

S3. Programación con MPI

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

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Curso 2024/25



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

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

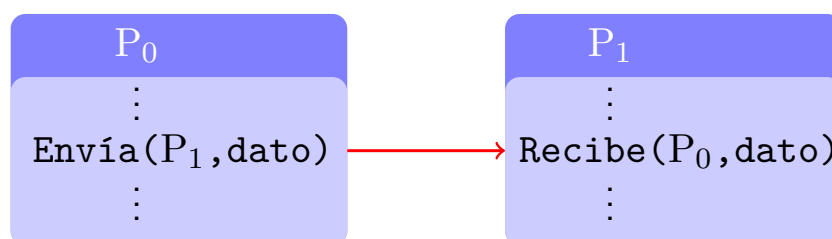
Conceptos Básicos

- 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

<https://www.mpi-forum.org>

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)
- MPI-4.0 (2020), última actualización MPI-4.1 (2021)

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)

El estándar especifica interfaz para C y Fortran

Hay muchas **implementaciones** disponibles:

- Propietarias: IBM, Cray, SGI, ...
- MPICH (www.mpich.org)
- Open MPI (www.open-mpi.org)
- MVAPICH (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, 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

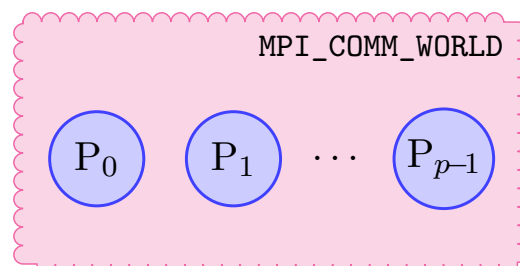
La ejecución de una aplicación suele hacerse con

```
mpixec -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

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. 43)

Nota: pasar `MPI_STATUS_IGNORE` si no se requiere

13

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

