Computación de alta performance

Sergio Nesmachnow (sergion@fing.edu.uy)
Universidad de la República, Uruguay

Message Passing Interface (MPI)

Sergio Nesmachnow (sergion@fing.edu.uy)
Universidad de la República, Uruguay

Contenido

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Desarrollo de aplicaciones paralelas/distribuidas en MPI

1. Conceptos y primitivas básicas
2. Modos de comunicación en MPI
3. La API de MPI
    - 3.1. Comunicaciones bloqueantes
    - 3.2. Comunicaciones no bloqueantes
    - 3.3. Operaciones colectivas
    - 3.4. Grupos de procesos
    - 3.5. Tipos de datos
    - 3.6. Topologías de procesos
4. Ejemplos y ejercicios

Contenido

Desarrollo de aplicaciones paralelas/distribuidas en MPI

Slide importante

Slide opcional

Slide de autoestudio

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

MPI: conceptos

El paradigma de pasaje de mensajes

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Modelo de programación para aplicaciones en sistemas de memoria distribuida
- No existe un recurso compartido común
- Los procesos concurrentes se comunican y sincronizan mediante el envío explícito de mensajes
- Utilizados en múltiples sistemas de computación paralela y distribuida:
  - Sockets y comunicación UDP/TCP
  - Colas de mensajes
  - Servicios web

- Habitualmente se utilizan buffers (espacios reservados de memoria) para almacenar información y colas para almacenar mensajes

Pasaje de mensajes

- Procesos en ejecución se comunican mediante mensajes explícitos

- Envío de mensajes (de datos, de sincronización)

```
send(origen,destino,tipo,msg,flags)
```

  - msg contiene los datos o información de sincronización enviados

- Recepción de mensajes

```
receive(origen,destino,tipo,msg,flags)
```

- El contenido de msg se almacena en memoria

- Utiliza buffers (espacios reservados de memoria) para almacenar información y colas para almacenar mensajes

Bibliotecas de pasaje de mensajes

- Simplifican la operativa de comunicación mediante pasaje de mensajes (brindan sincronismo, garantía de recepción, etc.)
- PVM: Parallel Virtual Machine
  - Estándar de facto para bibliotecas de pasajes de mensajes
  - Orientado al procesamiento paralelo en redes de computadores
- MPI: Message Passing Interface
  - Estándar definido por investigadores, empresas e industrias.
  - Especificación de protocolo, tiene varias implementaciones:
    - MPICH, LAM/MPI, OpenMPI, MS MPI, etc.
- Ambas bibliotecas siguen el paradigma de pasaje de mensajes
- Son bibliotecas, no lenguajes y son de código libre
- Tienen interfaces para lenguajes de alto nivel (C/C++)
- Cuentan con versiones estables y disponibles públicamente

MPI (Message Passing Interface)

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Biblioteca estándar para programación paralela bajo el paradigma de comunicación de procesos mediante pasaje de mensajes
  - Biblioteca, no lenguaje, que proporciona funciones
  - Creado por la industria, desarrolladores de software y aplicaciones y científicos (IBM, Intel, PVM, nCUBE)
- Objetivo: desarrollar un estándar portable y eficiente, para programación paralela
- Plataforma objetivo: memoria distribuida
- Paralelismo explícito (definido y controlado en su totalidad por el programador). Modelo de programas: SPMD
- Número de tareas fijado en tiempo pre-ejecución, no incluye una primitiva para lanzar procesos en tiempo de ejecución (spawn)
- 125 funciones (significativamente más que PVM)

MPI: objetivos específicos

1. Portabilidad
   - Definir entorno de programación único que sea genérico y no dependa de la arquitecturas que la soportan
2. Eficiencia
   - Sacar ventajas del hardware especializado que se tenga disponible en la infraestructura
3. Funcionalidad
   - Estructuras de datos avanzadas, manejo de grupos, comunicación optimizada

¿Qué incluye el estándar MPI?

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

1. Comunicaciones Punto a Punto
2. Operaciones Colectivas
3. Agrupamiento de procesos
4. Contextos de comunicación
5. Topologías de procesos
6. Soporte para Fortran y C (y otros … Python, Java, .NET, etc.)
7. Manejo del entorno de programación
8. Interfaz personalizada

¿Qué permite el estándar?

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Disponer de un amplio conjunto de rutinas de comunicación punto a punto y entre grupos de procesos
- Definir contextos de comunicación entre grupos de procesos
- Especificar diferentes topologías de comunicación
- Crear tipos de datos derivados para enviar mensajes que contengan datos no contiguos en memoria
- Hacer uso de comunicación asincrónica
- Administrar eficientemente el pasaje de mensajes
- Portabilidad total
- Existe una especificación formal de MPI
- Existen varias implementaciones de calidad y libre acceso de MPI disponibles (LAM, MPICH, CHIMP, OpenMPI, etc.), a su vez, varios fabricantes cuentan con su propia implementación

MPI: conceptos y definiciones

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

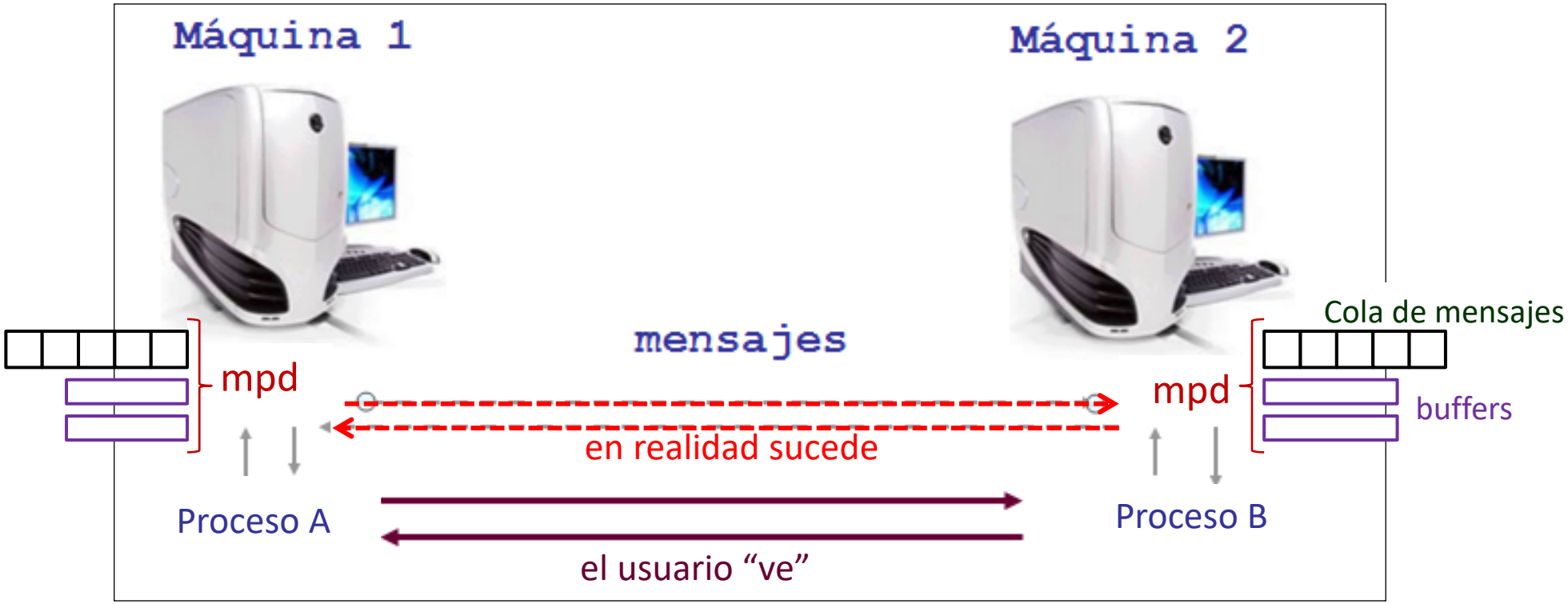
- Rango: Identificador de proceso (número entero). Es utilizado por el programador para especificar origen y destino de mensajes y para control de ejecución.
- Grupo: Conjunto ordenado de N procesos. Cada proceso en un grupo se asocia con un rango único, entre 0 y N-1.
- Contexto: Extiende el concepto de tag de mensajes (previamente usado en otras bibliotecas, e.g., PVM). Se definen contextos de comunicación que pasan a ser manejados por el sistema a través de los comunicadores.
- Comunicador: Asocia los conceptos de grupo y contexto para proveer un espacio común de comunicación entre procesos. Permite especificar el conjunto de procesos que participan en las operaciones colectivas.

MPI: conceptos y definiciones

- Buffer de la aplicación: espacio de memoria que contiene datos a enviar o recibidos por el programa
  - Es manejado explícitamente por el programador
- Buffer del sistema: espacio de memoria del sistema (MPI) para almacenar mensajes
  - Un mensaje en el buffer de la aplicación se copia a/desde el buffer de sistema para poder efectuar las operaciones send/receive
  - Permite la implementación de comunicaciones asincrónicas
  - El buffer del sistema es creado cuando comienza el programa (ni el programador ni el usuario lo manipulan de ninguna manera)
  - Existe un buffer del sistema por cada tarea, que se encarga de manejar múltiples mensajes. El mensaje será copiado del buffer del sistema a la tarea receptora del mensaje, cuando se invoque el receive

MPI: conceptos y definiciones

- Demonios, buffers y colas de mensajes
  - Las comunicaciones se realizan utilizando procesos en ejecución permanente (demonios), buffers internos y colas de mensajes para almacenar la información



Modos de envío

- Modos bloqueantes
  - Un send/receive bloqueante suspende la ejecución del programa hasta que el buffer que se está enviando/recibiendo es seguro de usar
  - En el caso de un send bloqueante, esto significa que los datos a ser enviados han sido copiados en el buffer del sistema (envío) y no necesariamente han sido recibidos por la tarea que recibe
- Modos no bloqueantes
  - Las llamadas no bloqueantes retornan inmediatamente después de ser iniciada la comunicación. El programador no sabe si los datos han sido enviados o copiados en el buffer del sistema (envío) o si los datos ha ser recibidos han llegado. Por ello, antes de utilizar el buffer de la aplicación, el programador debe verificar su status

