

# Programación Concurrente ATIC

## Programación Concurrente (redictado)

### Clase 2



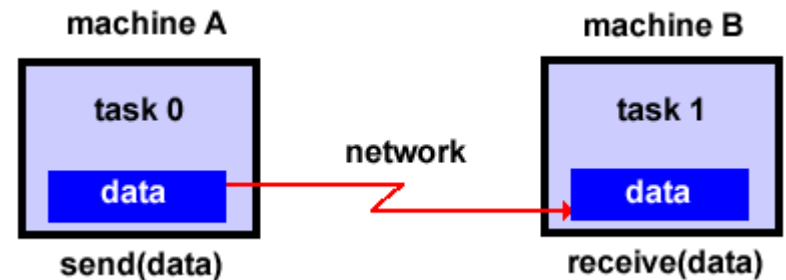
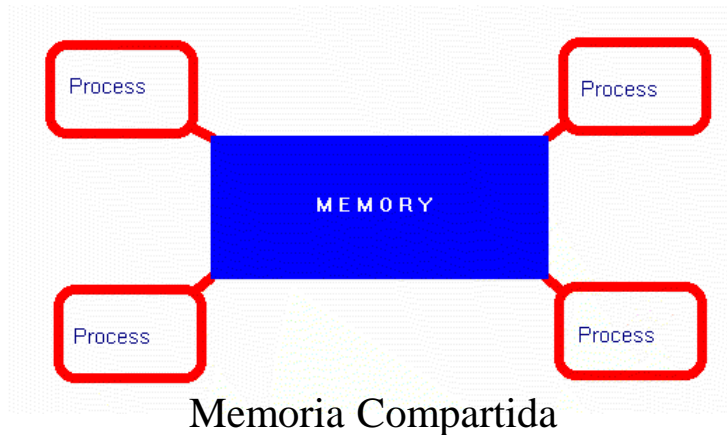
Facultad de Informática  
UNLP

# Conceptos básicos de concurrencia

## Comunicación entre procesos

La comunicación entre procesos concurrentes indica el modo en que se organiza y transmiten datos entre tareas concurrentes. Esta organización requiere especificar *protocolos* para controlar el progreso y la corrección. Los procesos se **COMUNICAN**:

- Por *Memoria Compartida*.
- Por *Pasaje de Mensajes*.



Pasaje de Mensajes

# Conceptos básicos de concurrencia

## Comunicación entre procesos

- **Memoria compartida**

- Los procesos intercambian información sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella.
- Lógicamente no pueden operar simultáneamente sobre la memoria compartida, lo que obliga a **bloquear y liberar** el acceso a la memoria.
- La solución más elemental es una variable de control tipo “semáforo” que habilite o no el acceso de un proceso a la memoria compartida.

- **Pasaje de mensajes**

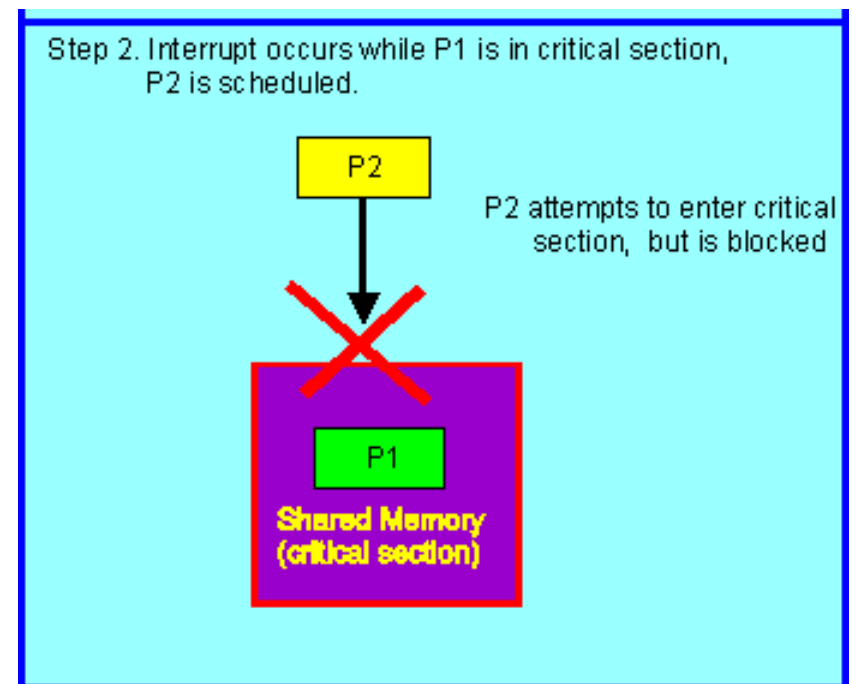
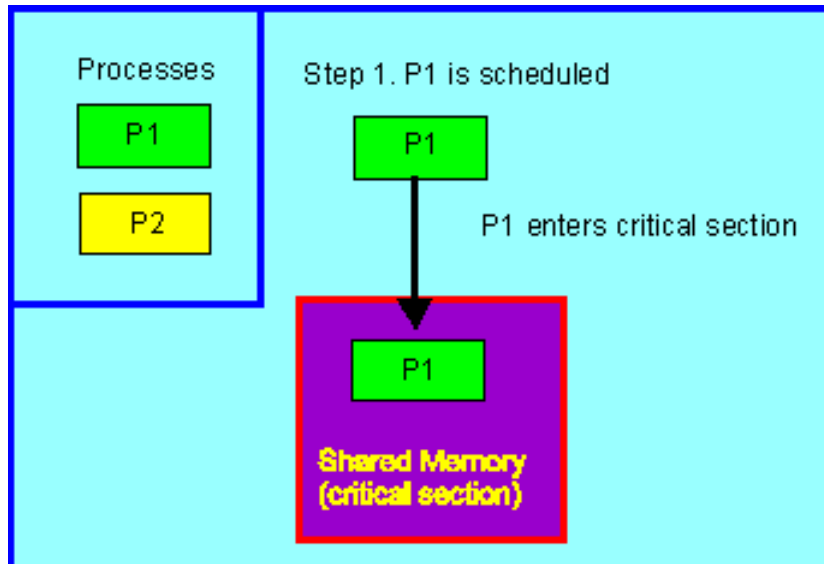
- Es necesario establecer un **canal** (lógico o físico) para transmitir información entre procesos.
- También el lenguaje debe proveer un protocolo adecuado.
- Para que la comunicación sea efectiva los procesos deben “saber” cuándo tienen mensajes para leer y cuando deben transmitir mensajes.

# Conceptos básicos de concurrencia

## Sincronización entre procesos

La **sincronización** es la posesión de información acerca de otro proceso para coordinar actividades. Los procesos se sincronizan:

- Por *exclusión mutua*.
- Por *condición*.



Ejemplo pasajes de micro.

# Conceptos básicos de concurrencia

## Sincronización entre procesos

**Ejemplo:** 3 puertas para entrar a un salón con lugar para  $N$  personas.

**Total = 0;**

**Puerta - 1**

```
{ while (total < N)
    esperar llegada
    total = total + 1;
}
```

**Puerta - 2**

```
{ while (total < N)
    esperar llegada
    total = total + 1;
}
```

**Puerta - 3**

```
{ while (total < N)
    esperar llegada
    total = total + 1;
}
```

*¿Cuántas personas podrían entrar a la sala ?*

# Conceptos básicos de concurrencia

## Sincronización entre procesos

- ♦ En la mayoría de los sistemas el tiempo absoluto no es importante.
- ♦ Con frecuencia los sistemas son actualizados con componentes más rápidas. La corrección del programa concurrente no debe depender del tiempo absoluto.
- ♦ El tiempo se ignora, sólo las secuencias son importantes

load	add	mult	store
load	add	mult	store

- ♦ Puede haber distintos ordenes (*interleavings*) en que se ejecutan las instrucciones de los diferentes procesos; los programas deben ser correctos para todos ellos.

# Conceptos básicos de concurrencia

## Sincronización entre procesos

- **Estado** de un programa concurrente.
- Una **acción atómica** es una acción que hace una transformación de estado indivisible (estados intermedios invisibles para otros procesos).
- Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas.
- Ejecución de un programa concurrente → **intercalado** (*interleaving*) de las acciones atómicas ejecutadas por procesos individuales.
- **Historia** de un programa concurrente (*trace*): ejecución de un programa concurrente con un *interleaving* particular.
- En general el número de posibles historias de un programa concurrente es enorme; pero no todas son válidas.

