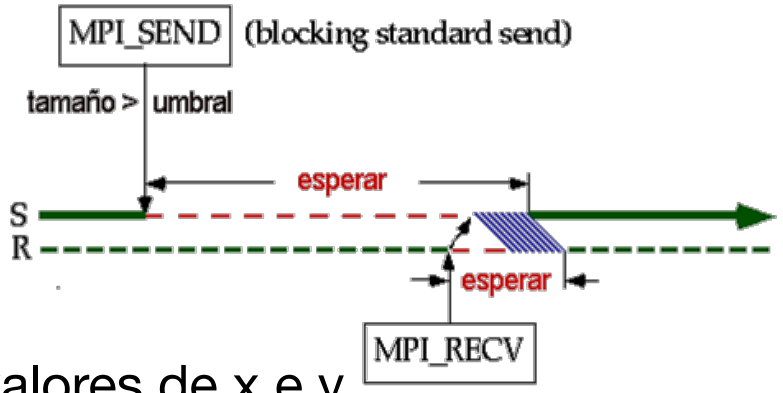
Problemas de interbloqueo

- El uso de blocking standard send (MPI_Send (...)) puede en ocasiones generar un interbloqueo en los procesos **cuando se sobrepasa** un determinado umbral **en el tamaño de los mensajes enviados.**

Proceso 0	Proceso 1
send(x, 1)	send(y, 0)
receive(y, 1)	receive(x, 0)

