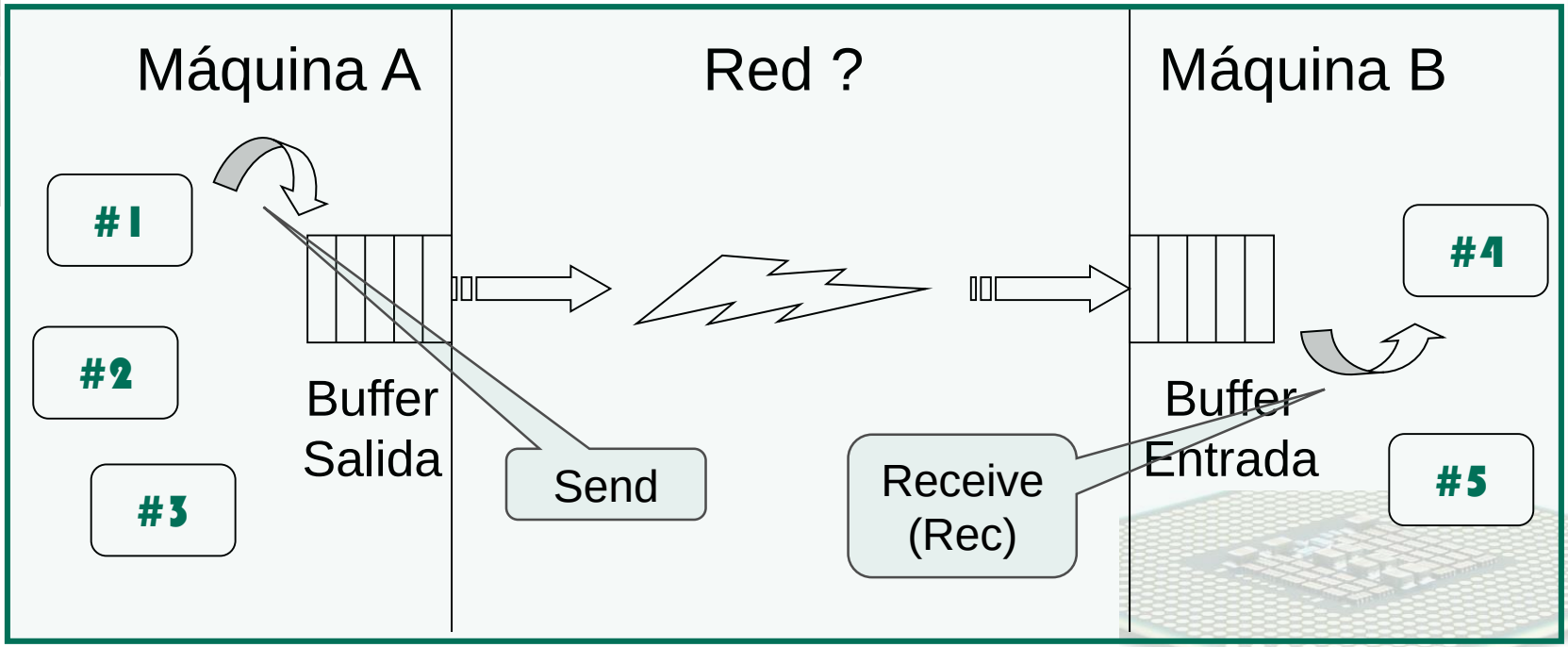
Índice General

1. Introducción e instalación.
2. Comunicación punto a punto bloqueante.
3. Comunicación punto a punto no bloqueante.
4. Llamadas colectivas.
   - Barrier, Broadcast, gather, scatter.
5. Llamadas colectivas.
   - Todos con todos, reduction.
6. Tipos de datos / Comunicadores.