# Conceptos básicos de concurrencia

## Sincronización entre procesos

- Algunas historias son válidas y otras no.

**process 1**

```
{ while (true)
  p1.1: read(x);
  p1.2: buffer = x;
}
```

**process 2**

```
{ while (true)
  p2.1: y = buffer;
  p2.2: print(y);
}
```

**Posibles historias:**

p11, p12, p21, p22, p11, p12, p21, p22, ...	<input checked="" type="checkbox"/>
p11, p12, p21, p11, p22, p12, p21, p22, ...	<input checked="" type="checkbox"/>
p11, p21, p12, p22, ....	<input type="checkbox"/>
p21, p11, p12, ....	<input type="checkbox"/>

- Se debe asegurar un orden temporal entre las acciones que ejecutan los procesos → las tareas se intercalan ⇒ deben fijarse restricciones.

*El objetivo de la sincronización es restringir las historias de un programa concurrente sólo a las permitidas.*



# Conceptos básicos de concurrencia

## Sincronización entre procesos

La **sincronización** es la posesión de información acerca de otro proceso para coordinar actividades. Los procesos se sincronizan:

- Por *exclusión mutua*.
- Por *condición*.

- **Sincronización por exclusión mutua**

- Asegurar que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo.
- Si el programa tiene **secciones críticas** que pueden compartir más de un proceso, la exclusión mutua evita que dos o más procesos puedan encontrarse en la misma sección crítica al mismo tiempo.

- **Sincronización por condición**

- Permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada.

*Ejemplo de los dos mecanismos de sincronización en un problema de utilización de un área de memoria compartida (buffer limitado con productores y consumidores).*

# Conceptos básicos de concurrencia

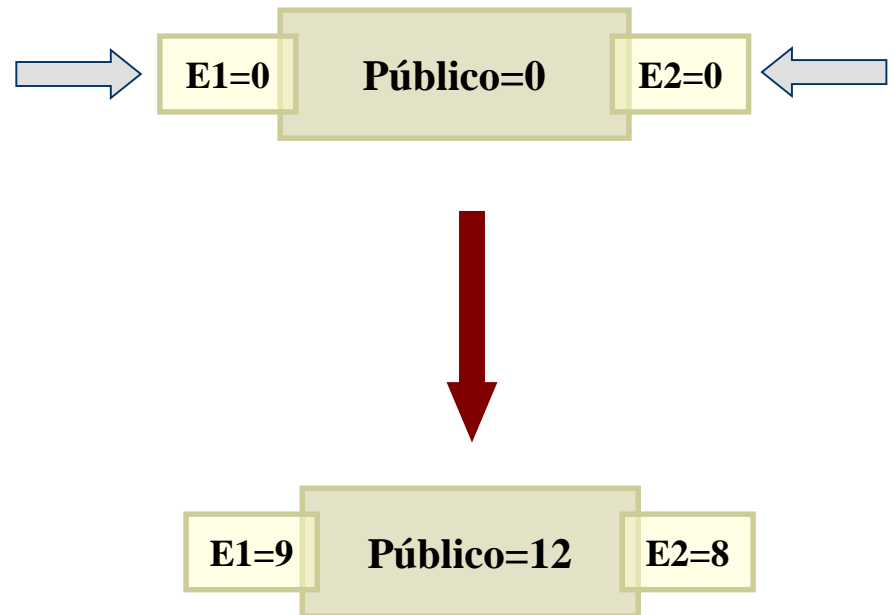
## Sincronización entre procesos

**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;
}
```



# Conceptos básicos de concurrencia

## Prioridad y granularidad

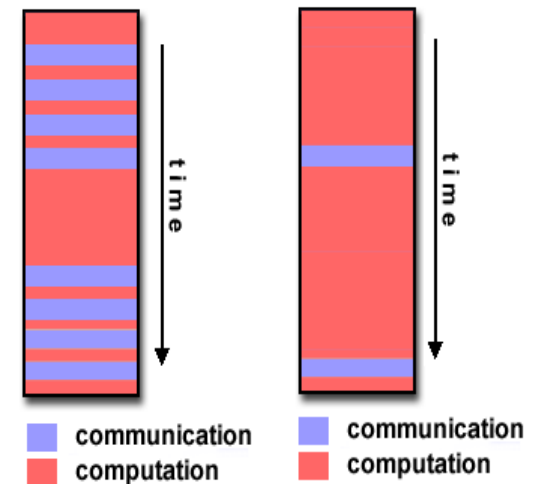
Un proceso que tiene mayor **prioridad** puede causar la suspensión (preemption) de otro proceso concurrente.

Análogamente puede tomar un recurso compartido, obligando a retirarse a otro proceso que lo tenga en un instante dado.

La **granularidad de una aplicación** está dada por la relación entre el cómputo y la comunicación.

Relación y adaptación a la arquitectura.

Grano fino y grano grueso.



# Conceptos básicos de concurrencia

## Manejo de los recursos

Uno de los temas principales de la programación concurrente es la **administración de recursos compartidos**:

- Esto incluye la asignación de recursos compartidos, métodos de acceso a los recursos, bloqueo y liberación de recursos, seguridad y consistencia.
- Una propiedad deseable en sistemas concurrentes es el equilibrio en el acceso a recursos compartidos por todos los procesos (***fairness***).
- Dos situaciones NO deseadas en los programas concurrentes son la ***inanición*** de un proceso (no logra acceder a los recursos compartidos) y el ***overloading*** de un proceso (la carga asignada excede su capacidad de procesamiento).
- Otro problema importante que se debe evitar es el ***deadlock***.

# Conceptos básicos de concurrencia

## Problema de deadlock



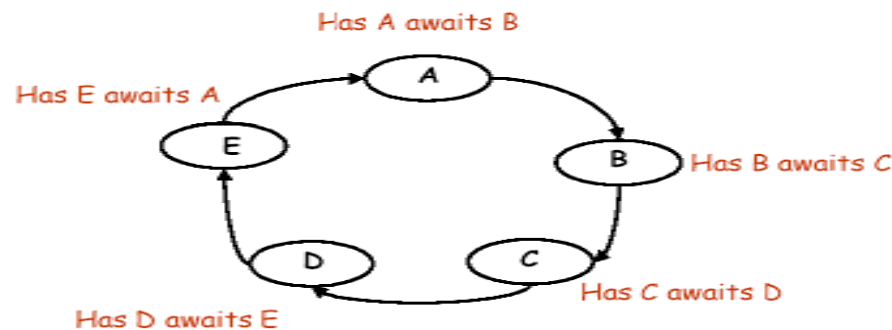
Dos (o más) procesos pueden entrar en *deadlock*, si por error de programación ambos se quedan esperando que el otro libere un recurso compartido. La ausencia de deadlock es una propiedad necesaria en los procesos concurrentes.

# Conceptos básicos de concurrencia

## Problema de deadlock

### 4 propiedades necesarias y suficientes para que exista deadlock son:

- **Recursos reusables serialmente**: los procesos comparten recursos que pueden usar con exclusión mutua.
- **Adquisición incremental**: los procesos mantienen los recursos que poseen mientras esperar adquirir recursos adicionales.
- **No-preemption**: una vez que son adquiridos por un proceso, los recursos no pueden quitarse de manera forzada sino que sólo son liberados voluntariamente.
- **Espera cíclica**: existe una cadena circular (ciclo) de procesos tal que cada uno tiene un recurso que su sucesor en el ciclo está esperando adquirir.



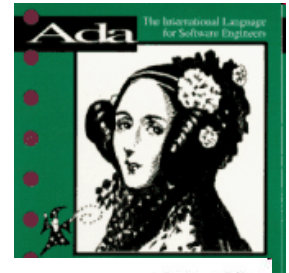
# Conceptos básicos de concurrencia

## Requerimientos para un lenguaje concurrente

Independientemente del mecanismo de comunicación / sincronización entre procesos, los **lenguajes de programación concurrente** deberán proveer primitivas adecuadas para la especificación e implementación de las mismas.

- **Requerimientos de un lenguaje de programación concurrente:**

- Indicar las tareas o procesos que pueden ejecutarse concurrentemente.
- Mecanismos de sincronización.
- Mecanismos de comunicación entre los procesos.



# Problemas asociados con la Programación Concurrente

- ◆ Los procesos no son independientes y comparten recursos. La necesidad de utilizar mecanismos de exclusión mutua y sincronización agrega complejidad a los programas.
- ◆ Los procesos iniciados dentro de un programa concurrente pueden NO estar “vivos”. Esta pérdida de la propiedad de *liveness* puede indicar deadlocks o una mala distribución de recursos.
- ◆ Hay un **no determinismo** implícito en el interleaving de procesos concurrentes. Esto significa que dos ejecuciones del mismo programa no necesariamente son idénticas  $\Rightarrow$  *dificultad para la interpretación y debug*.
- ◆ Posible reducción de performance por **overhead** de context switch, comunicación, sincronización, ...
- ◆ Mayor tiempo de desarrollo y puesta a punto. Difícil paralelizar algoritmos secuenciales.
- ◆ Necesidad de adaptar el software concurrente al hardware paralelo para mejora real en el rendimiento.



# Resumen de conceptos básicos

- La Concurrencia es un concepto de software.
- La **Programación Paralela** se asocia con la ejecución concurrente en múltiples unidades de procesamiento que pueden tener memoria compartida o no, y con un objetivo de incrementar la performance (reducir el tiempo de ejecución).
- En **Programación Concurrente** la organización de procesos y procesadores constituyen la arquitectura del sistema concurrente.

***Especificar la concurrencia es esencialmente especificar los procesos concurrentes, su comunicación y sincronización.***

# Clases de instrucciones

## Programación secuencial y concurrente

Un programa concurrente esta formado por un conjunto de programas secuenciales.

- La programación secuencial estructurada puede expresarse con 3 clases de instrucciones básicas: **asignación**, **alternativa** (decisión) e **iteración** (repetición con condición).
- Se requiere una clase de instrucción para representar 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))`

# Clases de instrucciones

## Programación secuencial y concurrente

### ASIGNACION

- Asignación simple:  $\mathbf{x = e}$
- Sentencia de asignación compuesta:  $\mathbf{x = x + 1; y = y - 1; z = x + y}$   
 $\mathbf{a[3] = 6; aa[2,5] = a[4]}$
- Llamado a funciones:  $\mathbf{x = f(y) + g(6) - 7}$
- swap:  $\mathbf{v1 := v2}$
- **skip**: termina inmediatamente y no tiene efecto sobre ninguna variable de programa.

# Clases de instrucciones

## Programación secuencial y concurrente

### ALTERNATIVA

- Sentencias de alternativa simple:  
    **if B  $\rightarrow$  S**  
    B expresión booleana. S instrucción simple o compuesta ( $\{ \}$ ).  
    **B “guarda” a S** pues S no se ejecuta si B no es verdadera.
- Sentencias de alternativa múltiple:  
    **if B1  $\rightarrow$  S1**  
     **$\square$  B2  $\rightarrow$  S2**  
    .....  
     **$\square$  Bn  $\rightarrow$  Sn**  
    **fi**  
    Las guardas se evalúan en algún orden arbitrario.  
    Elección no determinística.  
    Si ninguna guarda es verdadera el *if* no tiene efecto.
- Otra opción:  
    **if (cond) S;**  
    **if (cond) S1 else S2;**

# Clases de instrucciones

## Programación secuencial y concurrente

### Ejemplos de *Sentencia Alternativa Múltiple*

Ejemplo 1:

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

¿Puede terminar sin tener efecto?

Ejemplo 2:

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

¿Que sucede si  $p = 2$ ?

Ejemplo 3:

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

¿Que sucede con los siguiente valores de  $p = 1, 2, 3, 4, 5, 6, 7$ ?

# Clases de instrucciones

## Programación secuencial y concurrente

### ITERACIÓN

- Sentencias de alternativa ITERATIVA múltiple:

do  $B1 \rightarrow S1$

□  $B2 \rightarrow S2$

....

□  $Bn \rightarrow Sn$

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.

- For-all: forma general de repetición e iteración

**fa** cuantificadores  $\rightarrow$  Secuencia de Instrucciones **af**

Cuantificador  $\equiv$  **variable** := exp\_inicial **to** exp\_final **st** **B**

El cuerpo del *fa* se ejecuta 1 vez por cada combinación de valores de las variables de iteración. Si hay cláusula *such-that* (*st*), la variable de iteración toma sólo los valores para los que *B* es true.

Ejemplo: **fa**  $i := 1$  **to**  $n$ ,  $j := i+1$  **to**  $n$  **st**  $a[i] > a[j] \rightarrow a[i] := a[j]$  **af**

- Otra opción:

**while** (cond) **S**;

**for** [ $i = 1$  **to**  $n$ ,  $j = 1$  **to**  $n$  **st** ( $j \bmod 2 = 0$ )] **S**;

# Clases de instrucciones

## Programación secuencial y concurrente

### Ejemplos de *Sentencia Alternativa Iterativa Múltiple*

Ejemplo 1:

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

¿Cuándo termina?

Ejemplo 2:

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

¿Cuándo termina?

Ejemplo 3:

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

¿Cuándo termina?

¿Que sucede con  $p = 0, 3, 6, 9$ ?

Ejemplo 4:

```
do p == 1 → p = p * 2
  □ p == 2 → p = p + 3
  □ p == 4 → p = p / 2
od
```

¿Cuándo termina?

# Clases de instrucciones

## Programación secuencial y concurrente

### Ejemplos de *For-All*

$$\text{fa } i := 1 \text{ to } n \rightarrow a[i] = 0 \text{ af}$$

Inicialización de un vector

$$\text{fa } i := 1 \text{ to } n, j := i+1 \text{ to } n \rightarrow m[i,j] := m[j,i] \text{ af}$$

Trasposición de una matriz

$$\text{fa } i := 1 \text{ to } n, j := i+1 \text{ to } n \text{ st } a[i] > a[j] \rightarrow a[i] := a[j] \text{ af}$$

Ordenación de menor a mayor de un vector

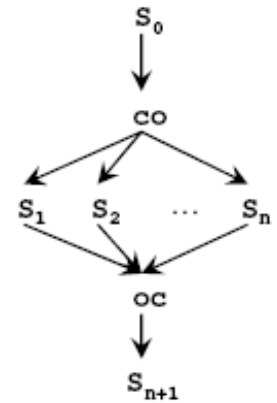


# Clases de instrucciones

## Programación secuencial y concurrente

### CONCURRENCIA

- Sentencia **co**:  
**co S1 // .... // Sn oc** → Ejecuta las  $S_i$  tareas concurrentemente.  
**co [i=1 to n] { a[i]=0; b[i]=0 } oc** → Crea  $n$  tareas concurrentes.  
Cuantificadores.  
La ejecución del **co** termina cuando todas las tareas terminaron.
- **Process**: otra forma de representar concurrencia  
**process A {sentencias}** → proceso único independiente.  
**process B [i=1 to n] {sentencias}** →  $n$  procesos independientes.  
Cuantificadores.
- **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.



# Clases de instrucciones


## Programación secuencial y concurrente

Ejemplo: ¿qué imprime en cada caso? ¿son equivalentes?

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

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

*No determinismo....*



---

# Paradigmas de resolución de programas concurrentes

---

# 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*).

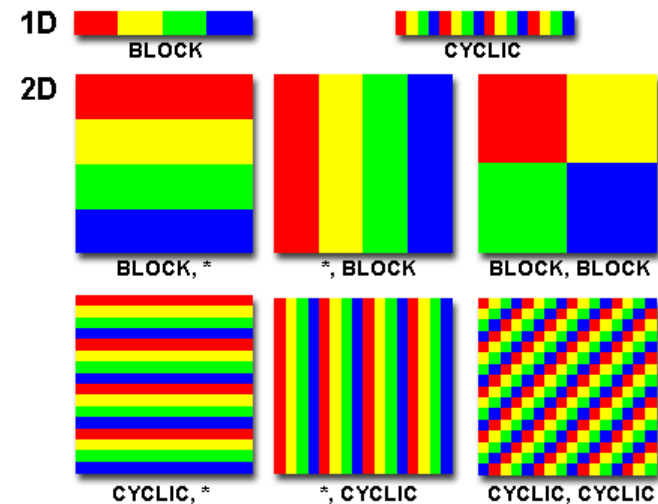
# Paradigmas de resolución de programas concurrentes

En el *paralelismo iterativo* un programa consta de un conjunto de procesos (posiblemente idénticos) cada uno de los cuales tiene 1 o más loops. Cada proceso es un programa iterativo.

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

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

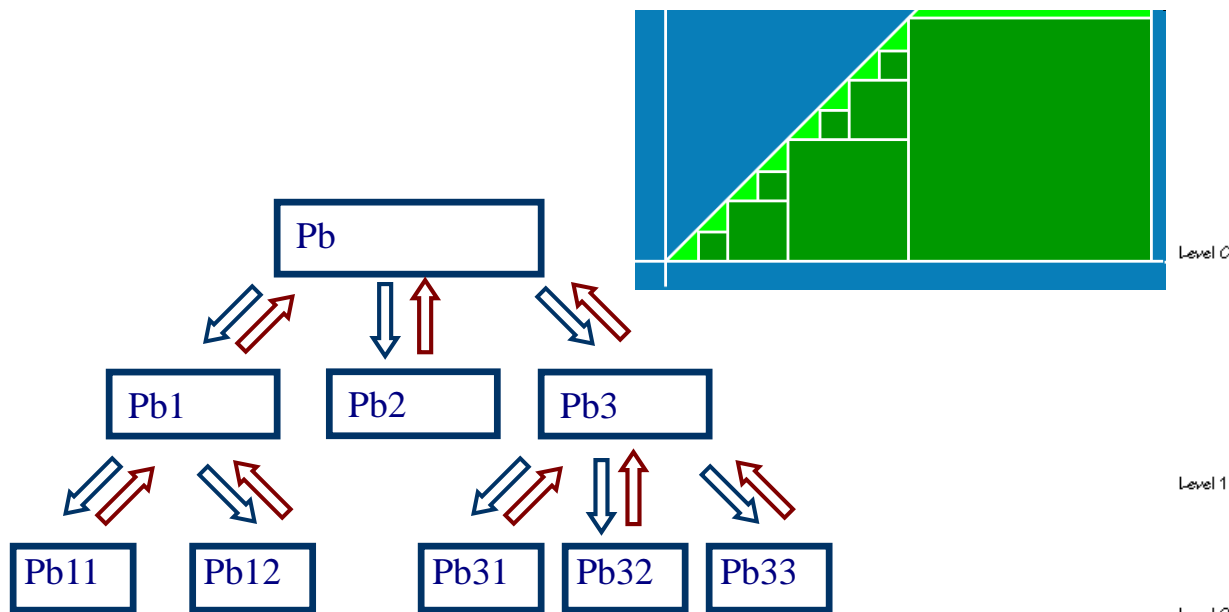
$$\begin{bmatrix} a_{11} & a_{12} & L & a_{1n} \\ a_{21} & a_{22} & L & a_{2n} \\ M & M & M & M \\ a_{n1} & a_{n2} & L & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & L & b_{1n} \\ b_{21} & b_{22} & L & b_{2n} \\ M & M & M & M \\ b_{n1} & b_{n2} & L & b_{nn} \end{bmatrix} = \begin{bmatrix}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



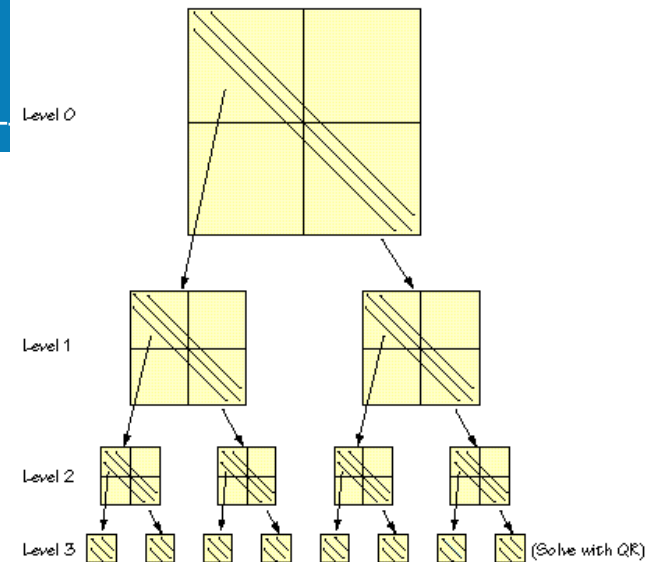