Modos de envío

- Envío bloqueante:
  - Obtener el espacio necesario del buffer del sistema, copiar los datos desde el buffer de aplicación
  - Es seguro enviar nuevos valores al buffer de aplicación luego de que la rutina retorna
- Envío NO bloqueante:
  - Retorna luego de solicitar espacio en el buffer del sistema, sin esperar que el sistema copie los datos del buffer de aplicación
  - No es seguro copiar nuevos datos al buffer de aplicación (el programador debe chequear que su buffer esté libre)

1 Bloqueante y no bloqueante refieren al CONTROL y a la seguridad de utilización del buffer de la aplicación

2 No debe confundirse con el concepto de comunicaciones SINCRÓNICAS y ASINCRÓNICAS

Ejecución en MPI

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Los demonios de MPI (mpd) actúan como representantes de los procesos ejecutando en cada host
  - Para el programador, la comunicación se realiza proceso a proceso (alto nivel)
  - MPI implementa la comunicación entre procesos y demonios, y entre demonios mpd distribuidos usando sockets (bajo nivel)
- Existen devices de comunicación que permiten a dos procesos ejecutando en el mismo host comunicarse utilizando memoria compartida
- En ciertas versiones de MPI existen scripts para levantar/bajar los demonios, o se utiliza un manejador específico (e.g., hydra)

La API de MPI

Las seis rutinas básicas de MPI

Presentes en todos los programas MPI

- Inicializar:

```
ierr = MPI_Init(&argc,&argv)
```

- Número de procesos en ejecución:

```
ierr = MPI_Comm_size(MPI_COMM_WORLD,&npes)
```

- Identificador del proceso es su contexto de comunicación:

```
ierr = MPI_Comm_rank(MPI_COMM_WORLD,&iam)
```

- Envio de mensajes:

```
ierr = MPI_Send(buff,count,datatype,destination,tag,comm)
```

- Recepción de mensajes:

```
ierr = MPI_Recv(buff,count,datatype,source,tag,comm,state)
```

- Finalizar:

```
ierr = MPI_Finalize()
```

Información del entorno

- MPI_COMM_WORLD identifica todos los procesos involucrados en un cómputo MPI
- MPI_Comm_size(comm, nproc)
  - Retorna en nproc el número de procesos en el comunicador comm
  - Sintaxis: `int MPI_Comm_size(MPI_Comm comm, int *nproc)`
- MPI_Comm_rank(comm, my_id)
  - Retorna el identificador o rango de un proceso en el comunicador comm
  - Sintaxis: `int MPI_Comm_rank(MPI_Comm comm, int *my_id)`
- MPI_Get_processor_name(name, resultlen)
  - Retorna en name el nombre del procesador en que ejecuta el proceso invocante y en resultlen la longitud de name, en número de caracteres
  - Sintaxis: `int MPI_Get_processor_name(char *name, int *resultlen)`

Medir tiempos

- MPI_Wtime retorna el número de segundos transcurridos a partir de cierto tiempo pasado
  - Sintaxis: `double MPI_Wtime(void)`
- MPI_Wtick determina los segundos entre tics sucesivos del reloj
  - Si el reloj esta implementado en hardware como un contador que se incrementa cada milisegundo, entonces MPI_Wtick retorna 10^{-3}.
  - Sintaxis: `double MPI_Wtick(void)`
- Ejemplo de aplicación

```
double tiempo_inicial, tiempo_final;
...
tiempo_inicial = MPI_Wtime();
/* ... código al que se le medira el tiempo ... */
tiempo_final = MPI_Wtime();
printf("Código ejecutado en %f segundos\n",tiempo_final – tiempo_inicial);
```

Compilación y ejecución con MPI

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Implementación MPICH de Argonne National Lab, USA
- Compilación C:
  - mpicc: Provee opciones y bibliotecas necesarias para compilación y linkedición de programas MPI escritos en C
- Ejecución
  - mpiexec: ejecución de programa MPI
  - mpirun es un alias para mpiexec
  - Sintaxis: `mpiexec -np <N> -hostfile <filename> <program>`
  - np: número de procesos paralelos
  - hostfile (o -machinefile): archivo con nombre de máquinas a utilizar (y procesos por máquina)
  - program: programa a ejecutar

Ejemplo: hello world

- Esquema de multiprocesamiento simétrico (SPMD)
- Procesos idénticos imprimen mensajes de forma centralizada, indicando su rango y el número de procesos

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

int main (int argc, char *argv[]) {
  int rank, size;

  MPI_Init(&argc, &argv); /* iniciar MPI */
  MPI_Comm_rank(MPI_COMM_WORLD,&rank); /* obtener id del proceso */
  MPI_Comm_size(MPI_COMM_WORLD,&size); /* obtener número de
                                          procesos */

  printf("Hello world from process %d of %d\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

Ejemplo: hello world

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- El código se salva en un archivo .c, por ejemplo, hello_world.c
- Compilar:
  - Interactivo en host local:

```
mpicc hello_world.c –o hello_world
```

- Ejecutar
  - Interactivo en host local: `mpirun –np 2 ./hello_world [-host mach]`
- Ver/obtener resultados
  - Interactivo: resultados se despliegan en pantalla

Ejemplo: hello world (2)

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Esquema de multiprocesamiento simétrico
- Procesos idénticos imprimen mensajes de forma centralizada, indicando nombre del host donde ejecutan

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

int main (int argc, char *argv[]) {
  int rank, size, length;
  char host[20];

  MPI_Init (&argc, &argv);    /* iniciar MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank); /* obtener id del proceso */
  MPI_Comm_size (MPI_COMM_WORLD, &size); /* obtener número de procesos */
  MPI_Get_processor_name(host,length); /* obtener nombre del host remoto */

  printf("Hello from process %d of %d, in host %s\n", rank, size, host);
  MPI_Finalize();
  return 0;
}
```

Funciones para envío de mensajes

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Formato genérico de pasaje de mensajes estándar:

send (address, length, destino, tag)

address: dirección de memoria del inicio del buffer que contiene los datos (buffer de la aplicación)

lenght: largo en bytes del mensaje

destino: identificación del proceso destino

tag: tipo de mensaje, permite discriminar en la recepción

Send en MPI


```
MPI_Send(buffer, count, datatype, dest, tag, comm)
```

buffer: Dirección de inicio del buffer de envío

count: Número de elementos a enviar o a recibir

datatype: Tipo de dato de cada elemento

dest: Identificador del destinatario (ranking)

tag: Etiqueta del mensaje

comm: Representa el dominio de comunicación

Receive en MPI

```
MPI_Recv(buffer, count, datatype, src, tag, comm, status)
```

buffer: Dirección de inicio del buffer de recepción

count: Número de elementos a enviar o a recibir

datatype: Tipo de dato de cada elemento

source: Rango del remitente (permite modo promiscuo con el wildcard MPI_ANY_SOURCE)

tag: Etiqueta del mensaje (permite el wildcard MPI_ANY_TYPE)

comm: Representa el dominio de comunicación

status: Permite obtener información del mensaje recibido

Contar elementos recibidos

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Cantidad de elementos recibidos: MPI_Get_count

- Retorna el número de elementos recibidos
- El parámetro datatype (tipo de dato) debe coincidir con el argumento de la llamada en la rutina de recepción

  - En C

```
int *count;
ierr = MPI_Get_count(status, datatype, &count);
```

Parámetro status

- El parámetro status retorna información adicional
  - Parámetros/resultados de alguna rutina MPI
  - Información adicional de los errores
- El tipo (tag) del mensaje recibido

  En C: `status.MPI_TAG`
- El rango del proceso origen (fuente)

  En C: `status.MPI_SOURCE`
- El código de error de la llamada MPI

  En C: `status.MPI_ERROR`

- Declaración en C: es una estructura predefinida

```
MPI_Status status;
```

Modos de comunicación

- Cuatro modos de envío:
  - estándar, sincrónico, buffereado y pronto
- Recepción estándar
- Se consideran umbrales para el largo de los mensajes, para determinar el bloqueo o no del proceso emisor
- Invocaciones bloqueantes de send/receive pueden estar en correspondencia con invocaciones no bloqueantes
  - bloqueante suspende la ejecución hasta que el buffer del sistema sea seguro de usar
  - no bloqueante no suspende la ejecución, permite simultaneidad con otras tareas (como contrapartida, es menos seguro)

Envío estándar

- Si se utiliza un buffer local, el invocador es liberado cuando se copió el mensaje y no es necesario esperar un receptor
- Si no se utiliza un buffer, el invocador debe esperar hasta que un receptor invoque receive. La operación es no-local, ya que depende del receptor
- Las implementaciones pueden utilizar el buffer local o no, dependiendo del tamaño del mensaje a transmitir
- En el envío estándar el emisor puede invocar el send sin importar si un receptor realizó un receive todavía

Envío estándar bloqueante

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

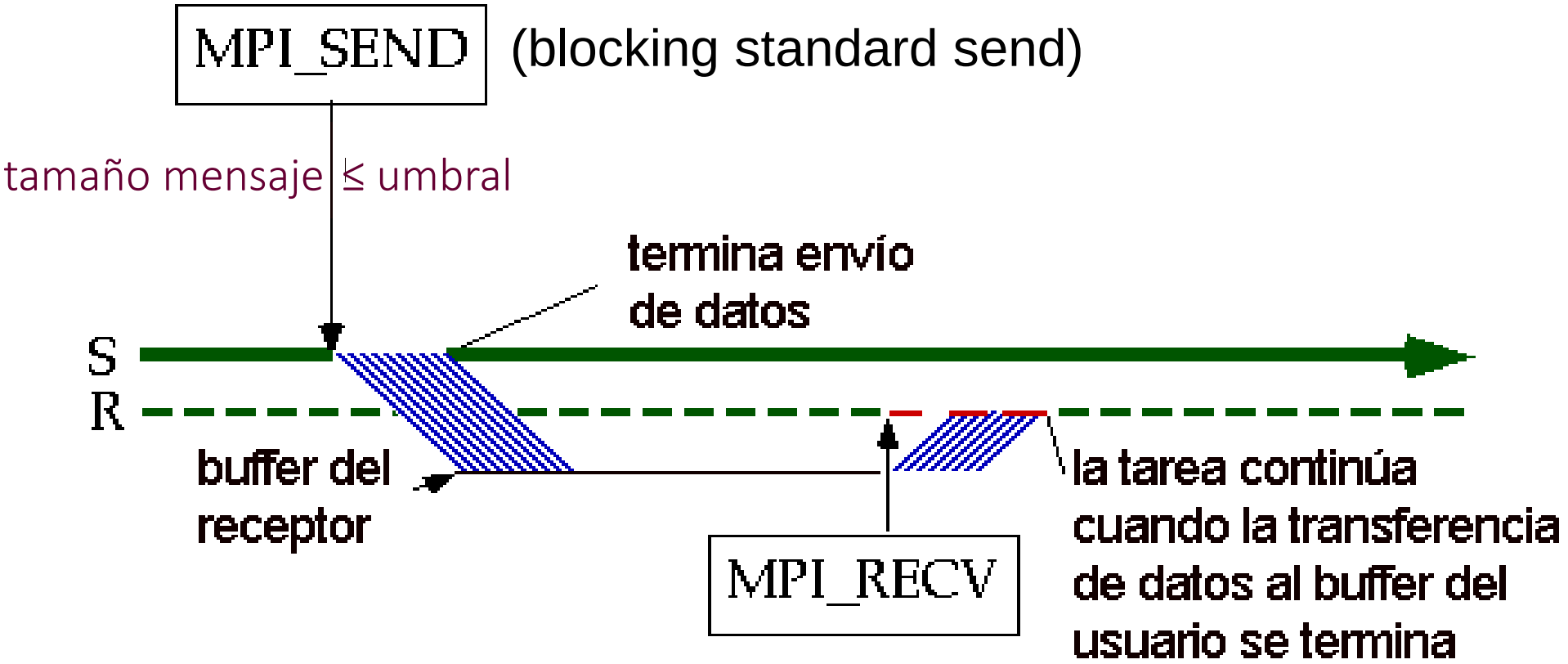
Blocking standard send MPI_Send

