TPP – Tecnología de la Programación Paralela

Máster en Computación Paralela y Distribuida

3. Paradigma de Paso de Mensajes

José E. Román[1] Paulo B. Vasconcelos[2]

[1]Universitat Politècnica de València

[2]Universidade do Porto, Portugal

Curso 2015/16

Contenido

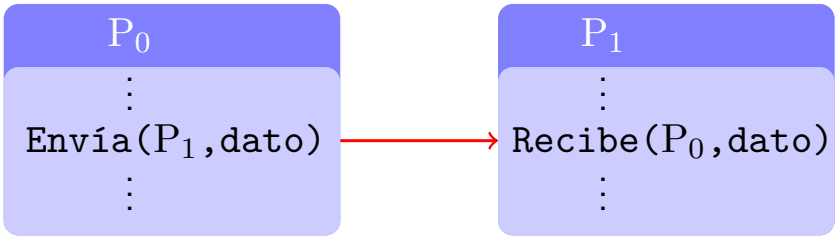
Apartado 1

Fundamentos

- Modelo de Paso de Mensajes
- Comunicación Punto a Punto
- Comunicación Colectiva

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

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)

Antecedentes:

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

Modelo de Programación

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

Ejemplo

```
#include <mpi.h>
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();
}
```

- Es obligatorio llamar a `MPI_Init` y `MPI_Finalize`
- Una vez inicializado, se pueden realizar diferentes operaciones

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
- Operaciones de alto nivel
  Grupos, contextos, comunicadores, atributos, topologías
- Gestión de datos
  Tipos de datos derivados (p.e. datos no contiguos en memoria)
- 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

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)
```

Modelo de Ejecución

El modelo de ejecución de MPI sigue un esquema de creación (*spawn*) simultánea de procesos al lanzar la aplicación

La ejecución de una aplicación suele hacerse con

