

## S3. Programación con MPI

J. M. Alonso, F. Alvarruiz, I. Blanquer, D. Guerrero, J. Ibáñez,  
E. Ramos, J. E. Román

Departament de Sistemes Informàtics i Computació  
Universitat Politècnica de València

Curso 2016/17



1

### Contenido

- 1 Conceptos Básicos
  - Modelo de Paso de Mensajes
  - El Estándar MPI
  - Modelo de Programación MPI
- 2 Comunicación Punto a Punto
  - Semántica
  - Primitivas Bloqueantes
  - Otras Primitivas
  - Ejemplos
- 3 Comunicación Colectiva
  - Sincronización
  - Difusión
  - Reparto
  - Reducción
- 4 Otras Funcionalidades
  - Tipos de Datos Derivados

2

## Apartado 1

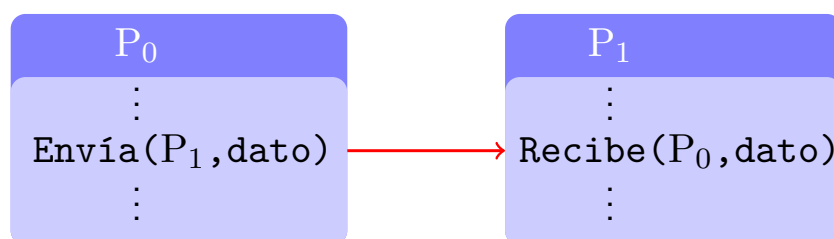
# Conceptos Básicos

- Modelo de Paso de Mensajes
- El Estándar MPI
- Modelo de Programación MPI

3

## Modelo de Paso de Mensajes

Intercambio de información mediante envío y recepción explícitos de mensajes



Modelo más usado en computación a gran escala – Bibliotecas de funciones (aprendizaje más fácil que un lenguaje nuevo)

Ventajas:

- Universalidad
- Fácil comprensión
- Gran expresividad
- Mayor eficiencia

Inconvenientes:

- Programación compleja
- Control total de las comunicaciones

4

## El Estándar MPI

MPI es una especificación propuesta por un comité de investigadores, usuarios y empresas

<http://www.mcs.anl.gov/mpi>

Especificaciones:

- MPI-1.0 (1994), última actualización MPI-1.3 (2008)
- MPI-2.0 (1997), última actualización MPI-2.2 (2009)
- MPI-3.0 (2012), última actualización MPI-3.1 (2015)

Antecedentes:

- Cada fabricante ofrecía su propio entorno (migración costosa)
- PVM (*Parallel Virtual Machine*) fue un primer intento de estandarización

5

## Características de MPI

Características principales:

- Es portable a cualquier plataforma paralela
- Es simple (con tan sólo 6 funciones se puede implementar cualquier programa)
- Es potente (más de 300 funciones)
- Soporta el modelo SPMD y MPMD

El estándar especifica interfaz para C y Fortran

Hay muchas implementaciones disponibles:

- Propietarias: IBM, Cray, SGI, ...
- MPICH ([www.mpich.org](http://www.mpich.org))
- Open MPI ([www.open-mpi.org](http://www.open-mpi.org))
- MVAPICH ([mvapich.cse.ohio-state.edu](http://mvapich.cse.ohio-state.edu))

6

## Modelo de Programación

La programación en MPI se basa en funciones de biblioteca  
Para su uso, se requiere una inicialización

### Ejemplo

```
#include <mpi.h>
int main(int argc, char* argv[]) {
    int k;          /* rango del proceso */
    int p;          /* número de procesos */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &k);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    printf("Soy el proceso %d de %d\n", k, p);
    MPI_Finalize();
    return 0;
}
```

- Es obligatorio llamar a MPI\_Init y MPI\_Finalize
- Una vez inicializado, se pueden realizar diferentes operaciones

7

## Modelo de Programación – Operaciones

Las operaciones se pueden agrupar en:

- Comunicación punto a punto  
*Intercambio de información entre pares de procesos*
- Comunicación colectiva  
*Intercambio de información entre conjuntos de procesos*
- Gestión de datos  
*Tipos de datos derivados (p.e. datos no contiguos en memoria)*
- Operaciones de alto nivel  
*Grupos, contextos, comunicadores, atributos, topologías*
- Operaciones avanzadas (MPI-2, MPI-3)  
*Entrada-salida, creación de procesos, comunicación unilateral*
- Utilidades  
*Interacción con el entorno del sistema*

La mayoría operan sobre comunicadores

8

## Modelo de Programación – Comunicadores

Un **comunicador** es una abstracción que engloba los siguientes conceptos:

