

# Sistemas Paralelos

## Sistemas Distribuidos y Paralelos

### Clase 2



Facultad de Informática  
UNLP



---

# Paralelismo Implícito

---

# Paralelismo Implícito en los Procesadores

- La idea básica de la arquitectura de un procesadores → CPU + Memoria + Bus de Comunicaciones (Cuellos de botella).
- Para mejorar la performance, nos podemos enfocar en cada uno de estos componentes “macro”.
- En los últimos tiempos el crecimiento de la velocidad del reloj de los procesadores y el nivel de integración (multiplicidad de unidades de procesamiento, unidades de memoria, buses) alcanzado son puntos a favor.
- El tema es como generar unidades funcionales que trabajen al mismo tiempo en el procesador y cómo controlar la sincronización y comunicación de las mismas → diferentes arquitecturas paralelas.

# Pipelining y la ejecución en las máquinas Superescalares

- En el *pipelining* de un procesador, se superponen temporalmente fases o etapas de la ejecución de una instrucción, para incrementar performance.
- En forma abstracta podemos pensar en ***FETCH – DECODE y EXECUTE como las tres etapas principales.***
- El modelo de referencia es la “línea de montaje”, aunque en la realidad las etapas pueden ser muchas más y la cola del pipe superar largamente las 3 etapas.

# Pipelining y la ejecución en las máquinas Superescalares

- El método de pipelining tiene limitaciones.
- La velocidad está limitada por la etapa más lenta → aumentar el número de etapas (más de 20).
- Sin embargo estadísticamente cada 5/6 instrucciones hay un salto condicional (cambia el contador de programa y se pierde lo adelantado en el pipe)
- Es importante predecir los saltos y “reiniciar” el pipe en la/s posibles instrucciones de destino, lo cual incrementa la complejidad → múltiples pipes simultáneos.

# Un ejemplo de ejecución superescalar con pipe

```

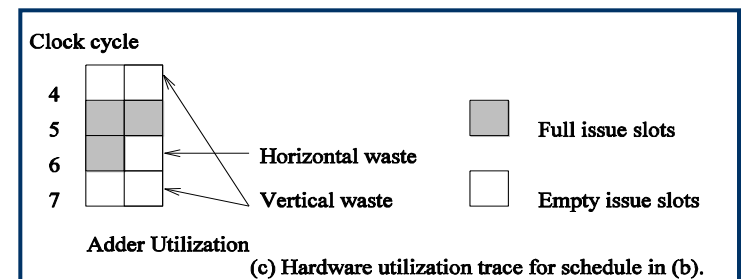
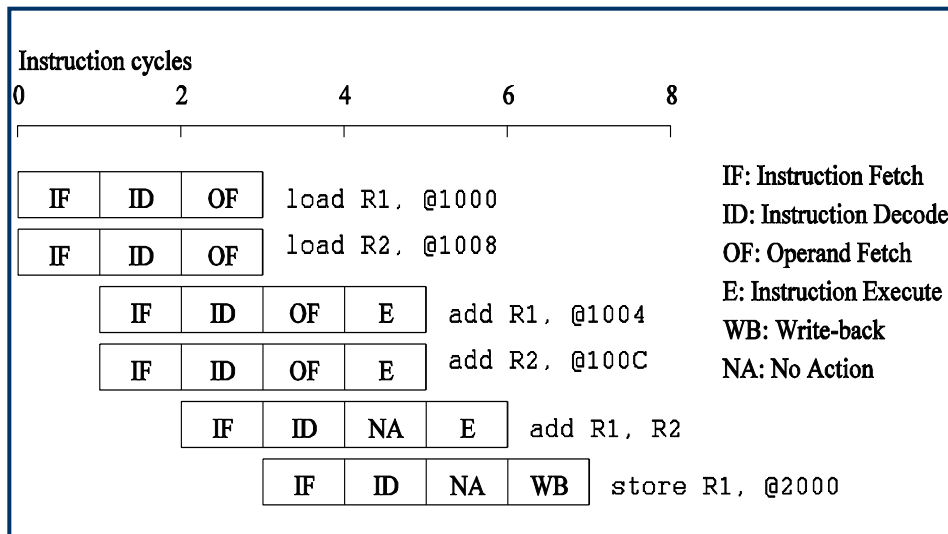
1. load R1, @1000
2. load R2, @1008
3. add R1, @1004
4. add R2, @100C
5. add R1, R2
6. store R1, @2000
    
```

```

1. load R1, @1000
2. add R1, @1004
3. add R1, @1008
4. add R1, @100C
5. store R1, @2000
    
```

```

1. load R1, @1000
2. add R1, @1004
3. load R2, @1008
4. add R2, @100C
5. add R1, R2
6. store R1, @2000
    
```



# Un ejemplo de ejecución superescalar con pipe

- Se puede apreciar que partiendo de la misma semántica (resultado final), se puede combinar de diferente modo las instrucciones (¿compilador?) y tener diferente código y diferente tiempo de ejecución global.
- También se aprecia un número de recursos no utilizados (lo cual se incrementará con la profundidad del pipe). Es muy difícil que todas las etapas estén activas (máximo paralelismo).

# Ejecución Superescalar

Al realizar el scheduling de las instrucciones, existe un número de factores a tener en cuenta:

1. ***True Data Dependency***: el resultado de una operación es entrada de la siguiente instrucción.
2. ***Resource Dependency***: dos operaciones requieren el mismo recurso (ejemplo, unidad de punto flotante).
3. ***Branch Dependency***: la instrucción de destino después de un branch no puede ser establecida en forma segura a priori.

*El scheduler (básicamente hardware) analiza un gran número de instrucciones en la cola de instrucciones y toma el subconjunto de ellas a ejecutar concurrentemente, en base a estos factores. La complejidad de este hardware (que condiciona su eficiencia) es una limitación importante para los procesadores superescalares.*



# Ejecución superescalar:

## Mecanismos de selección de instrucciones

- En el modelo más simple, las instrucciones sólo son tratadas en el orden en que están en la cola. Esto significa que si la instrucción  $N$  tiene una dependencia de datos con la instrucción  $N-1$ , se tratarán a lo sumo  $N-1$  instrucciones en el ciclo (caso peor  $N=2$ ).
  - Esto se denomina selección *in-order* ( $<$  performance)
- Un modelo más agresivo permite tratar instrucciones fuera de orden (en el ejemplo anterior la  $N-1$  con la  $N + 1$  demorando la instrucción  $N$ ).
  - Esto se denomina selección *dynamic*.

# Ejecución superescalar: Consideraciones de Eficiencia

- No todas las unidades funcionales pueden estar activas al mismo tiempo. Si en un ciclo del reloj ninguna es utilizada, se lo denomina “*vertical waste*”.
- Cuando en un ciclo de reloj hay algunas unidades funcionales no utilizadas, se denomina “*horizontal waste*”.
- Las dependencias entre instrucciones y la imposibilidad de los compiladores optimizadores y del scheduler de extraer el máximo paralelismo establecen un límite a la performance de los procesadores superescalares.

# Procesadores VLIW (Very Long Instruction Word)

- El costo y complejidad del hardware del scheduler fijan una limitación a los procesadores superescalares.
- El enfoque de los procesadores VLIW es lograr un análisis (y optimización) a nivel de compilación para identificar y juntar las instrucciones que pueden ejecutarse concurrentemente.
- Estas instrucciones se empaquetan y despachan en conjunto, dando lugar a la noción de VLIW.
- Este concepto ha sido usado del 1984 (Multiflow trace machine) y está presente en los procesadores Intel IA64.

# Consideraciones sobre la ejecución en Procesadores VLIW

- El hardware de selección y servicio de instrucciones es más simple.
  - Los compiladores deben manejar un contexto mayor para determinar la selección de las instrucciones que trabajarán concurrentemente.
  - Los compiladores tendrán políticas estáticas conservadores, ya que NO dispondrán de información de ejecución (como un cache miss por ejemplo)
  - La predicción de bifurcaciones es más difícil.
- *La performance de un procesador VLIW depende del compilador.*



---

# Limitaciones en la Performance del Sistema de Memoria

---

# Limitaciones en la performance del Sistema de Memoria

- En muchas aplicaciones la limitación está en el sistema de memoria, no en la velocidad y potencia de cálculo del procesador.
- Los dos parámetros esenciales son la latencia y el ancho de banda del sistema de memoria.
- La latencia es el tiempo que va desde el “*memory request*” hasta que los datos están disponibles.
- El ancho de banda es la velocidad con la cual el sistema de memoria puede ponerlos en el procesador.

# Latencia de la memoria: un ejemplo

- Consideremos un procesador con frecuencia de reloj de 1 GHz (1 ns clock) conectado a una memoria DRAM con una latencia de 100 ns (sin caches).
  - Supongamos que el procesador tiene dos unidades multiplicación-suma y es capaz de ejecutar 4 instrucciones por ciclo de reloj de 1 ns.
- **El pico de rendimiento del procesador es 4 Gflops.**
- **Dado que la memoria tarda 100 ciclos de reloj en responder a un pedido, esto significa que 1 pedido a memoria significa 100 ciclos de espera para el procesador.**

# Latencia de la memoria: un ejemplo

- Si ahora sobre este procesador tenemos que multiplicar dos vectores de  $K$  elementos cada uno.
- Básicamente en cada producto se requiere dos accesos a memoria para buscar los operandos a multiplicar.
- **Un producto asociado con dos lecturas, pone el ritmo de procesamiento en 1 operación de producto-suma cada 200 ns.**
- **Pasamos de 4 Gflops pico a 10 Mflops reales**



# Mejora de la latencia efectiva, utilizando Caches.

- La memoria cache se caracteriza por tener menor capacidad y menor latencia que la memoria DRAM (además de mayor costo).
- La idea es disminuir la latencia del sistema de memoria, maximizando el número de datos que se acceden de caché.
- Un “cache hit” es cuando buscamos un dato que está en caché. El % de hits es importante porque incide en la latencia global del sistema.

# Ejemplo del impacto de uso de Caché

- Tenemos el mismo procesador anterior, la misma memoria anterior.
- Agregamos una memoria caché de 32 KB con latencia de 1 ns.
- Trataremos de multiplicar 2 matrices A y B de 32 x 32 (esto permitiría alojar en caché las matrices A, B y el resultado C).

# Ejemplo del impacto de uso de Caché

- El *fetch* de las dos matrices para llevarlas de la caché al procesador significa leer 2K palabras:

$$2000 \times 100 \text{ ns} = 200 \mu\text{s}$$

- Multiplicar las dos matrices de  $n \times n$  significa  $2n^3$  operaciones. En nuestro problema  $n=32$  significa 64K operaciones, que pueden realizarse en 16K ciclos (o *16  $\mu\text{s}$* ) a 4 operaciones por ciclo de 1 ns.
- Cada operación son dos lecturas de cache. Aproximadamente:

$$64000 \times 2 \text{ ns} = 128 \mu\text{s}$$

- El tiempo total de cómputo pasa a ser entonces:

$$200 \mu\text{s} + 16 \mu\text{s} + 128 \mu\text{s} + 100 \mu\text{s} \text{ (store resultado en memoria)} = 444 \mu\text{s}$$

- Esto corresponde a  $64\text{K}/444 \mu\text{s} = 144\text{Mflops}$

# Impacto del Ancho de Banda de la Memoria

- El Ancho de Banda de la Memoria está determinado por el ancho en bits del Bus de datos y la velocidad de procesamiento de los mismos.
- Puede mejorarse incrementando el tamaño de los bloques de memoria que se transfieren en un ciclo de reloj ( $>$  complejidad del hardware de control).
- El hardware consume  $L$  unidades de tiempo (Latencia) para obtener  $B$  unidades de datos ( $B$  es el tamaño del Bloque medido en bits, bytes o words).

# Impacto del Ancho de Banda de la Memoria

- Si consideramos el ejemplo típico de un producto entre dos vectores en memoria, cada operación es una multiplicación y suma que se realiza en un ciclo de procesador.
- Normalmente la lectura de cada palabra de datos de los vectores significa de 10 a 100 veces el tiempo del producto y suma, por lo que conviene manejar un número mayor de palabras consecutivas (en este ejemplo es simple).
- En general se buscará que haya *localidad espacial* y *localidad temporal*.

# Impacto del Ancho de Banda de la Memoria

- En el primer ejemplo (sin caché) si aumentamos el ancho del bus de 1 palabra a 4 palabras (de 32 a 128 bits) podemos reducir 4 veces los tiempos de transferencia de datos al procesador

→ **Incrementar a 40 Mflops la performance del procesador**

- Habrá 75 % de cache hit

→ **Tiempo por palabra =  $0,25 \times 100 \text{ ns} + 0,75 \times 1 \text{ ns} = 25,75 \text{ ns}$**

# Impacto del Ancho de Banda de la Memoria. Ejemplo.

- Dado el siguiente código:

```
for (i = 0; i < 1000; i++)  
    { suma[i] = 0.0;  
      for (j = 0; j < 1000; j++) suma[i] += b[j][i];  
    }
```

- ¿Cuál es el efecto? → Sumar las columnas de la matriz **B** en el vector *suma*.
- ¿Cuál es el problema si la matriz está en memoria por filas?

