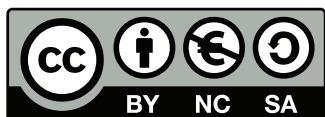


Arquitectura de Computadores

Paula Villanueva Núñez
Doble Grado de Informática y Matemáticas
Universidad de Granada



Este libro se distribuye bajo una licencia CC BY-NC-SA 4.0.

Eres libre de distribuir y adaptar el material siempre que reconozcas a los autores originales del documento, no lo utilices para fines comerciales y lo distribuyas bajo la misma licencia.

creativecommons.org/licenses/by-nc-sa/4.0/

Arquitectura de Computadores

Paula Villanueva Núñez
Doble Grado de Informática y Matemáticas
Universidad de Granada

Índice

1. Tema 1. Arquitecturas paralelas: clasificación y prestaciones	4
1.1. Lección 1. Clasificación del paralelismo implícito en una aplicación	4
1.1.1. 1.1 Objetivos	4
1.1.2. 1.2 Criterios de clasificaciones del paralelismo implícito en una aplicación.	4
1.1.3. 1.3 Dependencias de datos.	5
1.1.4. 1.4 Paralelismo implícito (nivel de detección), explícito y arquitecturas paralelas. . .	6
1.1.5. 1.5 Detección, utilización, implementación y extracción del paralelismo.	7
1.2. Lección 2. Clasificación de arquitecturas paralelas.	8
1.2.1. 2.1 Objetivos	8
1.2.2. 2.2 Computación paralela y computación distribuida.	9
1.2.3. 2.3 Clasificaciones de arquitecturas y sistemas paralelos.	9
1.2.4. 2.4 Nota histórica	23
1.3. Lección 3. Evaluación de prestaciones.	24
1.3.1. 3.1 Objetivos.	24
1.3.2. 3.2 Medidas usuales para evaluar prestaciones.	24
1.3.3. 3.3 Conjunto de programas de prueba (Benchmark).	28
1.3.4. 3.3.1 LINPACK.	29
1.3.5. 3.4 Ganancia en prestaciones.	30
2. Tema 2. Programación paralela	35
2.1. Lección 4. Herramientas, estilos y estructuras en programación paralela.	35
2.1.1. Objetivos.	35
2.1.2. 4.1 Problemas que plantea la programación paralela al programador. Punto de partida.	35
2.1.3. 4.2 Herramientas para obtener código paralelo.	37
2.1.4. 4.3 Estilos/paradigmas de programación paralela.	43
2.1.5. 4.4 Estructuras típicas de códigos paralelos.	44
2.2. Lección 5. Proceso de paralelización.	48
2.2.1. 5.1 Objetivos.	48

2.2.2. 5.2 Proceso de paralelización.	48
2.3. Lección 6. Evaluación de prestaciones en procesamiento paralelo.	57
2.3.1. Objetivos.	57
2.3.2. 6.1 Ganancia de prestaciones y escalabilidad.	57
2.3.3. 6.2 Ley de Amdahl.	60
2.3.4. 6.3 Ganancia escalable.	61
3. Tema 3. Arquitecturas con paralelismo a nivel de thread (TLP)	63
3.1. Lección 7. Arquitecturas TLP.	63
3.1.1. Objetivos.	63
3.1.2. 7.1 Clasificación y estructura de arquitecturas con TLP explícito y una instancia del SO.	63
3.1.3. 7.2 Multiprocesadores.	63
3.1.4. 7.3 Multicores.	66
3.1.5. 7.4 Cores Multithread.	67
3.1.6. 7.5 Hardware y arquitecturas TLP en un chip.	71
3.2. Lección 8. Coherencia del sistema de memoria.	72
3.2.1. Objetivos.	72
3.2.2. 8.1 Sistema de memoria en multiprocesadores.	72
3.2.3. 8.2 Concepto de coherencia en el sistema de memoria: situaciones de incoherencia y requisitos para evitar problemas en estos casos.	73
3.2.4. 8.3 Protocolos de mantenimiento de coherencia: clasificación y diseño.	81
3.2.5. 8.4 Protocolo MSI de espionaje.	81
3.2.6. 8.5 Protocolo MESI de espionaje.	87
3.2.7. 8.6 Protocolo MSI basado en directorios con o sin difusión.	89
3.3. Lección 9. Consistencia del sistema de memoria.	94
3.3.1. Objetivos.	94
3.3.2. 9.1 Concepto de consistencia de memoria.	94
3.3.3. 9.2 Consistencia secuencial (SC).	94
3.3.4. 9.3 Modelos de consistencia relajados.	96
3.4. Lección 10. Sincronización.	101
3.4.1. Objetivos.	101

3.4.2.	10.1 Comunicación en multiprocesadores y necesidad de usar código de sincronización.	101
3.4.3.	10.2 Soporte software y hardware para sincronización.	102
3.4.4.	10.3 Cerrojos.	103
3.4.5.	10.3.1 Cerrojos simples.	104
3.4.6.	10.3.2 Cerrojos con etiqueta.	104
3.4.7.	10.3.3 Barreras.	105
3.4.8.	10.3.4 Apoyo hardware a primitivas software.	106
4.	Bibliografía	111

1 Tema 1. Arquitecturas paralelas: clasificación y prestaciones

1.1 Lección 1. Clasificación del parallelismo implícito en una aplicación

1.1.1 1.1 Objetivos

- Clasificaciones del parallelismo implícito en una aplicación. Distinguir entre parallelismo de tareas y de datos.
- Distinguir entre dependencias RAW, WAW, WAR.
- Distinguir entre thread y proceso.
- Relacionar el parallelismo implícito en una aplicación con el nivel en el que se hace explícito para que se pueda utilizar (instrucción, thread, proceso) y con las arquitecturas paralelas que lo aprovechan.

1.1.2 1.2 Criterios de clasificaciones del parallelismo implícito en una aplicación.

- **Parallelismo funcional.**

- **Nivel de funciones.** Las funciones llamadas en un programa se pueden ejecutar en paralelo, siempre que no haya entre ellas dependencias inevitables, como dependencias de datos verdaderas (lectura después de escritura).
- **Nivel de bucle (bloques).** Se pueden ejecutar en paralelo las iteraciones de un bucle, siempre que se eliminen los problemas derivados de dependencias verdaderas. Para detectar dependencias habrá que analizar las entradas y las salidas de las iteraciones del bucle.
- **Nivel de operaciones.** Las operaciones independientes se pueden ejecutar en paralelo. En los procesadores de propósito específico y en los de propósito general podemos encontrar instrucciones compuestas de varias operaciones que se aplican en secuencia al mismo flujo de datos de entrada. Se pueden usar instrucciones compuestas, que van a evitar las penalizaciones por dependencias verdaderas.