```
mpirun -np p programa [opciones]
```

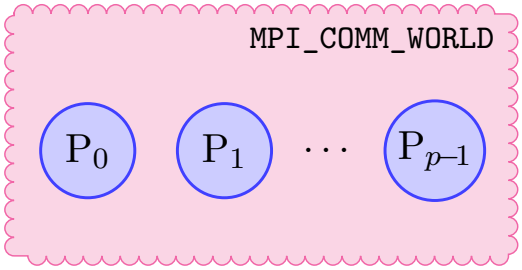
`mpirun` no es estándar; MPI-2 unifica la sintaxis (`mpiexec`)

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



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* o memoria intermedia donde está almacenada la información
- El número de elementos que componen el mensaje
- El tipo de datos de los elementos (p.e. `MPI_INT`)

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

Nota: pasar `MPI_STATUS_IGNORE` si no se requiere

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 preparado
- Modo de envío estándar

El modo estándar es el más utilizado

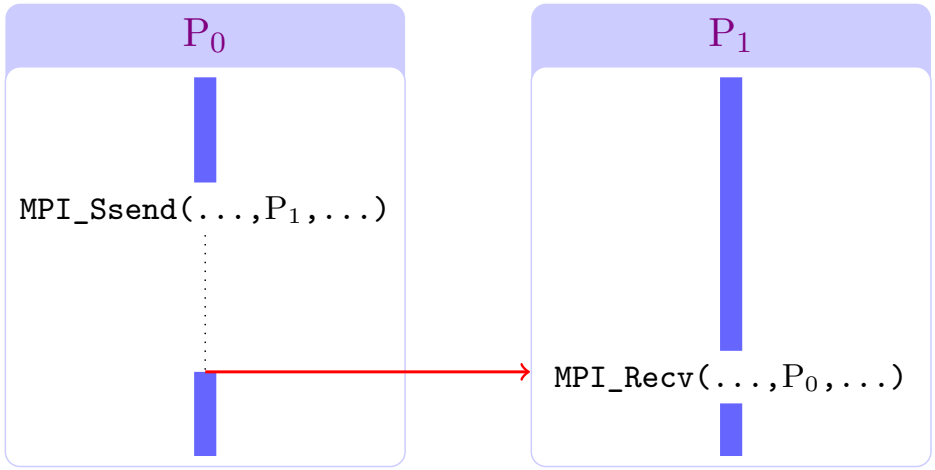
El resto de modos pueden ser útiles para obtener mejores prestaciones o mayor robustez

Para cada modo, existen primitivas bloqueantes y no bloqueantes

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

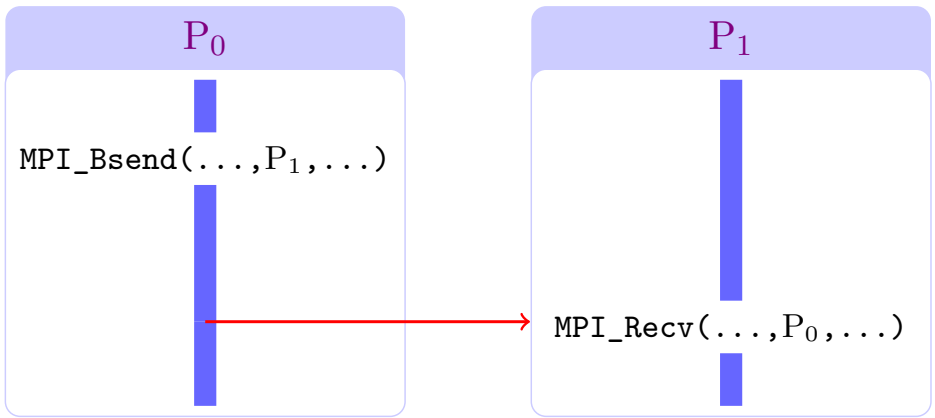


- Ineficiente: el emisor queda bloqueado sin hacer nada útil

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 el buffer (`MPI_buffer_attach`)

Modo de Envío Estándar

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

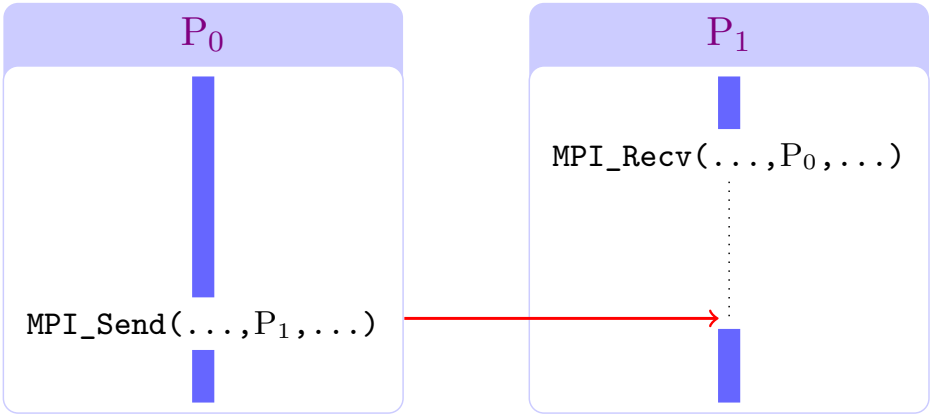
Garantiza el funcionamiento en todo tipo de sistemas ya que evita problemas de almacenamiento

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

Recepción Estándar

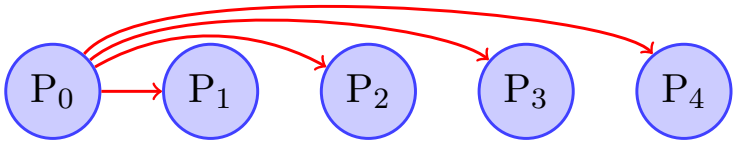
```
MPI_Recv(buf, count, datatype, src, tag, comm, stat)
```

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

Ejemplo – Difusión



Difusión de un valor numérico desde P_0

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

MPI_Init(&argc, &argv);
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);
}
```


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;
```

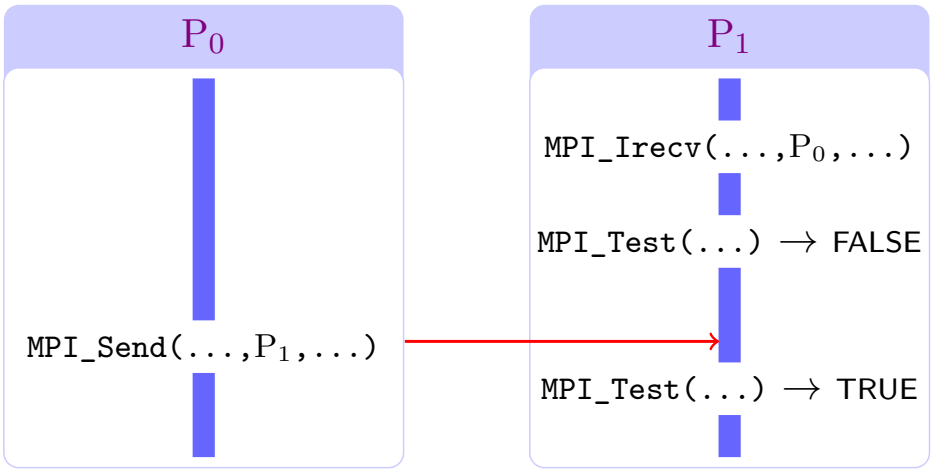
- Solapamiento de comunicación y cálculo sin copia extra
- Inconveniente: programación más difícil

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

Operaciones de Comunicación Colectiva

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

Operaciones disponibles:

- Sincronización (*Barrier*)
- Difusión (*Bcast*)
- Reparto (*Scatter*)
- Recogida (*Gather*)
- Multi-recogida (*Allgather*)
- Todos a todos (*Alltoall*)
- Reducción (*Reduce*)
- Prefijación (*Scan*)

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

Sincronización

