Programación Concurrente 2016

Clase 2

Facultad de Informática
UNLP

Resumen de la clase anterior

Concurrencia

-Procesos y programas concurrentes

-Procesamiento secuencial, concurrente y paralelo

-Objetivos de los sistemas concurrentes

-Monoprocesadores, Multiprocesadores de MC y de MD

-Clases de aplicaciones (multithreaded, distribuidas, paralelas)

-Sincronización entre procesos. Mecanismos.

-Comunicación entre procesos. Mecanismos.

-Prioridad, manejo de recursos, inanición, deadlock, no determinismo

-Requerimientos para un lenguaje de programación concurrente

Aspectos de Programación Secuencial

C/ proceso concurrente es un programa secuencial $\Rightarrow$ es necesario referirse a algunos aspectos de la programación secuencial

La Programación Secuencial estructurada puede expresarse con 3 clases de instrucciones básicas: asignación, alternativa (decisión) e iteración (repetición con condición).

Es necesaria alguna instrucción para expresar la concurrencia…

DECLARACIONES DE VARIABLES

- Variable simple: **tipo variable = valor . Ej: int x = 8; int z, y;**
- Arreglos: **int a[10]; int c[3:10]**
  int b[10] = ([10] 2)
  int aa[5,5]; int cc[3:10,2:9]
  int bb[5,5] = ([5] ([5] 2))

Aspectos de Programación Secuencial

ASIGNACION

- Asignación simple: **x = e**
- Sentencia de asignación compuesta: **x = x + 1; y = y – 1; z = x + y**
- Llamado a funciones: **x = f(y) + g(6) – 7**
- Swap: **v1 :=: v2**
- **skip** termina inmediatamente y no tiene efecto sobre ninguna variable de programa

Aspectos de Programación Secuencial

ALTERNATIVA

• Sentencias de alternativa simple **if B → S**

B expresión booleana. **S** instrucción simple o compuesta

B "guarda" a **S** pues **S** no se ejecuta si **B** no es verdadera

Ej: **if p > 0 → p = p - 1**

• Sentencias de alternativa múltiple:

if $B_1 \rightarrow S_1$

□ $B_2 \rightarrow S_2$

.....

□ $B_n \rightarrow S_n$

fi

Las guardas se evalúan en algún orden arbitrario.

Elección no determinística.

Si ninguna guarda es verdadera el **if** no tiene efecto

Aspectos de Programación Secuencial

Ej 1:

if p > 2 → p = p * 2
□ p < 2 → p = p * 3
□ p = 2 → p = 5
fi

Ej 2:

if p > 2 → p = p * 2
□ p < 2 → p = p * 3
fi

Ej 3:

if p > 2 → p = p * 2
□ p < 6 → p = p + 4
□ p = 4 → p = p / 2
fi

- Otra opción

if (cond) S; if (cond) S_1 else S_2

Aspectos de Programación Secuencial

ITERACION

- Sentencias de alternativa **ITERATIVA** múltiple:

do $B_1 \rightarrow S_1$
□ $B_2 \rightarrow S_2$
.....
□ $B_n \rightarrow S_n$
od

Las sentencias guardadas son evaluadas y ejecutadas hasta que todas las guardas sean falsas.

La elección es *no determinística* si más de una guarda es verdadera.

Aspectos de Programación Secuencial

Ej 1:

do p > 0 → p = p - 2
□ p < 0 → p = p + 3
□ p = 0 → p = random(x)
fi

Ej 2:

do p > 2 → p = p * 2
□ p < 2 → p = p * 3
od

Ej 3:

do p = 1 → p = p * 2
□ p = 2 → p = p + 4
□ p = 4 → p = p / 2
od

Ej 4:

do p > 0 → p = p - 2
□ p > 3 → p = p + 3
□ p > 6 → p = p / 2
od

- Otra forma

while (cond) S;

Aspectos de Programación Secuencial

- For-all: forma general de repetición e iteración:

  fa cuantificadores → Secuencia de Instrucciones af

  cuantificador ≡ variable := expr_inicial to expr_final st B

Cada cuantificador especifica un rango de valores p/ una vble de iteración (con un rango y una condición "such that"):

- El cuerpo del fa se ejecuta 1 vez por c/ valor de la vble de iteración.
- Si hay cláusula *such-that*, la vble de iteración toma sólo los valores para los que B es true.
- Si hay varios cuantificadores el cuerpo se ejecuta p/ c/ combinación

Aspectos de Programación Secuencial

Ej: fa i := 1 to n → a[i] = 0 af

Ej: fa i := 1 to n, j := i + 1 to n → m[i,j] :=: m[j,i] af

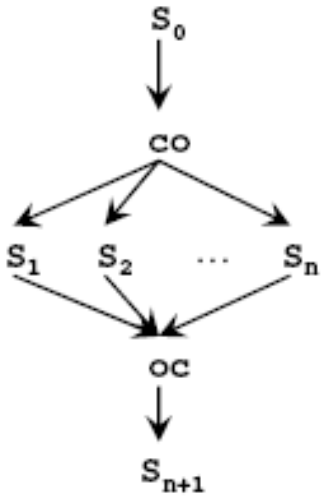
Ej: fa i := 1 to n, j := i+1 to n st a[i] > a[j] → a[i] :=: a[j] af

Concurrencia

- **CONCURRENCIA**

- Sentencia ***co***:
  - **co S_1 // ..... // S_n oc** → Ejecuta las *Si* tareas concurrentemente.
  - La ejecución del *co* termina cuando todas las tareas terminaron.

  Cuantificadores:
  - **co [i=1 to n] { a[i] = 0; b[i] = 0 } oc** → Crea *n* tareas concurrentes.



Concurrencia

- ***Process:*** otra forma de representar concurrencia
  - **process A {sentencias}** → proceso único independiente.
  - Cuantificadores
    process B [i=1 to n] {sentencias} → *n* procesos independientes.

- **Diferencia: *process*** ejecuta en ***background***, mientras el código que contiene un ***co*** espera a que el proceso creado por la sentencia ***co*** termine antes de ejecutar la siguiente sentencia.

Concurrencia

Ejemplo: qué imprime en cada caso?

```
process imprime10 {
        for [i=1 to 10]
            write(i);      }
```

```
process imprime1 [i=1 to 10] {
            write(i);
 }
```

No determinismo….

Paradigmas de resolución de programas concurrentes

Si bien el número de aplicaciones es muy grande, en general los “patrones” de resolución concurrentes son pocos:

1- Paralelismo iterativo

2- Paralelismo recursivo

3- Productores y consumidores (*pipelines* o *workflows*)

4- Clientes y servidores

5- Pares que interactúan (*interacting peers*)

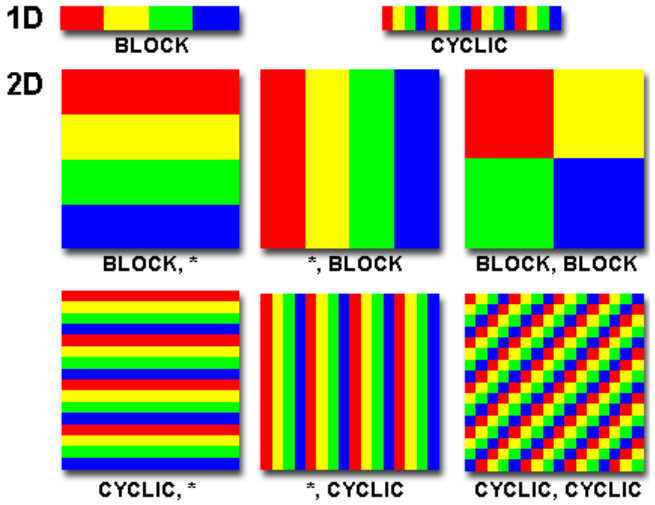
Paradigmas de resolución de programas concurrentes

En el *paralelismo iterativo* un programa consta de un conjunto de procesos (posiblemente idénticos) c/u de los cuales tiene 1 o más loops $\Rightarrow$ cada proceso es un programa iterativo.

Los procesos cooperan para resolver un único problema (x ej un sistema de ecuaciones), pueden trabajar independientemente, y comunicarse y sincronizar por memoria compartida o MP.

Gralmente, el dominio de datos se divide entre los procesos siguiendo diferentes patrones.

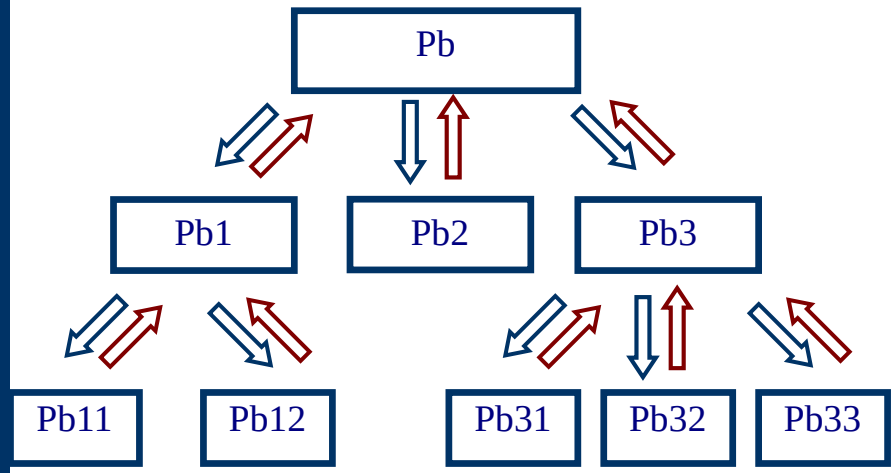
$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix}$$

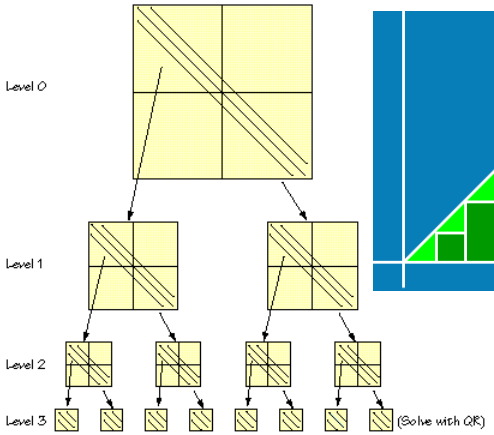


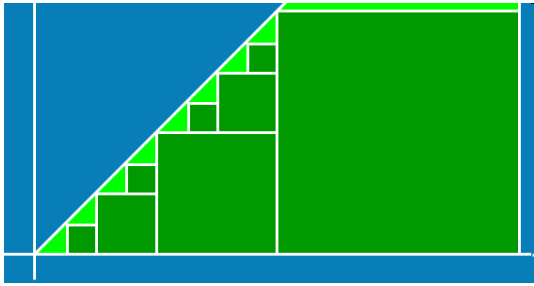
Paradigmas de resolución de programas concurrentes

En el *paralelismo recursivo* el problema general (programa) puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos (*Dividir y conquistar*)

Ejemplos clásicos son el sorting by merging, el cálculo de raíces en funciones continuas, problema del viajante, juegos (tipo ajedrez)







Divide and Conquer

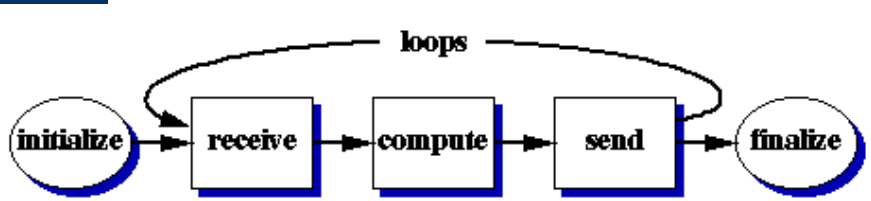
Paradigmas de resolución de programas concurrentes

Los esquemas *productor-consumidor* muestran procesos que se comunican.