- **Parallelismo de datos** (*data parallelism o DLP-Data Level Par.*). Se encuentra implícito en las operaciones con estructuras de datos (vectores y matrices). Se puede extraer de la representación matemática de la aplicación. Las operaciones vectoriales y matriciales engloban operaciones que se pueden realizar en paralelo. Por lo que el parallelismo de datos está relacionado con el parallelismo a nivel de bucle.
- **Parallelismo de tareas** (*task parallelism o TLP-Task Level Par.*). Se encuentra extrayendo la estructura lógica de funciones de una aplicación. Los bloques son funciones y se puede encontrar parallelismo entre las funciones.
- **Granularidad.** El grano más pequeño (*grano fino*) se asocia al parallelismo entre operaciones o instrucciones, el *grano medio* se asocia a los bloques funcionales lógicos y el *grano grueso* se asocia al parallelismo entre programas.

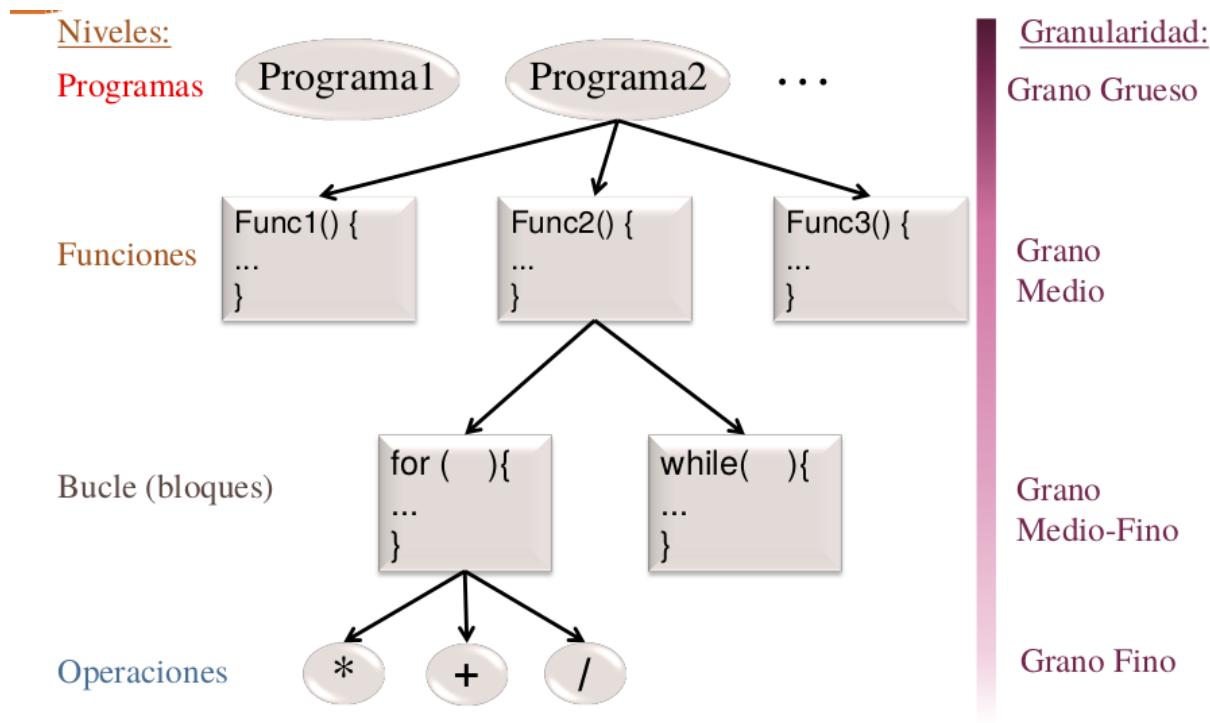


Figura 1:

1.1.3 1.3 Dependencias de datos.

Para que un bloque de código B_2 presente dependencia de datos con respecto a B_1 , deben hacer referencia a una misma posición de memoria M (variable) y B_1 aparece en la secuencia de código antes que B_2 .

Tipos de dependencias de datos (de B_2 respecto a B_1):

- **RAW** (*Read After Write*) o dependencia verdadera.
- **WAW** (*Write After Write*) o dependencia de salida.
- **WAR** (*Write After Read*) o antidependencia.

```

1 ...
2 a = b + c
3 ... //código que no usa a
4 d = a + c
5 ...

```

```

1 ...
2 a = b + c
3 ... //se lee a
4 a = d + e
5 ... //se lee a

```

```

1 ...
2 b = a + 1

```

```

3 ...
4 a = d + e
5 ... //se lee a

```

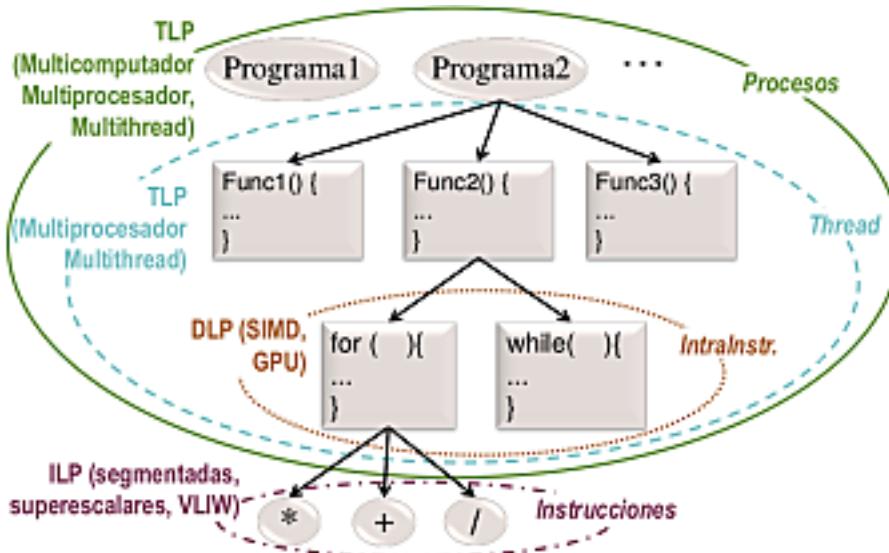
1.1.4 1.4 Parallelismo implícito (nivel de detección), explícito y arquitecturas paralelas.

El paralelismo entre **programas** se utiliza a nivel de procesos. Cuando se ejecuta un programa, se crea el proceso asociado al programa.

El paralelismo entre **funciones** se puede extraer para utilizarlo a nivel de procesos o de hebras.

El paralelismo dentro de un **bucle** se puede extraer a nivel de procesos o de hebras. Se puede aumentar la granularidad asociando un mayor número de iteraciones del ciclo a cada unidad a ejecutar en paralelo. Se puede hacer explícito dentro de una instrucción vectorial para que sea aprovechado por arquitecturas SIMD o vectoriales.