```
int MPI_Send(const void *buf, int count, MPI_Datatype
             datatype, int dest, int tag, MPI_Comm comm)
```

- Se copia el mensaje en el buffer del sistema del nodo receptor, entonces la tarea que ejecuta el MPI_Send continúa con su ejecución de modo seguro (no hay riesgo al reutilizar el buffer de aplicación)

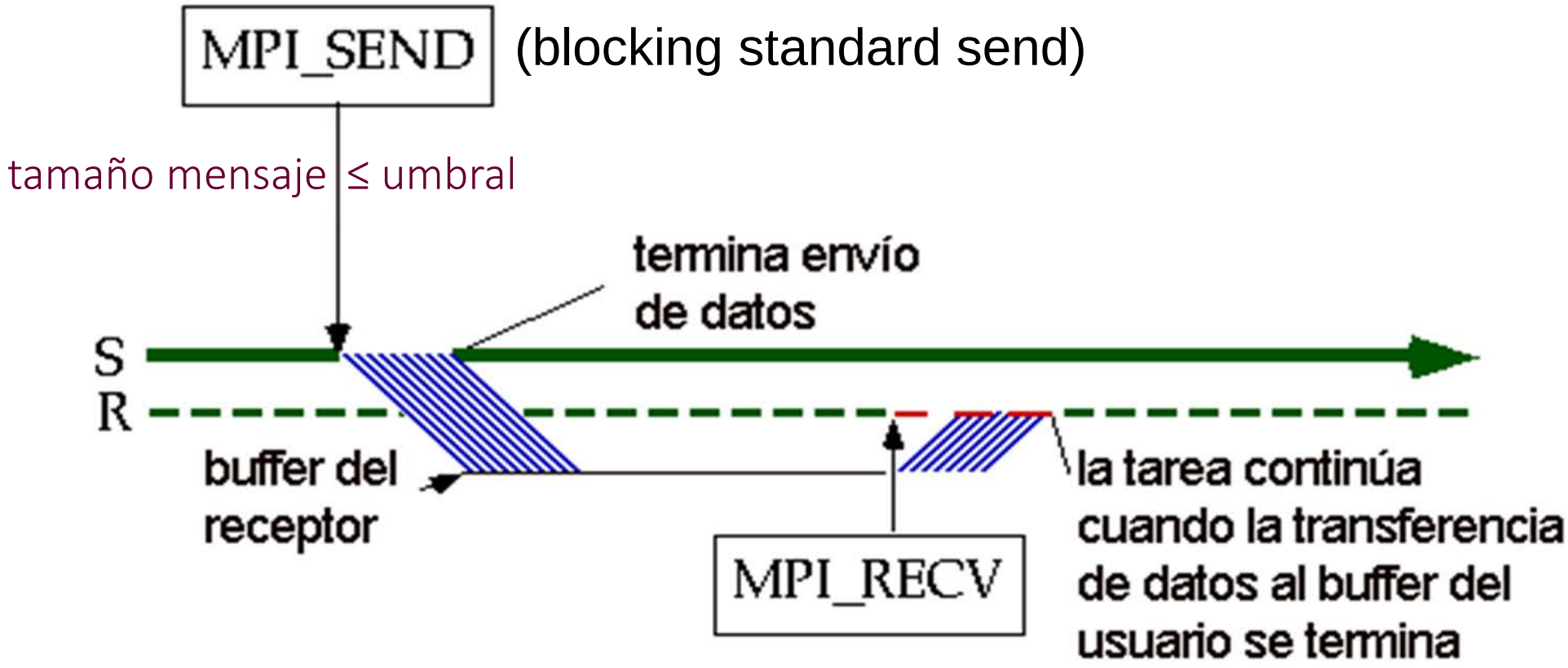
Envío estándar bloqueante

Caso 1) tamaño de mensaje ≤ umbral: el mensaje se almacena en el buffer del receptor



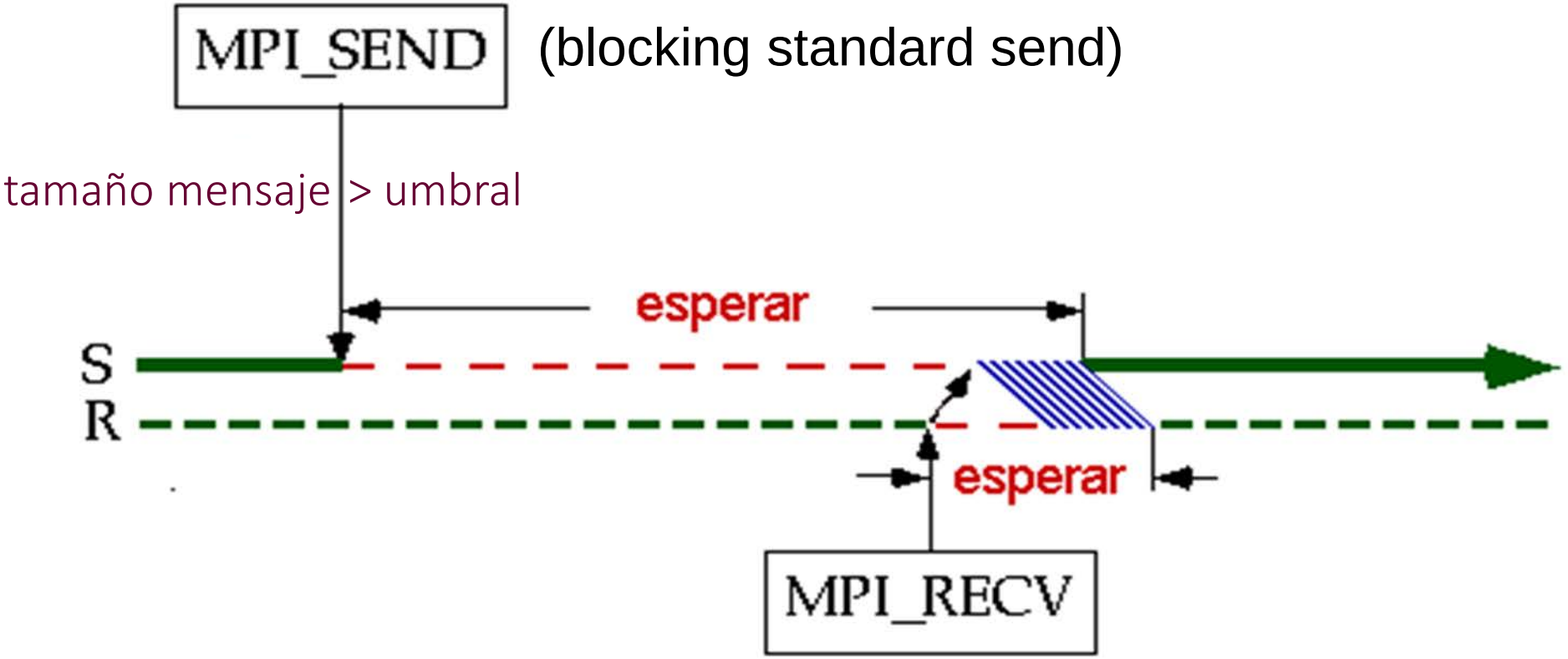
Envío estándar bloqueante

Caso 1) tamaño de mensaje ≤ umbral: el mensaje se almacena en el buffer del receptor



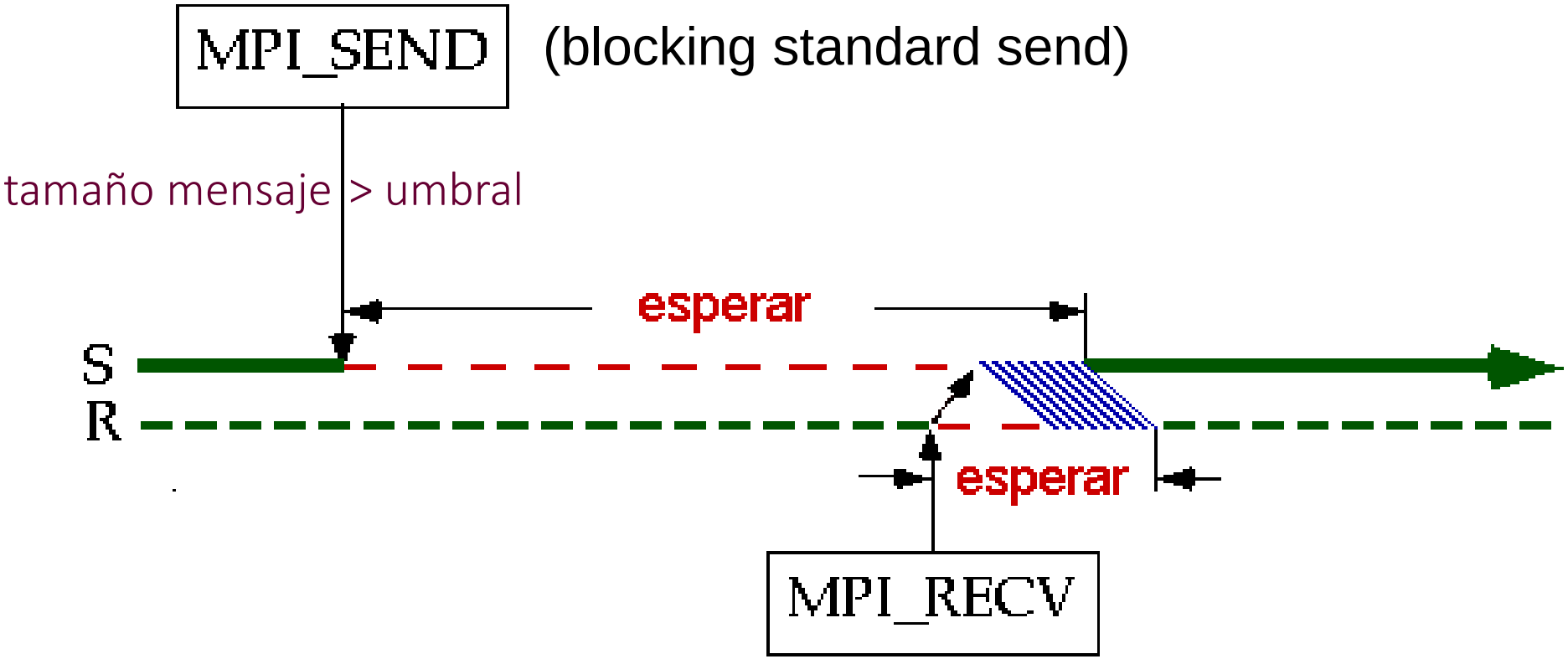
Envío estándar bloqueante

Caso 2) tamaño del mensaje > umbral: mensaje no se almacena en buffer del receptor



Envío estándar bloqueante

Caso 2) tamaño del mensaje > umbral: mensaje no se almacena en buffer del receptor



Umbral y modos de comunicación

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Umbral = “eager limit”, depende de la implementación y no del estándar
- Dos modos de comunicación:
  - Eager (“ansioso”): tan rápido como sea posible, usado para mensajes cortos
  - Rendezvous: “request to send/clear to send” (RTS/CTS), para mensajes largos
- Protocolos eager: útiles cuando la latencia es importante
  - Por ejemplo, enviar un mensaje de MPI de 4 bytes sobre una red InfiniBand tarda 1.5μs, pero con un protocolo CTS/RTS se incrementa a 4.5μs.
- Protocolos rendezvous: útiles cuando el uso de recursos es prioritario
  - Por ejemplo, si un proceso envía un mensaje de 10MB, el receptor puede postergar la recepción hasta que se ejecute MPI_RECV, implicando que existe un espacio de memoria (buffer) disponible para recibir el mensaje de 10MB.
  - Si el emisor enviara el mensaje de 10MB en modo “eager”, el receptor debería: a) reservar un buffer temporario de 10 MB para recibirlo y b) copiar el mensaje al destino especificado en el MPI_RECV.

Comunicaciones en MPI

- Ejemplo: hello world, modelo SPMD
- Procesos idénticos imprimen mensajes de forma centralizada, indicando nombre del host donde ejecutan

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

int main (int argc, char *argv[]) {
  int rank, size;

  MPI_Init(&argc, &argv); /* iniciar MPI */
  MPI_Comm_rank(MPI_COMM_WORLD,&rank); /* obtener id del proceso */
  MPI_Comm_size(MPI_COMM_WORLD,&size); /* obtener número de
                                          procesos */

  printf("Hello world from process %d of %d\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

Ejemplo: hello world con comunicaciones

- Esquema maestro-esclavo: un proceso (maestro) recibe mensajes de otros procesos (esclavos) y es el encargado de imprimir los mensajes de forma centralizada

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

int main (int argc; char *argv[]) {
  Ejercicio ☺
}
```