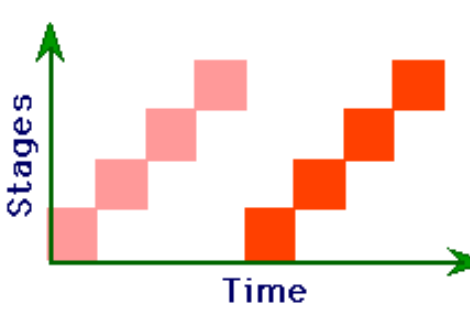
Es habitual que estos procesos se organicen en pipes a través de los cuales fluye la información.

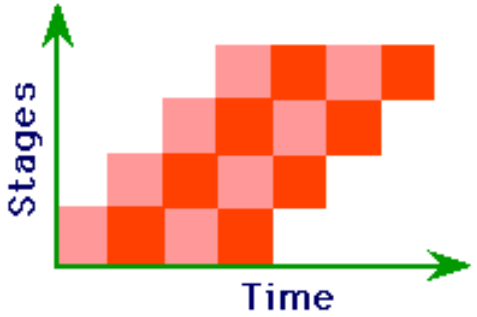
Cada proceso en el pipe es un filtro que consume la salida de su proceso predecesor y produce una salida para el proceso siguiente.

Ejemplos a distintos niveles de SO.



Pipe



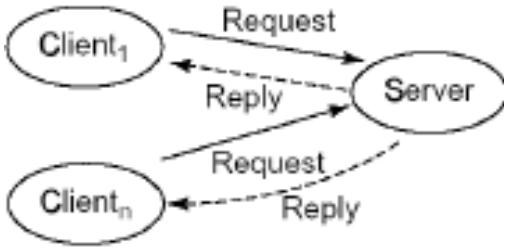


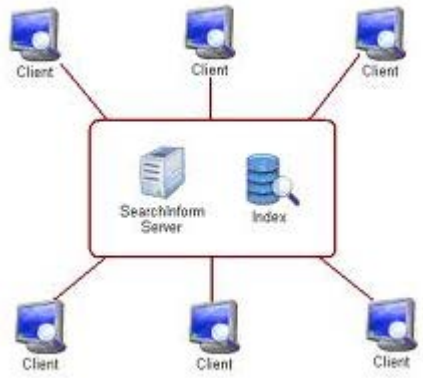
Paradigmas de resolución de programas concurrentes

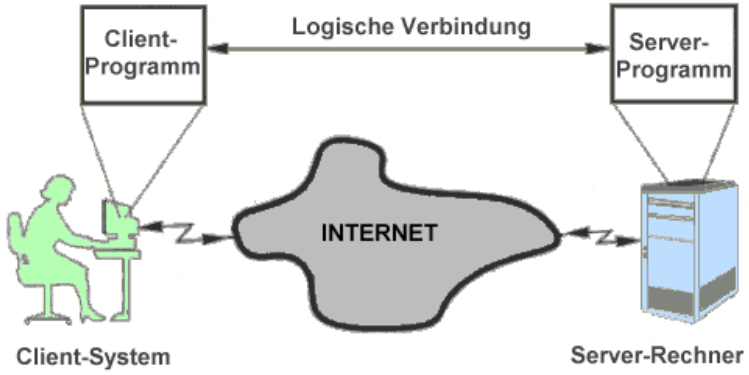
Cliente-servidor es el esquema dominante en las aplicaciones de procesamiento distribuido.

Los servidores son procesos que esperan pedidos de servicios de múltiples clientes. Unos y otros pueden ejecutarse en procesadores diferentes. Comunicación bidireccional. Atención de a un cliente o con multithreading a varios.

Mecanismos de invocación variados (rendezvous y RPC x ej en MD, monitores x ej en MC).
El soporte distribuido puede ser simple (LAN) o extendido a la WEB.





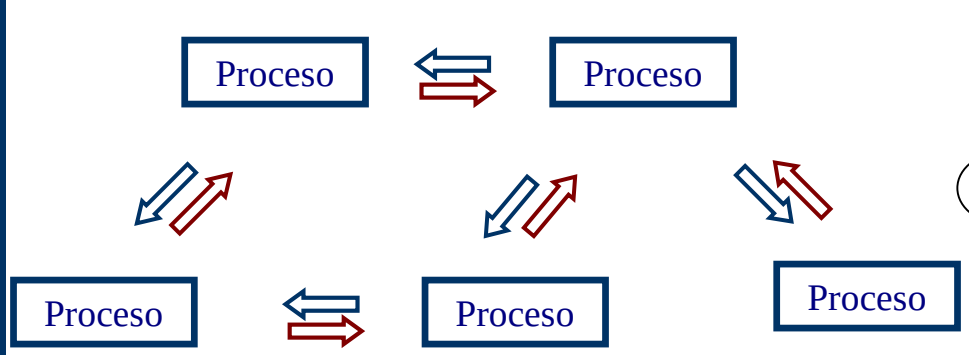


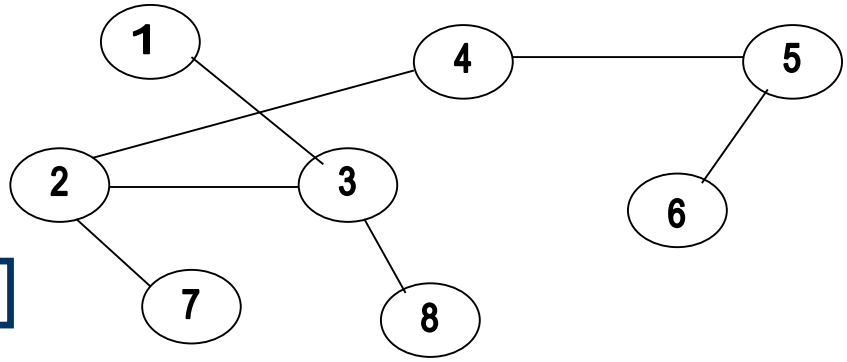
Paradigmas de resolución de programas concurrentes

En los esquemas de *pares que interactúan* los procesos (que forman parte de un programa distribuido) resuelven partes del problema (normalmente mediante código idéntico) e intercambian mensajes para avanzar en la tarea y completar el objetivo.

Permite mayor grado de asincronismo que C/S

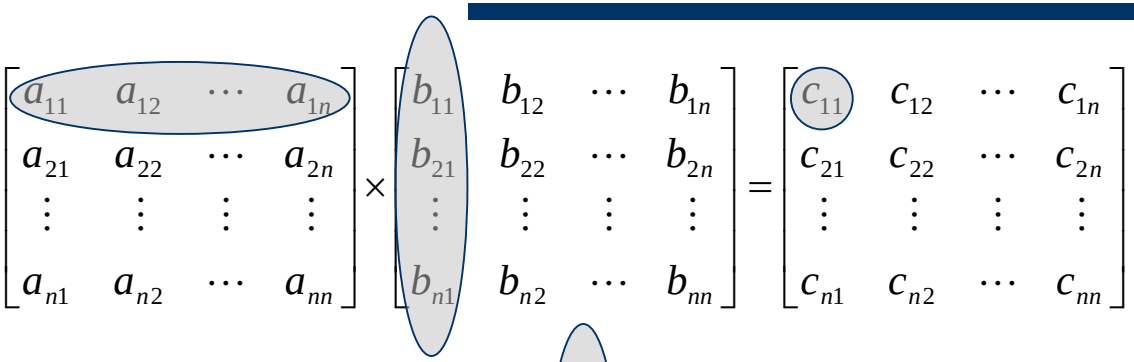
Configuraciones posibles: grilla, pipe circular, uno a uno, arbitraria

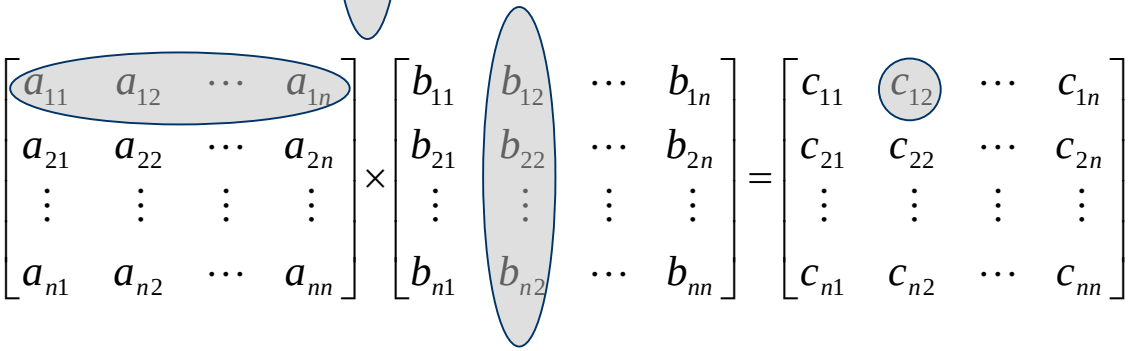


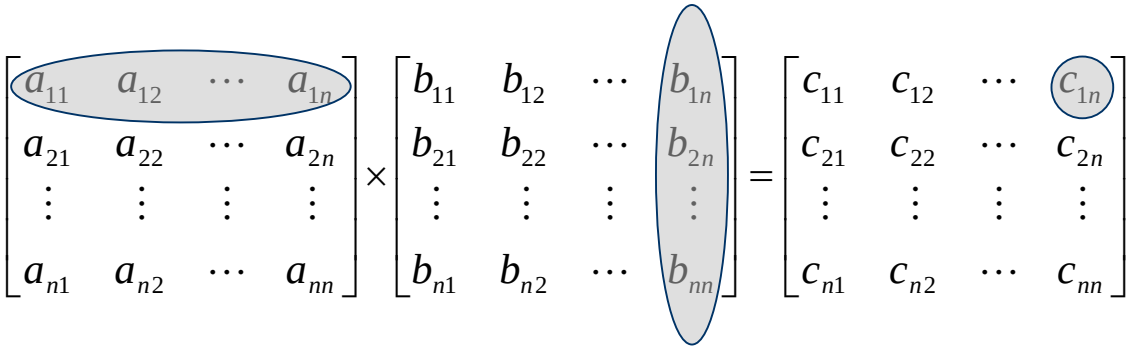


Pares que interactúan

Paralelismo iterativo: Multiplicación de matrices







Paralelismo iterativo: Multiplicación de matrices

Solución secuencial:

```
double a[n,n], b[n,n], c[n,n];
for [i = 1 to n] {
    for [j = 1 to n] {
       # computa el producto interno de a[i,*] y b[*,j]
       c[i,j] = 0.0;
       for [k = 1 to n]
          c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
 }
```

- El loop interno calcula el producto interno de la fila *i* de la matriz *a* por la columna *j* de la matriz *b* y obtiene *c*[*i*,,*j*].
- ***El cómputo de cada producto interno es independiente***

