CMCP – Conceptos y Métodos de la Computación Paralela

Master en Computación en la Nube y de Altas Prestaciones

T4. Programación en Memoria Distribuida

J. E. Roman

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Curso 2024-2025

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

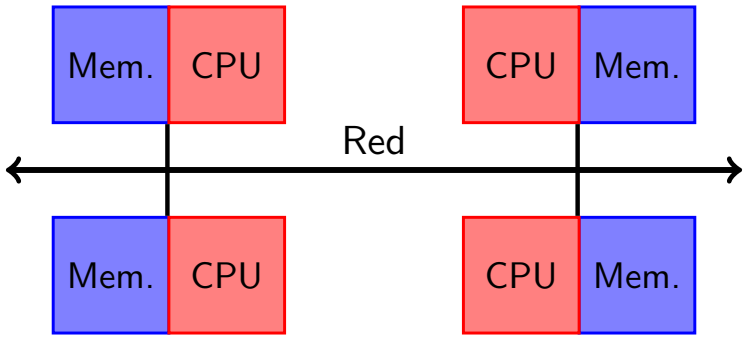
Contenido

Apartado 1

Arquitectura de Memoria Distribuida

Arquitecturas de Memoria Distribuida

Se requiere una red de comunicación para que los procesadores puedan acceder a la memoria no local



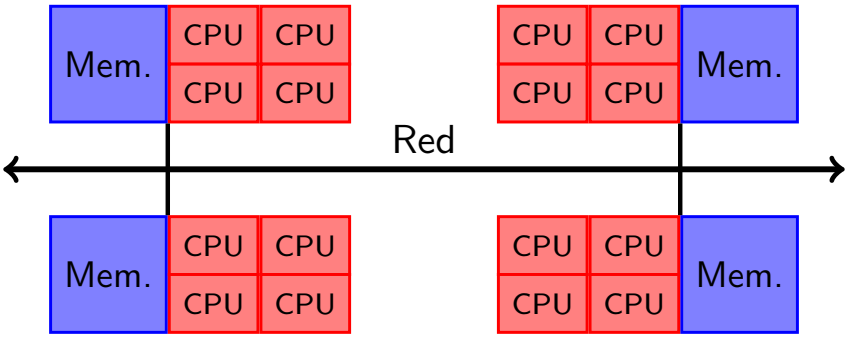
Características

- No existe el concepto de memoria global
- Procesadores independientes (no hay coherencia)
- El programador explicita el intercambio de datos

Ventajas: escalabilidad, precio; Desventajas: programación

Arq. Híbridas: *Distributed-Shared Memory*

Combinación de los dos modelos



Características

- Cada nodo es un multiprocesador (p.e. cc-NUMA)
- Comunicación para mover datos de un nodo a otro

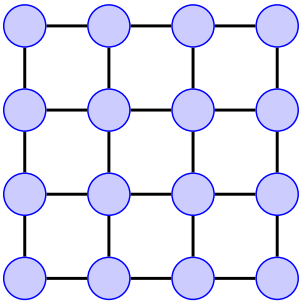
Actualmente, los supercomputadores suelen seguir este modelo (aunque cada vez más con multi-cores y GPUs)

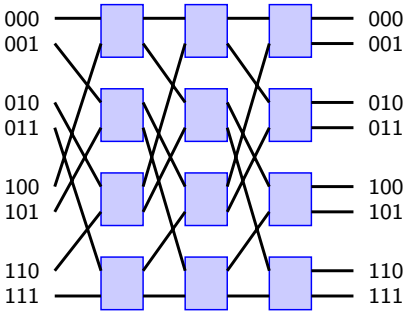
Redes de Interconexión

En todos los casos hace falta una red de interconexión

Redes estáticas: anillo, malla abierta, malla cerrada, toro 2D o 3D, hipercubo

Redes dinámicas: monoetapa, multietapa, *crossbar*





- Redes con latencia uniforme: poco escalables (coste)
- Redes con latencia no uniforme: más baratas, la latencia depende de la distancia

Redes de Interconexión Actuales

Redes de baja latencia, típicamente conmutadas (*switches*)

- Antecedentes: Myrinet, SCI, Quadrics
- Cercano a tecnología SAN, p.e. Fibre Channel
- InfiniBand: estándar impulsado por grupo de empresas
  - Diferentes productos compatibles, p.e. Intel Omni-Path
  - OpenFabrics Alliance desarrolla software open-source
- Ethernet: 10GbE, 40GbE, 100GbE

Es necesario reducir también la latencia software

- Técnicas RDMA (*remote direct memory access*)
- RoCE: *RDMA over Converged Ethernet*, usa UDP/IP
- iWARP es un protocolo similar sobre TCP/IP

Clusters

Un *cluster* es simplemente un conjunto de PCs o estaciones de trabajo conectados en red para computación paralela

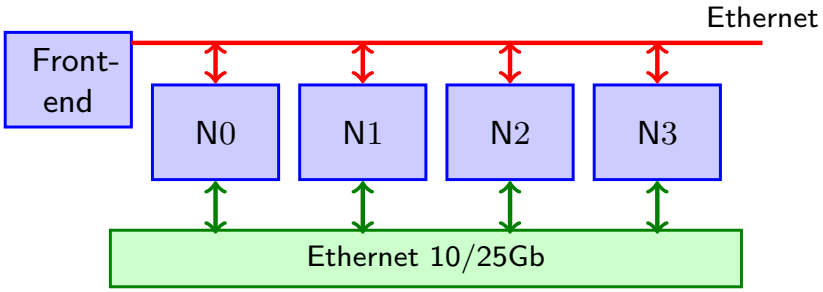
- El primero con PCs y Linux data de 1994 (Beowulf)

Actualmente, un cluster comercial se compone de

- Armario bastidor
- Un conjunto de nodos
  - Típicamente, dos procesadores multi-core, 1 disco
  - Opcionalmente, 1 GPU de gama alta
  - Formato compacto: 1U, 2U, *blades*
- Infraestructura de red
  - Típicamente, dos redes: ethernet y red de baja latencia
  - Redes de baja latencia: Infiniband, Myrinet, Quadrics, ...
  - Componentes: adaptador, conmutadores (*switches*), cables
- Nodo *front-end*

Cluster de Prácticas

Configuración hardware: 4 nodos



Cada nodo:

- 1 procesador AMD EPYC 7551P, 32 núcleos físicos (64 virtuales)
- 64GB de memoria RAM
- Disco SSD 240GB
- Ethernet 10/25Gb 2-port 622FLR -SFP28

