CPA – Computación Paralela

Grado en Ingeniería Informática

T3. Paso de Mensajes. Diseño Avanzado de Algoritmos Paralelos

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

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Curso 2021/22

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Contenido

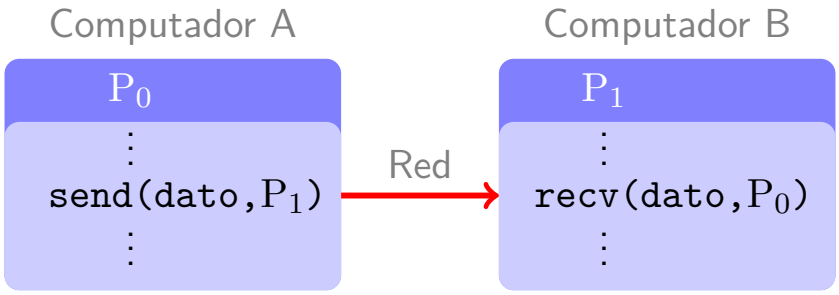
Apartado 1

Modelo de Paso de Mensajes

- Modelo
- Detalles

Modelo de Paso de Mensajes

- Las tareas manejan su espacio de memoria privado
- Se intercambian datos a través de mensajes
- La comunicación suele requerir operaciones coordinadas (p.e. envío y recepción)
- Programación laboriosa / control total de la paralelización



MPI: Message Passing Interface

Creación de Procesos

El programa paralelo se compone de diferentes procesos

- Suelen corresponderse con procesos del S.O.
- Normalmente un proceso por procesador
- Cada uno tiene un identificador o índice (entero)

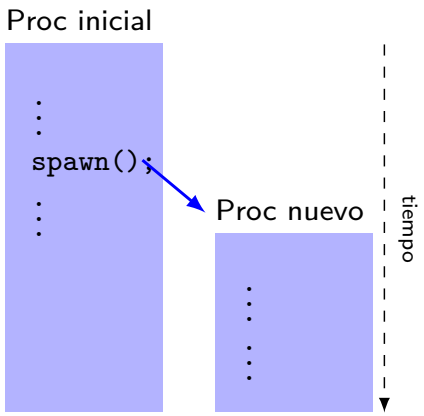
La creación de procesos puede ser:

Estática: al inicio del programa

- En línea de comandos (`mpiexec`)
- Existen durante toda la ejecución
- Es lo más habitual

Dinámica: durante la ejecución

- Primitiva `spawn()`



Comunicadores

Los procesos se organizan en grupos

- Para operaciones colectivas, como el envío 1 a todos
- Se definen mediante índices o con operaciones de conjuntos (unión, intersección, etc.)

Concepto más general: Comunicador = grupo + contexto

- La comunicación en un comunicador no puede interferir con la de otro
- Útil para aislar la comunicación dentro de una librería
- Se definen a partir de grupos u otros comunicadores
- Comunicadores predefinidos:
  - Mundo (*world*): formado por todos los procesos creados por `mpiexec`
  - Propio (*self*): formado por un solo proceso

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 o más variables

```
/* 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

Ejemplo: Suma de Vectores

$x = v + w$, $v \in \mathbb{R}^n$, $w \in \mathbb{R}^n$, $x \in \mathbb{R}^n$

- Suponemos $p = n$ procesos
- Inicialmente `v`, `w` están en P_0, y el resultado `x` debe almacenarse en P_0

```
SUB suma(v,w,x,n)
distribuir(v,w,vl,wl,n)
sumapar(v,w,vl,wl,x,xl,n)
combinar(x,xl,n)
```

```
SUB distribuir(v,w,vl,wl,n)
EN CADA P(i), i=0 HASTA n-1
  SI i == 0
    PARA j=1 HASTA n-1
      send(v[j],j)
      send(w[j],j)
    FPARA
  SI NO
    recv(vl,0)
    recv(wl,0)
  FSI
```

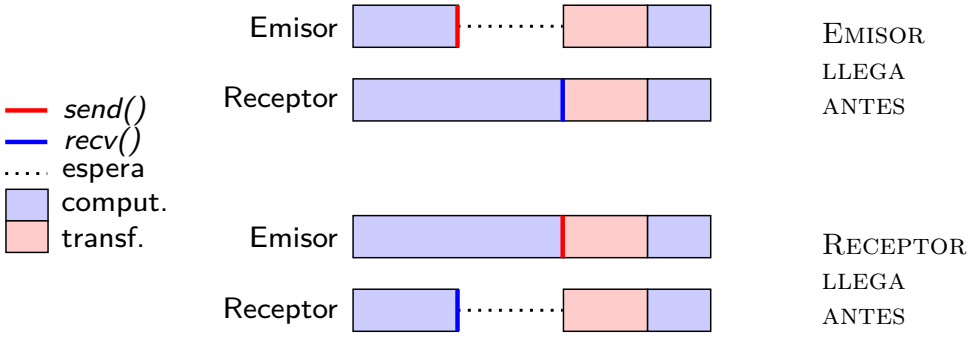
```
SUB sumapar(v,w,vl,wl,x,xl,n)
EN CADA P(i), i=0 HASTA n-1
  SI i == 0
    x[0] = v[0] + w[0];
  SI NO
    xl = vl + wl
  FSI
```

```
SUB combinar(x,xl,n)
EN CADA P(i), i=0 HASTA n-1
  SI i == 0
    PARA j=1 HASTA n-1
      recv(x[j],j)
    FPARA
  SI NO
    send(xl,0)
  FSI
```

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)



Modalidades de Envío/Recepción

Envío con buffer/envío síncrono

- Un *buffer* almacena una copia temporal del mensaje
- El `send` con buffer finaliza cuando el mensaje se ha copiado de memoria del programa a un buffer del sistema
- El `send` síncrono no finaliza hasta que se inicia el `recv` correspondiente en el otro proceso

Operaciones bloqueantes/no bloqueantes

- Al finalizar la llamada a `send` bloqueante es seguro modificar la variable que se envía
- Al finalizar la llamada a `recv` bloqueante se garantiza que la variable contiene el mensaje
- Las no bloqueantes simplemente inician la operación

Finalización de la Operación

En las operaciones no bloqueantes hay que poder determinar la finalización

- En el `recv` para poder leer el mensaje
- En el `send` para poder sobreescribir la variable

El `send` y `recv` no bloqueantes nos dan un número de operación *req*

Primitivas:

- `wait(req)`: el proceso se bloquea hasta que ha terminado la operación *req*
- `test(req)` indica si ha finalizado o no
- `waitany` y `waitall` cuando hay varias operaciones pendientes

Se puede usar para solapar comunicación y cálculo

Selección de Mensajes

La operación `recv` requiere un identificador de proceso *id*

- No concluye hasta que llega un mensaje de *id*
- Se ignoran los mensajes procedentes de otros procesos

Para más flexibilidad, se permite usar un "comodín" para recibir de cualquier proceso

Además, se usa una etiqueta (*tag*) para distinguir entre mensajes

- También permite comodín para indicar cualquier etiqueta

Ejemplo: `recv(z,any_src,any_tag,status)` aceptará el primer mensaje que entre

- La primitiva `recv` tiene un argumento `status` donde aparece el emisor y la etiqueta
- Los mensajes no seleccionados no se pierden, quedan en una "cola de mensajes"

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);
```

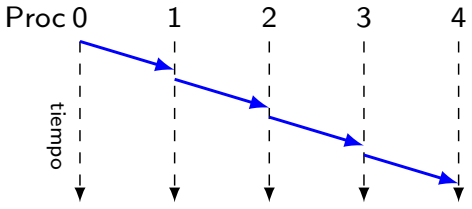
```
/* Proceso 1 */
recv(x,0);
send(y,0);
```

Posible solución: intercambiar el orden de uno de ellos

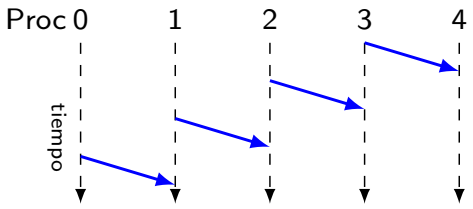
Problema: Serialización

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`

Comunicación Colectiva

Las operaciones colectivas involucran a todos los procesos de un comunicador (en muchos casos, uno de ellos tiene un papel destacado – proceso raíz)