El paralelismo entre **operaciones** se puede aprovechar en arquitecturas con paralelismo a nivel de instrucción (ILP) ejecutando en paralelo las instrucciones asociadas a estas operaciones independientes.



1.4.1 Nivel de paralelismo explícito.

1.4.1.1 Unidades en ejecución en un computador.

- **Instrucciones**. La unidad de control de un core o procesador gestiona la ejecución de instrucciones por la unidad de procesamiento.
- **Thread o light process**. Es la menor unidad de ejecución que gestiona el SO. Menor secuencia de instrucciones que se pueden ejecutar en paralelo o concurrentemente.
- **Proceso o process**. Mayor unidad de ejecución que gestiona el SO. Un proceso consta de uno o varios thread.

1.4.1.2 Threads versus procesos.

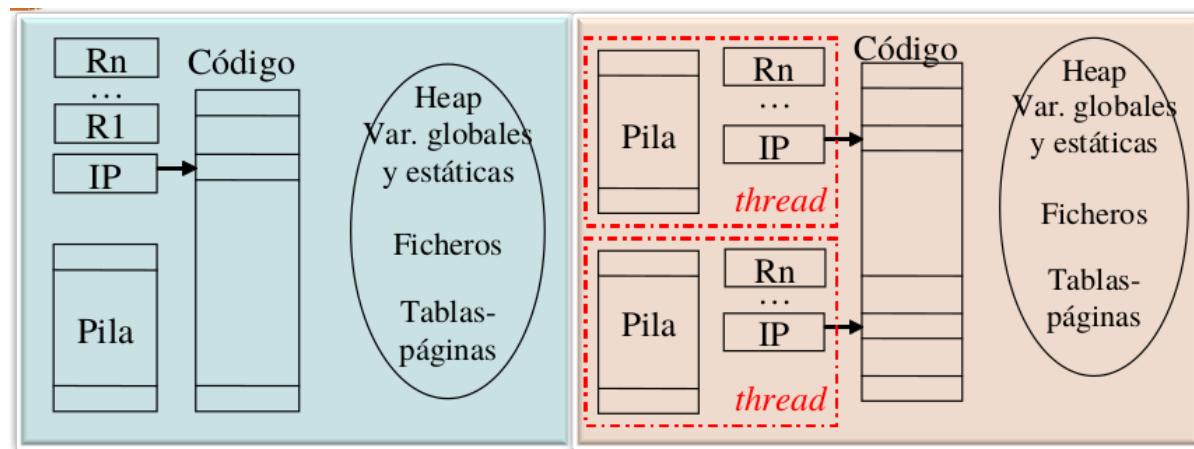
El hardware gestiona la ejecución de las instrucciones. A nivel superior, el SO se encarga de gestionar la ejecución de unidades de mayor granularidad, procesos y hebras. Cada proceso en ejecución tiene su propia asignación de memoria. Los SO **multihebra** permiten que un proceso se componga de una o varias hebras (hilos). Una **hebra** tiene su propia pila y contenido de registros, entre ellos el puntero de pila y el IP (Puntero de Instrucciones) que almacena la dirección de la siguiente instrucción a ejecutar de la hebra, pero comparte el código, las variables globales y otros recursos con las hebras del mismo proceso. Por lo que las hebras se pueden crear y destruir en menor tiempo que los procesos, y la comunicación (se usa la memoria que comparten), sincronización y conmutación entre hebras de un proceso es más rápida que entre procesos. Luego las hebras tienen menor granularidad que los procesos.

Un **proceso** comprende el código del programa y todo lo que hace falta para su ejecución:

- Datos en pila, segmentos (variables globales y estáticas) y en heap (BP1).
- Contenido de los registros.
- Tabla de páginas.
- Tabla de ficheros abiertos.

Para comunicar procesos hay que usar llamadas al SO.

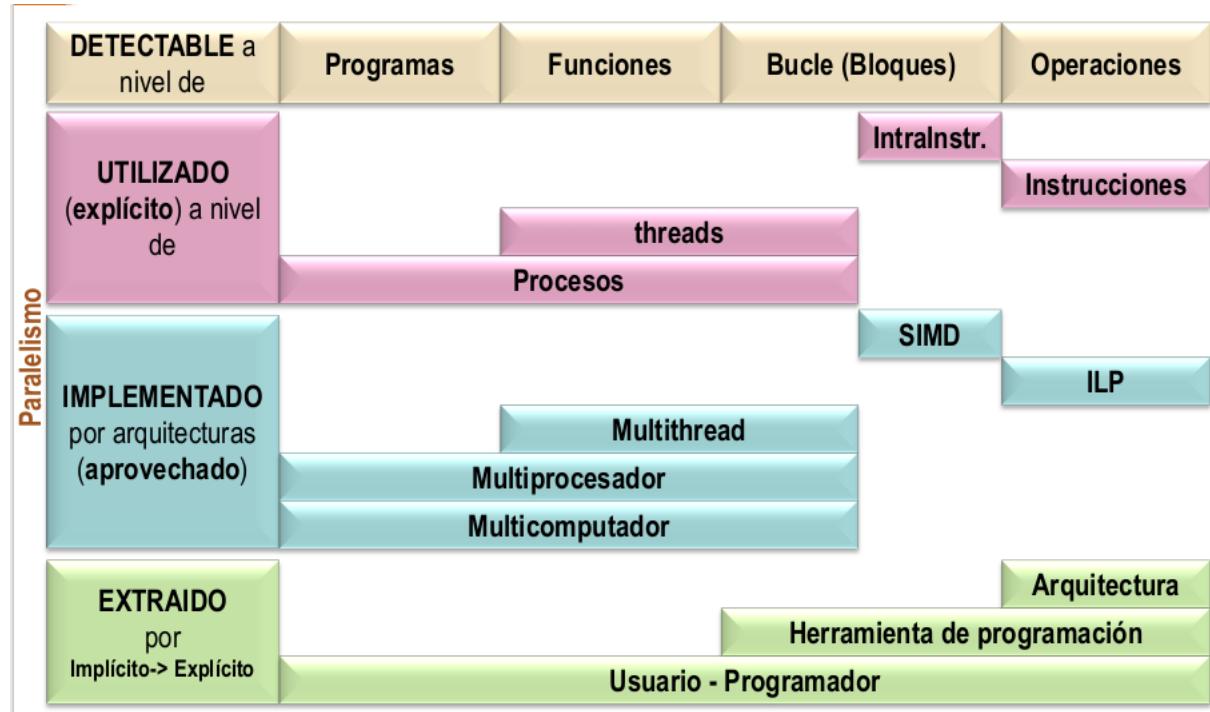
El paralelismo implícito en el código de una aplicación se puede hacer explícito a nivel de instrucciones, de hebras o de procesos.



1.1.5 1.5 Detección, utilización, implementación y extracción del paralelismo.

