

# **Sistemas de multiprocesadores**

José Galaviz

# Taxonomía de Flynn

- Michael J. Flynn, 1966.
- SISD. Single Instruction (stream), Single Data (stream).
- MISD. Multiple Instruction (stream), Single Data (stream).
- SIMD. Single Instruction (stream), Multiple Data (stream).
- MIMD. Multiple Instruction (stream), Multiple Data (stream).

# SISD

Este es el paradigma donde, de hecho, no hay paralelismo. Un único procesador ejecutando un único flujo de instrucciones sobre un único flujo de datos.

# MISD

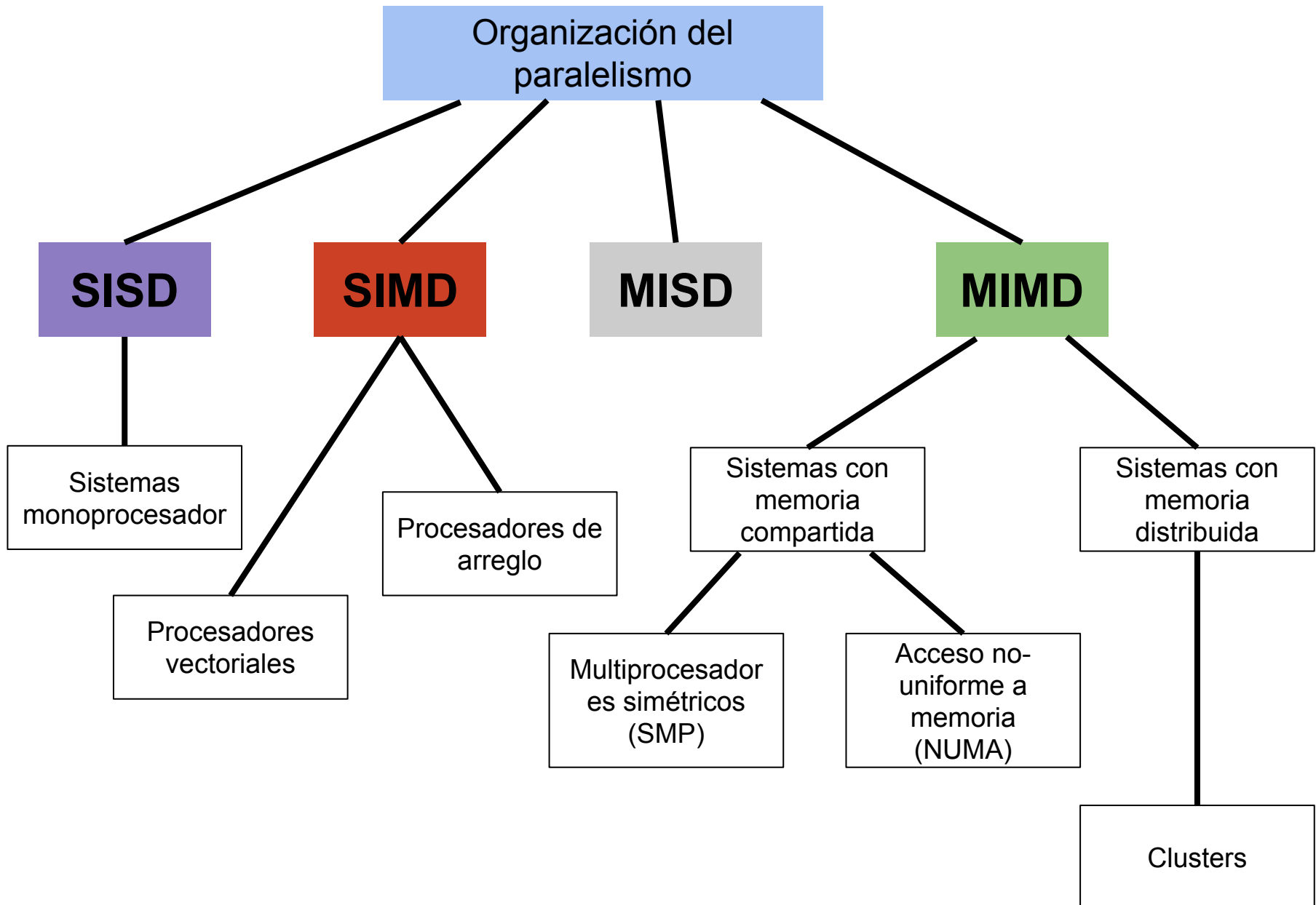
No es un paradigma muy usual. Hay textos que dicen que no han existido computadoras así. Algunos autores colocan a los arreglos sistólicos (Kung y Leiserson, 1978) en esta categoría. La Colossus Mark II (1944) estaba diseñada con un paradigma similar. En todo caso no es un paradigma comercial.

# SIMD

- Varios procesadores ejecutan la misma operación simultáneamente sobre diferentes puntos.
- Explota el paralelismo en los datos.
- Las supercomputadoras vectoriales del pasado eran SIMD.
- Las unidades de cómputo multimedia son SIMD.

# MIMD

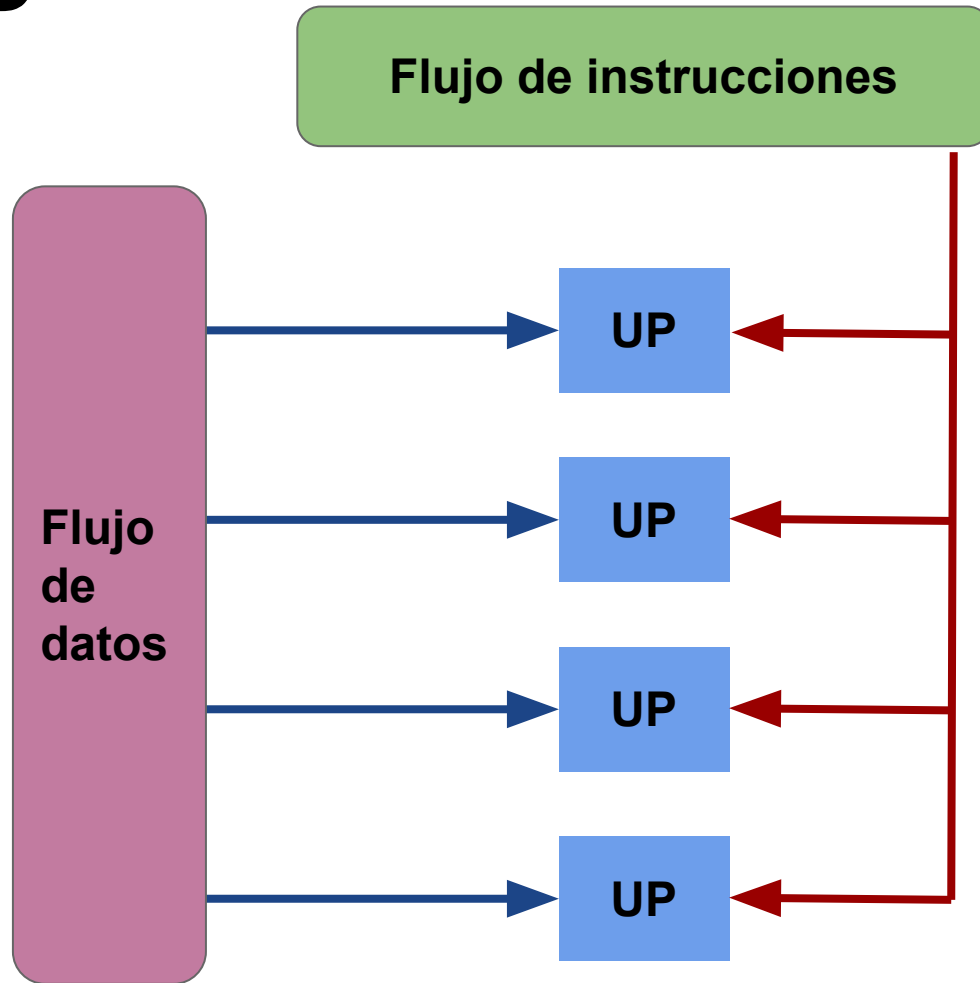
Varios procesadores ejecutando diferentes instrucciones sobre diferentes conjuntos de datos.



