Tema 2: Sistema Multicomputador: paso de mensajes en arquitectura con memoria distribuida

Multicomputador de Memoria Distribuida

Asignatura: Arquitectura de Sistemas Paralelos

Profesor: Francisco Javier Gómez Arribas
Departamento de Tecnología Electrónica y de las Comunicaciones

UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

Escuela Politécnica Superior

Tema 2: Contenidos

- **Arquitectura multicomputador/multiprocesador**
  - **Clasificación por acceso a memoria.**
  - **Modelo de comunicaciones**
  - **Eficiencia del sistema al paralelizar una aplicación**
- **Programación de multicomputadores por paso de mensajes**
  - **Paradigma Maestro/esclavo.**
  - **Modos de comunicación en MPI.**
    - Síncrona vs asíncrona.
    - Bloqueante vs no bloqueante
    - Buffer de comunicaciones
  - **Comunicadores en MPI**
    - Comunicaciones colectivas
  - **Planificación de tareas: estática, dinámica bajo demanda.**

Programación mediante Paso de Mensajes

Paso de Mensajes: Una interfaz de programación para aplicaciones paralelas con una comunicación eficiente.

- Evitando la copia de memoria a memoria y
- permitiendo la superposición de computación y comunicación

Que permita implementaciones en un ambiente heterogéneo.

- Comunicación mediante envío y recepción de mensajes
- Sincronización mediante comunicación bloqueante
- Gran distancia entre modelo de programación y la arquitectura

⋆Evolución de los Modelos de Programación:

	Paralelismo de datos →	SPMD →	Paso de mensajes
Sincronización	Cada instrucción	Datos compartidos	Comunicaciones bloqueantes
Código / procesador	Igual	Igual	Diferente

Paradigma de paso de mensajes

El paradigma es aplicables a casi todos los tipos de arquitecturas paralelas (con memoria distribuida / compartida) y contempla un conjunto de procesos que interactúan por paso de mensajes.

- **Implica la programación de una aplicación paralela con P procesos con diferentes espacios de memoria.**
  - **Todas la variables son privadas para cada proceso.**
  - **Cada proceso puede ejecutar código distinto y sobre diferentes datos (MIMD) o (SPMD)**

La comunicación tiene lugar por intercambio de mensajes.

- **El intercambio de mensajes se realiza via funciones de librería**
- **Diseño independiente de hardware o lenguajes de programación.**
- **El intercambio de mensajes es cooperativo:**
  - **Los datos deben ser enviados y recibidos explícitamente por funciones de librería [send() y receive()] que ocultan la complejidad del proceso de comunicación al programador.**

Paradigmas de Organización de Tareas

Paradigmas principales : Se ejecutan P copias iguales independientes. Las tareas se diferencian mediante el pid del proceso :

1) Maestro-esclavo (asimétrico)

- **Un proceso dirige (reparte el trabajo) y los demás hacen el trabajo**

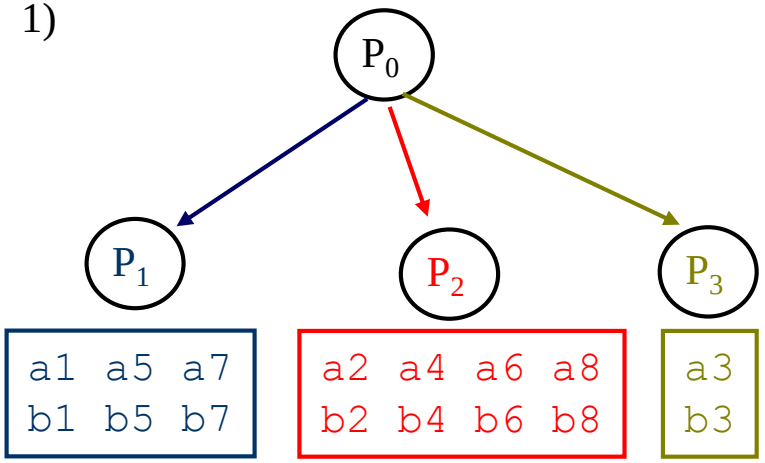
2) SPMD Simétrico (reparto de datos)

- **Todos los procesos hacen básicamente el mismo trabajo**

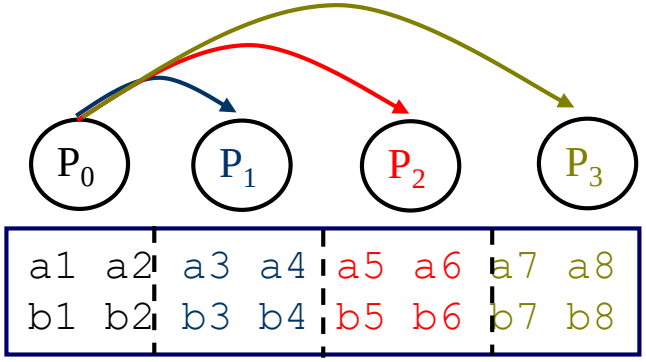
Ejemplo: *Suma de los elementos de dos vectores*

a1	a2	a3	a4	a5	a6	a7	a8
b1	b2	b3	b4	b5	b6	b7	b8

1)



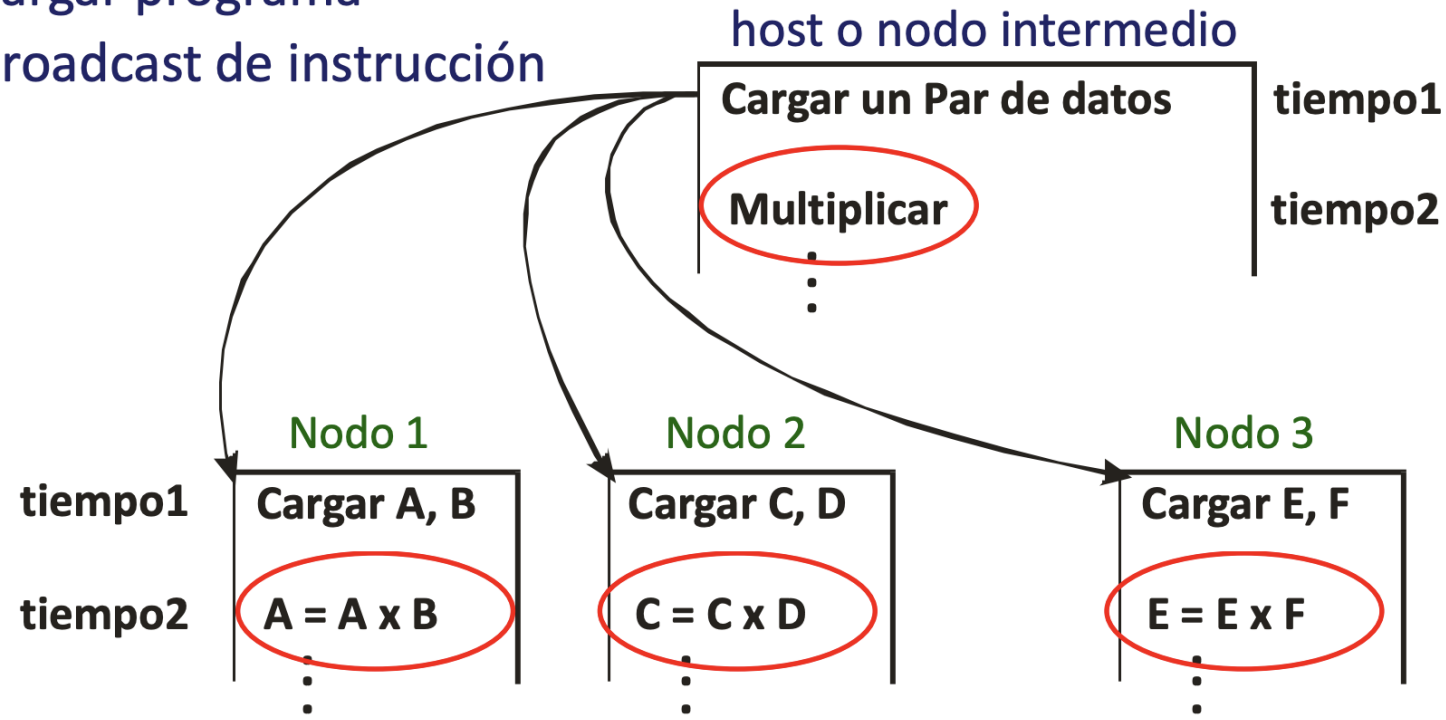
2)



Paradigmas de Organización de Tareas

MODELO DE PROGRAMA SIMD

1. Cargar programa
2. Broadcast de instrucción



los nodos reciben las instrucciones y las ejecutan

Paradigmas de Organización de Tareas

MODELO DE PROGRAMA SPMD

Es el modelo que utilizaremos en MPI !!

MODELO DE PROGRAMA SPMD

Un mismo programa se ejecuta sobre diferentes conjuntos de datos

Nodo 1

```
Conseguir datos
if ... positivo
→Hacer algo
if ... negativo
   Hacer otra cosa
if ... es cero
   Hacer una tercer
   cosa
```

Nodo 2

```
Conseguir datos
if ... positivo
   Hacer algo
if ... negativo
→Hacer otra cosa
if ... es cero
   Hacer una tercer
   cosa
```

Nodo 3

```
Conseguir datos
if ... positivo
   Hacer algo
if ... negativo
   Hacer otra cosa
if ... es cero
→Hacer una tercer
   cosa
```

Todos los nodos ejecutan el mismo programa, pero no necesariamente las mismas instrucciones

MODELO DE PROGRAMA SPMD

- Aplicable en la resolución de problemas homogéneos
  - Estructura geométrica regular, interacciones limitadas
  - La homogeneidad permite la distribución uniforme de datos
  - Las comunicaciones y sincronizaciones son reducidas y su costo es proporcional al tamaño de las fronteras entre los subdominios de datos
  - El patrón de comunicaciones es usualmente estructurado y predecible
- Si el problema es no-homogéneo, se requieren mecanismos adicionales para la distribución de datos y el balance de cargas
- El paradigma es altamente sensible al fallo en algún proceso
  - Puede causar un deadlock, o incluso una caída del sistema
  - Para resolverlo, deben aplicarse técnicas de tolerancia a fallos
- Es el modelo adoptado por la biblioteca MPI (Message Passing Interface)

Modelos de programación con paso de mensajes

SIMD/SPMD vs MIMD/MPMD.

- **Single Program Multiple Data: Realiza la misma tarea sobre diferentes datos (paralelismo de datos)**
- **Se suele diferenciar una parte que realiza repartos de datos con el paradigma MAESTRO/ESCLAVO y por tanto estrictamente es MPMD**

```
main () {
    if (process is to become master) {
        distribute data among slaves
        organise communication / synchronisation
    } else {
        compute something
        exchange data with other processes
    }
}
```

Modelos de programación con paso de mensajes

MIMD/MPMD.

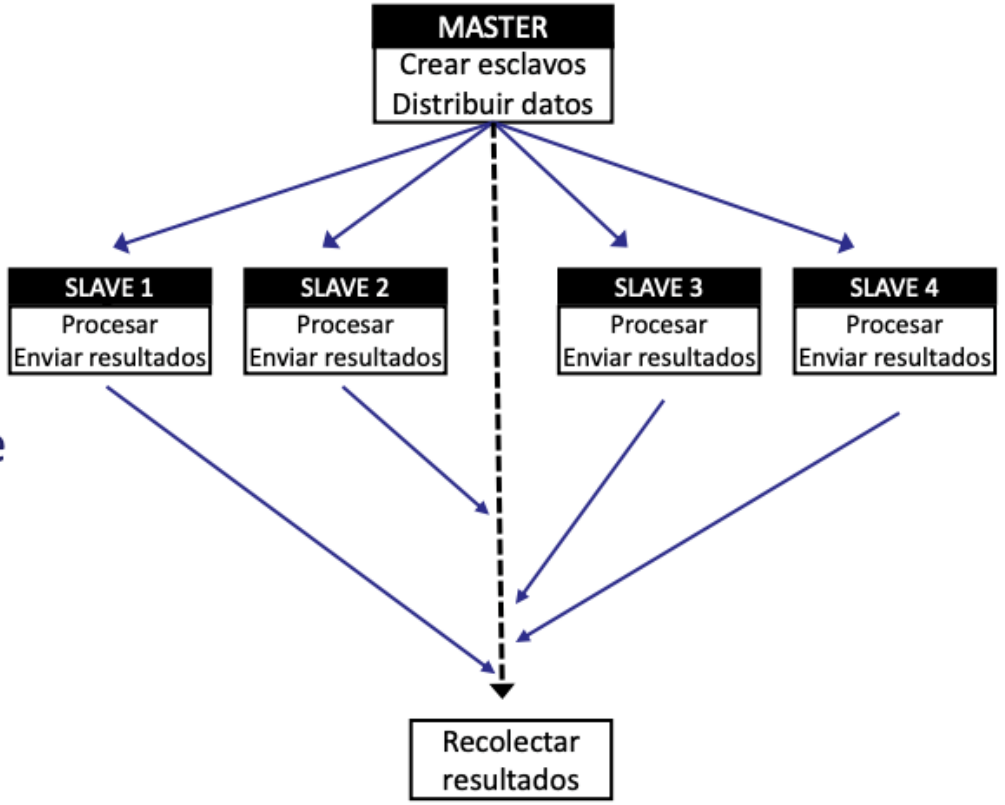
- **Multiple Program Multiple Data: Realiza diferentes tareas sobre diferentes datos (paralelismo de datos y paralelismo funcional)**
- **En el paradigma MAESTRO/ESCLAVO se puede diferenciar la tarea a realizar por cada esclavo**

```
main () {
    if (processID == 0) {
        compute something
    } else if (processID == 1) {
        compute something different
    } else if (processID == 2) { ... }
    :
}
```

Modelos de programación con paso de mensajes

En el paradigma MAESTRO/ESCLAVO se puede diferenciar variantes

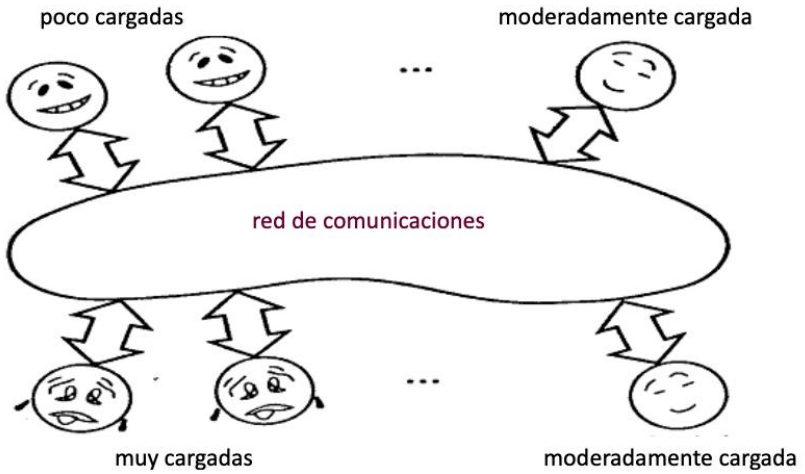
- Sincrónico o asincrónico
  - Sincronismo entre esclavos
- Activo o pasivo
  - El maestro procesa o no procesa
- Estático o dinámico
  - Mecanismo de asignación de datos por parte del master