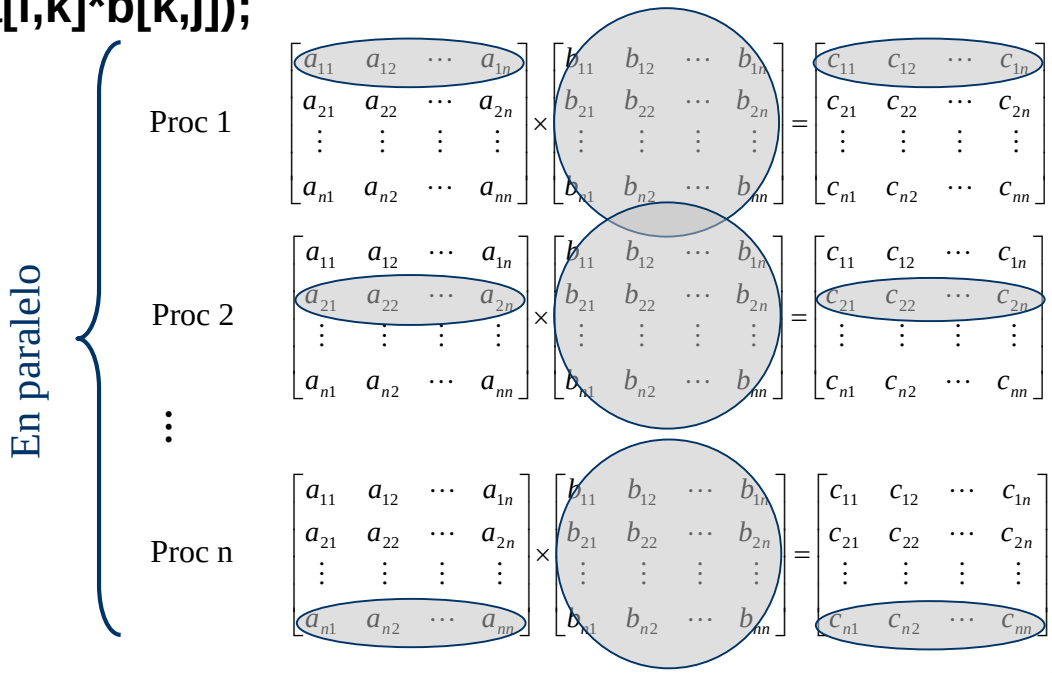
Aplicación *embarrasingly parallel.*

- Diferentes acciones paralelas posibles.

Paralelismo iterativo: Multiplicación de matrices

Solución paralela por fila:

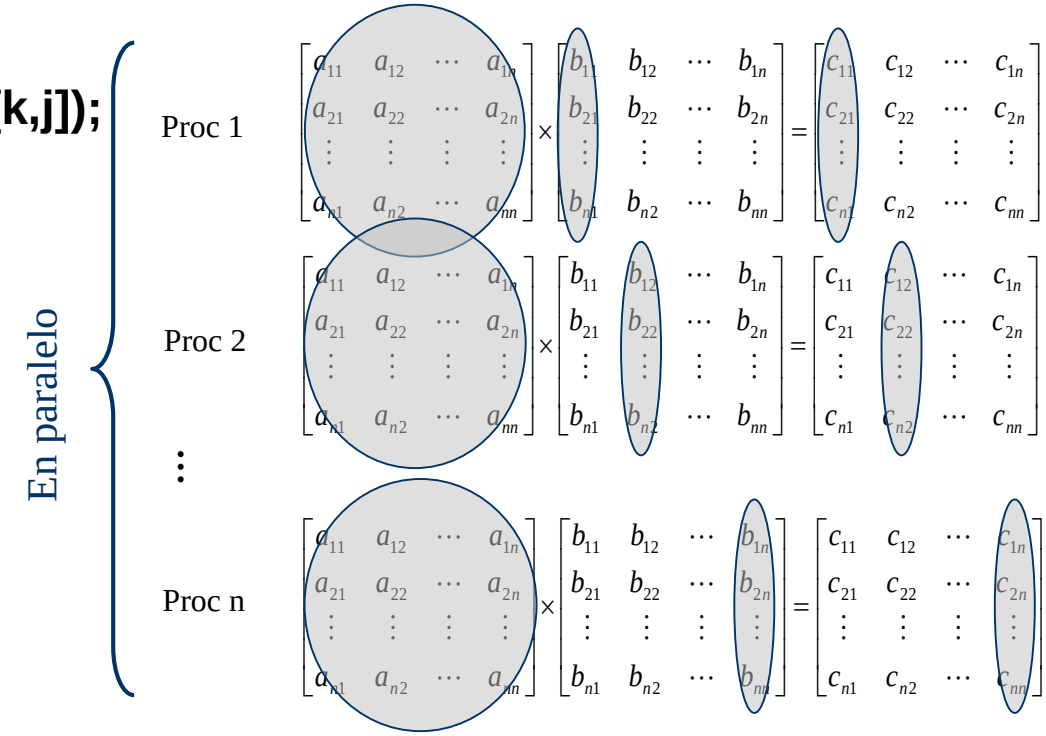
```
double a[n,n], b[n,n], c[n,n];
co [i = 1 to n]
  { for [j = 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
         c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
     }
  }
```



Paralelismo iterativo: Multiplicación de matrices

Solución paralela por columna:

```
double a[n,n], b[n,n], c[n,n];
co [j = 1 to n]
  { for [i = 1 to n]
      { c[i,j] = 0;
        for [k = 1 to n]
          c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
      }
  }
```



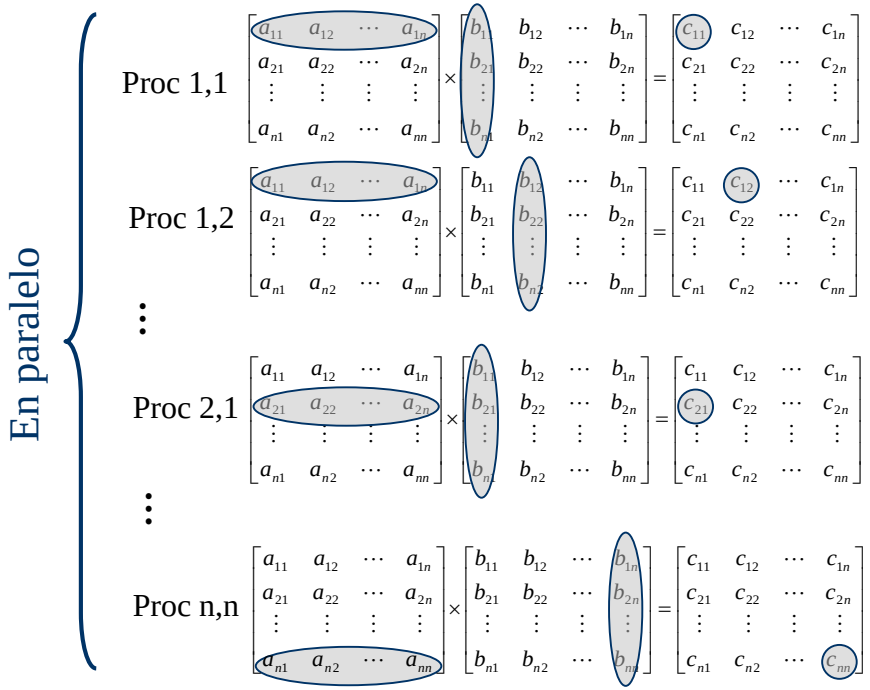
Paralelismo iterativo: Multiplicación de matrices

Solución paralela por celda (opción 1: TODAS las filas y columnas):

```
double a[n,n], b[n,n], c[n,n];
co [i = 1 to n , j= 1 to n]
    { c[i,j] = 0;
      for [k = 1 to n]
        c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
     }
```

Solución paralela por celda (opción 2: filas en paralelo, columnas en paralelo):

```
double a[n,n], b[n,n], c[n,n];
co [i = 1 to n]
  { co [j = 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
         c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
    }
  }
```



Multiplicación de matrices: uso de Process en lugar de *co*

Solución paralela por fila con process:

```
process fila [i = 1 to n]
  { for [j = 1 to n]
      { c[i,j] = 0;
        for [k = 1 to n]
          c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
      }
  }
```

Qué sucede si hay menos de *n* procesadores?

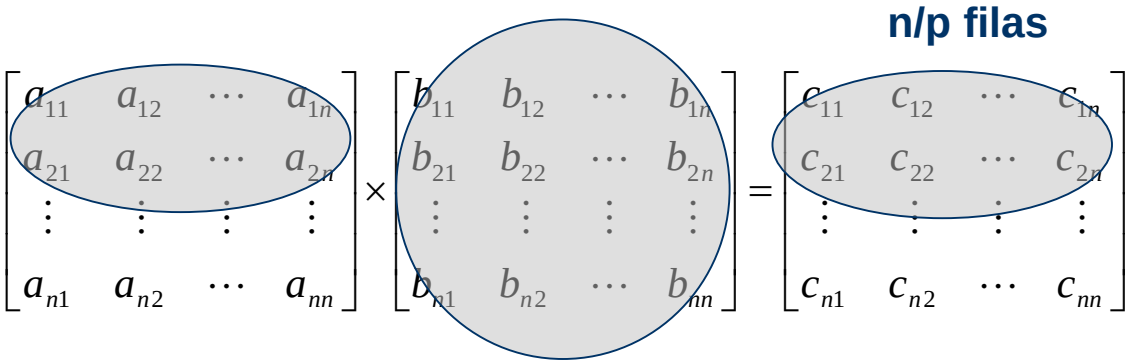
Se puede dividir la matriz resultado en *strips* (subconjuntos de filas o columnas) y usar un proceso ***worker*** por strip.

El tamaño del strip óptimo es un problema interesante para balancear costo de procesamiento con costo de comunicaciones.

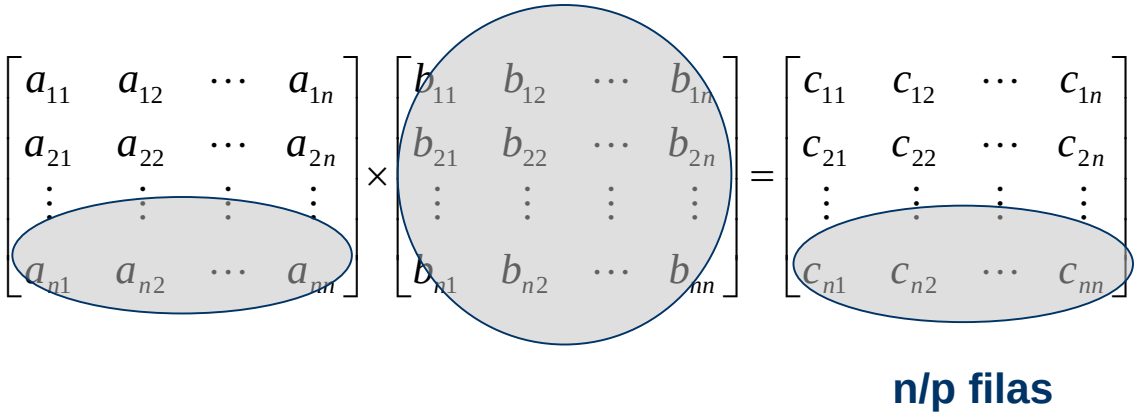
Paralelismo iterativo: Multiplicación de matrices