- Sincronización (`barrier`): todos los procesos esperan a que lleguen los demás
- Movimiento de datos: uno o varios envían a uno o varios
- Reducciones: además de comunicar se realiza un cálculo sobre los datos

Estas operaciones pueden realizarse con comunicación punto a punto, pero es recomendable usar la primitiva correspondiente

- Existen varios algoritmos para cada caso (lineal, árbol)
- La solución óptima suele depender de la arquitectura (topología de la red)

Comunicación Colectiva: Tipos

- Difusión uno a todos
  - Todos reciben lo que tiene el proceso raíz
- Reducción todos a uno
  - Operación dual a la difusión
  - Los datos se combinan mediante un operador asociativo
- Reparto (*scatter*)
  - El raíz envía un mensaje individualizado a cada uno
- Recogida o concatenación (*gather*)
  - Operación dual al reparto
  - Similar a la reducción pero sin operar
- Difusión todos a todos
  - p difusiones simultáneas, con proceso raíz distinto
  - Al final, todos almacenan todos los datos
- Reducción todos a todos
  - Operación dual a la difusión todos a todos

Apartado 2

Esquemas Algorítmicos (II)

- Paralelismo de Datos
- Esquemas en Árbol
- Otros Esquemas

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

Caso 1: Producto Matriz-Vector

Solución en paso de mensajes ($p = n$ procesadores)

- Suponemos que inicialmente `v`, `A` están en P_0
- El resultado `x` debe almacenarse en P_0

```
SUB matvec(A,v,x,n,m)
distribuir(A,Al,v,n,m)
mvlocal(Al,v,xl,n,m)
combinar(xl,x,n)
```

```
SUB distribuir(A,Al,v,n,m)
EN CADA P(i), i=0 HASTA n-1
  SI i == 0
    PARA j=1 HASTA n-1
      enviar(A[j,:],j)
      enviar(v[:],j)
    FPARA
    Al = A[0,:]
  SI NO
    recibir(Al,0)
    recibir(v[:],0)
  FSI
```

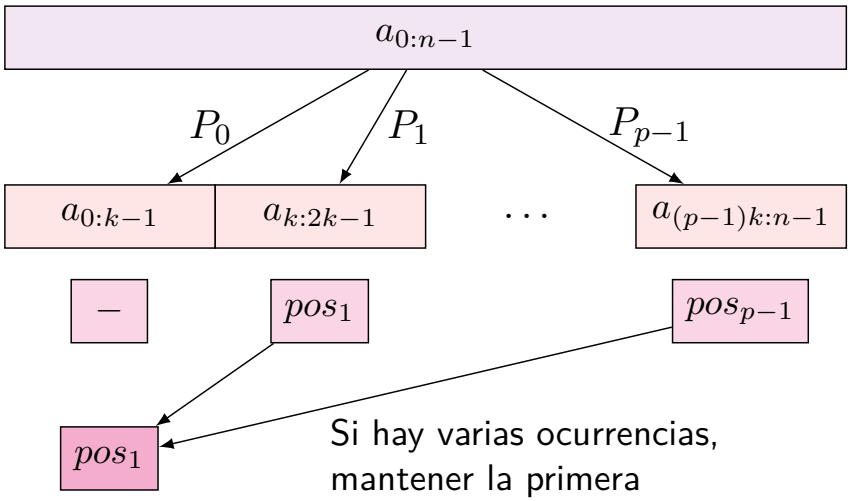
```
SUB mvlocal(Al,v,xl,n,m)
EN CADA P(i), i=0 HASTA n-1
  xl = 0
  PARA j=0 HASTA m-1
    xl = xl + Al[j] * v[j]
  FPARA
```

```
SUB combinar(xl,x,n)
EN CADA P(i), i=0 HASTA n-1
  SI i == 0
    x[0] = xl
    PARA j=1 HASTA n-1
      recibir(x[j],j)
    FPARA
  SI NO
    enviar(xl,0)
  FSI
```


Caso 2: Búsqueda Lineal

Dado un vector $a \in \mathbb{R}^n$ y un número $x \in \mathbb{R}$, encontrar un índice i tal que $x = a_i$ (puede haber varias ocurrencias)

Suponemos $n = k \cdot p$, cada proceso buscará en un sub-vector de k elementos



Caso 2: Búsqueda Lineal - Pseudocódigo

Solución en paso de mensajes

- Inicialmente x, a están en P_0, el resultado pos debe almacenarse en P_0

```
SUB busqueda(a,x,pos,n,p)
distribuir(a,apr,x,n,p)
buscaloc(apr,x,pos,n,p)
combinar(pos,n,p)
```

```
SUB distribuir(a,apr,x,n,p)
EN CADA P(i), i=0 HASTA p-1
  /* operaciones colectivas */
  /* el proceso 0 envía una
     porción de n/p de a, se
     recibe en apr */
  repartir(a,n/p,apr,n/p,p,0)
  /* el proceso 0 envía x a
     todos, se recibe en x */
  difundir(x,0)
```

```
SUB buscaloc(apr,x,pos,n,p)
EN CADA P(pr), pr=0 HASTA p-1
  pos = n;  i = 0
  MIENTRAS (i<n/p) Y (pos==n)
    SI apr[i] == x
      pos = i
    FSI
    i = i+1
  FMIENTRAS
```

```
SUB combinar(pos,n,p)
EN CADA P(pr), pr=0 HASTA p-1
  SI pr == 0
    PARA i=1 HASTA p-1
      recibir(aux,i)
      SI aux+(n/p*i)<pos
        pos = aux+(n/p*i)
      FSI
    FPARA
  SI NO
    enviar(pos,0)
  FSI
```


Caso 3: Suma de Elementos de un Vector

Solución en paso de mensajes

```
SUB suma(v,s,n,p)
distribuir(v,vloc,n,p)
sumalocal(vloc,sl,n,p)
reducir(sl,s,p)
```

```
SUB distribuir(v,vloc,n,p)
EN CADA P(i), i=0 HASTA p-1
  k = n/p
  SI i == 0
    PARA j=1 HASTA p-1
      enviar(v[j*k:(j+1)*k-1],j)
    FPARA
    vloc = v[0:k-1]
  SI NO
    recibir(vloc[0:k-1],0)
  FSI
```

```
SUB sumalocal(vloc,sl,n,p)
EN CADA P(i), i=0 HASTA p-1
  sl = 0
  PARA j=0 HASTA n/p-1
    sl = sl + vloc[j]
  FPARA
```

```
SUB reducir(sl,s,p)
EN CADA P(i), i=0 HASTA p-1
  SI i == 0
    s = sl
    PARA j=1 HASTA p-1
      recibir(saux,j)
      s = s + saux
    FPARA
  SI NO
    enviar(sl,0)
  FSI
```

Hay algoritmos mejores para la reducción: *recursive doubling*

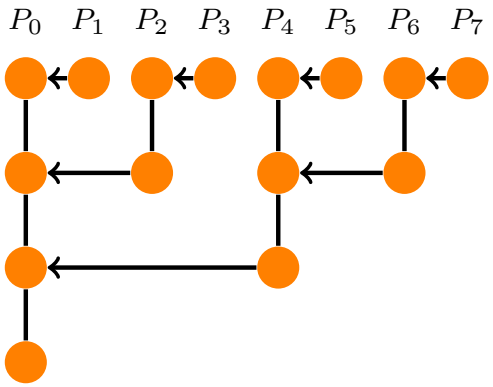
Esquemas en Árbol

Reducción mediante Recursive Doubling:

Se establecen $\log_2(p)$ etapas de comunicación

- Tras cada etapa intervienen la mitad de procesos
- El que recibe, acumula sobre su s local

```
EN CADA P(pr), pr=0 HASTA p-1
  s = sl;
  PARA j=1 HASTA log2(p)
    SI resto(pr,2^j)==0
      recibir(aux)
      s = s + aux
    SI NO
      SI resto(pr,2^(j-1))==0
        enviar(s,pr-2^(j-1))
      FSI
    FSI
  FPARA
```



Paralelismo de Tareas

En casos en que se generan más tareas que procesos, o la resolución de una tarea genera nuevas tareas

- La asignación estática no es viable o tiene problemas de carga desequilibrada
- Asignación dinámica: se van asignando a medida que los procesos quedan ociosos

---

Suele implementarse mediante un esquema asimétrico: maestro-trabajadores

- El maestro lleva cuenta de las tareas hechas/por hacer
- Los trabajadores reciben las tareas y notifican al maestro cuando las han terminado