Agregado: 4 proc, 256 núcleos, 256 GB

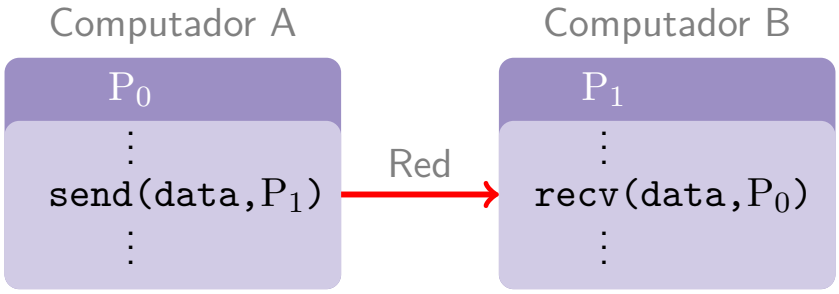
Apartado 2

Introducción a MPI

Modelo de Paso de Mensajes

Para programar computadores de memoria distribuida (clusters)

- Basado en procesos del SO (no en hilos)
- Espacio de memoria privado (sin variables compartidas)
- Intercambio de información mediante envío y recepción explícitos de mensajes
- Modelo más usado en computación de gran escala



MPI: Message Passing Interface

El Estándar MPI

`https://www.mpi-forum.org`

Especificación para programación paralela de paso de mensajes

- Implementado como una biblioteca (llamadas a función)
- El estándar especifica interfaz para C y Fortran
- Portable a cualquier plataforma paralela
- Mejora continua: versión actual es la 4.1

Hay muchas implementaciones disponibles:

- MPICH (`www.mpich.org`)
- Open MPI (`www.open-mpi.org`)
- MVAPICH (`mvapich.cse.ohio-state.edu`)
- Propietarias: Intel, Cray, NVIDIA, ...

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

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

Modelo de Ejecución

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

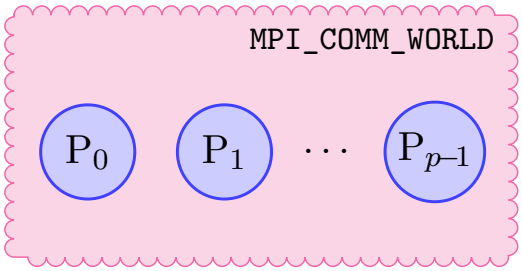
La ejecución de una aplicación suele hacerse con

```
mpiexec -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



Modelo de Programación – Comunicadores

Comunicador: una abstracción que engloba dos conceptos

- *Grupo*: conjunto de procesos
- *Contexto*: la comunicación en un comunicador no puede interferir con la comunicación en otro

Comunicadores predefinidos: `MPI_COMM_WORLD` (incluye todos los procesos creados con `mpiexec`) y `MPI_COMM_SELF` (incluye solo el proceso llamante)

Un comunicador agrupa a p procesos

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

Cada proceso tiene un identificador (rank) entre 0 y $p-1$

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

Apartado 3

Comunicación Punto a Punto

Operaciones Básicas de Envío/Recepción

La operación más común es la comunicación punto a punto

- Un proc. envía un mensaje (`send`) y otro lo recibe (`recv`)
- Cada `send` ha de tener un `recv` emparejado
- El mensaje es el contenido de una variable

```
/* Proceso 0 */
x = 10;
send(x,1);
x = 0;
```

```
/* Proceso 1 */
recv(y,0);
```

La operación `send` es segura desde el punto de vista semántico si se garantiza que el proceso 1 recibe el valor que tenía `x` antes del envío (10)

Existen diferentes modalidades de envío y recepción

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`)

Comunicación Punto a Punto – el Sobre

La comunicación se establece dentro del comunicador `comm`

- La fuente (`src`) y el destino (`dest`) se especifican mediante el identificador de proceso (*rank*)
- En la recepción se permite utilizar `src=MPI_ANY_SOURCE`

El argumento `tag` es un entero que puede usarse para distinguir mensajes de diferente tipo

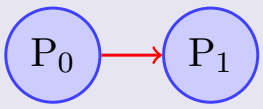
- En la recepción se permite utilizar `tag=MPI_ANY_TAG`

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

- Proceso emisor (`stat.MPI_SOURCE`), etiqueta (`stat.MPI_TAG`)
- Longitud del mensaje: en `MPI_Recv`, `count` es el tamaño del buffer disponible; la longitud real se puede obtener con `MPI_Get_count`
- Pasar `MPI_STATUS_IGNORE` si no se requiere

Ejemplo Punto a Punto

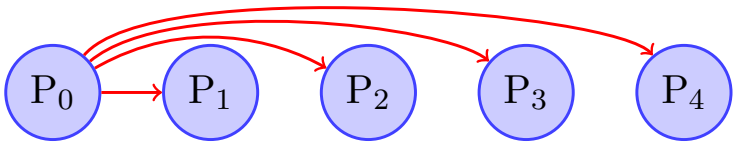
Enviar datos de P_0 a P_1




```
int rank, cnt;
double a[N], b[M];
MPI_Status stat;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  compute(a, N);      /* rellenar el array */
  MPI_Send(a, N, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
  MPI_Recv(b, M, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &stat);
  MPI_Get_count(&stat, MPI_DOUBLE, &cnt);
  process(b, cnt);      /* usar los datos recibidos */
}
```

- Si `N` $>$ `M` la operación `MPI_Recv` fallará
- En este ejemplo, `cnt` será siempre igual a `N`

Ejemplo Punto a Punto – Difusión



Difusión de un valor numérico desde P_0