```
MPI_Barrier(comm)
```

Operación pura de sincronización

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

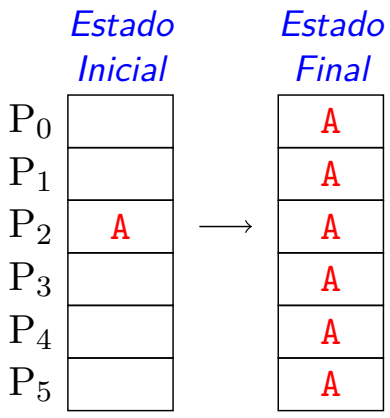
Ejemplo – medición de tiempos

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

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


Difusión

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

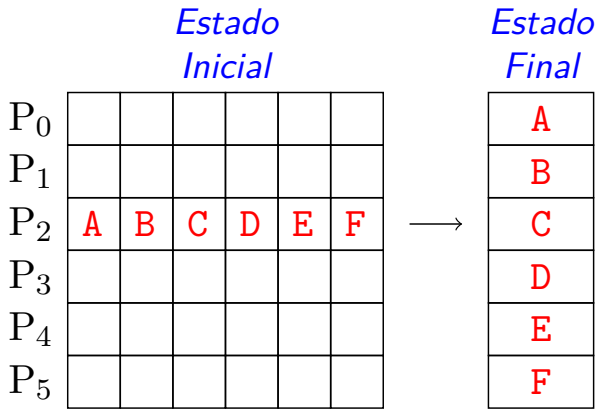
El proceso `root` difunde al resto de procesos el mensaje definido por los 3 primeros argumentos



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`

Reparto: Ejemplo

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

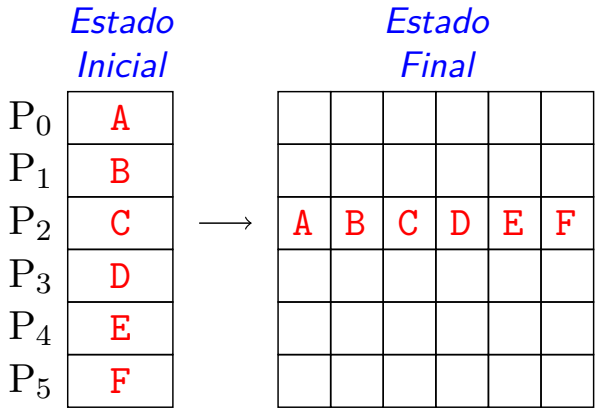
Ejemplo de reparto