En algunos casos es posible una solución simétrica: trabajadores replicados

Maestro-Trabajadores

Ejemplo: Fractales con paso de mensajes (`np` procesos)

Maestro

```
count=0; row=0;
for (k=1; k<np; k++) {
  send(row, k, data_tag);
  count++; row++;
}
do {
  recv({r,color}, slave, res_tag);
  count--;
  if (row<max_row) {
    send(row, slave, data_tag);
    count++; row++;
  }
  else
    send(row, slave, end_tag);
  display(r,color);
} while (count>0);
```

Trabajadores

```
recv(y, master, src_tag);

while(src_tag == data_tag) {
  /*
   * compute row colors
   */
  send( {y,color}, master,
       res_tag);
  recv(y, master, src_tag);
}
```

`count` representa el número de procesos con tarea asignada

Se procesan `max_row` líneas de la imagen (independientes)

Apartado 3

Evaluación de Prestaciones (II)

- Tiempo Paralelo
- Parámetros Relativos

Tiempo de Ejecución Paralelo

Tiempo que tarda un algoritmo paralelo en p procesadores

- Desde que empieza el primero hasta que acaba el último

Se descompone en tiempo aritmético y de comunicaciones

$$t(n,p) = t_a(n,p) + t_c(n,p)$$

t_a corresponde a todos los tiempos de cálculo

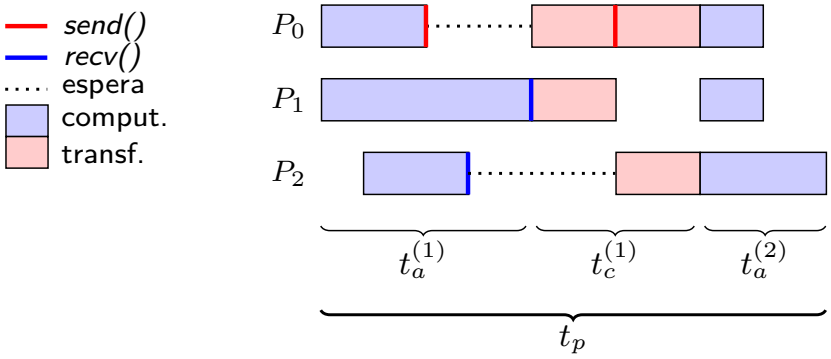
- Todos los procesadores calculan concurrentemente
- Es como mínimo igual al máximo tiempo aritmético

t_c corresponde a tiempos asociados a transferencia de datos

- En memoria distribuida t_c=tiempo de envío de mensajes
- En memoria compartida t_c=tiempo de sincronización

Tiempo de Ejecución Paralelo: Componentes

Ej.: paso de mensajes con tres procesos, P_0 envía a P_1 y P_2



En la práctica:

- No hay separación clara entre fases de cálculo y comunicación (P_1 no tiene que esperar)
- A veces se puede solapar comunicación y cálculo (con operaciones no bloqueantes, por ejemplo P_2)

$t_p = t_a + t_c - t_{\text{overlap}}$ $\quad$ t_{overlap}: tiempo de solapamiento

Modelado del Tiempo de Comunicación

Suponiendo paso de mensajes, P_0 y P_1 en nodos distintos con conexión directa

Tiempo necesario para enviar un mensaje de n bytes: $t_s + t_w n$

- Tiempo de establecimiento de la comunicación, t_s
- Ancho de banda, w (máximo número de bytes por seg.)
- Tiempo de envío de 1 byte, $t_w = 1/w$

En la práctica es más complicado:

- Red conmutada, de latencia no uniforme, colisiones, ...

Recomendaciones:

- Agrupar varios mensajes en uno solo (n grande, t_s único)
- Evitar muchas comunicaciones simultáneas

En memoria compartida, las consideraciones son distintas

Ejemplo: Producto Matriz-Vector (1)

$x = A \cdot v$, $A \in \mathbb{R}^{n \times n}$, $v \in \mathbb{R}^n$, $x \in \mathbb{R}^n$

Tiempo secuencial:
$t(n) = 2n^2$ flops

Paralelización con $p = n$ procesadores

Tiempo paralelo en memoria compartida:
$t(n, p) = 2n$ flops

Tiempo paralelo en paso de mensajes:

- `distribuir`: $2 \cdot (n-1) \cdot (t_s + t_w \cdot n)$
- `mvlocal`: $2n$ flops
- `combinar`: $(n-1) \cdot (t_s + t_w \cdot 1)$

$t(n, p) = 3 \cdot (n-1) \cdot t_s + (n-1) \cdot (2n+1) t_w + 2n$ flops

$\boxed{t(n, p) \approx 3nt_s + 2n^2 t_w + 2n \text{ flops}}$

Ejemplo: Producto Matriz-Vector (2)

Versión para $p < n$ proc. (distribución por bloques de filas)

```
SUB matvec(a,v,x,n,p)
distribuir(a,aloc,v,n,p)
mvlocal(aloc,v,x,n,p)
combinar(x,n,p)
```

```
SUB distribuir(a,aloc,v,n,p)
EN CADA P(i), i=0 HASTA p-1
  nb = n/p
  SI i == 0
    aloc = a[0:nb-1,:]
    PARA j=1 HASTA p-1
      send(a[j*nb:(j+1)*nb-1,:],j)
      send(v[:],j)
    FPARA
  SI NO
    recv(aloc,0)
    recv(v,0)
  FSI
```

```
SUB mvlocal(aloc,v,x,n,p)
EN CADA P(pr), pr=0 HASTA p-1
  nb = n/p
  PARA i=0 HASTA nb-1
    x[i] = 0
    PARA j=0 HASTA n-1
      x[i] += aloc[i,j] * v[j]
    FPARA
  FPARA
```

```
SUB combinar(x,n,p)
EN CADA P(i), i=0 HASTA p-1
  nb = n/p
  SI i == 0
    PARA j=1 HASTA p-1
      recv(x[j*nb:(j+1)*nb-1],j)
    FPARA
  SI NO
    send(x[0:nb-1],0)
  FSI
```


Ejemplo: Producto Matriz-Vector (3)

Paralelización con $p < n$ procesadores

Tiempo paralelo en paso de mensajes:

- `distribuir`:
  $(p-1)\cdot\left(t_s + t_w \cdot \frac{n^2}{p}\right)+(p-1)\cdot(t_s+t_w\cdot n) \approx 2pt_s+n^2t_w+pnt_w$
- `mvlocal`: $2\frac{n^2}{p}$ flops
- `combinar`: $(p-1)\cdot(t_s + t_w \cdot n/p) \approx pt_s + nt_w$

$$\boxed{t(n,p) \approx 3pt_s + (n^2+pn)t_w + 2\frac{n^2}{p} \text{ flops}}$$

Parámetros Relativos

Los parámetros relativos sirven para comparar un algoritmo paralelo con otro

- Speedup: $S(n, p)$
- Eficiencia: $E(n, p)$

Normalmente se aplican en el análisis experimental, aunque el speedup y la eficiencia se pueden obtener en el análisis teórico

Speedup y Eficiencia

El *speedup* indica la ganancia de velocidad que consigue el algoritmo paralelo con respecto a un algoritmo secuencial

$$S(n, p) = \frac{t(n)}{t(n, p)}$$

Hay que indicar a qué se refiere $t(n)$

- Puede ser el mejor algoritmo secuencial conocido
- Puede ser el algoritmo paralelo ejecutado en 1 procesador

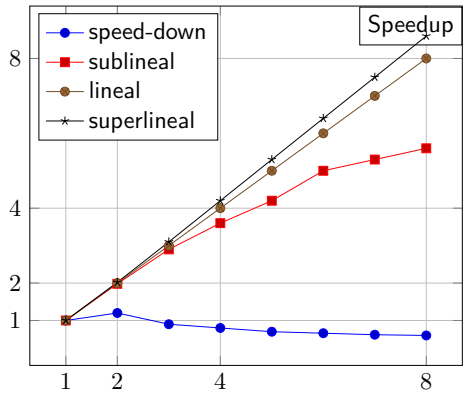
La eficiencia mide el grado de aprovechamiento que un algoritmo paralelo hace de un computador paralelo