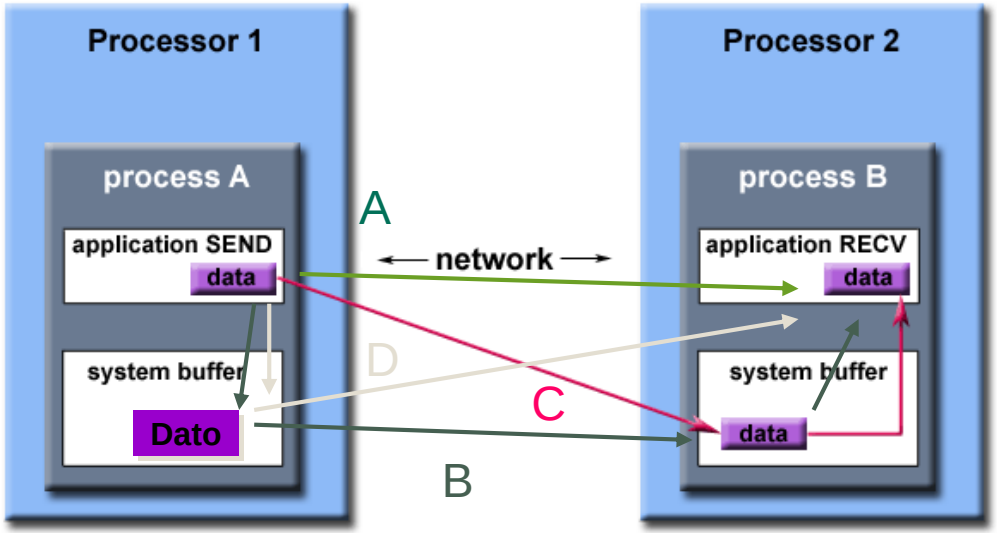
2. Comunicación. Fases

- Definición: Comunicación entre una pareja de procesos.
- Idealmente siempre se realiza en tres fases: **copia** de envío, **transferencia**, **copia** de recepción.



2. Comunicación. Fases

- En el mundo real depende de la implementación del estándar.
  - El buffer del sistema es transparente al programador.
  - Si lo hay, es un recurso finito fácilmente desbordable.
  - No suele estar muy documentado.
  - Puede existir en el envío, la recepción, ambos o ninguno (D, C, B, A).
  - Sólo se puede controlar en el **modo** de envío.

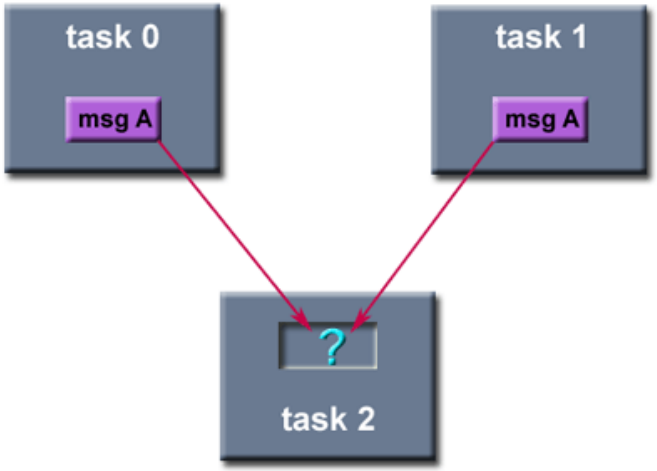


Path of a message buffered at the receiving process

2. Comunicación. Orden y Equidad

- Orden
  - MPI garantiza que los mensajes no se sobrescribirán entre ellos.
  - Si un emisor envía dos mensajes al mismo receptor, estos llegarán en orden.
  - Si hay varios procesos **no** se garantiza el orden.
- Equidad (***fairness***).
  - MPI **no** garantiza la equidad, es responsabilidad del programador evitar la inanición (*starvation*) de los mensajes y que otros se alcancen antes (*overtake*).
    - Ejemplo: Las tareas 0 y 1 envían mensaje en competición a la 2. Una recepción de la 2 coincide con el mensaje de A, luego sólo un send terminará.

Sólo en la recepción



2. Comunicación. Modelo

- Dos **modelos** fundamentales de comunicación:
  - **Bloqueante**: El proceso espera a que el buffer esté libre.
    - Bloqueante en el envío. Se define envío finalizado cuando el proceso puede utilizar el buffer de emisión sin problemas, **NO** cuando lo reciba el receptor.
    - Bloqueante en la recepción. Cuando se tenga un mensaje completo en el buffer de recepción.
  - **No Bloqueante**. Se realiza la operación inmediatamente, después se comprobará si la operación se ha completado.
    - No Bloqueante en el envío. Hay que tener mucho cuidado de no utilizar el buffer de envío sin habernos **asegurado** antes de que está listo, la responsabilidad es nuestra.
    - No Bloqueante en la recepción. Habrá funciones para preguntar por un mensaje.

2. Comunicación. Modos

- El **modo** se refiere al control sobre la comunicación (fases de buffer).
- Existen cuatro modos de **envío**:
  - Modo **Básico (basic)**. No se especifica cómo se hace la comunicación, depende de la implementación, normalmente con buffer para mensajes cortos y sin buffer para largos. Si no hay buffer la terminación no es local.
  - Modo **Buffereado (buffered)**. Se guarda una copia del mensaje en un buffer creado por nosotros. La terminación es local ya que hay buffer.
  - Modo **Síncrono (synchronous)**. Se espera a que el mensaje llegue a destino y se haya comenzado a recibir. Acaba cuando se hace "Receive" en la otra parte. La terminación por definición es no local.
  - Modo **Preparado (ready)**. Sólo acaba el envío si el otro extremo está preparado para la recepción (se ha hecho un "Receive"), en otro caso la llamada dará error ya que no suele haber copias del mensaje.
- La combinación de cuatro **modos** de envío de mensajes y dos **modelos** de comunicación da como resultado **ocho formas** de hacer un envío.
- En cuanto a la recepción sólo hay **dos**, una por modelo.