**SIMD**



# SIMD



**UP** = Unidad de Procesamiento

## Cómputo escalar

$$\begin{array}{ccccc} \boxed{A_x} & + & \boxed{B_x} & = & \boxed{C_x} \\ \boxed{A_y} & + & \boxed{B_y} & = & \boxed{C_y} \\ \boxed{A_z} & + & \boxed{B_z} & = & \boxed{C_z} \\ \boxed{A_w} & + & \boxed{B_w} & = & \boxed{C_w} \end{array}$$

## Cómputo vectorial

$$\begin{array}{|c|} \hline \boxed{A_x} \\ \hline \boxed{A_y} \\ \hline \boxed{A_z} \\ \hline \boxed{A_w} \\ \hline \end{array} + \begin{array}{|c|} \hline \boxed{B_x} \\ \hline \boxed{B_y} \\ \hline \boxed{B_z} \\ \hline \boxed{B_w} \\ \hline \end{array} = \begin{array}{|c|} \hline \boxed{C_x} \\ \hline \boxed{C_y} \\ \hline \boxed{C_z} \\ \hline \boxed{C_w} \\ \hline \end{array}$$

# Array vs. Vector Processors

Instruction Stream

```
LD  VR ← A[3:0]
ADD VR ← VR, 1
MUL VR ← VR, 2
ST  A[3:0] ← VR
```

ARRAY PROCESSOR



Same op @ same time



Different ops @ same space

Time

Space

VECTOR PROCESSOR



Different ops @ time



Same op @ space

Space

# Array Vs. Vector Processors

- La distinción es sutil.
- Hoy en día todos los procesadores SIMD son una combinación de ambos.

# Ventajas de SIMD

- Las operaciones inherentes al cómputo visual y multimedia son vectoriales. Por eso hoy día toda GPU es SIMD.
- Se hace fetch de una sola instrucción y se aplica a un varios puntos de una sola vez.
- Se replica la ruta de datos, pero no la unidad de control.

# Desventajas de SIMD

- No todo programa es fácilmente vectorizable y la vectorización se hace, mayormente, a mano.
- Alto consumo intrínseco al mantener un gran número de registros.
- Los conjuntos de instrucciones SIMD son particulares de cada arquitectura, las adecuaciones no son transportables.
- La memoria es un cuello de botella potencial.

# **supercomputadoras de antaño**

- Las supercomputadoras típicamente eran multiprocesadores vectoriales.
- Cray Research.
- 1970 a 1990.
- Luego su ventaja se desvaneció.

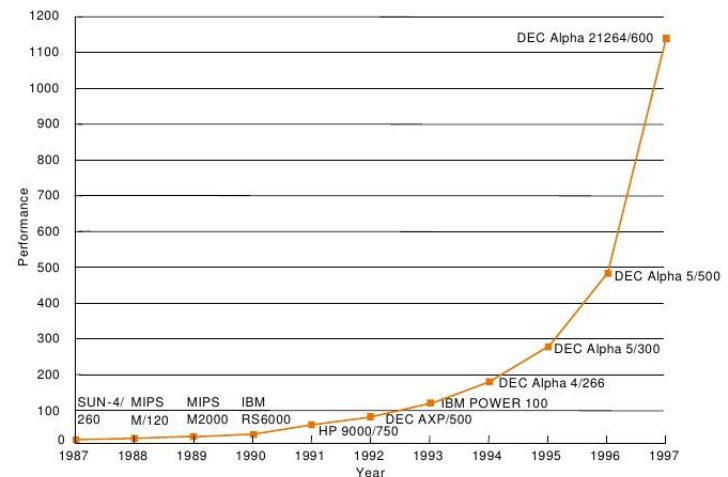
# Processor Perspective

- Putting performance growth in perspective:

	IBM POWER2 Workstation	Cray YMP Supercomputer
Year	1993	1988
MIPS	> 200 MIPS	< 50 MIPS
Linpack	140 MFLOPS	160 MFLOPS
Cost	\$120,000	\$1M (\$1.6M in 1994\$)
Clock	71.5 MHz	167 MHz
Cache	256 KB	0.25 KB
Memory	512 MB	256 MB

- 1988 supercomputer in 1993 server!