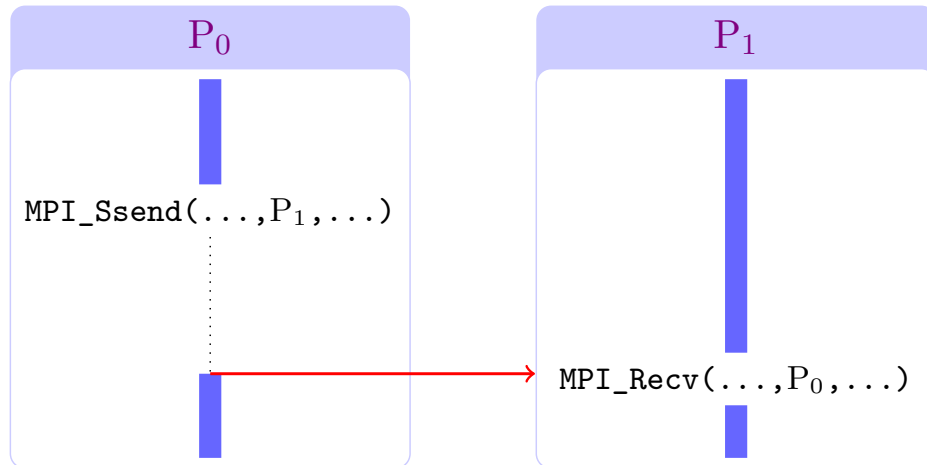
Para cada modo, existen primitivas bloqueantes y no bloqueantes

14

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



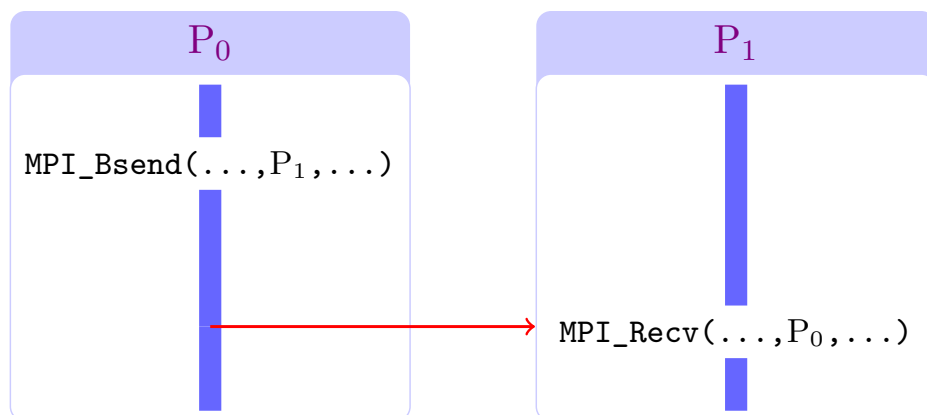
- Ineficiente: el emisor queda bloqueado sin hacer nada útil

15

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)

16

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

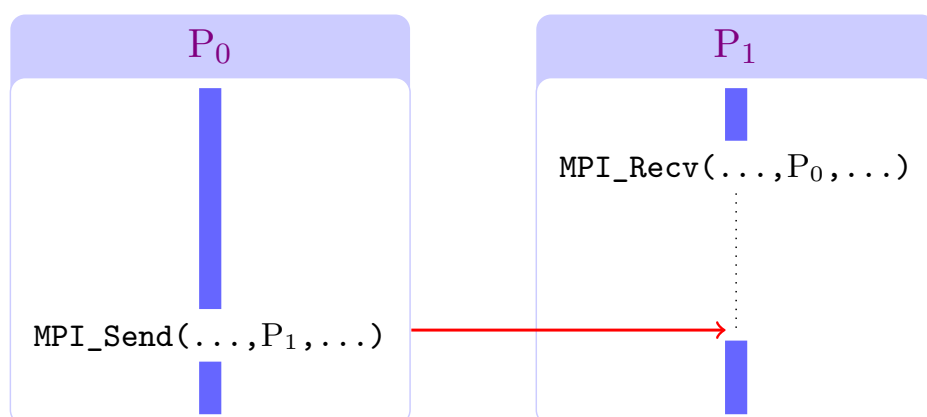
- Los mensajes cortos son enviados generalmente con MPI_Bsend
- Los mensajes largos son enviados generalmente con MPI_Ssend

17

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

18

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

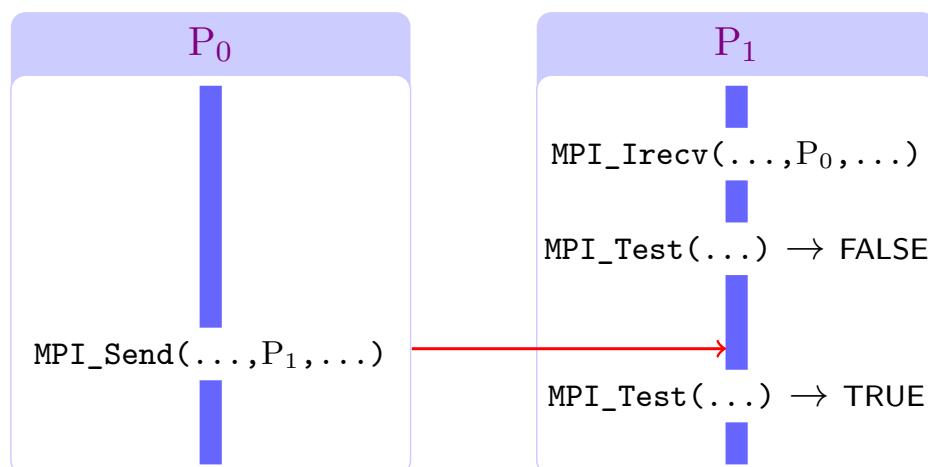
19

Recepción No Bloqueante

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

Se inicia la recepción, pero el receptor no se bloquea

- Se sustituye el argumento stat por req
- Es necesario comprobar después si el mensaje ha llegado



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

20

Operaciones Combinadas

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

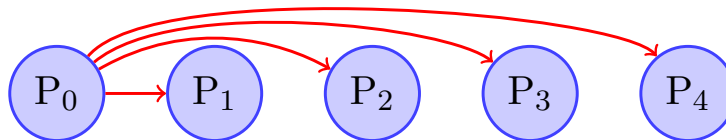
Realiza una operación de envío y recepción al mismo tiempo (no necesariamente con el mismo proceso)