2. Argumentos de llamadas

- Además, en las funciones de comunicación, podremos dar una serie de argumentos que califican a la comunicación y al mensaje.
  - **Destino**. Identificación del proceso receptor (int).
  - **Dirección** del Mensaje. Dirección de lo transmitido (void *).
  - **Tipo** de dato. Definido en MPI. (MPI_Datatype).
  - **Número** de datos. Número de datos del tipo anterior (int).
  - **Etiqueta**. Identificación del mensaje (int). Existe una constante de MPI para cualquier etiqueta: MPI_ANY_TAG.
  - **Comunicador**. Grupo de procesos donde tiene sentido la identificación (MPI_Comm).
  - **Fuente**. Identificación del proceso emisor (int). Existe una constante de MPI para cualquier fuente: MPI_ANY_SOURCE.
  - **Estatus**. Indicador de estado de la operación (MPI_Status).
  - **Recibo**. Sólo para recepción no bloqueante (MPI_Request).
  - **Comprobante**. Para saber si un mensaje ha llegado (int *).

Envío

Recep.

No blo

Probar

2. Tipos de datos

- Existe una serie de **tipos** de datos para definir de que está compuesto el mensaje, su equivalencia en ANSI C aparece a continuación:

MPI_CHAR	*signed char.*
MPI_SHORT	*signed short int.*
MPI_INT	*signed int.*
MPI_LONG	*signed long int.*
MPI_UNSIGNED_CHAR	*unsigned char.*
MPI_UNSIGNED_SHORT	*unsigned short int.*
MPI_UNSIGNED	*unsigned int.*
MPI_UNSIGNED_LONG	*unsigned long int.*
MPI_FLOAT	*float.*
MPI_DOUBLE	*double.*
MPI_LONG_DOUBLE	*long double.*
MPI_BYTE	*Sin equivalente.*
MPI_PACKED	*Sin equivalente.*

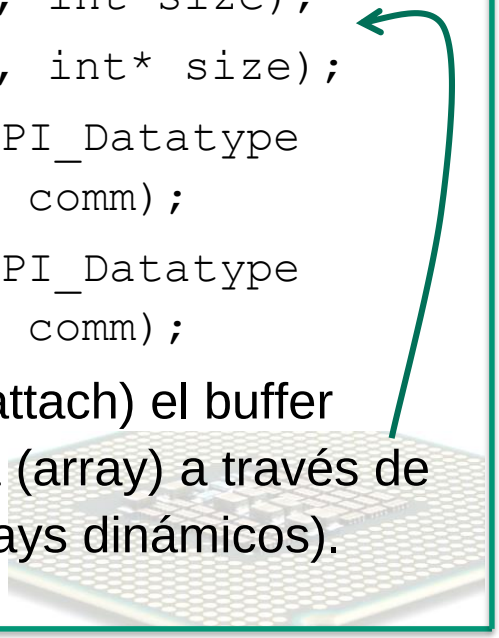
2. Tipos de datos

- En el caso del FORTRAN:

MPI_INTEGER	*INTEGER.*
MPI_REAL	*REAL.*
MPI_DOUBLE_PRECISION	*DOUBLE_PRECISION.*
MPI_COMPLEX	*COMPLEX.*
MPI_LOGICAL	*LOGICAL.*
MPI_CHARACTER	*CHARACTER.*
MPI_BYTE	*Sin equivalente.*
MPI_PACKED	*Sin equivalente.*

2. Comunicación Bloqueante

- Envío (Básico, con Buffer, Síncrono, Ready):
  - `int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`
  - `int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`
    - `int MPI_Buffer_attach(void* buffer, int size);`
    - `int MPI_Buffer_detach(void* buffer, int* size);`
  - `int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`
  - `int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`
- En envío con buffer nosotros le comunicamos (attach) el buffer creado o bien de forma dinámica o bien estática (array) a través de su dirección (en ANSI C 99 se pueden crear arrays dinámicos).