En paralelo

Worker 1



Worker p



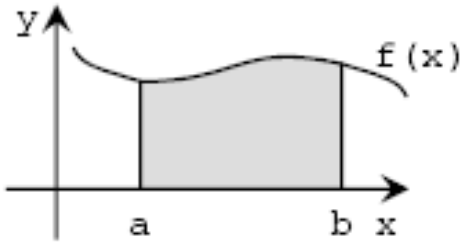
Multiplicación de matrices con *P* procesadores (*P* < *n*)

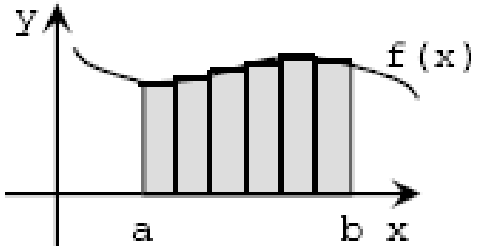
Solución paralela por strips: (*P* procesadores con *P*<*n*)

```
process worker [ w = 1 to P]
  { int primera = (w-1)*(n/P) + 1;
    int ultima = primera + (n/P) - 1;
    for [i = primera to ultima]
      { for [j = 1 to n]
          { c[i,j] = 0;
            for [k = 1 to n]
            c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
          }
      }
  }
```

Paralelismo recursivo: El problema de la cuadratura

Problema: calcular una aproximación de la integral de una función continua *f*(x) en el intervalo de *a* a *b*



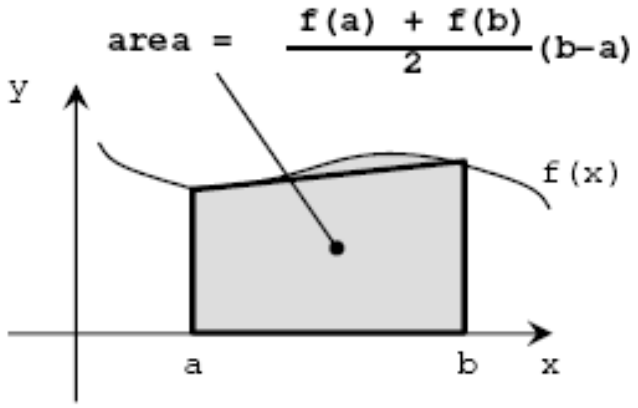


Solución secuencial iterativa (usando el método trapezoidal):

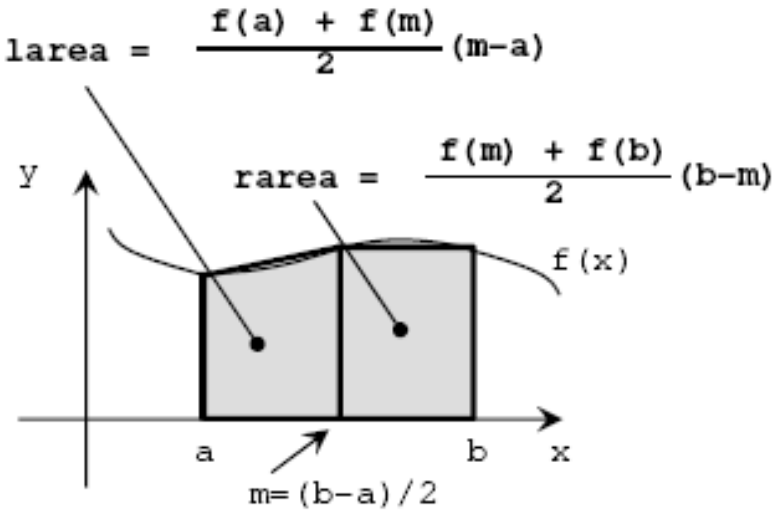
```
double fl = f(a), fr, area = 0.0;
double dx = (b-a)/ni;
for [x = (a + dx) to b by dx] {
   fr = f(x);
   area = area + (fl + fr) * dx / 2;
   fl = fr;
}
```

Paralelismo recursivo: El problema de la cuadratura

Procedimiento recursivo adaptivo



(a) First approximation (area)



(b) Second approximation (larea + rarea)

Si abs((larea + rarea) - area) > *e*, repetir el cómputo para cada intervalo [a,m] y [m,b] de manera similar hasta que la diferencia entre aproximaciones consecutivas esté dentro de un dado *e*

Paralelismo recursivo: El problema de la cuadratura

Procedimiento secuencial

```
double quad(double l,r,fl,fr,area) {
  double m = (l+r)/2;
  double fm = f(m);
  double larea = (fl+fm)*(m-l)/2;
  double rarea = (fm+fr)*(r-m)/2;
  if (abs((larea+rarea)-area) > e) {
     larea = quad(l,m,fl,fm,larea);
     rarea = quad(m,r,fm,fr,rarea);

  }
  return (larea+rarea);
}
```

Procedimiento paralelo

```
double quad(double l,r,fl,fr,area) {
   double m = (l+r)/2;
   double fm = f(m);
   double larea = (fl+fm)*(m-l)/2;
   double rarea = (fm+fr)*(r-m)/2;
   if (abs((larea+rarea)-area) > e) {
      co larea = quad(l,m,fl,fm,larea);
      || rarea = quad(m,r,fm,fr,rarea);
      oc
   }
   return (larea+rarea);
}
```

- **Dos llamados recursivos son independientes y pueden ejecutarse en paralelo**

- **Uso: area = quad (a, b ,f(a), f(b), (f(a) + f(b)) * (b-a) /2)**

Volviendo al hardware ...

... podemos identificar diferentes enfoques para clasificar las arquitecturas paralelas:

- por la organización del espacio de direcciones (memoria compartida / memoria distribuida)

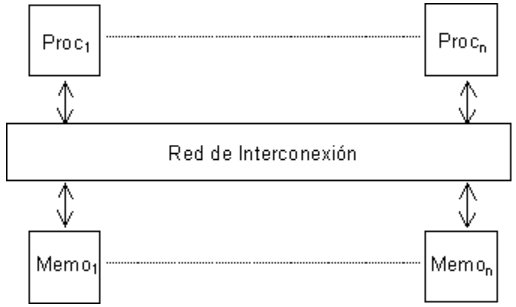
- por el mecanismo de control

- por la granularidad

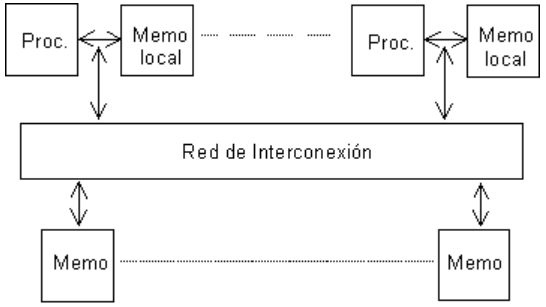
- por la red de interconexión

Clasificación por la organización del espacio de direcciones

- **Multiprocesadores de memoria compartida.**
  - Interacción modificando datos en la memoria compartida.
  - Problema de consistencia.



Esquema UMA



Esquema NUMA

- **Multiprocesadores con memoria distribuida.**
  - Memoria local (no hay problemas de consistencia).
  - Interacción es sólo por pasaje de mensajes.

Clasificación por mecanismo de control

Propuesta por Flynn (“Some computer organizations and their effectiveness”, 1972).

Se basa en la manera en que las *instrucciones* son ejecutadas sobre los *datos.*

$\Rightarrow$ 4 clases:

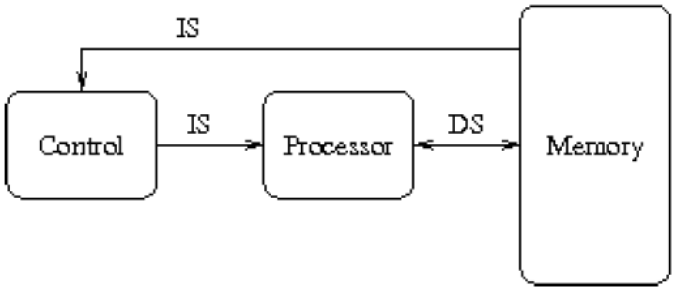
- SISD (Single Instruction Single Data)
- SIMD (Single Instruction Multiple Data)
- MISD (Multiple Instruction Single Data)
- MIMD (Multiple Instruction Multiple Data)

Clasificación por mecanismo de control

SISD: Single Instruction Single Data

Instrucciones ejecutadas en secuencia, una x ciclo de instrucción.

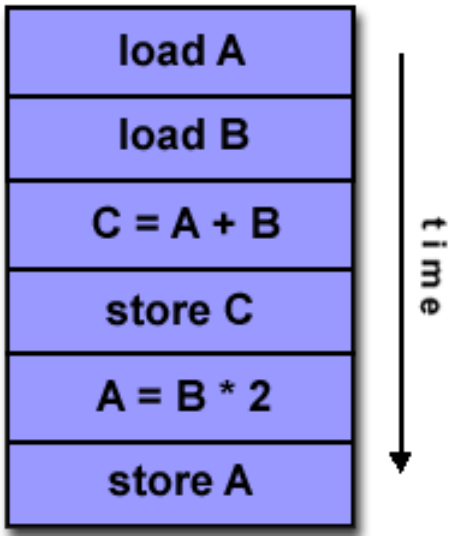
La memoria afectada es usada sólo por esta instrucción



Usada por la mayoría de los uniprocesadores.

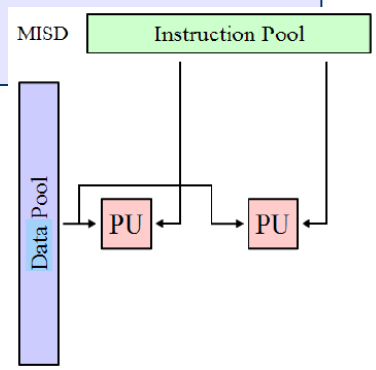
La CPU ejecuta instrucciones (decodificadas por la UC) sobre los datos. La memo recibe y almacena datos en las escrituras, y brinda datos en las lecturas.

Ejecución *determinística*



Clasificación por mecanismo de control

MISD: Multiple Instruction Single Data



Los procesadores ejecutan un flujo de instrucciones distinto pero comparten datos comunes

Operación sincrónica (en *lockstep*)

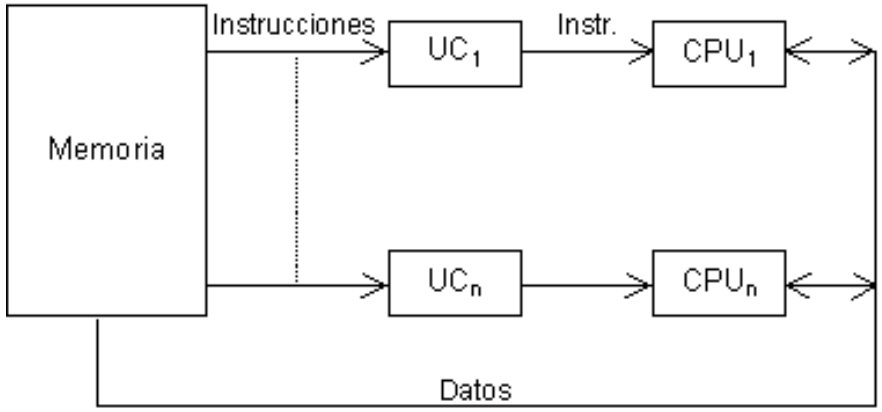
No son máquinas de propósito general ("hipotéticas", Duncan)

Ejemplos posibles:

- tolerancia a fallas

- múltiples filtros de frecuencia operando sobre una única señal

- múltiples algoritmos de criptografía intentando crackear un único mensaje codificado



Clasificación por mecanismo de control

SIMD: Single Instruction Multiple Data

Conjunto de procesadores idénticos, con sus memorias, que ejecutan la misma instrucción sobre distintos datos

Los procesadores en gral son muy simples.