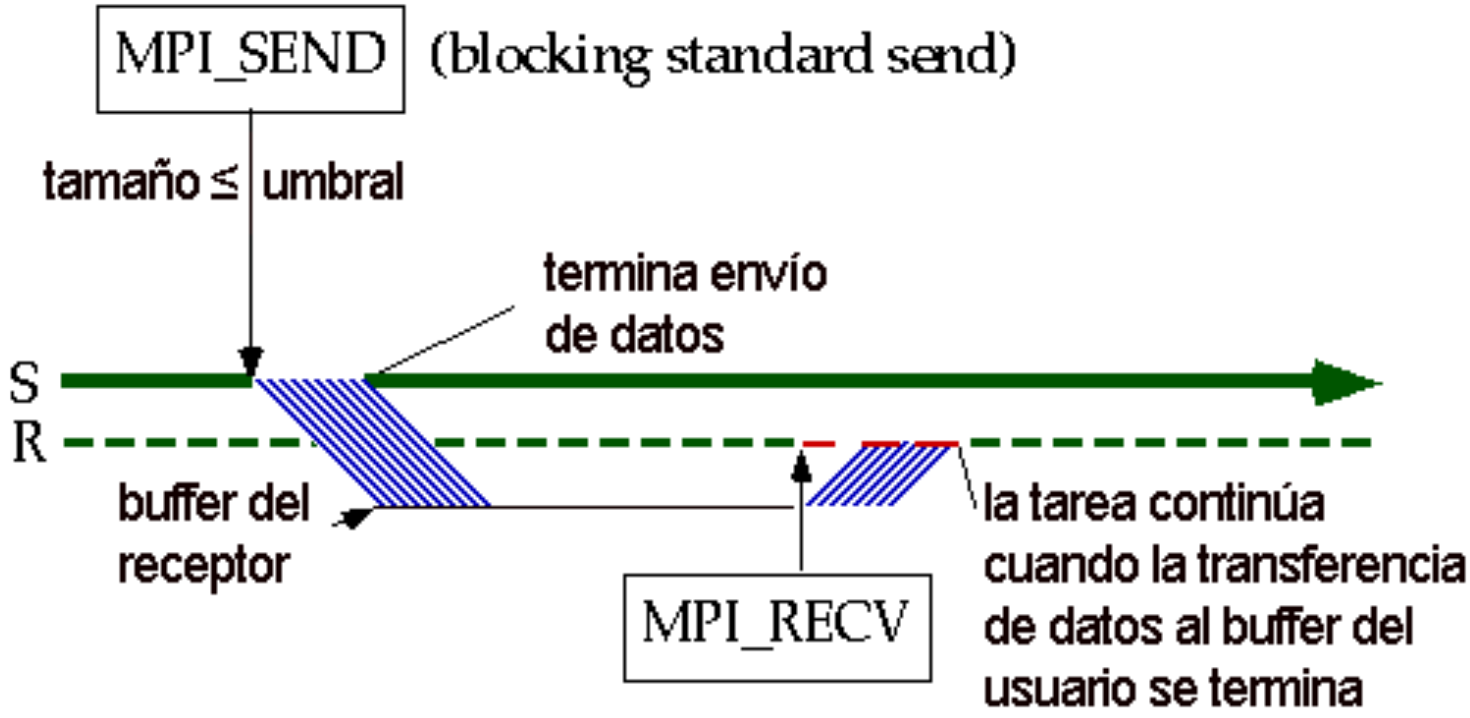


Ambos procesos pretenden intercambiar sus valores de x e y.

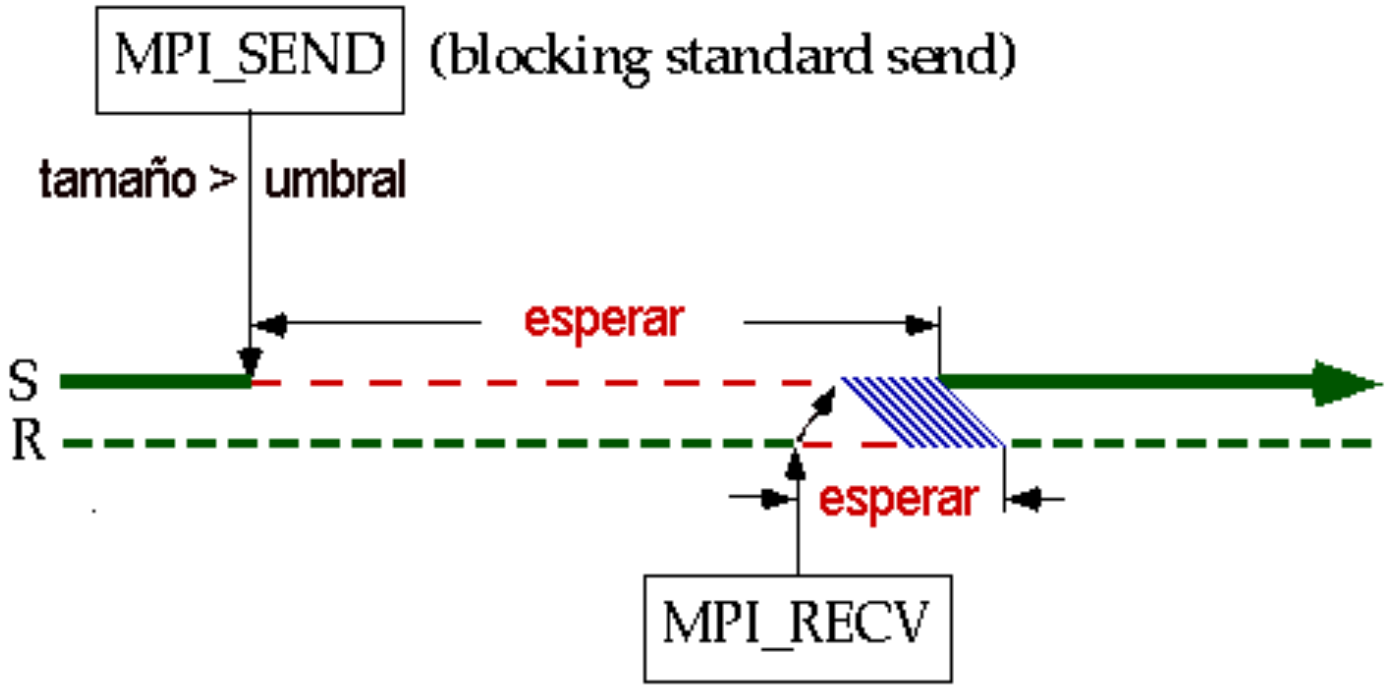
….pero

- La operación send (síncrona) de cada proceso está esperando el correspondiente receive del segundo proceso implicado en la operación.
- Asimismo, la operación receive de ambos procesos no se ejecuta nunca ya que la operación de envío no finaliza.
- Como consecuencia, ninguno de los procesos puede proceder con su ejecución, es decir, se encuentran interbloqueados.

