgsize, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
    MPI_Send (y, msgsize, MPI_INT, 0, tag, MPI_COMM_WORLD);
}
    MPI_Finalize();
exit(0);
}
```


Ejemplo

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
int main(int argc, char* argv[]) {
    int x, y, np, rank;
    int tag = 42;
    MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &np);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (np != 2) {
        if (rank == 0) {
            printf("Se deben usar dos procesos ");
            MPI_Finalize();
            exit(0);
        }
    }
```


```
x = 12345; y = rank;
if (rank == 0) {
    printf("Proceso %d envía a proceso 1\n", rank);
    MPI_Ssend(&x, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
    printf("Proceso %d recibe de proceso 1\n", rank);
    MPI_Recv (&y, 1, MPI_INT, 1, tag, MPI_COMM_WORLD,
&status);
    printf ("Process %d received value %d\n", me, y); }
else {
    MPI_Recv (&y, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
    MPI_Ssend (&y, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
}
MPI_Finalize();
exit(0);
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#define BUFFSIZE 10240
int main(int argc, char* argv[]) {
    int x, y, np, me, buff[BUFFSIZE];
    int size = BUFFSIZE;    int tag = 42;    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (np != 2) {
        if (rank == 0) {
            printf("Se deben usar dos procesos ");
            MPI_Finalize();
            exit(0); }
    MPI_Buffer_attach(buff, size); x = 12345; y = rank;
```

Ejemplo



```
if (rank == 0) {  
    printf("Proceso %d envia %d a proceso 1\n", rank, x);  
    MPI_Bsend(&x, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);  
    printf("Proceso %d recibe de proceso 1\n", rank);  
    MPI_Recv (&y, 1, MPI_INT, 1, tag, MPI_COMM_WORLD,  
    &status);  
    printf ("Proceso %d recibe el valor de  %d\n",rank, y); }  
else {  
    MPI_Recv (&y, 1, MPI_INT, 0, tag, MPI_COMM_WORLD,  
    &status);  
    MPI_Bsend (&y, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);}  
    MPI_Buffer_detach(&buff, &size);  
    MPI_Finalize();  
    exit(0); }
```

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

Input Parameters

- **Buf** initial address of send buffer (choice)**count** number of elements in send buffer (nonnegative integer)
- **Datatype** datatype of each send buffer element (handle)
- **Dest** rank of destination (integer)**tag** message tag (integer)
- **Comm** communicator (handle)



■ `int MPI_Buffer_attach(void *buffer, int size)`

Buffer initial buffer address (choice)

size buffer size, in bytes (integer)

No bloqueante send and receive

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
main(int argc, char* argv[])
{ int x, y, np, rank;
int tag = 42;
MPI_Status status;
MPI_Request send_req, recv_req;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &np); /
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (np != 2) { if (rank == 0) { printf("Solo 2 procesos"); }
MPI_Finalize(); exit(0); }
x = 12345; y = rank;
```

```
if (me == 0) {
printf("Proceso %d envía a proceso1\n", rank);
MPI_Isend(&x, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &send_req);
/*se puede realizar algo aquí*/
MPI_Wait(&send_req, &status);
printf("Proceso %d recibe deprocess 1\n", rank);
MPI_Irecv (&y, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &recv_req);
/*Se puede realizar computo mientras espera*/
printf ("Proceso %d tien el valor  %d\n", rank, y);
MPI_Wait(&recv_req, &status);
printf ("Proceso %d recibe el valor de %d\n", me, y); }
else {
MPI_Irecv (&y, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &recv_req);
/*Se puede realizar computo mientras espera*/
MPI_Wait(&recv_req, &status);
MPI_Isend (&y, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &send_req)
/*Se puede realizar computo mientras espera*/
MPI_Wait(&send_req, &status); }
MPI_Finalize(); exit(0); }
```

MPI_Wait

- Bloquea al proceso hasta que termine la operación de envío o de recibo especificada.
- **int MPI_Wait (MPI_Request *request, MPI_Status *status)**
- request :Dato de tipo MPI_Request, etiqueta que identifica una operación no bloqueante

MPI_Isend

- Función de punto a punto no bloqueante.
- **int MPI_Isend** (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

MPI_Irecv

Función para recepción punto a punto no bloqueante

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