Ejemplo: envío de vector

```
#include <mpi.h>
#include <stdio.h>
#define N 10 /* tamaño del vector */
int main (int argc, char **argv) {
  int pid, npr, origen, destino, ndat, tag, i, VA[N];
  MPI_Status info;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);

  for (i=0; i<N; i++) /* inicialización del vector */
    VA[i] = 0;

  if (pid == 0) {   /* el proceso 0 envía */
    for (i=0; i<N; i++)
      VA[i] = i;
    destino = 1;
    tag = 0;
    MPI_Send(&VA[0], N, MPI_INT, destino, tag, MPI_COMM_WORLD);
  }
```

Ejemplo: envío de vector

```
else if (pid == 1) {                    /* el proceso 1 recibe */
    printf("\n VA en P1 antes de recibir datos \n\n");
    for (i=0; i<N; i++)
      printf("%4d", VA[i]);
    printf("\n\n");
    origen = 0;
    tag = 0;
    MPI_Recv(&VA[0], N, MPI_INT, origen, tag, MPI_COMM_WORLD, &info);
    MPI_Get_count(&info, MPI_INT, &ndat);
    printf("P1 recibe de P%d: tag %d, ndat %d \n\n"
             info.MPI_SOURCE, info.MPI_TAG, ndat);
    for (i=0; i<ndat; i++)
      printf("%4d", VA[i]); printf("\n\n");
 }
 MPI_Finalize();
 return 0;
```

Ejemplo: envío de vector

Código completo para copiar, compilar y ejecutar

```
#include <mpi.h>
#include <stdio.h>
#define N 10
int main (int argc, char **argv) {
  int pid, npr, origen, destino, ndat, tag, i, VA[N];
  MPI_Status info;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);
  for (i=0; i<N; i++) VA[i] = 0;
  if (pid == 0) {
    for (i=0; i<N; i++) VA[i] = i;
    destino = 1; tag = 0;
    MPI_Send(&VA[0], N, MPI_INT, destino, tag, MPI_COMM_WORLD);
  } else if (pid == 1) {
    printf("\nVA en P1 antes de recibir datos\n\n");
    for (i=0; i<N; i++) printf("%4d", VA[i]); printf("\n\n");
    origen = 0; tag = 0;
    MPI_Recv(&VA[0], N, MPI_INT, origen, tag, MPI_COMM_WORLD, &info);
    MPI_Get_count(&info, MPI_INT, &ndat);
    printf("P1 recibe de P%d: tag %d, ndat %d \n\n", info.MPI_SOURCE, info.MPI_TAG, ndat);
    for (i=0; i<ndat; i++) printf("%4d", VA[i]); printf("\n\n");
 }
 MPI_Finalize();
 return 0;
```

Comunicaciones no bloqueantes

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Envío: MPI_ISend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request request)

- Los parámetros son los mismos que para MPI_Send, se agrega el parámetro request que establece un enlace entre la operación de comunicación y un objeto MPI (que está oculto)
- Envío no bloqueado (inmediato): se inicia el envío del mensaje al receptor y la rutina retorna inmediatamente (no se garantiza el uso seguro del buffer de la aplicación)