- *Grupo*: conjunto de procesos
- *Contexto*: para evitar interferencias entre mensajes distintos

Un comunicador agrupa a  $p$  procesos

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Cada proceso tiene un identificador (rango), un número entre 0 y  $p - 1$

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

9

## Modelo de Ejecución

El modelo de ejecución de MPI sigue un esquema de creación simultánea de procesos al lanzar la aplicación

La ejecución de una aplicación suele hacerse con

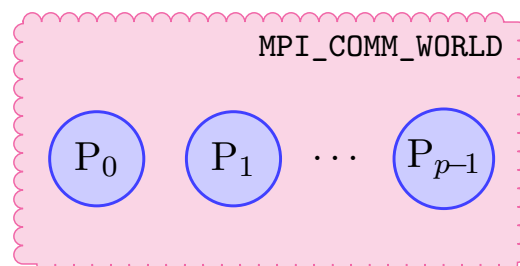
```
mpixexec -n p programa [argumentos]
```

Al ejecutar una aplicación:

- Se lanzan  $p$  copias del mismo ejecutable
- Se crea un comunicador `MPI_COMM_WORLD` que engloba a todos los procesos

Además del modelo SPMD, también se permite MPMD

MPI-2 ofrece un mecanismo para crear nuevos procesos



10

## Apartado 2

# Comunicación Punto a Punto

- Semántica
- Primitivas Bloqueantes
- Otras Primitivas
- Ejemplos

11

## Comunicación Punto a Punto – el Mensaje

Los mensajes deben ser enviados explícitamente por el emisor y recibidos explícitamente por el receptor

Envío estándar:

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

Recepción estándar:

```
MPI_Recv(buf, count, datatype, src, tag, comm, stat)
```

El contenido del mensaje viene definido por los 3 primeros argumentos:

- Un *buffer* de memoria donde está almacenada la información
- El número de elementos que componen el mensaje
- El tipo de datos de los elementos (p.e. MPI\_INT)

12

## Comunicación Punto a Punto – el Sobre

Para efectuar la comunicación, es necesario indicar el destino (`dest`) y el origen (`src`)

- La comunicación está permitida sólo dentro del mismo comunicador, `comm`
- El origen y el destino se indican mediante identificadores de procesos
- En la recepción se permite utilizar `src=MPI_ANY_SOURCE`

Se puede utilizar un número entero (etiqueta o `tag`) para distinguir mensajes de distinto tipo

- En la recepción se permite utilizar `tag=MPI_ANY_TAG`

En la recepción, el estado (`stat`) contiene información:

- Proceso emisor (`stat.MPI_SOURCE`), etiqueta (`stat.MPI_TAG`)
- Longitud del mensaje (explicado en p. 44)

Nota: pasar `MPI_STATUS_IGNORE` si no se requiere

13

## Comunicación Punto a Punto – Semántica

Hay que tener en cuenta los siguientes aspectos relativos a la semántica de las operaciones de comunicación punto a punto:

- Orden** Si un proceso envía dos mensajes al mismo destino, entonces el receptor no recibirá el segundo antes que el primero
- Progreso** Si el emisor desea enviar y el receptor recibir, eventualmente la operación terminará y ambos podrán continuar
- Equidad** No está garantizada—un proceso emisor podría ser repetidamente adelantado por otros emisores
- Recursos** La escasez de recursos puede afectar a algunas operaciones (p.e. error si un buffer no es suficientemente grande)

14

## Modos de Envío Punto a Punto

Existen los siguientes modos de envío:

- Modo de envío síncrono
- Modo de envío con memoria intermedia (buffer)
- Modo de envío estándar

El modo estándar es el más utilizado

El resto de modos pueden ser útiles para obtener mejores prestaciones o mayor robustez

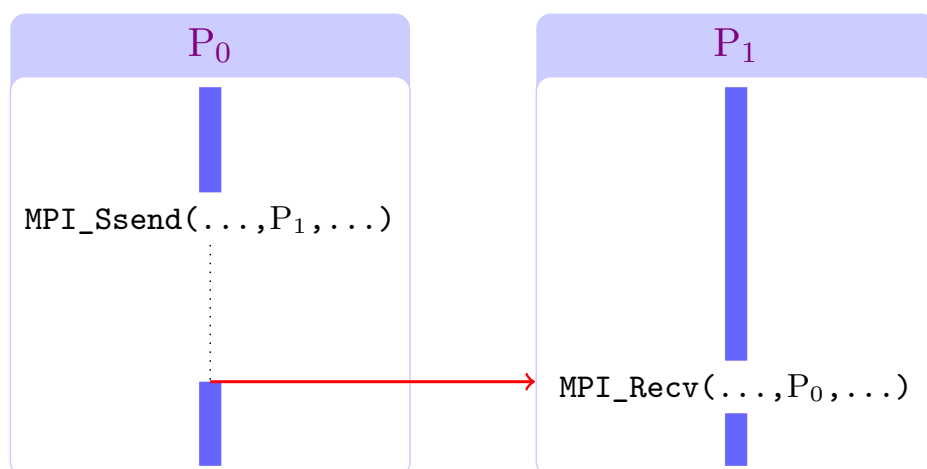
Para cada modo, existen primitivas bloqueantes y no bloqueantes