Modelos de programación con paso de mensajes

MAESTRO/ESCLAVO: Tecnicas de Balanceo de carga

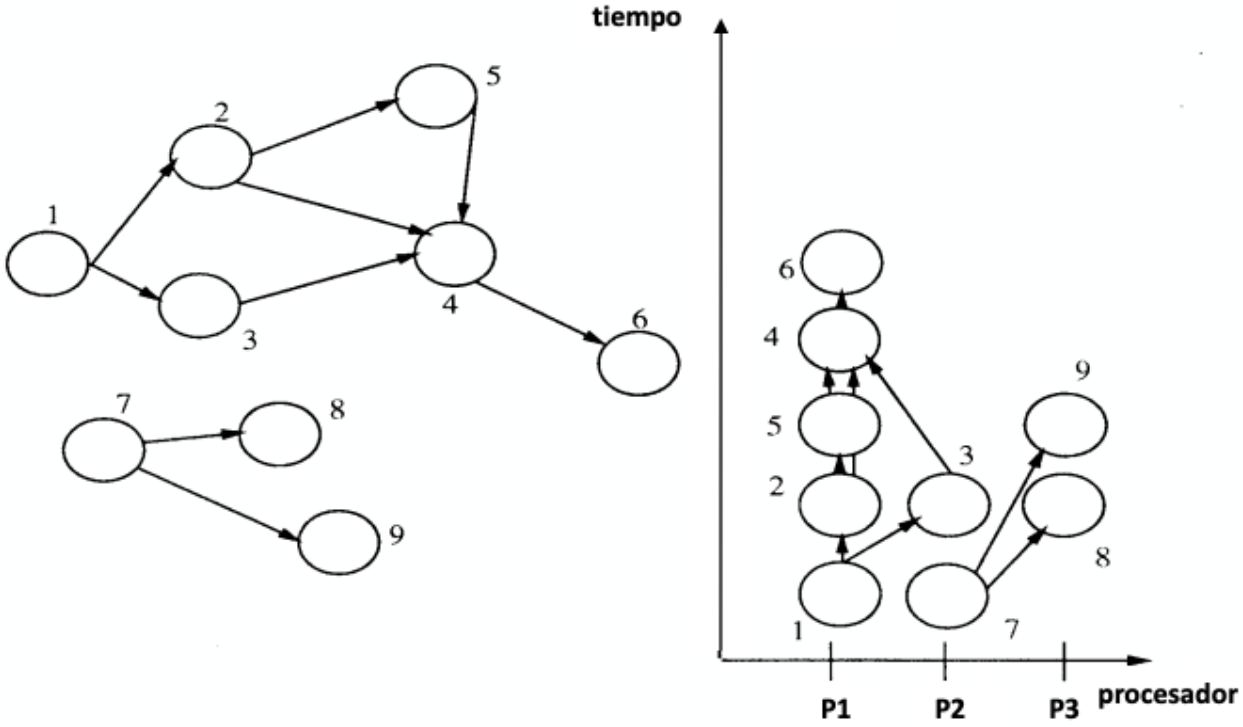
- También conocidas como "técnicas de despacho"
- Clasificación:
  - Técnicas estáticas (planificación)
  - Técnicas dinámicas (al momento de
  - Técnicas adaptativas



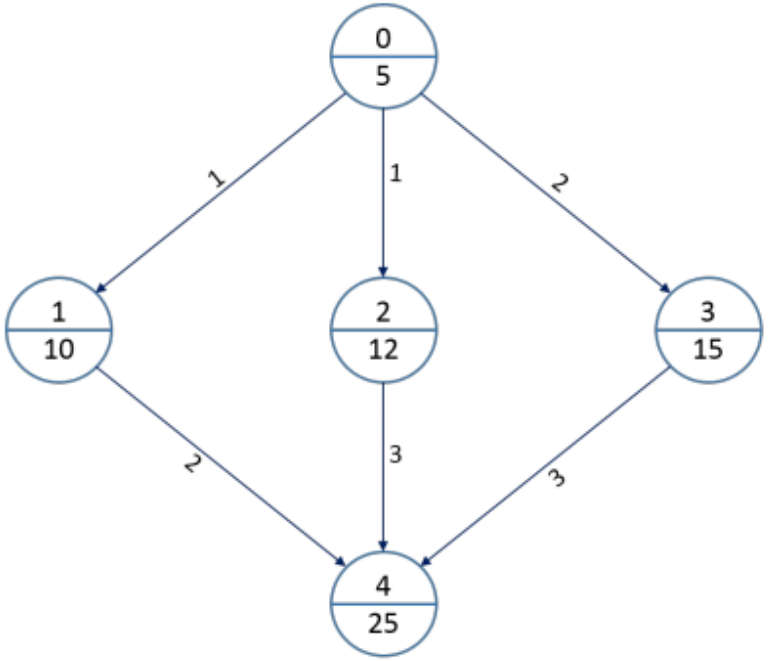
- Principales criterios utilizados:
  - Mantener los procesadores ocupados la mayor parte del tiempo
  - Minimizar las comunicaciones entre procesos

Modelos de programación con paso de mensajes

Planificación: El algoritmo de planificación relaciona el algoritmo (grafo de tareas) con el hardware disponible (procesador, tiempo)



Planificación de algoritmos en arquitecturas:



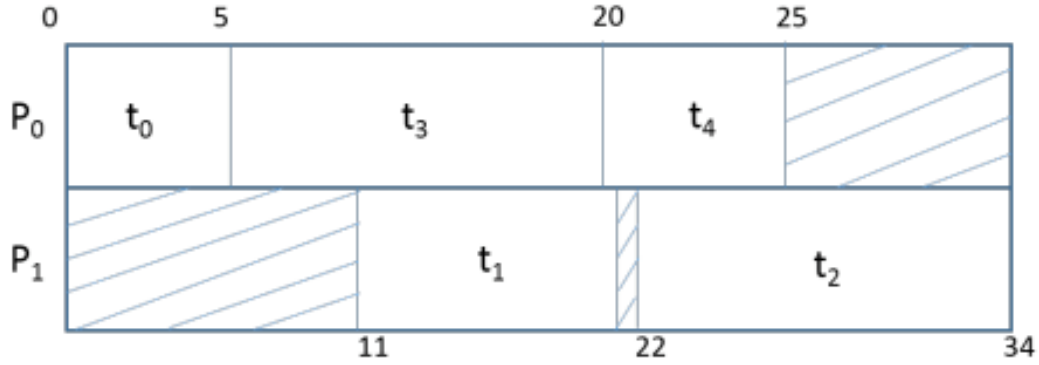


Fig. 11.3 A task graph with five tasks represented as nodes showing task numbers and task execution times (for example, milliseconds), and directed edges, indicating the order of execution of tasks, labeled with communication costs.

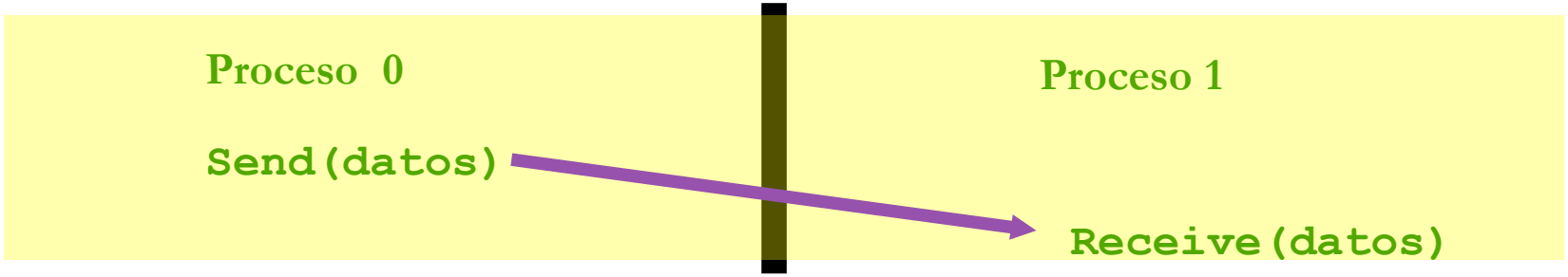
Conceptos genéricos de MPI

MPI (message-passing interface) es el "estándar" actual de **programación de los sistemas de memoria distribuida**, mediante paso de mensajes.

- MPI es, básicamente, **una librería (grande) de funciones de comunicación** para el envío y recepción de mensajes entre procesos.
- MPI indica explicitamente la comunicación entre procesos, es decir:
  - -- **los movimientos de datos**
  - -- **la sincronización**

Movimiento de datos por paso de mensajes

Para enviar y recibir mensajes hay que dar los detalles de la comunicación :

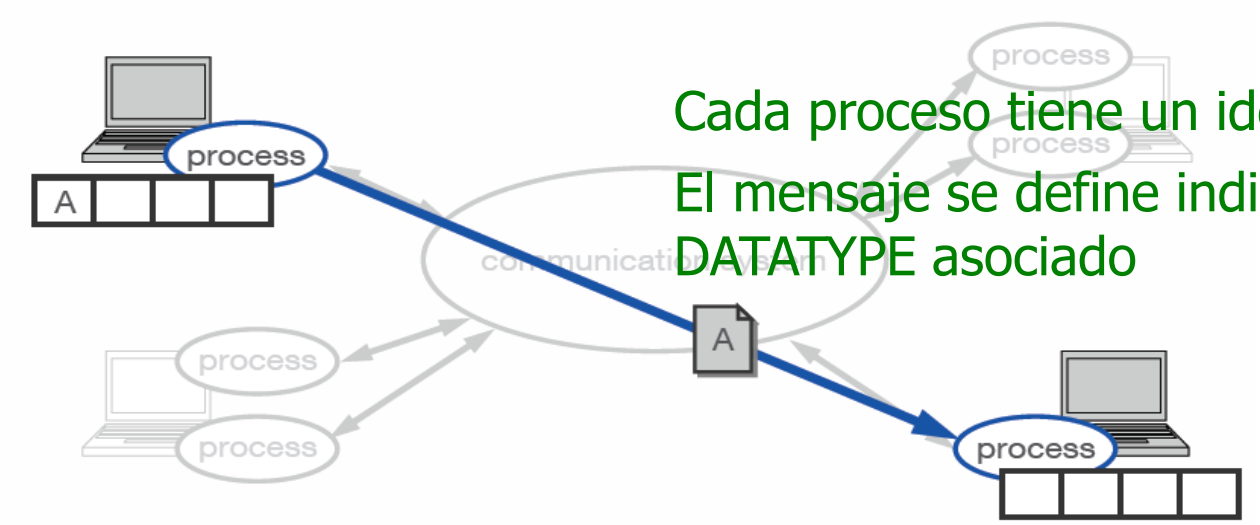


Se necesita especificar:

- **Cómo se describen los datos.**
- **Cómo se identifican los procesos.**
- **Cómo el receptor reconoce/descarta los mensajes.**
- **Cuando se completa la comunicación**

Comunicación punto a punto

Las funciones básicas de envío y recepción de mensajes punto a punto: MPI_Send y MPI_Recv



Cada proceso tiene un identificador o **`pid`**.

El mensaje se define indicando su tamaño y su DATATYPE asociado

- Los mensajes se envían con una etiqueta o tag (un entero definido por el usuario), para permitir al proceso receptor identificar el mensaje..
- Los mensajes pueden filtrarse en la parte de recepción especificando un TAG determinado o no filtrarse especificando como etiqueta **`MPI_ANY_TAG`**

Comunicación punto a punto

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

Protocolo de comunicación

Considere el siguiente fragmento de código:

P0	P1
a = 100;	receive(&a, 0, 1)
send(&a, 1, 1);	printf("%d\n", a);
a = 0;	

La semántica de la función send requiere que el valor recibido por el proceso P1 sea 100 y no el valor 0.

¿ Podría suceder que el proceso P1 recibiese el valor 0?

La respuesta es que SI ES POSIBLE y esto justifica la definición de los diferentes modos de comunicación para el funcionamiento de send y receive.

Modos de comunicación

MPI define cuatro modos de comunicación

- **Modo Synchronous (“más seguro")**
  - La comunicación no se produce hasta que emisor y receptor se ponen de acuerdo. Es bloqueante
- **Modo Ready (“menor overhead”)**
  - El receptor debe estar preparado antes de empezar el envío.
- **Modo Buffered (“desacoplo de emisor y receptor”)**
  - El emisor deja el mensaje en un búfer y retorna.
  - La comunicación se produce cuando el receptor está dispuesto a ello.
  - El búfer no se puede reutilizar hasta que se vacíe.
- **Modo Standard (“compromiso”)**

El modo de comunicación se selecciona con la rutina de envío (Send).

Modos de comunicación

MPI permite comunicaciones bloqueantes y no bloqueantes

- **Bloqueante: se para el programa hasta que es seguro usar el buffer del mensaje.**
- **No-bloqueante: no se garantiza el contenido de buffer, pero permite separar la comunicación de la computación.**

Se refiere al uso del buffer

- **Bloqueante y no bloqueante se refieren al CONTROL y a la seguridad de utilización del buffer de la aplicación**
- **No debe confundirse con el concepto de comunicaciones SINCRÓNAS y ASINCRÓNAS**

Funciones MPI para los diferentes modos de comunicación

Envío	Bloqueante	No Bloqueante
Estándar	MPI_Send	MPI_Isend
Buffer	MPI_Bsend	MPI_Ibsend
Sincrono	MPI_Ssend	MPI_Issend
Listo	MPI_Rsend	MPI_Irsend

- Recepción:
  - Estándar
  - Puede ser bloqueante o no bloqueante
  - Recibe desde cualquiera de las 8 operaciones de envío

Recepción	Bloqueante	No Bloqueante
Estándar	MPI_Recv	MPI_Irecv

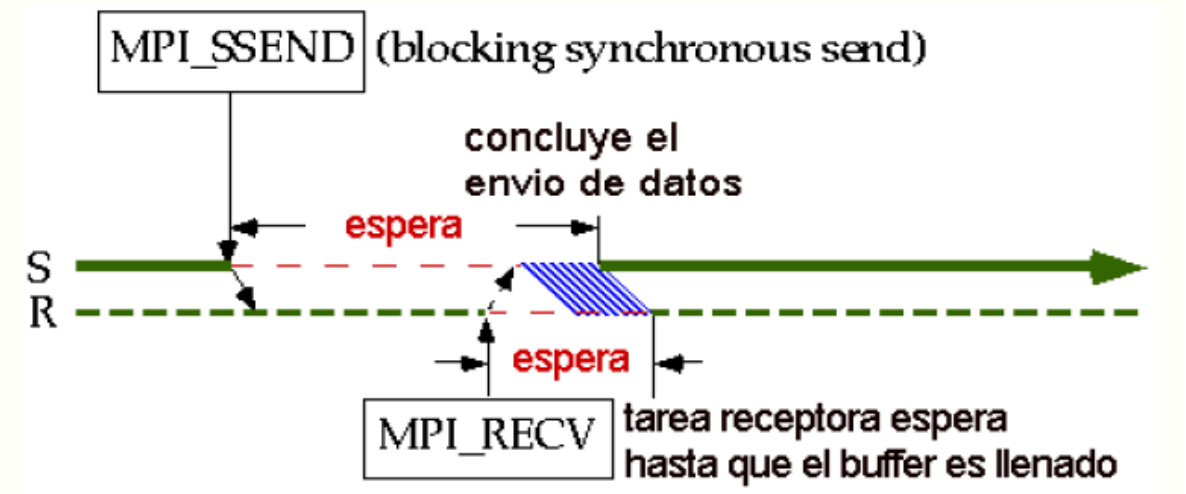
Características de los modos de comunicación

Modo	Ventajas	Desventajas
Synchronous	Más seguro y por lo tanto más portable. Orden SEND/RECV es crítico. Tamaño de buffer irrelevante.	Puede incurrir en un overhead de sincronización importante
Ready	Overhead más bajo. No requiere negociación SEND/RECV	RECV debe preceder al SEND
Buffered	Desacopla el SEND del RECV No overhead en SEND. Orden SEND/RECV irrelevante. El programador puede controlar tamaño del buffer	Overhead adicional del sistema debido a la copia al buffer
Standard	Bueno para la mayoría de casos	Funcionamiento dependiente del sistema