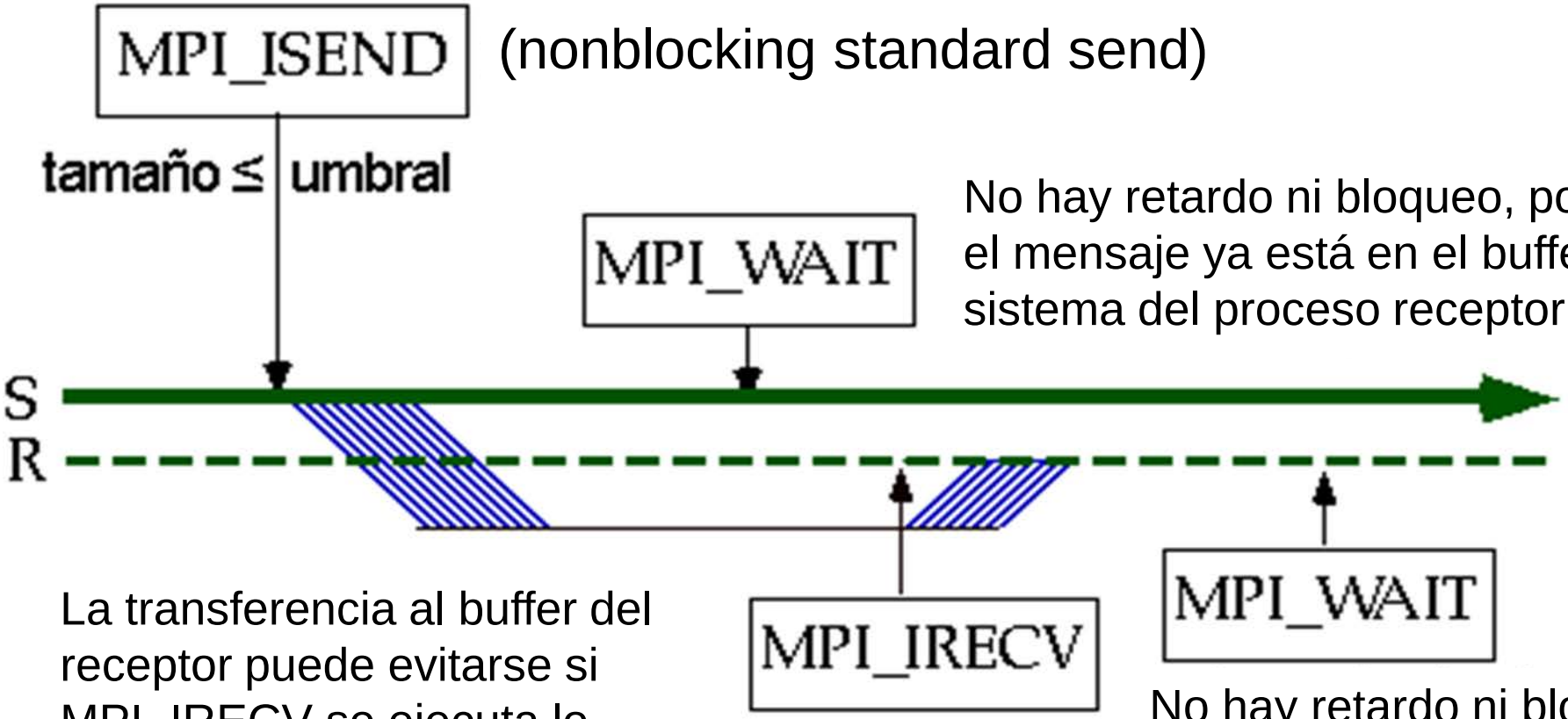
Comunicaciones no bloqueantes

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

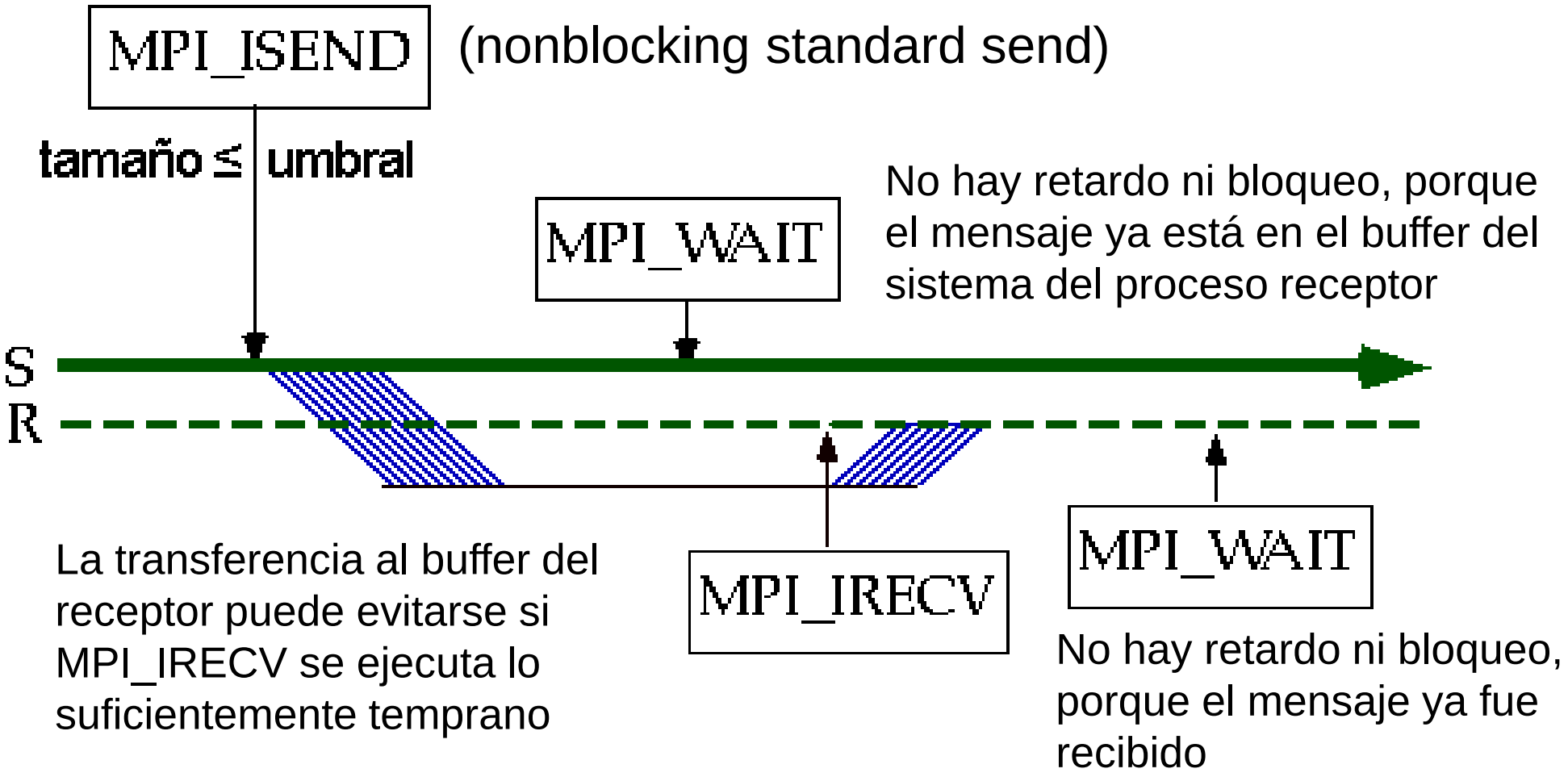
- Los parámetros son los de MPI_Recv. Se sustituye el parámetro status por request para establecer un enlace entre la operación de recepción y un objeto MPI (que está oculto)

- Recepción sin bloqueo (inmediata): se retorna el control apenas se inicia la copia de los datos desde el buffer del sistema.