Problemas de interbloqueo



Problemas de interbloqueo



Ejemplo 10

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


int main(int argc, char **argv)
{
   int msglen;
   double *mensaje1, *mensaje2;
   int rank,dest,ori,numprocs;
   int send_eti, recv_eti, i;
   MPI_Status status;


   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   if (numprocs != 2) {
         if (rank == 0) printf("Ejecuta con solo 2 procesos!!\n");
         MPI_Finalize ();
         return 0;
   }
   printf("Soy el proceso %d de un total de %d\n",rank,numprocs);
   if (rank == 0) {
         printf("Longitud de los vectores a enviar: \n");
         scanf("%i",&msglen);
   }
  MPI_Bcast(&msglen// 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
         mensaje1 = (double *)malloc(msglen * sizeof(double));
         mensaje2 = (double *)malloc(msglen * sizeof(double));
    for (i=0; i<msglen; i++)
         {
             mensaje1[i]= 100;
             mensaje2[i]=-100;
         }
 if ( rank == 0 ) {
         dest = 1;
         ori = 1;
         send_eti = 10;
         recv_eti = 20;
    } else {
         dest = 0;
         ori = 0;
         send_eti = 20;
         recv_eti = 10;
    }
    printf(" Tarea %d esta enviando el mensaje\n", rank);
MPI_Send(mensaje1,msglen,MPI_DOUBLE,dest,send_eti, MPI_COMM_WORLD);
//   for (i=0; i<5; i++) printf("%6.1f", mensaje2[i]); printf("\n");
MPI_Recv(mensaje2,msglen,MPI_DOUBLE,ori,recv_eti,MPI_COMM_WORLD,&status);
    for (i=0; i<5; i++) printf("%6.1f", mensaje2[i]);  printf("\n");
    printf(" Tarea %d ha recibido el mensaje\n", rank);
    free(mensaje1);
    free(mensaje2);
    MPI_Finalize();
}
```

Este programa entra en interbloqueo cuando se ejecuta con un valor elevado de *msglen*

Problemas de interbloqueo

MPI proporciona varias alternativas con las que resolver estos problemas de interbloqueo:

- Utilizar la función **MPI_Bsend** que permite **gestionar** su propio **buffer** para la comunicación y garantizar que la función de envío hace la copia del mensaje de forma correcta. Debe utilizarse con las funciones de gestión del buffer.
- Usar la función **MPI_Sendrecv** que combina, **en una sola llamada, el envío y la recepción**. Es una función bloqueante que permite prevenir interbloqueos como consecuencia de situaciones de espera circular.
- Hacer uso de las operaciones punto-a-punto **no bloqueantes MPI_Isend** y **MPI_Irecv** en combinación con las funciones **MPI_Test** y **MPI_Wait** que permiten validar o esperar el resultado de una operación no bloqueante.

Problemas de interbloqueo:
Nonblocking Standard Send

int MPI_Isend(*buf, count, datatype, dest, tag, comm, *request)

- **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 **dest:** Número lógico del proceso al cual se ha transferido información.
- int **tag:** Identifica el envío. Generalmente es cero y sólo cambia cuando se ha de comunicar más de un envío.
- MPI_Comm **comm:** Comunicador.
- MPI_Request **request:** En combinación con las funciones MPI_TEST y MPI_WAIT proporciona información sobre el estado de la función MPI_ISEND.

Envía un mensaje a otro proceso. El proceso origen continúa su trabajo sin esperar a que el proceso destinatario haya recibido el mensaje.

Problemas de interbloqueo:

Nonblocking Standard Receive

int MPI_Irecv(*buf, count, datatype, source, tag, comm, *request)

- **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 **source:** Número lógico del proceso desde el cual se espera recibir información.
- int **tag:** Identifica el envío. Generalmente es cero y sólo cambia cuando se ha de comunicar más de un envío.
- MPI_Comm **comm:** Comunicador.
- MPI_Request **request:** En combinación con las funciones MPI_TEST y MPI_WAIT proporciona información sobre el estado de la función MPI_IRECV.

Se dispone a recibir un mensaje de parte de otro proceso y continua su trabajo sin esperar a recibirlo por completo.

Problemas de interbloqueo:

Nonblocking Standard Send/Receive

int MPI_Wait(*request, *status)

- MPI_Request **request:** En combinación con las funciones MPI_TEST y MPI_WAIT proporciona información sobre el estado de la función MPI_ISEND y MPI_IRECV.
- MPI_Status **status:** Auxiliar necesario para conocer el estado de ejecución de una función MPI.

Una llamada a la función MPI_WAIT regresa cuando la operación no bloqueada identificada por **request** ha concluido.

Ejemplo 11

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

int main(int argc, char **argv)
{
   int msglen;
   double *mensaje1, *mensaje2;
   int rank,dest,ori,numprocs;
   int send_eti, recv_eti, i;
   MPI_Status status;
   MPI_Request request1, request2;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   if (numprocs != 2) {
         if (rank == 0) printf("Ejecuta con solo 2 procesos!!\n");
         MPI_Finalize ();
         return 0;
   }
   printf("Soy el proceso %d de un total de %d\n",rank,numprocs);
   if (rank == 0) {
         printf("Longitud de los vectores a enviar: \n");
         scanf("%i",&msglen);
   }
   MPI_Bcast(&msglen  // 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

   mensaje1 = (double *)malloc(msglen * sizeof(double));
   mensaje2 = (double *)malloc(msglen * sizeof(double));
   for (i=0; i<msglen; i++)
        {
            mensaje1[i]= 100;
            mensaje2[i]=-100;
     }
  if ( rank == 0 ) {
         dest = 1;
         ori = 1;
         send_eti = 10;
         recv_eti = 20;
   } else {
         dest = 0;
         ori = 0;
         send_eti = 20;
         recv_eti = 10;
   }
MPI_Isend(mensaje1,msglen,MPI_DOUBLE,dest,send_eti,MPI_COMM_WORLD,
      &request1);
MPI_Irecv(mensaje2,msglen,MPI_DOUBLE,ori,recv_eti,MPI_COMM_WORLD,
      &request2);
// calculos
   printf("Antes del wait Tarea %d: ", rank);
   for (i=0; i<3; i++) printf("m2[%d]=%6.1f ", i, mensaje2[i]);
   printf("\n");
   MPI_Wait ( &request1, &status );
   MPI_Wait ( &request2, &status );
   printf("Tarea %d: ", rank);
//       for (i=0; i<3; i++) printf("m2[%d]=%6.1f ", i, mensaje2[i]);
      printf("\n");
   free(mensaje1);
   free(mensaje2);
   MPI_Finalize();
}
```