En los procesadores ILP superescalares o segmentados la arquitectura extrae paralelismo. Para ello, eliminan dependencias de datos falsas entre instrucciones y evitan problemas debidos a dependencias de datos, de control y de recursos. La arquitectura extrae paralelismo implícito en las entradas en tiempo de ejecución (dinámicamente). El grado de paralelismo de las instrucciones aprovechado se puede incrementar con ayuda del compilador y del programador. Podemos definir el grado de paralelismo de un conjunto de entradas a un sistema como el máximo número de entradas del conjunto que se puede ejecutar en paralelo. Para los procesadores las entradas son instrucciones. En las arquitecturas ILP VLIW el paralelismo que se va a aprovechar está ya explícito en las entradas. Las instrucciones que se van a

ejecutar en paralelo se captan juntas de memoria. El análisis de dependencias entre instrucciones en este caso es estático.



Hay compiladores que extraen el paralelismo de datos implícito a nivel de bucle. Algunos compiladores lo hacen explícito a nivel de hebra, y otros dentro de una instrucción para que se pueda aprovechar en arquitecturas SIMD o vectoriales. El usuario, como programador, puede extraer el paralelismo implícito en un bucle o entre funciones definiendo hebras y/o procesos. La distribución de las tareas independientes entre hebras o entre procesos dependerán de

- la granularidad de las unidades de código independientes,
- la posibilidad que ofrezca la herramienta para programación paralela disponible de definir hebras o procesos,
- la arquitectura disponible para aprovechar el paralelismo,
- el SO disponible.

Por último, los usuarios del sistema al ejecutar programas están creando procesos que se pueden ejecutar en el sistema concurrentemente o en paralelo.

1.2 Lección 2. Clasificación de arquitecturas paralelas.

1.2.1 2.1 Objetivos

- Distinguir entre procesamiento o computación paralela y distribuida.
- Clasificar los computadores según segmento del mercado.
- Distinguir entre las diferentes clases de arquitecturas de la clasificación de Flynn.
- Diferenciar un multiprocesador de un multicamputador.

- Distinguir entre NUMA y SMP.
- Distinguir entre arquitecturas DLP, ILP, TLP.
- Distinguir entre arquitecturas TLP con una instancia de SO y TLP con varias instancias de SO.

1.2.2 2.2 Computación paralela y computación distribuida.

- **Computación paralela.** Estudia los aspectos hardware y software relacionados con el desarrollo y ejecución de aplicaciones en un sistema de cómputo compuesto por múltiples cores/procesadores/computadores que es visto externamente como una unidad autónoma (multicores, multiprocesadores, multicomputadores, cluster).
- **Computación distribuida.** Estudia los aspectos hardware y software relacionados con el desarrollo y ejecución de aplicaciones en un sistema distribuido; es decir, en una colección de recursos autónomos (PC, servidores -de datos, software, . . . -, supercomputadores. . .) situados en distintas localizaciones físicas.
 - **Computación distribuida baja escala.** Estudia os aspectos relacionados con el desarrollo y ejecución de aplicaciones en una colección de recursos autónomos de un dominio administrativo situados en distintas localizaciones físicas conectados a través de infraestructura de red local.
 - **Computación distribuida gran escala.**
 - **Computación grid.** Estudia los aspectos relacionados con el desarrollo y ejecución de aplicaciones en una colección de recursos autónomos de múltiples dominios administrativos geográficamente distribuidos conectados con infraestructura de telecomunicaciones.
 - **Computación cloud.** Comprende los aspectos relacionados con el desarrollo y ejecución de aplicaciones en un sistema cloud. El sistema cloud ofrece servicios de infraestructura, plataforma y/o software, por los que se paga cuando se necesitan (pay-per-use) y a los que se accede típicamente a través de una interfaz (web) de auto-servicio. El sistema cloud consta de recursos virtuales que son una abstracción de los recursos físicos, parecen ilimitados en número y capacidad y son reclutados/liberados de forma inmediata sin interacción con el proveedor, soportan el acceso de múltiples clientes (multitenant) y están conectados con métodos estándar independientes de la plataforma de acceso.

1.2.3 2.3 Clasificaciones de arquitecturas y sistemas paralelos.

2.3.1 Criterios de clasificación de computadores

- Comercial. Segmento del mercado: embebidos, servidores gama baja. . .

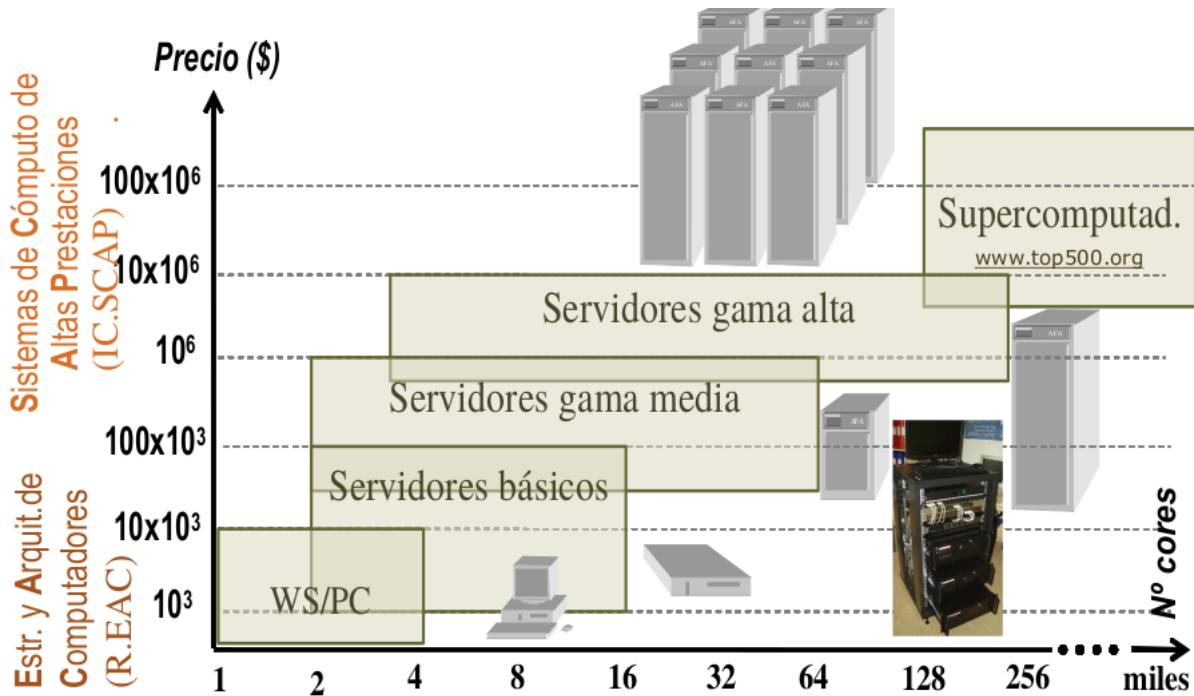


- Educación, investigación (también usados por fabricantes y vendedores):
 - Flujos de control y flujos de datos: clasificación de Flynn.
 - Sistemas de memoria.
 - Flujos de control (propuesta de clasificación de arquitecturas con múltiples flujos de control).
 - Nivel del paralelismo aprovechado (propuesta de clasificación).