El host hace broadcast de la instr. Ejecución sincrónica y determinística.

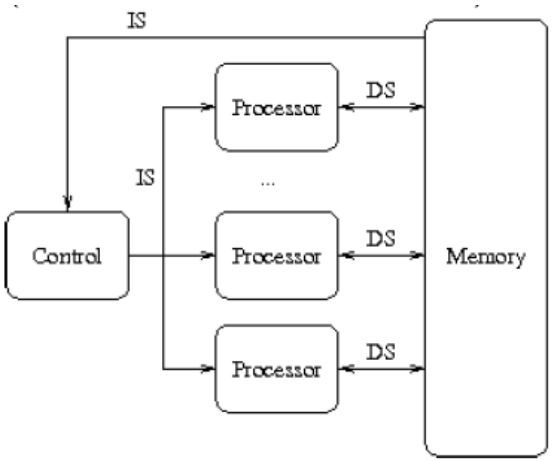
Pueden deshabilitarse y habilitarse selectivamente procesadores para que ejecuten o no instrucciones

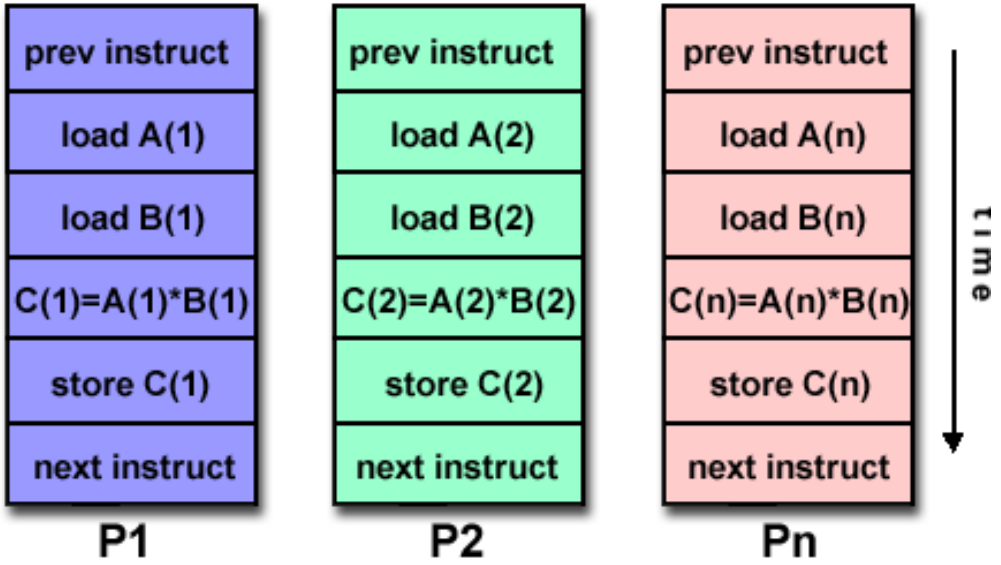
Adecuados para aplicaciones con alto grado de regularidad, (x ej. procesamiento de imágenes).

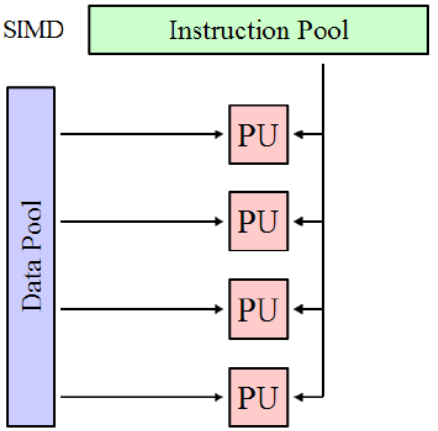
Clasificación por mecanismo de control

SIMD: Single Instruction Multiple Data

Ej: Array Processors. CM-2, Maspar MP-1 y 2, Stream processors en GPU

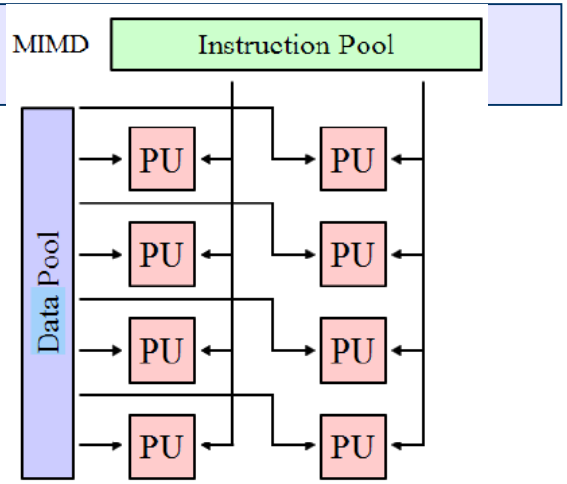






Clasificación por mecanismo de control

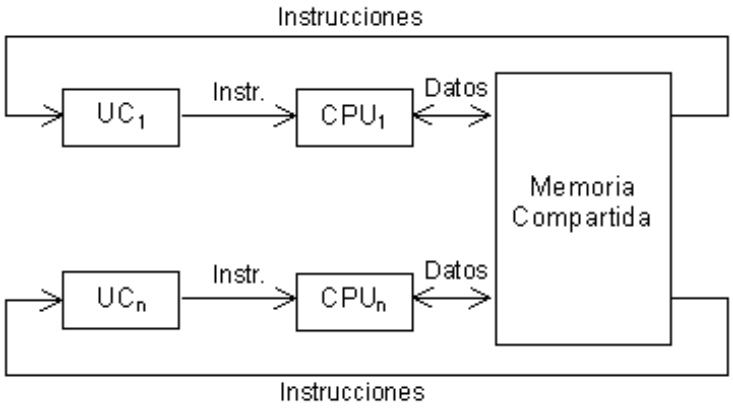
MIMD: Multiple Instruction Multiple Data

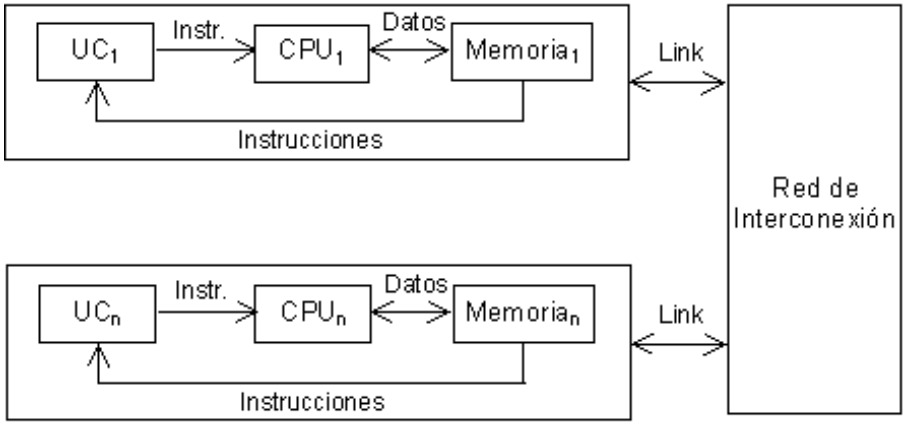


C/ procesador tiene su propio flujo de instrucciones y de datos

⇒ c/u ejecuta su propio programa

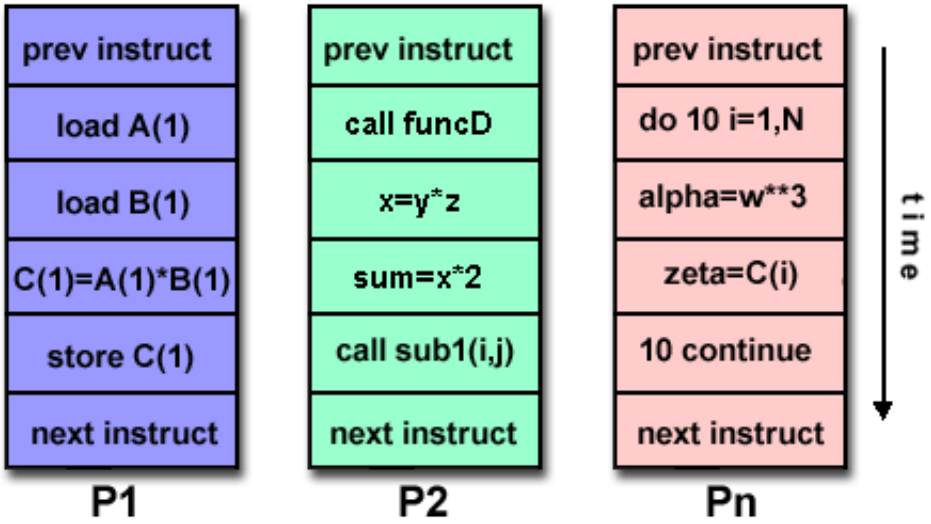
Pueden ser con memoria compartida o distribuida





Clasificación por mecanismo de control

MIMD: Multiple Instruction Multiple Data



Ejemplos: nCube 2, iPSC, CM-5, Paragon XP/S, máquinas DataFlow, red de transputers.

Sub-clasificación de MIMD:

- ***MPMD*** (multiple program multiple data): c/ procesador ejecuta su propio programa (ejemplo con PVM).
- ***SPMD*** (single program multiple data): hay un único programa fuente y cada procesador ejecuta su copia independientemente (ejemplo con MPI).

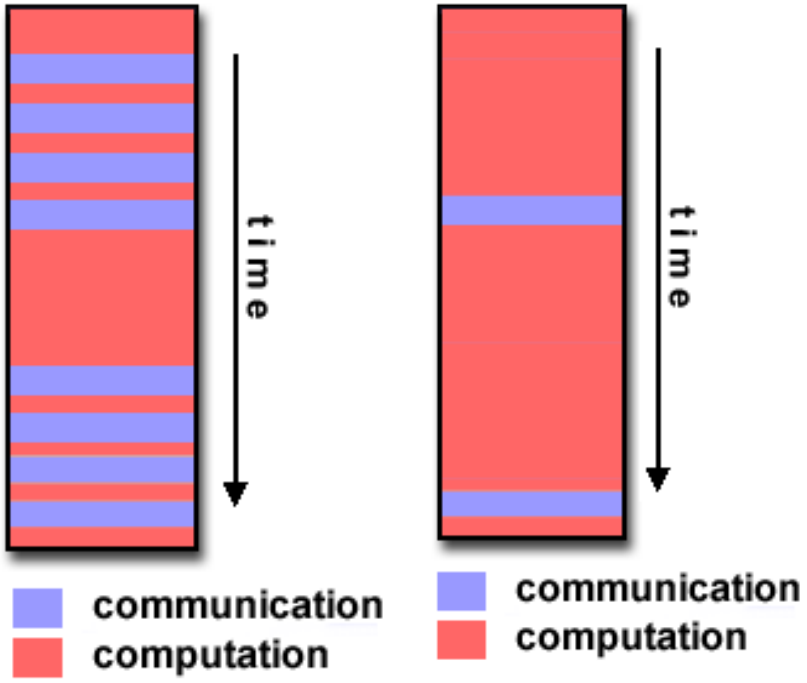
Clasificación por la granularidad de los procesadores

Granularidad

Relación entre el número de procesadores y el tamaño de memoria total.