Ejemplo 11

Salida del ejemplo11.c

Distintas ejecuciones:

```
Longitud de los vectores a enviar: 100000
Tarea 0: m2[0]= 100.0 m2[1]= 100.0 m2[2]= 100.0
Tarea 1: m2[0]= 100.0 m2[1]= 100.0 m2[2]= 100.0
```

Correcto!!

Si en el ejemplo11.c se eliminan los MPI_Wait
el comportamiento es incorrecto.

Distintas ejecuciones:

```
     Longitud de los vectores a enviar: 999
Tarea 0: m2[0]=-100.0 m2[1]=-100.0 m2[2]=-100.0
Tarea 1: m2[0]=-100.0 m2[1]=-100.0 m2[2]=-100.0

   Longitud de los vectores a enviar: 100000
Tarea 0: m2[0]=-100.0 m2[1]=-100.0 m2[2]=-100.0
Tarea 1: m2[0]=-100.0 m2[1]=-100.0 m2[2]=-100.0

===================================================
BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
                EXIT CODE: 11
        CLEANING UP REMAINING PROCESSES
      YOU CAN IGNORE THE BELOW CLEANUP MESSAGES
===================================================
```

Incorrecto !!!!

Ejemplo 12

Blocking calls
pueden estar en correspondencia con
non-blocking calls

Ejemplo 12

```
int main(int argc, char **argv)   {
double *mensaje1, *mensaje2;
int rank,dest,ori,numprocs, msglen;
int send_eti, recv_eti, i;
 MPI_Status status;
 MPI_Request request;


 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
 if (numprocs != 2) {
 if (rank == 0) printf("Ejecuta con solo 2 procesos!!\n");
        MPI_Finalize ();
        return 0;  }
  if (rank == 0) {
        printf("Longitud de los vectores a enviar: \n");
        scanf("%i",&msglen);
        printf("%d\n",msglen); }
  MPI_Bcast(&msglen,1,MPI_INT,0,
                     MPI_COMM_WORLD);
  mensaje1 = (double *)malloc(msglen * sizeof(double));
  mensaje2 = (double *)malloc(msglen * sizeof(double));
  for (i=0; i<msglen; i++)   {
           mensaje1[i]= 100;
           mensaje2[i]=-100;      }
```

```
 if ( rank == 0 ) {
         dest = 1;
         ori = 1;
         send_eti = 10;
         recv_eti = 20;
   } else {
         dest = 0;
         ori = 0;
         send_eti = 20;
         recv_eti = 10;
   }
  MPI_Isend(mensaje1,msglen,MPI_DOUBLE,dest,send_eti,
                     MPI_COMM_WORLD,&request);
  MPI_Recv(mensaje2,msglen,MPI_DOUBLE,ori,recv_eti,
                     MPI_COMM_WORLD, &status);
  MPI_Wait( &request, &status );

   printf("Tarea %d: ", rank);
   for(i=0; i<3; i++)
   printf("m2[%d]=%6.1f ", i, mensaje2[i]);
   printf("\n");
   free(mensaje1);
   free(mensaje2);
   MPI_Finalize();
}
```

En este ejemplo el resultado sería el mismo poniendo MPI_Wait que no poniéndolo.

Problemas de interbloqueo:

Blocking Buffered Send

int MPI_Bsend(*buf, count, datatype, dest, tag, 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 **dest:** Número lógico del proceso al cual se ha transferido información.
- int **tag:** Identifica el envío. Generalmente es cero y sólo cambia cuando se ha de comunicar más de un envío.
- MPI_Comm **comm:** Comunicador.

Realiza un envío en modo blocking buffered. No depende de una operación de recepción para finalizar. Si no existe recepción, el mensaje se dirige al buffer para completar la llamada. **MPI_Bsend** no garantiza que el mensaje se ha enviado, sino que queda en el buffer hasta que su correspondiente recepción. Necesita del uso de la función **MPI_Buffer_attach**.

Problemas de interbloqueo:

Blocking Buffered Send (espacio en buffer)

int MPI_Buffer_attach(*buffer, size)

- **buffer:** Debe apuntar a un array existente que no debe ser usado por el programador (input).
- int **size:** Tamaño en bytes del buffer (input).

Esta función provee a MPI de un buffer en el espacio de memoria del usuario que se utiliza para el envío de mensajes en modo buffered. Sólo un buffer puede ser declarado para una tarea en cada momento.

El buffer puede ser liberado con
int MPI_Buffer_detach(void *buffer, int *size)

Problemas de interbloqueo:

Blocking Buffered Send (espacio en buffer)

int MPI_Pack_size(incount, datatype, comm, *size)

- int **incount:** Número de elementos del tipo datatype (input).
- MPI_Datatype **datatype:** Tipo de la variable (input).
- MPI_Comm **comm:** Comunicador (input).
- int **size:** Cota superior del tamaño de mensaje (output, en bytes).

Devuelve una cota superior de la cantidad de espacio (en bytes) requerido para un mensaje de tamaño *incount* y tipo *datatype.*

Adicionalmente, un send utiliza algo más de espacio determinado por MPI_BSEND_OVERHEAD, con lo que la **cantidad de espacio requerido para un mensaje** de tamaño *incount* y tipo *datatype* es:

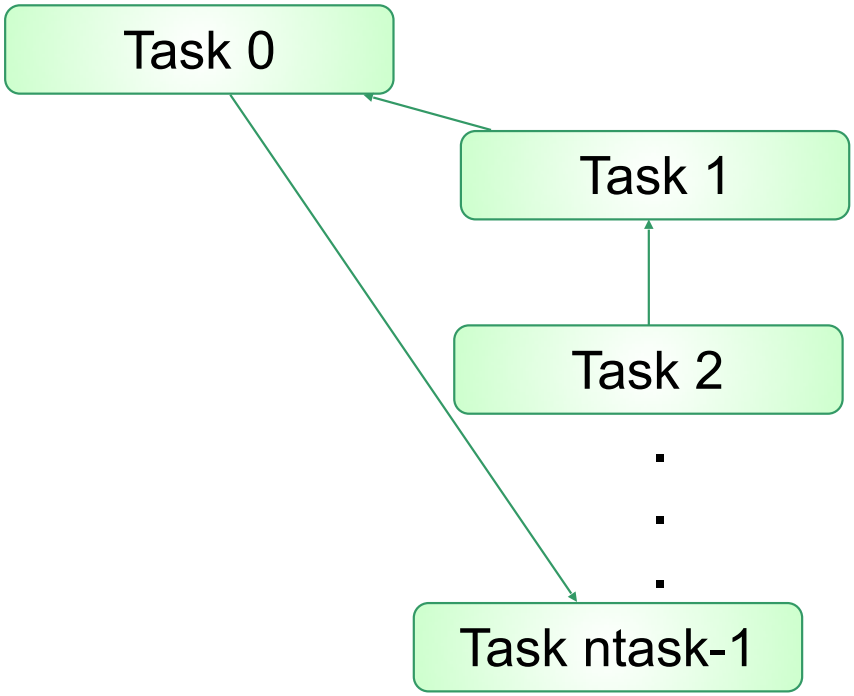
size + MPI_BSEND_OVERHEAD

Ejemplo 14

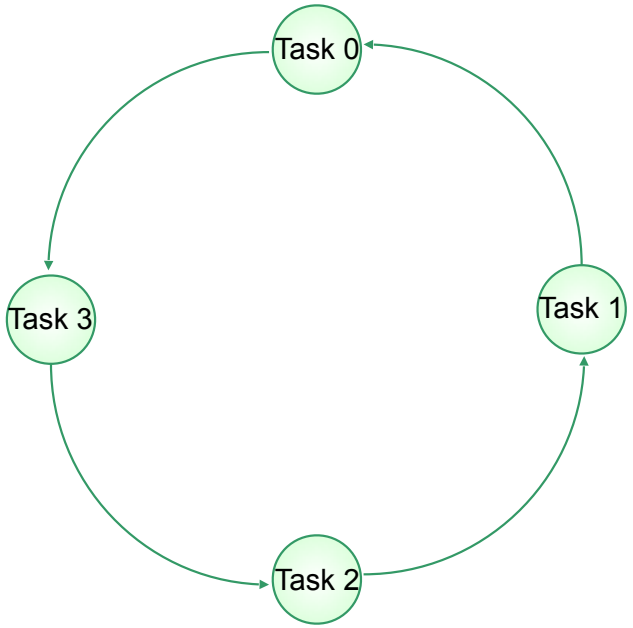
- Se generan un total de *ntask* tareas.
- Cada una define un vector de tamaño *size.*
- Cada tarea *i* envía el vector a la tarea *i-1*. La tarea *0* envía a la tarea *ntask-1*. Este proceso se realiza *ntask* veces, con lo que al final cada proceso debe almacenar el mismo vector que el inicial.
- Uso de Nonblocking Standard Send:
  - Al finalizar, los procesos no almacenan el mismo vector inicial.
  - Comportamiento no determinista debido al exceso de mensajes entre procesos.
- Uso de Blocking Buffered Send:
  - Al finalizar, los procesos SÍ almacenan el mismo vector.
  - El uso del buffer garantiza que el dato enviado será recibido aunque se modifique en el proceso origen.