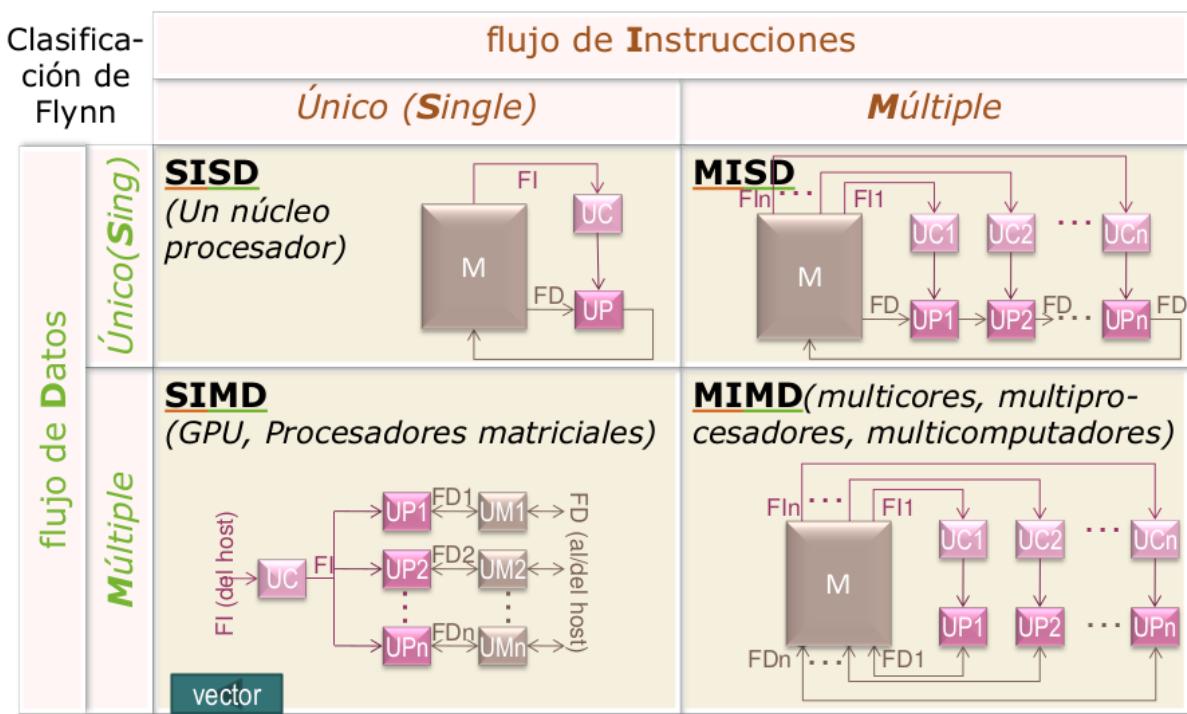
2.3.2 Clasificación de computadores según segmento

- **Externo** (*desktop, laptop, server, cluster...*) - R.EAC, IC.SCAP. Para todo tipo de aplicaciones:
 - Oficina, entretenimiento...
 - Procesamiento de transacciones o OLTP, sistemas de soporte de decisiones o DSS, e-commerce...
 - Científicas (medicina, biología, predicción del tiempo...) y animación (películas animadas, efectos especiales...).
- **Empotrado** (oculto) - IC.SCAE. Aplicaciones de propósito específico (videojuegos, teléfonos, coches, electrodomésticos...). Las restricciones típicas son: consumo de potencia, precio, tamaño reducido, tiempo real...

2.3.3 Clasificación de computadores externos según segmento del mercado



2.3.4 Clasificación de Flynn de arquitecturas (flujo instrucciones / flujo de datos)

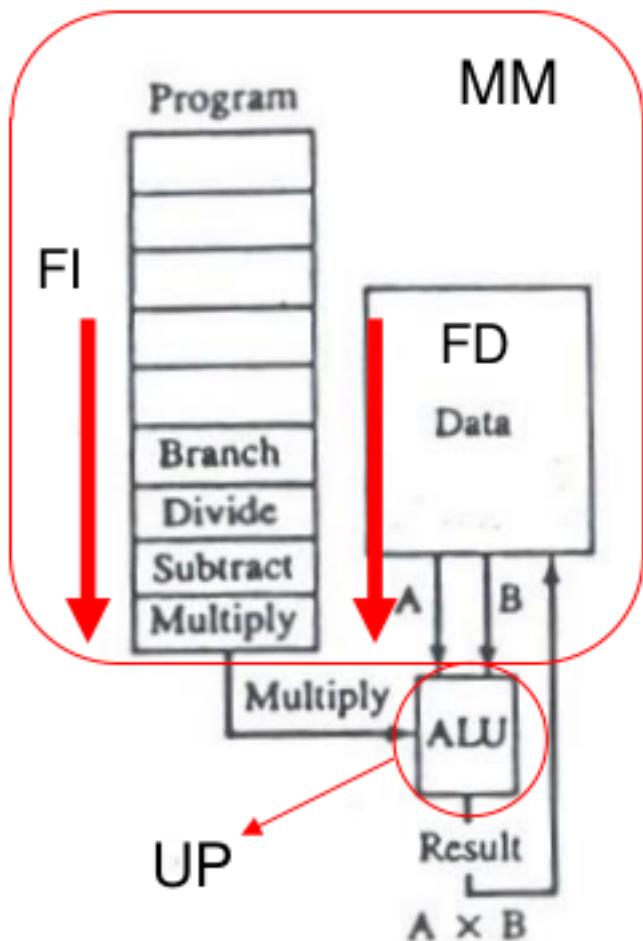


- **Computadores SISD.** Un único flujo de instrucciones (SI) procesa operandos y genera resultados, definiendo un único flujo de datos (SD).