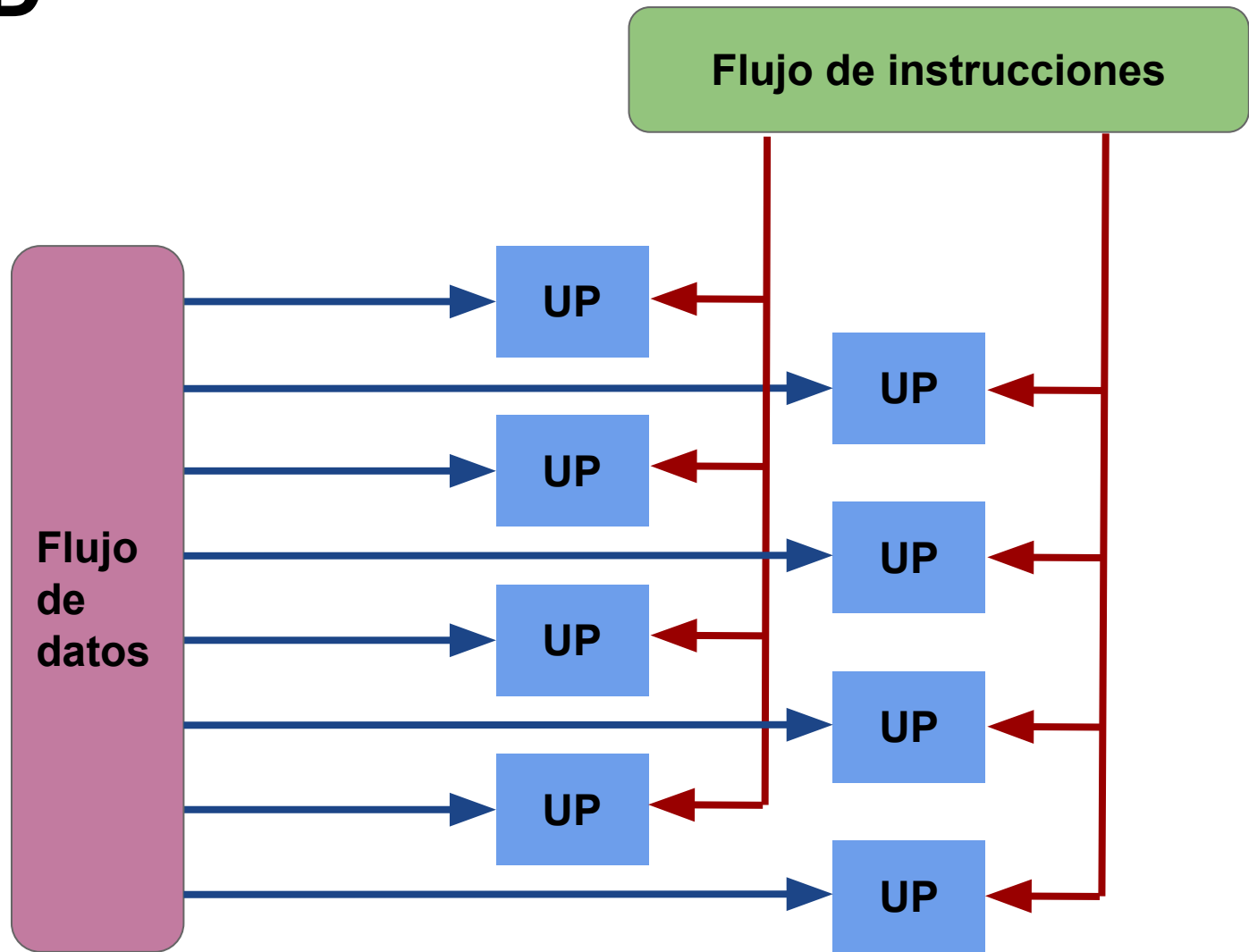
## Increase in workstation performance





**MIMD**

# MIMD



**UP** = Unidad de Procesamiento

# MIMD: sincronización

- Cada quien trabaja por su lado.
- Pero deben hacer trabajo colaborativo.
- Así que deben ponerse de acuerdo (sincronizarse) de alguna manera.

# Subclases

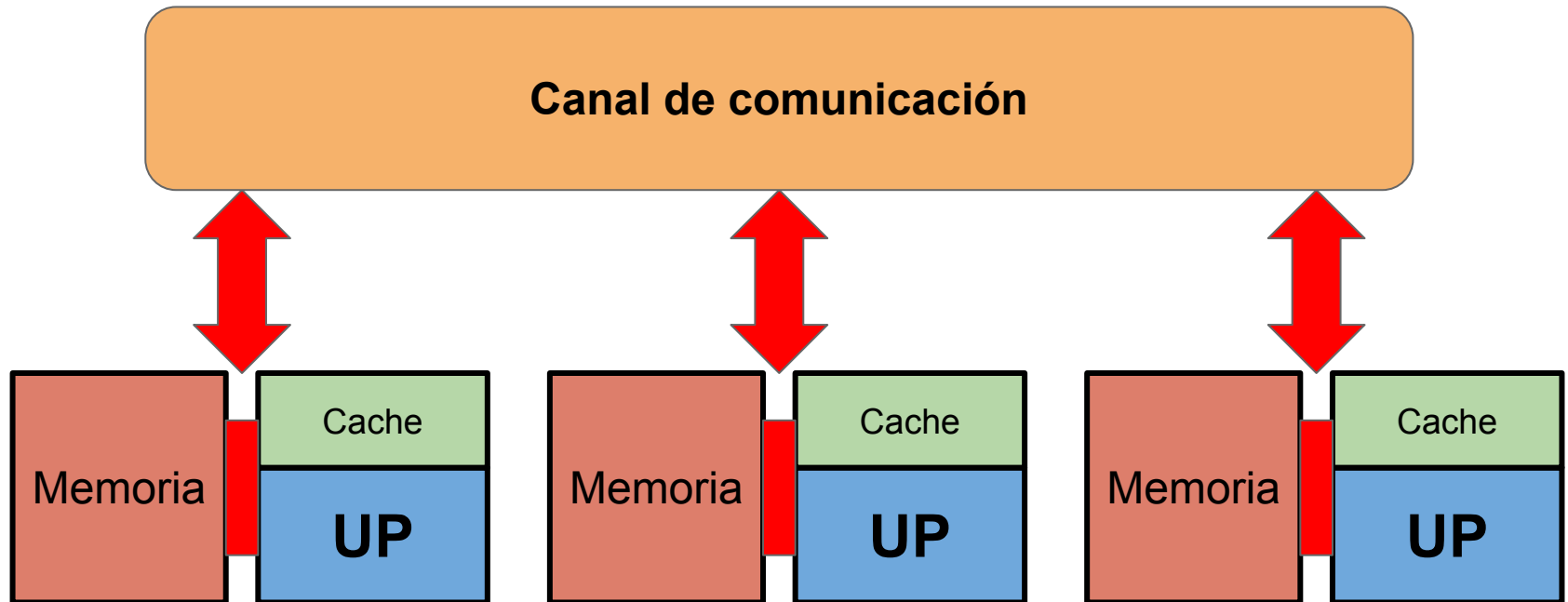
- Memoria compartida (*Shared Memory*).
  - Los procesadores se comunican a través de un mismo espacio de direcciones.
  - Multiprocesadores Fuertemente Acoplados (Tightly Coupled Multiprocessors).
- Memoria distribuida (*Distributed Memory*).
  - Los procesadores tienen espacios de direcciones ajenos.
  - Los procesadores se comunican mediante paso de mensajes (*Message Passing*).
  - Multiprocesadores Débilmente Acoplados (Loosely Coupled Multiprocessors).

# Memoria compartida

Dos casos: La memoria que comparten ¿está más cerca de unos procesadores que de otros?