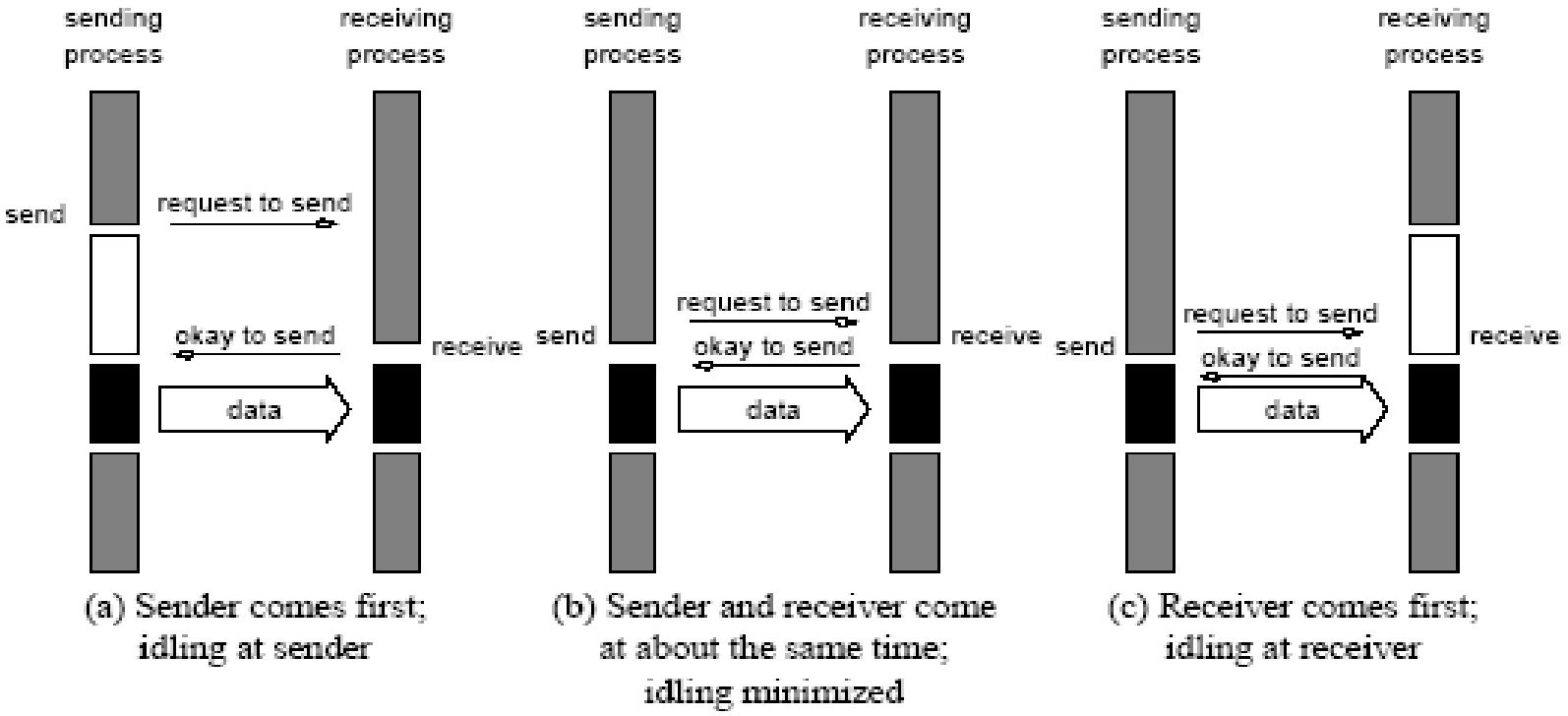
Comunicación Síncrona

El remitente (el que envía) se bloquea hasta que el destinatario comience a recibir el mensaje. La Secuencia es la siguiente:

1. **El que envía : ejecuta *send: se bloquea.***
2. **El receptor: cuando ejecuta *receive se envía* mensaje interno de confirmación al que envía: *listo para recibir datos y se bloquea.***
3. **Se transfieren los datos.**
4. **Se desbloquean ambos.**

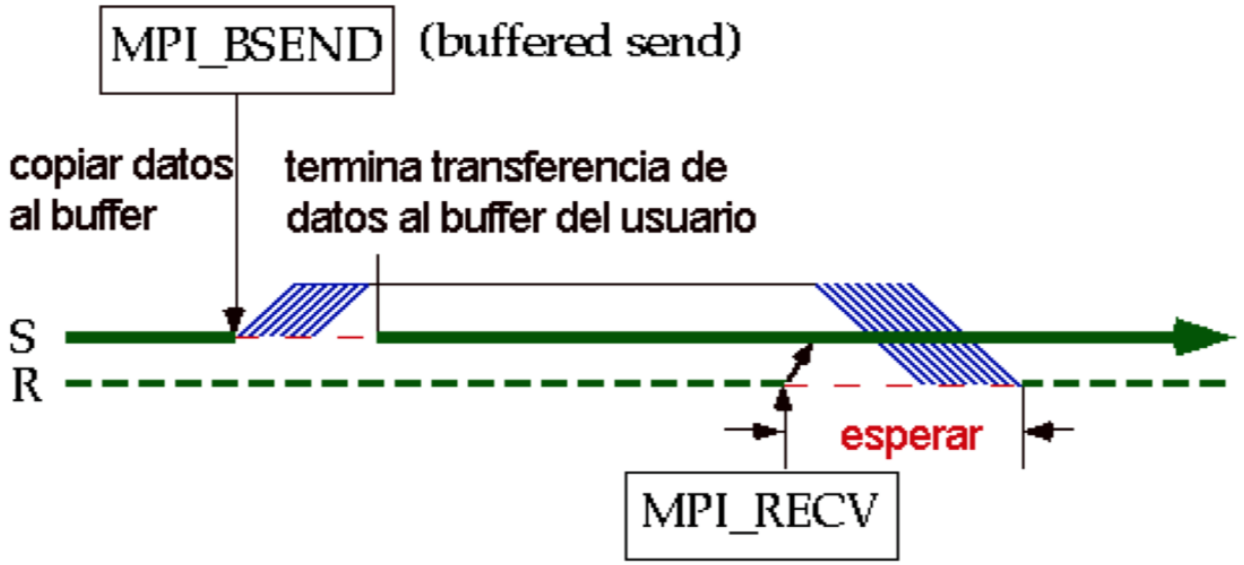


Comunicación bloqueante (sin Buffer)



Handshake for a blocking non-buffered send/receive operation.
It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

Comunicación con Buffer



•Blocking Buffered Send MPI_Bsend(...)

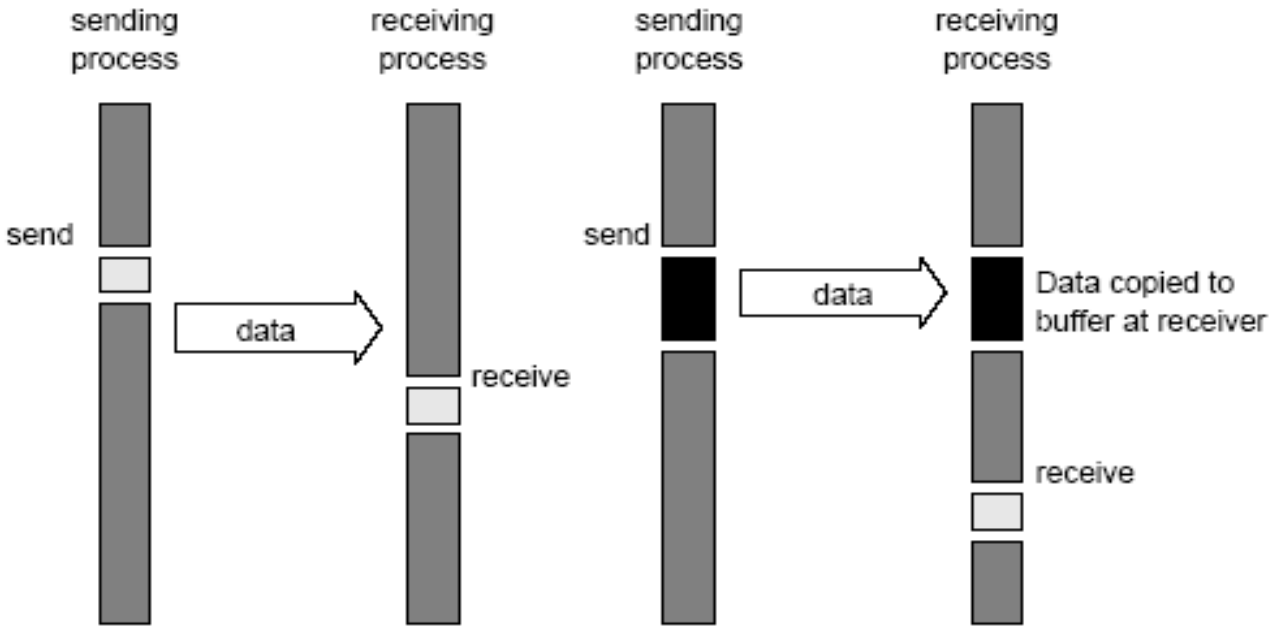
– Copia los datos desde el buffer de mensajes a un buffer dado por el usuario y luego retorna.

- Los datos serán copiados desde el buffer dado por el usuario a la red cuando se reciba una notificación indicando "estoy listo para recibir"

Comunicación con Buffer

- 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

Comunicación con Buffer



Blocking buffered transfer protocols:

(a) in the presence of communication hardware with buffers at send and receive ends; and

(b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Limitaciones de la comunicación bloqueante con buffer

El tamaño "limitado" de los buffers pueden tener un impacto muy importante en el rendimiento y en overheads difíciles de controlar.

P0

```
for (i = 0; i < 1000; i++){
    produce_data(&a);
     send(&a, 1, 1);
 }
```

P1

```
for (i = 0; i < 1000; i++){
receive(&a, 0, 1);
consume_data(&a);
}
```

¿Qué sucederá si el consumidor es mucho mas lento que el productor?

Deadlocks en comunicación bloqueante Buffer

Incluso con Buffer de envió los bloqueos (deadlocks) se pueden seguir produciendo debido a que los receive siguen siendo bloqueantes.

```
P0                          P1

receive(&a, 1, 1);          receive(&a, 0, 1);

send(&b, 1, 1);             send(&b, 0, 1);
```

Bloqueos (Deadlocks)

Con MPI_Send bloqueante, pueden producir deadlock.

- Usando non-blocking operations se pueden evitar la mayor parte de los deadlocks.

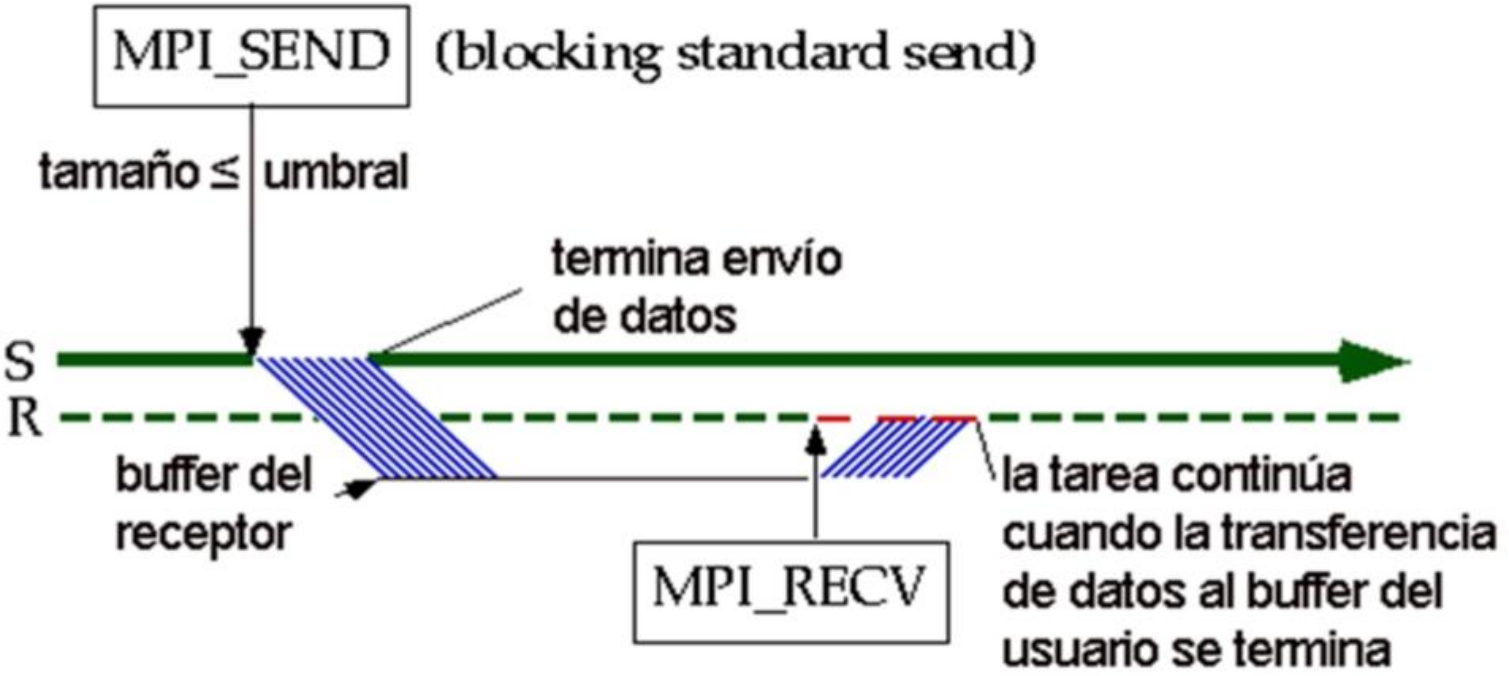
```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
   MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
}
...
```

Replacing either the send or the receive operations with non-blocking counterparts fixes this deadlock.

Comunicación Estándar Bloqueante

Caso a)

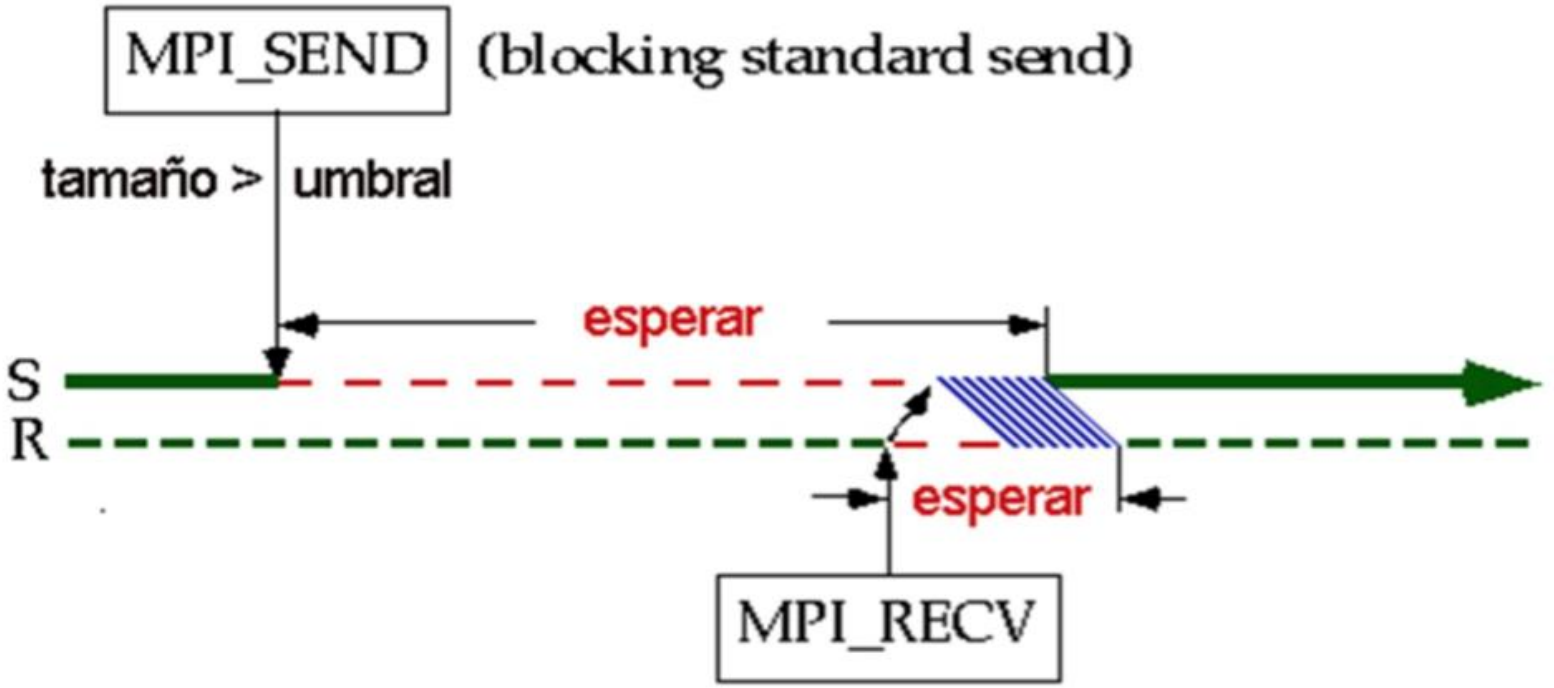
tamaño del mensaje ≤ umbral: mensaje se almacena en buffer del receptor



Comunicación Estándar Bloqueante

Caso b)

tamaño del mensaje > umbral: mensaje NO se almacena en buffer del receptor



¿Qué modo de comunicación es mejor?