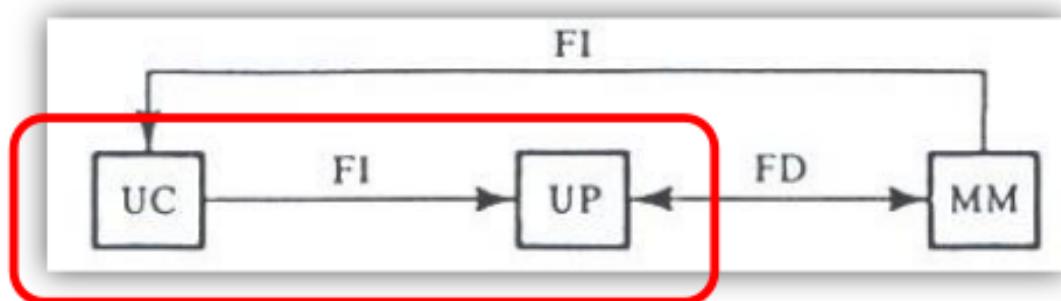
Corresponde a los computadores uni-procesador, ya que existe una única unidad de control (UC) que recibe las instrucciones de memoria, las decodifica y genera los códigos que definen la operación correspondiente a cada instrucción que debe realizar la unidad de procesamiento

(UP) de datos.

Descripción Funcional



Descripción Estructural

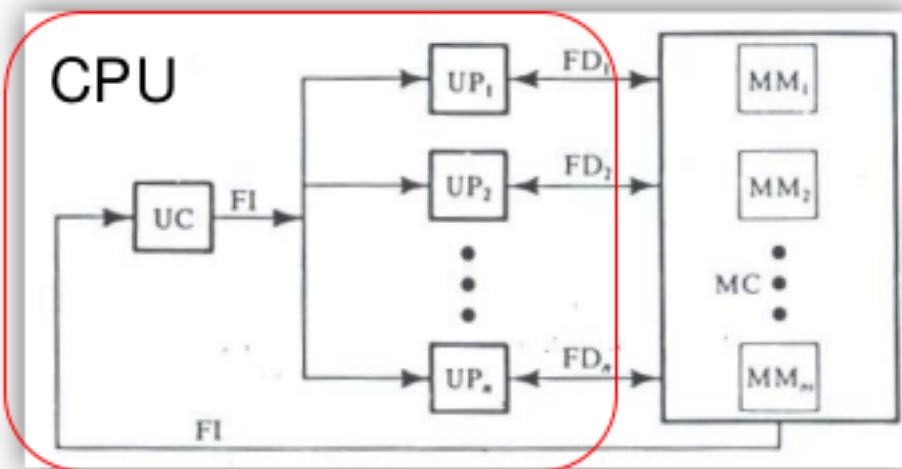


CPU, núcleo, procesador

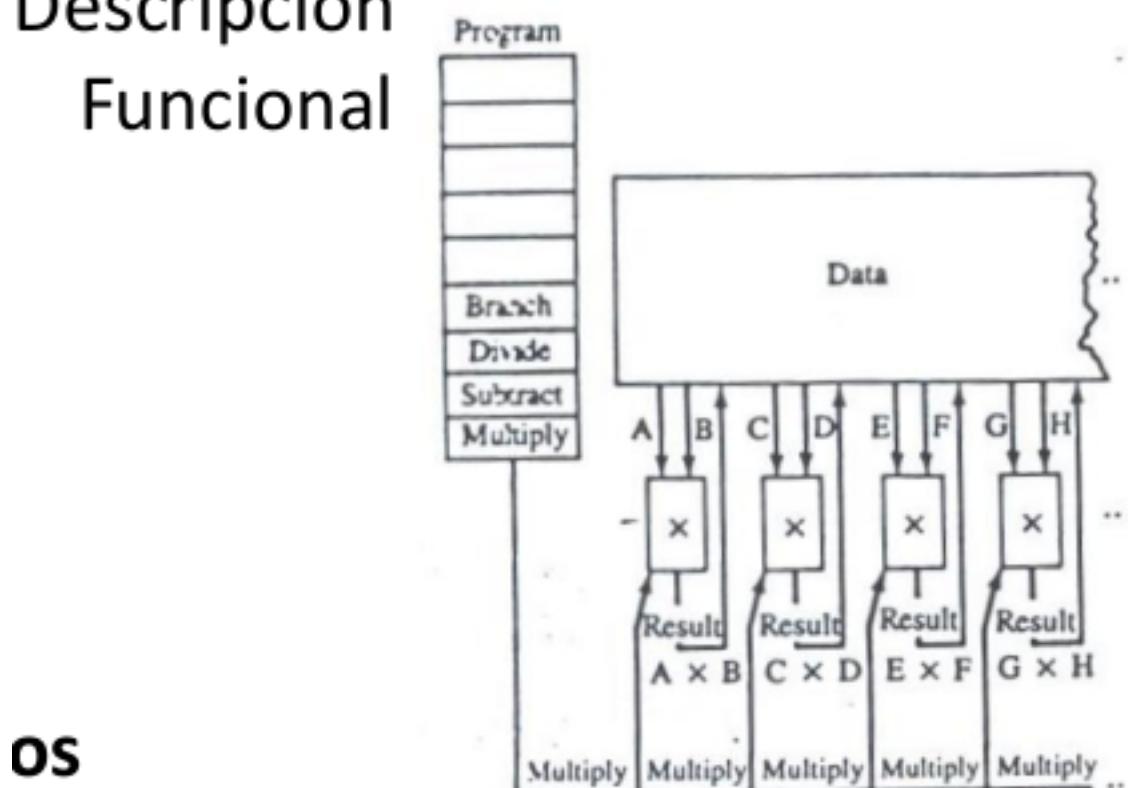
- **Computadores SIMD.** Un único flujo de instrucciones (SI) procesa operandos y genera resultados, definiendo varios flujos de datos (MD), dado que cada instrucción codifica realmente varias operaciones iguales, cada una actuando sobre operadores distintos.

Los códigos que generan la única unidad de control a partir de cada instrucción actúan sobre varias unidades de procesamiento distintas (UP_i). Por lo que se pueden realizar varias operaciones similares simultáneas con operandos distintos. Aprovechan paralelismo de datos.