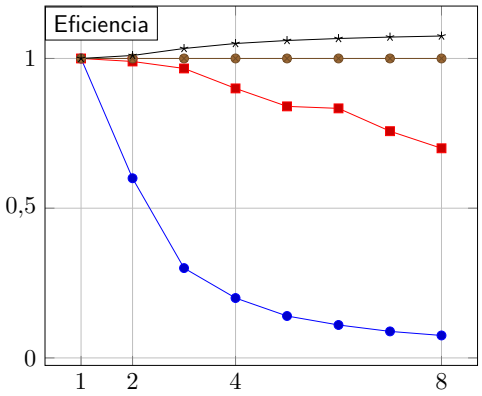
$$E(n, p) = \frac{S(n, p)}{p}$$

Suele expresarse en tanto por cien (o tanto por 1)

Speedup: Casos Posibles

$S(n,p) < 1$	$1 < S(n,p) < p$	$S(n,p) = p$	$S(n,p) > p$
"Speed-down"	Caso sublineal	Caso lineal	Caso superlineal
El algoritmo paralelo es más lento que el algoritmo secuencial	El algoritmo paralelo es más rápido que el secuencial, pero no aprovecha toda la capacidad de los proc.	El algoritmo paralelo es lo más rápido posible, aprovechamiento de los procesadores al 100 %	Situación anómala, el algoritmo paralelo tiene menor coste que el secuencial





Ejemplo: Producto Matriz-Vector

Tiempo secuencial: $t(n) = 2n^2$ flops

Paralelización por filas ($p = n$ procesadores)

En memoria compartida:
$t(n,p) = 2n$
$S(n,p) = n$
$E(n,p) = 1$

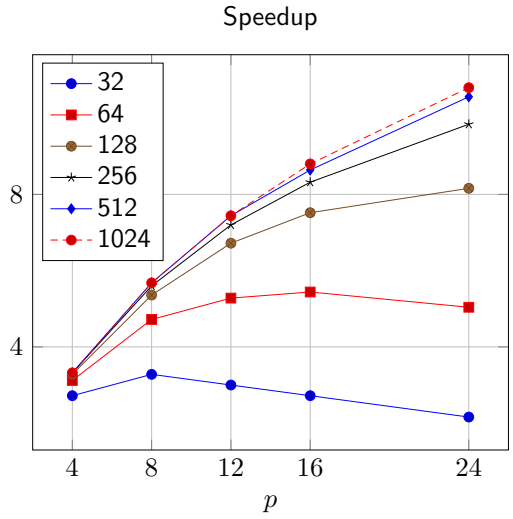
En paso de mensajes:
$t(n,p) = 2n^2 t_w + 3nt_s + 2n$
$S(n,p) \to 1/t_w$
$E(n,p) \to 0$

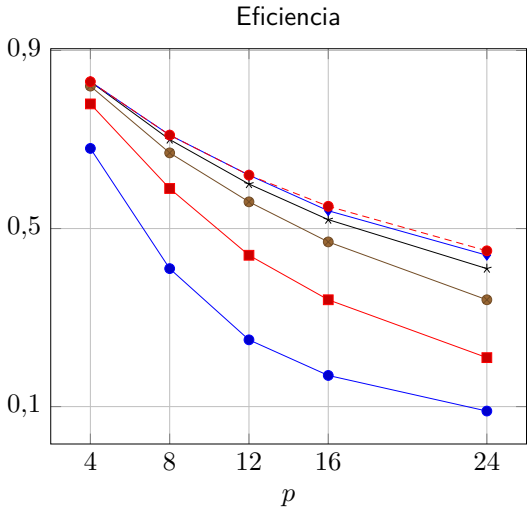
Paralelización por bloques de filas ($p < n$ procesadores)

En paso de mensajes:
$t(n,p) = 3pt_s + (n^2 + pn)t_w + 2\frac{n^2}{p}$
$S(n,p) \to \frac{2p}{pt_w+2}$
$E(n,p) \to \frac{2}{pt_w+2}$

Variación de Prestaciones

- Normalmente la eficiencia disminuye a medida que se incrementa el número de procesadores
- El efecto es menos acusado para tamaños de problema más grandes





Ley de Amdahl

Muchas veces una parte del problema no se puede paralelizar
$\rightarrow$ La Ley de Amdahl mide el speedup máximo alcanzable

Dado un algoritmo secuencial, descomponemos $t(n) = t_s + t_p$

- t_s es el tiempo de la parte intrínsecamente secuencial
- t_p es el tiempo de la parte perfectamente paralelizable (se puede resolver con p procesadores)

El tiempo mínimo paralelo alcanzable será $t(n,p) = t_s + \frac{t_p}{p}$

Speedup máximo:

$$\lim_{p\to\infty} S(n,p) = \lim_{p\to\infty} \frac{t(n)}{t(n,p)} = \lim_{p\to\infty} \frac{t_s + t_p}{t_s + \frac{t_p}{p}} = 1 + \frac{t_p}{t_s}$$

Apartado 4

Diseño de Algoritmos: Asignación de Tareas

- El Problema de la Asignación
- Estrategias de Agrupamiento y Replicación

Asignación de Tareas

- La fase de descomposición ha dado lugar a un conjunto de tareas
- Tenemos un algoritmo paralelo abstracto *independiente* de la plataforma hardware e *ineficiente*
- Es necesario *adaptar* la descomposición obtenida a una arquitectura particular

---

La asignación de tareas o planificación de tareas consiste en determinar

- en qué unidades de procesamiento y
- en qué orden

se ejecutará cada tarea

Procesos y Procesadores

- Proceso: Unidad lógica de cómputo capaz de ejecutar tareas computacionales
- Procesador: Unidad hardware que realiza cálculos

Un algoritmo paralelo se compone de procesos que ejecutan tareas

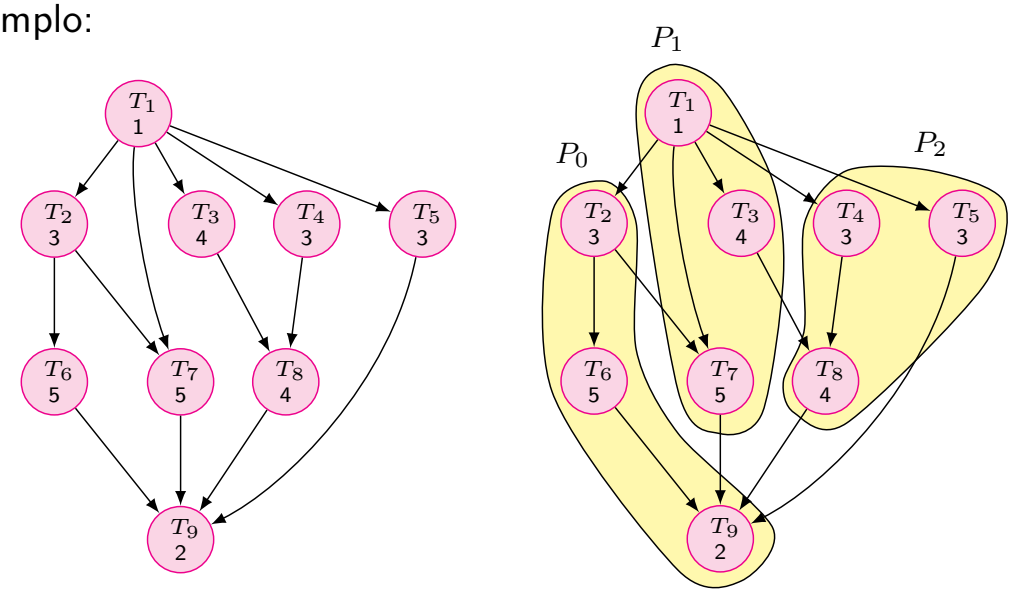
- La asignación establece correspondencia entre tareas y procesos en la fase de diseño
- La correspondencia entre procesos y procesadores se hace al final y probablemente en ejecución

El Problema de la Asignación. Ejemplo (1)

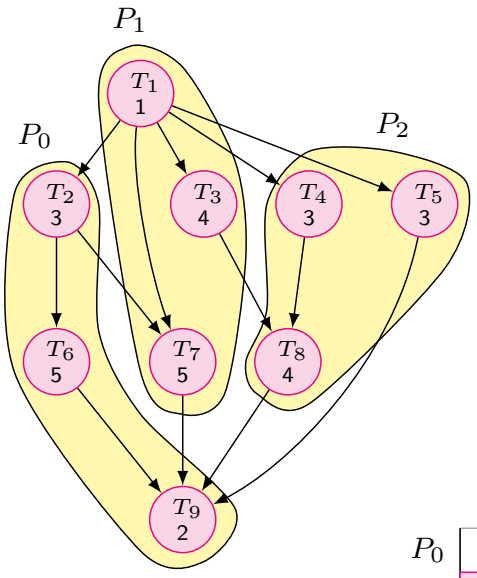
Asignación: Establecer correspondencia tarea–proceso y seleccionar el orden de ejecución