# Paradigmas de resolución de programas concurrentes

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.



Divide and Conquer

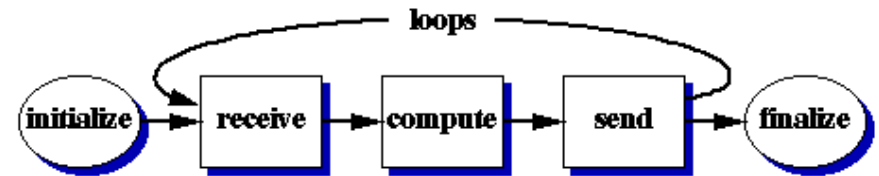


# Paradigmas de resolución de programas concurrentes

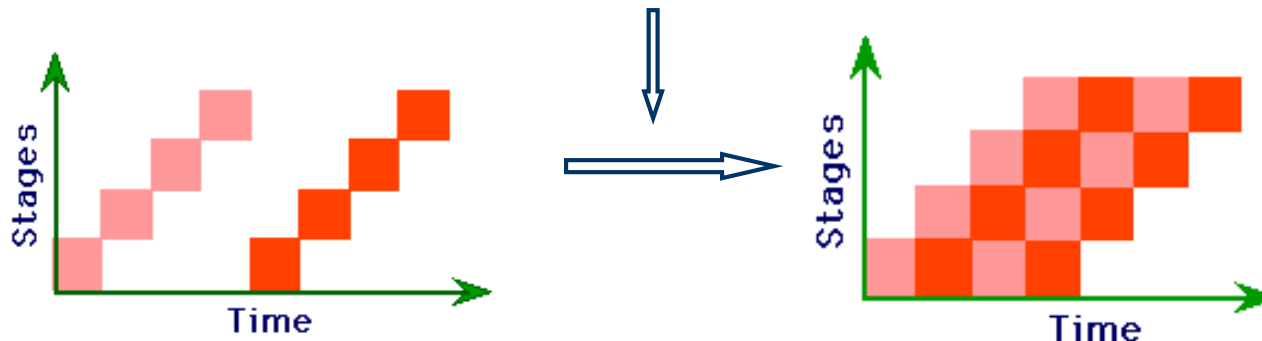
Los esquemas *productor-consumidor* muestran procesos que se comunican.

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, secuencia de filtros sobre imágenes, ¿ordenación de un vector?.



Pipe



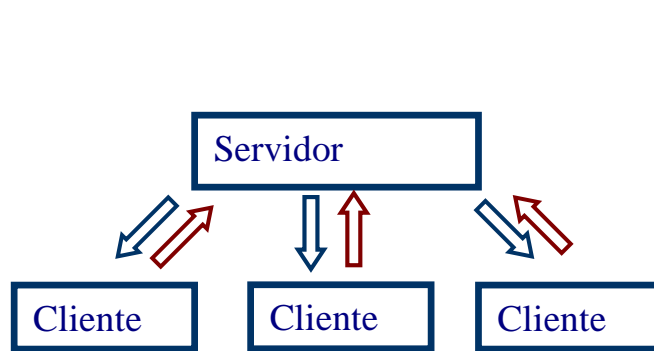
# Paradigmas de resolución de programas concurrentes

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

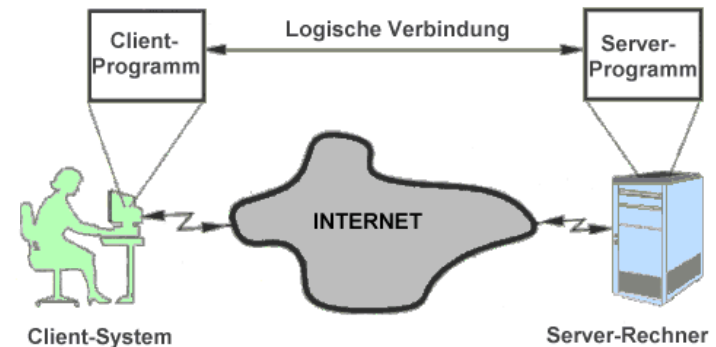
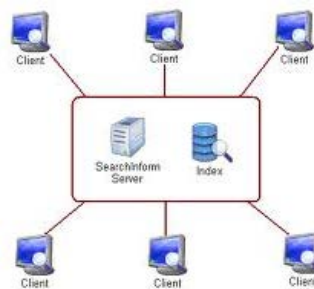
Los servidores son procesos que esperan pedidos de servicios de múltiples clientes. Naturalmente unos y otros pueden ejecutarse en procesadores diferentes. Comunicación bidireccional. Atención de a un cliente a la vez, o a varios con multithreading.

Mecanismos de invocación variados (rendezvous, RPC, monitores).

El soporte distribuido puede ser simple (LAN) o extendido a la WEB.



Cliente/Servidor



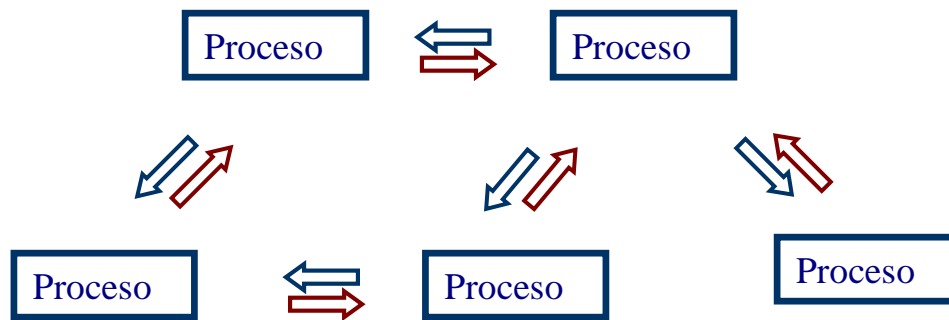


# Paradigmas de resolución de programas concurrentes

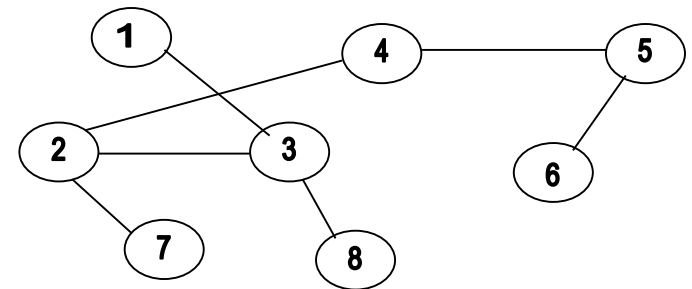
En los esquemas de *pares que interactúan* los procesos (que forman parte de un programa distribuido) resuelven partes del problema (normalmente mediante código idéntico) e intercambian mensajes para avanzar en la tarea y completar el objetivo.

El esquema permite mayor grado de asincronismo que cliente-servidor.

Posibles configuraciones: grilla, pipe circular, uno a uno, arbitraria.



Pares que interactúan



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

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

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

.....

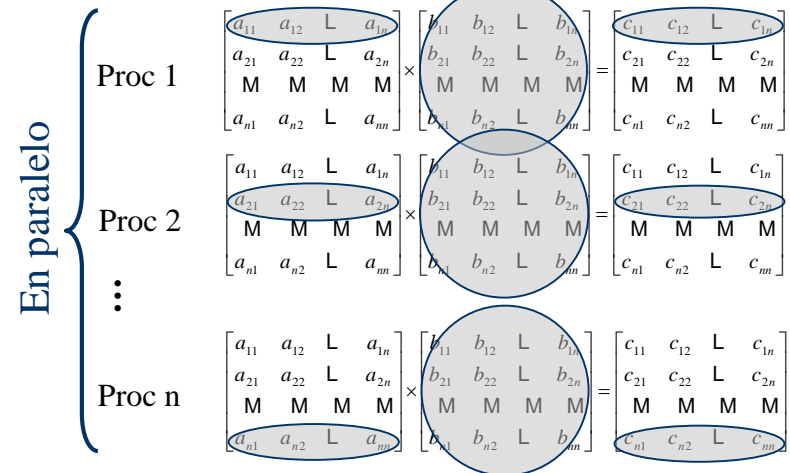
$$\begin{bmatrix} a_{11} & a_{12} & L & a_{1n} \\ a_{21} & a_{22} & L & a_{2n} \\ M & M & M & M \\ a_{n1} & a_{n2} & L & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & L & b_{1n} \\ b_{21} & b_{22} & L & b_{2n} \\ M & M & M & M \\ b_{n1} & b_{n2} & L & b_{nn} \end{bmatrix} = \begin{bmatrix}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

- 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 *embarrassingly parallel* (muchas operaciones en paralelas).
- Diferentes acciones paralelas posibles.

# Ejemplo de paralelismo iterativo: multiplicación de matrices

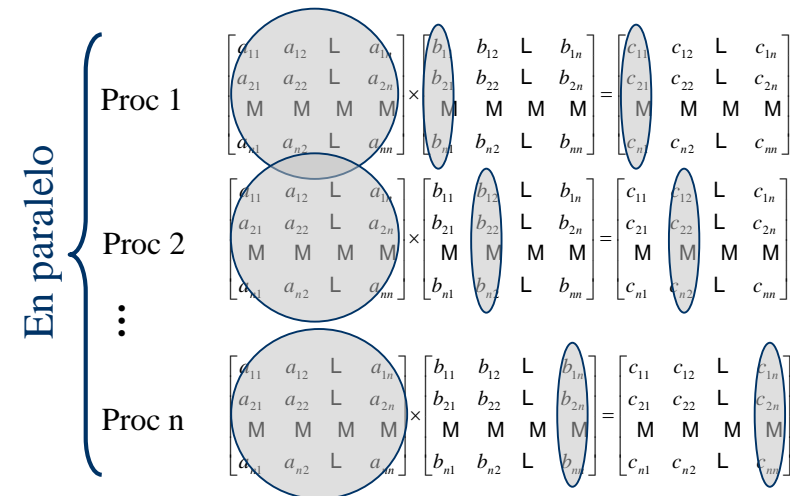
## Solución paralela por fila:

```
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]);
    }
  }
```



## 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]);
    }
  }
```



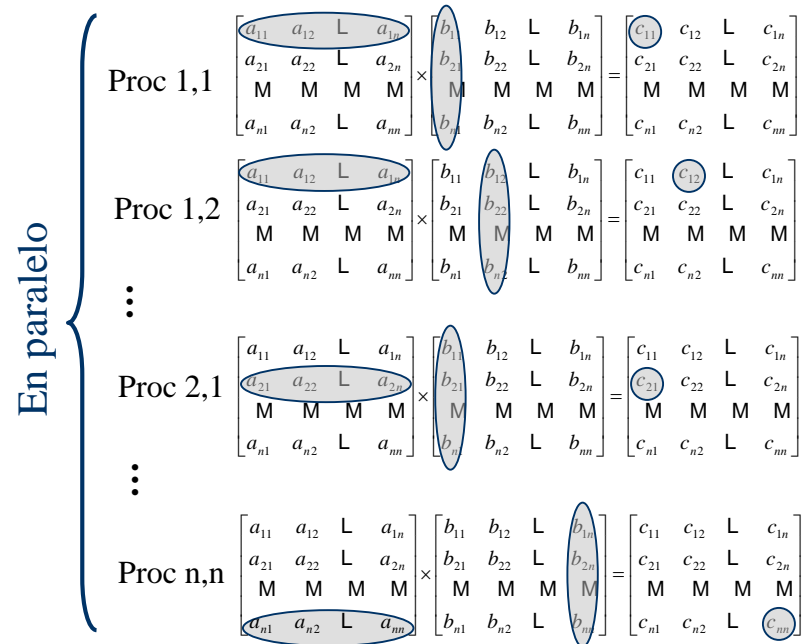
# Ejemplo de paralelismo iterativo: multiplicación de matrices

## Solución paralela por celda (opción 1):

```
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):

```
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]);
  }
}
```



# Ejemplo de paralelismo iterativo: multiplicación de matrices. Uso de Process.

## 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?

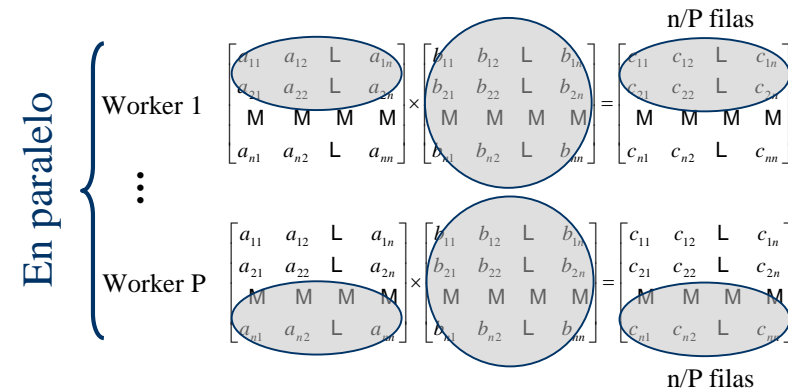
- Se puede dividir la matriz resultado en *strips* (subconjuntos de filas o columnas) y usar un proceso por strip.
- El tamaño del strip óptimo es un problema interesante para balancear costo de procesamiento con costo de comunicaciones.

# Ejemplo de paralelismo iterativo: multiplicación de matrices. Uso de Process.

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