Fila 0	Fila 1	Fila 2	....	Fila 999
--------	--------	--------	------	----------

- ¿Cuál es el efecto y la diferencia al modificar el código?

```
for (i = 0; i < 1000; i++) column_sum[i] = 0.0;  
for (j = 0; j < 1000; j++)  
    for (i = 0; i < 1000; i++) column_sum[i] += b[j][i];
```

# Resumen de ideas sobre la performance del Sistema de Memoria

- Hay que tratar de explotar la localidad temporal y espacial de los datos.
- La relación entre el número de operaciones de procesamiento y el número de accesos a memoria es muy importante para establecer la performance efectiva.
- La relación entre el algoritmo y la forma de estructurar los datos en memoria influirá sensiblemente en la performance.





---

# Plataformas de Cómputo Paralelo

---

# Plataformas de cómputo paralelo

Las plataformas de cómputo paralelo se pueden clasificar por:

- **Organización Lógica:** clasificación desde el punto de vista del programador.
  - **Mecanismo de Control:** forma de expresar tareas paralelas.
  - **Modelo de comunicación:** mecanismo para especificar la interacción entre las tareas.
- **Organización Física:** clasificación desde el punto de vista del hardware.
  - **Espacio de direcciones:** relacionado con el modelo de comunicación.
  - **Red de interconexión:** forma que se conectan entre sí los procesadores y/o los bloques de memorias.
  - **Granularidad:** relación entre la cantidad y potencia de procesadores y la velocidad de la red de comunicación.

# Clasificación por el Mecanismo de Control

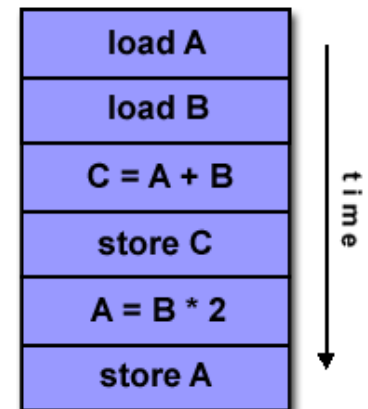
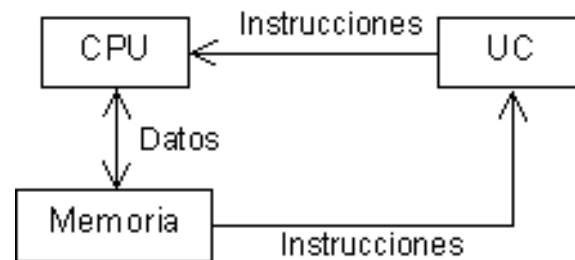
Flynn clasifica las computadoras según el número de instrucciones y datos que se pueden procesar simultáneamente:

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

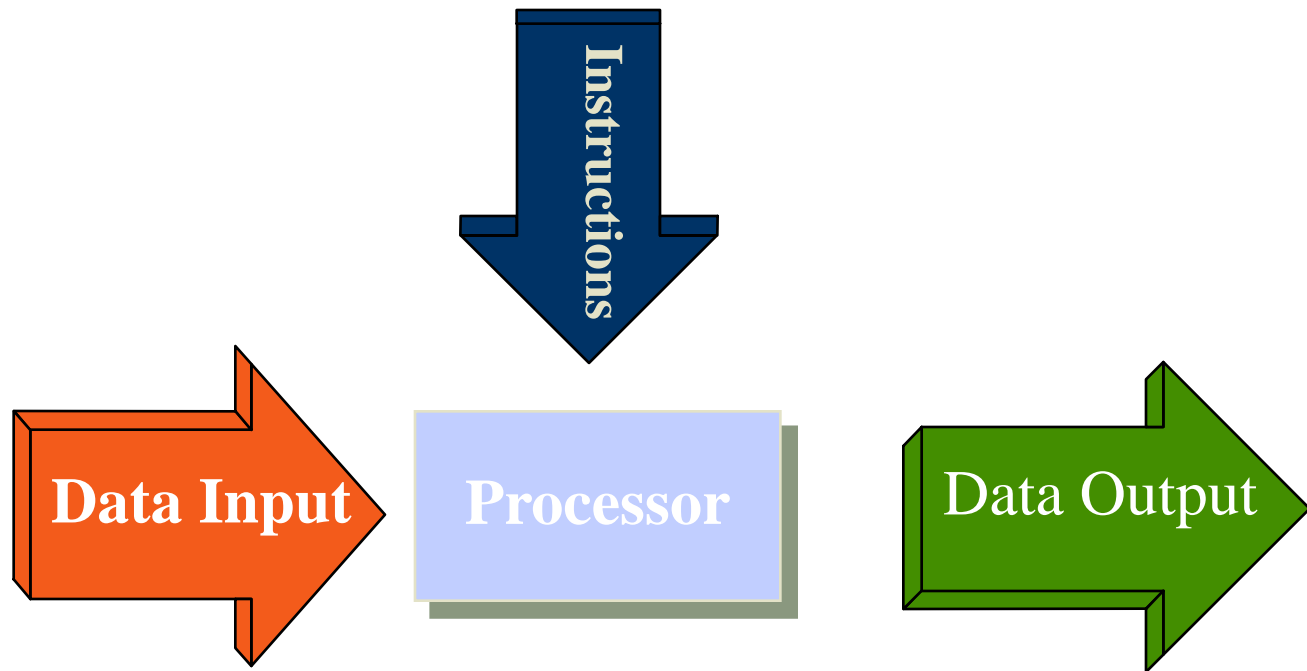
# Clasificación por el Mecanismo de Control: *SISD*

## *SISD: Single Instruction and Single Data*

- Instrucciones ejecutadas en secuencia, una por ciclo de instrucción.
- La memoria afectada es usada sólo por ésta instrucción.
- 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 por el Mecanismo de Control: *SISD*

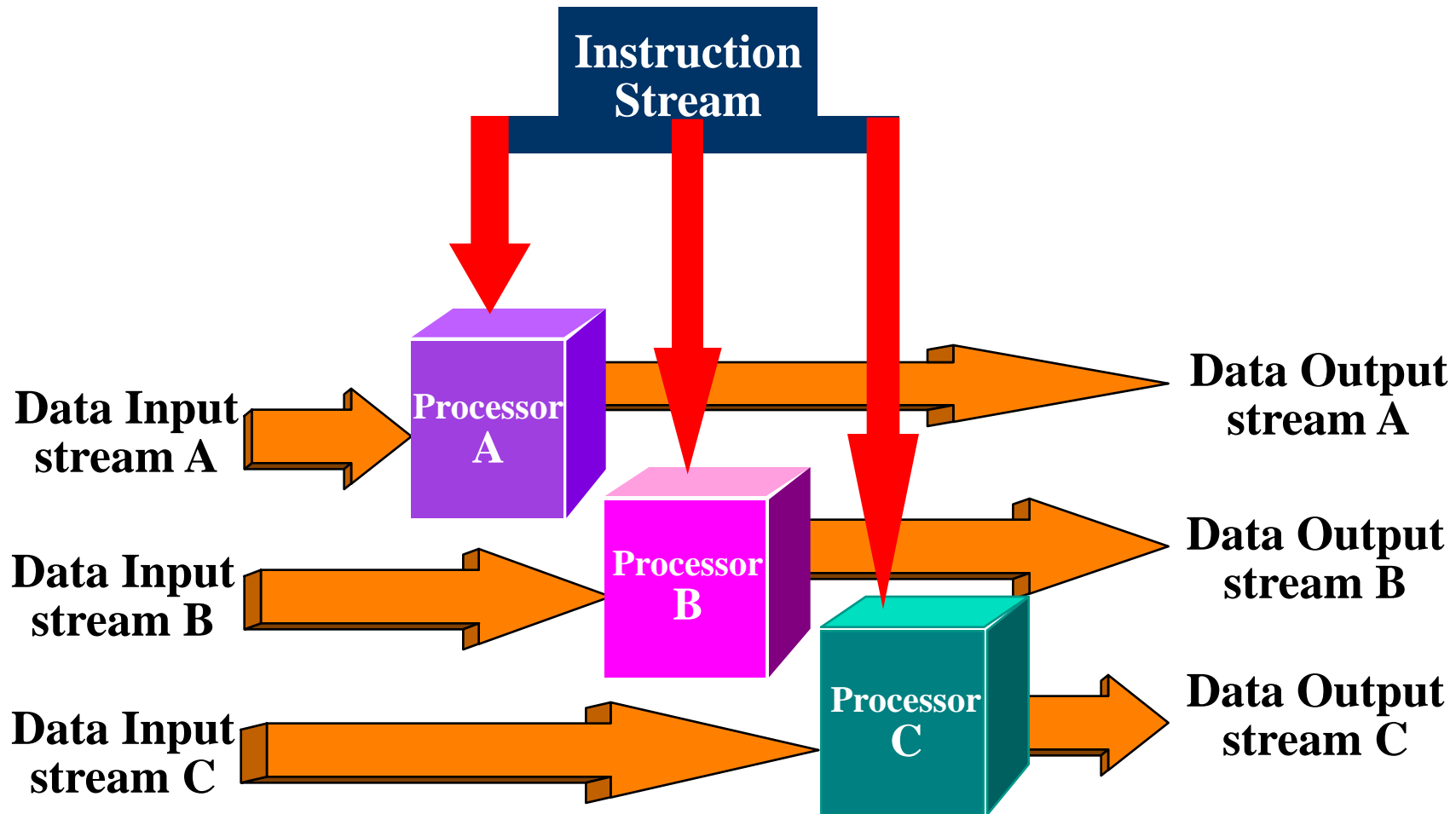


# Clasificación por el Mecanismo de Control: *SIMD*

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

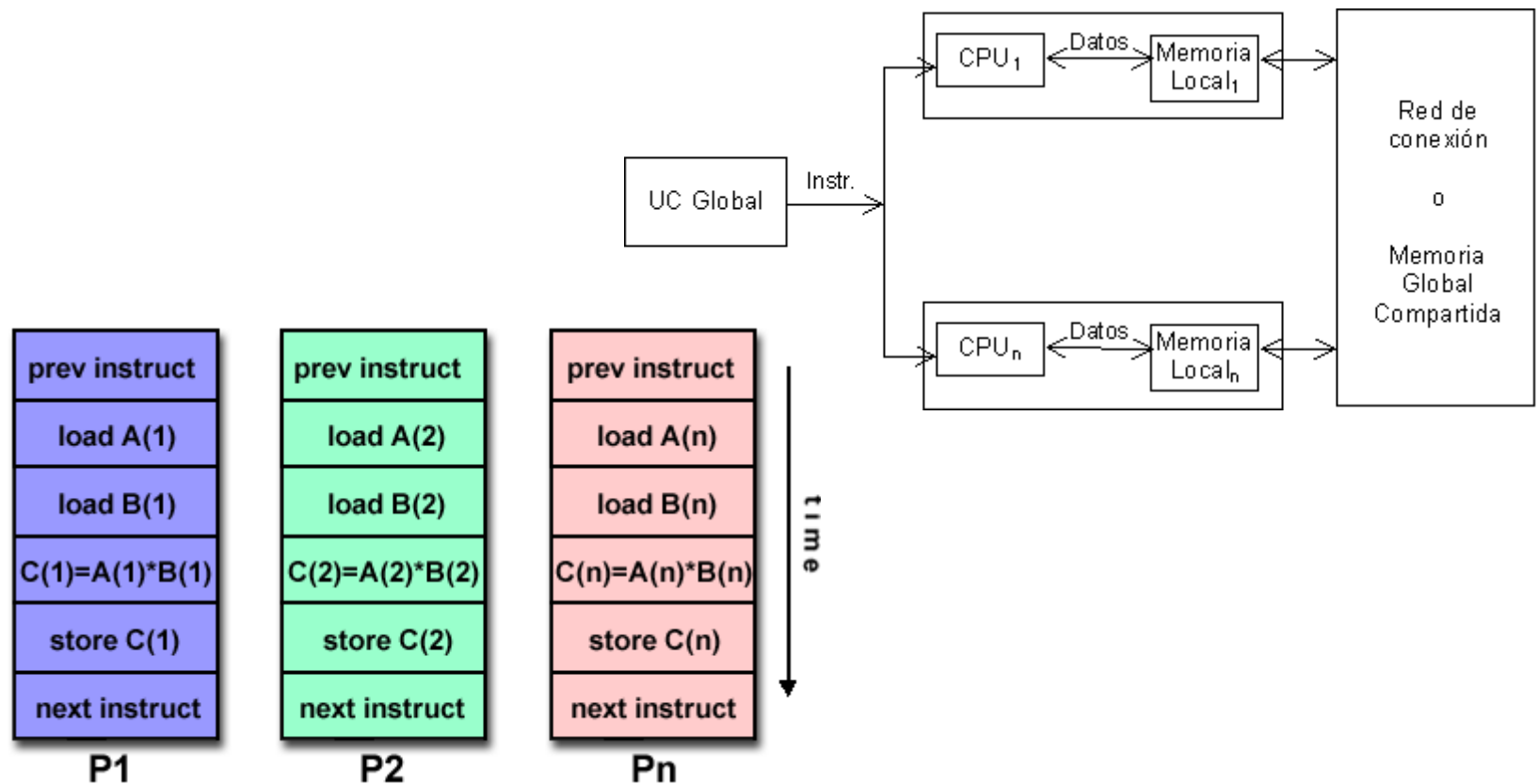
- Conjunto de procesadores idénticos, con sus memorias, que ejecutan la misma instrucción sobre distintos datos.
- Los procesadores en general son muy simples.
- El *host* hace broadcast de la instrucción.
- Ejecución sincrónica y determinística.
- Pueden deshabilitarse y habilitarse selectivamente procesadores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