```
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();
}
```

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

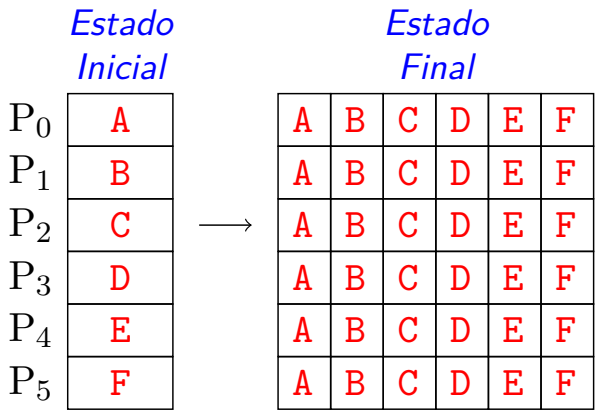


Versión asimétrica: `MPI_Gatherv`

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`

Todos a Todos

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

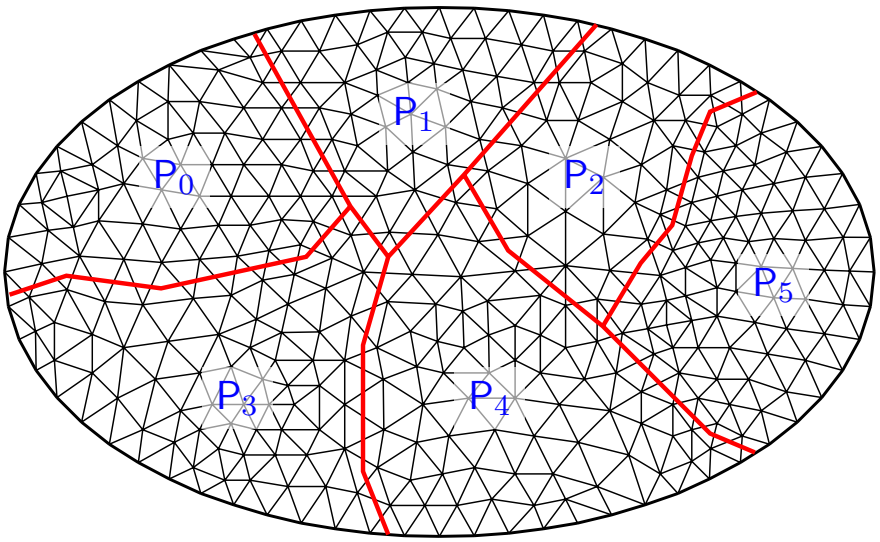
Es una extensión de la operación `MPI_Allgather`, cada proceso envía unos datos distintos y recibe datos del resto

	Estado Inicial							*Estado Final*					
P_0	A_0	A_1	A_2	A_3	A_4	A_5	$\longrightarrow$	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`

Aplicación de `MPI_Alltoallv`

En aplicaciones basadas en mallas



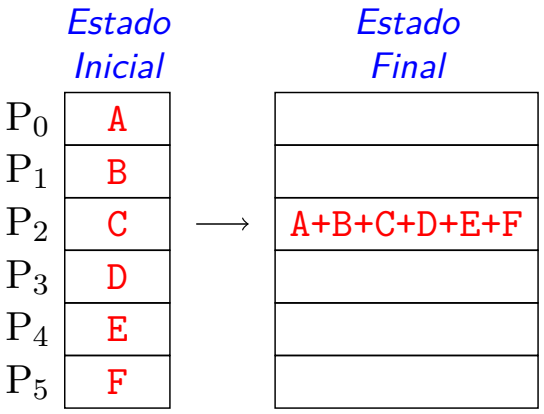
Cada proceso tiene que enviar las incógnitas que están en la interfaz a los procesos vecinos correspondientes, y recibir a su vez las de ellos

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`



Multi-Reducción

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

Extensión de `MPI_Reduce` en que todos reciben el resultado

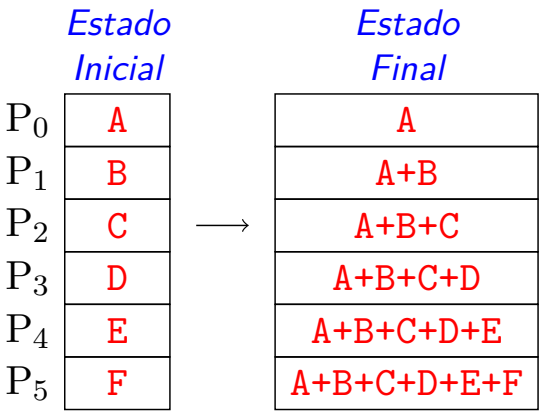
Producto escalar de vectores

```
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;
}
```

Prefijación

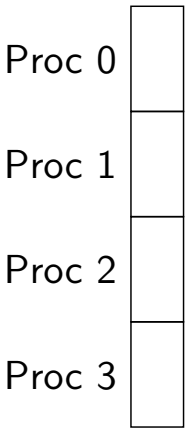
```
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm)
```

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



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;
```

Apartado 2

Funcionalidad Avanzada

- Tipos de Datos Derivados
- Comunicación Persistente
- Comunicadores y Topologías
- Creación de Procesos
- Comunicación Unilateral
- Entrada/Salida

Tipos de Datos Básicos

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

`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`

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

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

Mensajes Empaquetados

```
MPI_Pack(data, count, type, buf, size, pos, comm)
MPI_Unpack(buf, size, pos, data, count, type, comm)
```