```
#include <mpi.h>
#define MAXPROC 8
int main(int argc, char* argv[]) {
    int i, x, np, me;
    int tag = 42;
    MPI_Status status[MAXPROC];
    MPI_Request send_req[MAXPROC], recv_req[MAXPROC];
    int y[MAXPROC];
    MPI_Init(&argc, &argv)
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (np<2 || np>MAXPROC) {
        if (me == 0) {
            printf("Se requieren minimo 2 maximo %d procesos\n", MAXPROC);
        }
    }
    MPI_Finalize();    exit(0); }    x = me;
```

```

if (me == 0)
    printf("Proceso %d envía a todos los demás \n", rank);
for (i=1; i<np; i++) {
    printf("Proceso %d envía a %d\n", rank, i);
    MPI_Isend(&x, 1, MPI_INT, i, tag, MPI_COMM_WORLD, &send_req[i]); }

/* Mientras los mensajes son enviados se puede realizar cómputo*/
/* Esperar hasta que todos los mensajes han sido enviados*/
    MPI_Waitall(np-1, &send_req[1], &status[1]);
    printf("Proceso %d recibe de todos\n", rank);
for (i=1; i<np; i++) {
    MPI_Irecv (&y[i], 1, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD,
&recv_req[i]); }

/* Mientras los mensajes son enviados se puede realizar cómputo*/
/* Esperar hasta que todos los mensajes han sido recibidos*/
    MPI_Waitall(np-1, &recv_req[1], &status[1]);
for (i=1; i<np; i++) {
    printf("Proceso %d recibe el mensaje %d\n", rank, y[i]);
} printf("Proceso %d listo\n", rank);
}

```




```
else {  
    MPI_Irecv (&y, 1, MPI_INT, 0, tag, MPI_COMM_WORLD,  
    &recv_req[0]);  
    MPI_Wait(&recv_req[0], &status[0]); MPI_Isend (&x, 1,  
    MPI_INT, 0, tag, MPI_COMM_WORLD, &send_req[0]);  
    /*Se puede realizar algun calculo mientras espera*/  
    MPI_Wait(&send_req[0], &status[0]); }  
    MPI_Finalize();  
    exit(0);  
}
```

MPI_Waitall

Bloquea al proceso hasta que terminen todas las operaciones de envío o recepción.

```
int MPI_Waitall(int count, MPI_Request  
*array_of_requests, MPI_Status*array_of_statuses)
```



count :indica el número de operaciones que se van a especificar.

array_of_requests :Vector de datos de tipo MPI_Request, en el se guarda una etiqueta que identifica una operación no bloqueante. El puntero especificado debe apuntar a la primera posición del vector de request que se espera comprobar.

array_of_statuses :Vector de objetos de tipo MPI_Status, cada elemento contiene datos relevantes sobre el mensaje al que se refiere



Comunicaciones Colectivas

- Comunicaciones realizadas entre un grupo de procesos los cuales están especificados en un comunicador
- Todos los procesos en el comunicador deben llamar la operación colectiva

Algunas funciones para comunicaciones colectivas

Process 0	Process 1 ^s	Process 2	Process 3	Function Used	Process 0	Process 1 ^s	Process 2	Process 3
a	b	c	d	<u>MPI_Gather</u>		a,b,c,d		
a	b	c	d	<u>MPI_Allgather</u>	a,b,c,d	a,b,c,d	a,b,c,d	a,b,c,d
	a,b,c,d			<u>MPI_Scatter</u>	a	b	c	d
a,b,c,d	e,f,g,h	ij,k,l	m,n,o,p	<u>MPI_Alltoall</u>	a,e, i,m	b,f, j,n	c,g, k,o	d,h, l,p
	b			<u>MPI_Bcast</u>	b	b	b	b
Send Buffer	Send Buffer	Send Buffer	Send Buffer		Receive Buffer	Receive Buffer	Receive Buffer	Receive Buffer

MPI_Bcast

- Envía datos de un proceso a todos los demás.

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

Buffer (Input/Output): Dirección de los datos

Count: Número de elementos en buffer

datatype: Tipo de datos

Root: Rank del proceso que contiene los datos que serán replicados

Comm: Comunicador



MPI_Gather

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

MPI_Scatter

- realiza la operación simétrica a MPI_Gather() .

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