Cada estrategia tiene sus ventajas e inconvenientes:

- **Síncrona**: es **más rápida** si el receptor está dispuesto a recibir; nos ahorramos la copia en el buffer.
  - El intercambio de datos, sirve para sincronizar los procesos.
  - Al ser bloqueante es posible un **deadlock**!

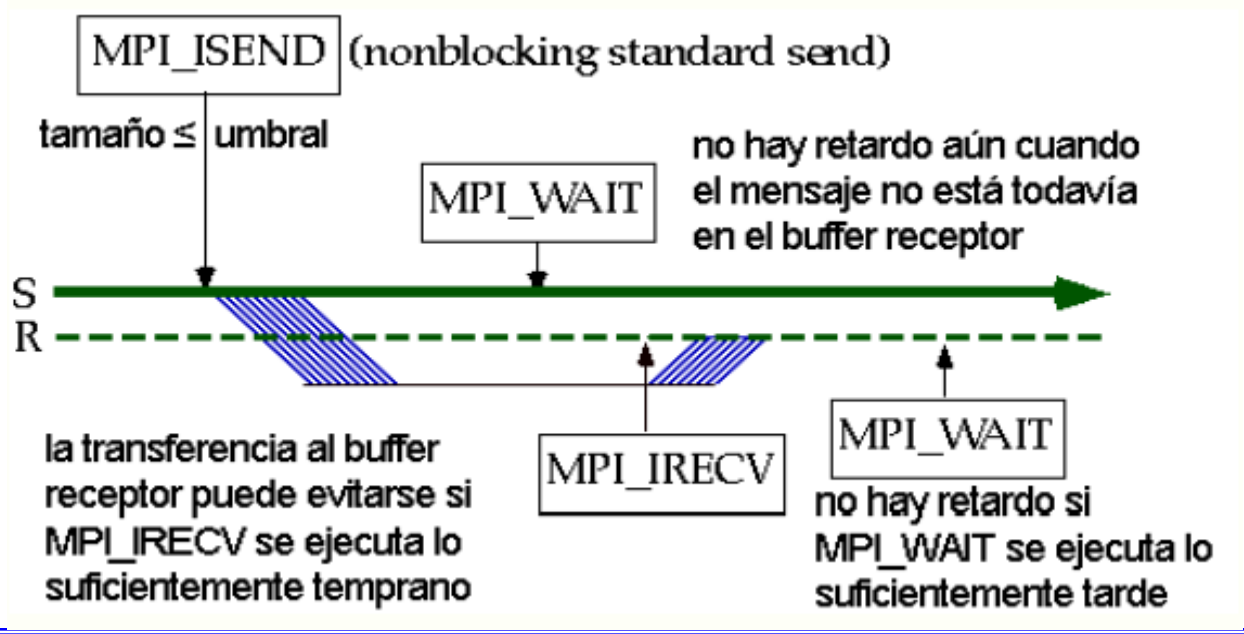
- **Buffered**: el emisor **no se bloquea** aunque el receptor no esté disponible (el receptor si se bloquea)
  - Hay que hacer **copia(s)** del mensaje (es más lento).

Comunicación Asíncrona

Las llamadas **asíncronas son no bloqueantes**

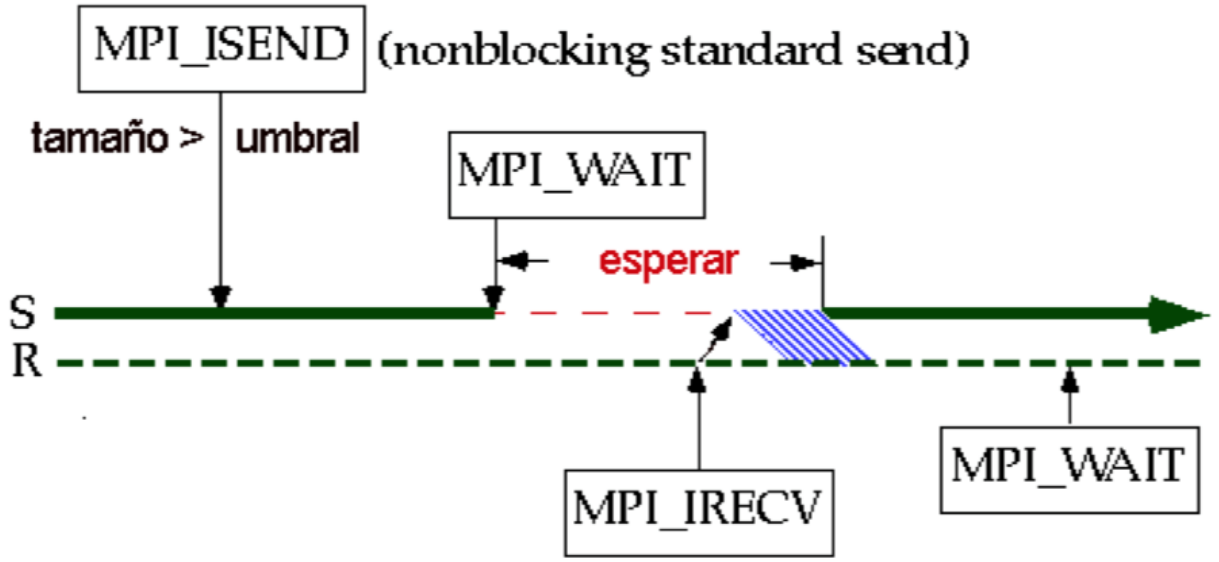
- **El programador no puede presuponer que los datos hayan sido enviados o copiados en el buffer ni si han llegado al destinatario.**
- **Antes de utilizar el buffer de mensajes, el programador debe verificar su estado.**

Primitivas en MPI: MPI_ISEND/IRECV, MPI_WAIT, MPI_TEST



Comunicación Asíncrona

tamaño del mensaje > umbral: con solape de comunicaciçpn y computación.



Comunicaciones no-bloqueantes

Se separa la comunicación en tres fases:

- **Inicio de la comunicación no bloqueante**
- **Hacer trabajo adicional:**
  - **Quizás incluyendo otras comunicaciones**
- **Espera a que finalice la comunicación no bloqueante.**
  - **Se puede esperar *wait* o comprobar con *test* la finalización:**
    - `MPI_Wait(&request, &status)`
    - `MPI_Test(&request, &flag, &status)`
  - **MPI_PROBE , MPI_IPROBE : Admiten comprobar los mensajes que llegan sin recibirlos**

    Nota: MPI_PROBE es bloqueante. Espera hasta que haya algo que probar.
  - **MPI_CANCEL: Cancela comunicaciones pendientes (Ultimo recurso para limpiar las comunicaciones)**
    - `MPI_Cancel(&request)`

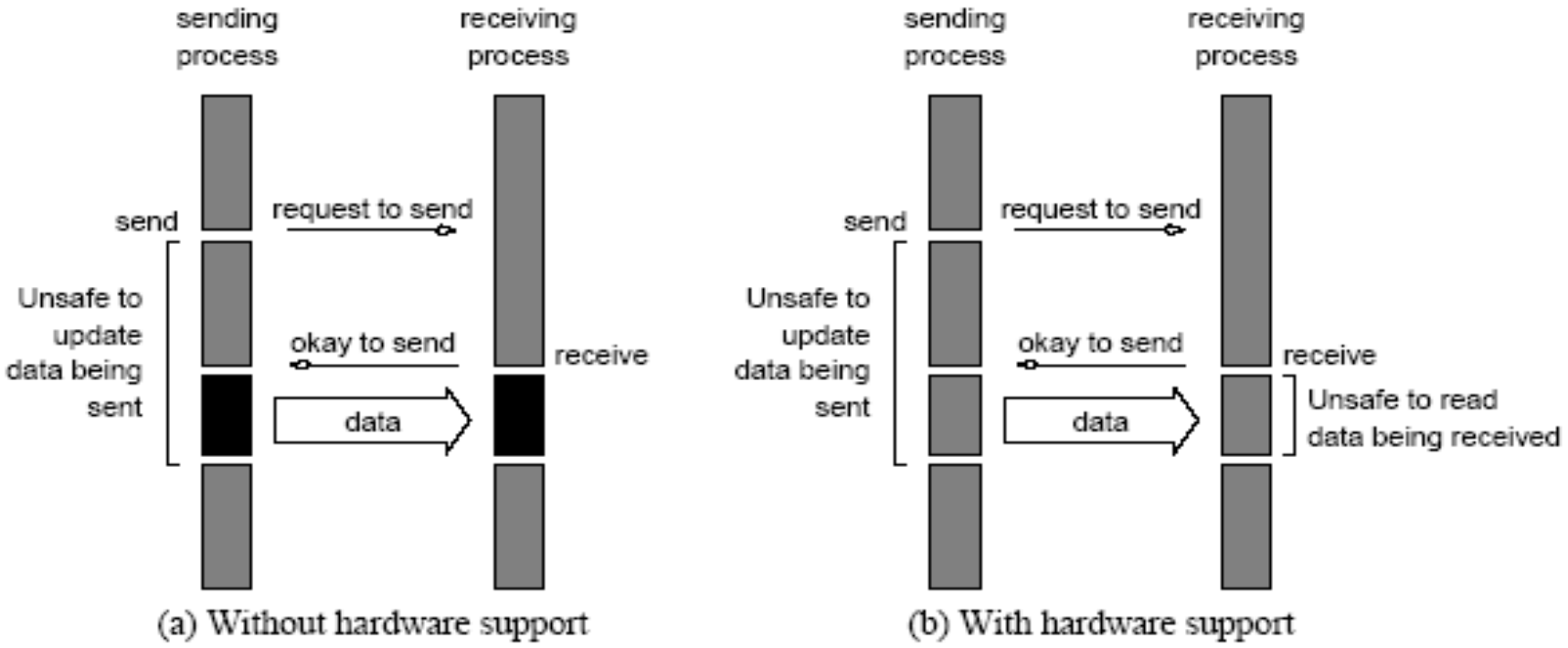
Solapa de Comunicación y computación

```
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 */
```

- **Los modos no bloqueantes permiten solapar procesamiento y comunicaciones**
  - **MPI_WAIT indica si una operación (identificada por un request) se completó**
  - **Retorna información sobre la operación completada en el parámetro status**

- **Wait espera la finalización:**
  - **`MPI_Wait(&request, &status)`**

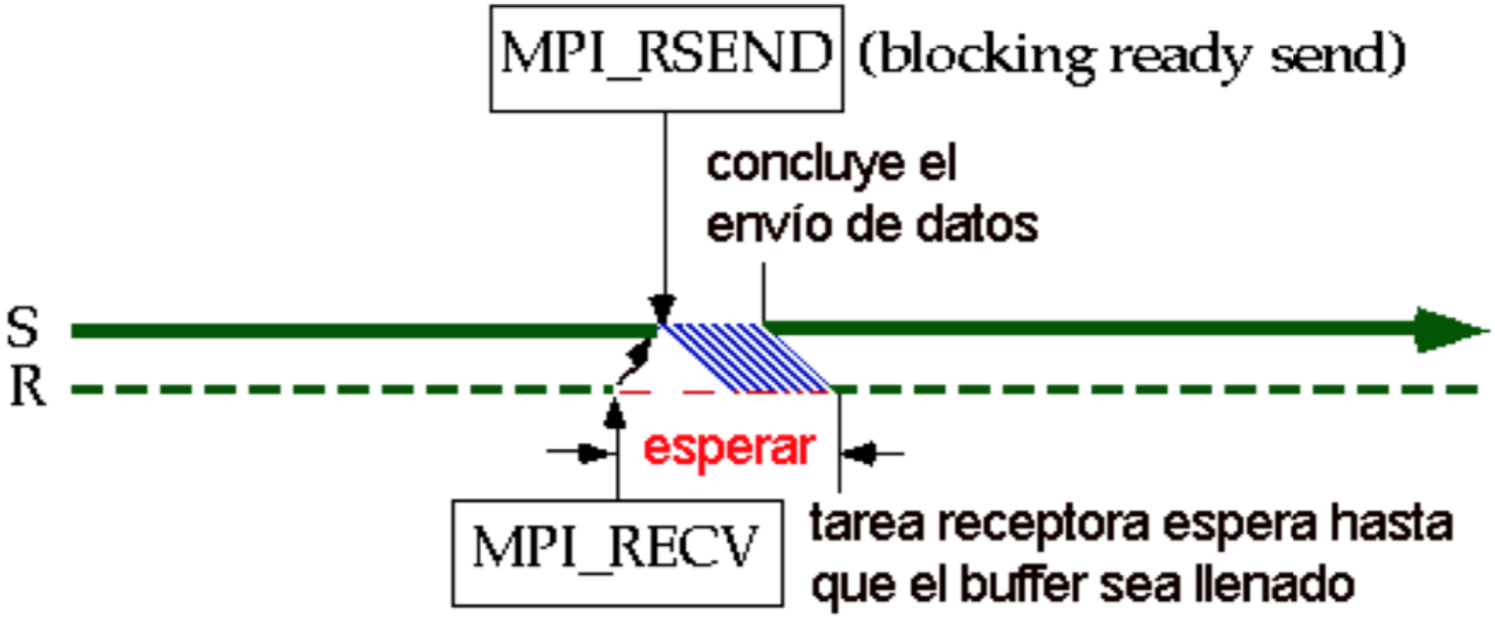
Comunicación no Bloqueante sin buffers



Non-blocking non-buffered send and receive operations
(a) In absence of communication hardware;
(b) in presence of communication hardware.

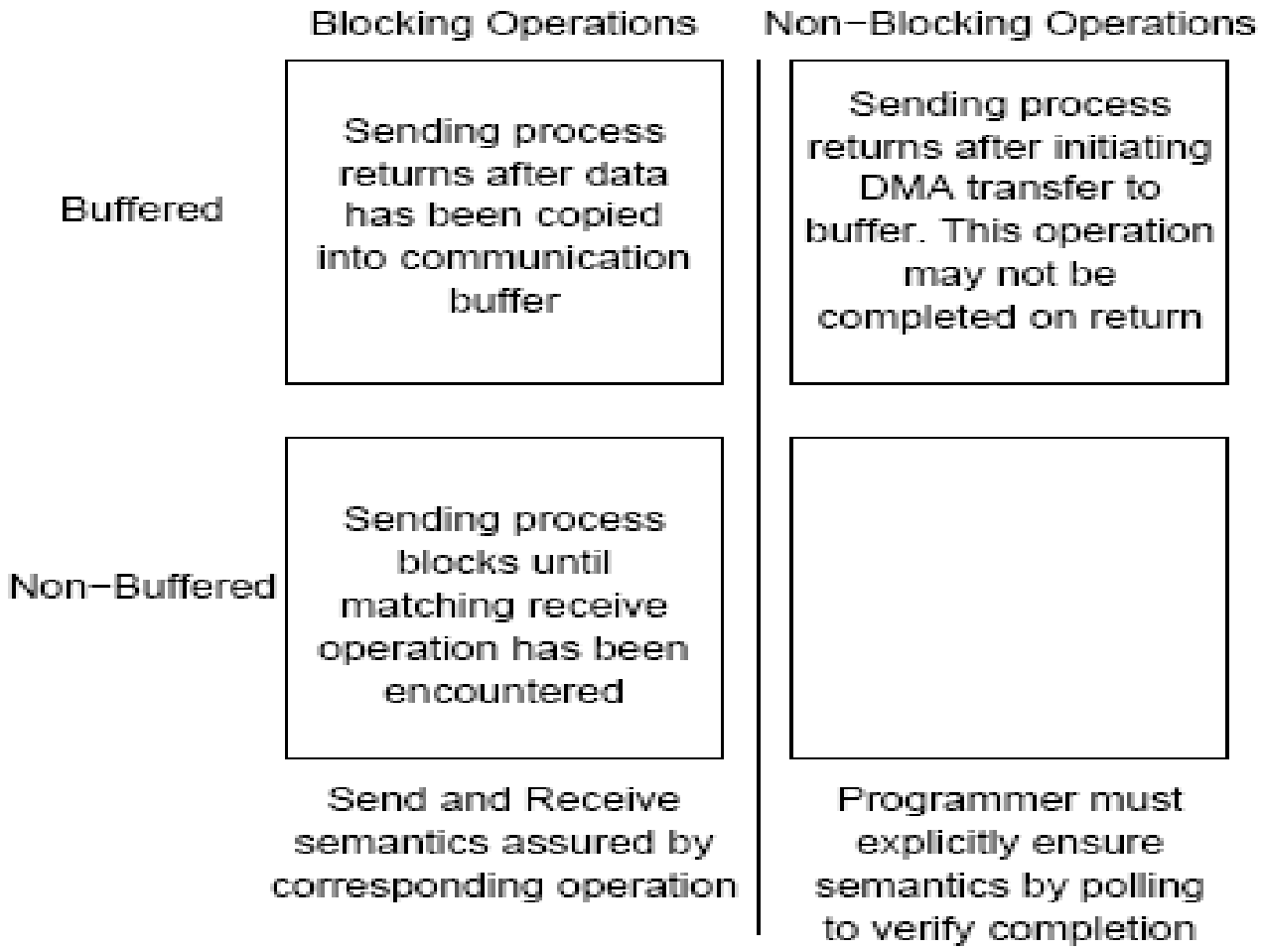
Comunicación: Modo Ready

Evita copias porque NO se almacena en buffer del sistema



Posibilidades para Send y Receive

Espacio de opciones en el funcionamiento de la comunicación



Paso de mensajes con opciones de sincronización y bloqueo

Sincronización:

- **El proceso que envía especifica el modo síncrono.**

En Bloqueos:

- **Ambos procesos esperan hasta que la transición ha terminado.**

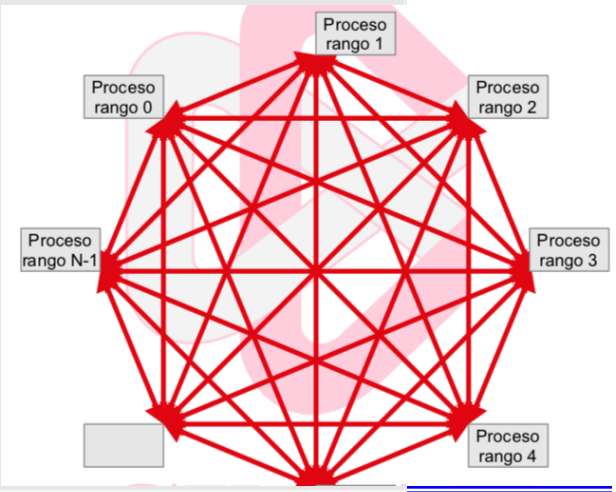
Además la comunicación necesita para su buen funcionamiento:

- **El proceso que envía debe especificar un rank destino válido.**
- **El receptor debe especificar un rank fuente válido.**
- **El comunicador debe ser el mismo.**
- **Las etiquetas (Tags) deben coincidir.**
- **El tipo de mensajes debe coincidir.**
- **El buffer del receptor debe ser suficientemente grande.**

MPI

Ejemplo: comunicación donde todos los nodos envian mensajes a todos los nodos.

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

#define TAG_MESSAGE 1

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

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int* send_data = new int[size];
  int* recv_data = new int[size];

  MPI_Request* send_request = new  MPI_Request[size];
  MPI_Request* recv_request =new MPI_Request[size];

  for (int r = 0; r < size; ++r)
  {
    if (r != rank)
    {
      send_data[r] = rank*1000 + r;
      MPI_Isend(&send_data[r], 1, MPI_INT, r, TAG_MESSAGE,
      → MPI_COMM_WORLD, &send_request[r]);
      MPI_Irecv(&recv_data[r], 1, MPI_INT, r, TAG_MESSAGE,
      → MPI_COMM_WORLD, &recv_request[r]);
    }
    else
    {
      send_request[r] = MPI_REQUEST_NULL;
      recv_request[r] = MPI_REQUEST_NULL;
    }
  }

  int received = 0;
  do
  {
    int r;
    MPI_Waitany(size, recv_request, &r, MPI_STATUS_IGNORE);
    printf("%i recibio mensaje '%i' desde %i\n", rank, recv_data[r], r);
    ++received;
  } while (received < size - 1);

  delete [] recv_request;
  delete [] send_request;
  delete [] recv_data;
  delete [] send_data;

  MPI_Finalize();
  return 0;
}
```



Comunicaciones colectivas

- Muchas aplicaciones requieren de operaciones de comunicación en las que participan muchos procesos.

  La comunicación es colectiva si participan en ella todos los procesos del comunicador.

Ejemplo: un broadcast, envío de datos desde un proceso a todos los demás.

- ¿Enviando mensajes Uno a uno en un bucle?

Parádigma de comunicación para la distribución de un dato

En un sistema paralelo, con p procesadores que no soporta acceso concurrente (EREW PRAM), se puede hacer p copias de un dato usando la técnica de copiado recursivo (recursive doubling).

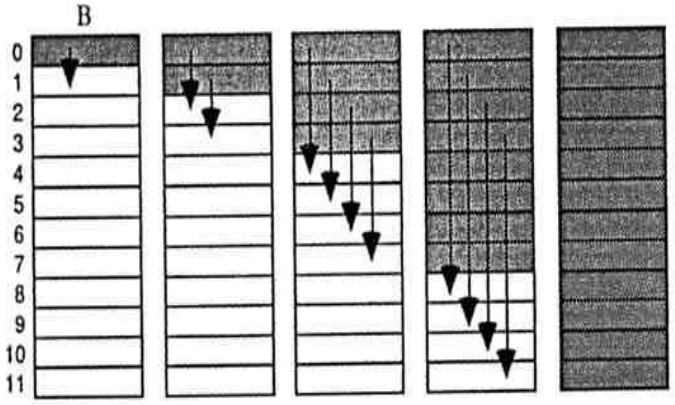
Making p copies of $B[0]$ by recursive doubling

for $k = 0$ to $\lceil \log_2 p \rceil - 1$ Processor j, $0 \leq j < p$, do
  Copy $B[j]$ into $B[j + 2^k]$
endfor

EREW PRAM broadcasting sin redundancia

EREW PRAM algorithm for broadcasting by Processor i

Processor i write the data value into $B[0]$
$s := 1$
while $s < p$ Processor j, $0 \leq j < \min(s, p - s)$, do
  Copy $B[j]$ into $B[j + s]$
  $s := 2s$
endwhile
Processor j, $0 \leq j < p$, read the data value in $B[j]$



Operaciones colectivas

Comunicación colectiva: envío de un mensaje de uno a muchos.

- **Típicamente un master envía los mismos datos a sus esclavos.**
- **Por ejemplo, en la paralelización del entrenamiento de una red neuronal enviamos a todos los esclavos los nuevos pesos al final de cada época de entrenamiento.**

Operaciones de reducción colectivas: se realiza una operación matemática distribuida y se devuelve el resultado al root de la operación

- **Es típico en un esquema master-slave.**
- **Se utiliza por ejemplo en la suma de los errores de todos los esclavos.**
- **Operaciones definidas:**
  - **Aritméticas: suma, multiplicación…**
  - **Lógicas: AND, OR…**

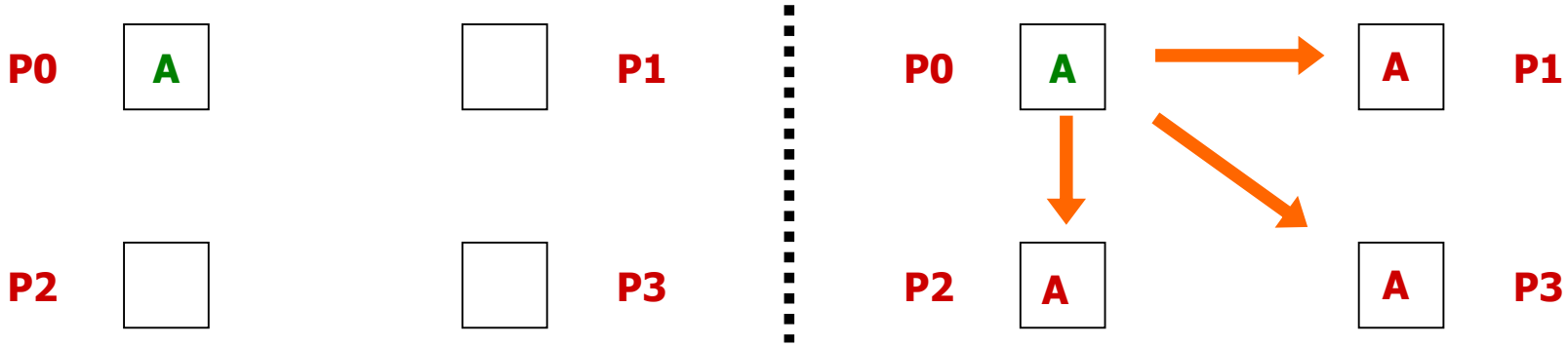
Comunicaciones colectivas

- **TODAS** las funciones de comunicación colectiva **son bloqueantes.**
  - **Todos los procesos que forman parte del comunicador deben ejecutar la función.**
  - **Pueden construirse su funcionalidad a partir de las funciones básicas de comunicación punto a punto.**
  - **Facilitan la legibilidad y mantenimiento del código al reemplazar múltiples llamadas de send/receive.**
  - **La implementación puede optimizar la efectividad de la comunicación**

- Tres tipos
  1. **Movimiento de datos**
  2. **Operaciones en grupo**
  3. **Sincronización**

Com. Colectivas: movimiento de datos

1a Broadcast: envío de datos desde un proceso (`root`) a todos los demás.



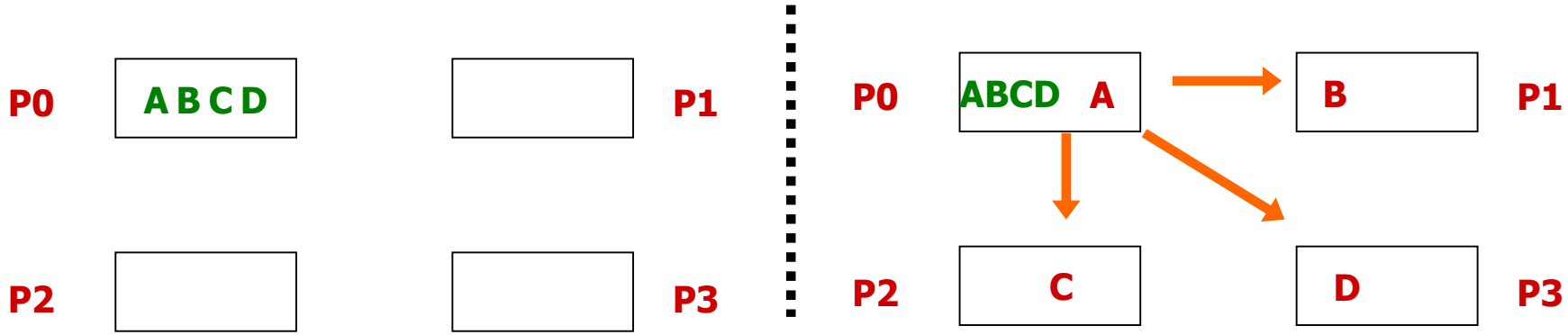

```
> MPI_Bcast(&mess, count, type, root, comm);
```

- (implementación logarítmica en árbol)

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

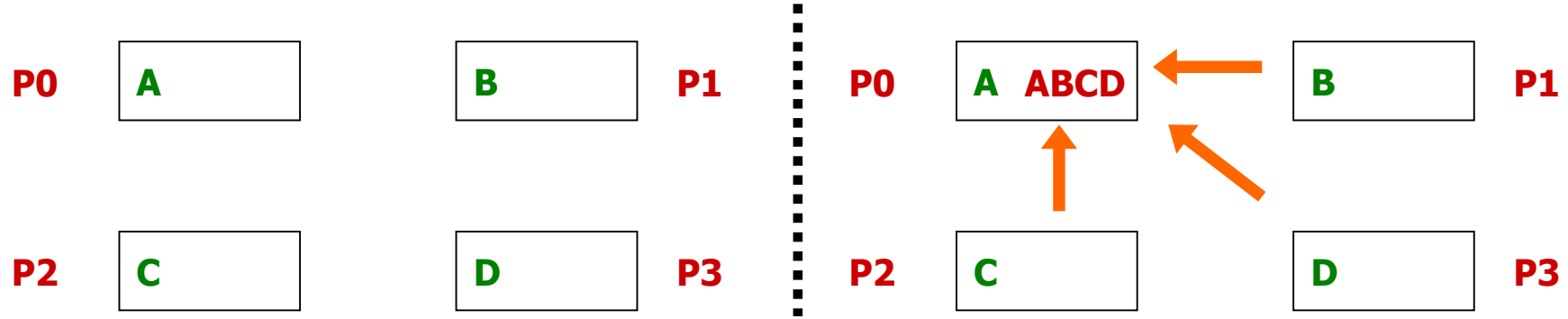
Com. Colectivas: movimiento de datos

1b Scatter: reparto de datos desde un proceso al resto




```
int MPI_Scatter ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void
  *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm )
```

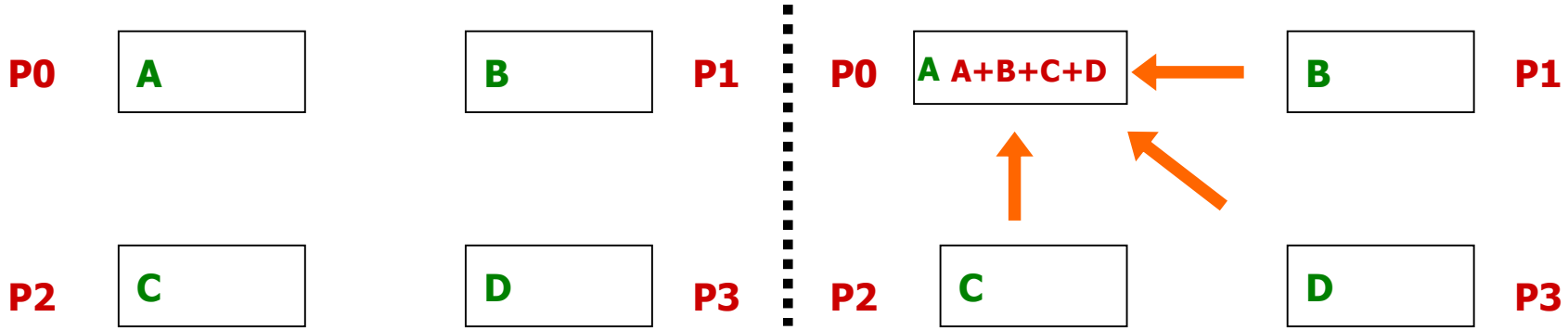
1c Gather: **recolección** de datos de todos los procesos




```
int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void
  *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm )
```

Com. Colectivas: movimiento de datos

2. Reduce (en árbol)