# Clasificación por el Mecanismo de Control: *SIMD*



# Clasificación por el Mecanismo de Control: *SIMD*

**Ejemplos de máquina SIMD:** Array Processors o Máquinas Vectoriales.



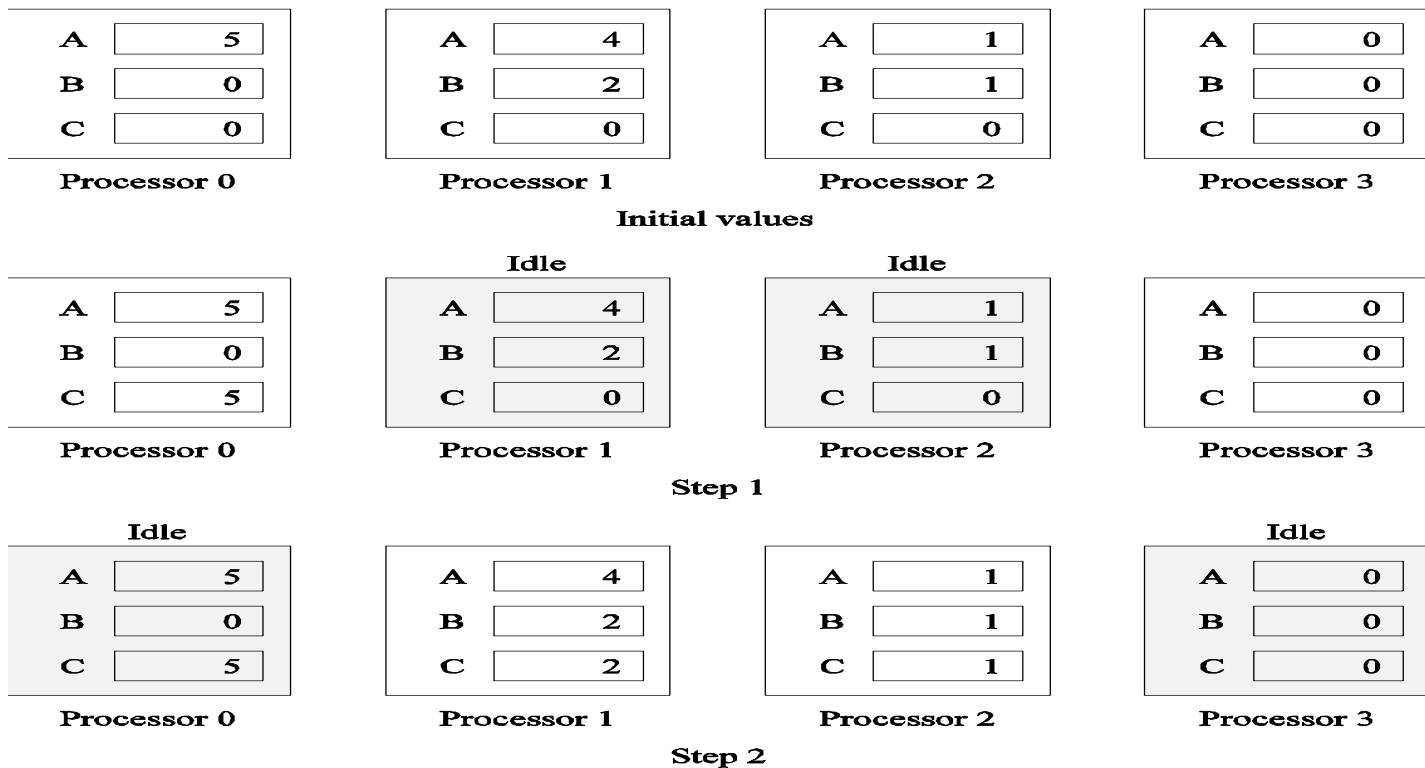


# Clasificación por el Mecanismo de Control: *SIMD*

## Ejecución de una sentencia condicional

```
if (B == 0)
    C = A;
else
    C = A/B;
```

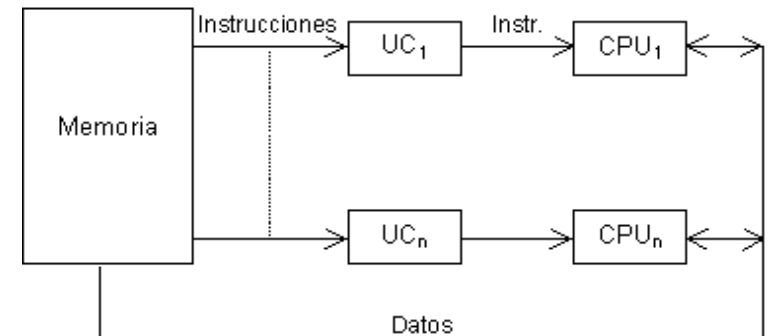
(a)



# Clasificación por el Mecanismo de Control: *MISD*

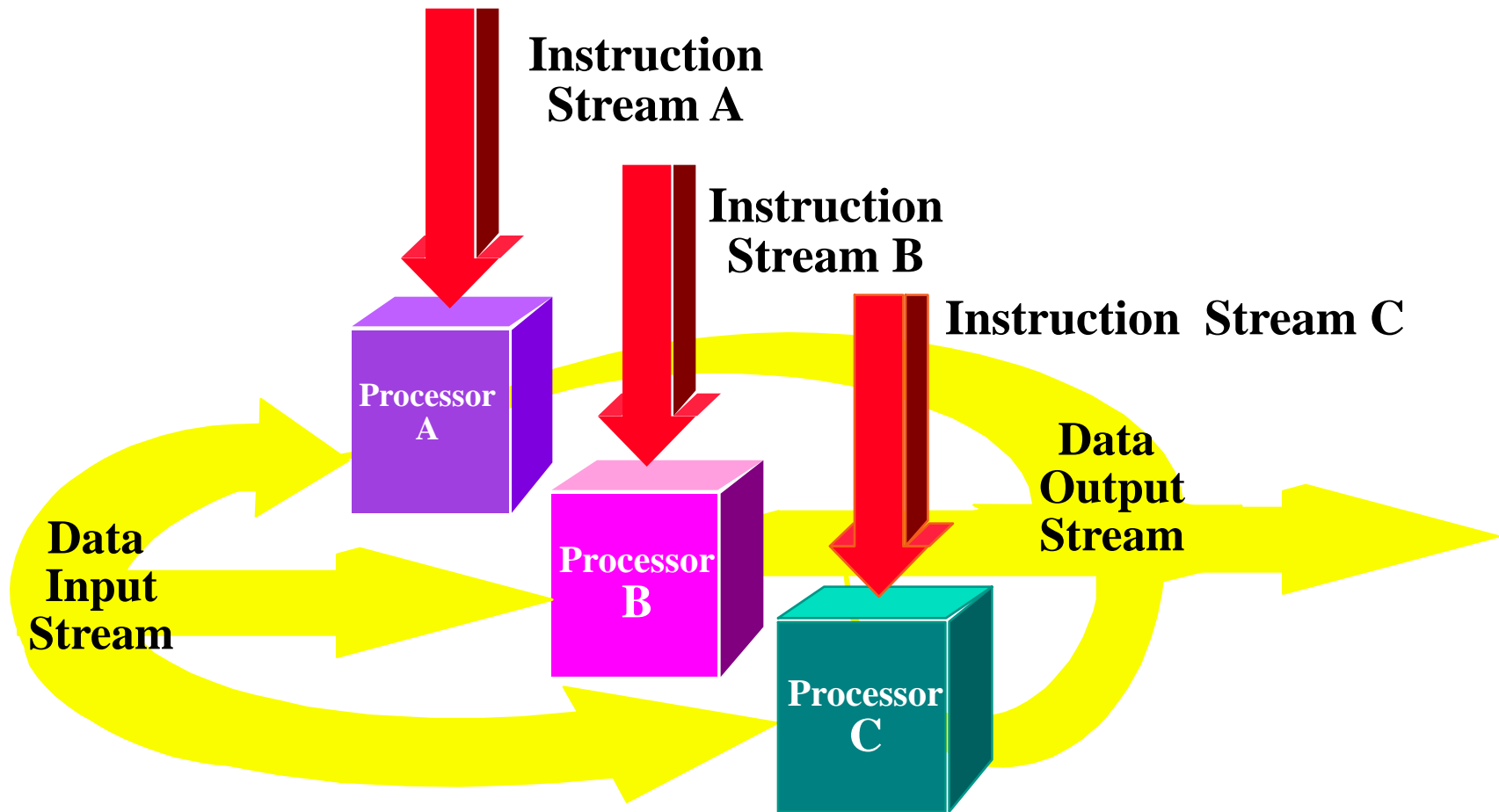
## *MISD: Multiple Instruction and Single Data*

- Los procesadores ejecutan un flujo de instrucciones distinto pero comparten datos comunes.
- Operación sincrónica (en lockstep).
- No son máquinas de propósito general.



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

# Clasificación por el Mecanismo de Control: *MISD*



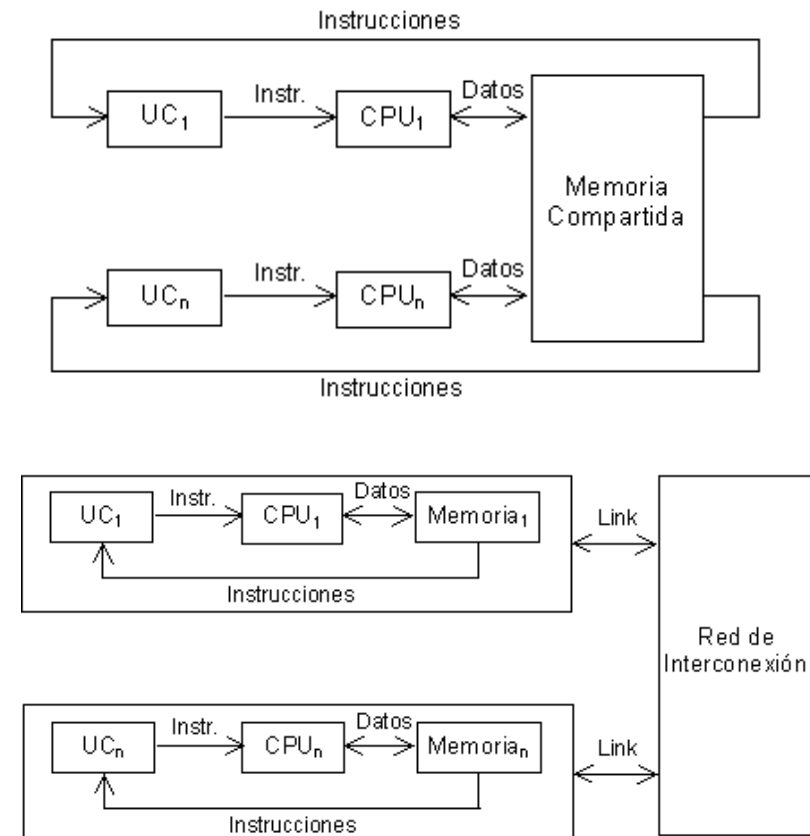
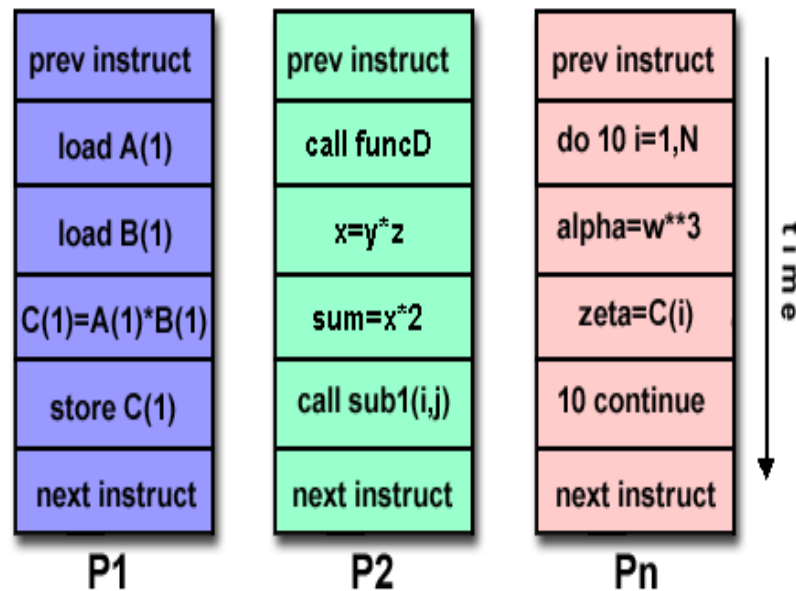
# Clasificación por el Mecanismo de Control: *MIMD*