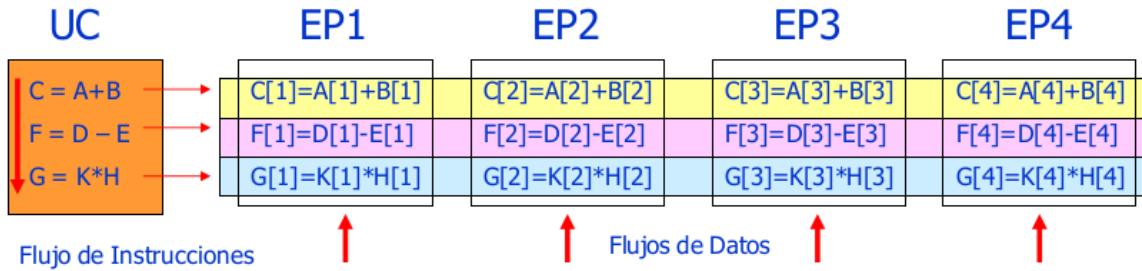
Descripción Estructural



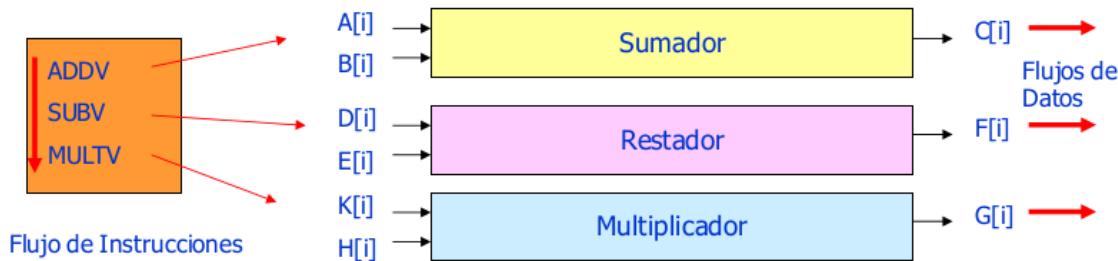
Descripción Funcional



Procesador Matricial

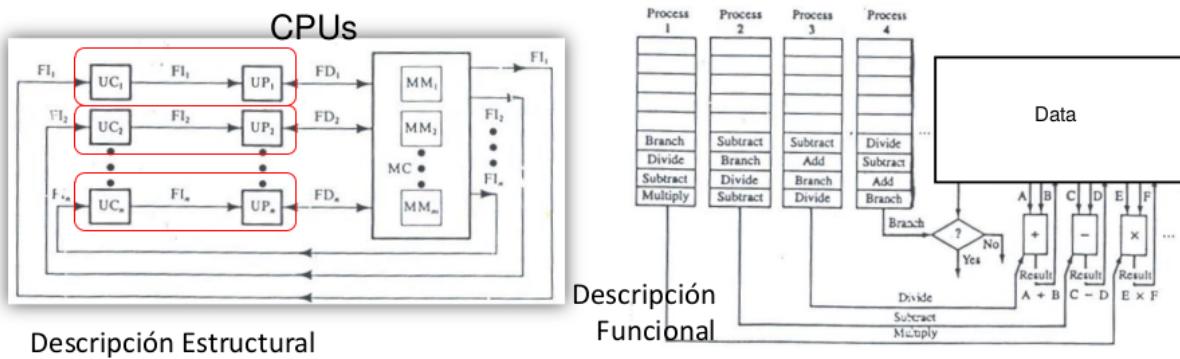


Procesador Vectorial



- **Computadores MIMD.** El computador ejecuta varias secuencias o flujos distintos de instrucciones (MI) y cada uno de ellos procesa operandos y genera resultados definiendo un único flujo de instrucciones, de forma que existen varios flujos de datos (MD) uno por cada flujo de instrucciones.

Corresponde con multinúcleos, multiprocesadores y multicomputadores. Puede aprovechar parallelismo funcional. Existen varias unidades de control que decodifican las instrucciones correspondientes a distintos programas. Cada uno de estos programas procesa conjuntos de datos diferentes, que definen distintos flujos de datos.



Corresponde con Multinúcleos, Multiprocesadores y Multicomputadores: Puede aprovechar, además, **paralelismo funcional**

```
for i:=1 to 4 do
begin
  C[i]:=A[i]+B[i];
end;
```

Proc 1

```
for i:=1 to 4 do
begin
  F[i]:=D[i]-E[i];
end;
```

Proc 2

```
for i:=1 to 4 do
begin
  G[i]:=K[i]*H[i];
end;
```

Proc 3

- **Computadores MISD.** Se ejecutan varios flujos distintos de instrucciones (MI) aunque todos actúan sobre el mismo flujo de datos (SD).

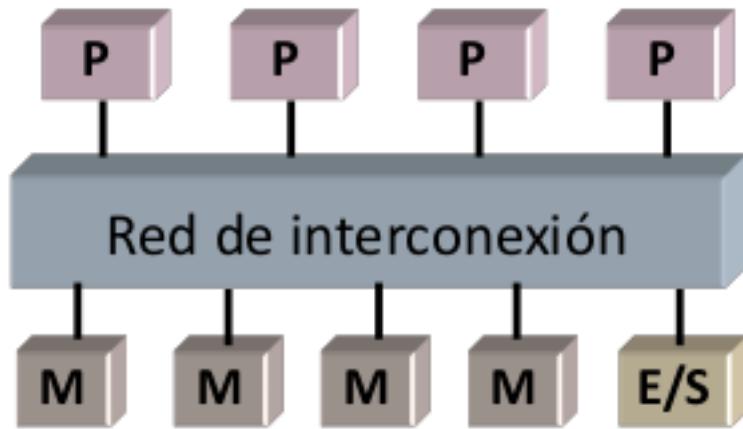
No existen computadores que funcionen según este modelo. Se puede simular en un código este modelo para aplicaciones que procesan una secuencia o flujo de datos.

2.3.5 Sistemas de memoria

2.3.5.1 Clasificación de computadores paralelos MIMD según el sistema de memoria

Los sistemas multiprocesadores se han clasificado atendiendo a la organización del sistema de memoria:

- **Sistemas con memoria compartida (SM) o multiprocesadores.** Son sistemas en los que todos los procesadores comparten el mismo espacio de direcciones. El programador no necesita conocer dónde están almacenados los datos.
- **Sistemas con memoria distribuida (DM) o multicomputadores.** Son sistemas en los que cada procesador tiene su propio espacio de direcciones particular. El programador necesita conocer dónde están almacenados los datos.



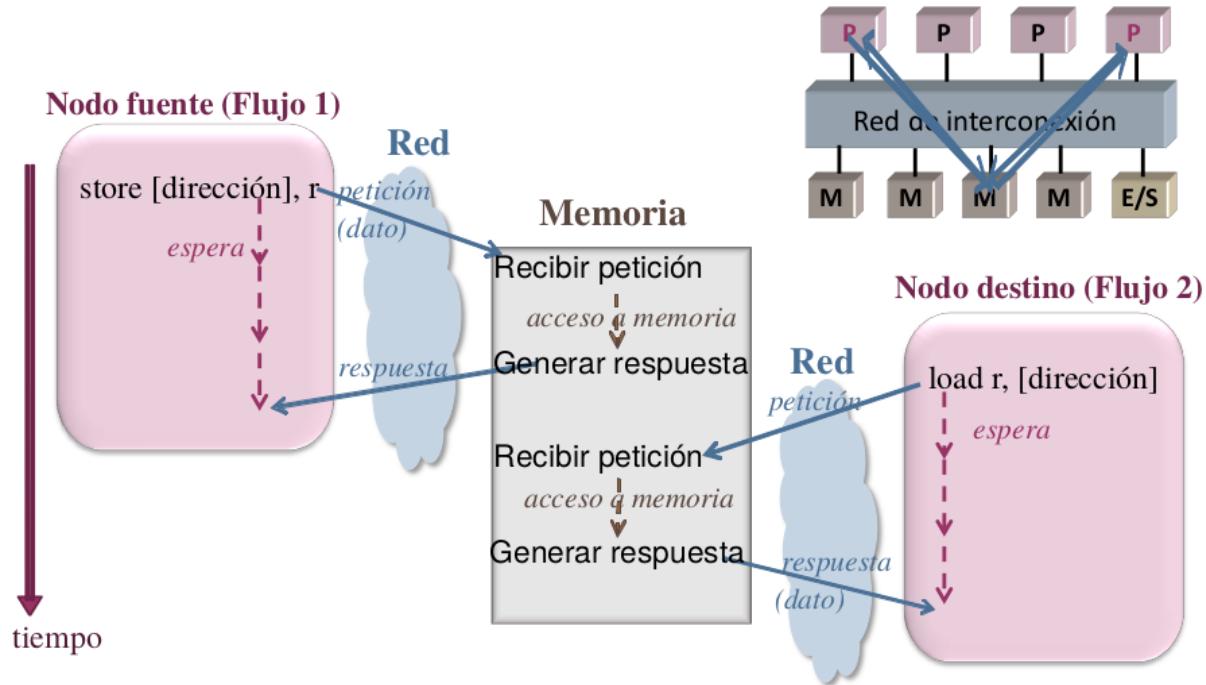
2.3.5.2 Comparativa SMP (Symmetric MultiProcessor) y multicomputadores.