Para enviar conjuntamente datos de distinto tipo

Mensaje empaquetado

```
if (my_rank == 0) {
  pos = 0;
  MPI_Pack(a_ptr, 1, MPI_FLOAT, buffer, 100, &pos, comm);
  MPI_Pack(n_ptr, 1, MPI_INT, buffer, 100, &pos, comm);
  MPI_Bcast(buffer, pos, MPI_PACKED, 0, comm);
} else {
  MPI_Bcast(buffer, 100, MPI_PACKED, 0, comm);
  pos = 0;
  MPI_Unpack(buffer, 100, &pos, a_ptr, 1, MPI_FLOAT, comm);
  MPI_Unpack(buffer, 100, &pos, n_ptr, 1, MPI_INT, comm);
}
```

Inconveniente: cada llamada tiene un *overhead*

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

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)

Creación de un tipo `struct`

```
struct {
  char c[5];
  double x,y,z;
} miestruc;
MPI_Datatype types[2] = {MPI_CHAR,MPI_DOUBLE}, newtype;
int lengths[2] = { 5, 1 };
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);
```

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

Creación de un tipo `vector`

```
double A[10][10];
MPI_Type_vector(10, 1, 10, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
if (my_rank == 0) {          /* envía la 3ª columna */
  MPI_Send(&A[0][2], 1, newtype, 1, 0, comm);
} else {
  MPI_Recv(&A[0][2], 1, newtype, 0, 0, comm, &status);
}
```

Constructores relacionados:

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

Comunicación Persistente

Las comunicaciones persistentes pueden reducir el *overhead* de las comunicaciones en programas que repiten muchas veces las mismas primitivas punto a punto con idénticos argumentos

Pasos:

1. Crear las peticiones persistentes indicando el *buffer* y los otros argumentos; existe una primitiva por cada tipo de comunicación punto a punto: `MPI_Send_init`, `MPI_Bsend_init`, `MPI_Recv_init`, etc.
2. Iniciar la comunicación, de una en una (`MPI_Start`) o varias a la vez (`MPI_Startall`)
3. Esperar la finalización, ya que estas operaciones son no-bloqueantes; usar `MPI_Wait`, `MPI_Test` o similar
4. Destruir las peticiones persistentes, `MPI_Request_free`

Comunicación Persistente - Ejemplo

Envío y recepción dentro de un bucle

```
MPI_Status stats[2];
MPI_Request reqs[2];
...

MPI_Recv_init(rbuff, n, MPI_CHAR, src, tag, comm, &reqs[0]);
MPI_Send_init(sbuff, n, MPI_CHAR, dest, tag, comm, &reqs[1]);

for (i=0; i<REPS; i++) {
  ...
  MPI_Startall(2, reqs);
  ...
  MPI_Waitall(2, reqs, stats);
  ...
}

MPI_Request_free(&reqs[0]);
MPI_Request_free(&reqs[1]);
```


Comunicadores

Un comunicador es una abstracción que engloba los siguientes conceptos:

- *Grupo*: conjunto ordenado de p procesos, identificados por un entero entre 0 y $p-1$
- *Contexto*: para evitar interferencias entre mensajes distintos

---

En algoritmos paralelos complicados, es conveniente:

- Poder crear nuevos comunicadores para restringir la comunicación a un subconjunto de los procesos
- Poder asociar información adicional a cada proceso, además del identificador de proceso

Creación de Comunicadores

Una posible forma de crear un comunicador es construir previamente un grupo

Creación de un comunicador a partir de un grupo

```
q = (int) sqrt((double) p);

ranks = (int*) malloc(q*sizeof(int));
for (proc = 0; proc < q; proc++)
  ranks[proc] = proc;

MPI_Comm_group(MPI_COMM_WORLD, &group_world);
MPI_Group_incl(group_world, q, ranks, &first_row_group);
MPI_Comm_create(MPI_COMM_WORLD, first_row_group,
                &first_row_comm);
```

Hay operaciones que permiten crear comunicadores nuevos directamente a partir de otro comunicador: `MPI_Comm_split`

Atributos

MPI permite asociar información a los comunicadores: atributos

```
MPI_Attr_put(comm, keyval, attribute_val)
MPI_Attr_get(comm, keyval, attribute_val, flag)
```

Estas operaciones son locales y, por tanto, el valor del atributo es distinto en cada proceso

Se permite un mecanismo de funciones *callback* para la creación y destrucción de atributos

---

La principal utilidad de los atributos es almacenar información acerca de la "posición relativa" de unos procesos respecto a otros: topologías

- Las topologías más comunes tienen soporte adicional

Topologías de Procesos

Las topologías de procesos son un mecanismo para asociar diferentes esquemas de direccionamiento de los procesos