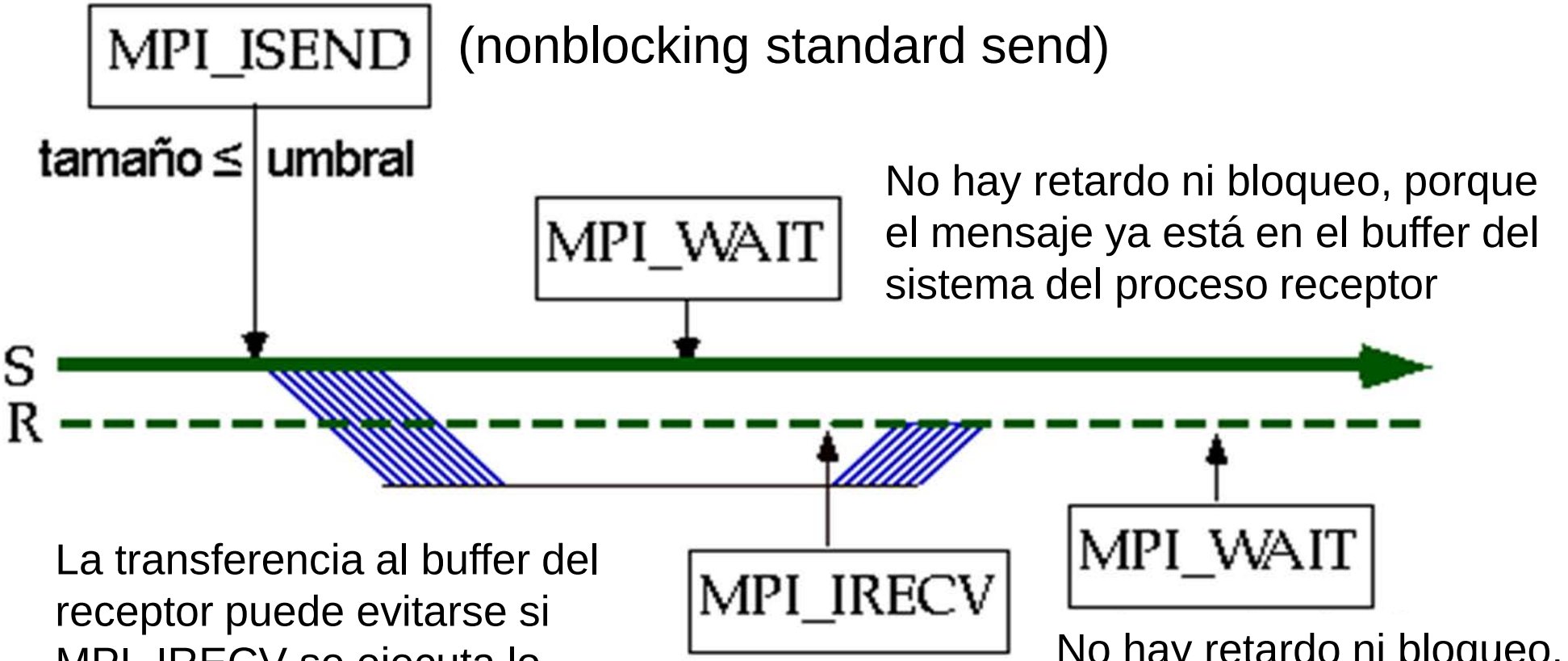
Envío estándar no bloqueante



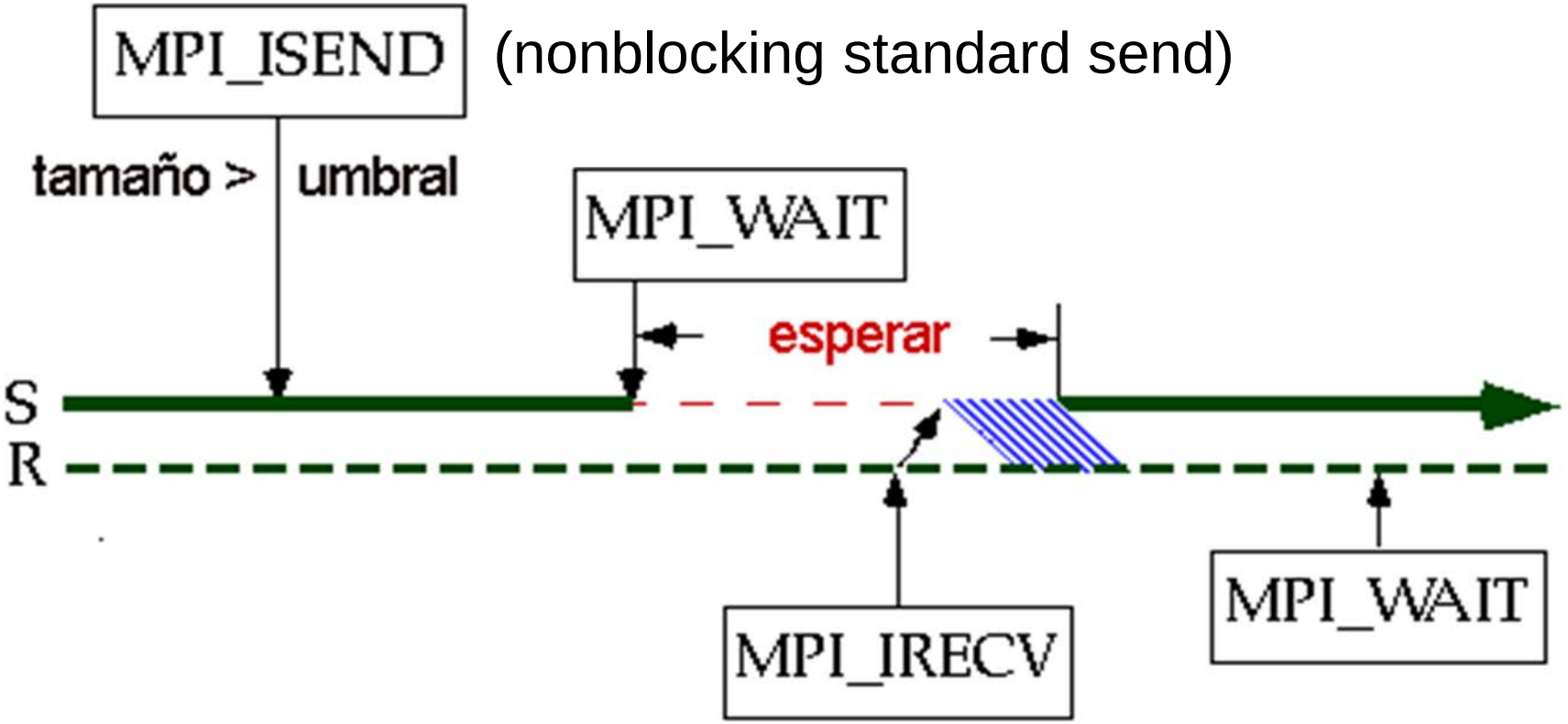
Envío estándar no bloqueante



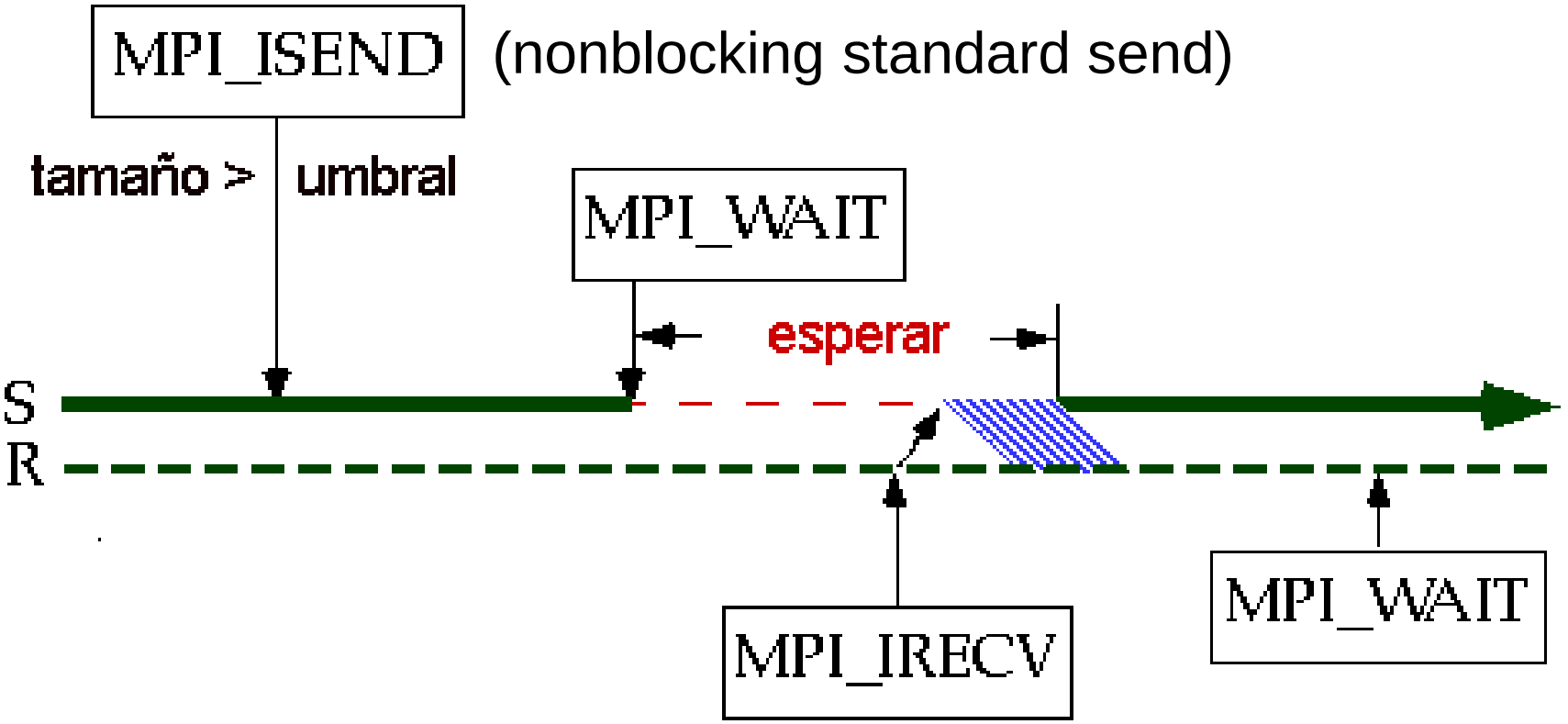
Envío estándar no bloqueante



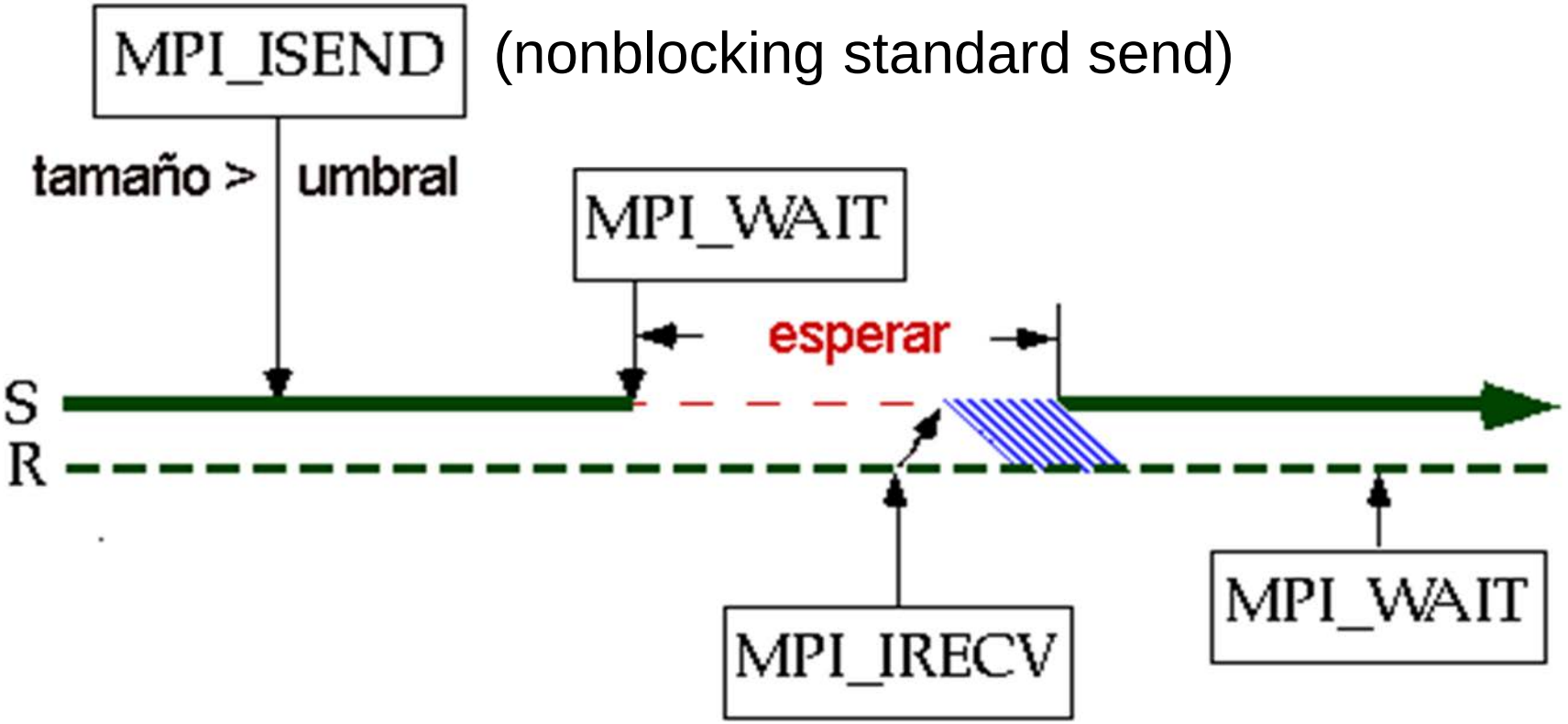
Envío estándar no bloqueante



Envío estándar no bloqueante



Envío estándar no bloqueante



Envío estándar no bloqueante

Non-blocking standard send MPI_ISend

```
int MPI_ISend(const void *buf, int count, MPI_Datatype
              datatype, int dest, int tag, MPI_Comm comm,
              MPI_Request request)
```

- Se inicia el envío del mensaje al receptor y la rutina retorna inmediatamente (no se garantiza el uso seguro del buffer de la aplicación)
- Permite solapar cómputo y comunicaciones

Solapar cómputo y comunicaciones

- Los modos no bloqueantes permiten solapar procesamiento y comunicaciones
- MPI_WAIT espera por una operación (identificada por un request), retorna información en el parámetro status

```
MPI_Request *request;
MPI_Status *status;
```

```
/* Envío (potencialmente muchos datos) */
MPI_ISend(buffer,count,datatype,dest,tag,com,request)
/* Cálculos que no modifican el buffer */
Compute();
```

```
/* Espera por terminación de envío */
MPI_Wait(request, status)
/* Es seguro modificar el buffer */
```

Solapa comunicación y cómputo

Solapar cómputo y comunicaciones

- Send bloqueante

```
buf[] = contenido;
MPI_Send(buf, len, ...);
//cuando MPI_Send retorna se puede reusar el buffer
buf[0] = 1;
MPI_Send(buf, 1, ...);
```

- Send no bloqueante: el envío se ejecuta en background, debe tenerse cuidado de no modificar el buffer de aplicación antes que el envío se complete

```
MPI_Request req;
buf[] = contenido;
MPI_ISend(buf, len, ..., &req);
...
buf[0] = 1; // buf podría estar aún en uso por MPI_ISend(buf,len, ...);
```