15

## Modo de Envío Síncrono

```
MPI_Ssend(buf, count, datatype, dest, tag, comm)
```

Implementa el modelo de envío con “rendezvous”: el emisor se bloquea hasta que el receptor desea recibir el mensaje



- Ineficiente: el emisor queda bloqueado sin hacer nada útil

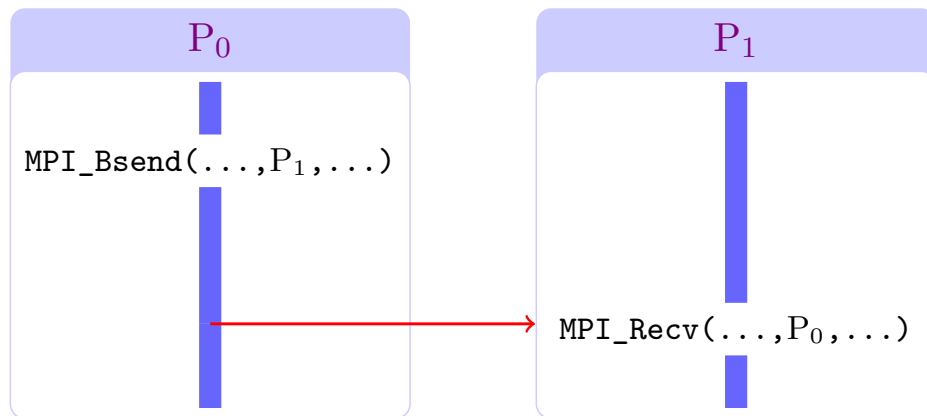
16



## Modo de Envío con Buffer

```
MPI_Bsend(buf, count, datatype, dest, tag, comm)
```

El mensaje se copia a una memoria intermedia y el proceso emisor continúa su ejecución



- Inconvenientes: copia adicional y posibilidad de fallo
- Se puede proporcionar un buffer (MPI\_Buffer\_attach)

17

## Modo de Envío Estándar

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

Garantiza el funcionamiento en todo tipo de sistemas ya que evita problemas de almacenamiento

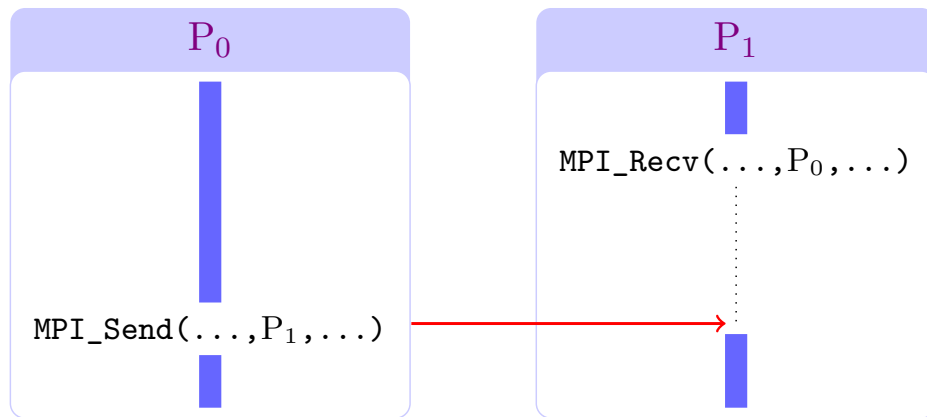
- Los mensajes cortos son enviados generalmente con MPI\_Bsend
- Los mensajes largos son enviados generalmente con MPI\_Send

18

## Recepción Estándar

```
MPI_Recv(buf, count, datatype, src, tag, comm, stat)
```

Implementa el modelo de recepción con “rendezvous”: el receptor se bloquea hasta que el mensaje llega



- Ineficiente: el proceso receptor queda bloqueado sin hacer nada útil

19

## Primitivas de Envío No Bloqueantes

```
MPI_Isend(buf, count, datatype, dest, tag, comm, req)
```

Se inicia el envío, pero el emisor no se bloquea

- Tiene un argumento adicional (`req`)
- Para reutilizar el buffer es necesario asegurarse de que el envío se ha completado