```

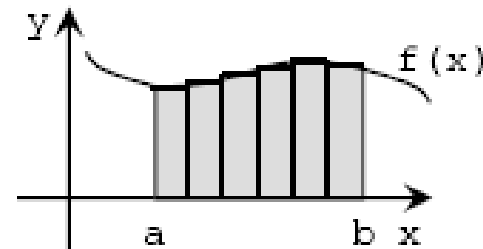
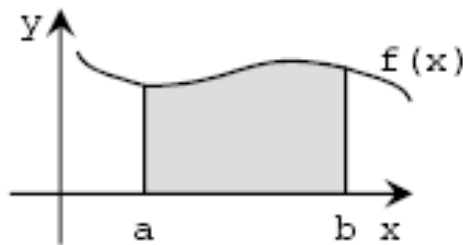
process worker [ w = 1 to P]
{ int primera = (w-1)*(n/P) + 1;
  int ultima = primera + (n/P) - 1;
  for [i = primera to ultima]
    { for [j = 1 to n]
      { c[i,j] = 0;
        for [k = 1 to n]
          c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
        }
      }
    }
}
    
```



- Ejercicio:** a) Si  $P=8$  y  $n=120$ . ¿Cuántas asignaciones, sumas y productos hace cada procesador?.
- b) Si  $P_1=...=P_7$  y los tiempos de asignación son 1, de suma 2 y de producto 3; y si  $P_8$  es 2 veces más lento. ¿Cuánto tarda el proceso total?. ¿Cuál es el speedup?. ¿Qué puede hacerse para mejorar el speedup?.

# Ejemplo de 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  $b$

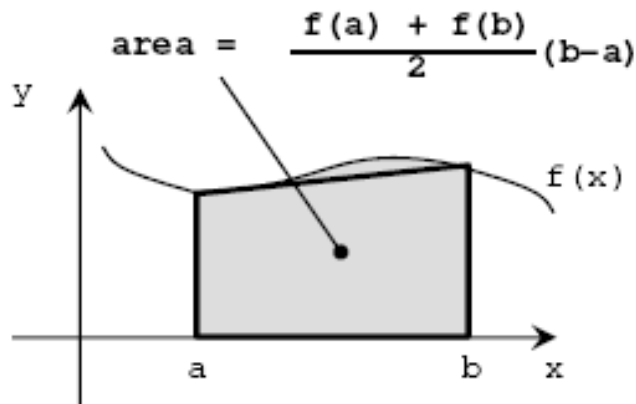


Solución secuencial iterativa (usando el método trapezoidal):

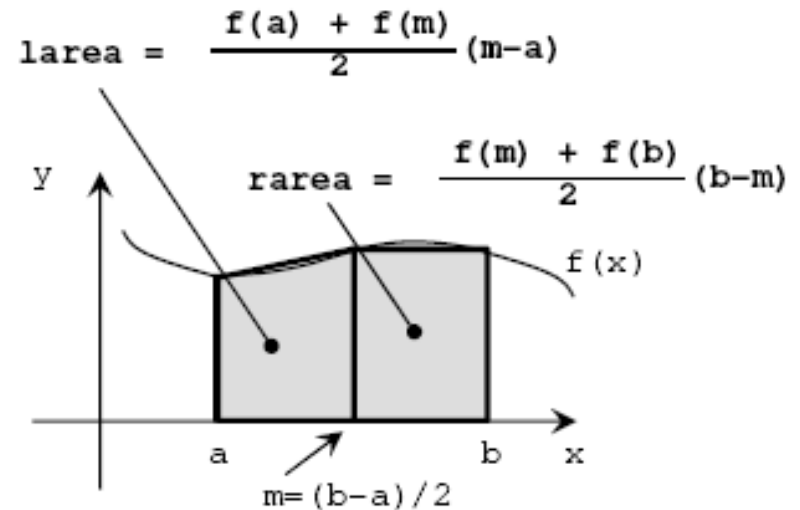
```
double fl = f(a), fr, area = 0.0;
double dx = (b-a)/ni;
for [x = (a+dx) to b by dx]
{
    fr = f(x);
    area = area + (fl+fr) * dx / 2;
    fl = fr;
}
```

# Ejemplo de 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  $e$ .



# Ejemplo de 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:  $\text{area} = \text{quad}(a, b, f(a), f(b), (f(a) + f(b)) * (b-a) / 2)$



---

# Clasificación de arquitecturas paralelas

---

# Clasificación de arquitecturas paralelas

Hay diferentes enfoques para clasificar las arquitecturas paralelas:

- *Por la organización del espacio de direcciones.*
- *Por la granularidad.*
- **Por el mecanismo de control.**
- Por la red de interconexión.

# Clasificación de arquitecturas paralelas

## Por el 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*.

Clasifica las arquitecturas en 4 clases:

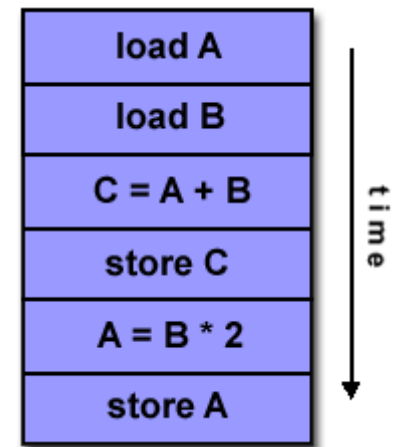
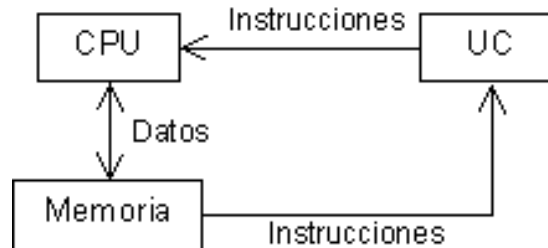
- **SISD** (Single Instruction Single Data).
- **SIMD** (Single Instruction Multiple Data).
- **MISD** (Multiple Instruction Single Data).
- **MIMD** (Multiple Instruction Multiple Data).