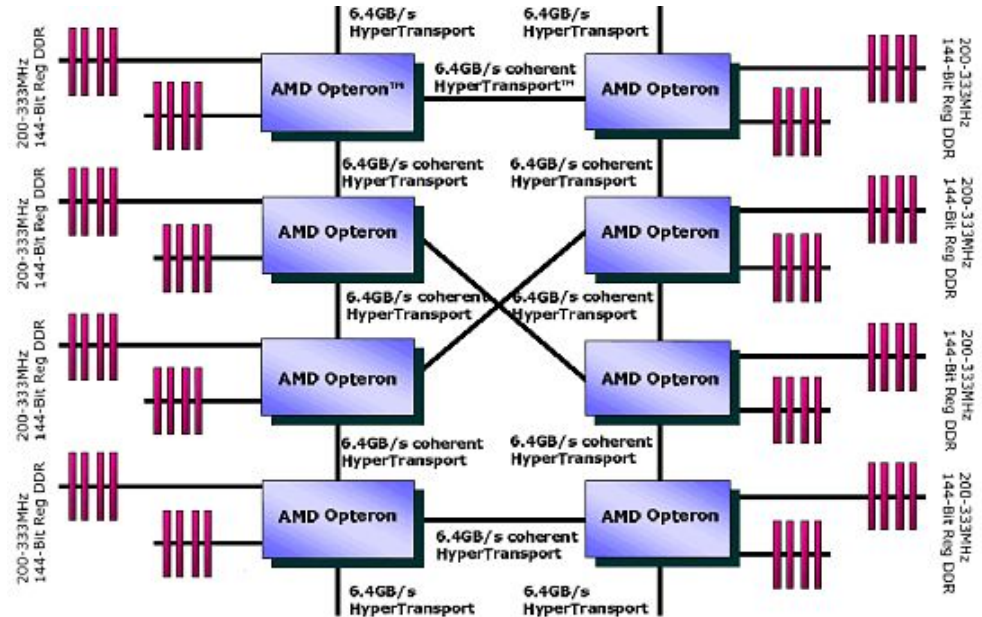
- Sí. Entonces acceder a algunas direcciones es más rápido que a otras. NUMA (*Non Uniform Memory Access*).
- No. Entonces acceder a cualquier dirección toma el mismo tiempo. UMA, en particular SMP (*Symmetric Multiprocessor*).

# NUMA

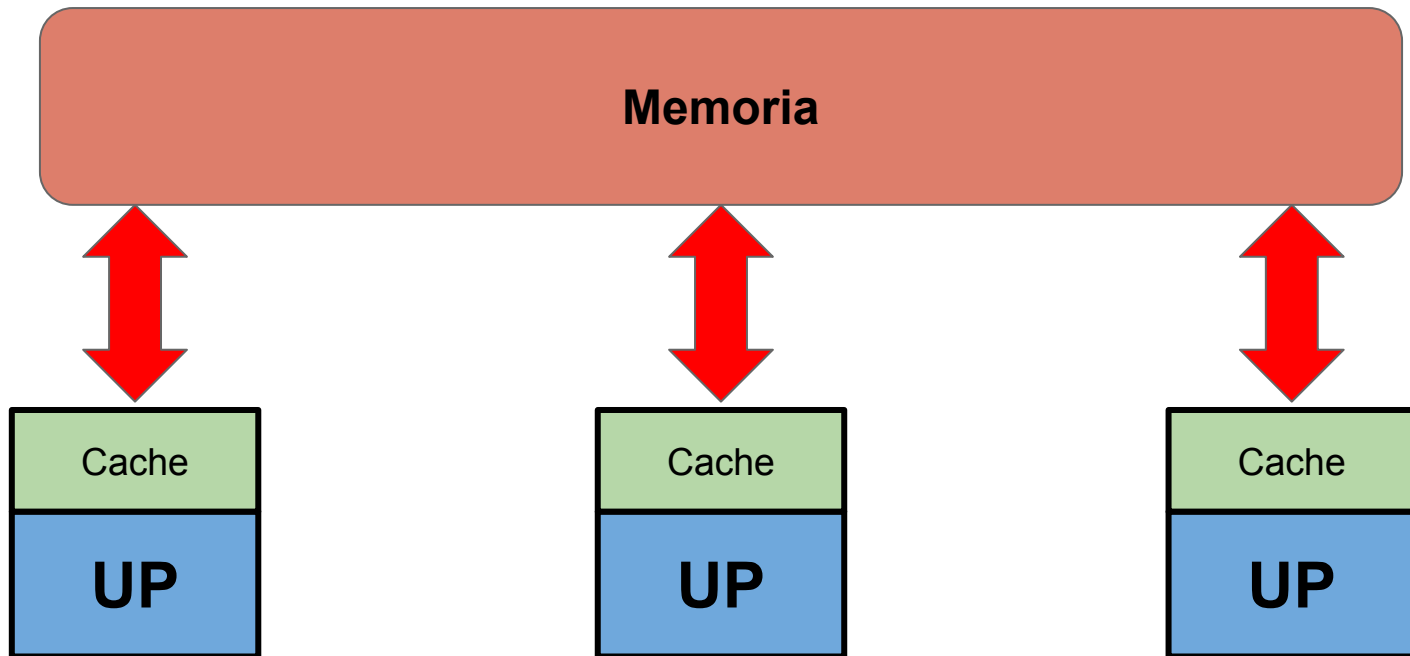


# NUMA

- Es como un cluster pero altamente acoplado (*tightly coupled*).
- AMD conectó al Opteron (2003) usando HyperTransport en una arquitectura NUMA.



# SMP





# Arquitecturas multinúcleo

- Son SMP. Cada núcleo tiene su propio cache, pero la memoria principal es compartida por todos.

# Motivación

# ¿Por qué?

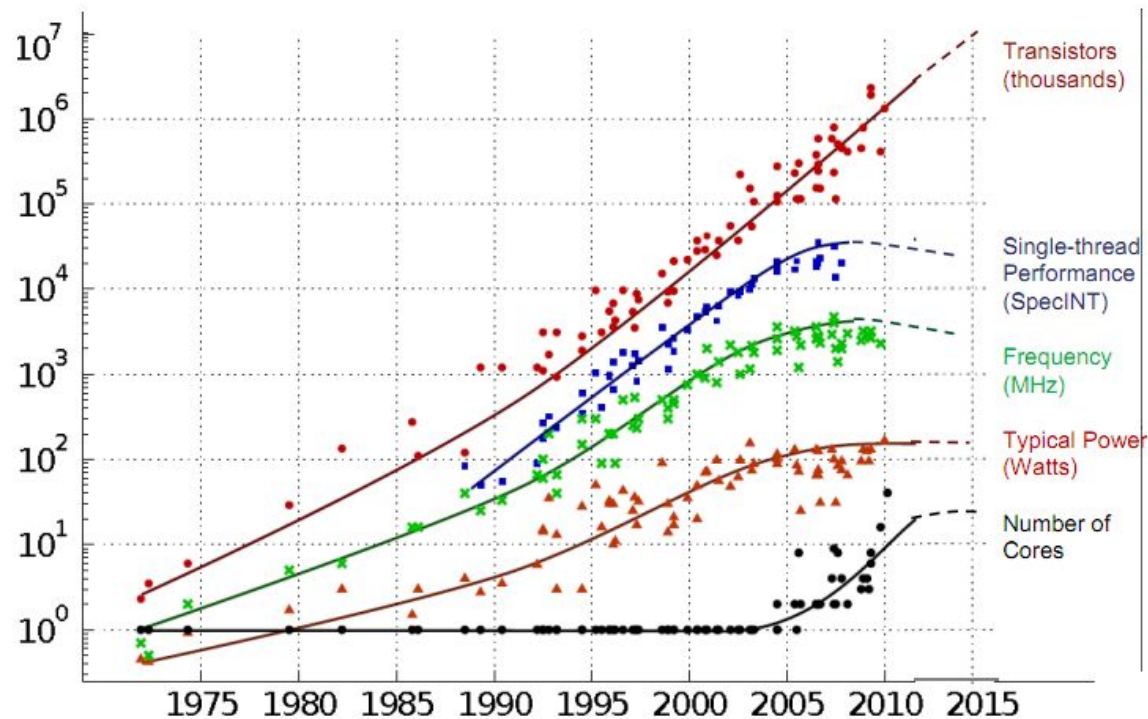
- Hasta 2004:
  - Ley de Moore, 2X transistores cada 18 meses. Traía aparejado el desempeño.
  - Técnicas de ILP: ejecución fuera de orden, predicción de salto, especulación.
  - Mayores y mejores jerarquías de memoria (más caches, mejores, en el chip).
  - Frecuencias de operación cada vez más altas.
  - Arquitecturas mononúcleo compatibles hacia atrás cada vez más poderosas.
- Mayor frecuencia de operación y pipelines muy profundos implican mucho calor.
- Ya no se puede obtener mayor ILP.