### Ejemplo

```
MPI_Isend(A, n, MPI_DOUBLE, dest, tag, comm, &req);  
...  
/* Comprobar que el envío ha terminado,  
   con MPI_Test o MPI_Wait */  
A[10] = 2.6;
```

- Solapamiento de comunicación y cálculo sin copia extra
- Inconveniente: programación más difícil

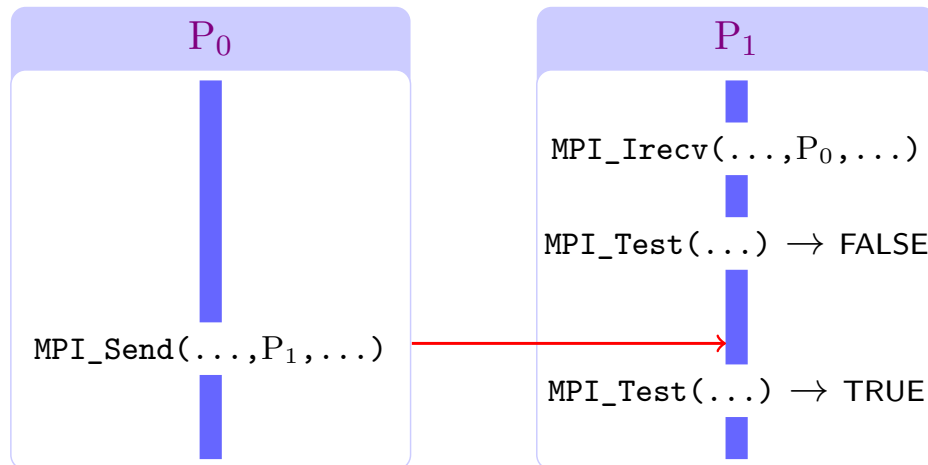
20

## Recepción No Bloqueante

```
MPI_Irecv(buf, count, type, src, tag, comm, req)
```

Se inicia la recepción, pero el receptor no se bloquea

- Tiene un argumento adicional (req)
- Es necesario comprobar después si el mensaje ha llegado



- Ventaja: solapamiento de comunicación y cálculo
- Inconveniente: programación más difícil

21

## Operaciones Combinadas

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,  
recvbuf, recvcount, recvtype, source, recvtag, comm,  
status)
```

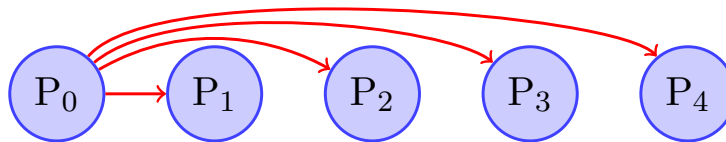
Realiza una operación de envío y recepción al mismo tiempo (no necesariamente con el mismo proceso)

```
MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag,  
source, recvtag, comm, status)
```

Realiza una operación de envío y recepción al mismo tiempo sobre la misma variable

22

## Ejemplo – Difusión



### Difusión de un valor numérico desde $P_0$

```
double val;
MPI_Status status;
int p, rank, i;

MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
    val = 3.14159;    /* valor a difundir */
    for (i=1; i<p; i++)
        MPI_Send(&val, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
} else {
    MPI_Recv(&val, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
}
```

23

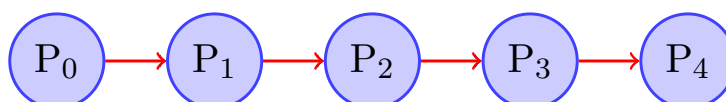
## Ejemplo – Desplazamiento en Malla 1-D (1)

Cada procesador ha de enviar su dato al vecino derecho y sustituirlo por el dato que recibe del vecino izquierdo

### Desplazamiento en Malla 1-D – versión trivial

```
if (rank == 0) {
    MPI_Send(&val, 1, MPI_DOUBLE, rank+1, 0, comm);
} else if (rank == p-1) {
    MPI_Recv(&val, 1, MPI_DOUBLE, rank-1, 0, comm, &status);
} else {
    MPI_Send(&val, 1, MPI_DOUBLE, rank+1, 0, comm);
    MPI_Recv(&val, 1, MPI_DOUBLE, rank-1, 0, comm, &status);
}
```

Inconveniente: Secuencialización - las comunicaciones se realizan (probablemente) secuencialmente, sin concurrencia



24

## Ejemplo – Desplazamiento en Malla 1-D (2)

En algunos casos, la programación se puede simplificar utilizando procesos nulos

### Desplazamiento en Malla 1-D – procesos nulos