```
#define TAG 111
double val;
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,TAG,MPI_COMM_WORLD);
} else {
  MPI_Recv(&val,1,MPI_DOUBLE,0,TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}
```

Modos de Envío Punto a Punto

Hay varias primitivas de envío (con los mismos argumentos):

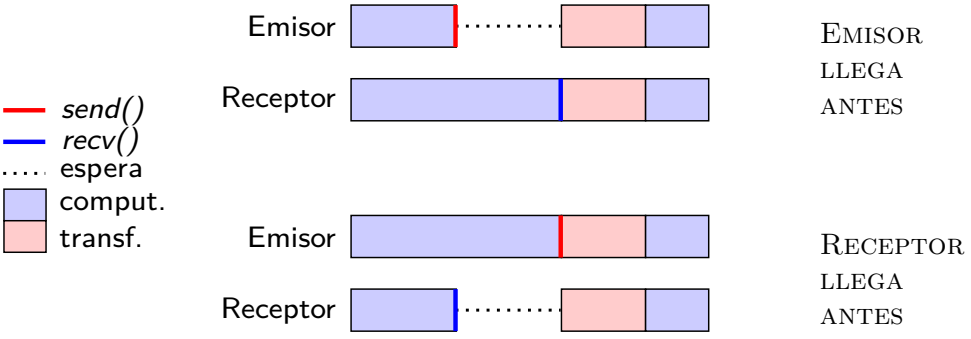
- Envío síncrono (`MPI_Ssend`)
  $\longrightarrow$ el emisor se bloquea hasta que el receptor hace la operación `recv`
- Envío con buffer (`MPI_Bsend`)
  $\longrightarrow$ se copia el mensaje a un buffer intermedio y el proceso continúa su ejecución
  (podría fallar si se queda sin memoria; se puede añadir más con `MPI_Buffer_attach`)
- Envío estándar (`MPI_Send`)
  $\longrightarrow$ Mensajes largos se envían con `MPI_Ssend`
  $\longrightarrow$ Mensajes cortos se envían con `MPI_Bsend`

Para cada modo, existen primitivas bloqueantes (`MPI_Send`, `MPI_Recv`) y no bloqueantes (`MPI_Isend`, `MPI_Irecv`)

Envío con Sincronización

En el modo síncrono, la operación `send` no termina hasta que el otro proceso ha efectuado el `recv` correspondiente

- Además de la transferencia de datos, los procesos se sincronizan
- Requiere un protocolo para que emisor y receptor sepan que puede comenzar la transmisión (es transparente al programador)



Operaciones Bloqueantes/No Bloqueantes

Las primitivas bloqueantes son seguras semánticamente

- Cuando una llamada a `send` bloqueante retorna es seguro modificar la variable enviada
- Cuando una llamada a `recv` bloqueante retorna se garantiza que la variable contiene el mensaje

Las no bloqueantes inician la operación, retornan enseguida

- Necesitamos saber si la operación ha concluido
- Un argumento adicional *req* (*request number*)

Primitivas de finalización de operación:

- `MPI_Wait`: el proceso se bloquea hasta que la operación *req* ha finalizado
- `MPI_Test`: comprueba si la operación ha finalizado o no
- Más de una op. pendiente: `MPI_Waitany`, `MPI_Waitall`

Esto se puede usar para solapar comunicación y cálculo

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

Problema: Interbloqueo

Un mal uso de `send` y `recv` puede producir interbloqueo

Caso de comunicación síncrona:

```
/* Proceso 0 */
send(x,1);
recv(y,1);
```

```
/* Proceso 1 */
send(y,0);
recv(x,0);
```

- Ambos quedan bloqueados en el envío

Caso de envío con buffer:

- El ejemplo anterior no causaría interbloqueo
- Puede haber otras situaciones con interbloqueo

```
/* Proceso 0 */
recv(y,1);
send(x,1);
```

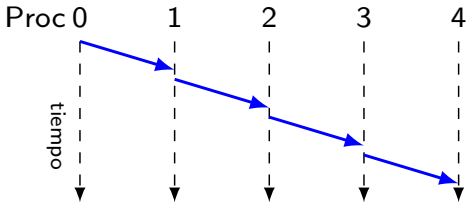
```
/* Proceso 1 */
recv(x,0);
send(y,0);
```

Posible solución: intercambiar el orden de uno de ellos

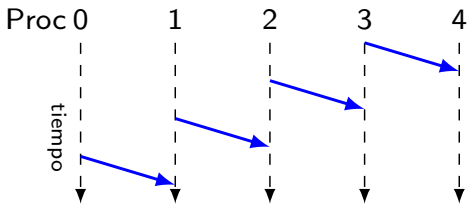
Problema: Serialización

Cada proceso tiene que enviar un dato a su vecino derecho

```
/* Proceso i */
recv(y,i-1);
send(x,i+1);
```




```
/* Proceso i */
send(x,i+1);
recv(y,i-1);
```



Posibles soluciones:

- Protocolo pares-impares: los procesos pares hacen una variante, los impares la otra
- `send` o `recv` no bloqueante
- Operaciones combinadas: `sendrecv`

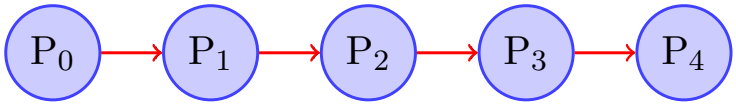
Ejemplo – Desplazamiento en Malla 1-D (1)

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

Inconveniente: Secuencialización - en caso de envío síncrono, las comunicaciones se realizan secuencialmente, sin concurrencia



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(a, N, MPI_DOUBLE, next, 0, comm);
MPI_Recv(a, N, 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

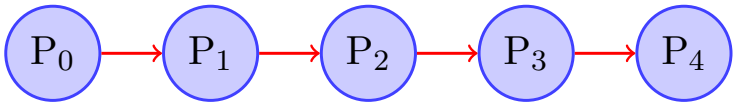
Esta versión no resuelve el problema de la secuencialización

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(a, N, MPI_DOUBLE, next, 0, comm);
  MPI_Recv(a, N, MPI_DOUBLE, prev, 0, comm, &status);
} else {
  MPI_Recv(tmp, N, MPI_DOUBLE, prev, 0, comm, &status);
  MPI_Send(a, N, MPI_DOUBLE, next, 0, comm);
  for (i=0;i<N;i++) a[i] = tmp[i]
}
```

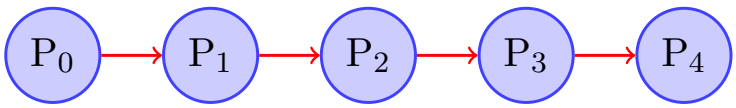


Ejemplo – Desplazamiento en Malla 1-D (4)

Solución a la secuencialización: Operaciones Combinadas

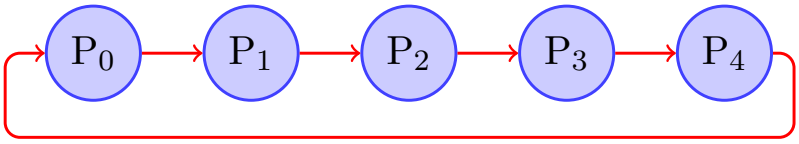
Desplazamiento en Malla 1-D – sendrecv

```
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(a, N, MPI_DOUBLE, next, 0, prev, 0,
                     comm, &status);
```



Ejemplo – Desplazamiento en Anillo

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



Desplazamiento en Anillo – versión trivial

```
next = (rank + 1) % p;
prev = (rank + p - 1) % p;

MPI_Send(a, N, MPI_DOUBLE, next, 0, comm);
MPI_Recv(a, N, MPI_DOUBLE, prev, 0, comm, &status);
```

Se producirá interbloqueo en el caso de envío síncrono
Soluciones: protocolo pares-impares u operaciones combinadas

Apartado 4

Tipos de Paralelismo

- Paralelismo de Tareas
- Paralelismo de Datos

Tipos de Paralelismo

Paralelismo de Tareas

- Los procesos MPI ejecutan diferentes tareas
- El programador debe implementar una política de asignación de tareas: qué proceso ejecuta qué tarea
- Las aristas del grafo de dependencias implican comunicar
  $\longrightarrow$ minimizar la comunicación pasa a ser un objetivo (junto con el equilibrio de la carga)
- Sincronización de tareas gracias a primitivas bloqueantes