# Problemas

- *Prescott* (2004) fue un avance mediocre sobre *Northwood* (2002).
- El 7 de mayo de 2004 Intel anuncia la cancelación del proyecto *Tejas*, el nombre código del procesador que seguiría al Pentium 4 *Prescott*. Microarquitectura Netburst.

# ¡bamos bien...

## 35 YEARS OF MICROPROCESSOR TREND DATA



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten  
Dotted line extrapolations by C. Moore

# ¿Qué ocurrió?

- El consumo del procesador es:  $P = k f V^2$ .
- $P$  es el consumo en Watts,  $k$  es una constante (relacionada con la capacitancia),  $f$  es la frecuencia de operación y  $V$  es el voltaje.
- A mayor frecuencia, mayor consumo.
- A mayor consumo, mayor temperatura.

- Para mantener el desempeño aparejado con la ley de Moore, había que subir la frecuencia del procesador.
- Y entonces consume mucho.
- Y se calienta mucho.

Hasta que ya no se pudo.

# ¿Y entonces?

- Ya no es posible mayor ILP.
- Ya no es costeable mayor frecuencia de operación.
- Ya es muy difícil mantener una temperatura de operación.
  - Hay que pasar al nivel semántico siguiente:  
**thread.**
- Paralelismo a nivel de thread (TLP).



# Thread

- Es la secuencia mínima de instrucciones que tiene sentido manejar de manera independiente para ejecutarse concurrentemente con otras similares.
- Es un concepto jerárquicamente menor al de proceso.
- Un proceso puede tener varios threads. Los threads de un proceso comparten memoria y otros recursos (archivos abiertos, conexiones).
- Cada thread tiene su propio stack y su propio contexto (estado de ejecución).
- Cada thread recibe tiempo de procesamiento de parte del SO. El SO calendariza threads.
- Multithreading no significa que los threads se ejecutan en paralelo, pero así debe considerarse. El programa con varios hilos debe ser consistente si es así o no.

# Multinúcleo

- Para obtener el mejor desempeño en un paradigma multithreading se requiere hardware multinúcleo.
- Si hay que ejecutar threads en paralelo, entonces se requieren núcleos de procesamiento independientes. Cada núcleo ejecuta un thread a la vez.
- Intel declara en 2004, que enfocará sus esfuerzos en arquitecturas multinúcleo (AKA, Chip Multiprocessors o CMP).

# El problema

- Cada núcleo con su propio cache (al menos nivel L1).
- Los demás niveles de la jerarquía son compartidos.
- Eso es lo que hace posible la comunicación (sincronización) entre threads.
- ¿Cómo garantizar que sea consistente el estado de la memoria con lo que han hecho los procesos?
- ¿Cómo garantizar que esta consistencia se refleje también en los caches?

# Dos ámbitos

¿Qué ocurre cuando dos o más procesadores acceder de manera concurrente a ...

- diferentes localidades de memoria?
  - consistencia de memoria
- la misma localidad de memoria?
  - coherencia de caches

# **Consistencia de Memoria**

# Modelos de consistencia de memoria

Cuando hay más de un proceso en interacción con la memoria accediendo a variables compartidas se necesita establecer un contrato entre los procesos y la memoria.



Si los procesos garantizan cierto comportamiento, la memoria garantiza cómo se verán reflejados los accesos a las variables compartidas.

# Consistencia estricta

- Linealizable o atómica.
- Es el modelo más exigente.
- Cualquier lectura a la localidad de memoria retorna el valor almacenado por la última operación de escritura (antes de la lectura).
- Determinar cuál fue la escritura más reciente no siempre es posible.
- Supone la existencia de un tiempo global conocido por todos los procesadores.
- Supone que toda escritura y lectura se hagan instantánea y atómicamente.
- Por tanto es imposible.

# Estricta: Ejemplo y contraejemplo

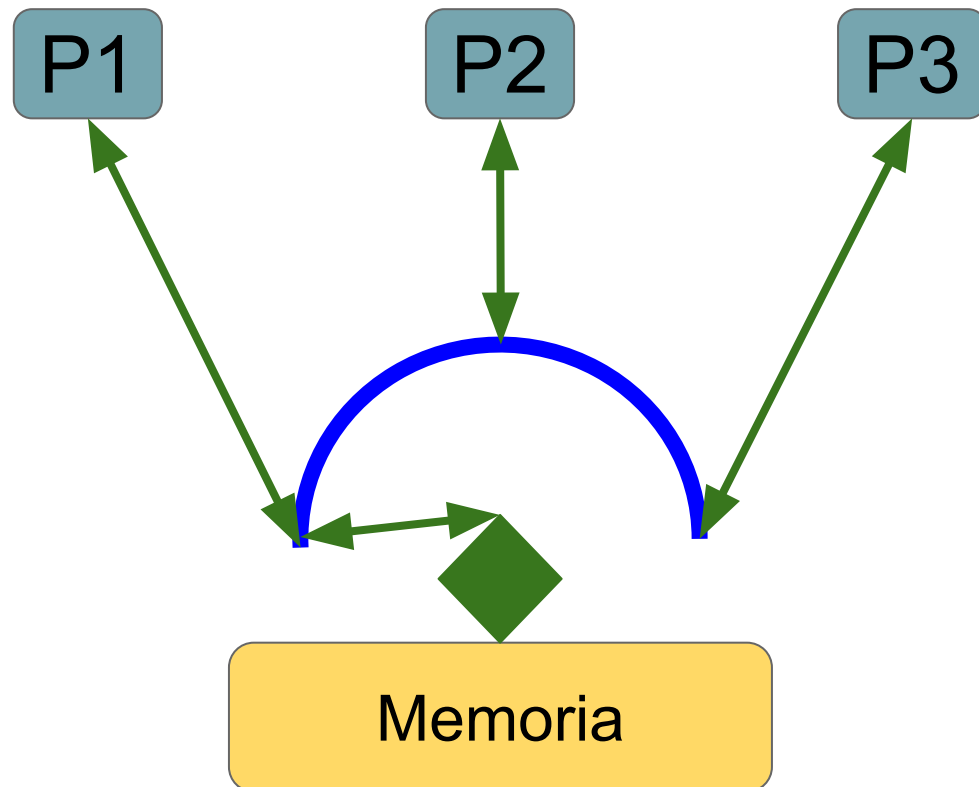
Inicialmente:  $[X] := 0$

<b>P1</b>	$W: a \rightarrow [X]$	
<b>P2</b>	$R: [X] \rightarrow a$	
<b>P1</b>	$W: a \rightarrow [X]$	
<b>P2</b>	$R: [X] \rightarrow 0$	



# Consistencia secuencial

- Para cada proceso: se mantiene el orden en el que aparecen las operaciones en él.
- Existe un cierto orden secuencial entre los procesos.
- Las actualizaciones de memoria compartida se ven en el mismo orden en todos los procesos.