Ejemplo 14

Comunicación en anillo
rtask tareas



ntask = 4



Ejemplo 14

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

#define maxntask  8
#define maxsize 1000000
int main(int argc, char **argv)
{
   int ntask, myrank;
   int i,j,k;
   double *a, *c, *buffer;
   int size, sizeBuffer;
   int arriba, abajo, numerror;
   int irequest,control;
   MPI_Status status;
   MPI_Request request;

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


   if (ntask > maxntask) {
         MPI_Finalize();
         return 0;
   }
   if ( myrank == 0 ) {
         printf("Orden de cada vector: \n");
         scanf("%d", &size);
         printf("Variable de control: 0(problemas en ISEND),1 (Solucion con uso de BSEND) \n");
         scanf("%d", &control);
   }
```

Ejemplo 14

```
MPI_Bcast(&size,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&control,1,MPI_INT,0,MPI_COMM_WORLD);


   if (((control != 0) && (control != 1)) || (size > maxsize)){
         if (myrank == 0) printf("Parametros no validos....saliendo!!!\n");
         MPI_Finalize();
         return 0;
   }


   abajo = (myrank + 1) % ntask;
   arriba = (myrank + ntask - 1) % ntask;

   a = (double *)malloc(size * sizeof(double));
   c = (double *)malloc(size * sizeof(double));


   for (i=0; i<size; i++) {
       a[i] = myrank;
       c[i] = myrank;
   }
```

```
 if (control == 1) {
      MPI_Pack_size(size,MPI_DOUBLE,MPI_COMM_WORLD, &sizeBuffer); // Obtenemos una cota
                    superior para el mensaje: vector de tamaño size tipo doble
      if (myrank == 0) printf("Espacio requerido en buffer: %d\n",sizeBuffer);

     sizeBuffer = ntask*(sizeBuffer + MPI_BSEND_OVERHEAD);
                    // Cada proceso crea un buffer de tamaño ntask*(sizeBuffer +
                    MPI_BSEND_OVERHEAD) porque va a realizar ntask
                    envios cada uno de los cuales de tamaño sizeBuffer, de esta forma nos
                    aseguramos que hay espacio suficiente en el buffer para todos los envíos.
      buffer = (double *)malloc(sizeBuffer);
      MPI_Buffer_attach(buffer, sizeBuffer); // provee de un buffer de nombre
                                             buffer y tamaño sizeBuffer en bytes
   }
   for (i=0; i<ntask; i++) {
       if (control == 0) {
          MPI_Isend(a,size,MPI_DOUBLE,arriba,i,MPI_COMM_WORLD,&request);
          MPI_Recv(a,size,MPI_DOUBLE,abajo,i,MPI_COMM_WORLD,&status);
          MPI_Wait(&request,&status);
       }
       else if (control == 1) {
          MPI_Bsend(a,size,MPI_DOUBLE,arriba,i,MPI_COMM_WORLD);
          MPI_Recv(a,size,MPI_DOUBLE,abajo,i,MPI_COMM_WORLD,&status);
       }
   }
   if (control == 1) MPI_Buffer_detach(buffer, &sizeBuffer);
          numerror=0;
   for (i=0; i< size; i++) {
      if (a[i]-c[i] != 0) numerror=numerror+1;
   }
   printf("Numero de identificacion de proceso %d, Numero de errores: %d\n", myrank, numerror);
   free(a);
   free(c);
   if (control == 1) free(buffer);
   MPI_Finalize();
}
```

Mediciones de tiempo

- Los sistemas operativos generalmente proporcionan comandos de línea que permiten cronometrar la ejecución de un código de principio a fin.
- Aún cuando esta opción es valiosa para el desarrollador, en ocasiones es necesario cronometrar la ejecución de segmentos de código para estimar su eficiencia.

double MPI_Wtime()

double MPI_Wtick()

Mediciones de tiempo

double MPI_Wtime()

Devuelve un número en coma flotante, de segundos que representan un cierto lapso de tiempo con respecto a un tiempo pasado.

---

double MPI_Wtick()

Regresa la resolución de reloj, o cantidad de segundos entre cuentas sucesivas de reloj, asociada a MPI_Wtime. El valor se reporta en segundos y como un número de doble precisión. De esta manera, una resolución de 0.001 indica que el sistema incrementa el contador del reloj cada milisegundo.

Mediciones de tiempo:

uso

```
…
double start_time, end_time, clock_res;
…
start_time = MPI_Wtime();
…
… cálculos
…
end_time = MPI_Wtime();
clock_res = MPI_Wtick();

printf("El sistema tiene una resolucion de reloj = %f\n',clock_res);
printf("Tiempo de ejecucion = %f seg\n", end_time - start_time);
```