Grano fino y grano grueso

Puede verse también como la relación entre cómputo y comunicación



Clasificación por la granularidad de los procesadores

De grano grueso (*coarse-grained*): pocos procesadores muy poderosos.

De grano fino (*fine-grained*): gran número de procesadores menos potentes.

De grano medio (*medium-grained*).

Aplicaciones adecuadas para una u otra clase:

- si tienen concurrencia limitada pueden usar eficientemente pocos procesadores $\Rightarrow$ convienen máquinas de grano grueso
- las máquinas de grano fino son más efectivas en costo para aplicaciones con alta concurrencia

$\Rightarrow$ Es importante el *matching* entre la arquitectura y la aplicación...

Clasificación por la red de interconexión

Tanto las máquinas de MC como de MP pueden construirse conectando procesadores y memorias usando diversas redes de interconexión, *estáticas* y *dinámicas*.

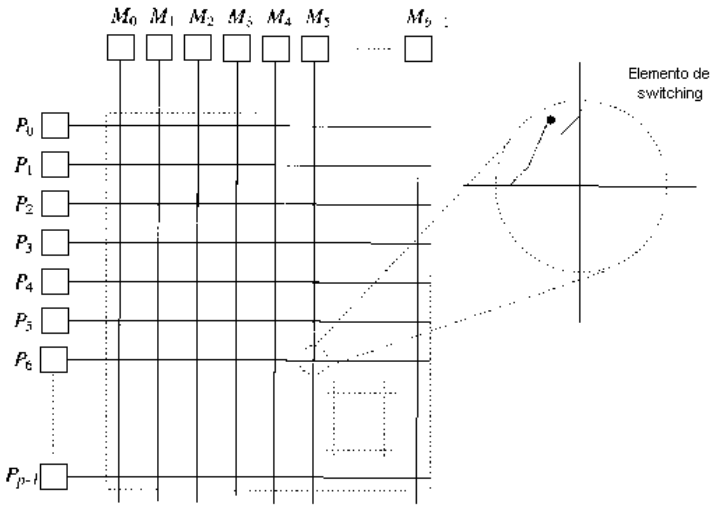
Las redes **estáticas** constan de *links* punto a punto. Típicamente se usan para máquinas de MP

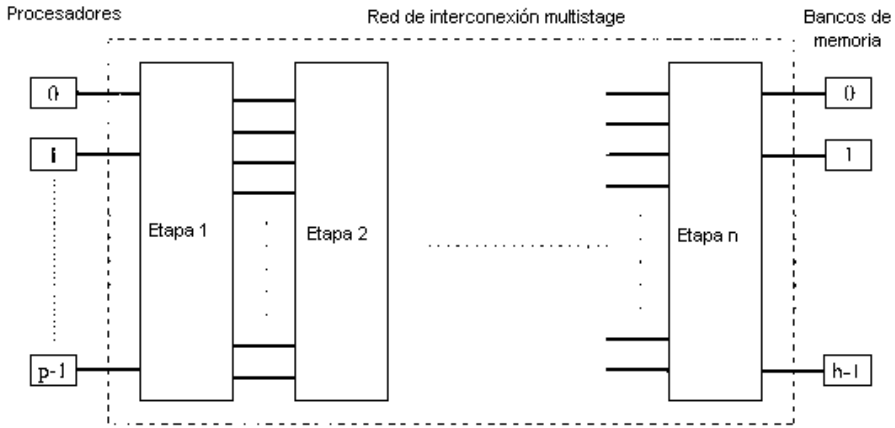
Las redes **dinámicas** están construidas usando switches y enlaces de comunicación. Normalmente para máquinas de MC

El *diseño* de la red de interconexión depende de una serie de factores (ancho de banda, tiempo de startup, paths estáticos o dinámicos, operación sincrónica o asincrónica, topología, ...)

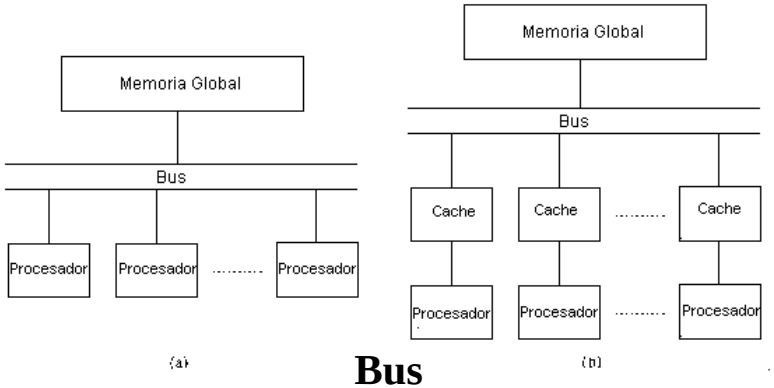
Clasificación por la red de interconexión

Redes de interconexión dinámicas

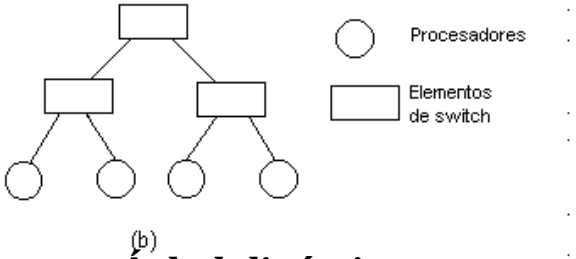




Multistage



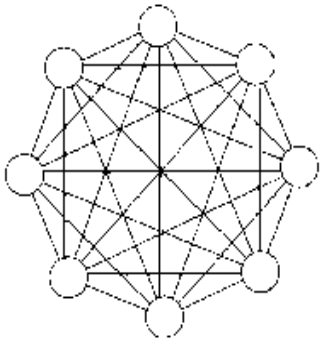
Bus



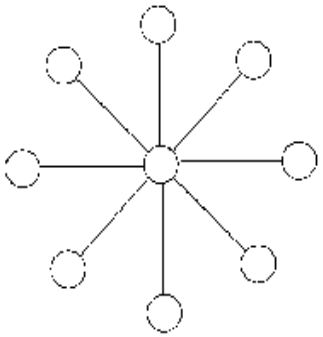
Árbol dinámico

Clasificación por la red de interconexión

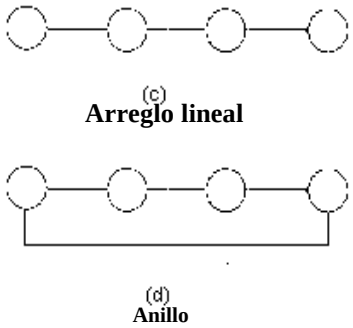
Redes de interconexión estáticas: topología



(a)
Completamente conectada

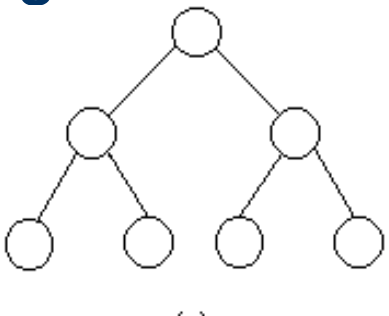


(b)
Estrella

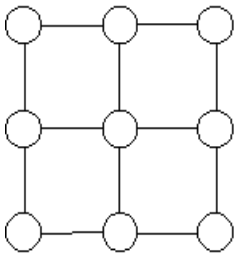


(c)
Arreglo lineal

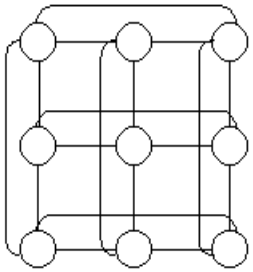
(d)
Anillo



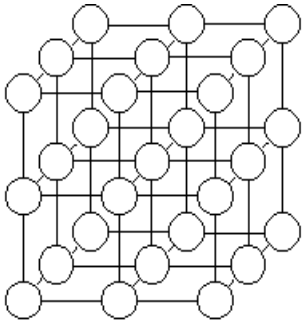
(a)
Árbol estático



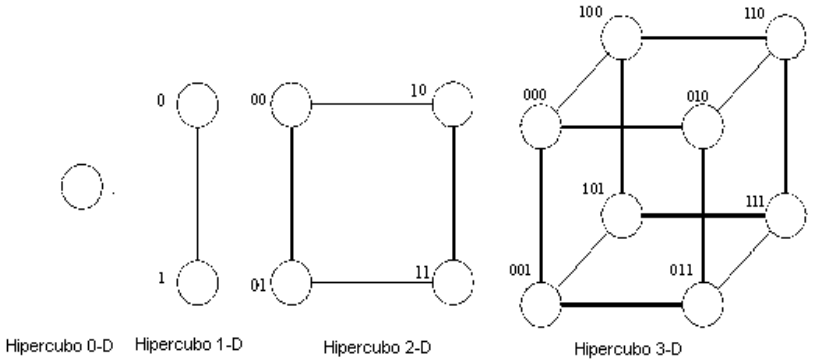
(a)
Mesh



(b)
Toro



(c)
Mesh 3D



Un hipercubo d-dimensional tiene $p=2^d$ procesadores

Acciones Atómicas y Sincronización

- Estado de un Programa concurrente
- Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas.
- *Una* ***acción atómica*** hace una transformación de estado indivisibles (estados intermedios invisibles para otros procesos).
- Ejecución de un programa concurrente → ***intercalado*** (*interleaving*) de las acciones atómicas ejecutadas por procesos individuales.
- ***Interacción*** → no todos los interleavings son aceptables.
- ***Historia*** de un programa concurrente (*trace*).

Acciones atómicas y Sincronización

```
process 1 {                          process 2 {
      while (true)                         while (true)
   p11:  read (x)                       p21:  y = buffer
   p12:  buffer = x;  }                 p22:  print (y);  }
```

Posibles interleavings (historias):

p11, p12, p21, p22, p11, p12, p21, p22, ... ☑

p11, p12, p21, p11, p22, p12, p21, p11, p12, p22, ... ☑

p11, p21, p12, p22, ... ☒

p21, p11, p12, ... ☒

Hay algunas historias que no son válidas…

Acciones atómicas y Sincronización

Sincronizar ⇒ Combinar acciones atómicas de grano fino (fine-grained) en acciones (compuestas) de grano grueso (coarse grained) que den la exclusión mutua.

Sincronizar ⇒ Demorar un proceso hasta que el estado de programa satisfaga algún predicado (por condición).

El objetivo de la sincronización es prevenir los interleavings indeseables restringiendo las historias de un programa concurrente sólo a las permitidas

Acciones atómicas y Sincronización. Atomicidad de grano fino

Una acción atómica de grano fino se debe implementar por hardware

La operación de asignación A=A+3 es atómica?

NO $\Rightarrow$ Es necesario cargar el valor de A en un registro, sumarle 3 y volver a cargar el valor del registro en A

Qué sucede con algo del tipo X=X+X?