- Ejemplo: *(fila, columna)* en procesos dispuestos en 2-D

Las topologías MPI son virtuales

- No tienen relación directa con la topología física
- Sin embargo, pueden ayudar al mapeo físico

Se implementan mediante atributos

Tipos de topologías:

- Grafo
- Cartesiana (o de malla)

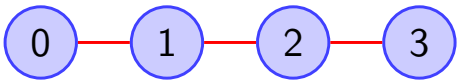
La topología cartesiana es un caso particular (muy utilizado) de la topología de grafo

Topología Cartesiana

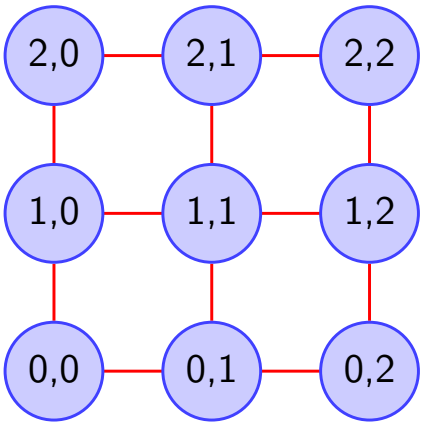
Los procesos se disponen en una malla y se conectan de forma lógica con los vecinos

La conexión puede ser periódica (malla cerrada) o no periódica (malla abierta)

Anillo/Malla 1-D:

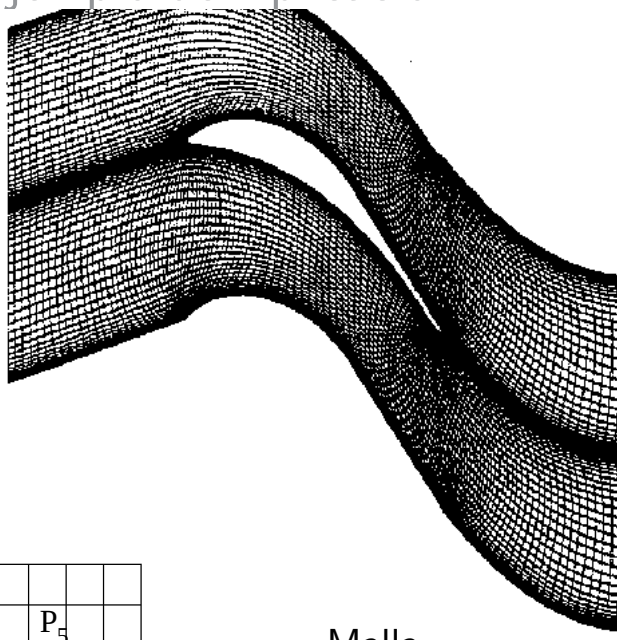


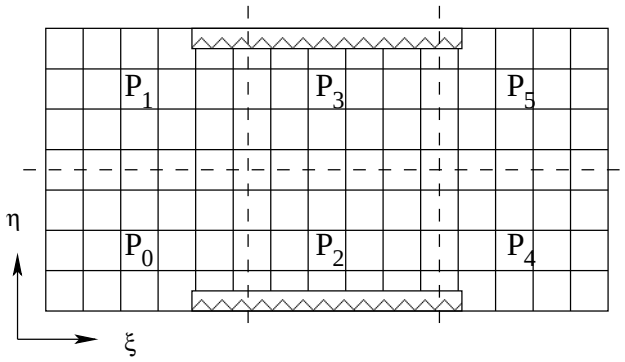
Toro/Malla 2-D:



Topología Cartesiana – Ejemplo de Aplicación

Estudio del flujo de aire alrededor de los álabes de una turbina (ecuaciones de Navier-Stokes)





Malla (parcialmente) periódica en la dimensión vertical

Topología Cartesiana – Uso (1)

A partir de un comunicador se crea otro con la info de la malla

```
MPI_Cart_create(comm, dims, sz, period, reord, gcomm)
```

Ejemplo de topología cartesiana

```
q = (int) sqrt((double) p);
reorder = 0;
sizes[0] = sizes[1] = q;
period[0] = period[1] = 1;
MPI_Cart_create(comm, 2, sizes, period, reorder, &gcomm);

MPI_Comm_rank(gcomm, &my_grid_rank);
MPI_Cart_coords(gcomm, my_grid_rank, 2, coordinates);

MPI_Cart_rank(gcomm, coordinates, &grid_rank);
```

Posteriormente, se pueden crear comunicadores para las filas o columnas de la malla con `MPI_Cart_sub`