```
if (rank == 0) prev = MPI_PROC_NULL;
else prev = rank-1;
if (rank == p-1) next = MPI_PROC_NULL;
else next = rank+1;

MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
MPI_Recv(&val, 1, MPI_DOUBLE, prev, 0, comm, &status);
```

El envío al proceso MPI\_PROC\_NULL finaliza enseguida; la recepción de un mensaje del proceso MPI\_PROC\_NULL no recibe nada y finaliza enseguida

Esta versión no resuelve el problema de la secuencialización

25

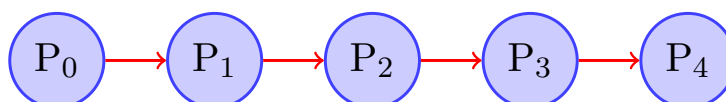
## Ejemplo – Desplazamiento en Malla 1-D (3)

Solución a la secuencialización: Protocolo Pares-Impares

### Desplazamiento en Malla 1-D – pares-impares

```
if (rank == 0) prev = MPI_PROC_NULL;
else prev = rank-1;
if (rank == p-1) next = MPI_PROC_NULL;
else next = rank+1;

if (rank%2 == 0) {
    MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
    MPI_Recv(&val, 1, MPI_DOUBLE, prev, 0, comm, &status);
} else {
    MPI_Recv(&tmp, 1, MPI_DOUBLE, prev, 0, comm, &status);
    MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
    val = tmp;
}
```



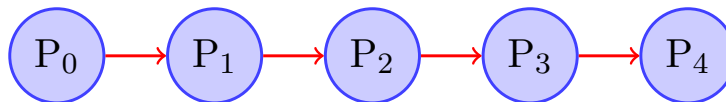
26

## Ejemplo – Desplazamiento en Malla 1-D (4)

Solución a la secuencialización: Operaciones Combinadas

### Desplazamiento en Malla 1-D – sendrecv

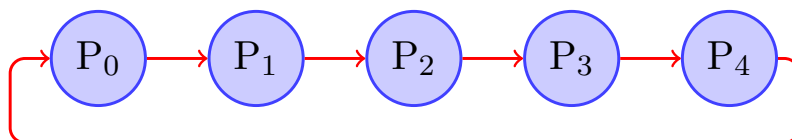
```
if (rank == 0) prev = MPI_PROC_NULL;  
else prev = rank-1;  
if (rank == p-1) next = MPI_PROC_NULL;  
else next = rank+1;  
  
MPI_Sendrecv_replace(&val, 1, MPI_DOUBLE, next, 0, prev, 0,  
                    comm, &status);
```



27

## Ejemplo – Desplazamiento en Anillo

En el caso del anillo, todos los procesos han de enviar y recibir



### Desplazamiento en Anillo – versión trivial

```
if (rank == 0) prev = p-1;  
else prev = rank-1;  
if (rank == p-1) next = 0;  
else next = rank+1;  
  
MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);  
MPI_Recv(&val, 1, MPI_DOUBLE, prev, 0, comm, &status);
```

Se producirá **interbloqueo** en el caso de envío síncrono

Soluciones: protocolo pares-impares u operaciones combinadas

28

### Apartado 3

## Comunicación Colectiva

- Sincronización
- Difusión
- Reparto
- Reducción

29

## Operaciones de Comunicación Colectiva

Involucran a **todos los procesos** de un grupo (comunicador) – todos ellos deben ejecutar la operación

Operaciones disponibles:

- |                                     |                                       |
|-------------------------------------|---------------------------------------|
| ■ Sincronización ( <i>Barrier</i> ) | ■ Multi-recogida ( <i>Allgather</i> ) |
| ■ Difusión ( <i>Bcast</i> )         | ■ Todos a todos ( <i>Alltoall</i> )   |
| ■ Reparto ( <i>Scatter</i> )        | ■ Reducción ( <i>Reduce</i> )         |
| ■ Recogida ( <i>Gather</i> )        | ■ Prefijación ( <i>Scan</i> )         |

Estas operaciones suelen tener como argumento un proceso (root) que realiza un papel especial

Prefijo “All”: Todos los procesos reciben el resultado

Sufijo “v”: La cantidad de datos en cada proceso es distinta

30

## Sincronización

`MPI_Barrier(comm)`

Operación pura de sincronización

- Todos los procesos de `comm` se detienen hasta que todos ellos han invocado esta operación

### Ejemplo – medición de tiempos

```
MPI_Barrier(comm);
t1 = MPI_Wtime();
/*
   ...
*/
MPI_Barrier(comm);
t2 = MPI_Wtime();