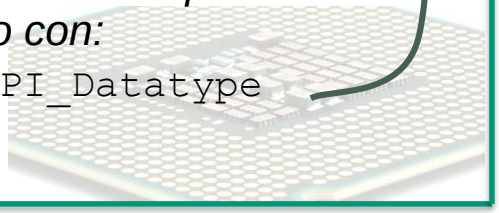
2. Comunicación Bloqueante

- Recepción (sólo hay modo básico):
  - `int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);`
- MPI_Status es un dato definido en ANSI C (en FORTRAN es un array) como una estructura:

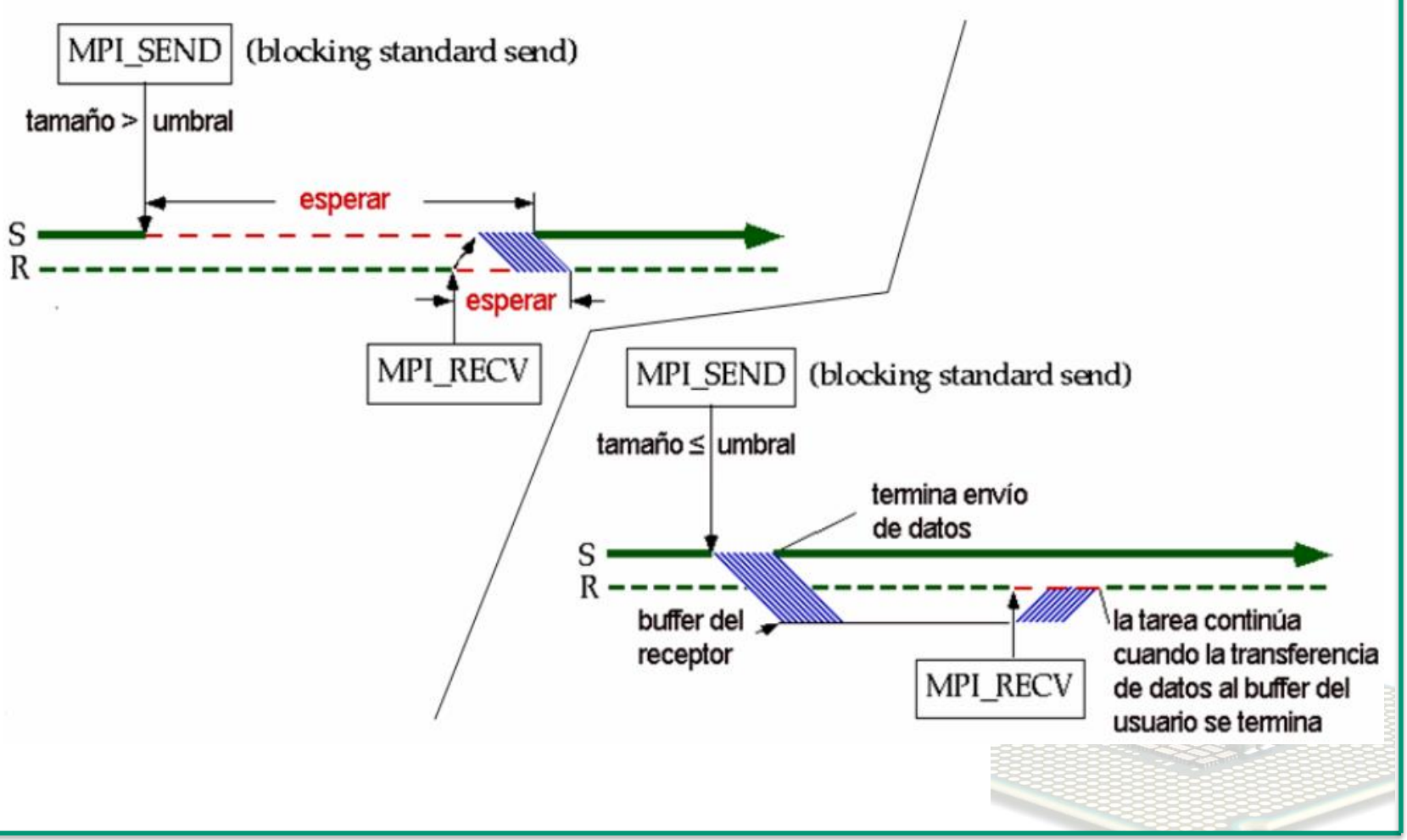
```
typedef struct {
   int MPI_SOURCE;
   int MPI_TAG;
   int MPI_ERROR;
   /* otros campos no accesibles directamente */
} MPI_Status;
```

  - Nos da por tanto la fuente, la etiqueta y si hay error. *Se puede averiguar también la longitud del mensaje recibido con:*