Paralelismo de Datos

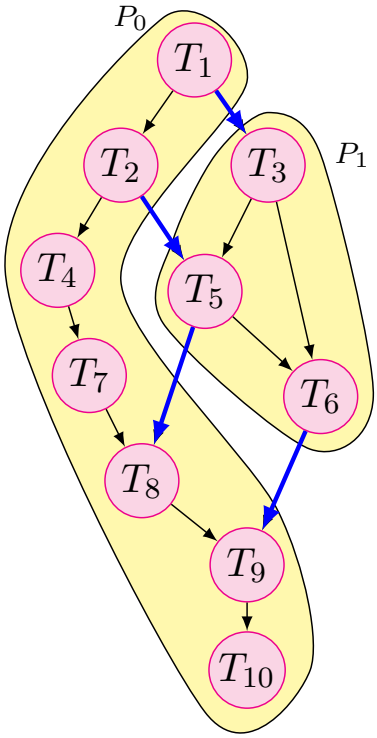
- Los procesos ejecutan diferentes iteraciones de un bucle
- Los bucles suelen recorrer una estructura de datos grande
  $\longrightarrow$ no hay variables compartidas, la estructure de datos debe ser distribuida (cada proceso almacena parte de ella)

El Problema de la Asignación

Establecer la correspondencia tarea–proceso y elegir el orden de ejecución

Ejemplo con 2 procesos:

```
double A[n][n], B[n][n], C[n][n],
       D[n][n], E[n][n], F[n][n];
f1(n,A);      /* Tarea T1 */
f2(n,D,A);    /* Tarea T2 */
f2(n,F,A);    /* Tarea T3 */
f2(n,B,D);    /* Tarea T4 */
f3(n,E,F,D);  /* Tarea T5 */
f3(n,C,E,F);  /* Tarea T6 */
f1(n,B);      /* Tarea T7 */
f2(n,E,B);    /* Tarea T8 */
f3(n,C,E,E);  /* Tarea T9 */
f1(n,C);      /* Tarea T10 */
```

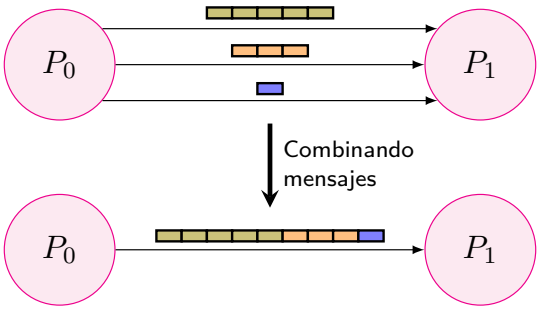


- Datos iniciales en todos los proc.
- Funciones modifican el 2º arg.
- Coste de `f1`, `f2` y `f3` es $\frac{1}{3}n^3$, n^3 y $2n^3$ flops, respectivamente

Asignación: Agrupamiento y Replicación

Agrupar tareas

- Pocas tareas grandes preferible a muchas tareas pequeñas
- Especialmente interesante si permite agrupar mensajes



Replicación

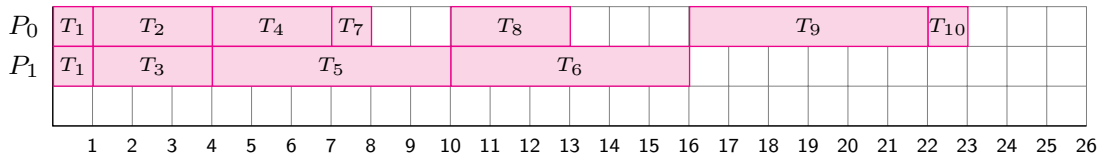
- Replicar datos en varios procesos puede reducir mensajes
- Replicar cómputo a veces puede evitar mensajes, p.e., tarea T_1 en ejemplo anterior

Asignación Mediante Identificador de Proceso

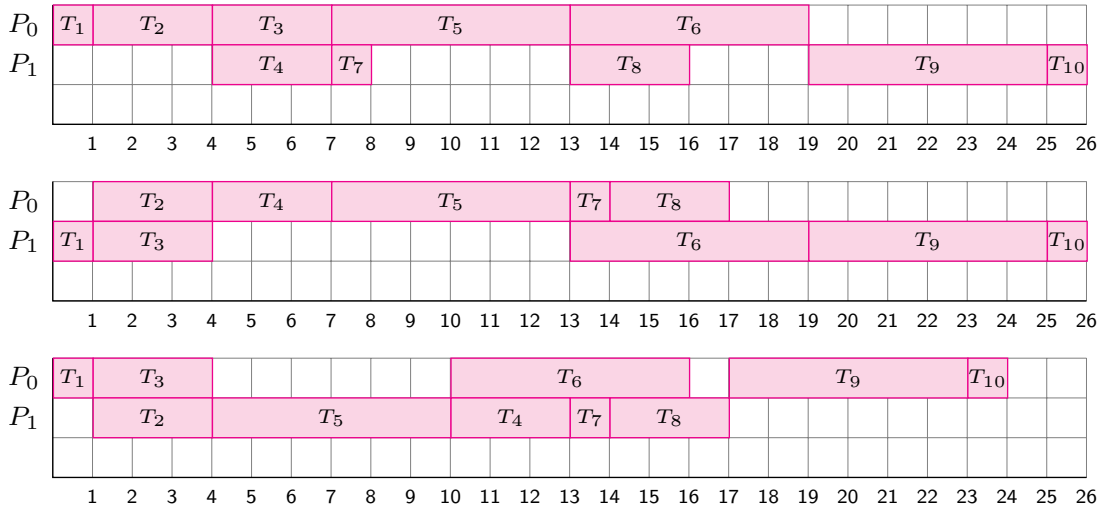
```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
f1(n,A);        /* Tarea T1, ejecutada por ambos procesos */
if (rank==0) {
  f2(n,D,A);    /* Tarea T2 */
  MPI_Send(D, n*n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
  f2(n,B,D);    /* Tarea T4 */
  f1(n,B);      /* Tarea T7 */
  MPI_Recv(E, n*n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  f2(n,E,B);    /* Tarea T8 */
  MPI_Recv(C, n*n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  f3(n,C,E,E);  /* Tarea T9 */
  f1(n,C);      /* Tarea T10 */
} else if (rank==1) {
  f2(n,F,A);    /* Tarea T3 */
  MPI_Recv(D, n*n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  f3(n,E,F,D);  /* Tarea T5 */
  MPI_Send(E, n*n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  f3(n,C,F,F);  /* Tarea T6 */
  MPI_Send(C, n*n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
```

Asignación: Tiempo de Ejecución

Orden de ejecución de tareas con la asignación anterior:



Con otras posibles asignaciones:



Estrategias de Asignación

Objetivo: minimizar tiempo de ejecución $\longrightarrow$ maximizar concurrencia (evitar desequilibrio de carga), minimizar tiempo de ocio, minimizar comunicación

Asignación Estática

- La asignación se determina a priori
- Apropiado cuando el coste de las tareas es conocido
- La asignación óptima no se conoce en el caso general

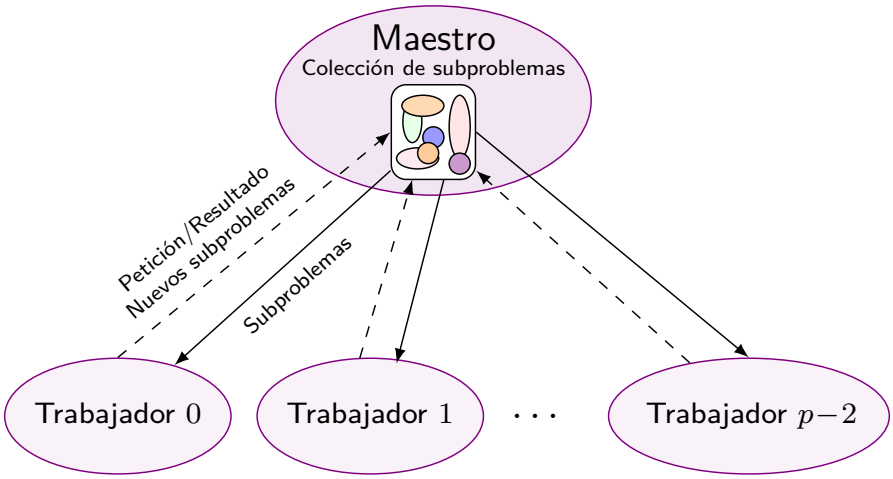
Asignación Dinámica

- Las tareas se van asignando durante la ejecución
- Apropiado cuando el coste es desconocido o cuando las tareas se generan dinámicamente
- La implementación tiene un *overhead*
  - Decidir en tiempo de ejecución qué proceso hace qué tarea
  - Intercambiar info sobre tareas (y carga)

Asignación Dinámica

Normalmente implementada mediante un esquema asimétrico

- Maestro: tiene la colección de tareas, asigna tareas pendientes a trabajadores
- Trabajadores: reciben tareas y notifican al maestro cuando acaban (pueden generar nuevas tareas, enviadas de vuelta al maestro)



Paralelismo de Datos / Particionado de Datos

En algoritmos con muchos datos que se tratan de forma similar (típicamente algoritmos matriciales)

- En memoria compartida, se paralelizan los bucles (cada hilo opera sobre una parte de los datos)
- En paso de mensajes, se realiza un particionado de datos explícito

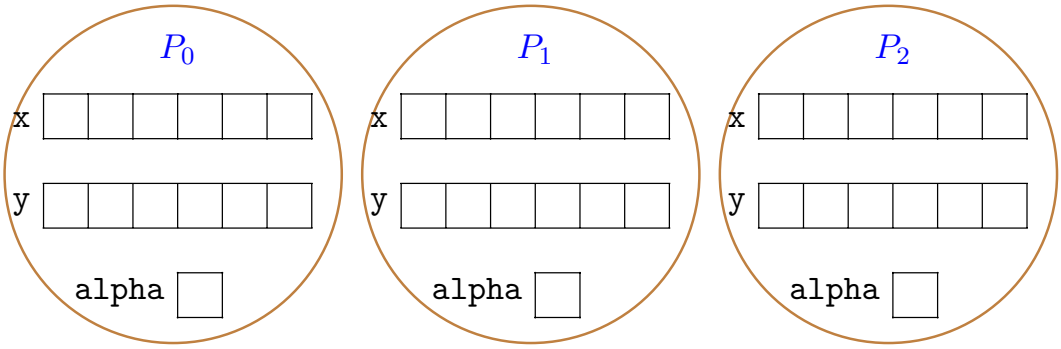
En paso de mensajes puede ser desaconsejable paralelizar

- El volumen de computación debe ser al menos un orden de magnitud mayor que el de comunicaciones
  - ✗ Vector-vector: coste $\mathcal{O}(n)$ frente a $\mathcal{O}(n)$ comunicación
  - ✗ Matriz-vector: coste $\mathcal{O}(n^2)$ frente a $\mathcal{O}(n^2)$ comunicación
  - ✓ Matriz-matriz: coste $\mathcal{O}(n^3)$ frente a $\mathcal{O}(n^2)$ comunicación
- A menudo los datos están ya distribuidos

Ejemplo: AXPY de Vectores

$y = \alpha x + y, \quad x, y \in \mathbb{R}^n, \ \alpha \in \mathbb{R}$

- Tenemos un comunicador con p procesos
- Los vectores están ya distribuidos: arrays `x`, `y` almacenan la parte "local" del vector (de longitud `nlocal` $\approx n/p$)
- `alpha` debe tener el mismo valor en todos los procesos




```
void par_axpy(int nlocal, double alpha, double *x, double *y) {
  int i;
  for (i=0; i<nlocal; i++) y[i] += alpha*x[i];
}
```


Ejemplo: Norma de un Vector

$r = \|x\|_2, \quad x \in \mathbb{R}^n, \quad r = \sqrt{\sum x_i^2}$

- El ejemplo anterior es trivialmente paralelizable (sin com.)
- Normalmente se requiere algún tipo de comunicación, incluso si los datos ya están distribuidos
- En este caso, necesitamos realizar una reducción, cuyo resultado debe estar disponible en todos los procesos

```
double par_norm(int nlocal, double *x) {
  int i;
  double s=0.0, slocal=0.0;
  for (i=0; i<nlocal; i++) slocal += x[i]*x[i];
  /* Comunicacion (no se muestra).
     Posible implementacion:
     1) todos los procesos envian slocal al proceso 0
     2) el proceso 0 recibe los valores y los suma a s
     3) el proceso 0 envía la suma total s al resto de procesos
  */
  return sqrt(s);
}
```

Ejemplo: Búsqueda en un Vector

Dado $\alpha \in \mathbb{R}$, $x \in \mathbb{R}^n$ encontrar (primer) k tal que $x_k = \alpha$

- Los ejemplos previos usan índices "locales" (`0..nlocal-1`)
- Algunos algoritmos requieren convertir a índices "globales"
- Aquí, si hay varias ocurrencias, se devuelve la primera