# Clasificación de arquitecturas paralelas

## Por el mecanismo de control

### SISD: Single Instruction Single Data

- Instrucciones ejecutadas en secuencia, una por ciclo de instrucción.
- La memoria afectada es usada sólo por ésta instrucción.
- Usada por la mayoría de los uní procesadores.
- La CPU ejecuta instrucciones (decodificadas por la UC) sobre los datos. La memoria recibe y almacena datos en las escrituras, y brinda datos en las lecturas.
- Ejecución *determinística*.

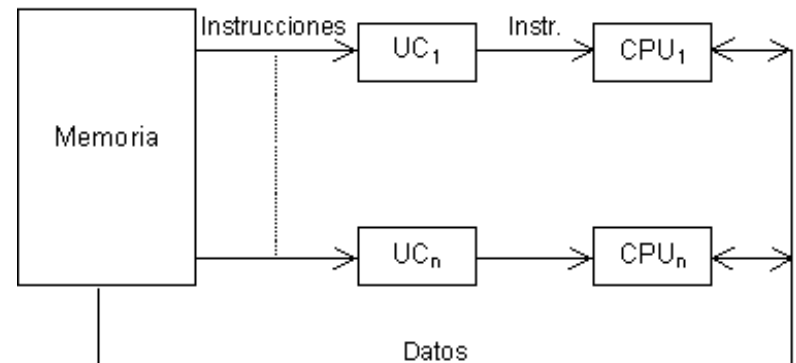


# Clasificación de arquitecturas paralelas

## Por el 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:
  - Múltiples filtros de frecuencia operando sobre una única señal.
  - Múltiples algoritmos de criptografía intentando crackear un único mensaje codificado.



# Clasificación de arquitecturas paralelas

## Por el mecanismo de control

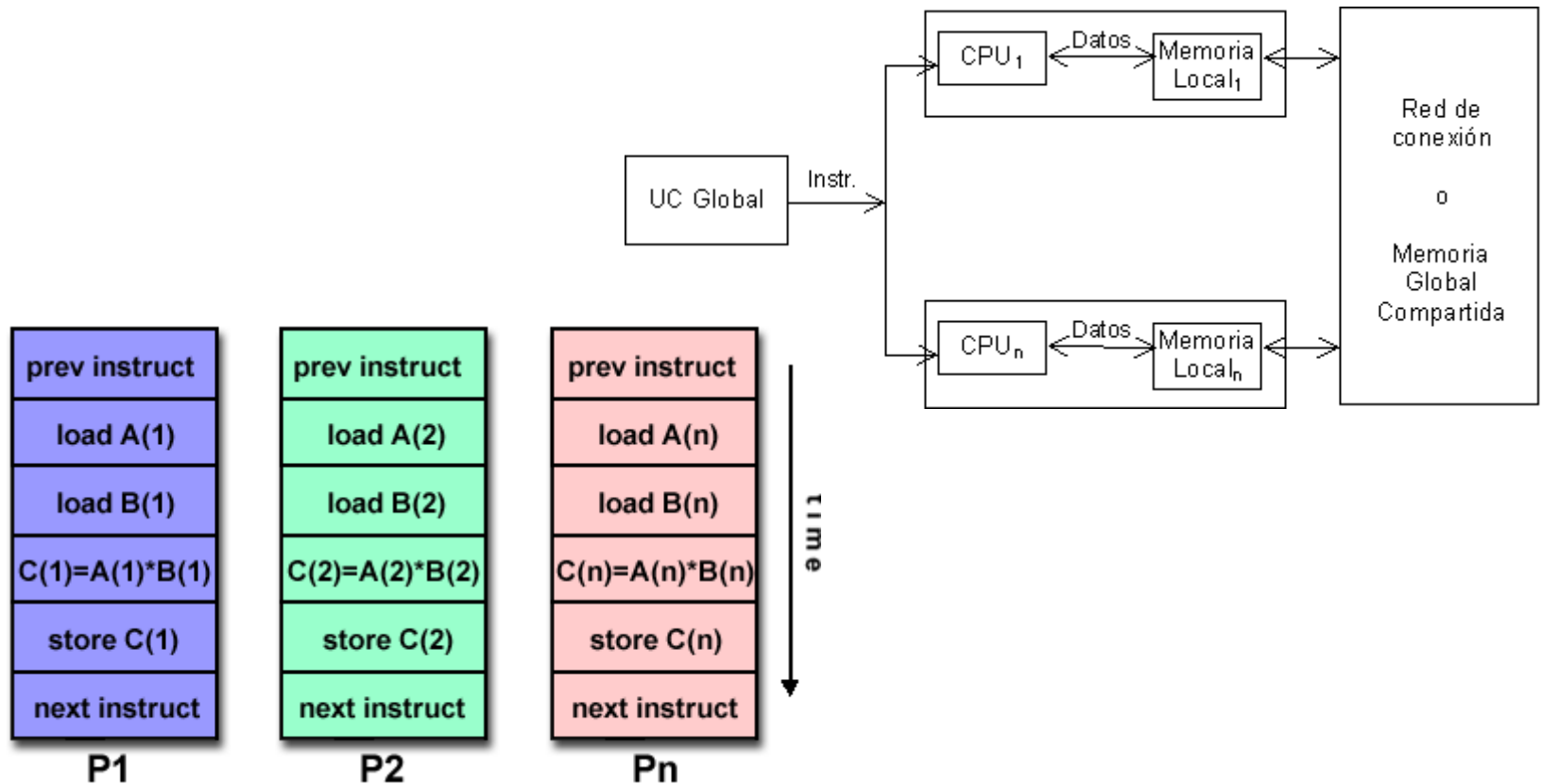
### **SIMD:** Single Instruction Multiple Data

- Conjunto de procesadores idénticos, con sus memorias, que ejecutan la misma instrucción sobre distintos datos.
- Los procesadores en general son muy simples.
- El *host* hace *broadcast* de la instrucción. Ejecución sincrónica y determinística.
- Pueden deshabilitarse y habilitarse selectivamente procesadores para que ejecuten o no instrucciones.
- Adecuados para aplicaciones con alto grado de regularidad, (por ejemplo procesamiento de imágenes).

# Clasificación de arquitecturas paralelas

## Por el mecanismo de control

**Ejemplos de máquina SIMD:** Array Processors. CM-2, Maspar MP-1 y 2, Illiac IV.



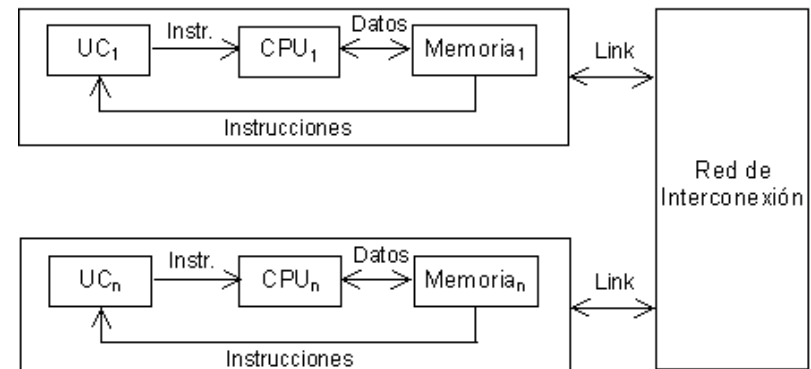
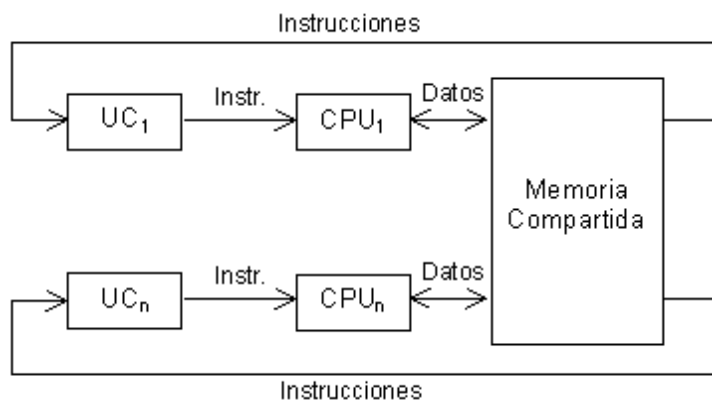


# Clasificación de arquitecturas paralelas

## Por el mecanismo de control

### MIMD: Multiple Instruction Multiple Data

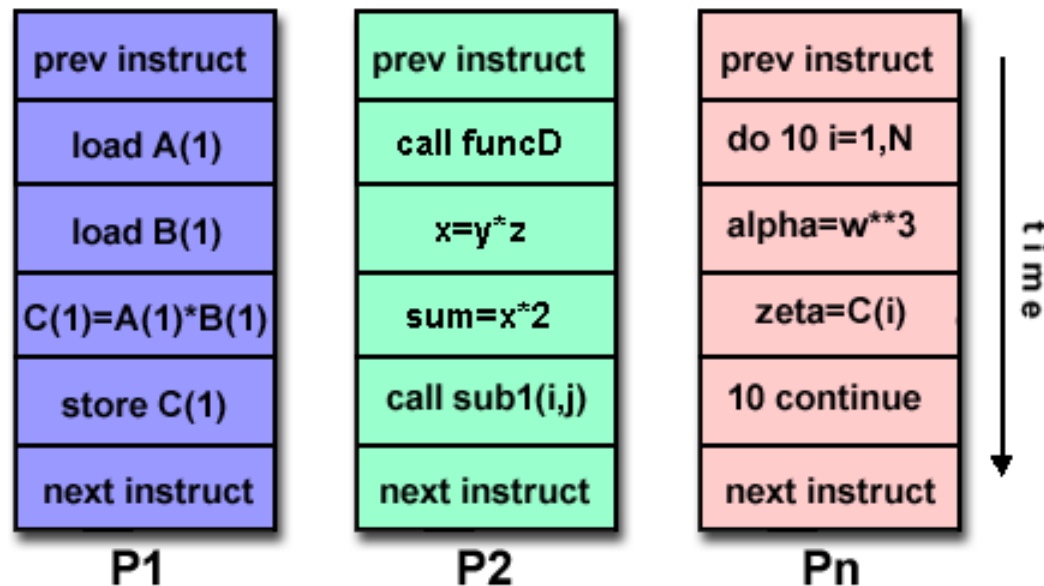
- Cada procesador tiene su propio flujo de instrucciones y de datos  $\Rightarrow$  cada uno ejecuta su propio programa.
- Pueden ser con memoria compartida o distribuida.
- Sub-clasificación de MIMD:
  - **MPMD** (multiple program multiple data): cada procesador ejecuta su propio programa (ejemplo con PVM).