Topología Cartesiana – Uso (2)

Hay utilidades para obtener el rango de los procesos vecinos

```
MPI_Cart_shift(comm, direction, displ, source, dest)
```

Ejemplo de `MPI_Cart_shift`

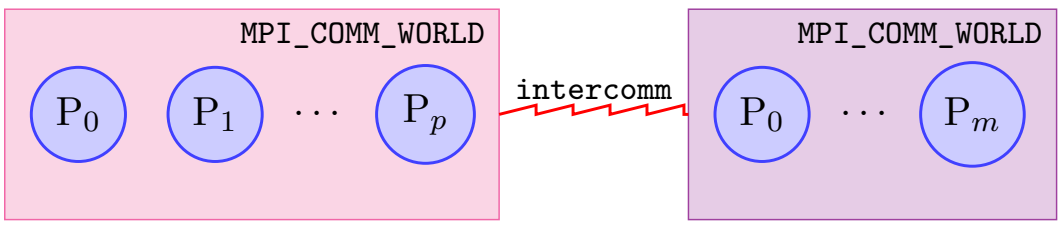
```
dims[0] = nrow;
dims[1] = ncol;
period[0] = 1;      /* ciclico en esta direccion */
period[1] = 0;      /* no ciclico en esta direccion */
MPI_Cart_create(comm, 2, dims, period, reorder, &comm2D);
MPI_Comm_rank(comm2D, &me);

index =  0;    /* desplazar en la primera dimension */
displ =  1;    /* desplazar en 1 */

MPI_Cart_shift(comm2D, index, displ, &neig1, &neig2);
```


Creación de Procesos

El modelo de procesos de MPI-2 es dinámico, permitiendo crear nuevos procesos

```
MPI_Comm_spawn(command, argv, maxprocs, info, root, comm,
                    intercomm, errcodes)
```

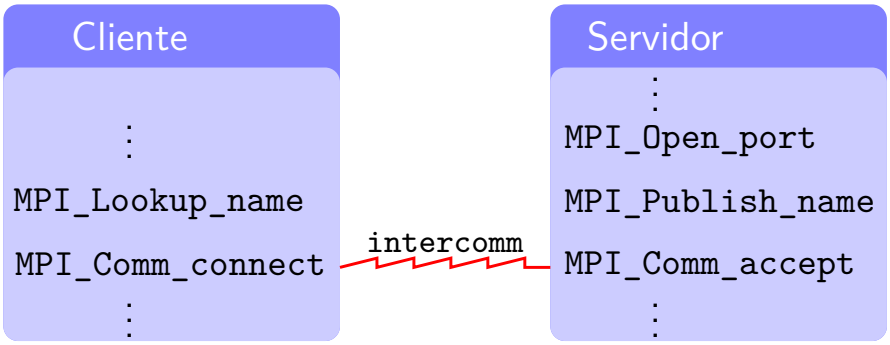
- Se crean `maxprocs` procesos hijo
- Los procesos hijo tienen su propio `MPI_COMM_WORLD`
- Se crea un inter-comunicador que engloba a los procesos padres e hijos



Comunicación Cliente-Servidor

El modelo de procesos de MPI-2 permite establecer una comunicación de tipo cliente-servidor entre dos grupos a través de un inter-comunicador

- Se usa el modelo de puertos
- Existe un mecanismo de directorio de servicios



Ejemplo de uso: programa de visualización que se conecta a un código de simulación

Comunicación Unilateral

MPI-2 añade mecanismos de comunicación unilateral (*one-sided*)

- Modelo RMA: *Remote Memory Access*
- Permite a un proceso especificar todos los parámetros de la comunicación, tanto del lado emisor como del receptor
- Facilita la programación en algunas aplicaciones

Los miembros de un grupo declaran una "ventana" para RMA

```
MPI_Win_create(base, size, disp_unit, info, comm, win)
```

Se separan los aspectos de comunicación y sincronización

- Comunicación: `MPI_Put`, `MPI_Get`, `MPI_Accumulate`
- Sincronización: `MPI_Fence` y otras

Comunicación Unilateral - Ejemplo

Ejemplo de comunicación unilateral

```
if (myid == 0)
  MPI_Win_create(&pi, sizeof(double), 1, MPI_INFO_NULL, comm, &win);
else
  MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, comm, &win);

h   = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
  x = h * ((double)i - 0.5);
  sum += (4.0 / (1.0 + x*x));
}
mypi = h * sum;

MPI_Win_fence(0, win);
MPI_Accumulate(&mypi, 1, MPI_DOUBLE, 0, 0, 1, MPI_DOUBLE,
               MPI_SUM, win);
MPI_Win_fence(0, win);
MPI_Win_free(&win);
```