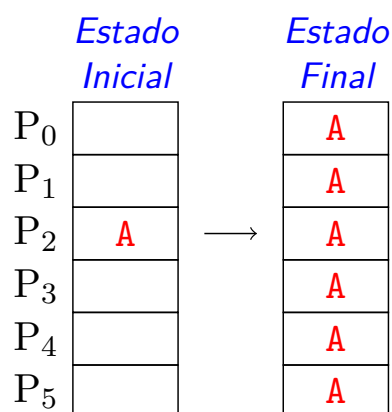
if (!rank) printf("Tiempo transcurrido: %f s.\n", t2-t1);
```

31

## Difusión

`MPI_Bcast(buffer, count, datatype, root, comm)`

El proceso `root` difunde al resto de procesos el mensaje definido por los 3 primeros argumentos



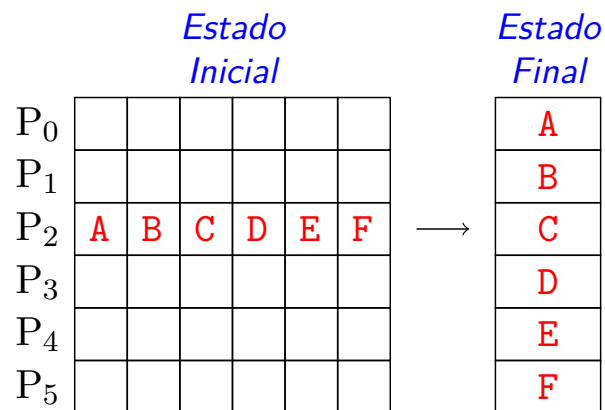
32



## Reparto

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf,  
            recvcount, recvtype, root, comm)
```

El proceso root distribuye una serie de datos al resto de procesos



Versión asimétrica: MPI\_Scatterv

33

## Reparto: Ejemplo

El proceso  $P_0$  reparte un vector de 15 elementos (a) entre 3 procesos que reciben los datos en el vector b

### Ejemplo de reparto

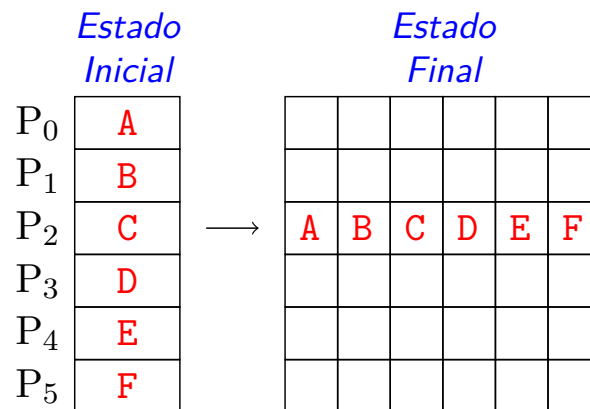
```
int main(int argc, char *argv[])  
{  
    int i, myproc;  
    int a[15], b[5];  
  
    MPI_Init(&argc, &argv);  
    MPI_Comm_rank(MPI_COMM_WORLD, &myproc);  
    if (myproc==0) for (i=0;i<15;i++) a[i] = i+1;  
  
    MPI_Scatter(a, 5, MPI_INT, b, 5, MPI_INT, 0, MPI_COMM_WORLD);  
  
    MPI_Finalize();  
    return 0;  
}
```

34

## Recogida

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf,  
recvcount, recvtype, root, comm)
```

Es la operación inversa de MPI\_Scatter: Cada proceso envía un mensaje a root, el cual lo almacena de forma ordenada de acuerdo al índice del proceso en el buffer de recepción



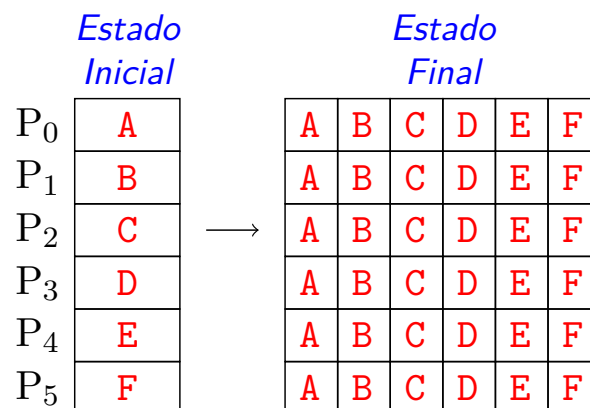
Versión asimétrica: MPI\_Gatherv

35

## Multi-Recogida

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf,  
recvcount, recvtype, comm)
```

Similar a la operación MPI\_Gather, pero todos los procesos obtienen el resultado