Suele incluir también el agrupamiento previo de algunas tareas

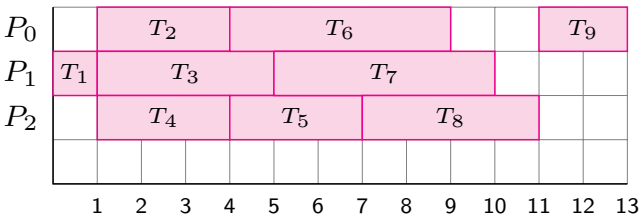
Ejemplo:



El Problema de la Asignación. Ejemplo (2)



Orden de ejecución de las tareas según la asignación realizada



Objetivos de la Asignación

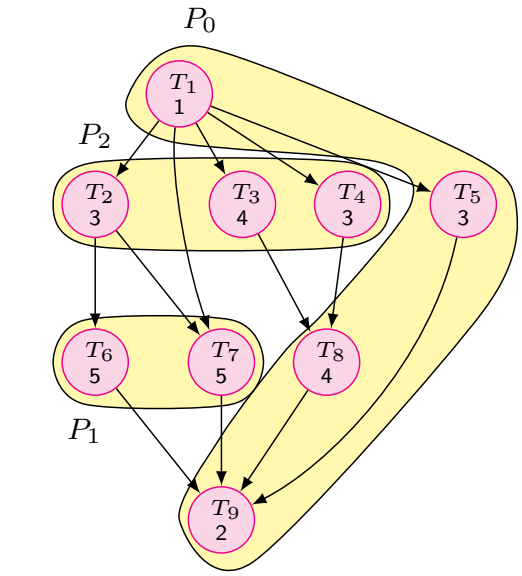
Objetivo: Minimizar el tiempo de ejecución

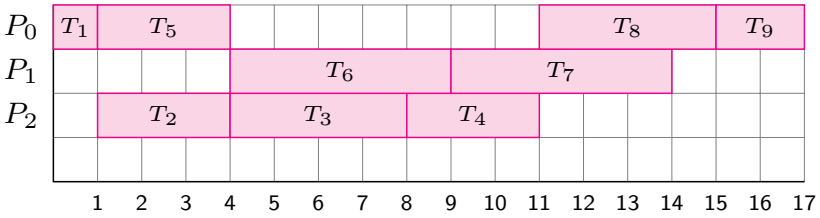
Composición del tiempo de ejecución de un algoritmo paralelo y estrategias de minimización:

- Tiempo de computación: Maximizar concurrencia asignando tareas independientes a procesos distintos
- Tiempo de comunicación: Asignar tareas que se comuniquen mucho al mismo proceso
- Tiempo de ocio: Minimizar las dos causas de ociosidad:
  - Desequilibrios de carga: se busca equilibrar los cálculos y las comunicaciones entre procesos (diagrama anterior)
  - Espera: se busca minimizar la espera de tareas que no están preparadas

Objetivos de la Asignación. Ejemplo

Ejemplo de asignación con carga equilibrada pero con más *esperas*





Estrategias Generales de Asignación (1)

Asignación estática o planificación determinista: Las decisiones de asignación se toman antes de la ejecución. Pasos:

1. Se estima el número de tareas, su tiempo de ejecución y los costes de comunicación
2. Se agrupan tareas en otras mayores para reducir coste de comunicación
3. Se asocian tareas a procesos

El problema de asignación estática óptima es *NP-completo* para el caso general[1]

Ventajas de la asignación estática:

- No añade ninguna sobrecarga en tiempo de ejecución
- Diseño e implementación son más sencillos

[1] No se conoce ningún algoritmo con tiempo polinomial que solucione el problema

Estrategias Generales de Asignación (2)

Asignación dinámica: El reparto del trabajo computacional se realiza en tiempo de ejecución

Este tipo de asignación se emplea cuando:

- Las tareas se generan dinámicamente
- El tamaño de las tareas no se conoce a priori

En general, las técnicas dinámicas son más complejas. La principal desventaja es la sobrecarga inducida debida a

- La transferencia de información de carga y de trabajo computacional entre los procesos
- La toma de decisiones para mover carga entre procesos (se realiza en tiempo de ejecución)

Ventaja: No es necesario conocer el comportamiento a priori, son flexibles y apropiadas para arquitecturas heterogéneas

Agrupamiento (1)

El agrupamiento se utiliza para reducir el número de tareas con objeto de:

- Limitar costes de creación y destrucción de tareas
- Minimizar los retardos debidos a la interacción entre tareas (acceso a datos locales frente a remotos)

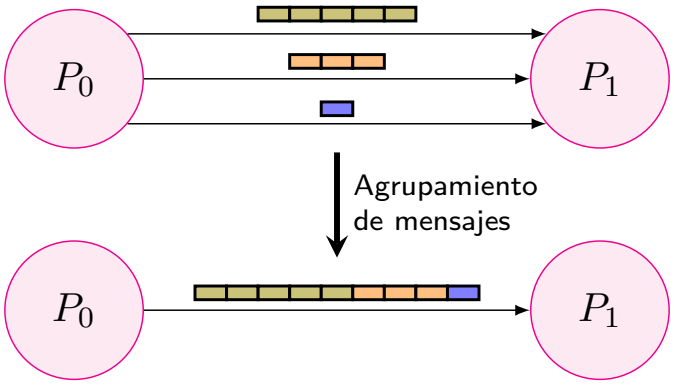
Estrategias de agrupamiento:

- *Minimización del volumen de datos transferidos*. Distribución de tareas basada en bloques de datos (algoritmos matriciales), agrupamiento de tareas no concurrentes (grafos de tareas estáticos), almacenamiento temporal de resultados intermedios (ej., producto escalar de dos vectores)
- *Reducción de la frecuencia de interacciones*. Minimizar número de transferencias y aumentar el volumen de datos a transferir en cada una

Agrupamiento (2)

Reducción de la frecuencia de interacciones

- En *paso de mensajes* significa reducir el número de mensajes (latencia) y aumentar el tamaño de éstos



- En *memoria compartida* significa reducir el número de fallos de caché

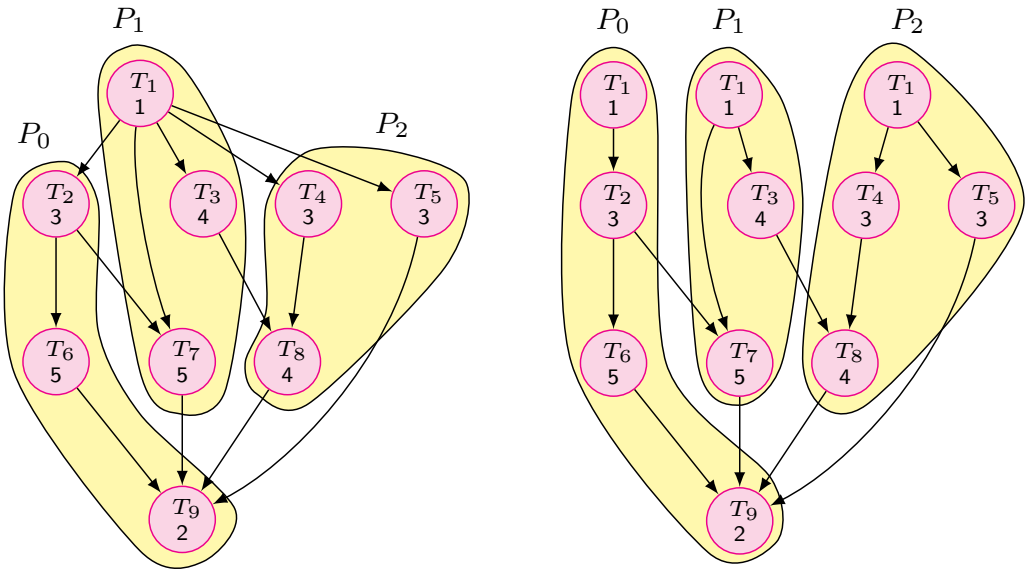
Replicación

La replicación implica que parte de los cálculos o datos del problema no se reparten sino que son realizados o manejados por todos