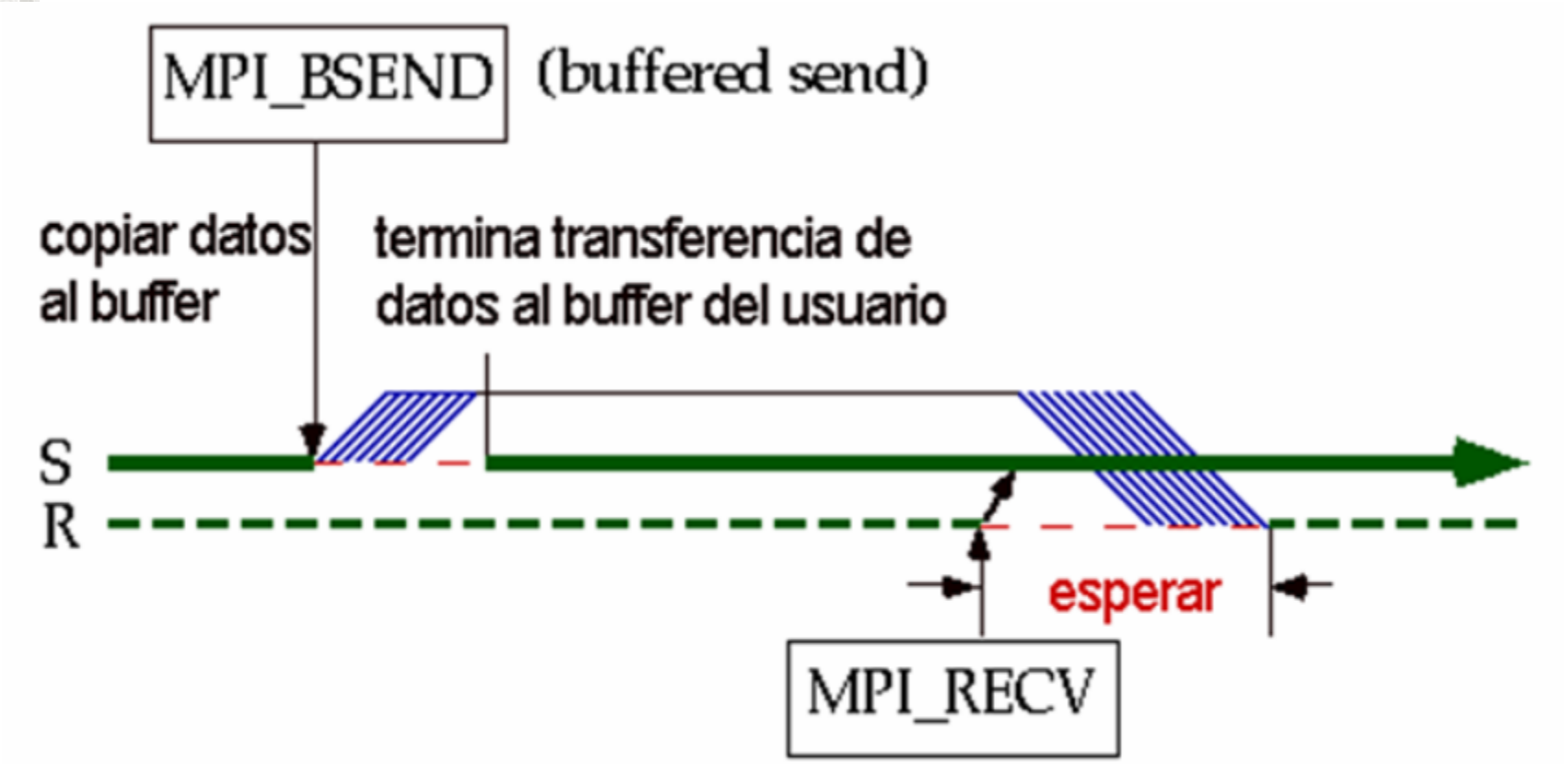
```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count);
```