```
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**

21

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) {  
    read_value(&val);    /* 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);  
}
```

22

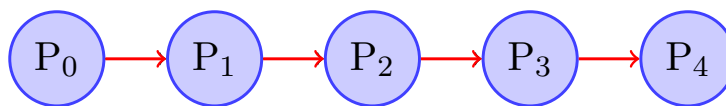
Ejemplo – Desplazamiento en Malla 1-D (1)

Cada proceso 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



23

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

24

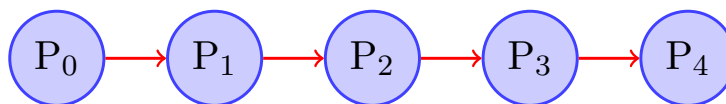
Ejemplo – Desplazamiento en Malla 1-D (3)

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;
}
```



25

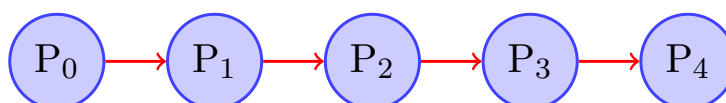
Ejemplo – Desplazamiento en Malla 1-D (4)

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;

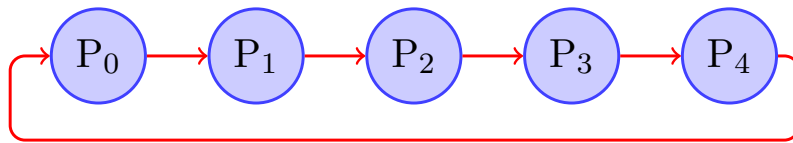
MPI_Sendrecv_replace(&val, 1, MPI_DOUBLE, next, 0, prev, 0,
                    comm, &status);
```



26

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

27

Apartado 3

Comunicación Colectiva

- Sincronización
- Difusión
- Reparto
- Reducción

28

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

29

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();

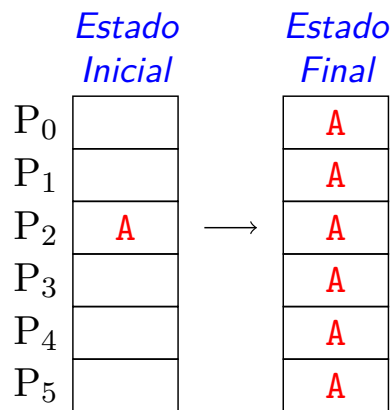
if (!rank) printf("Tiempo transcurrido: %f s.\n", t2-t1);
```

30

Difusión

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

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

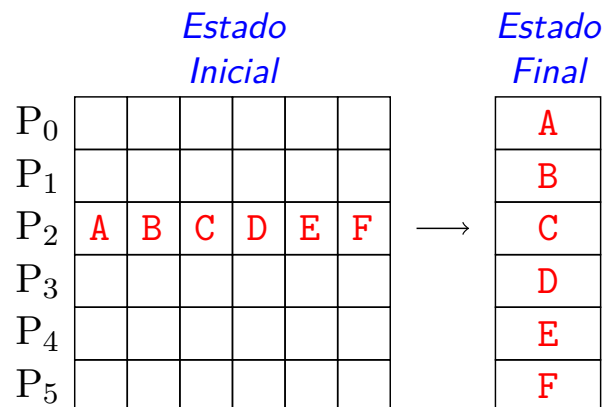


31

Reparto

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

El proceso root distribuye una serie de fragmentos consecutivos del buffer al resto de procesos (incluyendo él mismo)



Versión asimétrica: MPI_Scatterv

32

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);