- Replicación de datos: consiste en copiar datos de acceso común en los distintos procesos con objeto de reducir las comunicaciones
  - En *memoria compartida* es implícito ya que solo afecta a la memoria caché
  - En *memoria distribuida* puede conllevar una mejora considerable del rendimiento y simplificación del diseño
- Replicación de cómputo y comunicación: consiste en repetir un cálculo en cada uno de los procesos en caso de que este cálculo sea menor que el coste de transferencia y cálculo, tanto secuencial como paralelo

Replicación. Ejemplo

Ejemplo de replicación de cómputo y comunicaciones. Sea el siguiente grafo, suponiendo que las comunicaciones tienen un coste. Replicamos la tarea T1



Apartado 5

Esquemas de Asignación

- Esquemas de Asignación Estática
- Equilibrado Dinámico de la Carga

Esquemas de Asignación Estática

Esquemas estáticos para descomposición de dominio

- Se centran en estructuras de datos de gran tamaño
- La asignación de tareas a procesos consiste en repartir los datos entre los procesos
- Principalmente dos tipos:
  - Distribución de matrices por bloques
  - División estática de grafos

Esquemas sobre grafos de dependencias estáticos

- Se suelen obtener mediante descomposición funcional del flujo de datos o descomposición recursiva

Distribuciones de Matrices por Bloques

En computación matricial suele suceder que el cálculo de un elemento depende de elementos contiguos (localidad espacial)

- La asignación hace corresponder porciones contiguas (bloques) del dominio de datos (matriz)

Distribuciones por bloques más usuales:

1. Distribución unidimensional por bloques de un vector
2. Distribución unidimensional por bloques de filas de una matriz
3. Distribución unidimensional por bloques de columnas de una matriz
4. Distribución bidimensional por bloques de una matriz

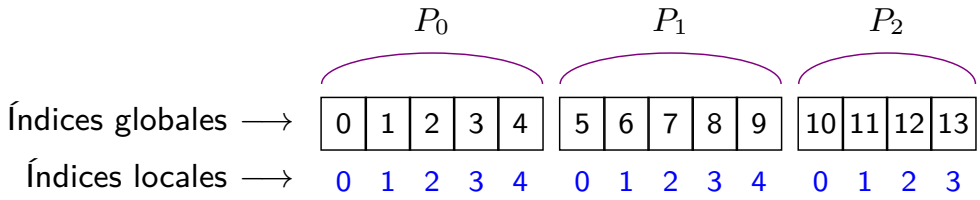
También veremos las variantes cíclicas

Distribución Unidimensional por Bloques

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

El índice local es $i \mod m_b$ (resto de la división entera)

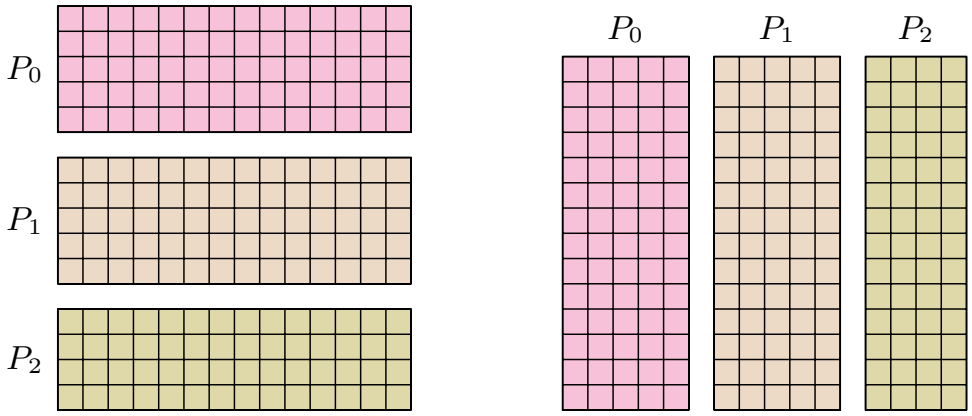
Ejemplo: para un vector de 14 elementos entre 3 procesos



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

Distribución Unidimensional por Bloques. Ejemplo

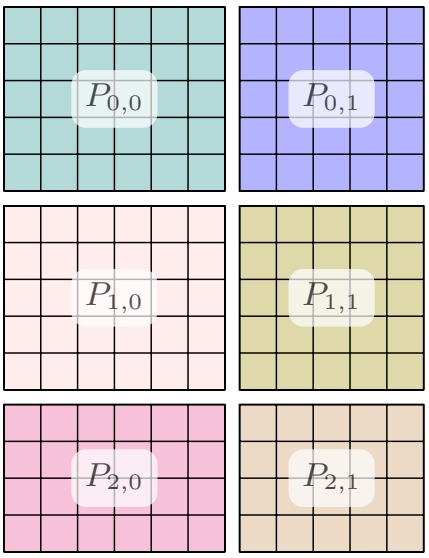
Ejemplo para una matriz bidimensional de 14×14 entradas entre 3 procesos por bloques de filas y bloques de columnas



Cada proceso posee $m_b = \lceil n/p \rceil$ 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)

Ejemplo: Diferencias Finitas (1)

Cálculo iterativo sobre una matriz $A \in \mathbb{R}^{n \times n}$

- Al inicio tiene un valor dado $A^{(0)}$
- En la iteración k-ésima ($k = 0, 1, \dots$) se obtiene un nuevo valor $A^{(k+1)} = \left(a_{i,j}^{(k+1)}\right)$, $i, j = 0, \dots, n-1$, donde

$$a_{i,j}^{(k+1)} = a_{i,j}^{(k)} - \Delta t \left(\frac{a_{i+1,j}^{(k)} - a_{i-1,j}^{(k)}}{0{,}1} + \frac{a_{i,j+1}^{(k)} - a_{i,j-1}^{(k)}}{0{,}02} \right)$$

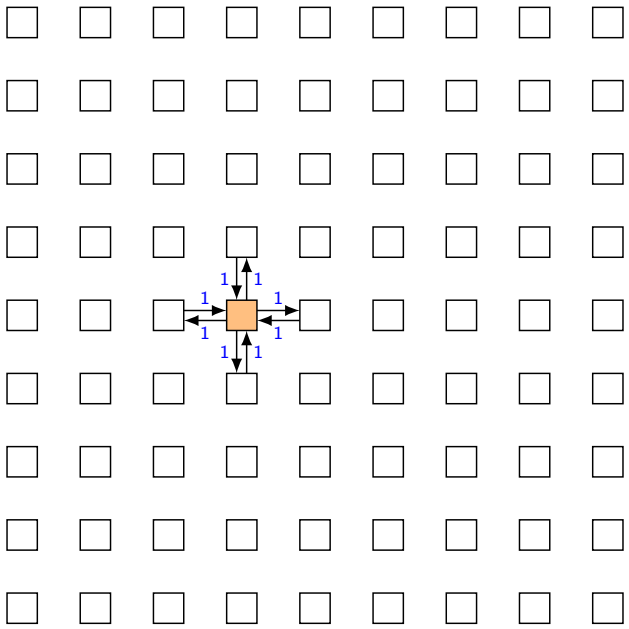
y determinadas condiciones de contorno

Vemos a continuación el esquema de comunicaciones del algoritmo para diferentes distribuciones (para $n = 9$)

Ejemplo: Diferencias Finitas (2)

Sin agrupamiento

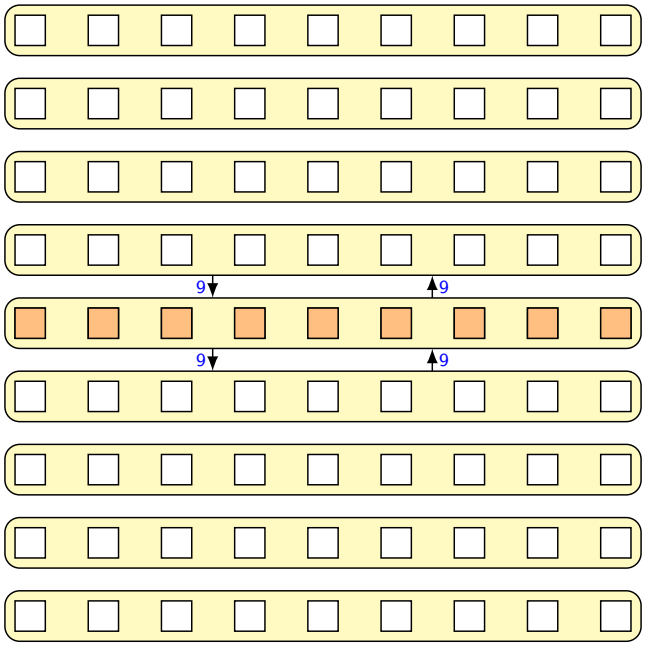
- 4 envíos por tarea (1 elemento cada uno)
- 288 envíos totales, 288 elementos transferidos