```
int MPI_Reduce ( void *sendbuf, void *recvbuf, int count,  MPI_Datatype datatype, MPI_Op op,
   int root, MPI_Comm comm )
```

Allreduce: todos obtienen el resultado (Reduce + Broadcast)

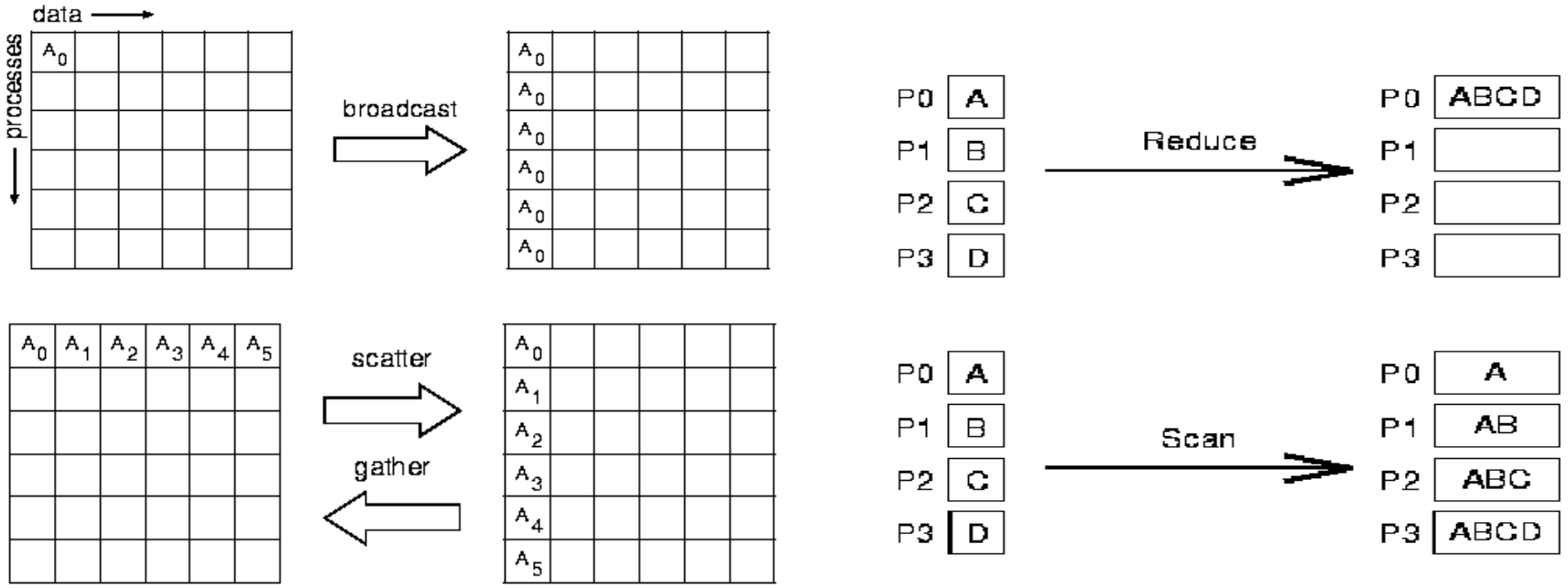
Comunicaciones Colectivas: Reduce

Operaciones predefinidas para MPI_Reduce:

- **MPI_MAX máximo**
- **MPI_MIN mínimo**
- **MPI_SUM suma**
- **MPI_PROD producto**
- **MPI_LAND "and" lógico**
- **MPI_LOR "or" lógico**
- **MPI_LXOR "xor" lógico**
- **MPI_BAND "bitwise and"**
- **MPI_BOR "bitwise or"**
- **MPI_BXOR "bitwise xor"**
- **MPI_MAXLOC máximo e índice del máximo**
- **MPI_MINLOC mínimo e índice del mínimo**

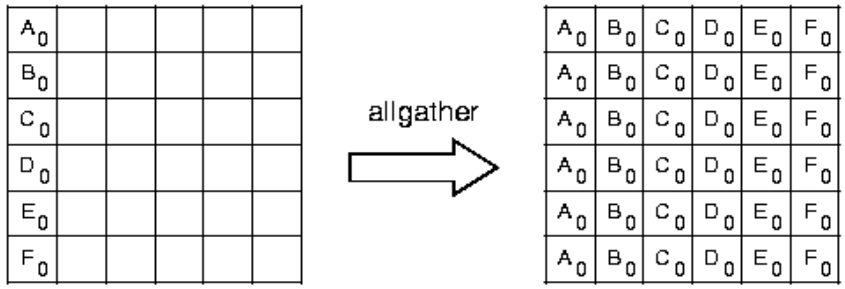
Además existe un mecanismo para que el usuario cree sus propias funciones para el "Reduce".

Comunicaciones colectivas: Reduce vs Scan

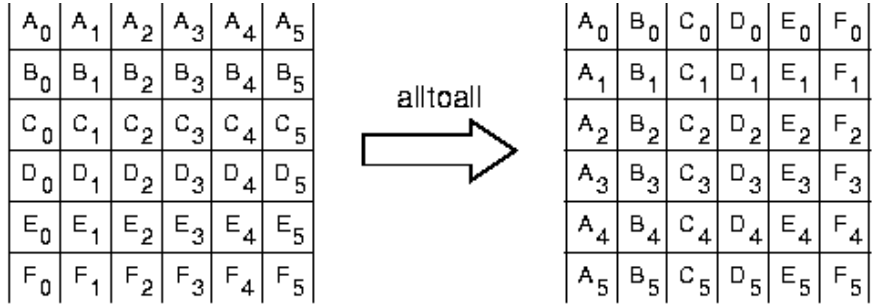



```
int MPI_Scan ( void *sendbuf, void *recvbuf, int count,  MPI_Datatype
   datatype, MPI_Op op, MPI_Comm comm )

int MPI_Reduce ( void *sendbuf, void *recvbuf, int count,  MPI_Datatype datatype, MPI_Op op,
   int root, MPI_Comm comm )
```

Comunicaciones colectivas




```
int MPI_Allgather(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount,
  MPI_Datatype rdtype,                                    MPI_Comm comm)
```




```
int MPI_Alltoall( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int
  recvcnt, MPI_Datatype recvtype, MPI_Comm comm )
```

Comunicaciones colectivas

> Ejercicio: V(i) = V(i) * Σ V(j)

```
sum = 0;
for (j=0; j<N; i++) sum = sum + V[j];
for (i=0; i<N; i++) V[i] = V[i] * sum;
```

1. Leer N (el pid = 0)
2. **Broadcast** de N/npr (tamaño del vector local)
3. **Scatter** del vector V (trozo correspondiente)
4. Cálculo local de la suma parcial
5. **Allreduce** de sumas parciales (todos obtienen suma total)
6. Cálculo local de V(i) * sum
7. **Gather** de resultados
8. Imprimir resultados (el pid = 0)

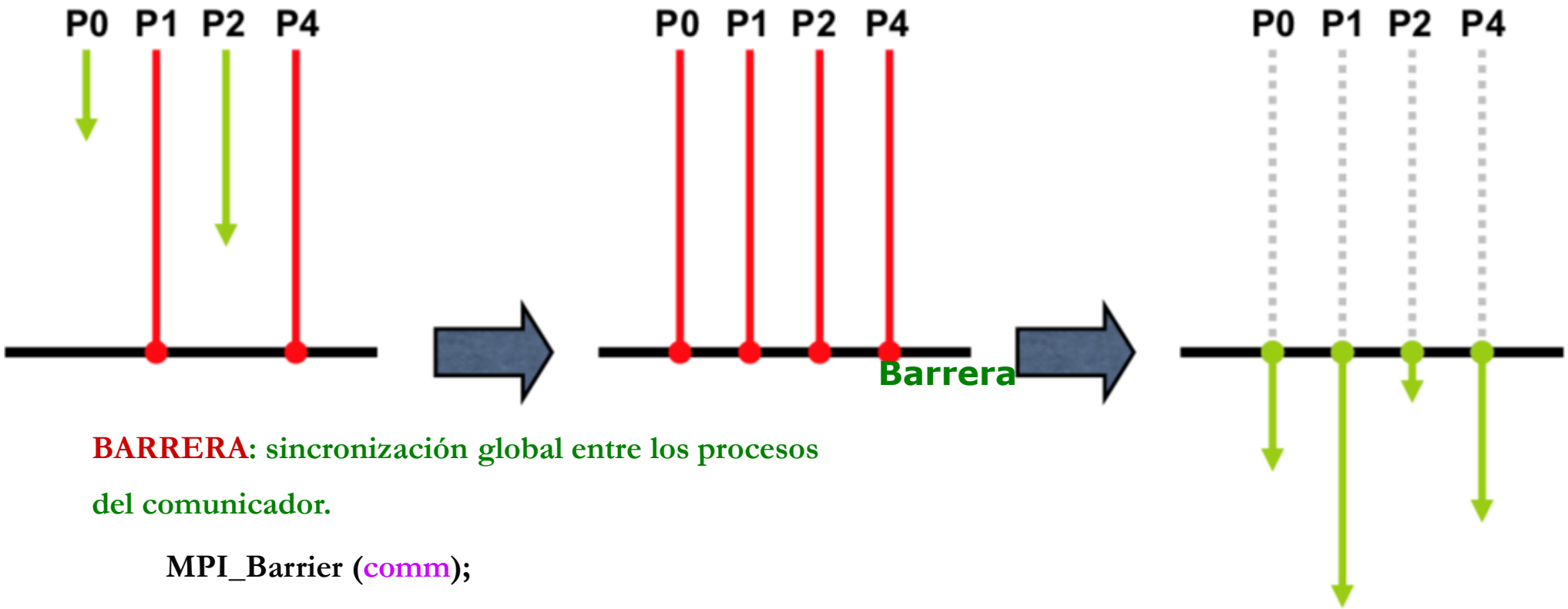
Ej.: cálculo del número pi

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

int main (int argc, char **argv)
{
 int pid, npr, i, n;
 double   PI = 3.1415926536;
 double h, x, pi_loc, pi_glob, sum;

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

 if (pid == 0)
 {  printf(" Núm de intervalos";
    scanf("%d",&n);
 }

 MPI_Bcast(&n, 1, MPI_INT, 0,
          MPI_COMM_WORLD);
```

```
h = 1.0 / (double) n;
sum = 0.0;
for (i=pid; i<n; i+=npr)
{
  x = (i + 0.5) *  h;
  sum += 4.0 / (1.0 + x*x);
}
pi_loc = h * sum;

MPI_Reduce(&pi_loc, &pi_glob, 1,
        MPI_DOUBLE, MPI_SUM, 0,
        MPI_COMM_WORLD);

if (pid == 0)
  printf("pi(+/-) = %.16f, error
  = %.16f\n", pi_glob,
  fabs(pi_glob - PI));

MPI_Finalize();
}
```

Sincronización: Barreras

3.- Sincronización de procesos



BARRERA: sincronización global entre los procesos del comunicador.

MPI_Barrier (comm);

La función se bloquea hasta que todos los procesos del comunicador llaman al la función (la ejecutan)

int MPI_Barrier (MPI_Comm comm)

Common mistakes with collectives

- Using a collective operation within one branch of an if-test of the rank

```
IF (my_id == 0) CALL MPI_BCAST(...
```

  - All processes, both the root (the sender or the gatherer) and the rest (receivers or senders), must call the collective routine
- Assuming that all processes making a collective call would complete at the same time
- Using the input buffer as the output buffer

```
CALL MPI_ALLREDUCE(a, a, n, MPI_REAL, MPI_SUM, ...
```

Mejora comunicaciones: agrupamiento de datos

Optimizar mensajes de envío y recepción agrupando los datos a enviar.

- Es mejor enviar un paquete con tres datos que tres paquetes con un dato cada uno.
- Las funciones Send y Recv indican explícitamente la dirección de comienzo del mensaje y el número de elementos que lo componen, pero hay una restricción: deben ser elementos consecutivos y del mismo tipo.

Opciones

- Con empaquetamiento.
- Con tipos derivados : definición de tipos de datos con estructura.

Objetivo: Disminuir el tiempo de comunicación:

$$\text{Tiempo de comunicación} = n^o \text{ mensajes} * (t_s + Lt_w)$$

L: tamaño de mensaje (en múltiplos del tipo de dato)

t_s: latencia y t_w : coste de envío por cada dato adicional.

Empaquetamiento

Optimizar la comunicación "empaquetando" los datos que hay que enviar en posiciones consecutivas de memoria.

Objetivo: Enviar en un mensaje tipos de datos diferentes.

Estrategia: **La comunicación se efectúa en tres fases:**

- **Antes de enviar el mensaje, se van añadiendo los datos a un *buffer* (vector) mediante una función MPI de empaquetamiento.**
- **Se envía el vector, que es de tipo `MPI_PACKED`.**
- **El receptor desempaqueta los datos recibidos mediante la función MPI de desempaquetamiento.**

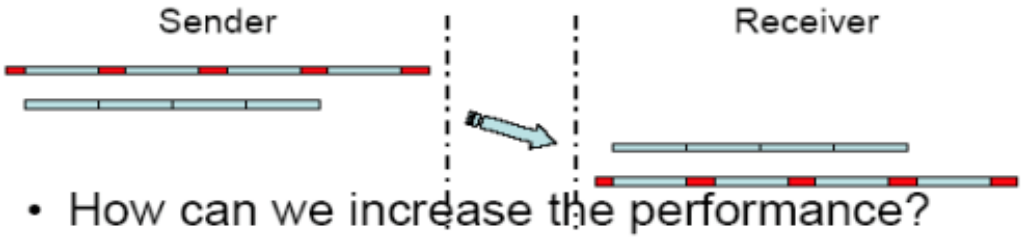
Empaquetamiento

```
void Leer_Datos (float* A, B; int* C; int pid) {
 char bufer[100]; int pos;
 if (pid == 0) {
   printf("-> A, B y C\n");
   scanf("%f %f %d", A, B, C);     // A, B, C, punteros a las vbles.
   pos = 0;
   MPI_Pack(A, 1, MPI_FLOAT, bufer, 100, &pos, MPI_COMM_WORLD);
   MPI_Pack(B, 1, MPI_FLOAT, bufer, 100, &pos, MPI_COMM_WORLD);
   MPI_Pack(C, 1, MPI_INT, bufer, 100, &pos, MPI_COMM_WORLD);
   MPI_Bcast(bufer, pos, MPI_PACKED, 0, MPI_COMM_WORLD);
 } else {
   MPI_Bcast(bufer, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
   pos = 0;
   MPI_Unpack(bufer, 100, &pos, A, 1, MPI_FLOAT, MPI_COMM_WORLD);
   MPI_Unpack(bufer, 100, &pos, B, 1, MPI_FLOAT, MPI_COMM_WORLD);
   MPI_Unpack(bufer, 100, &pos, C, 1, MPI_INT, MPI_COMM_WORLD);
 }
}
```

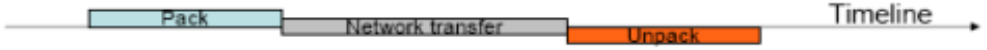
Mejora de comunicaciones: Utilizando tipos de datos para lograr alto rendimiento.

High performance MPI datatype

- Pack/Unpack Approach



- How can we increase the performance?
  - 3 distinct steps: pack, network transfer, unpack
  - No computation / communication recovery



- Increasing the computation / communication recovery
  - Split the computations in small slices

Tipos derivados

Objetivo: **Definir** nuevos tipos de datos**, en los que se permita agrupar datos de diferente tipo, tamaño... para formar un mensaje con cierta "estructura".**

El proceso de generación de nuevos tipos de datos se efectúa mediante dos funciones:

1. definición del tipo de datos
2. creación del tipo (*commit*)

Los nuevos tipos de datos que se crean se declaran como **MPI_Datatype**.

Tipos derivados

Tipo "vector"

Se define un vector de *n* elementos, del mismo tipo y tamaño, a distancia (*stride*) constante.

(normalmente un subconjunto de un *array* mayor).


```
> MPI_Type_vector(num_elem, tam, stride, tipo,
                  &nuevo_tipo);
```

El nuevo tipo contiene `num_elem` elementos de tamaño `tam`, a distancia `stride` uno de otro.

```
[ - MPI_Type_contiguous(num_elem, tipo, &nuevo_tipo); ]
```

Tipos derivados

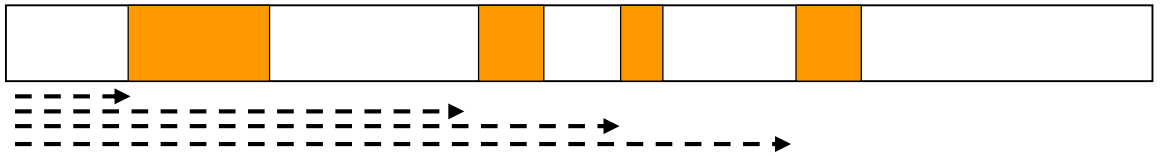
- **Ejemplo: enviar la** columna 2 **de la matriz** A**, de P0 a P1.**

```
int A[N][M];
MPI_Datatype Columna;
...
MPI_Type_vector(N, 1, M, MPI_INT, &Columna);
MPI_Type_commit(&Columna);

orig = 0; dest = 1;

if (pid == orig)
  MPI_Send(&A[0][2], 1, Columna, dest, 0, MPI_COMM_WORLD);
else if (pid == dest)
  MPI_Recv(&A[0][2], 1, Columna, orig, 0, MPI_COMM_WORLD, &info);
```

Tipos derivados

Tipo "indexed"

Se define un vector de *n* elementos, del mismo tipo, con tamaño y *stride* variable.