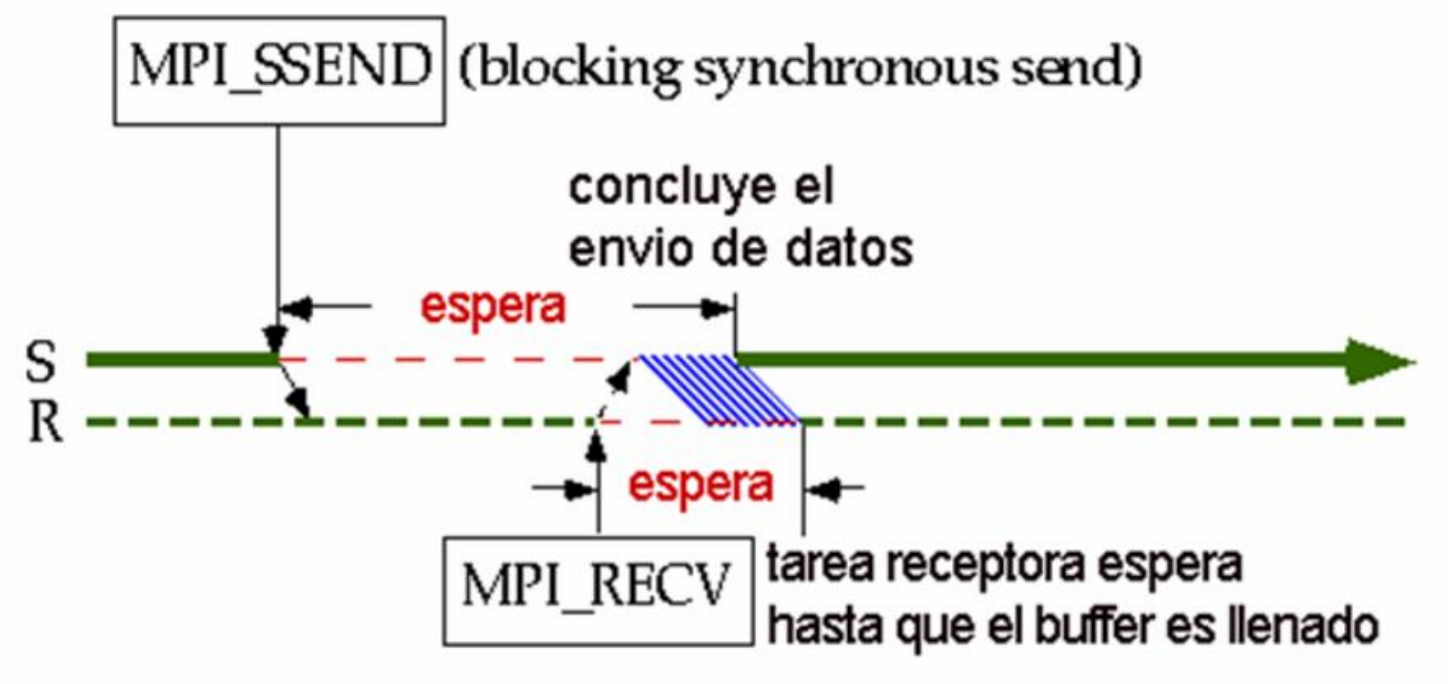
2. Modo Básico



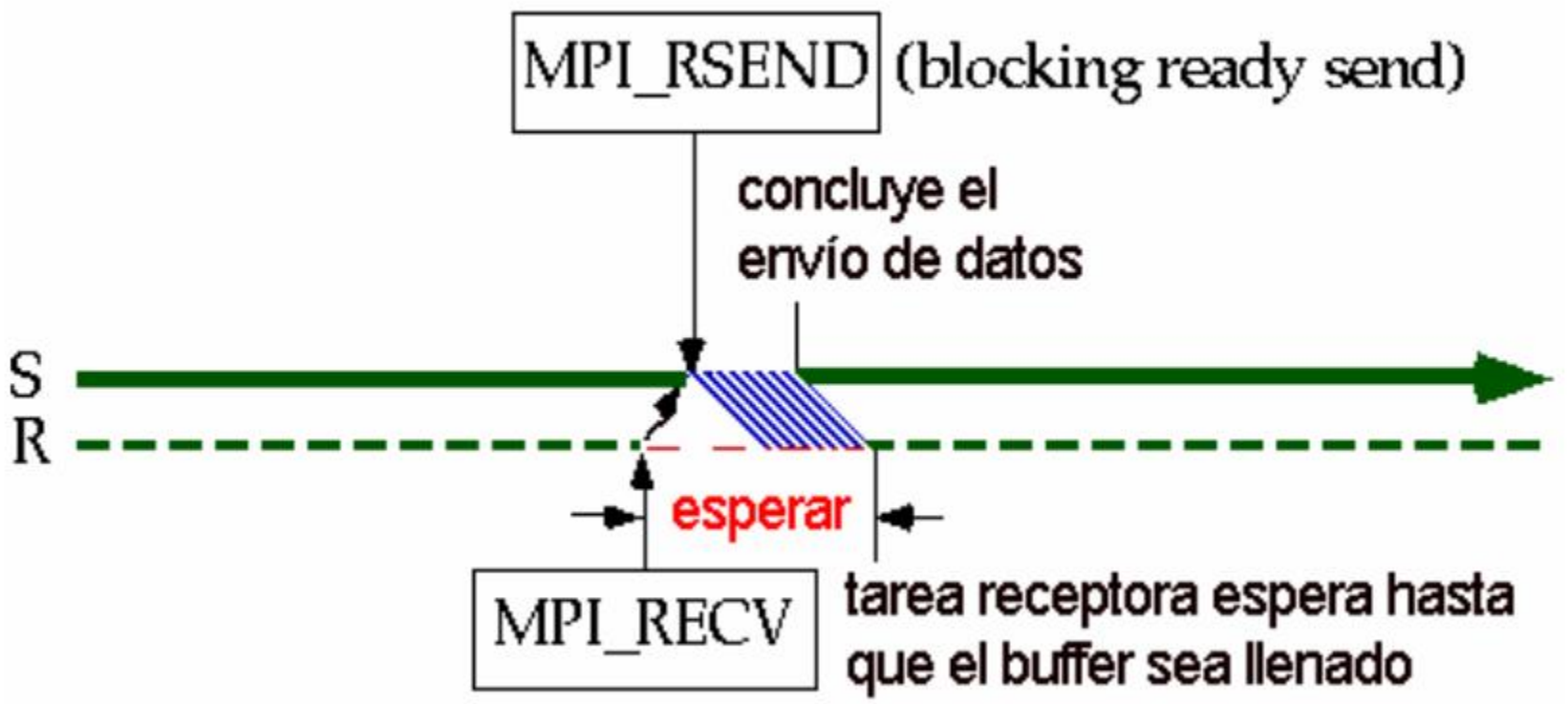
2. Modo bufereado



2. Modo Síncrono



2. Modo Ready


MPI_RSEND (blocking ready send)
concluye el
envío de datos
S
R
esperar
MPI_RECV
tarea receptora espera hasta
que el buffer sea llenado

2. Comunicación Bloqueante

- Ejemplo 1:

Versión segura aunque no haya buffer

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &yo);
if (yo == 0) {
  MPI_Send(buf, count, type, 1, tag, comm);
  MPI_Recv(buf, count, type, 1, tag, comm,
&status);     }
else if (yo == 1) {
  MPI_Recv(buf, count, type, 0, tag, comm,
&status);
  MPI_Send(buf, count, type, 0, tag, comm);
}
...
```

2. Comunicación Bloqueante

- Ejemplo 2:

Versión con bloqueo

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &yo);
if (yo == 0) {
  MPI_Recv(buf, count, type, 1, tag, comm,
&status);
  MPI_Send(buf, count, type, 1, tag, comm);
}
else if (yo ==1) {
  MPI_Recv(buf, count, type, 0, tag, comm,
&status);
  MPI_Send(buf, count, type, 0, tag, comm);
} …
```

2. Comunicación Bloqueante

- Ejemplo 3:

Versión no segura: funciona si hay buffer suficiente

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &yo);
if (yo == 0) {
  MPI_Send(buf, count, type, 1, tag, comm);
  MPI_Recv(buf, count, type, 1, tag, comm,
&status);
}
else if (yo ==1) {
  MPI_Send(buf, count, type, 0, tag, comm);
  MPI_Recv(buf, count, type, 0, tag, comm,
&status);
}   ...
```

2. Comunicación Bloqueante

- Existe una llamada para hacer una pareja de operaciones envío/recepción sin bloquearse (de forma segura):
  - `int` **`MPI_Sendrecv`**`( void* buf, int count, MPI_Datatype datatype, int dest, int tag, void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);`
  - `int` **`MPI_Sendrecv_replace`** `...`

2. Manejo de errores

- Cuando se trabaja con paso de mensajes pueden ocurrir varios tipos de errores:
  - Error de transmisión. Este no nos debe preocupar ya que MPI se encarga de él y nos garantiza su no existencia.
  - Error de programa. Si utilizamos mal las llamadas disponibles, por ejemplo, por argumentos o fallos de punteros.
  - Error de recursos. Si al hacer la llamada el sistema no nos puede proporcionar lo que estamos pidiendo, por ejemplo, por demasiados mensajes pendientes o falta de buffers del sistema.
- Una buena implementación de MPI manejará todas las excepciones, si algunas no están cubiertas, se tratarán por el *run time system - ORTE*.

2. Manejo de errores

- Cuando ocurre un error, un manejador de errores de MPI será invocado, y éste puede hacer dos cosas:
  - Parar todos los programas.
  - Enviar un código de error a través de la llamada.
- Hay que tener en cuenta que después de un error, el estado de MPI no está definido y puede que el usuario no pueda continuar su programa, y si lo puede hacer, deberá tomar medidas para recoger mensajes de error y salir del programa.
- Existen dos llamadas para terminar el programa, la conocida MPI_Finalize y abort que intentará terminar todas las tareas de un comunicador
  - `int MPI_Abort(MPI_Comm comm, int errorcode);`

2. Manejo de errores

- En MPI, como ocurre en algunas lenguajes como ADA o en POSIX con las señales, se pueden asociar manejadores (en un comunicador) a un error de forma local a un proceso. Existen de todas maneras dos determinados:
  - MPI_ERRORS_ARE_FATAL. Cuando es llamado actúa igual que si hubiéramos llamado a MPI_ABORT.
  - MPI_ERRORS_RETURN. No tiene efecto y devuelve el código del error.
- Normalmente no se utilizan manejadores, simplemente se toma el error de las llamadas y se averigua a que error corresponde con:
  - int MPI_Error_string(int errorcode, char *string, int *resultlen);
- Los errores dependen de la implementación pero para estandarizarlos MPI ha creado una serie de clases de error.

2. Debugging

- Cuando depuramos un programa secuencial, normalmente utilizamos un debugger (tipo gdb).
- En programas paralelos esto no es tan sencillo (se puede hacer en algunas implementaciones pero los resultados no son tan buenos), de hecho es uno de los tópicos de investigación.
- Para depurar los programas se puede recurrir al método clásico de print/flush, pero cuando hay muchos procesos esto puede dar más problemas que resolverlos.

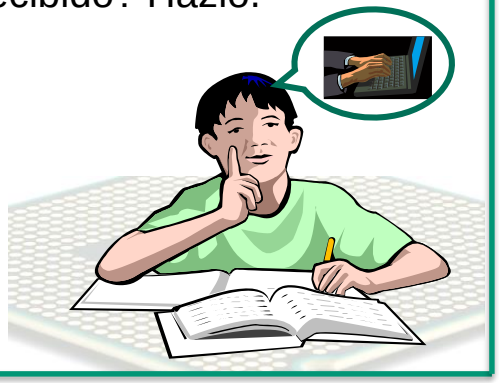
2. Debugging

- Quizás la mejor manera de evitar los errores es conocer los fallos clásicos que se suelen cometer:
  - Todos los fallos secuenciales (variable son inicializadas, direcciones erróneas, tamaños insuficientes...).
  - Intentar recibir datos cuando todavía no han sido enviados, producirá un deadlock.
  - Equivocarse en los parámetros de la llamada, bien en el proceso receptor o emisor.
  - Tenemos que saber y no sorprendernos que ante un fallo, diferentes sistemas responden de distintas maneras.
  - Los programas deberían funcionar de igual manera independientemente del tamaño de los datos, el número de procesos o del tiempo de simulación.

Práctica 1

- Práctica 1: Envío / Recepción de un mensaje. Modo básico. Ojo al buffer.
  A. Haz un programa en que varios procesos manden al raíz diferentes mensajes etiquetados cada uno de manera diferente e indica el orden de llegada.
  B. Realiza un [illegible]guro (como el ejemplo 1) de dos procesos en los que se ha[illegible] de un dato por ambos). El envío bloqueant[illegible]á un array de enteros de longitud defini[illegible]
  C. Copia y cambia las líneas de or[illegible] (ejemplo 2).
  D. Copia y cambia el programa (como el ejemplo 3) [illegible]nciona. Si funciona di porqué. Ejecuta varias veces el programa para distintos [illegible]maños de mensaje hasta que se bloquee. Di por qué ocurre.
  E. ¿Cómo conseguirías la longitud del mensaje en bytes recibido? Hazlo.

Ver ejercicio 1



Práctica 1

- Práctica 2: Envío / Recepción de un mensaje. Otros modos de envío. El mundo no es tan perfecto.
  - A. Usando la práctica 1.B, copia y modifica el programa y usa un buffer para el sistema de 1 byte y comunica el array anterior con las llamadas bufereadas ¿qué ocurre o debiera ocurrir? Prueba diferentes valores de longitud de mensaje.
  - B. Copia y cambia otra vez el programa y haz q… so haga dos envíos y el otro dos recepciones con distintas… se envían 1 y 2 y se recibe 2 y 1. ¿Cómo de… ara que el orden de los mensajes se… or qué?
  - C. Lee … prueba si tu implementación lo cumple.

Ver ejercicio 2

Práctica 1

- Práctica 3: El programa clásico “master/slave”: Como no toda la memoria de un problema tiene que estar en todos los procesadores.
  - Realiza un programa MPI con P procesos, en que todos deben colaborar para realizar la suma de los N primeros números naturales (P debe ser menor que N y divisor). A cada proceso le tocaran N/P datos
    A. Se creará un array de N enteros, y [illegible]s procesos sumarán su parte y se la devolverán [illegible] todos los datos los sumará y los pr[illegible]ser.

$$\frac{[illegible]1)}{2}$$

    B. Haz[illegible]ersión en que sólo se necesite un subarray de N/P. La inicialización se producirá en cada proceso.
    C. ¿Se puede hacer sin array?
    D. Pasa el número N por argumento al programa.

Ver ejercicio 3