$\Rightarrow$ (i) Load PosMemX, Acumulador
(ii) Add PosMemX, Acumulador
(iii) Store Acumulador, PosMemX

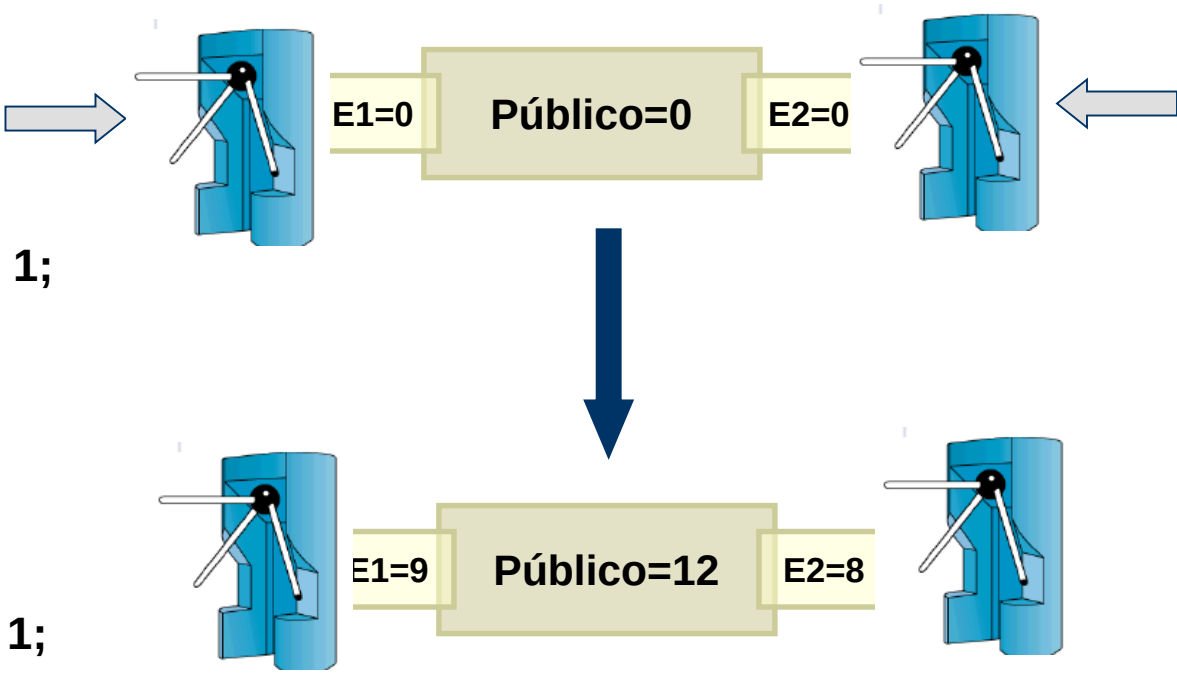
Acciones atómicas y Sincronización. El problema de la *interferencia*

Interferencia: un proceso toma una acción que invalida las suposiciones hechas por otro proceso.

Ejemplo: ¿Qué puede suceder con los valores de E1, E2 y público?

```
process 1
 { while (true)
       esperar llegada
       E1 = E1 + 1;
       Público = Público + 1;
 }

process 2
 { while (true)
       esperar llegada
       E2 = E2 + 1;
       Público = Público + 1;
 }
```



Acciones atómicas y Sincronización. Atomicidad de grano fino

La lectura y escritura de las variables x,y,z son atómicas.

```
y = 4;  x = 0; z=2;
co
    x = y + z          (1)
 // y = 3              (2)
 // z = 4              (3)
oc
```

⇒ Cuáles son los posibles resultados con hasta 3 procesadores (no necesariamente de igual velocidad) ejecutando los procesos

(1) Puede descomponerse por ejemplo en:

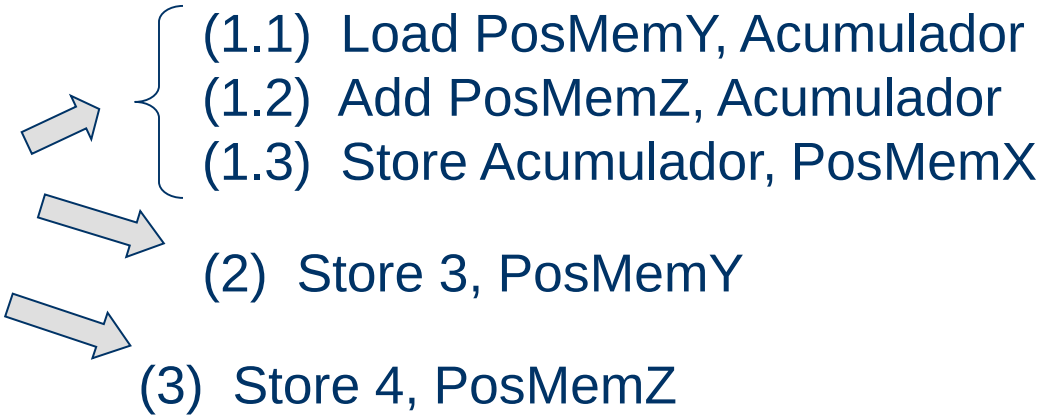
(1.1) Load PosMemY, Acumulador

(1.2) Add PosMemZ, Acumulador

(1.3) Store Acumulador, PosMemX

Acciones atómicas y Sincronización. Atomicidad de grano fino

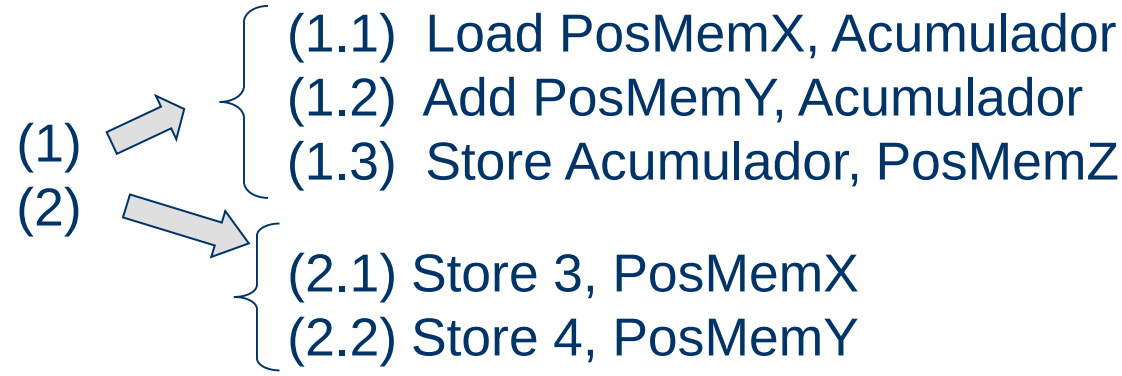
```
y = 4;  x = 0; z=2;
co
    x = y + z          (1)
 // y = 3               (2)
 // z = 4               (3)
oc
```



y = 3, z = 4 en todos los casos
x puede ser:
6 si ejecuta (1)(2)(3) o (1)(3)(2)
5 si ejecuta (2)(1)(3)
6 si ejecuta (3)(1)(2)
7 si ejecuta (2)(3)(1) o (3)(2)(1)
6 si ejecuta (1.1)(2)(1.2)(1.3)(3)
8 si ejecuta (1.1)(3)(1.2)(1.3)(2)
.....

Acciones atómicas y Sincronización. Atomicidad de grano fino

```
x = 2; y = 2;
co
    z = x + y
 // x = 3; y = 4
oc
```



x = 3, y = 4 en todos los casos

z puede ser 4, 5, 6 o 7

Pero nunca podríamos parar el programa y ver un estado en que x+y=6 !!!!

Acciones atómicas y Sincronización. Atomicidad de grano fino

“Interleaving extremo”

(Ben-Ari & Burns)

Dos procesos que realizan (c/u) *k* iteraciones de la sentencia N=N+1 (N compartida init 0)

```
Process P1{                          Process P2{
   fa i=1 to K → N=N+1 af               fa i=1 to K → N=N+1 af
   }                                    }
```

Cuál puede ser el valor final de N?

- 2K
- entre K+1 y 2K-1
- K
- <K (incluso 2…)

Acciones atómicas y Sincronización. Atomicidad de grano fino

Cuándo valdrá k?

1. Proceso 1: Load N
2. Proceso 2: Load N
3. Proceso 1: Incrementa su copia
4. Proceso 2: Incrementa su copia
5. Proceso 1: Store N
6. Proceso 2: Store N

Acciones atómicas y Sincronización. Atomicidad de grano fino

Cuándo valdrá 2?

1. Proceso 1: Load N
2. Proceso 2: Hace k-1 iteraciones del loop
3. Proceso 1: Incrementa su copia
4. Proceso 1: Store N
5. Proceso 2: Load N
6. Proceso 1: Hace k-1 iteraciones del loop
7. Proceso 2: Incrementa su copia
8. Proceso 2: Store N

… no podemos confiar en la intuición para analizar un programa concurrente…

Acciones atómicas y Sincronización. Atomicidad de grano fino

En lo que sigue, supondremos máquinas con las siguientes características:

Los valores de los tipos básicos se almacenan en elementos de memoria leídos y escritos como acciones atómicas

Los valores se cargan en registros, se opera sobre ellos, y luego se almacenan los resultados en memoria

Cada proceso tiene su propio conjunto de registros (conjuntos disjuntos o context switching)

Todo resultado intermedio de evaluar una expresión compleja se almacena en registros o en memoria privada del proceso

Acciones atómicas y Sincronización. Atomicidad de grano fino

⇒ Si una expresión **e** en un proceso no referencia una variable alterada por otro proceso, la evaluación será atómica, aunque requiera ejecutar varias acciones atómicas de grano fino.

⇒ *Si una asignación **x = e** en un proceso no referencia ninguna variable alterada por otro proceso, la ejecución de la asignación será atómica*

Pero ... normalmente los programas concurrentes no son disjuntos
⇒ Es necesario establecer algún requerimiento más débil ...

Acciones atómicas y Sincronización. Propiedad de *"A lo sumo una vez"*

Referencia crítica en una expresión ⇒ referencia a una vble que es modificada por otro proceso.
Asumamos que toda referencia crítica es a una variable simple leída y escrita atómicamente.

*Una sentencia de asignación **x = e** satisface la propiedad de A lo sumo una vez si*
*(1) **e** contiene a lo sumo una referencia crítica y **x** no es referenciada por otro proceso, o*
*(2) **e** no contiene referencias críticas, en cuyo caso **x** puede ser leída por otro proceso*

Puede haber a lo sumo una variable compartida, y puede ser referenciada a lo sumo una vez.
Una def. similar se aplica a expresiones que no están en sentencias de asignación (satisface ASV si no contiene más de una ref. crítica)

Acciones atómicas y Sincronización. Propiedad de *"A lo sumo una vez"*

EFECTO: Si una sentencia de asignación cumple la propiedad ASV, entonces su ejecución *parece* atómica, pues la variable compartida será leída o escrita sólo una vez.

Ejemplos:

int x=0, y=0; co x=x+1 // y=y+1 oc;	No hay ref. críticas en ningún proceso x = 1 e y = 1 ∀ historia
int x=0, y=0; co x=y+1 // y=y+1 oc;	El 1er proc tiene 1 ref. crítica. El 2do ninguna. y = 1 siempre, x = 1 o 2
int x=0, y=0; co x=y+1 // y=x+1 oc;	Ninguna asignación satisface ASV Posibles: x=1, y=2 / x=2, y=1 / x=1, y=1 Pero no podría ocurrir!!!

Especificación de la sincronización

Si una expresión o asignación no satisface ASV con frecuencia es necesario ejecutarla atómicamente

En general, es necesario ejecutar secuencias de sentencias como una única acción atómica

Una acción atómica de grano grueso es una especificación en alto nivel del comportamiento requerido de un programa que puede ser implementada de distintas maneras, dependiendo del mecanismo de sincronización disponible.