## **MIMD: *Multiple Instruction and Multiple Data***

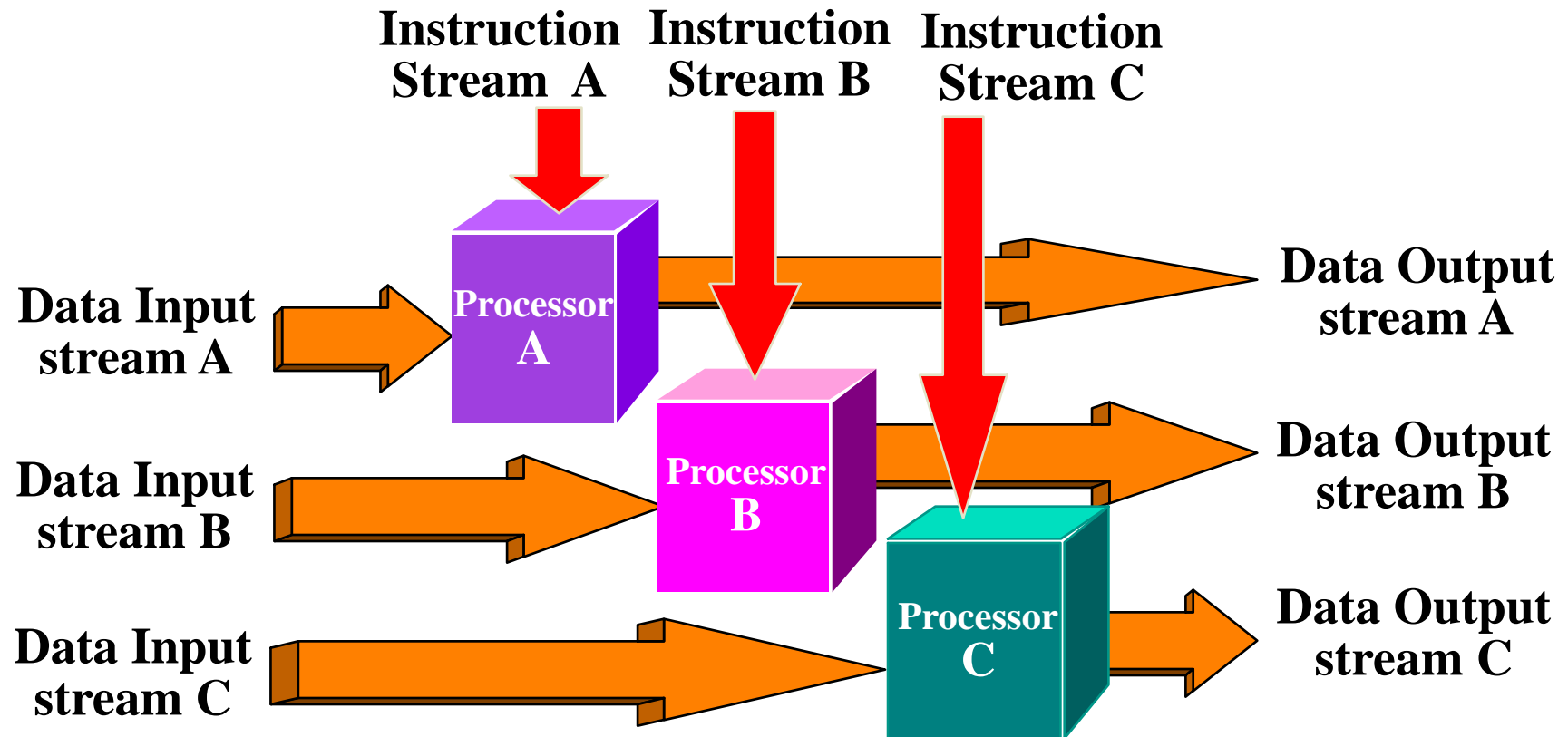
- Cada procesador tiene su propio flujo de instrucciones y de datos → cada uno ejecuta su propio programa en forma asincrónica.
- Pueden ser con memoria compartida o distribuida.
- Los esquemas paralelos MIMD tienden a ser más económicos (para igual performance global) que los esquemas SIMD.
- Subclasificación de MIMD:
  - *MPMD (multiple program multiple data)*.
  - *SPMD (single program multiple data)*.

# Clasificación por el Mecanismo de Control: *MIMD*

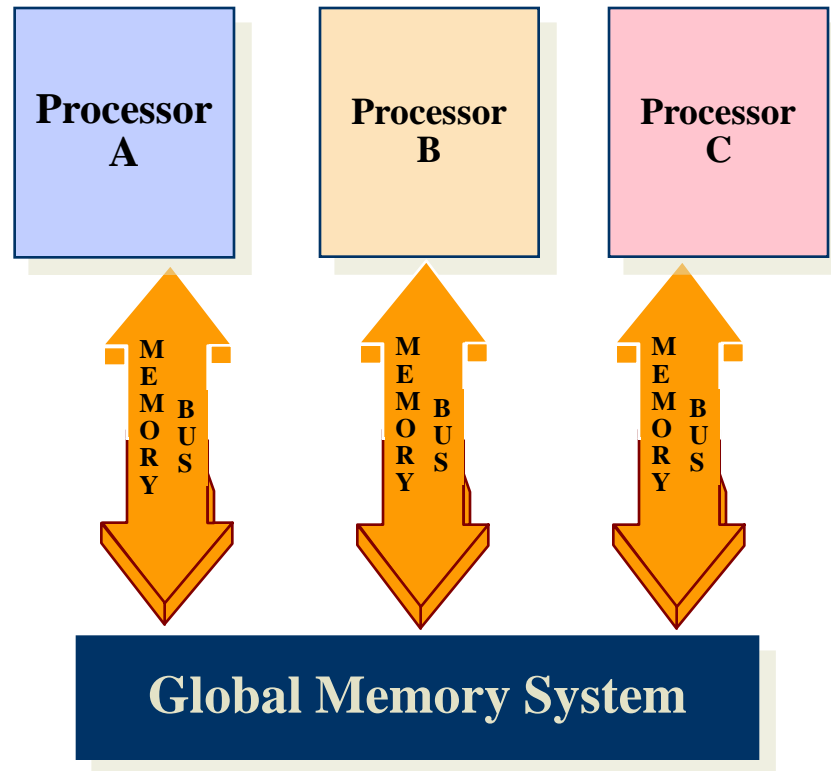
**Ejemplos de máquina MIMD:** multicores, clusters, grid, etc.



# Clasificación por el Mecanismo de Control: *MIMD*

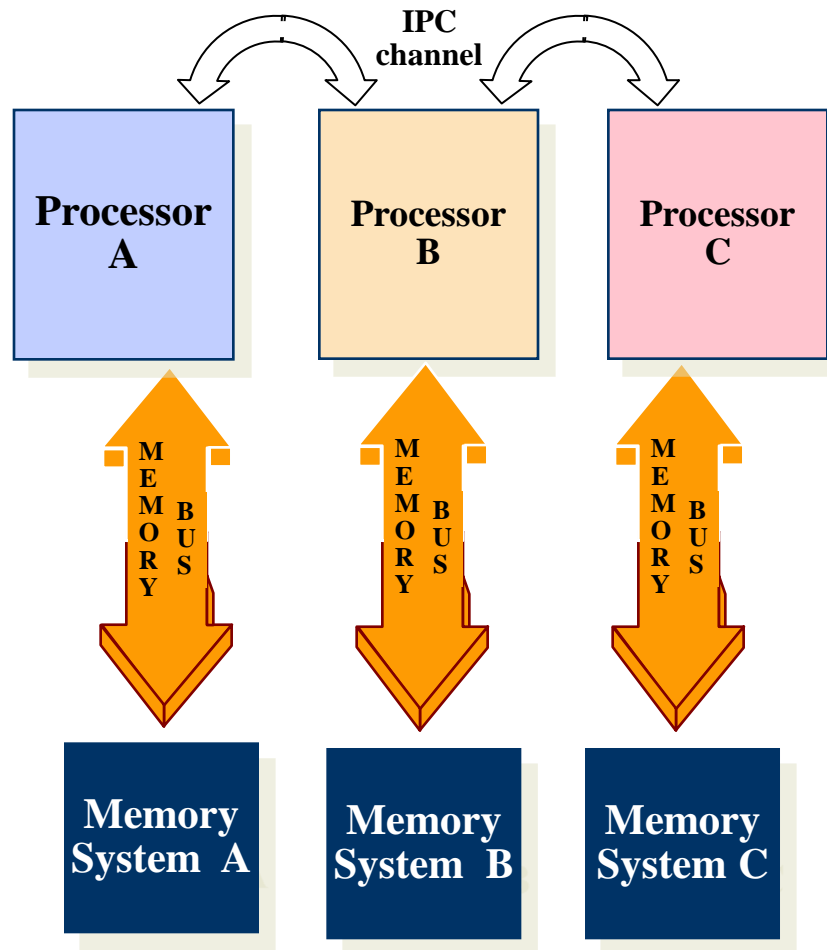


# Arquitectura MIMD con Memoria Compartida



- Fácil de construir. SO convencional. Software portable.
- Limitación: Confiabilidad y expandibilidad.
- El aumento del número de procesadores tiende a memory contention.

# Arquitectura MIMD con Memoria Distribuida



➤ Comunicación : IPC (Inter-Process Communication) vía Red de Alta Velocidad.

➤ La red puede ser configurada como Tree, Mesh, Cube, etc.

→ *Facilidad y bajo costo de expansión.*

→ *Altamente fiable (la falla de una CPU no afecta el sistema completo)*



# Clasificación por el modelo de comunicación

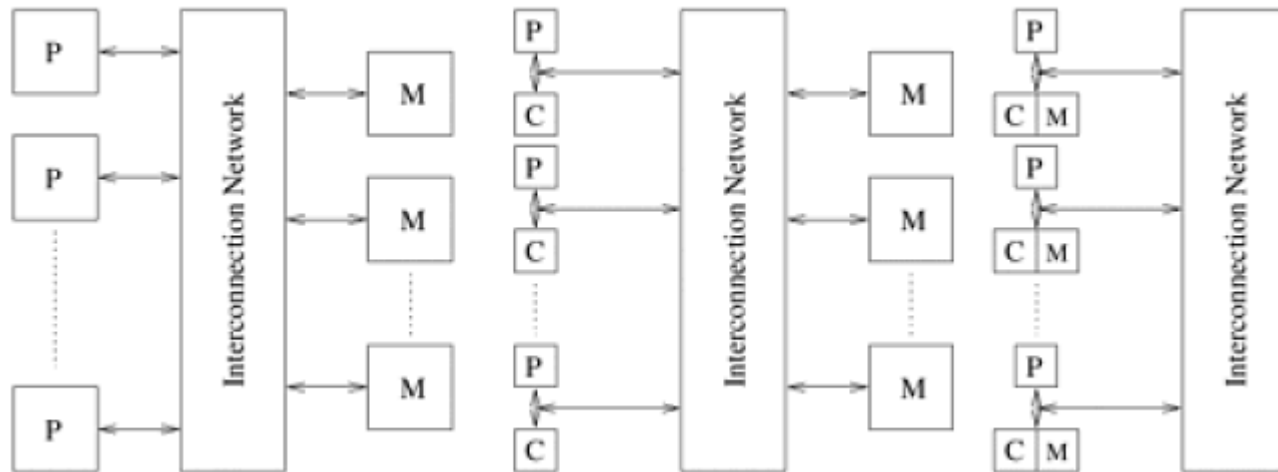
- Los procesos se pueden comunicar de diferente manera:
  - Accediendo a Memoria Compartida.
  - Intercambiando Mensajes.
- Para ambos modelos de comunicación existen plataformas de procesamiento paralelo.
- En algunos casos también tenemos en la misma plataforma ambos mecanismos.