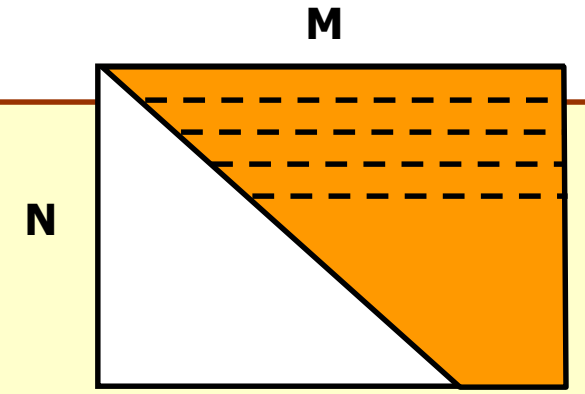
```
> MPI_Type_indexed(num_elem, tam, desp, tipo,
                   &nuevo_tipo);
```

`tam[]`: *array* que contiene el número de componentes de cada elemento que forma el nuevo tipo.

`desp[]`: *array* que contiene el desplazamiento necesario para acceder desde el comienzo del nuevo tipo a cada elemento.

Tipos derivados

- **Ejemplo: enviar de P0 a P1 el** triángulo superior **de la matriz** A**.**




```
int A[N][M], T[N][M];
MPI_Datatype Mtri;
...

for(i=0; i<N; i++) {
  long_bl[i] = M - i;
  desp[i] = (M+1) * i;
}

MPI_Type_indexed (N, long_bl, desp, MPI_INT, &Mtri);
MPI_Type_commit (&Mtri);

if (pid == 0)      MPI_Send(A, 1, Mtri, 1, 0, MPI_COMM_WORLD);
else if (pid == 1) MPI_Recv(T, 1, Mtri, 0, 0, MPI_COMM_WORLD, &info);
```

Tipos derivados

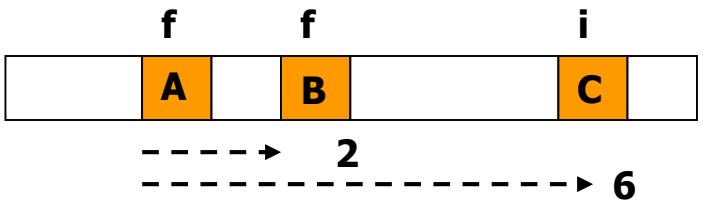
Tipo "struct"

Es un caso más general, en el que se agrupan *n* elementos de tamaño y tipo diferente.

Por ejemplo, P0 tiene que enviar a todos los procesadores tres parámetros: **`A`** y **`B`**, flotantes, y **`C`**, entero. Podemos formar un "paquete" con los tres parámetros y efectuar un único envío.

Para ello construimos un ***struct*** con:

- número de elementos
- tipo de cada elemento
- desplazamiento de cada elemento desde el origen



Tipos derivados

La función para definir el tipo **"struct"** es:

```
> MPI_Type_create_struct(num_elem, tam, desp,
          tipo, &nuevo_tipo);
```

tam[]: número de componentes de cada elemento que forma el nuevo tipo.

desp[]: desplazamiento necesario para acceder desde el comienzo del nuevo tipo a cada elemento (MPI_Aint).

tipo[]: tipo de cada elemento (MPI_Datatype).

```
> MPI_Get_address (&A, &dirA)
```

Devuelve la dirección de **A** en **dirA** (de tipo MPI_Aint)

Tipos derivados

```
void Crear_Tipo (float* A, B; int* C; MPI_Datatype* Mensaje) {
  int          tam[3];
  MPI_Aint     desp[3], dir1, dir2;          // address int
  MPI_Dataype  tipo[3];
  tam[0] = tam[1] = tam[2] = 1;
  tipo[0] = tipo[1] = MPI_FLOAT;
  tipo[2] = MPI_INT;

  despl[0] = 0;
  MPI_Get_address (A, &dir1)
  MPI_Get_address (B, &dir2)
  despl[1] = dir2 - dir1;
  MPI_Get_address (C, &dir2);
  despl[2] = dir2 - dir1;

  MPI_Type_struct(3, tam, desp, tipo, Mensaje);
  MPI_Type_commit(Mensaje);
}
```

```
... (A, B, C, punteros a las variables)
Crear_Tipo(A, B, C, &Mensaje);
MPI_Bcast(A, 1, Mensaje, 0, MPI_COMM_WORLD);
...
```

Procesos MPI y comunicadores

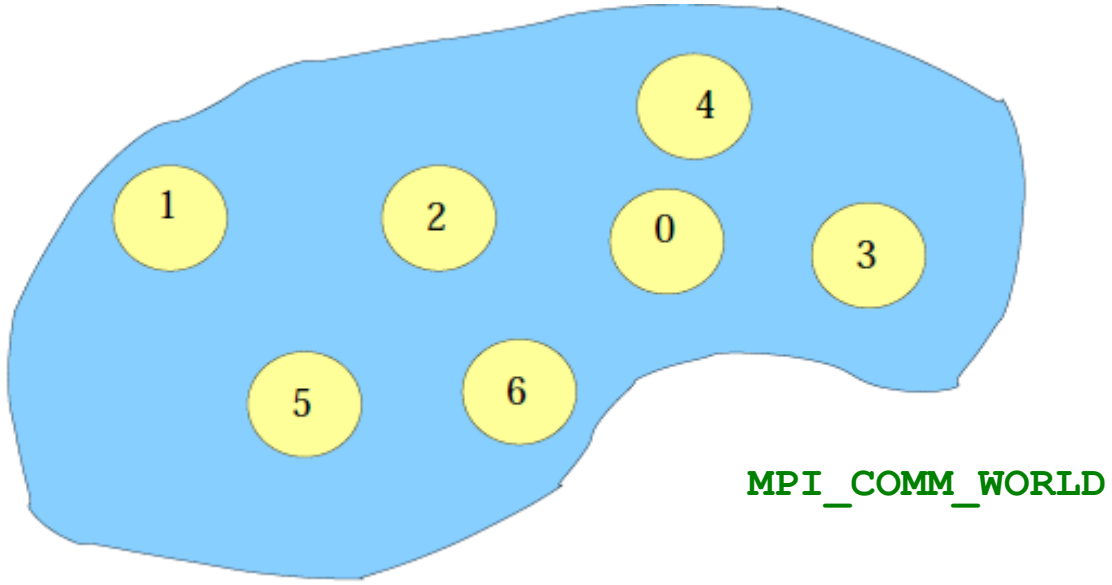
MPI gestiona los procesos **estáticamente** (número y asignación) (MPI2 también dinámicamente).

- MPI agrupa los procesos implicados en una ejecución paralela en "**comunicadores**".
- Un comunicador agrupa a procesos que pueden intercambiarse mensajes.
  - **Los procesos pueden congregarse("collected") en *grupos*.**
  - **Cada mensaje es enviado en un *contexto*, y debe recibirse en el mismo contexto.**
  - **Un grupo y un contexto juntos forman un *comunicador*.**
  - **Cada proceso se identifica por un numero de orden (*rank)* en el grupo asociado con un comunicador.**
  - **Un proceso puede formar parte de varios comunicadores y su número de identificación rank es privado a cada comunicador.**

MPI: Comunicadores

Las comunicaciones ocurren entre un grupo de procesos.

- **MPI_Init define el conjunto de procesos, en la ejecución paralela, entre los cuales la comunicación puede tener lugar.**
- **MPI_COMM_WORLD es el nombre del comunicador que contiene todos los procesos en ejecución cuando se inicia el programa.**



Comunicadores en MPI

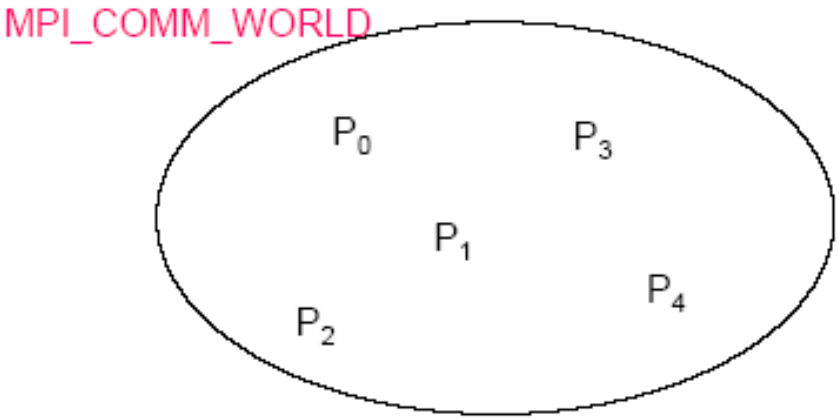
Los comunicadores proporcionan un manejador o "handle" para un grupo de procesos/procesadores.

- **Dentro de un comunicador a cada proceso se le asigna un número o "rank".**
- **Las funciones MPI_Comm_rank y MPI_Comm_size devuelve el identificador de orden (rank) y el numero de procesos.**
- **La asignación de procesos con su número de identificación (rank) depende de la implementación de MPI.**
- **El comunicador aparece en casi todos las llamadas a funciones MPI.**

Se pueden crear nuevos comunicadores:

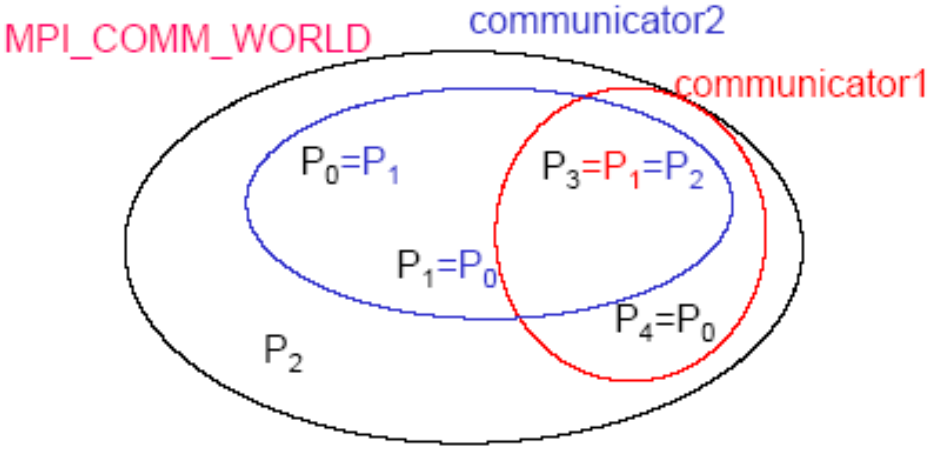
- **Permiten crear subgrupos de procesadores.**
- **Puedes usarse para implementar determinadas topologías de comunicación entre procesadores.**

MPI Comunicadores



Communicators:
Logical grouping of processes.

Predefined:
MPI_COMM_WORLD

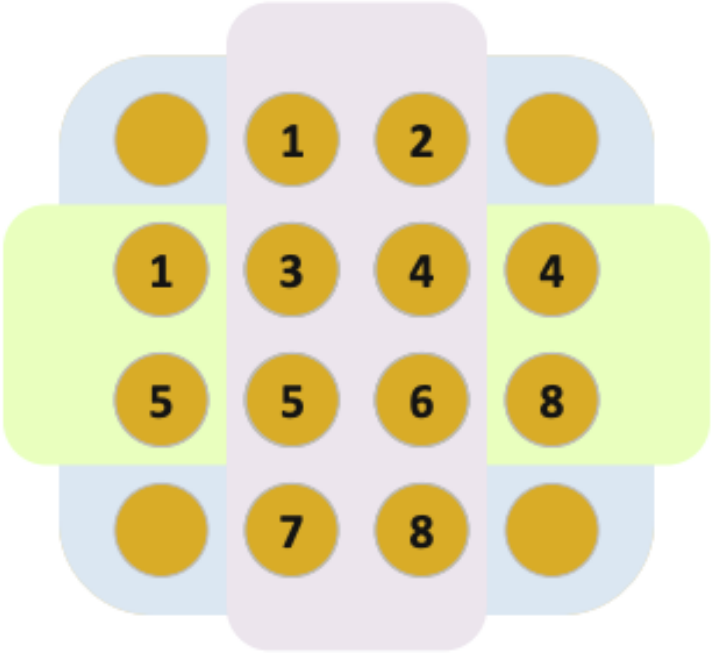


My own communicators:
communicator1,
communicator2

MPI Comunicadores

```
mpiexec -np 16 ./test
```

Communicators do not need to contain all processes in the system



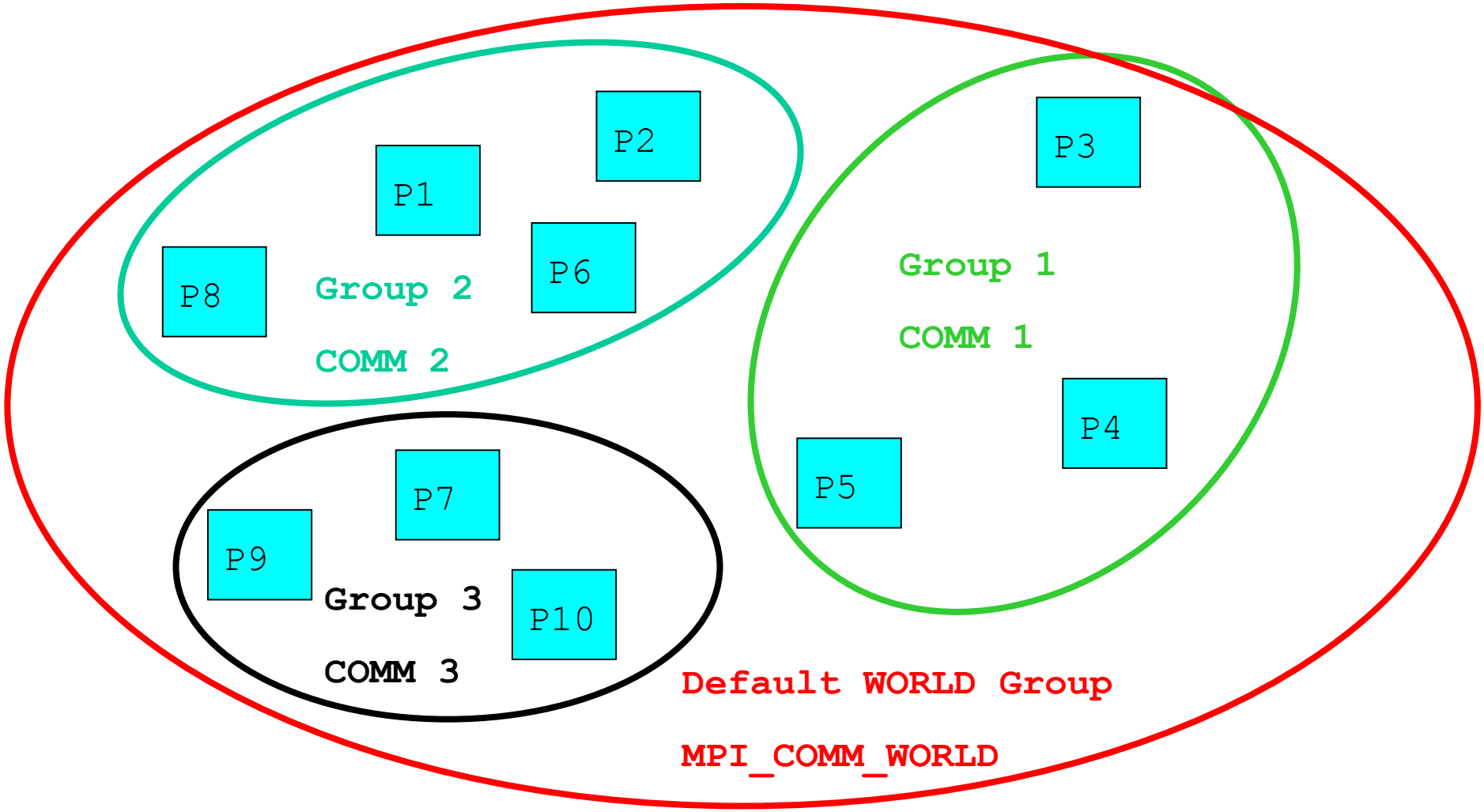
When you start an MPI program, there is one predefined communicator **MPI_COMM_WORLD**