⇒ Mecanismo de sincronización para construir una acción atómica *de* grano grueso (*coarse grained*) como secuencia de acciones atómicas de grano fino (*fine grained*) que aparecen como indivisibles

Ej: BD con dos valores que en todo momento deben ser iguales

Ej: Productor y consumidor con una lista enlazada

Especificación de la sincronización

⟨**e**⟩ indica que la expresión ***e*** debe ser evaluada atómicamente

⟨**await (B) S;**⟩ se utiliza para especificar sincronización

La expresión booleana B especifica una condición de demora.
S es una secuencia de sentencias que se garantiza que termina.
Se garantiza que B es true cuando comienza la ejecución de S.
Ningún estado interno de S es visible para los otros procesos.

Ej: ⟨**await (s>0) s=s-1;**⟩

Sentencia con alto poder expresivo, pero el costo de implementación de la forma general de await (EM y SxC) es alto

Especificación de la sincronización

Sólo exclusión mutua ⇒ ⟨S⟩
Ej: ⟨x = x + 1; y = y + 1⟩
El estado interno en el cual x e y son incrementadas resulta invisible a otros procesos que las referencian

Sólo sincronización por condición ⇒ ⟨await (B)⟩
Ej: ⟨await (count > 0) ⟩

Si B satisface ASV, puede implementarse como *busy waiting* o *spinning*: **do (not B) → skip od** **(while (not B);)**

Acciones atómicas incondicionales y condicionales

Especificación de la sincronización

Ejemplo: productor/consumidor con buffer de tamaño N.

```
cant: int = 0;
Buffer: cola;

process Productor
 { while (true)
      <await  (cant < N); push(buffer, elemento); cant++ >
 }

process Consumidor
 { while (true)
      <await  (cant > 0); pop(buffer, elemento); cant-- >
 }
```

Qué pasa si se guardan en un arreglo??

Seguridad y vida

Una *propiedad* de un programa concurrente es un atributo verdadero en cualquiera de las historias de ejecución del mismo

Toda propiedad puede ser formulada en términos de dos clases: seguridad y vida.
Son dos aspectos complementarios de la *corrección*

seguridad **(safety): n**ada malo le ocurre a un objeto: asegura estados consistentes

- una *falla de seguridad* indica que algo anda mal
- Ej: ausencia de deadlock y ausencia de interferencia (exclusión mutua) entre procesos.

vida **(liveness):** eventualmente ocurre algo bueno con una actividad: progresa, no hay deadlocks

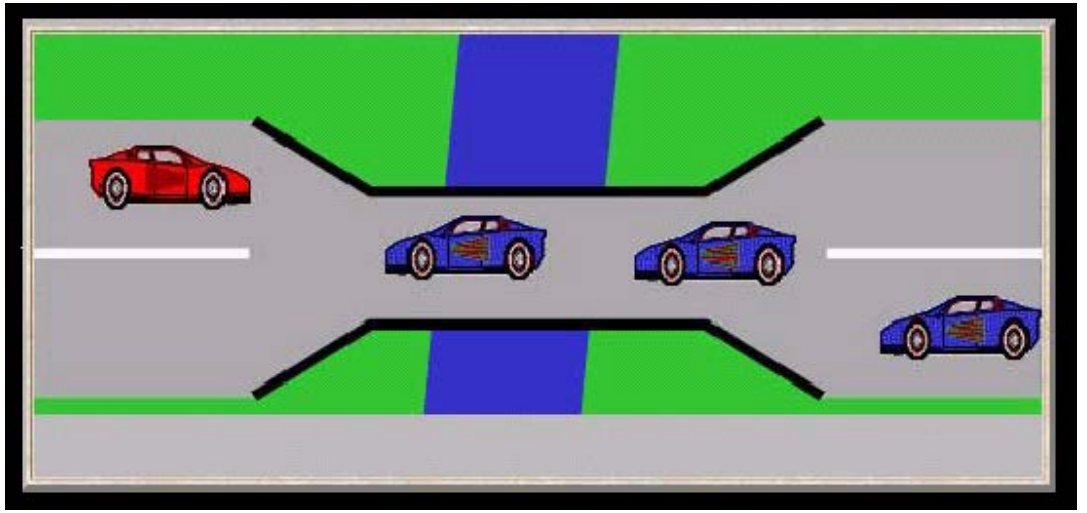
- una *falla de vida* indica que se deja de ejecutar
- Ej: *terminación*, *asegurar que un pedido de servicio será atendido, que un mensaje llega a destino, que un proceso eventualmente alcanzará su SC, etc* $\Rightarrow$ ***dependen de las políticas de scheduling.***

Seguridad y vida

Ejemplo: Puente de una sola vía

Puente sobre río con ancho sólo para una fila de tráfico $\Rightarrow$ los autos pueden moverse concurrentemente si van *en la misma dirección*

- Violación de *seguridad* si dos autos en distintas direcciones entran al puente al mismo tiempo



- Vida: c/ auto tendrá *eventualmente* oportunidad de cruzar el puente?

Los temas de seguridad deben balancearse con los de vida

Seguridad y vida

Seguridad → Fallas típicas (*race conditions*):

- Conflictos de read/write: un proceso lee un campo y otro lo escribe (el valor visto por el lector depende de quién ganó la "carrera").
- Conflictos de write/write: dos procesos escriben el mismo campo (quién gana la "carrera").

Vida → Fallas:

- *Temporarias*: Bloqueo temporarios, Espera, Contención de CPU, Falla recuperable.
- *Permanente*: Deadlock, Señales perdidas, Anidamiento de bloqueos, Livelock, Inanición, Agotamiento de recursos, Falla distribuida.

Fairness y políticas de scheduling

***Fairness*: trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los demás**

Una acción atómica en un proceso es elegible si es la próxima acción atómica en el proceso que será ejecutado
Si hay varios procesos ⇒ hay *varias acciones atómicas elegibles*

Una *política de scheduling* determina cuál será la próxima en ejecutarse

Política: asignar un procesador a un proceso hasta que termina o se demora. Qué sucede en este caso??

```
bool continue = true;
co while (continue);
   // continue = false;
oc
```

Fairness y políticas de scheduling

Fairness Incondicional. Una política de scheduling es incondicionalmente fair (o *imparcial*) si toda acción atómica incondicional que es elegible eventualmente es ejecutada.

En el ej. anterior, RR es incondicionalmente fair en monoprocesador, y la ejecución paralela lo es en un multiprocesador (si ningún procesador puede monopolizar el acceso a la vble compartida)

Fairness Débil. Una política de scheduling es débilmente fair si (1) es incondicionalmente fair y (2) toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve true y permanece true hasta que es vista por el proceso que ejecuta la acción atómica condicional

RR es débilmente fair en el ej anterior

No es suficiente para asegurar que cualquier sentencia await elegible eventualmente se ejecuta: la guarda podría cambiar el valor (de false a true y nuevamente a false) mientras un proceso está demorado.

Fairness y políticas de scheduling

Fairness Fuerte**:* Una política de scheduling es ***fuertemente fair si (1) es incondicionalmente fair y (2) toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en true con infinita frecuencia.

Este programa termina??

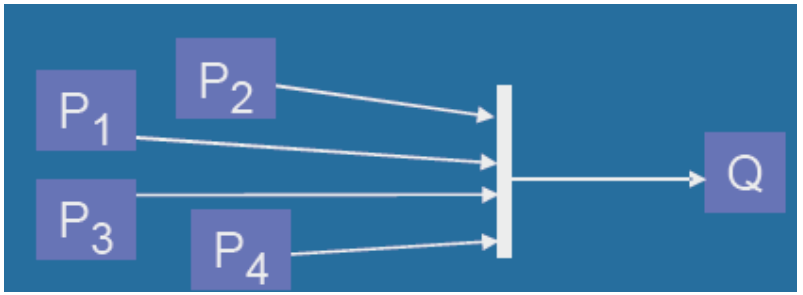
```
bool continue = true, try = false;
co while (continue) { try = true; try = false; }
 // ⟨await (try) continue = false ⟩
oc
```

No es simple tener una política que sea práctica y fuertemente fair.
En el ej anterior, con 1 procesador, una política que alterna las acciones de los procesos sería fuertemente fair, pero es impráctica.
Round-robin es práctica pero no es fuertemente fair.

Tipos de Problemas Básicos de Concurrencia

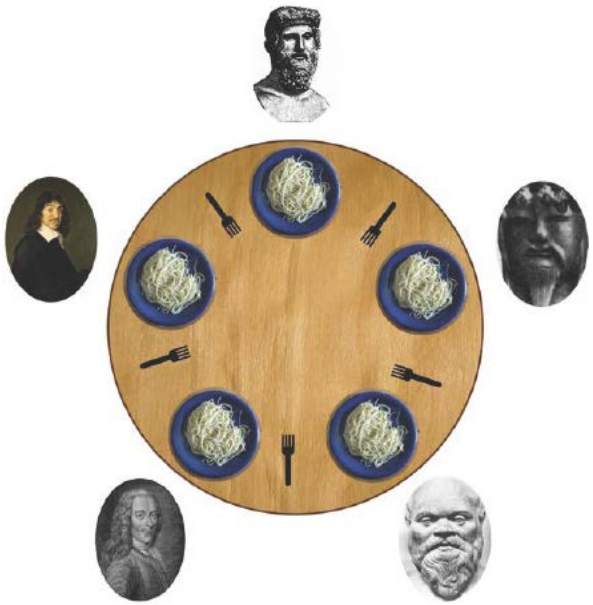
***Exclusión mutua**:* problema de la sección crítica. (Administración de recursos).

Barreras: punto de sincronización



Comunicación

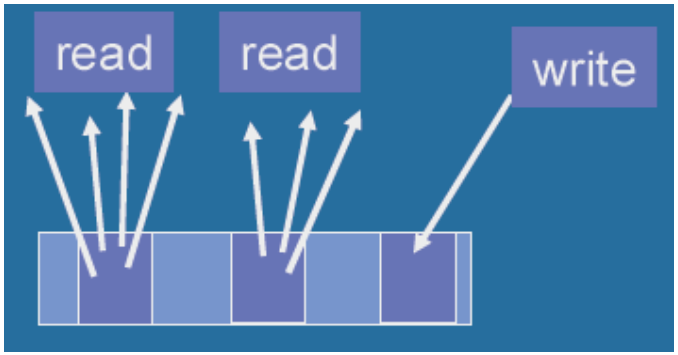
Filósofos: Dijkstra, 1971.
Sincronización multiproceso.
Evitar deadlock e inanición.
Exclusión mutua selectiva



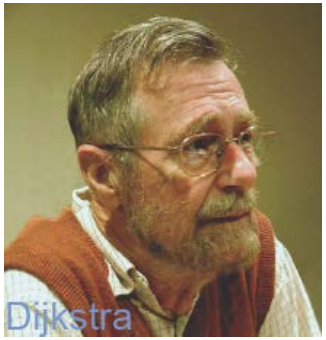
Tipos de Problemas Básicos de Concurrencia

Productor-consumidor

Lectores-escritores



Sleeping barber: Dijkstra.
Sincronización – rendezvous.



Tareas propuestas

- Leer los capítulos 2 y 3 del libro de Andrews

- Investigar el problema de la *sección crítica* y algunas posibles soluciones al mismo