Ejemplo: Diferencias Finitas (3)

Agrupamiento unidimensional

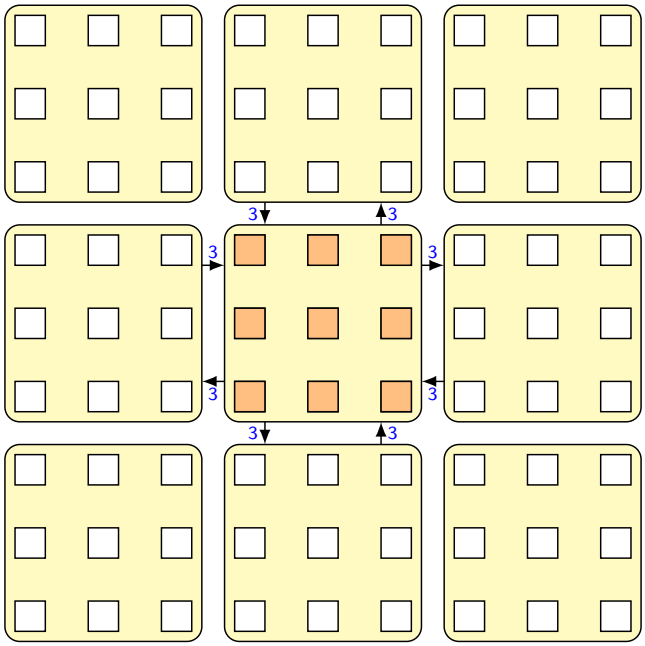
- 2 envíos por tarea (9 elementos cada uno)
- 16 envíos totales, 144 elementos transferidos



Ejemplo: Diferencias Finitas (4)

Agrupamiento bidimensional:

- 4 envíos por tarea (3 elementos cada uno)
- 24 envíos totales, 72 elementos transferidos



Efecto Volumen-Superficie

El agrupamiento mejora la localidad

- Reduce el volumen de comunicaciones
- Interesa el agrupamiento con más computación y menos comunicaciones

Efecto volumen-superficie

- La carga de computación aumenta proporcionalmente al número de elementos asignados a la tarea (volumen en matrices 3D)
- El coste de comunicaciones aumenta proporcionalmente al perímetro de la tarea (superficie en matrices 3D)

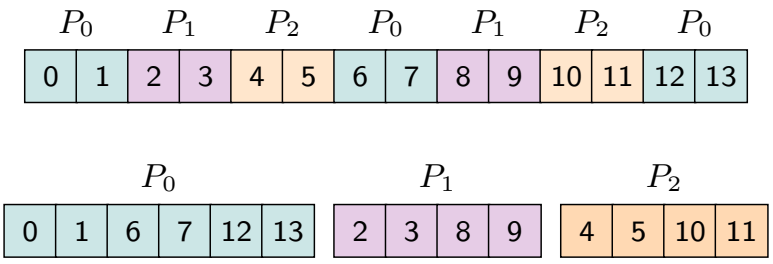
Este efecto aumenta con el número de dimensiones de la matriz

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

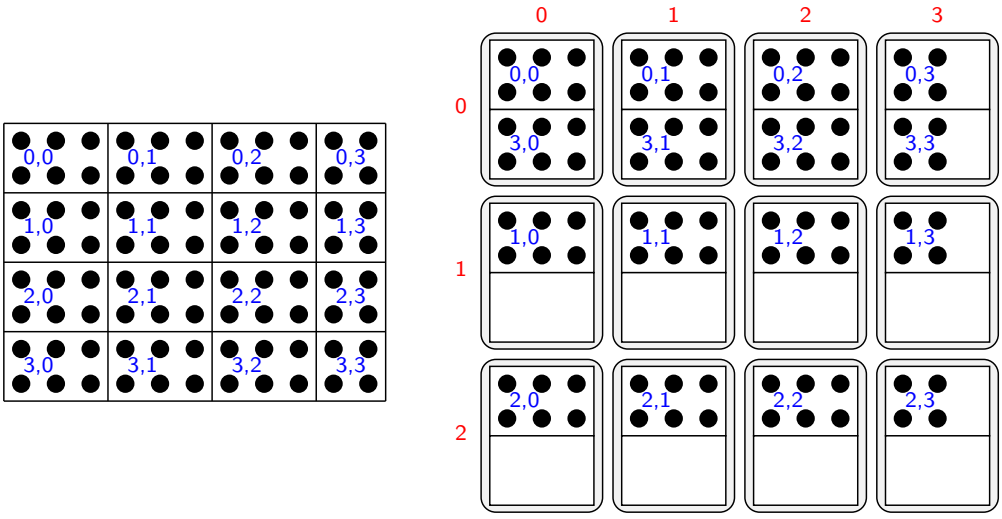
Distribución cíclica unidimensional (tamaño de bloque 2):



Se aplica igualmente a matrices (por filas o columnas)

Distribución Cíclica Bidimensional

Ejemplo de distribución cíclica por bloques bidimensional:
Matriz de 8×11 elementos en bloques de 2×3 en una malla de 3×4 procesos



Asignación Basada en Grafos de Dependencias Estáticos

Caso de descomposición funcional

- Suponemos un grafo de dependencias estático y coste de tareas conocido

El problema de encontrar asignaciones óptimas es NP-completo

Existen situaciones para las que se conocen algoritmos óptimos o enfoques heurísticos

Ejemplos:

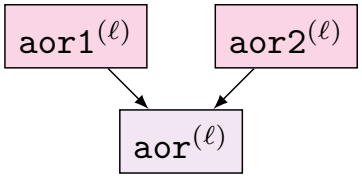
- *Estructura de árbol binomial*
- *Estructura de hipercubo*

Ejemplo: Ordenación de un Vector (1)

Dado un vector de números (a) ordenarlo en otro vector (aor)

```
SUB mergesort(a,aor,n)
SI n<=k
  ordena(a,aor,n)
SI NO
  m = n/2
  a1 = a[0:m-1]
  a2 = a[m:n-1]
  mergesort(a1,aor1,m)
  mergesort(a2,aor2,n-m)
  merge(aor1,aor2,aor)
FSI
```

Grafo de dependencias simplificado:

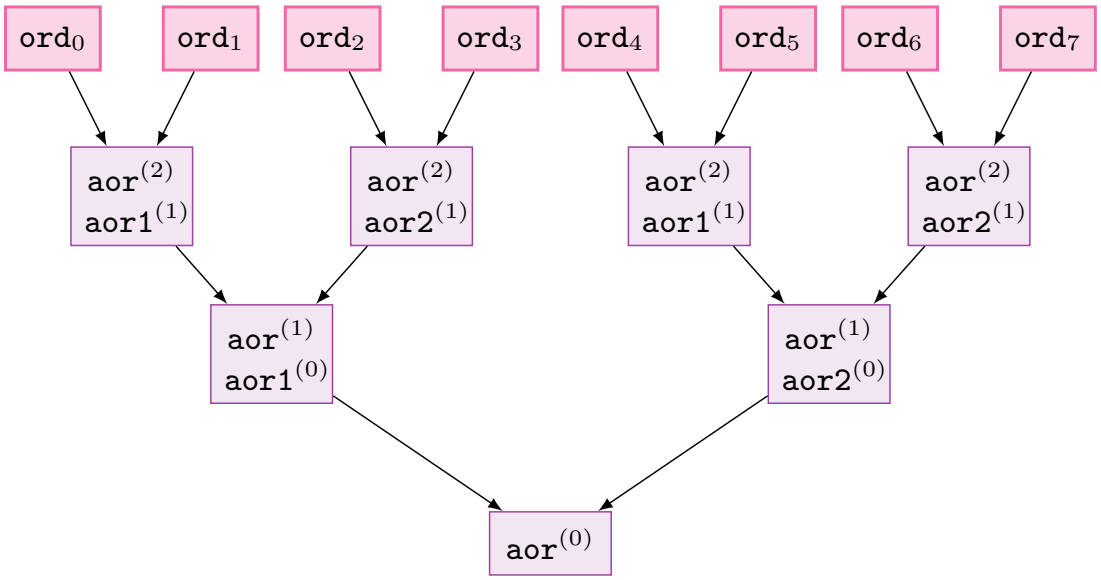


Estrategia de paralelización:

- $\log_2(n/k)$ niveles de recursión
- Distribuir tareas
  - Hojas del árbol (ordena)
  - merge en cada nivel ℓ

Ejemplo: Ordenación de un Vector (2)

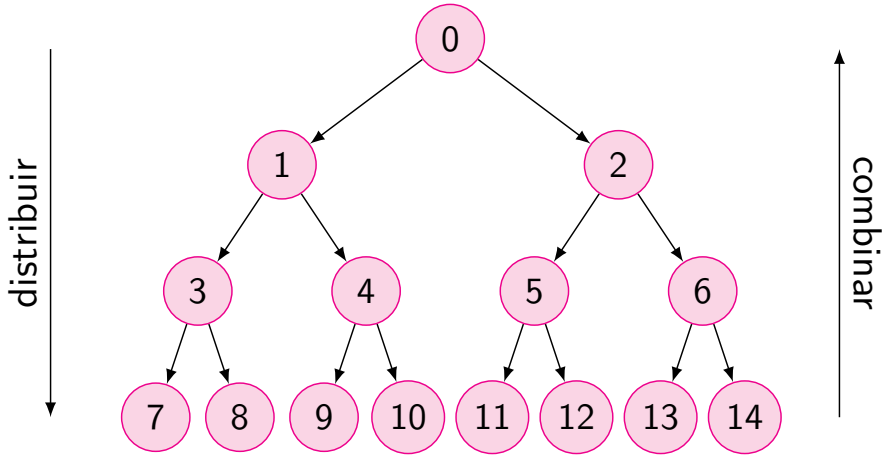
Grafo de dependencias completo para $n = 8k$



Ejemplo: Ordenación de un Vector (3)

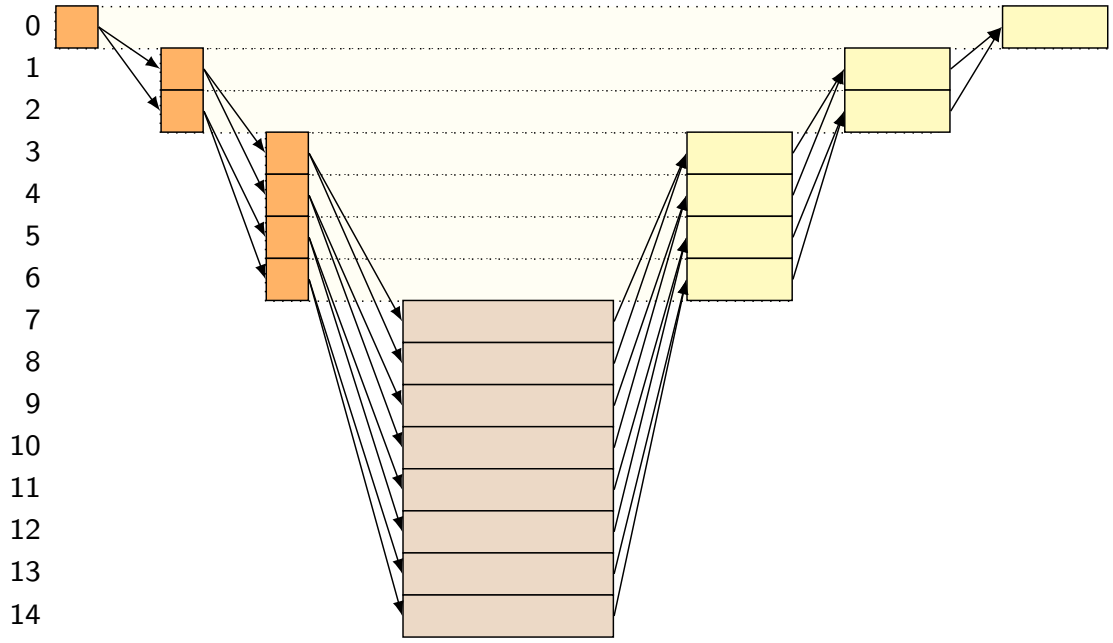
Supongamos p procesadores con topología de árbol binario

- Último nivel ordena, el resto combina
- Se podrá llegar hasta un tamaño $k * (p + 1)/2$



No es necesario tener una topología física en árbol, se emula

Ejemplo: Ordenación de un Vector (4)



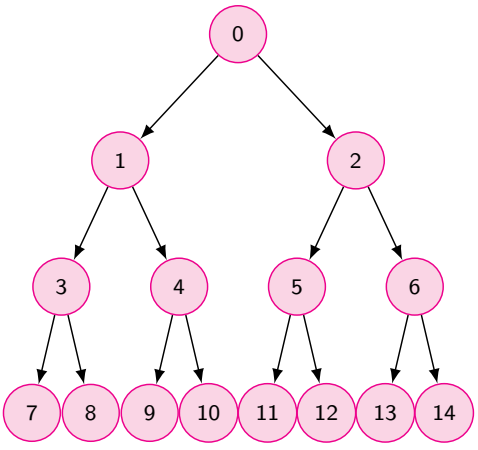
Mejor eficiencia si "reutilizamos" procesadores

- Por ejemplo, tantos procesadores como hojas del árbol

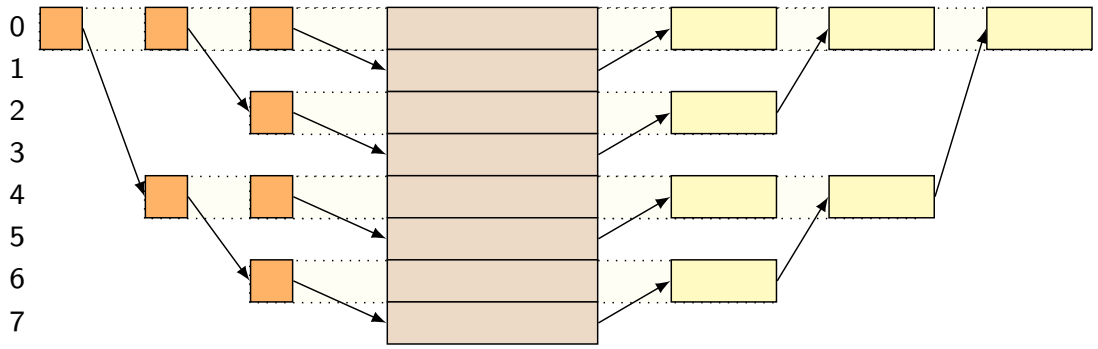
Ejemplo: Ordenación de un Vector (5)

Paso de mensajes: tantos procesadores como nodos del árbol

```
EN CADA P(pr), pr=0 HASTA p-1
  SI pr <> 0
    recibir(a,(pr-1)/2)
  FSI
  SI log2(pr+1) < log2(p)
    n = len(a)
    a1 = a[0:n/2-1]
    a2 = a[n/2:n-1]
    enviar(a1,pr*2+1)
    enviar(a2,pr*2+2)
    recibir(aor1,pr*2+1)
    recibir(aor2,pr*2+2)
    merge(aor1,aor2,aor)
  SI NO
    ordena(a,aor)
  FSI
  SI pr <> 0
    enviar(aor,(pr-1)/2)
  FSI
```



Ejemplo: Ordenación de un Vector (6)



Esquemas de Equilibrado Dinámico de la Carga

Cuando la asignación estática no es factible

- Las tareas obtenidas en la descomposición pasan a ser estructuras de datos que representan subproblemas
- Los subproblemas se mantienen en una colección desde la cual se van asignando a los procesos
- La solución de un subproblema puede conducir a la creación dinámica de otros subproblemas
- Necesitan un mecanismo de detección de fin para terminar

Tipos:

- Esquema centralizado, un proceso maestro gestiona la colección de subproblemas y los va asignando
- Esquema distribuido, sin maestro; equilibrado de la carga no trivial