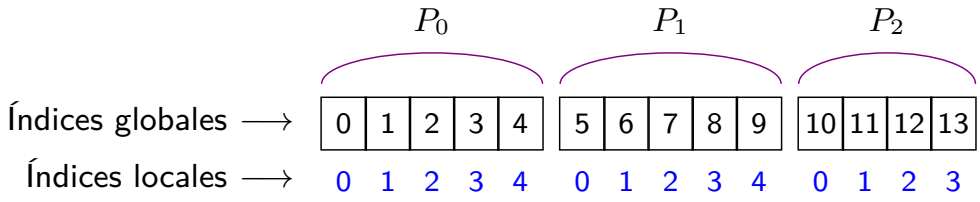
```
int par_search(int nlocal, double *x, double alpha) {
  int i, k, klocal=INF, kglobal, rank;
  for (i=0; i<nlocal; i++)
    if (x[i]==alpha) {
      klocal=i;
      break;
    }
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  kglobal = klocal + nlocal*rank;  /* asume todos los nlocal iguales */
  /* Comunicacion (no se muestra).
     1) todos los procesos envian kglobal al proceso 0
     2) el proceso 0 recibe los valores y toma el minimo en k
     3) el proceso 0 envia k al resto de procesos
  */
  return k;
}
```


Distribución Unidimensional por Bloques

El índice global i se asigna al proceso $\lfloor i/n_b \rfloor$ donde $n_b = \lceil n/p \rceil$ es el tamaño de bloque

El índice local es $i \bmod n_b$ (resto de la división entera)

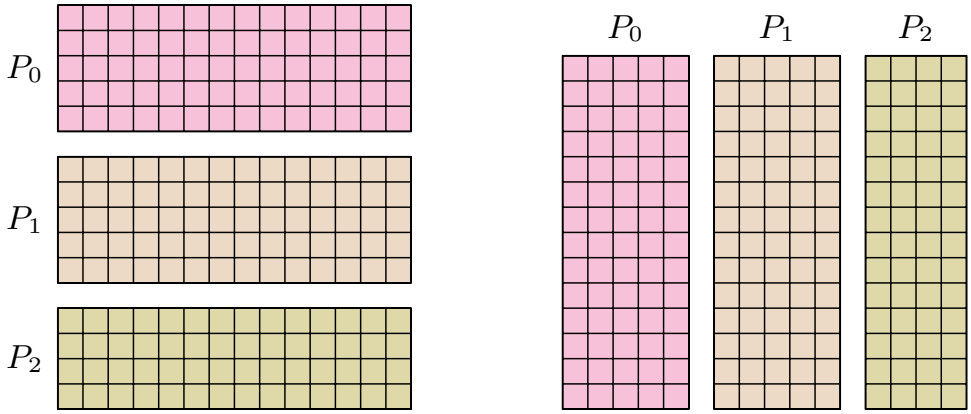
Ejemplo: para un vector de 14 elementos entre 3 procesos



$n_b = \lceil 14/3 \rceil = 5$
Cada proceso tiene n_b elementos (excepto el último)

Distribución Unidimensional por Bloques. Ejemplo

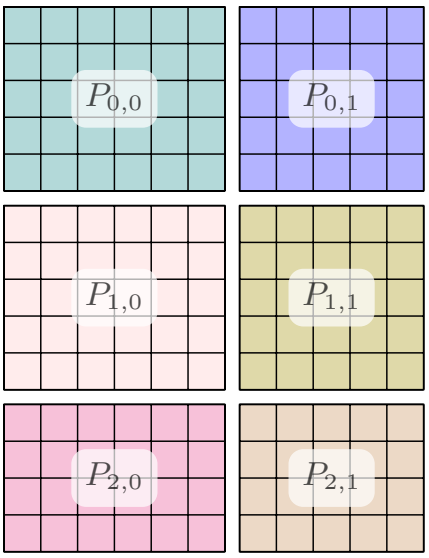
Ejemplo para una matriz bidimensional de 14×14 entradas entre 3 procesos por bloques de filas y bloques de columnas



Cada proceso posee $n_b = \lceil n/p \rceil$ filas (o columnas)

Distribución Bidimensional por Bloques

Ejemplo de distribución bidimensional por bloques para una matriz de $m \times n = 14 \times 11$ entre 6 procesos organizados en una malla lógica de 3×2



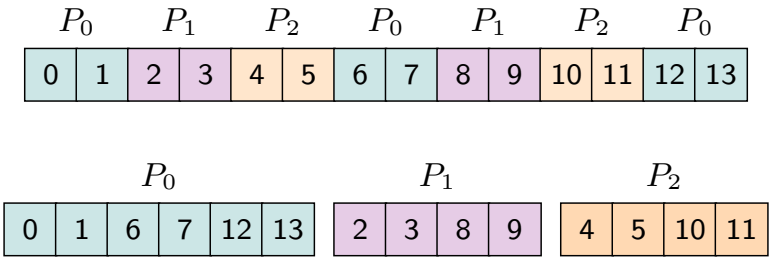
Cada proceso posee un bloque de $m_b \times n_b = \lceil m/p_m \rceil \times \lceil n/p_n \rceil$, donde p_m y p_n son la primera y segunda dimensión de la malla de procesos, respectivamente (3 y 2 en el ejemplo)

Distribuciones Cíclicas

Objetivo: equilibrar la carga durante todo el tiempo de ejecución

- Mayor coste de comunicación al reducirse la localidad
- Se combina con los esquemas por bloques
- Debe existir un equilibrio entre reparto de carga y costes de comunicación: tamaño adecuado de bloque

Distribución unidimensional cíclica por bloques (tamaño de bloque 2):



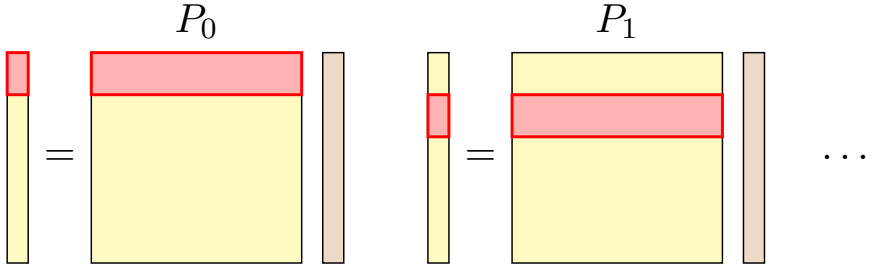
Se aplica igualmente a matrices (por filas, columnas o 2-D)

Ejemplo: Producto Matriz-Vector (1)

$y = Ax, \quad A \in \mathbb{R}^{n \times n},\ x, y \in \mathbb{R}^n$

```
SUB matvec(n, A, x, y)
  FOR i=0 TO n-1
    y[i] = 0
    FOR j=0 TO n-1
      y[i] = y[i] + A[i,j] * x[j]
    END
  END
```

- A distribuida por bloques de filas, x, y distribuidos por bloques (cada proceso calcula su parte local de y)
- Comunicación para obtener una copia completa de x en todos los procesos



Ejemplo: Producto Matriz-Vector (2)