Solapar cómputo y comunicaciones

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Un correcto uso de la comunicación no bloqueante

```
MPI_Request req;
buf[] = contenido;
MPI_Isend(buf, len, ..., &req);
...
// Realizar operaciones que no involucren modificar buf
...
...
MPI_Wait(&req, MPI_STATUS_IGNORE);
// Espera: permite asegurarse que la operación de envío se completó

// buf puede ser reutilizado
buf[0] = 1;
MPI_Send(buf, 1, ...);
```

Comunicaciones no bloqueantes: wait y test

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- MPI_WAIT y MPI_TEST permiten esperar o verificar si se completó una comunicación no bloqueante
- Competar un envío no bloqueante
  - Esperar a que el usuario sea libre de utilizar de modo seguro el buffer de la aplicación sin perder información
  - No significa que el mensaje se haya recibido, sino que ya fue correctamente almacenado en los buffers de MPI
- Completar una recepción:
  - Indica que el buffer de recepción contiene el mensaje, y que el proceso receptor está en condiciones de acceder a la información
  - No indica que la operación de envío se ha completado, pero si que se ha iniciado (send)

Comunicaciones no bloqueantes

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Espera: MPI_Wait

- Permite completar comunicaciones no-bloqueantes
- Es una operación no local
- La finalización de una operación de envío permite al remitente modificar libremente el buffer de envío
- La finalización de una recepción indica al destinatario que el buffer de recepción ya contiene el mensaje esperado
  - En C: ierr = MPI_Wait(request,status);
- Parámetros:
  - request: Enlace con la operación
  - status: Información de la operación

Comunicaciones no bloqueantes

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Verificar: MPI_Test

- Consulta si una comunicación sin bloqueo ha finalizado
- Es una operación local y no involucra esperas
- El valor de flag indica si el mensaje ha sido transferido
  - En C

```
int *flag;
ierr = MPI_Test(request,flag,status);
```

- Parámetros:
  - request: Enlace con la operación
  - flag: Cierto si la operación se completo
  - status: Información de la operación

Comunicaciones no bloqueantes

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Verificar: MPI_Test

- Consulta si una comunicación sin bloqueo ha finalizado
- Es una operación local y no involucra esperas
- El valor de flag indica si el mensaje ha sido transferido
  - En C

```
int *flag;
ierr = MPI_Test(request,flag,status);
```

- Parámetros:
  - request: Enlace con la operación
  - flag: Cierto si la operación se completo
  - status: Información de la operación

Comunicaciones no bloqueantes

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- MPI_Test retorna true exactamente en las situaciones donde MPI_Wait retorna (ambas retornan el mismo valor en status)
- Un MPI_Wait (que puede detener la ejecución del programa) puede reemplazarse en tal caso por un MPI_Test (que no lo detiene)
- Ejemplo de envío no bloqueante y test (en busy waiting), que actúa como una llamada a MPI_Wait

```
ierr = MPI_ISend(buffer, count, datatype, dest,tag, com, request)
/* Cálculos que no modifican el buffer */
repetir
    ierr = MPI_Test(request,flag,status)
mientras ( not flag )
```

Comunicaciones no bloqueantes

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

int main(int argc, char *argv[]){
    int myid,numprocs,tag,source,destination,count,buffer;
    MPI_Status status;
    MPI_Request request;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=1234; source=0; destination=1; count=1;
    request = MPI_REQUEST_NULL;

    if(myid == source){
       buffer=5678;
     MPI_Isend(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD,&request);
    }
```

Comunicaciones no bloqueantes