# Parafraseando

Todas las escrituras y las lecturas de todas las localidades de memoria compartida deben ser ejecutadas de acuerdo con un cierto orden total que respeta el orden parcial definido por cada proceso.

# Secuencial: Ejemplo y contraejemplo

Inicialmente:  $[X] := 0$



<b>P1</b>	$W:a \rightarrow [X]$	
<b>P2</b>	$W:b \rightarrow [X]$	
<b>P3</b>	$R:[X] \rightarrow b \qquad R:[X] \rightarrow a$	
<b>P4</b>	$R:[X] \rightarrow b \qquad R:[X] \rightarrow a$	
<b>P1</b>	$W:a \rightarrow [X]$	
<b>P2</b>	$W:b \rightarrow [X]$	
<b>P3</b>	$R:[X] \rightarrow b \qquad R:[X] \rightarrow a$	
<b>P4</b>	$R:[X] \rightarrow a \qquad R:[X] \rightarrow b$	

# Consistencia causal

- Las actualizaciones se propagan de acuerdo con una relación de causalidad.
- Si dos operaciones de memoria tienen una relación de causalidad entonces todos los procesos ven que ocurren en el mismo orden.
- Las escrituras no relacionadas por causalidad pueden ser vistas en orden arbitrario por diferentes procesadores.
- Cuando un procesador hace un read y luego un write, aún en variables diferentes, el read tiene relación causal con el write y deben verse en el mismo orden en todo procesador. Porque el valor escrito por el write puede depender del leído por el read.
- Un write seguido de un read, sobre la misma variable, también tiene relación causal porque el read podría extraer el valor escrito por el write.
- Dos write realizados por el mismo procesador están relacionados causalmente en el orden en que aparecen en el programa.
- La relación de causalidad es transitiva: Si A está ordenado causalmente antes que B y este antes que C, entonces A antes que C.

# Causal: Ejemplo y contraejemplo

Inicialmente:  $[X] := 0$



P1	$W:a \rightarrow [X]$	
P2	$W:b \rightarrow [X]$	
P3	$R:[X] \rightarrow b \quad R:[X] \rightarrow a$	
P4	$R:[X] \rightarrow a \quad R:[X] \rightarrow b$	
P1	$W:a \rightarrow [X]$	
P2	$R:[X] \rightarrow a \quad W:b \rightarrow [X]$	
P3	$R:[X] \rightarrow b \quad R:[X] \rightarrow a$	
P4	$R:[X] \rightarrow a \quad R:[X] \rightarrow b$	

# Consistencia de procesador

- Todas las escrituras hechas por un procesador deben verse en el mismo orden en que son ejecutadas por él, por todos los demás.
- Las escrituras hechas por diferentes procesadores pueden verse en diferente orden por diferentes procesadores.
- Las escrituras hechas a la misma variable se ven siempre en el mismo orden en todos los procesadores.

# De procesador: Ejemplo y contraejemplo

Inicialmente:  $[X] := 0$

P1	W:a → [X]      W:b → [X]	
P2	R:[X] → a   R:[X] → b	
P3	R:[X] → a   R:[X] → b	
P1	W:a → [X]      W:b → [X]	
P2	R:[X] → a   R:[X] → b	
P3	R:[X] → b   R:[X] → a	

# Consistencia PRAM (Pipelined RAM)

- Todas las escrituras hechas por un procesador deben verse en el mismo orden en que son ejecutadas por él, por todos los demás.
- Las escrituras hechas por diferentes procesadores pueden verse en diferente orden por diferentes procesadores.



# **Coherencia de caches**

# Definición 1

Dos invariantes:

1. Único escritor múltiples lectores. Para toda localidad de memoria  $X$ , en todo instante de tiempo, o bien existe un único procesador que puede escribir en  $X$ , o bien algunos procesadores pueden leer de  $X$ .
2. Valor del dato almacenado. El valor almacenado en una localidad de memoria al inicio de un periodo de tiempo en el que es accedida, es el mismo que tenía al final del último periodo de tiempo en el que fue accedida para escritura.

# Definición 2

Dos invariantes:

1. El efecto de toda operación de escritura es visible por todos los procesadores eventualmente.
2. Las escrituras hechas a una misma localidad de memoria son serializadas (visibles en el mismo orden por todos los procesadores).

## **Definición 3: Similar a consistencia secuencial**

Todas las escrituras y las lecturas sobre una localidad de memoria compartida deben ser ejecutadas de acuerdo con un cierto orden total que respeta el orden parcial definido por cada proceso.

# Definición 4 (Hennessy y Patterson)

Tres invariantes:

1. Si el procesador P lee de la localidad X luego de haber escrito en ella, y ningún otro procesador escribió en X entre la escritura y la lectura de P, entonces P debe leer el mismo valor que escribió en X.
2. Si P1 lee de X, luego de un *tiempo suficiente* de que P2 escribiera en X; entonces P1 debe leer el valor que P2 escribió, si ningún otro procesador escribió en X entre la escritura de P2 y la lectura de P1.
3. Las escrituras sobre una misma localidad están serializadas. Si P1 escribe en X el valor A y luego P2 escribe en X el valor B, entonces ningún procesador puede leer A de X si ya leyó B.

# Claro que...

Si se usan caches write through no hay problema de coherencia.

# Técnicas para garantizar coherencia

- Basados en directorio. Entidad central en la que se registran los accesos a memoria y el estado de los bloques compartidos.
- Snooping. Cada cache “escucha” de un canal de direcciones compartido.  
Típicamente el canal entre los caches privados y el compartido.

# Directorio

- Parte del controlador de acceso a memoria se dedica a registrar todos los accesos: dirección solicitada e ID del procesador que la pidió.
- Si un procesador requiere una escritura en un dato, solicita al controlador acceso exclusivo al bloque que lo contiene.
- El controlador emite una señal que invalida cualquier copia de ese dato en los otros caches.
- Cuando un procesador requiere leer un bloque que ya ha sido escrito por otro, el controlador le indica a este que haga write back.
- SGI Origin 2000 (1996).



# Snooping

Los procesadores (sus controladores de cache) escuchan un canal compartido con el controlador (central) de memoria.