Entrada/Salida

El modelo de fichero POSIX es muy limitado para E/S paralela

Para garantizar eficiencia y escalabilidad, se necesita:

- Soporte para particionado de ficheros entre procesos
- Operaciones colectivas para trasferencia memoria-fichero

Además, este diseño permite optimizaciones adicionales:

- Uso de E/S asíncrona
- Accesos entrelazados a los datos
- Control de la ubicación física en el dispositivo

MPI-2 añade mecanismos de entrada/salida paralela

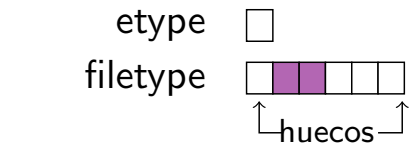
- Los ficheros se consideran colecciones ordenadas de items con tipo (no simples ristras de bytes)
- Se permite acceso secuencial y aleatorio a cualquier item
- Apertura colectiva de ficheros por un grupo de procesos

Entrada/Salida - Tipos de Datos

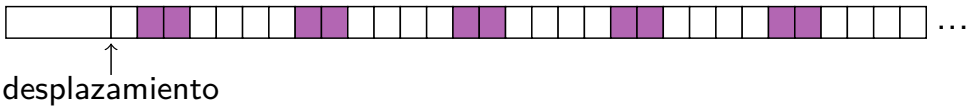
Filosofía: tipos de datos derivados para expresar el particionado

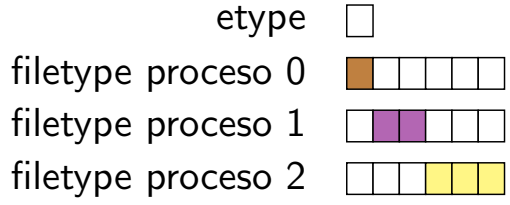
- **etype** Tipo elemental, unidad de acceso y posicionamiento
- **filetype** Define una plantilla de acceso al fichero y es la base del particionado (distinto en cada proceso)
- **displacement** Posición en bytes desde el inicio del fichero
- **view** Define los datos visibles actualmente en un fichero abierto (distinto en cada proceso)
- **offset** Posición en el fichero relativa al *view*

Entrada/Salida - Tipos de Datos



file consiste en repeticiones de filetype:





file consiste en repeticiones de filetype:

Entrada/Salida - Operaciones

Las operaciones de acceso a datos (lectura y escritura) se definen para diferente...

- Posicionamiento: offset explícito, puntero de fichero individual, puntero de fichero compartido
- Coordinación: colectiva o no colectiva
- Sincronismo: bloqueante, no bloqueante, dividido en fases *begin*/*end*

Tras la creación, los ficheros están asociados a una vista (*view*) por proceso

- Una vista define el conjunto de datos visibles y accesibles
- La vista puede ser modificada en cualquier momento

Entrada/Salida - Operaciones

Operaciones disponibles:

- Manipulación de ficheros: `MPI_File_open`, `MPI_File_close`
- Vistas: `MPI_File_set_view`, `MPI_File_get_view`
- Acceso con offset explícito: `MPI_File_read_at`, `MPI_File_read_at_all`
- Acceso con punteros de fichero individuales: `MPI_File_read`, `MPI_File_read_all`
- Acceso con puntero de fichero compartido: `MPI_File_read_shared`, `MPI_File_read_ordered`
- Consistencia: `MPI_File_sync`, `MPI_File_set_atomicity`

Variantes:

- Operaciones colectivas: funciones acabadas en `_all`
- Versiones no bloqueantes: `MPI_File_iread`, etc.

Entrada/Salida - Ejemplo

Entrada/Salida MPI a ficheros separados

```
for (i=0; i<BUFSIZE; i++)
  buf[i] = myrank * BUFSIZE + i;
sprintf(filename, "testfile.%d", myrank);
MPI_File_open(MPI_COMM_SELF, filename, MPI_MODE_WRONLY,
              MPI_INFO_NULL, &myfile);
MPI_File_write(myfile, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);
MPI_File_close(&myfile);
```

Entrada/Salida MPI a un fichero único

```
for (i=0; i<BUFSIZE; i++)
  buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile", MPI_MODE_WRONLY,
              MPI_INFO_NULL, &thefile);
MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int), MPI_INT,
              MPI_INT, "native", MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);
MPI_File_close(&thefile);
```