# Clasificación por el modelo de comunicación:

## *Memoria Compartida*

### Multiprocesadores de memoria compartida

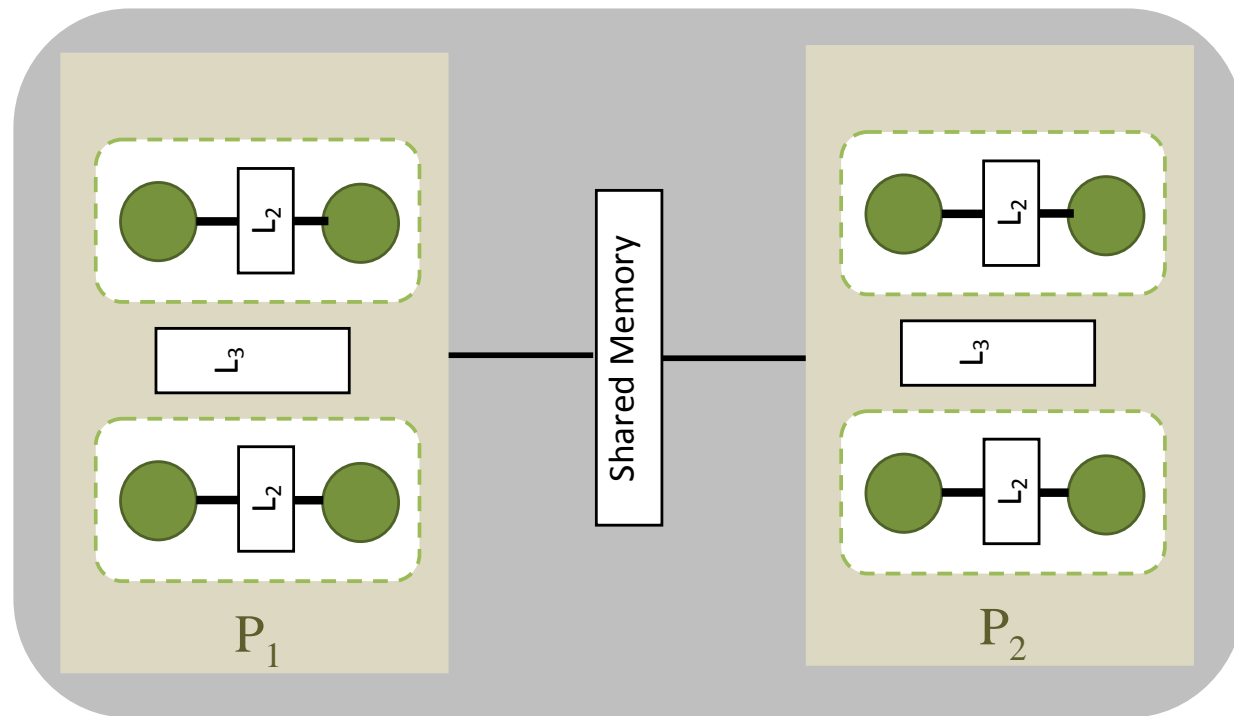
- Interacción modificando datos en la memoria compartida.
- Esquemas UMA con bus o crossbar switch (Shared Memory).
- Esquemas NUMA para mayor número de procesadores distribuidos (Shared Address Space).
- Problema de consistencia.



# Clasificación por el modelo de comunicación:

## *Memoria Compartida*

Ejemplo de *multiprocesador de memoria compartida*: multicore de 8 núcleos.



# Clasificación por el modelo de comunicación:

## *Memoria Distribuida*

### **Multiprocesadores de memoria distribuida**

- Procesadores conectados por una red.
- Memoria local (no hay problemas de consistencia).
- Interacción es sólo por pasaje de mensajes.
- Grado de acoplamiento de los procesadores:
  - Multicomputadores (tightly coupled machine). Procesadores y red físicamente cerca. Pocas aplicaciones a la vez, cada una usando un conjunto de procesadores. Alto ancho de banda y velocidad.
  - Memoria compartida distribuida.
  - NOWs / Clusters.
  - Redes (loosely coupled multiprocessor).



# Clasificación por la Granularidad

Relación entre cómputo/comunicación:

- **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.

Es importante el *matching* entre la arquitectura y la aplicación.

# Clasificación 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.).



---

# Topologías de Redes de Interconexión

---

# Topologías de Redes de Interconexión

- Existe una variedad de topologías.
- El compromiso es entre costo y performance.
- En muchos casos las topologías se combinan en esquemas híbridos para optimizar la performance global de los sistemas distribuidos/paralelos.
- En todos los casos, el conocimiento de la topología DEBE influir en el diseño de los algoritmos.

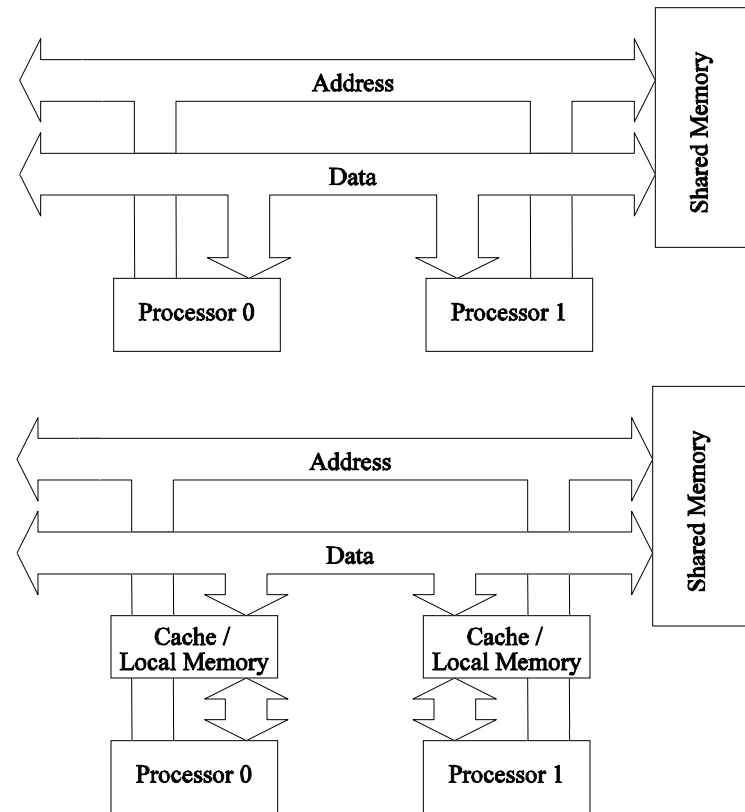


# Topologías de Redes: *Buses*

- Desde los inicios de las máquinas paralelas, la topología más simple ha sido el BUS.
- Todos los procesadores acceden al bus para el intercambio de datos.
- Ventajas:
  - Distancia entre nodos es  $O(1)$ .
  - Mecanismo simple de broadcasting.
  - El costo escala linealmente con el número de nodos.
- Desventaja: Ancho de Banda limitado.

Red Limitada a decenas de Nodos

Mejora usando Memoria Local



# Topologías de Redes: *Crossbars*

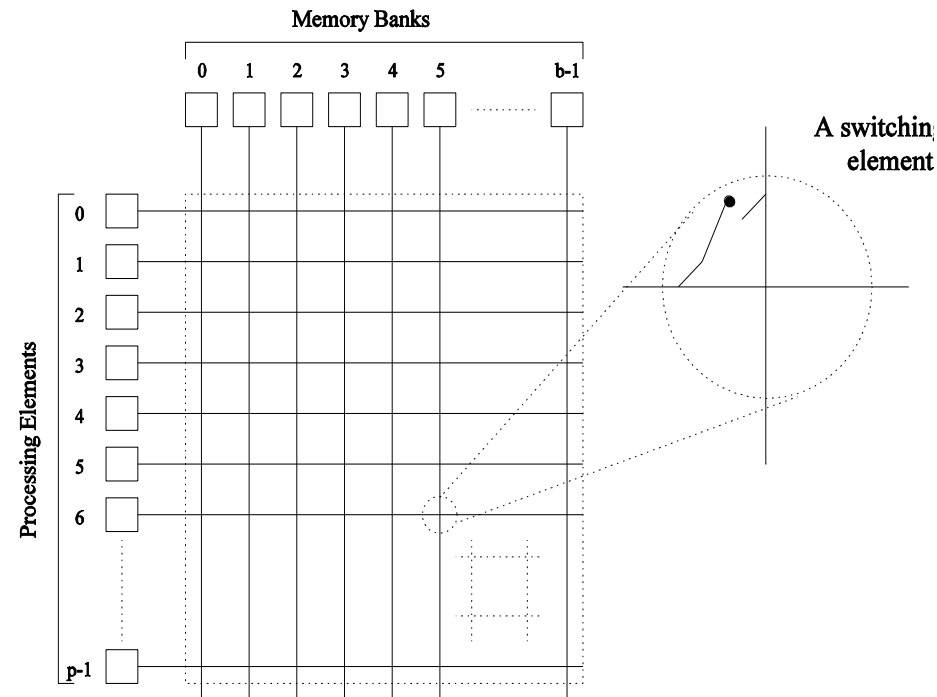
- Una red crossbar utilizar una grilla de  $p \times m$  switches para conectar  $p$  inputs a  $m$  outputs, de modo no bloqueante.

- Ventajas:

- Acceso rápido y constante.
- Acceso no bloqueante.

- Desventajas:

- Costo crece en  $O(p \times m)$ .
- Difícil de escalar los esquemas que suponen “full connection” entre los nodos.



$P$  procesadores conectados con  $B$  memorias

# Topologías de Redes: *Multistage*

## ➤ Buses:

- Excelente escalabilidad por costo.
- Pobre performance (muchos procesadores).

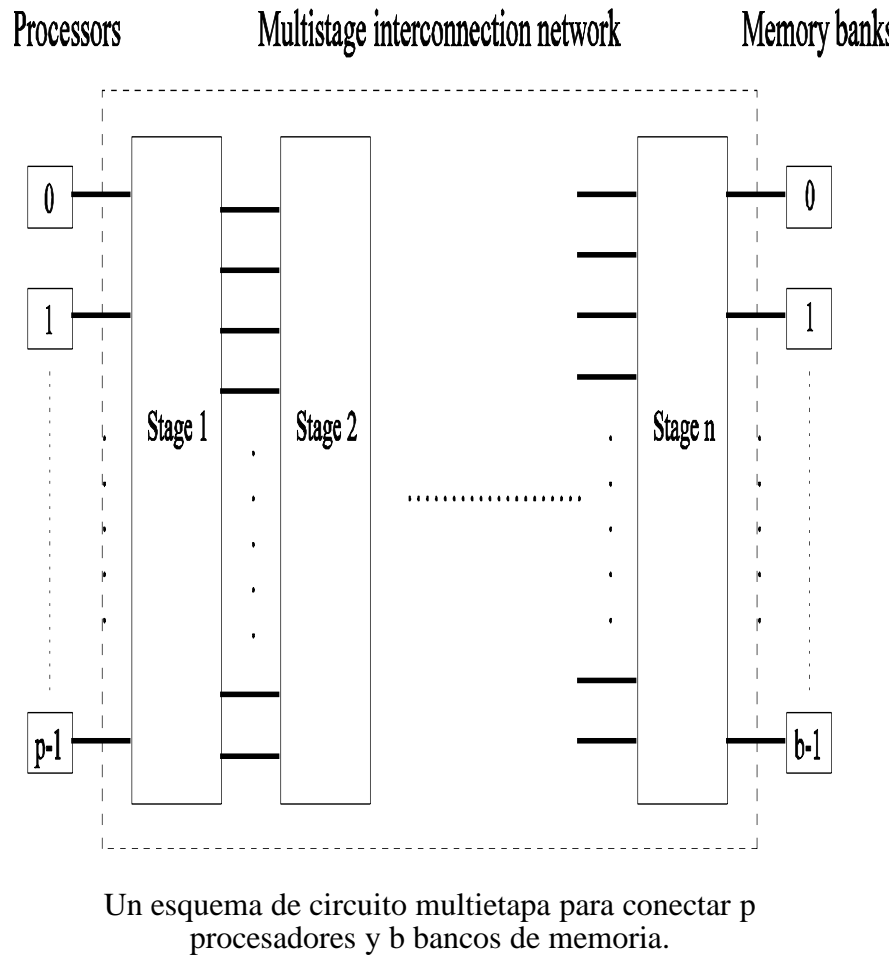
## ➤ Crossbar:

- Baja escalabilidad por costo.
- Excelente performance.