Every process in a communicator has an ID called as "rank"

Can make copies of this communicator (same group of processes, but different "aliases")

The same process might have different ranks in different communicators

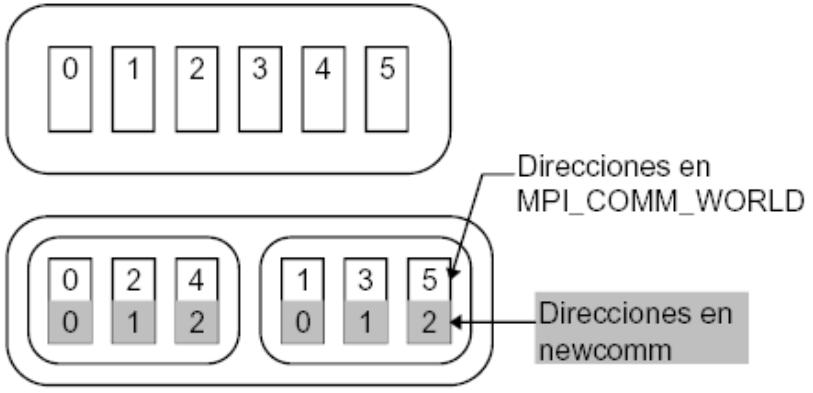
Comunicadores y grupos



Grupos y comunicadores

MPI_Comm_split(MPI_COMM_WORLD, int color, int Key, MPI_Comm newcom)

- Es el mecanismo mas simple si hay que crear comunicadores "similares" a partir de uno determinado
  - Los procesos de **`com1`** = MPI_COMM_WORLD se van a agrupar en nuevos comunicadores en función de la característica (color) con la que llamen a la función.
  - Se crean tantos comunicadores como características diferentes se utilicen, todos con el mismo nombre com2=newcom y dentro de cada grupo el nuevo id se asigna en base al parámetro key, resolviendo id similares de acuerdo al valor del antiguo id en com1.



riba: una colección de procesos en el comunicador MPI_COMM_WOF

Grupos y comunicadores

- Ejemplo: crear comunicadores para todas las filas de un grupo de n × n procesos:

```
// pid = identificador en el comunicador global
mi_fila = pid / n;
MPI_Comm_split(MPI_COMM_WORLD, mi_fila, 1, &CFILA);
```

⋆Si había 3 × 3 procesos, ahora tendremos un nuevo comunicador en el que el grupo de procesos tiene los identificaridores {0,1,2} en P0,P1,P2 / {0,1,2} en P3,P4,P5 / {0,1,2} en P6,P7,P8

⋆Se trata de una **función colectiva**; todos los procesos del comunicador original tienen que ejecutarla. Si un proceso no va a formar parte de los nuevos comunicadores, llama a la función con la clave **MPI_UNDEFINED** (formará parte del comunicador **MPI_COMM_NULL**).

Definir grupos y sus *comunicadores*

```
MPI_Comm new_comm;
MPI_Group orig_grp, new_grp;
int ranks1[];
```

1. **Obtener un conjunto de IDs de un comunicador existente**

   MPI_Comm_group (MPI_COMM_WORLD, &orig_group)

2. **Crear un grupo como un subconjunto de un dado grupo**

   MPI_Group_incl (org_grp, count, ranks1, &new_grp)

   MPI_Group_excl (org_grp, count, ranks1, &new_grp)

3. **Definir un nuevo comunicador para el grupo.**

   MPI_Comm_create (MPI_COMM_WORLD, **new_grp, &newcomm,)**

Creación de comunicadores

- Para crear grupos de procesos entre los que poder intercambiar información, hay que seguir los siguientes pasos:

1.- Extraer de un comunicador inicial, **com1**, el grupo de procesos asociados, **gr1**:

```
> MPI_Comm_group(com1, &gr1);
```

Si **com1** es **MPI_COMM_WORLD**, **gr1** contendrá a todos los procesos.

2.- Crear un nuevo grupo de procesos, gr2, eligiendo del grupo gr1 aquellos que queremos que formen parte del nuevo comunicador:

```
> MPI_Group_incl(gr1, npr_gr2, pid_gr1, &gr2);
```

pid_gr1 es un array de **npr_gr2** elementos que indica el **pid** de los procesos que se eligen para formar el nuevo grupo.

3. Finalmente, se crea el comunicador **com2** con los procesos del grupo **gr2** extraídos del comunicador **com1**:

```
> MPI_Comm_create(com1, gr2, &com2);
```

Creación de comunicadores

- Ejemplo: una matriz está repartida por bloques en n × n procesos. Así definiríamos el grupo "fila 0" de procesos y haríamos un *broadcast* en esa fila:

```
MPI_Group  gr1, grf0;
MPI_Comm   CF0;

for(i=0; i<n; i++) pids[i] = i;    /* lista de proc. */

MPI_Comm_group (MPI_COMM_WORLD, &gr1);
MPI_Group_incl (gr1, n, pids, &grf0);
MPI_Comm_create(MPI_COMM_WORLD, grf0, &CF0);
...
MPI_Comm_rank(CF0, &pid_f0);
if (pid_f0 == 0)
   MPI_Broadcast(&A[0][0], tam, MPI_FLOAT, 0, CF0);
...
```

Creación de comunicadores

- Algunas precisiones:
  - Los grupos de procesos son variables de tipo **`MPI_Group`** y los comunicadores son de tipo **`MPI_Comm`**.
  - **`MPI_Comm_create`** es una operación colectiva, que debe ser llamada por todos los procesos del comunicador original (aunque no vayan a formar parte del nuevo comunicador; se les devuelve el valor **`MPI_COMM_NULL`**).
  - Al finalizar su uso hay que "deshacer" el comunicador, ejecutando **`MPI_Comm_free`**.
  - Otras funciones para trabajar con grupos: **`MPI_Group_rank`**, **`size`**, **`free`**, **`union`**, **`intersection`**...

Topologías virtuales:

Un comunicador puede incluir, además del grupo y el contexto, otro tipo de información o atributos; por ejemplo, una topología (virtual).

- **MPI tiene rutinas que permiten definir un *grid* o retícula de procesos que se adapte bien a una geometría particular del cálculo.**
- **El concepto de *topología virtual* es una característica de MPI que permite manejar un grid de procesos.**
- **El objetivo es obtener códigos mas concisos y simples, permitiendo también optimizar la comunicación entre los nodos.**
- **Una topología virtual se asocia con un comunicador.**

Dos tipos de topologías ➔ Cartesianas
➔ Graph (generales)

Las topologías **cartesianas** son rejillas cartesianas uni o bi-dimensionales

Son un caso particular de las generales, pero como son muy utilizadas, hay subrutinas dedicadas

Para asociar un punto de la rejilla virtual a cada procesador es preciso especificar la siguiente información:

1. Numero de dimensiones de la rejilla (1 o 2)
2. Largo de cada dimension (# de procs por fila y/o columna)
3. Periodicidad de la dimensión (condiciones de contorno: anillos, toros)
4. Optimización (reordenar los procesadores físicos)

Topología cartesiana

Para trabajar con una distribución 2D de datos, podemos definir una "retícula" de procesos (p.e., de 4x4 si tenemos 16 procesadores).

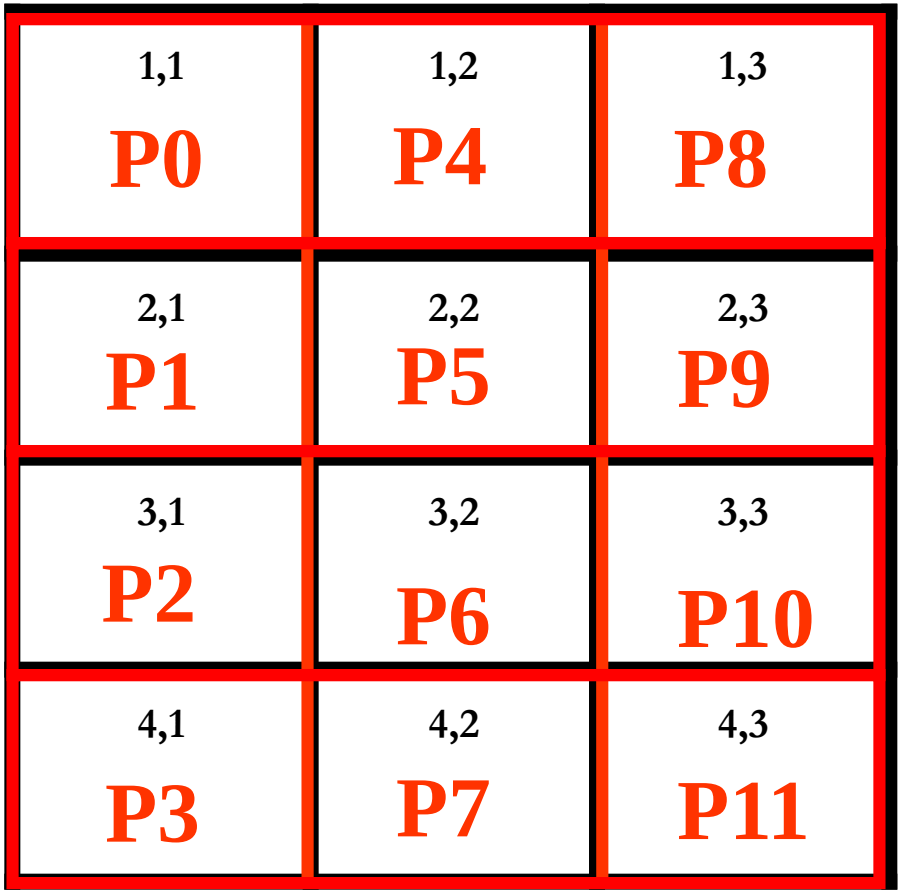
```
MPI_Comm  old_com, comm_cart;
int ndims =2, dims[2]={4,4}, warp_around[2]={0,0}, reorder=0;
```

MPI_CART_CREATE(old_comm, ndims, dims, wrap_around, reorder, &comm_cart)

ndims: dimensión

dims: vector con la dimensión de cada eje de coordenadas
wrap_around: vector lógico que indica si existe periodicidad o no en cada eje
reorder: **.false.** Si los datos ya fueron distribuidos. Usa los ranks del comunicador original
.true. Si los datos todavía no fueron distribuidos. Reordena los procesos para optimizar la comunicación

Definir una matriz de procesos!



Grid 2-d

y definir comunicadores fila y comunicadores columna

Ejemplo Topología cartesiana

Crear una topología cartesiana bidimensional de q x q, con condiciones de contorno periódicas (toro) y el sistema reordena id procesos

```
MPI_Comm grid_comm;
int      dim_sizes(2), wrap_around(2), reorder;

reorder = 1
dim_sizes(1) = q
dim_sizes(2) = q
wrap_around(1) = 1
wrap_around(2) = 1

MPI_Cart_create( MPI_COMM_WORLD , 2 , dim_sizes, wrap_around , reorder ,
&grid_comm );
```

¿Cómo obtener las coordenadas?

```
MPI_Cart_coord( grid_comm , mi_grid_rank , 2 , coord );
MPI_Comm_rank( grid_comm , &mi_grid_rank );
```

Topologías

- Se puede trabajar con una identificación doble de los procesos: el `pid` en el nuevo comunicador, y las coordenadas cartesianas de la topología asociada.

  Dos funciones permiten pasar de una a otra:

```
> MPI_Cart_coords(com2, pid2, dim, coord);
```

  com2: nuevo comunicador con topología

  pid2: identificador del proceso en el nuevo comunicador

  dim: número de dimensiones

  coord: *array* donde se devuelven las coordenadas

```
> MPI_Cart_rank(com2, coord, pid);
```

  Es el "inverso" de MPI_Cart_coord

Topologías

- Se pueden obtener las direcciones de los "vecinos" en la topología:

MPI_Cart_shift(grid_comm , dir, disp , rank_source, rank_dest)

Obtiene el rank del proceso de origen y el de destino (para usar en un send/recv) entre procesos separados por *disp* procesos en la dirección *dir*. "Averigua quiénes son los vecinos".

MPI_Cart_get(grid_comm , numdim , dim_sizes , wrap_around , coord)

Se llama con numdim=2 y el comunicador de la topología cartesiana (grid_comm.

Obtiene las coordenadas del proceso además de los tamaños de las dimensiones y la periodicidad de un comunicador cartesiano dado.

vectores de salida: dim_sizes[2] , wrap_around[2] , coord[2]

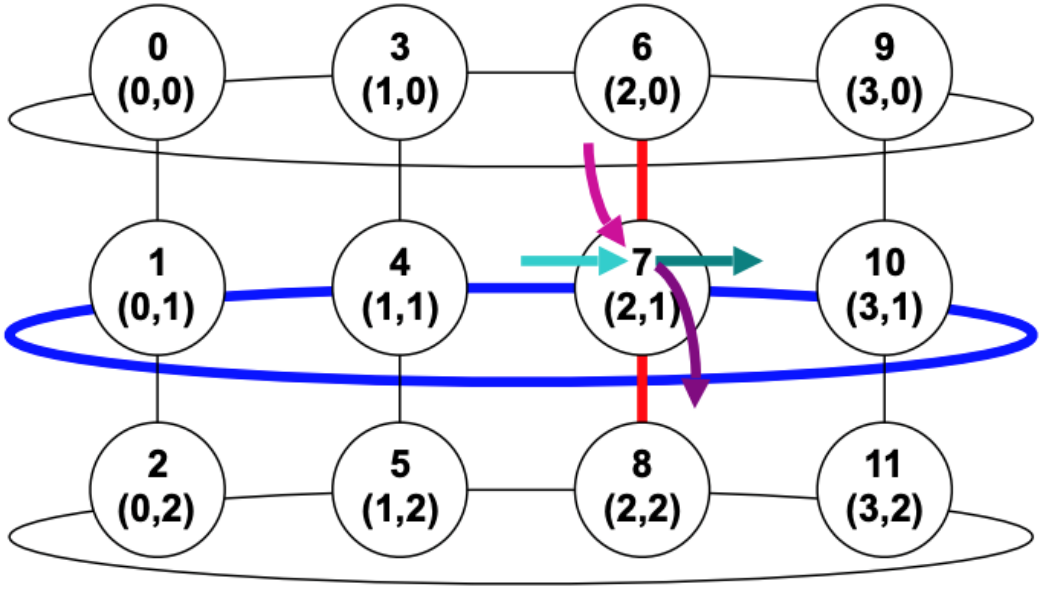
- EN RESUMEN un comunicador representa a un grupos de procesos que se reconocen entre sí como posible origen y destino de sus comunicaciones.

  Una topología es un mecanismo alternativo de identificación "lógica" de procesos dentro de un grupo, que genera un nuevo comunicador y, en su caso, subgrupos.

Topologías: MPI_Cart_shift – Example

MPI_Cart_shift(grid_comm , dir, disp , rank_source, rank_dest)

Obtiene el rank del proceso de origen y el de destino (para usar en un send/recv) entre procesos separados por *disp* procesos en la dirección *dir*. "Averigua quiénes son los vecinos".



invisible input argument: my_rank of the running process executing:

- MPI_Cart_shift(cart, direction, displace, rank_prev, rank_next, ierror)

example on	0 or	+1	4	10
process rank=7	1	+1	6	8

Paralelizar algoritmos

- **El algoritmo deberá:**
  - **Balancear de carga**
  - **Minimizar la comunicación**
- **Afortunadamente, ya hay desarrollados algoritmos paralelos, particularmente en el área del álgebra computacional.**
  - **No hay que reinventar la rueda, hay que sacar partido del trabajo hecho por otro programador.**
  - **Usar, si es posible, la librerías paralelas ya disponibles:**

    ScaLAPACK, PETSc, UCSparseLib, etc.

MPI

MPI specification

http://www.mpi-forum.org/

- Free Implementations:
  - MPICH MPI Implementation

    http://www-unix.mcs.anl.gov/mpi/mpich/
  - LAMMPI Implementation

    http://www.lam-mpi.org/
- Newsgroup

  comp.parallel.mpi
- FAQ

  http://www.erc.msstate.edu/mpi/mpi-faq.html