Suponemos A, x almacenados inicialmente en P_0, al final y debe estar también en P_0

```
SUB matvec(n,A,x,y)
  distribute(n,A,Aloc,x)
  mvlocal(n,Aloc,x,yloc)
  combine(n,yloc,y)
```

```
SUB distribute(n,A,Aloc,x)
  FOREACH PROC, rank=0 TO p-1
  nb = n/p
  IF rank == 0
    Aloc = A[0:nb-1,:]
    FOR k=1 TO p-1
      send(A[k*nb:(k+1)*nb-1,:],k)
      send(x,k)
    END
  ELSE
    recv(Aloc,0)
    recv(x,0)
  END
```

```
SUB mvlocal(n,Aloc,x,yloc)
  FOREACH PROC, rank=0 TO p-1
  nb = n/p
  FOR i=0 TO nb-1
    yloc[i] = 0
    FOR j=0 TO n-1
      yloc[i] += Aloc[i,j]*x[j]
    END
  END
```

```
SUB combine(n,yloc,y)
  FOREACH PROC, rank=0 TO p-1
  nb = n/p
  IF rank == 0
    y[0:nb-1] = yloc
    FOR k=1 TO p-1
      recv(y[k*nb:(k+1)*nb-1],k)
    END
  ELSE
    send(yloc,0)
  END
```


Apartado 5

Comunicación Avanzada

- Comunicación Colectiva
- Comunicación Persistente
- Comunicación Unilateral

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

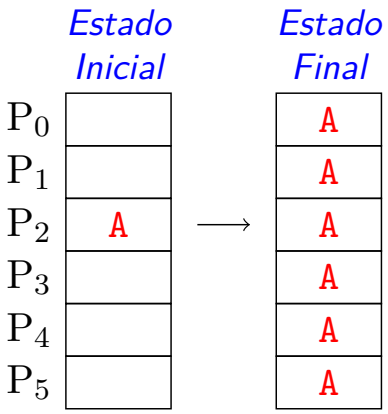
Ejemplo – medición de tiempos

```
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)
```

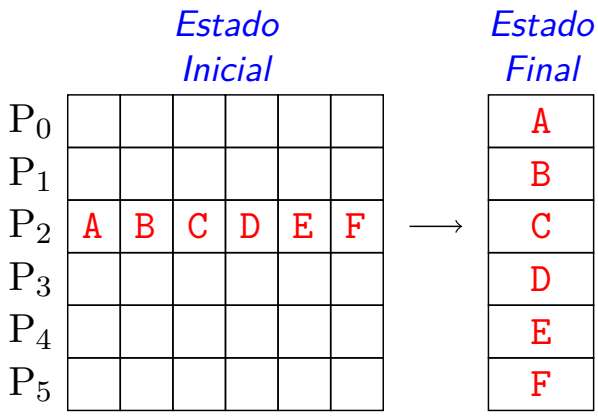
El proceso `root` difunde al resto de procesos el mensaje definido por los 3 primeros argumentos



Reparto

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

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



Versión asimétrica: `MPI_Scatterv`

Reparto: Ejemplo

El proceso P_0 reparte un vector a de longitud n, cada proceso almacena la parte local en aloc (incluido P_0)

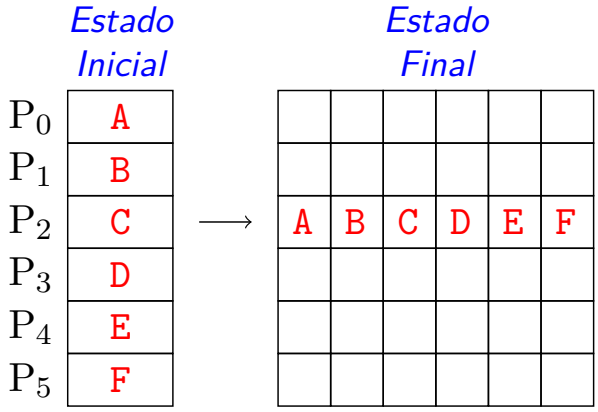
```
int i, p, rank, n=..., nloc;
double *a, *aloc;

MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) {
  a = malloc(n*sizeof(double));
  for (i=0;i<n;i++) a[i] = ...;  /* rellena array */
}
nloc = n/p;   /* asumimos division exacta */
aloc = malloc(nloc*sizeof(double));
MPI_Scatter(a, nloc, MPI_DOUBLE, aloc, nloc, MPI_DOUBLE,
            0, MPI_COMM_WORLD);
```