***Redes Multistage***: tratan de lograr un compromiso entre ambas.

# Topologías de Redes: *Multistage Omega*

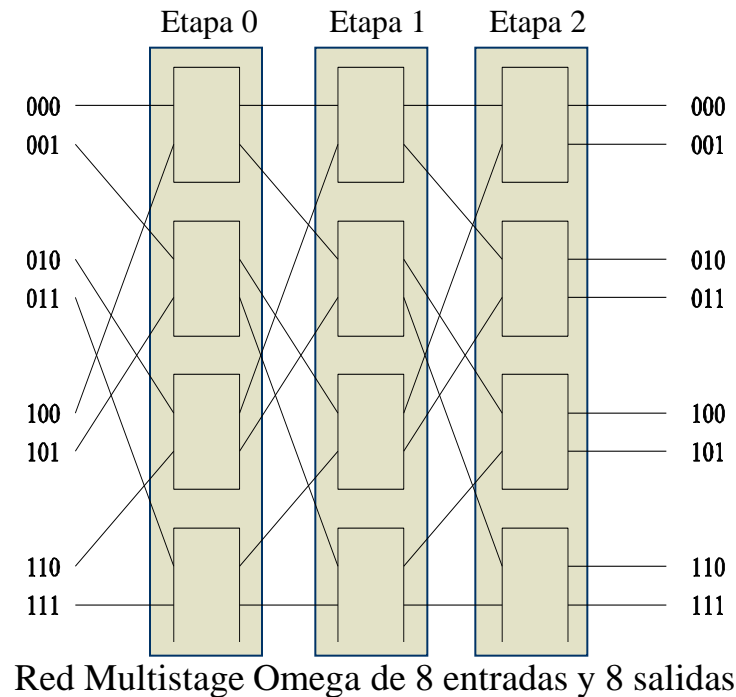
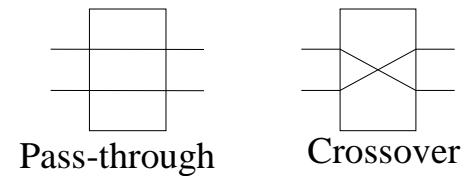


- El esquema Omega es una de las formas clásicas de conexión en redes multistage.
- Cuenta con  $\log p$  etapas, donde  $p$  es el número de E y S.
- Cada etapa consiste de un patrón de interconexión llamado Perfect Shuffle que conecta  $p$  E y  $p$  S.
- Existe un link entre la entrada  $i$  y la salida  $j$  si:

$$j = \begin{cases} 2i, & 0 \leq i \leq \frac{p}{2} - 1 \\ 2i + 1 - p, & \frac{p}{2} \leq i \leq p - 1 \end{cases}$$

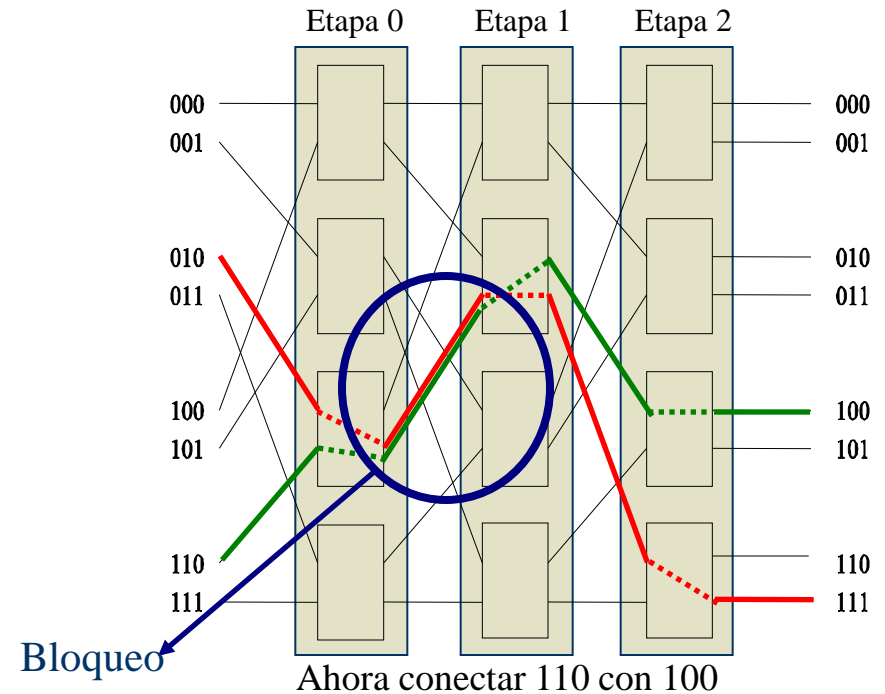
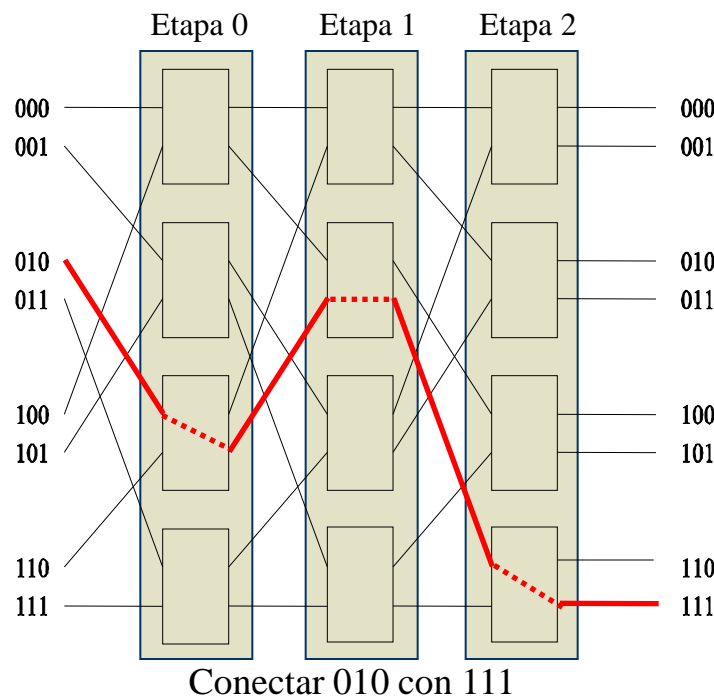
# Topologías de Redes: *Multistage Omega*

- Cada etapa tiene  $p/2$  switch de tipo  $2 \times 2$  (2 E y 2 S).
- Cada switch tiene 2 posibles modos:
  - Pass-through.
  - Crossover



# Topologías de Redes: *Multistage Omega*

- Si tenemos que conectar la entrada O y la salida D se debe:
- Los datos pasan al nodo de switch correspondiente de la primera etapa. Si los bits más significativos de O y D coinciden, se rutea usando modo “pass through”, si no, modo “crossover”.
  - Esto se repite en cada una de las  $\log p$  etapas con los bits siguientes.



# Topologías de Redes: *Multistage Omega*

- El costo es:  $\left(\frac{p}{2} \times \log p\right)$  switch ( $<$  que  $p^2$  de la red Crossbar).
- La comunicación se realiza en  $\log p$  pasos.
- No es totalmente no bloqueante (Blocking Network).

# Topologías de Redes: *Conectadas Completamente*

➤ Links directos entre todos los nodos ( $p^2$  links).

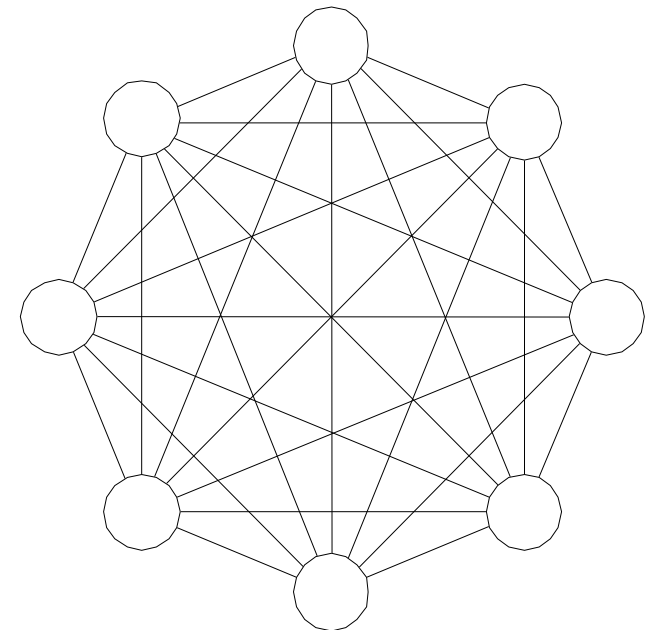
➤ Contraparte estática de las redes Crossbar.

➤ Ventajas:

- Excelente performance.
- No bloqueante.

➤ Desventajas:

- Baja escalabilidad por costo.

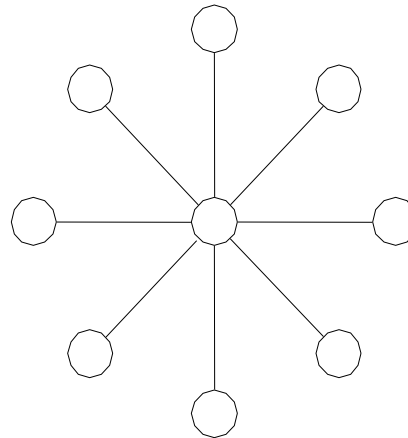


Red Conectada Completamente de 8 nodos



# Topologías de Redes: *Conectadas en Estrella*

- Todos los nodos conectados a un único nodo central.
- Cada comunicación se hace en dos pasos a través del nodo central (cuello de botella).
- Contraparte estática de los Buses.



Red Conectada en estrella de 9 nodos

# Otras Topologías Estáticas

- Redes conectadas completamente requieren muchos links (costosa escalabilidad).

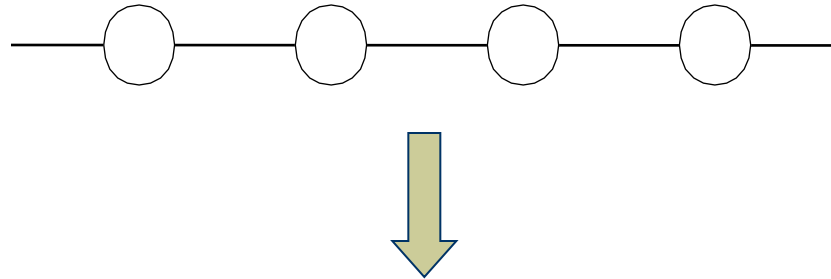


## Redes esparcidas.

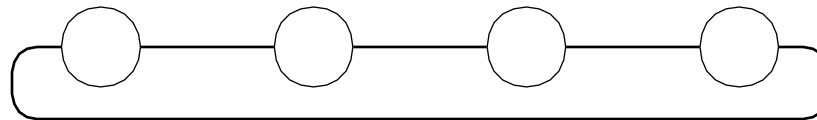
- Arreglos Lineales.
- k-d Meshes.

# Topologías de Redes: *Arreglos Lineales*

- Cada nodo tiene 2 vecinos (salvo los extremos).
- Comunicación entre extremos → costosa.

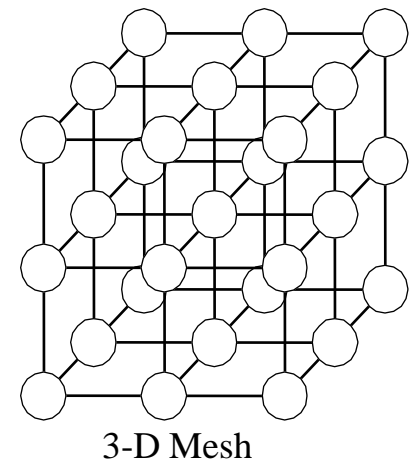
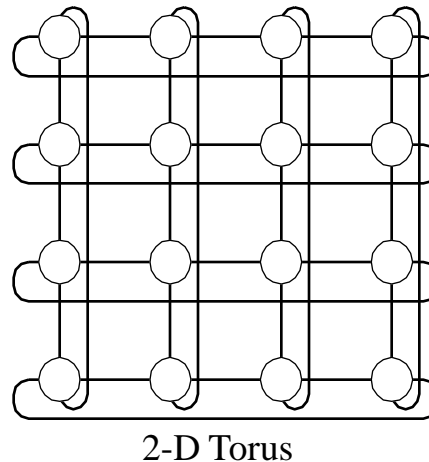
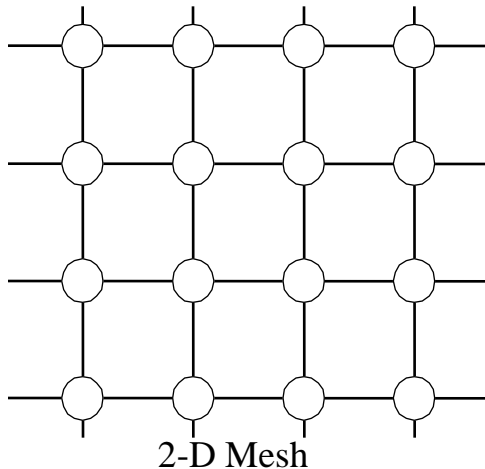


- Redes Ring → nodos extremos unidos.
- Comunicación entre extremos mitad de costosa.