- Write invalidate. Una escritura hace que todos los caches que contienen la dirección accedida la marquen como inválida.
- Write update (snarfing). Escuchan el canal de datos para actualizar el valor en el cache. Se debe hacer un broadcast del valor nuevo y la dirección.

# Snooping Vs. Directorio

- El directorio puede ser un cuello de botella, pero se degrada lentamente el desempeño.
- En el esquema de snooping, como hay que difundir toda dirección y a veces también el dato, el tráfico es intenso y no escala muy bien con el número de procesadores. Pero son más simples.

# Invalidación Vs. actualización

- En el de invalidación sólo se necesita la dirección.
- En el de actualización se necesita la dirección y el dato, consume mayor ancho de banda del canal.
- El de actualización complica la implementación del modelo de consistencia porque es difícil mantener la atomicidad de una operación si hay que actualizar muchas copias.

# Por lo general...

Nos vamos a encontrar.

- Esquemas de snooping.
- Con invalidación.

El esquema luego se usa para implementar un protocolo de coherencia de cache que garantice que, desde los caches, se lleva a cabo el contrato de consistencia de memoria.

# MESI

Protocolo más popular.

Cada bloque de cache, junto con su tag, tiene un indicador de estado. El estado de un bloque cambia en función de lo que el cache (procesador) haga y de lo que los otros caches hagan.

# M: Modified

El bloque está solamente en este cache y ha sido escrito (está sucio, dirty).

- Cuando se escucha que otro procesador lo requiere para leer, se hace write-back y el estado cambia a compartido (Shared).
- Mientras el bloque se siga leyendo o escribiendo por el mismo procesador el estado se mantiene.

# E: Exclusive

Este bloque sólo está en este cache y no ha sido escrito por él (está limpio, clean) dado que está acorde con memoria principal.

- Si se escucha que alguien lo pide para leer se pasa a compartido (shared).
- Si se escucha que alguien lo pide para escribir se pasa a inválido.
- Mientras se siga pidiendo para leer en este procesador se mantiene el estado.

# S: Shared

Este bloque está en este cache y probablemente en algún otro. Está limpio, empata con memoria principal.

- Mientras se siga leyendo (por este procesador u otro) se mantiene el estado.
- Si el procesador lo escribe, pasa a modificado (Modified).
- Si alguien más solicita escribir se pasa a inválido.



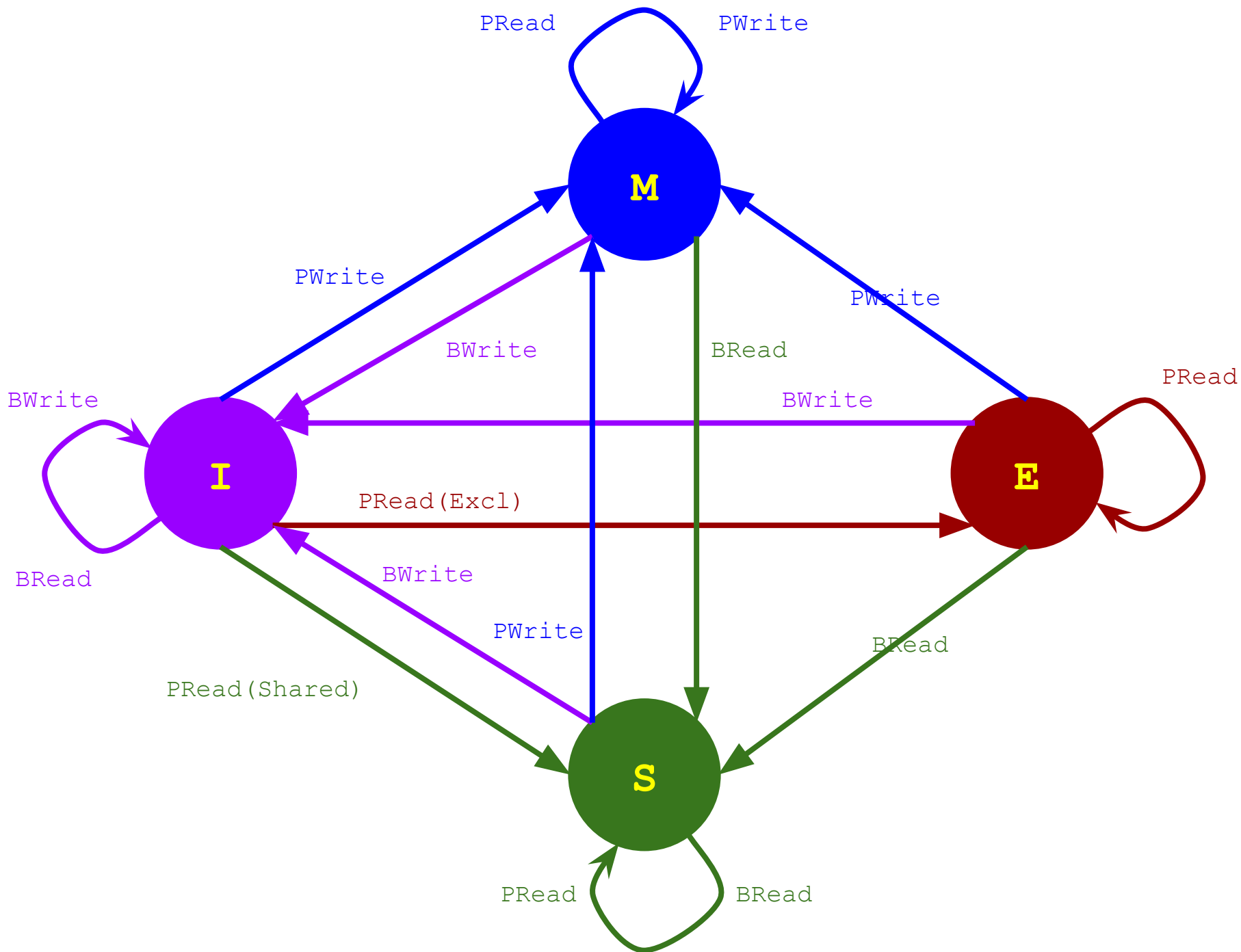
# I: Invalid

Este bloque ya no contiene un dato válido.

- Si hay una petición de lectura por el mismo procesador, hay miss, se lee el dato de nuevo si otro cache tiene el dato: Shared, si no Exclusive.
- Si la petición es de escritura, hay miss, se lee el dato de nuevo y se pasa a Modified.