Si la división n/p no es exacta, se debe usar MPI_Scatterv

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

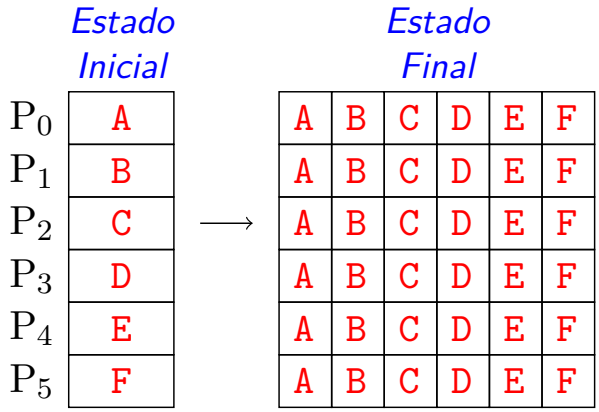


Versión asimétrica: MPI_Gatherv

Multi-Recogida

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

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

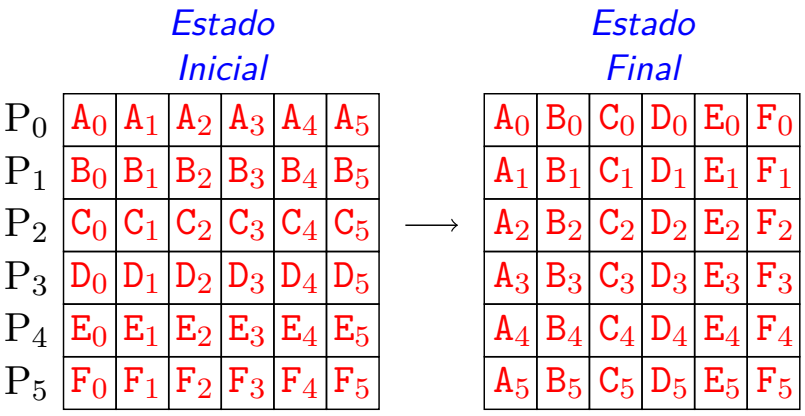


Versión asimétrica: `MPI_Allgatherv`

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



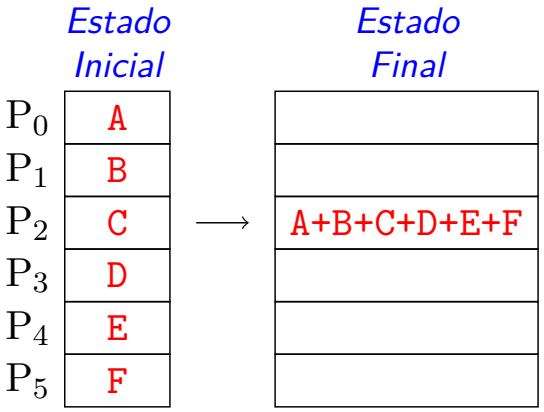
Versión asimétrica: `MPI_Alltoallv`

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`



Multi-Reducción

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

Extensión de `MPI_Reduce` en que todos reciben el resultado

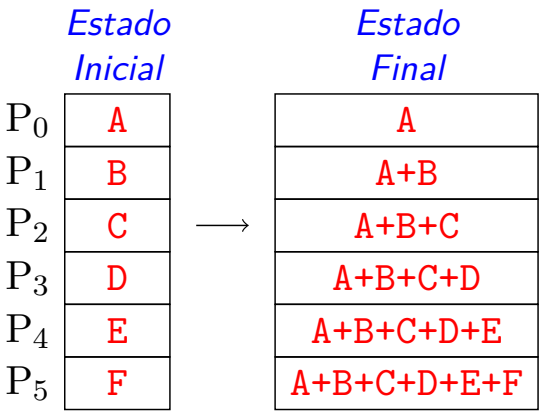
Producto escalar de vectores

```
double par_dot(int nlocal, double *x, double *y)
{
  double dot, dot_local=0.0;
  int i;

  for (i=0; i<nlocal; i++) dot_local += x[i]*y[i];
  MPI_Allreduce(&dot_local, &dot, 1, MPI_DOUBLE,
                MPI_SUM, MPI_COMM_WORLD);
  return dot;
}
```

Prefijación

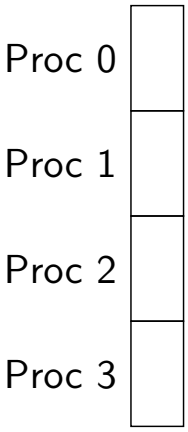
```
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm)
```

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;
```

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]);
```

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);
```

Apartado 6

Otras Funcionalidades

- Tipos de Datos Derivados
- Topologías
- Varios

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(pos, 1, MPI_INT, 0, comm);
MPI_Bcast(buffer, pos, MPI_PACKED, 0, comm);
if (my_rank != 0) {
  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 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

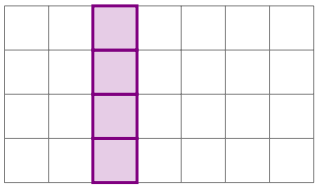
1. De cuántos bloques se compone (`count`)
2. De qué longitud son los bloques (`length`)
3. Qué separación hay entre un elemento de un bloque y el mismo elemento del siguiente bloque (`stride`)
4. De qué tipo son los elementos individuales (`type`)

Constructores relacionados:

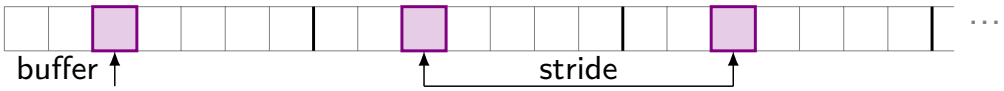
- `MPI_Type_contiguous`: elementos contiguos
- `MPI_Type_indexed`: longitud y desplazamiento variable

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 columna;
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);
}
```


Uso de Tipos Derivados con count

Al usar tipos derivados en una primitiva de comunicación con `count=2`, por ejemplo, para el segundo se usa la misma plantilla, empezando en la última posición en que quedó

- En el ejemplo anterior, si queremos enviar dos columnas de `A[4][7]` con `count=2`, no funcionaría porque nos saldríamos de la matriz
- `MPI_Type_create_resized` permite "mover" el final del tipo

```
double A[4][7];
MPI_Datatype col, col_r;
MPI_Type_vector(4, 1, 7, MPI_DOUBLE, &col);
MPI_Type_commit(&col);
MPI_Type_create_resized(col, 0, sizeof(double), &col_r);
MPI_Type_commit(&col_r);
if (my_rank == 0) {       /* envía la 3ª y 4ª columnas */
  MPI_Send(&A[0][2], 2, col_r, 1, 0, comm);
}
```

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);
```


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)

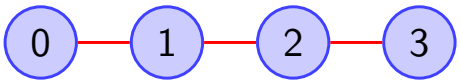
La topología cartesiana es un caso particular (muy utilizado) de la topología de grafo

Topología Cartesiana

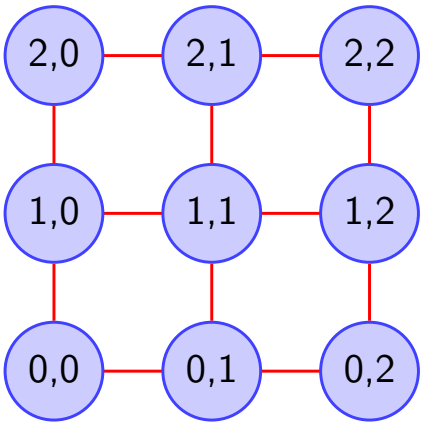
Los procesos se disponen en una malla y se conectan de forma lógica con los vecinos

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

Anillo/Malla 1-D:



Toro/Malla 2-D:



Topología Cartesiana – 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;     /* se puede hacer MPI_Dims_create(size,2,dims) */
dims[1] = ncol;
period[0] = 1;      /* periodico en esta direccion */
period[1] = 0;      /* no periodico 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;    /* desplazamiento, puede ser negativo */

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

Soporte para OpenMP (1)

Si las llamadas a MPI se hacen siempre fuera de las regiones paralelas de OpenMP, los dos paradigmas se pueden combinar sin problemas

Para poder hacer llamadas a MPI dentro de regiones paralelas:

```
MPI_Init_thread(argc, argv, required, provided)
```

Sustituto de `MPI_Init`, además inicializa el soporte para hilos. Posibilidades:

- `MPI_THREAD_SINGLE`: Solo un hilo
- `MPI_THREAD_FUNNELED`: Se permiten varios hilos, pero solo el hilo principal realiza las llamadas de MPI
- `MPI_THREAD_SERIALIZED`: Varios hilos pueden hacer llamadas a MPI, pero uno tras otro
- `MPI_THREAD_MULTIPLE`: Varios hilos, sin restricciones

Soporte para OpenMP (2)

Si está disponible un nivel `MPI_THREAD_FUNNELED` o superior, el modelo híbrido MPI-OpenMP permite potencialmente solapar comunicación y computación

Ejemplo de solapamiento de comunicación y cálculo

```
#pragma omp parallel
{
    if (thread_id==id1) {
      MPI_routine1(...);
    }
    else if (thread_id=id2) {
      MPI_routine2(...);
    }
    else {
      do_compute();
    }
}
```