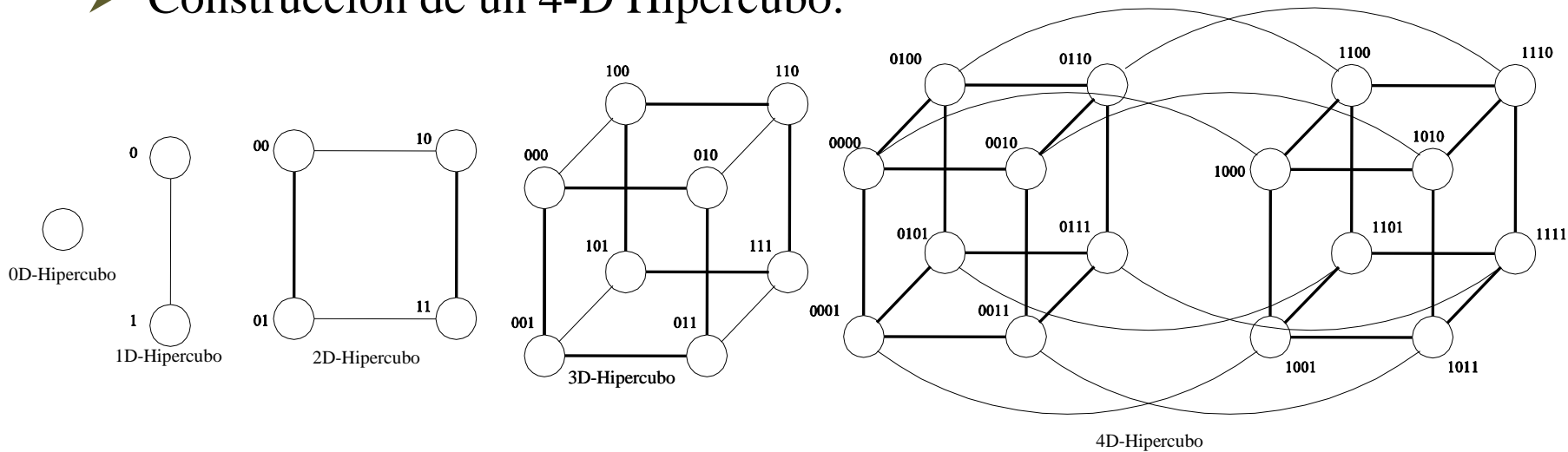
# Topologías de Redes: *k-d Mesh*

- Posee  $d$  dimensiones. Con 2 vecinos en cada dimensión ( $2d$  vecinos en total) (salvo los periféricos) .
- En cada dimensión  $k = \sqrt[d]{p}$  nodos.
- De arreglos lineales ( $d = 1$ ) a hipercubos ( $d = \log p$  y  $k = 2$ ).
- Si los extremos se conectan en forma circular se llama Torus.



# Topologías de Redes: *Hipercubos*

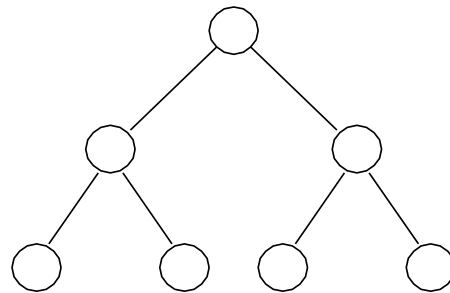
## ➤ Construcción de un 4-D Hipercubo.



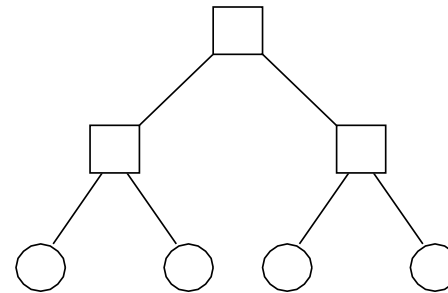
## ***Propiedades:***

- Cada nodo tiene  $\log p$  vecinos (están todos en la periferia).
- Distancia entre dos nodos  $\rightarrow$  bits diferentes entre los nodos.
- Distancia máxima  $\rightarrow \log p$ .

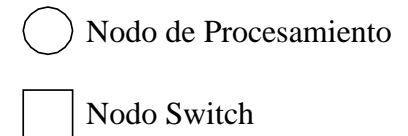
# Topologías de Redes: *Redes Basadas en Árboles* (*Estáticas y Dinámicas*)



Estática



Dinámica



➤ Solo existe un camino entre cada par de nodos.

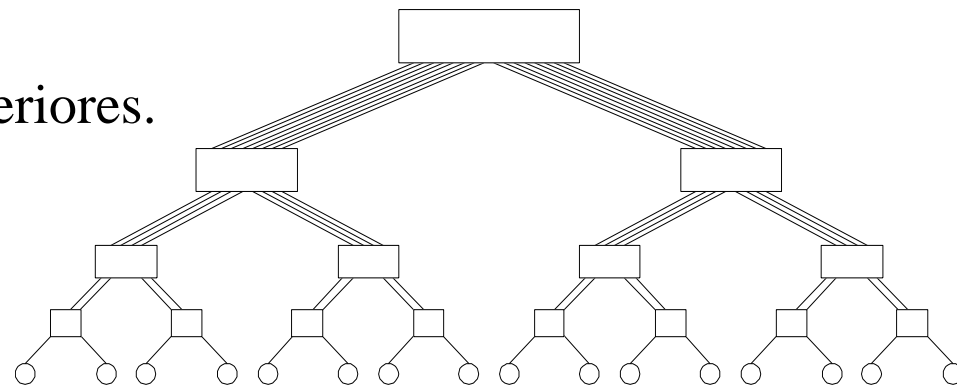
➤ Distancia máxima:

- Redes Estáticas:  $2 (\log(p+1) - 1)$ .
- Redes Dinámicas:  $2 \log(p)$ .


➤ Cuello de botella en nodos superiores.



***Redes Fat Tree***



Red Dinámica Fat Tree con 16 nodos de Procesamiento



---

# Coherencia de cache en memoria compartida

---

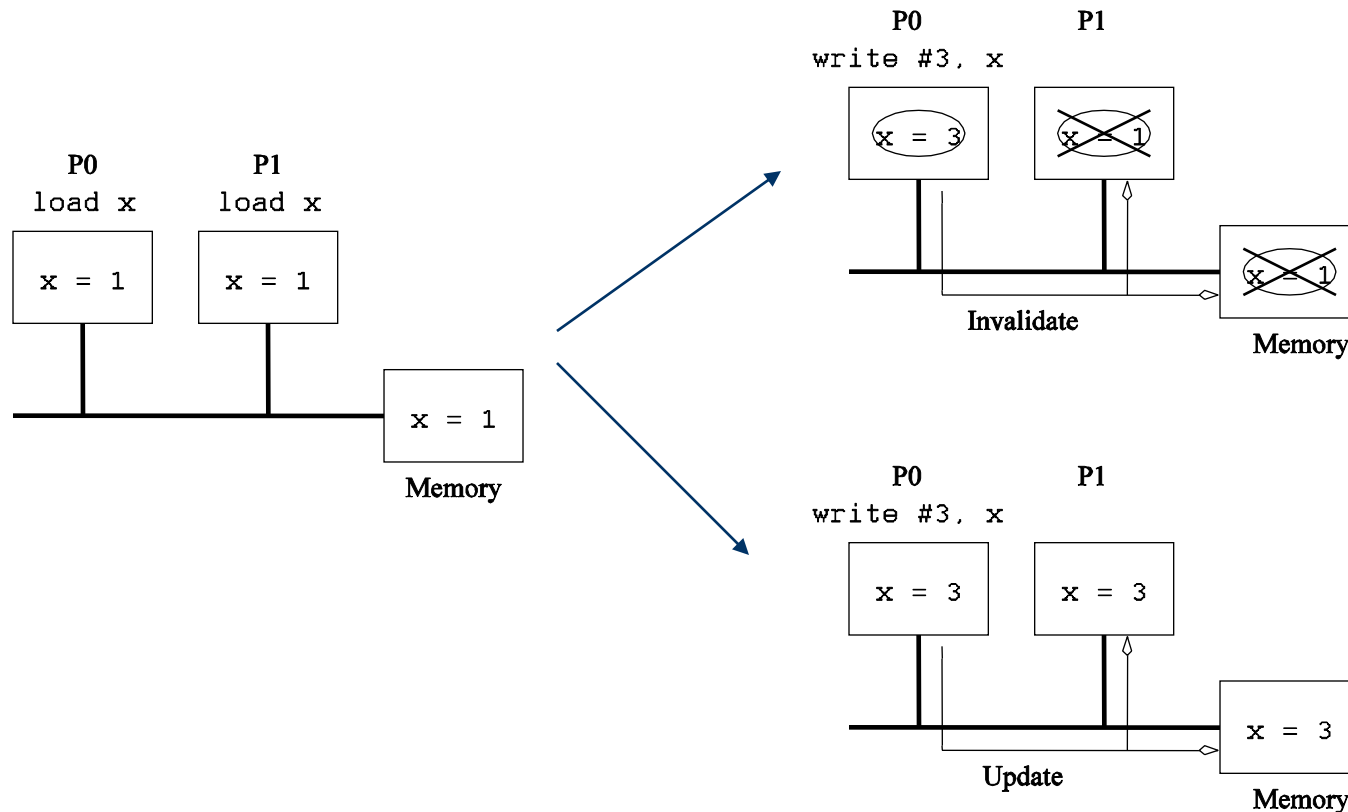
# Coherencia de Cache en Sistemas Multiprocesador

- Los esquemas de interconexión aseguran la comunicación (o transferencia de datos).
- Si tenemos memoria compartida entre procesadores (que pueden tener datos replicados) debemos asegurar la coordinación en el acceso a los mismos.
- Además hay que asegurar la semántica de esta coordinación para que los datos conserven coherencia. Normalmente esto significará alguna serialización en las instrucciones sobre la máquina paralela.



# Coherencia de Cache en Sistemas Multiprocesador

Cuando el contenido de una variable cambia, todas sus copias deben ser actualizadas o invalidadas.

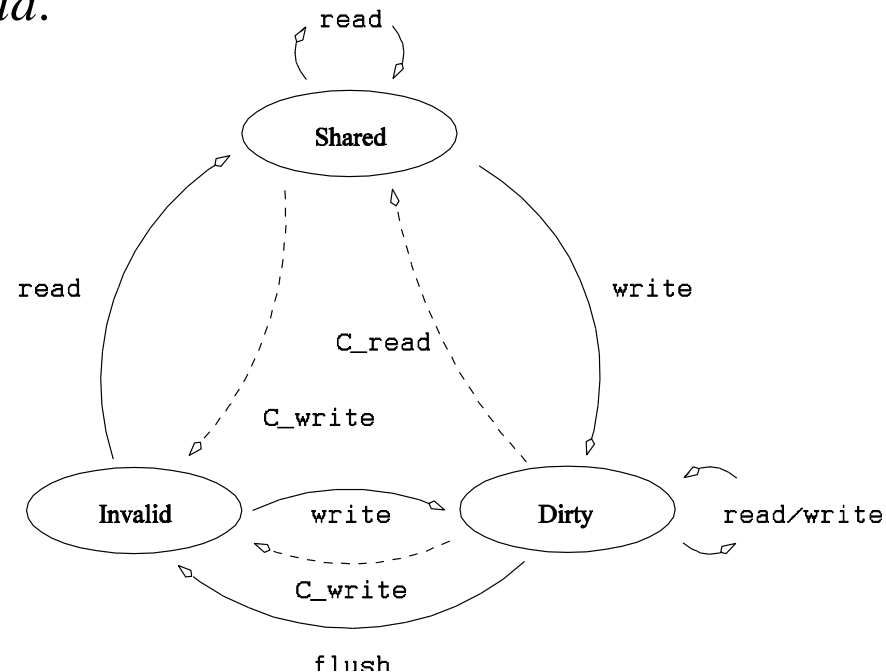


# Coherencia de Cache: Protocolos Update e Invalidate

- Si un procesador sólo lee un valor una vez y no lo necesita más, un protocolo update puede generar un overhead significativo innecesario. En este caso es mejor el protocolo invalidate.
- Si 2 procesadores trabajan en forma coordinada sobre una variable (alternativamente) será mejor un protocolo de actualización.
- Ambos protocolos sufren de “false sharing” cuando en la misma línea de memoria caché se almacenan datos no compartidos (overhead innecesario).
- La mayoría de las máquinas paralelas utilizan protocolos de invalidación.
- Tradeoff entre overhead de comunicación (update) y ociosidad de procesadores (invalidate).

# Manteniendo la coherencia con protocolos Invalidate

- Cada copia de un dato se asocia con uno de tres estados: *shared*, *invalid* o *dirty*.
- En el estado *shared*, hay múltiples copias válidas del dato (en diferentes memorias). Si hubiera una WRITE, se producirá en ese lugar (pasando a estado *dirty*) y el resto pasará a *invalid*.
- En estado *dirty*, sólo hay una copia y las actualizaciones se hacen a partir de esa copia.
- En estado *invalid* un READ genera un pedido de dato al punto donde esté la copia válida (estado *dirty*).