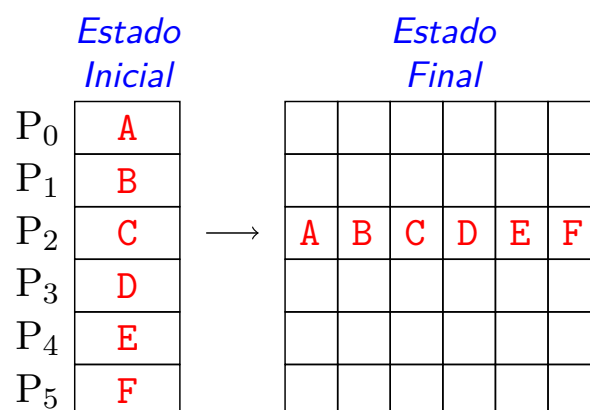
    MPI_Finalize();
    return 0;
}
```

33

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

34

Multi-Recogida

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

Similar a la operación MPI_Gather, pero todos los procesos obtienen el resultado

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

Versión asimétrica: MPI_Allgatherv

35

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 ₀	A ₀ A ₁ A ₂ A ₃ A ₄ A ₅	→	A ₀ B ₀ C ₀ D ₀ E ₀ F ₀
P ₁	B ₀ B ₁ B ₂ B ₃ B ₄ B ₅		A ₁ B ₁ C ₁ D ₁ E ₁ F ₁
P ₂	C ₀ C ₁ C ₂ C ₃ C ₄ C ₅		A ₂ B ₂ C ₂ D ₂ E ₂ F ₂
P ₃	D ₀ D ₁ D ₂ D ₃ D ₄ D ₅		A ₃ B ₃ C ₃ D ₃ E ₃ F ₃
P ₄	E ₀ E ₁ E ₂ E ₃ E ₄ E ₅		A ₄ B ₄ C ₄ D ₄ E ₄ F ₄
P ₅	F ₀ F ₁ F ₂ F ₃ F ₄ F ₅		A ₅ B ₅ C ₅ D ₅ E ₅ F ₅

Versión asimétrica: MPI_Alltoallv

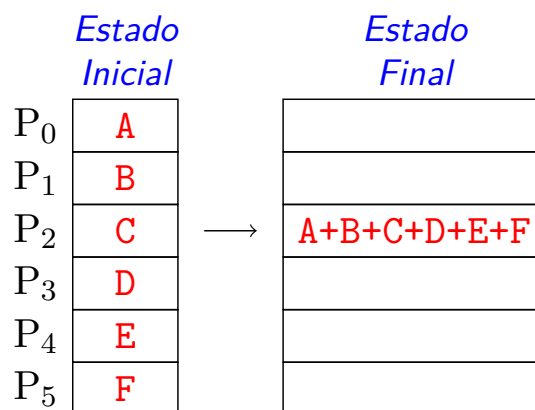
36

Reducción

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

Similar a MPI_Gather, pero en lugar de concatenación, se realiza una operación aritmética o lógica (suma, max, and, ..., o definida por el usuario)

El resultado final se devuelve en el proceso root



37

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;
}
```

38

Prefijación

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

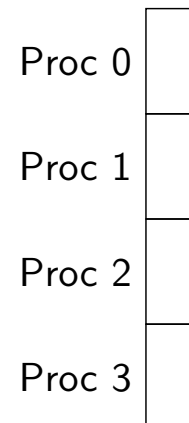
Extensión de las operaciones de reducción en que cada proceso recibe el resultado del procesamiento de los elementos de los procesos desde el 0 hasta él mismo

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

39

Ejemplo de Prefijación

Dado un vector de longitud N , distribuido entre los procesos, donde cada proceso tiene n_{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;
```

40

Apartado 4

Otras Funcionalidades

■ Tipos de Datos Derivados

41

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`

42

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

43

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

44

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`)

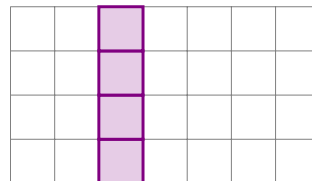
Constructores relacionados:

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

45

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

46