```
    if(myid == destination){
        MPI_Irecv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&request);
    }
    /* Ejecutar cualquier código que no modifique la variable buffer */
    /* Permite solapar cómputo con las comunicaciones ya iniciadas */
    MPI_Wait(&request,&status);
    if(myid == source){
      printf("processor %d  sent %d\n",myid,buffer);
    }
    if(myid == destination){
      printf("processor %d  got %d\n",myid,buffer);
    }
    MPI_Finalize();
}
```

Otros modos de envío

- Envío sincrónico, buffereado y pronto

AUTOESTUDIO:
SLIDES 67 A 73

Envío sincrónico

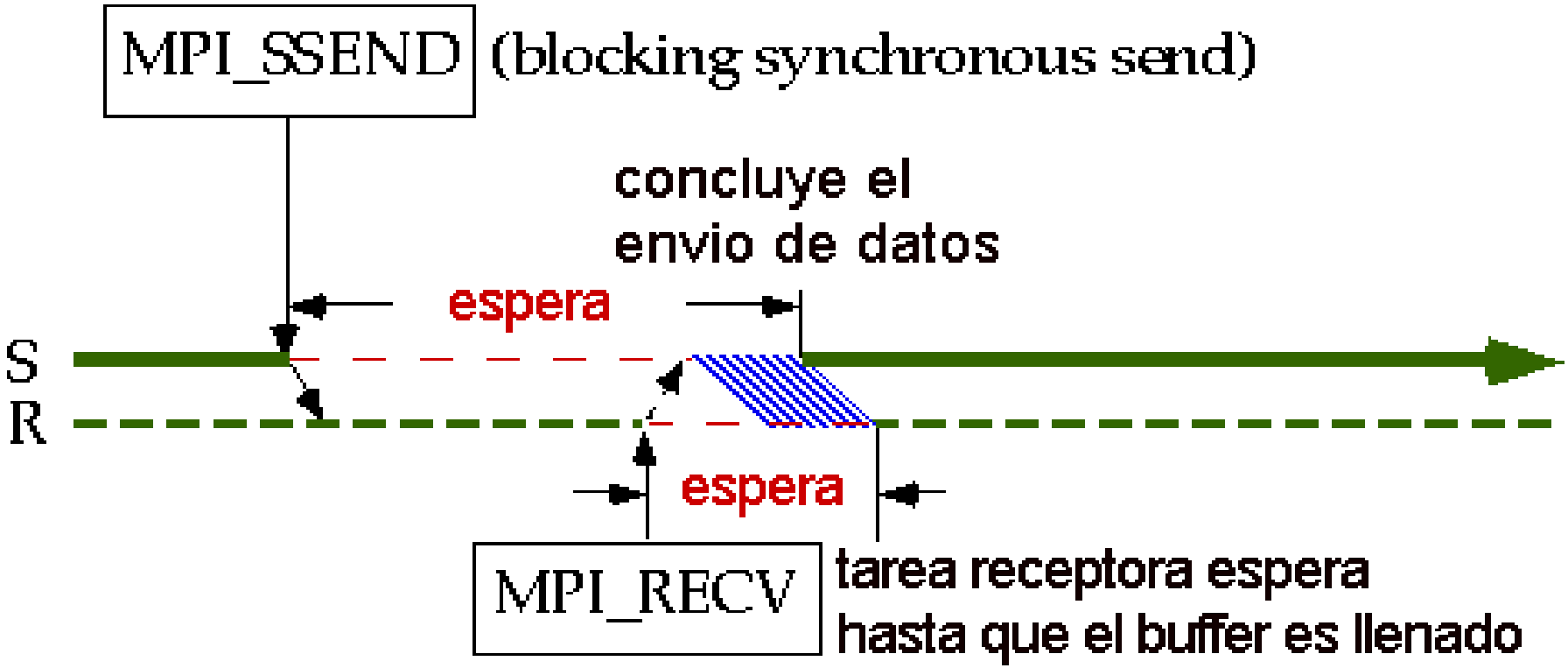
- El emisor puede realizar la invocación sin importar que el receptor haya realizado el receive
- El emisor completará satisfactoriamente la operación una vez que el receptor haya realizado el receive
- Dado su forma de funcionamiento, el emisor tiene claro, luego que se complete la ejecución, en que parte de código está el receptor
- Si el emisor y el receptor utilizan los dos operaciones bloqueantes, el uso del modo sincrónico provee la semántica de comunicación sincrónica.
- Es una operación no-local (non-local)

Envío sincrónico

- Blocking Synchronous Send MPI_Ssend(...)
  - Quien envía el mensaje se bloquea hasta tanto el destinatario comienza a recibir el mensaje. Cuando el Blocking Synchronous Send se ejecuta, la tarea que hace el envío le indica al receptor que tiene un mensaje para él y espera a que el receptor le envíe un mensaje indicándole que está listo para recibir el mensaje. Entonces los datos son transferidos.

- Hay dos fuentes de *overhead:*
  - del sistema (el trabajo que realiza el sistema para enviar el mensaje hacia la tarea destino: copiar el mensaje de la red al buffer de quien recibe)
  - de sincronización (el tiempo esperando para que ocurra un evento en la otra tarea)

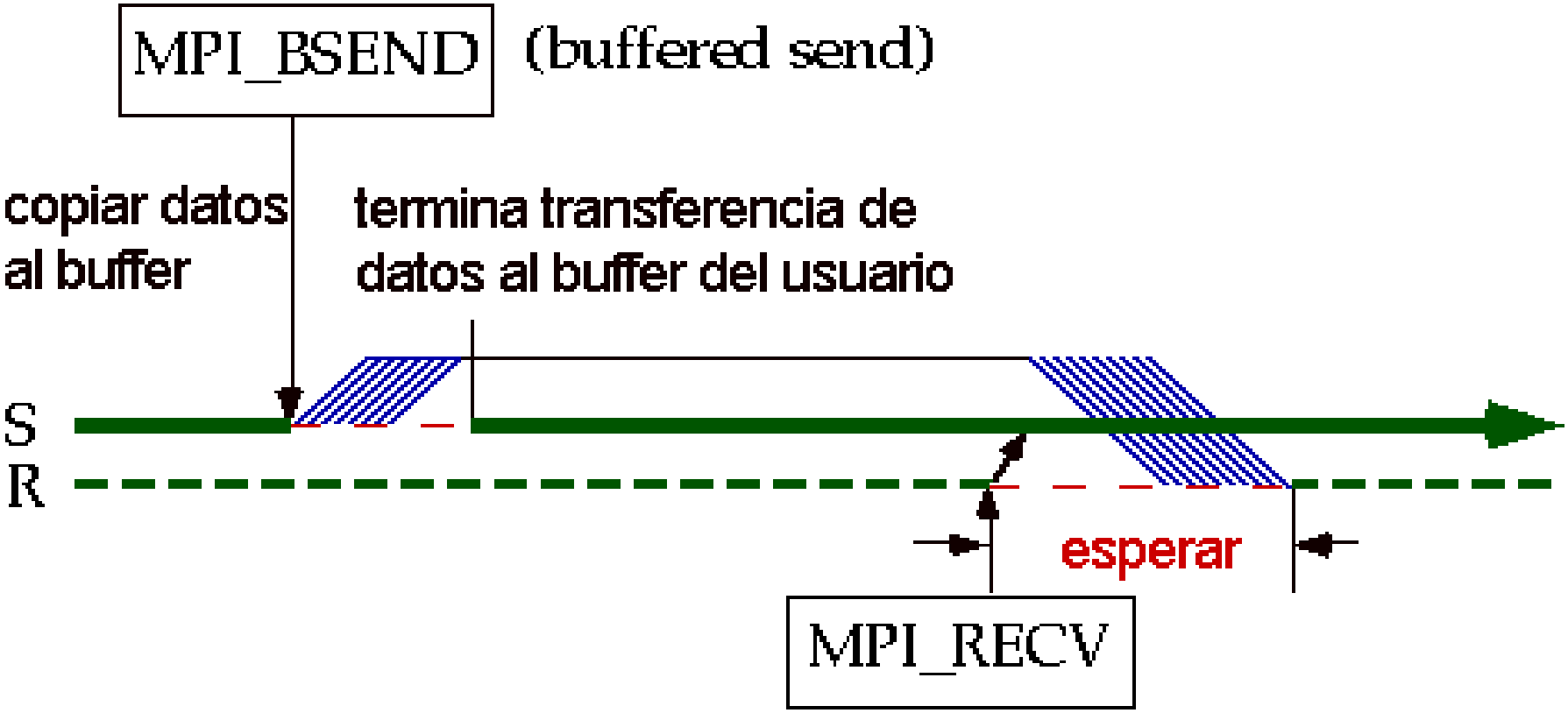
Envío sincrónico



Envío buffereado

- El emisor puede realizar la invocación sin importar que el receptor haya realizado el receive
- La operación puede finalizar completamente antes que el receptor haya realizado el receive, pero, a diferencia del envío estándar, esta operación no debe depender del receptor
- La operación es catalogada como local
- Si el receptor no realizó el receive, la implementación debe copiar el mensaje para permitir que el emisor complete la operación
- El espacio de buffer es administrado completamente por el usuario a través de las operaciones MPI_BUFFER_ATTACH y MPI_BUFFER_DETTACH
- Blocking Buffered Send MPI_Bsend(...)
  - Copia los datos desde el buffer de mensajes a un buffer dado por el usuario y luego retorna. Los datos serán copiados desde el buffer dado por el usuario a la red cuando se reciba una notificación indicando "estoy listo para recibir"

Envío buffereado

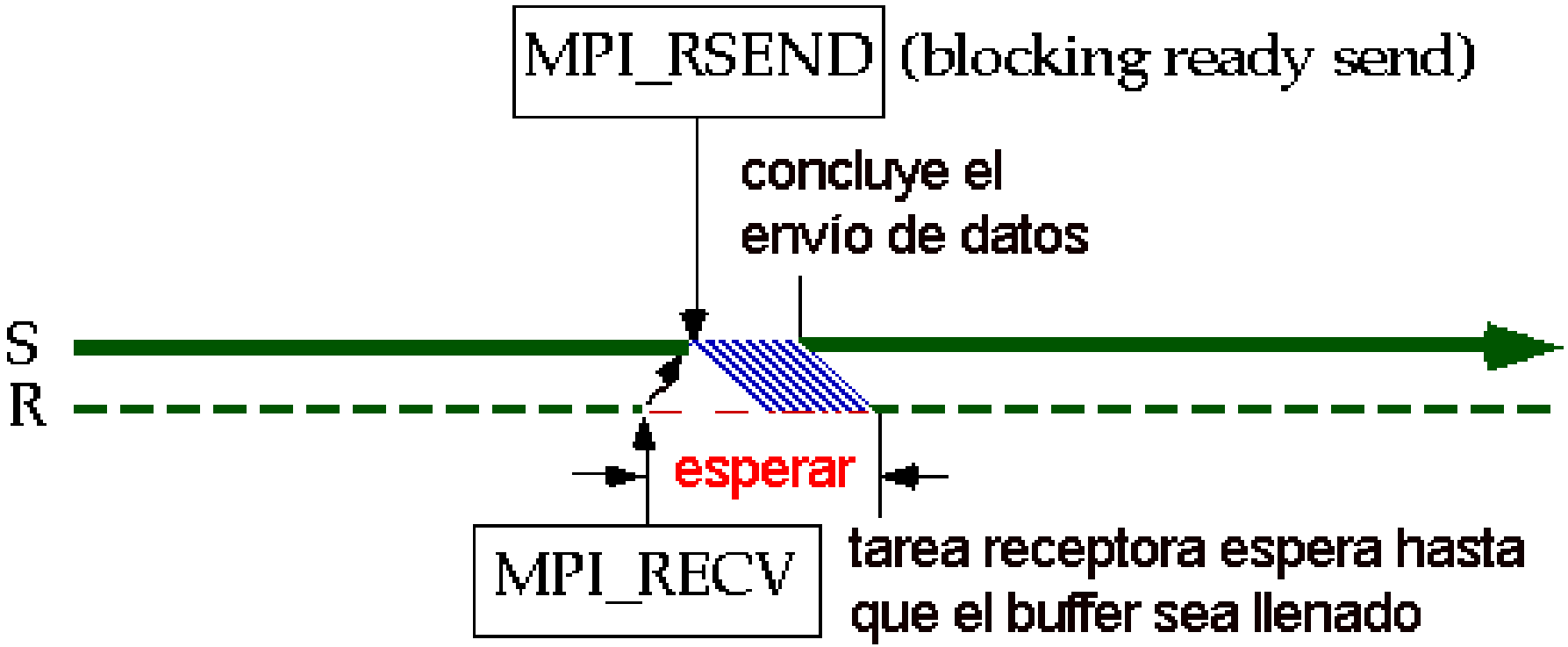


Envío pronto

- El emisor solo puede realizar la invocación si el receptor ya realizó el receive
- Es una operación que permite eliminar controles de sincronización (hand-shake)
- De este modo, permite implementar protocolos específicos de comunicación, de forma de lograr mayor performance

- Blocking Ready Send MPI_Rsend(...)
  - Permite al programador notificar al sistema que un receive ya ha sido invocado, por ello puede utilizar un protocolo más rápido si está disponible. El MPI_Rsend envía un mensaje a través de la red. Requiere que haya llegado previamente una notificación indicando "estoy listo para recibir" enviada por una tarea que espera recibir. Si esa notificación no ha llegado, entonces ocurre un error

Envío pronto



Comunicaciones y deadlocks

- Caso de estudio: se debe enviar un mensaje de un proceso (0) a otro (1)
  - No se puede garantizar de forma genérica que exista espacio para almacenar el mensaje: el proceso que envía debe esperar la ejecución de un receive.
- Qué sucede con un programa que requiera envíos cruzados?

Proceso 0
`send (a proceso 1)` `receive (de proceso 1)`

Proceso 1
`send (a proceso 0)` `receive (de proceso 0)`

- El programa es “no seguro“, porque depende de la disponibilidad de buffers para almacenar los mensajes

Comunicaciones y deadlocks: soluciones

- Soluciones para evitar deadlocks y situaciones “no seguras”
- Ordenar los envíos

Proceso 0
`MPI_Send (a proceso 1)` `MPI_Recv (de proceso 1)`

Proceso 1
`MPI_Recv (de proceso 0)` `MPI_Send (a proceso 0)`

- Utilizar envío buffereado (utiliza un buffer explícito)

Proceso 0
`MPI_BSend (a proceso 1)` `MPI_Recv (de proceso 1)`

Proceso 1
`MPI_BSend (a proceso 0)` `MPI_Recv (de proceso 0)`

Comunicaciones y deadlocks: soluciones

- Soluciones para evitar deadlocks y situaciones “no seguras”
- Utilizar comunicaciones no bloqueantes

Proceso 0

`MPI_ISend (a proceso 1)`
`MPI_IRecv (de proceso 1)`
`MPI_Waitall`

Proceso 1

`MPI_ISend (a proceso 0)`
`MPI_IRecv (a proceso 0)`
`MPI_Waitall`

- Utilizar `MPI_Sendrecv()`
  - Envío y recepción simultáneos

Proceso 0
`MPI_Sendrecv (a proceso 1)`

Proceso 1
`MPI_Sendrecv (a proceso 0)`

Comunicaciones y deadlocks: soluciones

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

int main(int argc, char *argv[]){
    int myid, numprocs, left, right;
    int buffer[10], buffer2[10];
    MPI_Request request;
    MPI_Status status;

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

    right = (myid + 1) % numprocs;
    left = myid - 1;
    if (left < 0)
        left = numprocs - 1;

    MPI_Sendrecv(buffer, 10, MPI_INT, left, 123, buffer2, 10, MPI_INT, right,
                 123, MPI_COMM_WORLD, &status);

    MPI_Finalize();
    return 0;
}
```

Comunicaciones y deadlocks: soluciones

- `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;`
- Ejecuta simultáneamente un envío y recepción bloqueantes. Ambas Operaciones utilizan el mismo comunicador, pero los tags pueden ser diferentes.
- Los buffers de envío y recepción deben ser disjuntos.
- Los buffers de envío y recepción pueden tener diferentes largos y diferentes tipos de datos.

Ejemplo: send bloqueante por tamaño del buffer

```
#include <stdio.h>
#include "mpi.h"
#define N 100000
int main(int argc, char** argv) {
 int pid, kont; // Identificador del proceso
 int a[N], b[N], c[N], d[N];
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &pid);
 for (kont=100; kont<=N; kont=kont+100) {
   if (pid == 0) {
     MPI_Send(&a[0], kont, MPI_INT, 1, 0, MPI_COMM_WORLD);
     MPI_Recv(&b[0], kont, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
     printf("emisor %d \n", kont);
   } else {
     MPI_Send(&c[0], kont, MPI_INT, 0, 0, MPI_COMM_WORLD);
     MPI_Recv(&d[0], kont, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
     printf(" receptor %d \n", kont);
   }
 }
 MPI_Finalize();
 return 0;
} /* main */
```