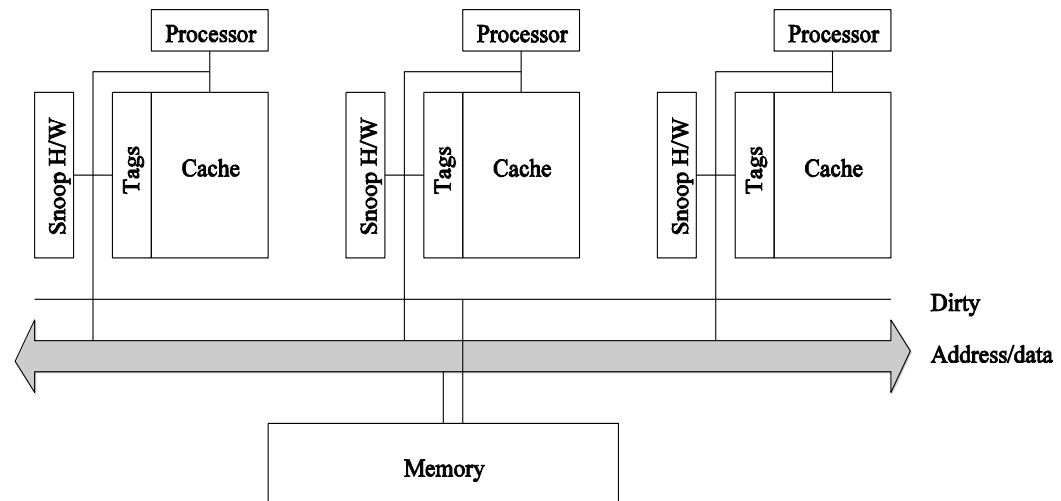
# Manteniendo la coherencia con protocolos Invalidate:

## Ejemplo de ejecución paralela

Time ↓	Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
					x = 5, D y = 12, D
	read x	read y	x = 5, S	y = 12, S	x = 5, S y = 12, S
	x = x + 1	y = y + 1	x = 6, D	y = 13, D	x = 5, I y = 12, I
	read y	read x	y = 13, S x = 6, S	y = 13, S x = 6, S	y = 13, S x = 6, S
	x = x + y	y = x + y	x = 19, D y = 13, I	x = 6, I y = 19, D	x = 6, I y = 13, I
	x = x + 1	y = y + 1	x = 20, D	y = 20, D	x = 6, I y = 13, I

# Sistemas de Cache Snoopy

- Asociado a sistemas basado en broadcast (bus y ring).
- En los modelos de caché “snoopy” hay un canal de broadcast que se encarga de enviar los mensajes para asegurar la coherencia local.
- Si un procesador detecta un READ sobre una variable cuya copia está en estado *dirty* envía el dato.
- Si un procesador detecta un WRITE sobre una variable y tiene una copia la pone en estado *invalid*.



# Sistemas de Cache Snoopy

Hay operaciones que no agregan overhead (se manejan en forma local). Las operaciones de coherencia (no locales) producen overhead innecesario al enviar broadcast avisando a todos los procesadores (aunque no tengan una copia).



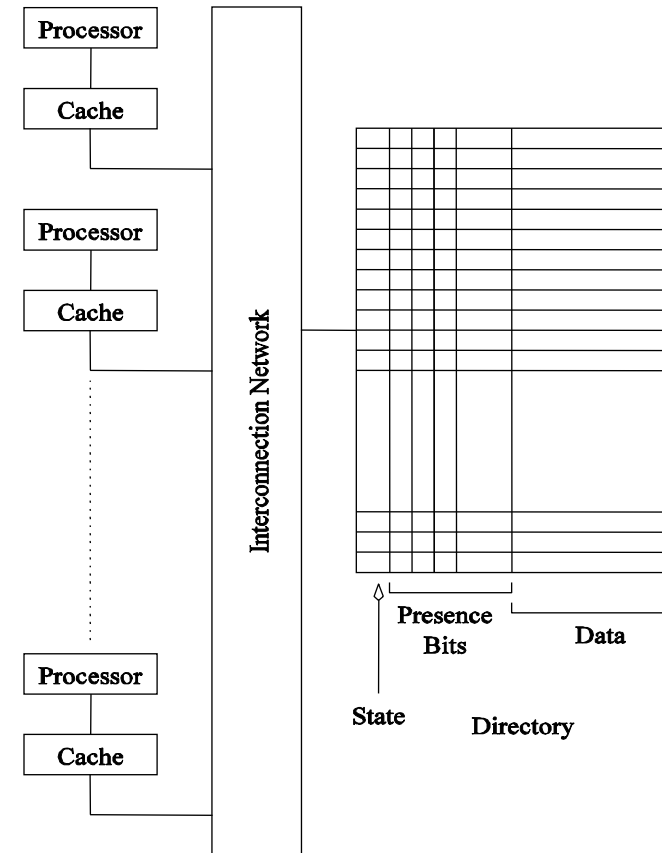
**Cuellos de Botella**



**Sistemas basados en directorios**

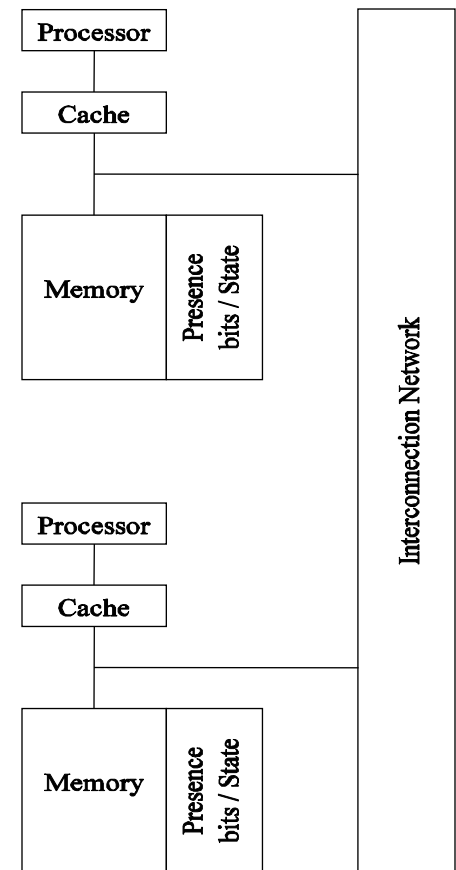
# Sistemas basados en Directorios

- En los cachés con un sistema snoopy, las operaciones para mantener la coherencia se mandan como broadcast a todos los procesadores → baja en performance.
- Una idea simple es mantener un Directorio con la relación entre el dato y los procesadores que tienen copia del mismo → notificar sólo a los procesadores correctos (reduce el overhead innecesario).
- El directorio en memoria limita la cantidad de operaciones de READ/WRITE simultaneas → **Cuello de Botella**.
  - Menos bloques de memoria de mayor tamaño (Falso Sharing).
  - Sistema de Directorios Distribuidos.



# Sistemas basados en Directorios Distribuidos

- Memoria físicamente distribuida entre los procesadores.
- Cada procesador es responsable de la coherencia de sus bloques (posee el directorio correspondiente).
- Cuando un procesador requiere un READ calcula el procesador dueño del bloque y le hace el pedido. De acuerdo al estado y al *presence bit* este resuelve el pedido.
- Cuando se hace un WRITE se propaga un *invalidate* al procesador dueño, el cual lo reenvía a los procesadores que tienen una copia.
- Más escalable que los Directorios Centralizados.
- La latencia y el ancho de banda de la red son el principal cuello de botella.







---

# Costos de comunicación

---

# Costos de la comunicación en Máquinas Paralelas

- Uno de los principales overhead en programas paralelos se relaciona con la comunicación de información entre procesadores.
- El costo de comunicaciones depende de muchos factores (además del medio físico). Entre éstos están los protocolos de software, la topología de la red, las técnicas de routing y manejo de datos.
- Costos diferentes según la forma de comunicación:
  - Pasaje de Mensajes.
  - Memoria Compartida.

# Costos del pasaje de mensajes en Máquinas Paralelas

- El tiempo total para transferir un mensaje sobre una red involucra:
  - ***Startup time ( $t_s$ )***: Es el tiempo para preparar el mensaje y establecer el routing, conectando virtualmente a los nodos origen y destino.
  - ***Per-hop time ( $t_h$ )***: Es el tiempo que tarda en llegar el header de un mensaje de un nodo al siguiente (***latencia del nodo***).
  - ***Per-word transfer time ( $t_w$ )***: Es el tiempo para comunicar una palabra del mensaje entre dos nodos vecinos. Siendo  $r$  el ancho de banda de la red,  $t_w = 1/r$ .
- El costo de la comunicación depende de la técnica de ruteo que se utilice. Por ejemplo:
  - ***Store-and-Forward.***
  - ***Cut-Through.***

# Ruteo Store-and-Forward

- Un mensaje que atraviesa múltiples hops debe recibirse completo en los nodos intermedios antes de hacer un forward al siguiente hop.

- Si el mensaje tiene un tamaño de  $m$  palabras y atraviesa  $l$  links de comunicación:

$$t_{comm} = t_s + (mt_w + t_h)l.$$

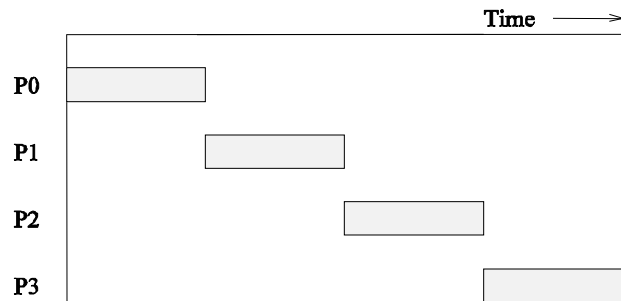
- En la mayoría de los casos,  $t_h$  es pequeño y la expresión puede aproximarse por:

$$t_{comm} = t_s + mlt_w.$$

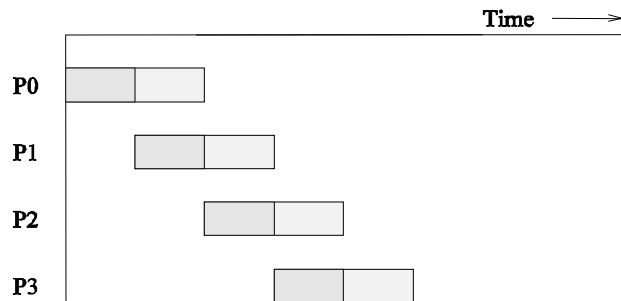
## Ruteo de Paquetes

- Dividir el mensaje en paquetes más pequeños y transmitirlos separados (reduce el tiempo de la comunicación) {Paquetes vs mensajes}
  - Mientras envía un paquete al siguiente nodo puede recibir otro paquete.
  - Bajo overhead por paquetes perdidos.
  - Envío de paquetes por caminos diferentes.
  - Transporta información extra: ruteo, corrección de errores y secuenciación de paquetes.
  - Tiempo de comunicación aproximado:  $t_{comm} = t_s + t_h l + t_w m$ .

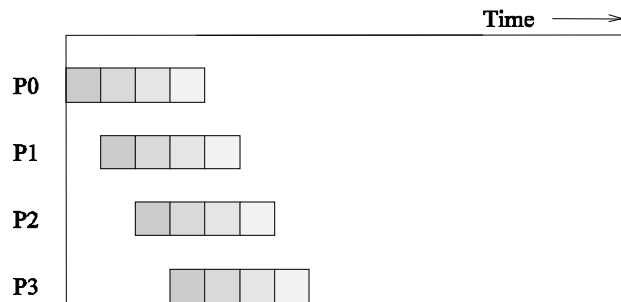
# Ruteo Store-and-Forward



Mensaje enviado con la técnica Store-and-Forward



Mensaje dividido en dos partes y enviados



Mensaje dividido en cuatro partes y enviados

# Ruteo Cut-Through

- Esquema optimizado donde se reduce el overhead mediante:
  - Todos los paquetes por el mismo camino ➔ Elimina información de ruteo.
  - Los paquetes se envían en orden ➔ Elimina información de secuencialidad.
  - Información de errores a nivel de mensaje ➔ Reduce información de detección y corrección de errores.
- **Tracer** para establecer conexión entre origen y destino.
- Mensajes divididos en **flits** (flow control digits).
- Los **Flits** siguen el camino del **tracer**. Cada nodo recibe un **flit** y lo envía inmediatamente al siguiente nodo.
- Tiempo de comunicación:  $t_{comm} = t_s + l t_h + t_w m \rightarrow t_{comm} \cong t_s + t_w m$

# Modelo de costo para máquinas de memoria compartida

En la comunicación con memoria compartida hay que considerar una serie de factores cuando se tiene memoria compartida:

- UMA o NUMA.
- Tamaños de caché y overflows de caches (genera operaciones de coherencia).
- Overhead variable de las operaciones de Update / Invalidate.
- Es difícil modelar el efecto de la ubicación espacial y de las técnicas de prefetching que se puedan tener en cada procesador/unidad de memoria.
- Falsos sharing no predecibles.