Versión asimétrica: MPI\_Allgatherv

36

## Todos a Todos

```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf,  
recvcount, recvtype, comm)
```

Es una extensión de la operación MPI\_Allgather, cada proceso envía unos datos distintos y recibe datos del resto

	<i>Estado Inicial</i>							<i>Estado Final</i>					
P <sub>0</sub>	A <sub>0</sub>	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>	A <sub>5</sub>	→	A <sub>0</sub>	B <sub>0</sub>	C <sub>0</sub>	D <sub>0</sub>	E <sub>0</sub>	F <sub>0</sub>
P <sub>1</sub>	B <sub>0</sub>	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>		A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	D <sub>1</sub>	E <sub>1</sub>	F <sub>1</sub>
P <sub>2</sub>	C <sub>0</sub>	C <sub>1</sub>	C <sub>2</sub>	C <sub>3</sub>	C <sub>4</sub>	C <sub>5</sub>		A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	D <sub>2</sub>	E <sub>2</sub>	F <sub>2</sub>
P <sub>3</sub>	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>		A <sub>3</sub>	B <sub>3</sub>	C <sub>3</sub>	D <sub>3</sub>	E <sub>3</sub>	F <sub>3</sub>
P <sub>4</sub>	E <sub>0</sub>	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>		A <sub>4</sub>	B <sub>4</sub>	C <sub>4</sub>	D <sub>4</sub>	E <sub>4</sub>	F <sub>4</sub>
P <sub>5</sub>	F <sub>0</sub>	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>	F <sub>4</sub>	F <sub>5</sub>		A <sub>5</sub>	B <sub>5</sub>	C <sub>5</sub>	D <sub>5</sub>	E <sub>5</sub>	F <sub>5</sub>

Versión asimétrica: MPI\_Alltoallv

37

## Reducción

```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root,  
comm)
```

Además de la comunicación se realiza una operación aritmética o lógica (suma, max, and, ..., o definida por el usuario)

El resultado final se devuelve en el proceso root

	<i>Estado Inicial</i>		<i>Estado Final</i>
P <sub>0</sub>	A		
P <sub>1</sub>	B		
P <sub>2</sub>	C	→	A+B+C+D+E+F
P <sub>3</sub>	D		
P <sub>4</sub>	E		
P <sub>5</sub>	F		

38

## Multi-Reducción

```
MPI_Allreduce(sendbuf, recvbuf, count, type, op, comm)
```

Extensión de MPI\_Reduce en que todos reciben el resultado

### Producto escalar de vectores

```
double par_dot(double local_x[], double local_y[], int local_n)
{
    double local_dot;
    double dot;

    local_dot = seq_dot(local_x, local_y, local_n);
    MPI_Allreduce(&local_dot, &dot, 1, MPI_DOUBLE,
                 MPI_SUM, MPI_COMM_WORLD);
    return dot;
}
```

39

## Prefijación

```
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm)
```

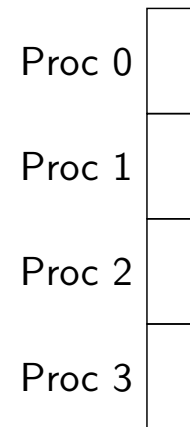
Extensión de las operaciones de reducción en que cada proceso recibe el resultado del procesamiento de los elementos de los procesos desde el 0 hasta él mismo

	<i>Estado Inicial</i>		<i>Estado Final</i>
P <sub>0</sub>	A	→	A
P <sub>1</sub>	B		A+B
P <sub>2</sub>	C		A+B+C
P <sub>3</sub>	D		A+B+C+D
P <sub>4</sub>	E		A+B+C+D+E
P <sub>5</sub>	F		A+B+C+D+E+F

40

## Ejemplo de Prefijación

Dado un vector de longitud  $N$ , distribuido entre los procesos, donde cada proceso tiene  $n_{\text{local}}$  elementos consecutivos del vector, se quiere obtener la posición inicial del subvector local



### Cálculo del índice inicial de un vector paralelo

```
int rstart, nlocal, N;  
  
calcula_nlocal(N,&nlocal);    /* por ejemplo, nlocal=N/p */  
MPI_Scan(&nlocal,&rstart,1,MPI_INT,MPI_SUM,comm);  
rstart -= nlocal;
```

41

## Apartado 4

### Otras Funcionalidades

- Tipos de Datos Derivados

42

## Tipos de Datos Básicos

Los tipos de datos básicos en lenguaje C son los siguientes:

<code>MPI_CHAR</code>	<code>signed char</code>
<code>MPI_SHORT</code>	<code>signed short int</code>
<code>MPI_INT</code>	<code>signed int</code>
<code>MPI_LONG</code>	<code>signed long int</code>
<code>MPI_UNSIGNED_CHAR</code>	<code>unsigned char</code>
<code>MPI_UNSIGNED_SHORT</code>	<code>unsigned short int</code>
<code>MPI_UNSIGNED</code>	<code>unsigned int</code>
<code>MPI_UNSIGNED_LONG</code>	<code>unsigned long int</code>
<code>MPI_FLOAT</code>	<code>float</code>
<code>MPI_DOUBLE</code>	<code>double</code>
<code>MPI_LONG_DOUBLE</code>	<code>long double</code>

- Para Fortran existen definiciones similares
- Además de los anteriores, están los tipos especiales `MPI_BYTE` y `MPI_PACKED`

43

## Datos Múltiples

Se permite el envío/recepción de múltiples datos:

- El emisor indica el número de datos a enviar en el argumento `count`
- El mensaje lo componen los `count` elementos **contiguos en memoria**
- En el receptor, el argumento `count` indica el tamaño del buffer – para saber el tamaño del mensaje:

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

Este sistema no sirve para:

- Componer un mensaje con varios datos de diferente tipo
- Enviar datos del mismo tipo pero que no estén contiguos en memoria

44

## Tipos de Datos Derivados

En MPI se permite definir **tipos nuevos** a partir de otros tipos

El funcionamiento se basa en las siguientes fases:

- 1 El programador define el nuevo tipo, indicando
  - Los tipos de los diferentes elementos que lo componen
  - El número de elementos de cada tipo
  - Los desplazamientos relativos de cada elemento
- 2 Se registra como un nuevo tipo de datos MPI (`commit`)
- 3 A partir de entonces, se puede usar para crear mensajes como si fuera un tipo de datos básico
- 4 Cuando no se va a usar más, el tipo se destruye (`free`)

---

Ventajas:

- Simplifica la programación cuando se repite muchas veces
- No hay copia intermedia, se compacta sólo en el momento del envío

45

## Tipos de Datos Derivados Regulares

```
MPI_Type_vector(count, length, stride, type, newtype)
```

Crea un tipo de datos homogéneo y regular a partir de elementos de un *array* equiespaciados

- 1 De cuántos bloques se compone (`count`)
- 2 De qué longitud son los bloques (`length`)
- 3 Qué separación hay entre un elemento de un bloque y el mismo elemento del siguiente bloque (`stride`)
- 4 De qué tipo son los elementos individuales (`type`)

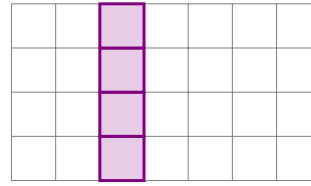
Constructores relacionados:

- `MPI_Type_contiguous`: elementos contiguos
- `MPI_Type_indexed`: longitud y desplazamiento variable

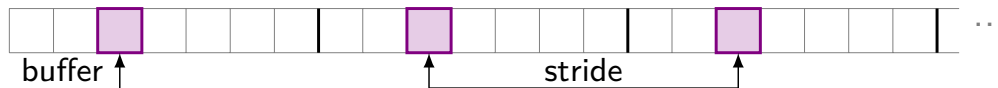
46

## Tipos de Datos Derivados Regulares: Ejemplo

Queremos enviar una *columna* de una matriz A[4][7]



En C, los *arrays* bidimensionales se almacenan por filas



```
double A[4][7];
MPI_Type_vector(4, 1, 7, MPI_DOUBLE, &columna);
MPI_Type_commit(&columna);
if (my_rank == 0) {          /* envía la 3ª columna */
    MPI_Send(&A[0][2], 1, columna, 1, 0, comm);
} else {
    MPI_Recv(&A[0][2], 1, columna, 0, 0, comm, &status);
}
```

47

## Tipos de Datos Derivados Irregulares

`MPI_Type_struct(count, lens, displs, types, newtype)`

Crea un tipo de datos heterogéneo (p.e. un struct de C)

```
struct {
    char c[5];
    double x,y,z;
} miestruc;
MPI_Datatype types[2] = {MPI_CHAR,MPI_DOUBLE}, newtype;
int lengths[2] = { 5, 1 }; /* solo queremos enviar c y z */
MPI_Aint ad1,ad2,ad3,displs[2];

MPI_Get_address(&miestruc, &ad1);
MPI_Get_address(&miestruc.c[0], &ad2);
MPI_Get_address(&miestruc.z, &ad3);
displs[0] = ad2 - ad1;
displs[1] = ad3 - ad1;
MPI_Type_struct(2, lengths, displs, types, &newtype);
MPI_Type_commit(&newtype);
```

48