# Tabla de transición

Estado actual	Evento	Acción	Estado siguiente
E	ReadX	-	S
E	WriteX	-	I
E	ReadL	Hit, Entregar	E
E	WriteL	Hit, Escribir	M
I	ReadX	-	I
I	WriteX	-	I
I	WriteL	Miss, Escribir	M
I	ReadL	Miss, Fetch	S
M	ReadX	-	S
M	WriteX	-	I
M	ReadL	Hit, Entregar	M
M	WriteL	Hit, Escribir	M
S	ReadX	-	S
S	WriteX	-	I
S	ReadL	Hit, Entregar	S
S	WriteL	Hit, Escribir	M
I	ReadLExcl	Miss, Fetch	E



# Clusters

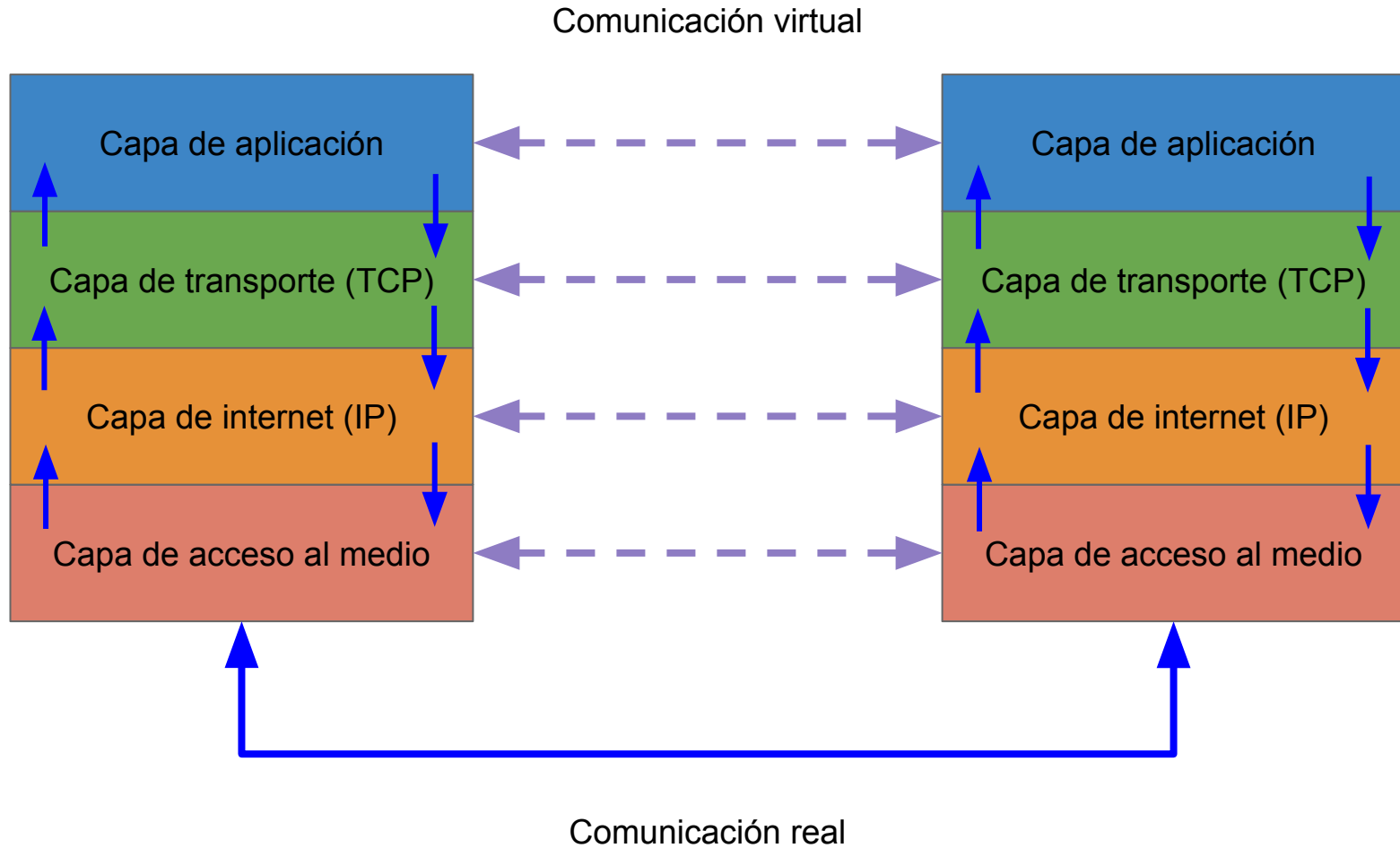
# Paso de mensajes

- Es decir: una multicomputadora.
- Cada procesador tiene su propio espacio de direcciones.
- **Memoria distribuida.**
- Los mensajes sirven para coordinar trabajo, informar resultados parciales: send, receive.
- Se usa una red de interconexión.

# Tipos de cluster

- Clusters de alto desempeño (High Performance Computing Clusters). El objetivo es lograr un sistema de cómputo con un muy breve tiempo de respuesta.
- Clusters de alto rendimiento (High Throughput Computing Clusters). El objetivo es lograr un sistema de cómputo que realice muchas tareas por unidad de tiempo.
- Clusters de alta disponibilidad (High Availability Computing Clusters). El objetivo es tener un sistema de cómputo permanentemente accesible, posee redundancia y tolerancia a fallos.

# Latencia



# Red

Red	Ancho de banda	Latencia aproximada	Ref
Fast Ethernet	100 Mbps	80 $\mu$ s (min)	[1]
Gigabit Ethernet	1 Gbps	50 a 80 $\mu$ s	[2]
Myrinet	1.2 Gbps	15 a 40 $\mu$ s	[2]
Myrinet 2000	2 Gbps	2.6 a 3.2 $\mu$ s	[3]
Myri-10G	10 Gbps	2 $\mu$ s	[3]
InfiniBand	14 Gbps	0.7 $\mu$ s	[4]

[1] Dietz, Hank, *Linux Parallel Processing HOWTO*, V 2.0, 2004.

[2] Ali, Mustafa I, "Performance evaluation of Gigabit Ethernet and Myrinet for System-Area-Networks", *Computer Networks*, 2005.

[3] Myricom (<http://www.myricom.com/myrinet/overview/>)

[4] Mellanox, InfiniBand fact sheet, 2015,



# Middleware y SO

- Varios sistemas adaptados para trabajar en paralelo y virtualizar.
- Linux, Windows Server, Solaris, FreeBSD, Mac OS X.
- Middleware que facilita la interacción del SO y las aplicaciones: OpenMOSIX
  - Migrar procesos, balancear carga, ofrecer una única interfaz, priorizar.

# Paso de mensajes

- Envío y recepción de mensajes.
- Con bloqueo (esperar hasta que haya mensaje).
- Sin bloqueo (esperar un tiempo prudente o nada).

# PVM

- Parallel Virtual Machine. 1989.
- Hacer que un conjunto de máquinas Unix se vean como una sola máquina virtual.
- Ambiente de ejecución + biblioteca de paso de mensajes + administración de tareas.
- Primitivas de bloqueo pero con variantes con time-out.
- En cada máquina se ejecuta un *daemon* que maneja la conexión con los otros.
- Se pueden crear procesos.

# MPI

- Message Passing Interface. 1994.
- Sistema estandarizado de paso de mensajes creado por un conjunto de académicos e investigadores de la industria.
- Protocolo y servicios de comunicación de bloqueo y no-bloqueo.
- No crea procesos, agrupa los ya existentes, les da contexto y les asocia un administrador de comunicación.
- Se puede crear una topología virtual de conectividad.
- Define tipos de datos.