  - **SPMD** (single program multiple data): hay un único programa fuente y cada procesador ejecuta su copia independientemente (ejemplo con MPI).



# Clasificación de arquitecturas paralelas

## Por el mecanismo de control

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



# Clasificación de arquitecturas paralelas

Hay diferentes enfoques para clasificar las arquitecturas paralelas:

- *Por la organización del espacio de direcciones.*
- *Por la granularidad.*
- *Por el mecanismo de control.*
- **Por la red de interconexión.**

# Clasificación de arquitecturas paralelas

## Por la red de interconexión

Tanto en memoria compartida como en pasaje de mensajes las máquinas pueden construirse conectando procesadores y memorias usando diversas redes de interconexión:

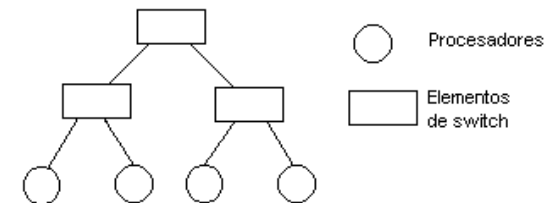
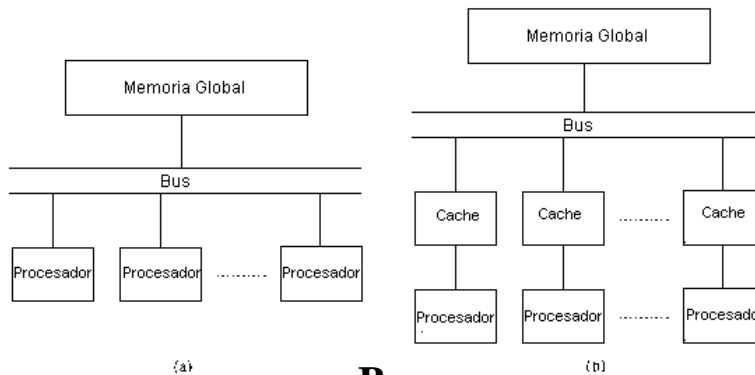
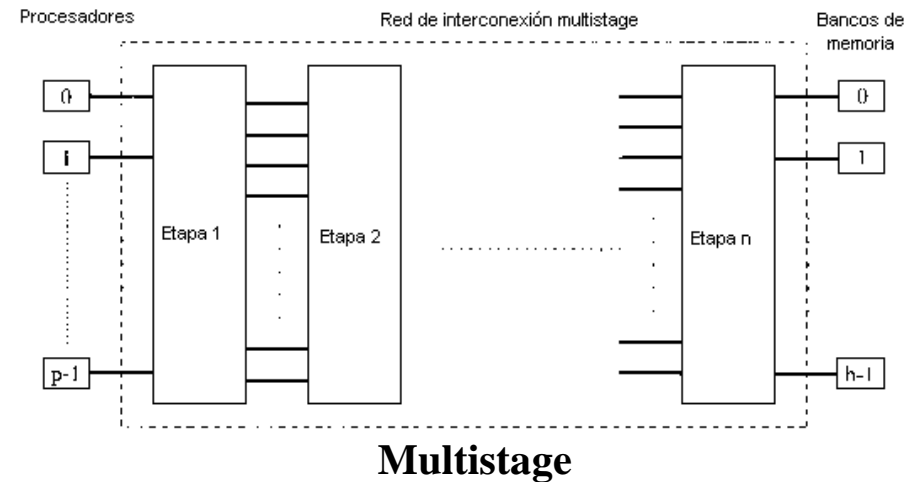
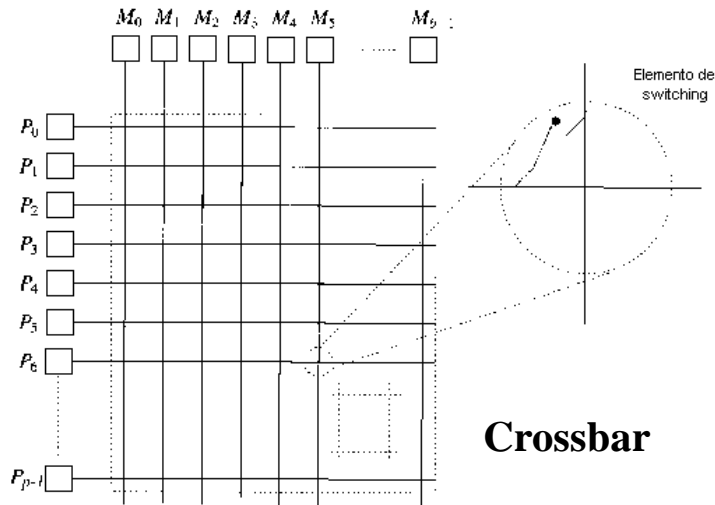
- Las **redes estáticas** constan de *links* punto a punto. Típicamente se usan para máquinas de pasaje de mensajes.
- Las **redes dinámicas** están construidas usando switches y enlaces de comunicación. Normalmente para máquinas de memoria compartida.

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, costo, etc.).

# Clasificación de arquitecturas paralelas

## Por la red de interconexión

### Redes de interconexión dinámicas

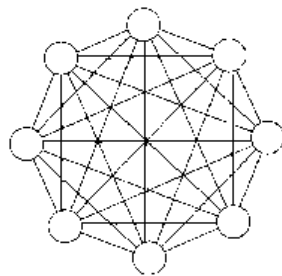


(b)  
**Árbol dinámico**

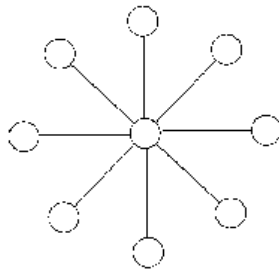
# Clasificación de arquitecturas paralelas

## Por la red de interconexión

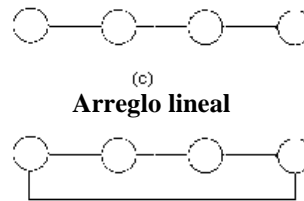
### Redes de interconexión estáticas



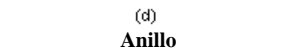
(a)  
**Completamente conectada**



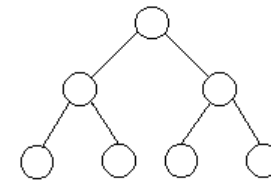
(b)  
**Estrella**



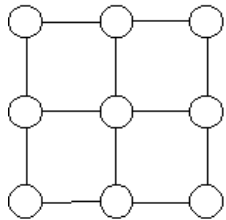
(c)  
**Arreglo lineal**



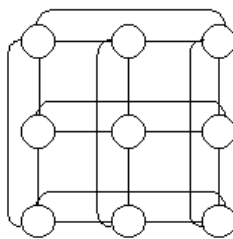
(d)  
**Anillo**



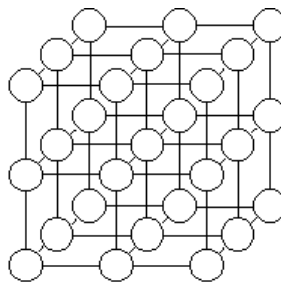
(a)  
**Árbol estático**



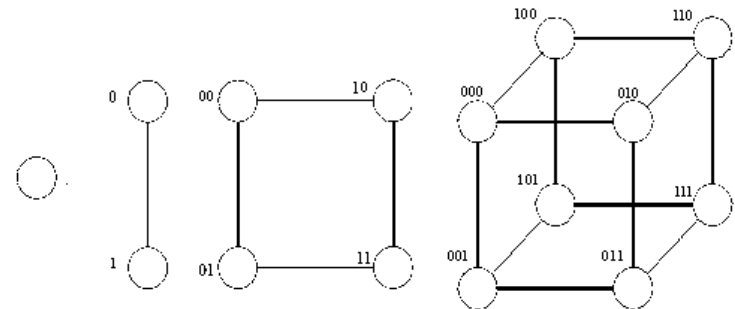
(a)  
**Mesh**



(b)  
**Toro**



(c)  
**Mesh 3D**



Hipercubo 0-D

Hipercubo 1-D

Hipercubo 2-D

Hipercubo 3-D

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