- **Multiprocesador con memoria centralizada (SMP).** Es un multiprocesador en el que el tiempo de acceso de los procesadores a memoria es igual sea cual sea la posición de memoria a la que acceden, es una estructura simétrica.
 - Mayor latencia.
 - Poco escalable.
 - La comunicación es implícita mediante variables compartidas.
 - Los datos no están duplicados en memoria principal.
 - Necesita implementar primitivas de sincronización.
 - La distribución de código y datos entre procesadores no es necesaria.
 - La programación es más sencilla.
- **Multicomputador.**
 - Menor latencia.
 - Más escalable.
 - La comunicación es explícita mediante software para paso de mensajes (*send/receive*).
 - Los datos están duplicados en memoria principal y se copian datos entre módulos de memoria de diferentes procesadores.
 - La sincronización se hace mediante software de comunicación.
 - La distribución de código y datos entre procesadores es necesaria y se necesitan herramientas de programación más sofisticadas.
 - La programación es más difícil.

2.3.5.3 Comunicación uno-a-uno en un multiprocesador.

Los diferentes procesadores que ejecutan una aplicación pueden requerir sincronizarse en algún momento. Por ejemplo, si el procesador A utiliza un dato que produce el procesador B, A deberá esperar a que B lo genere. En la siguiente imagen vemos la transferencia de datos en un multiprocesador. El proceso que ejecuta la instrucción de carga espera hasta recibir el contenido de la dirección. El proceso que ejecuta la instrucción de almacenamiento puede esperar a que termine para garantizar que se mantiene un orden

en los accesos a memoria. Obsérvese que para que la transferencia de datos e realice de forma efectiva habría que sincronizar los procesos fuente y destino.



Secuencial	Paralelo	
<code>...</code> <code>A=valor;</code> <code>...</code> <code>copia=A;</code> <code>...</code>	F1 <code>...</code> <code>A=valor;</code> <code>...</code>	F2 <code>...</code> <code>copia=A;</code> <code>...</code>

F1 es el flujo de control productor del dato (*envía el dato*)

F2 es el flujo de control consumidor del dato (*recibe el dato*)

Paralelo multiproc. ($K=0$)

F1

...

A=valor;
K=1;

...

F2

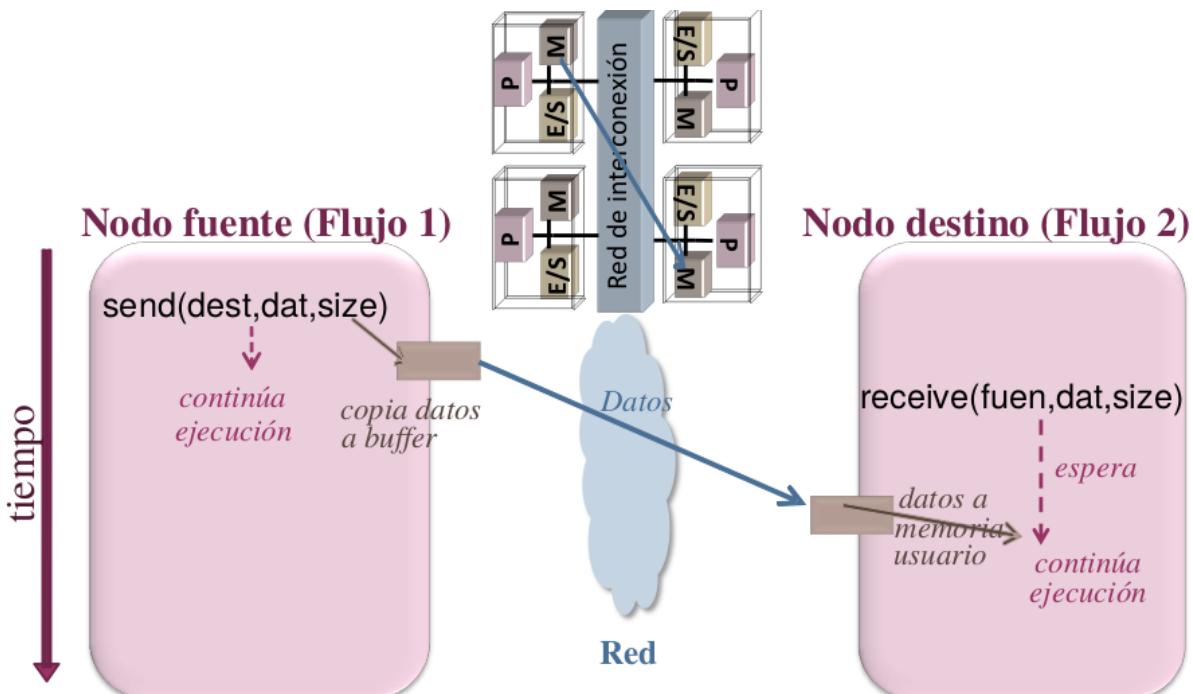
...

while (K==0) { };
copia=A;

...

Se debe garantizar que el flujo de control **consumidor** del dato lea la variable compartida (A) cuando el **productor** haya escrito en la variable el dato.

En multicomputadores se aprovechan los mecanismos de comunicación para implementar sincronización. Con una función de recepción bloqueante, es decir, que deja al proceso que la ejecuta detenido hasta que se reciba el dato, se puede implementar sincronización. En la siguiente figura podemos ver la transferencia asíncrona (con función *receive* bloqueante) de datos en un multicomputador. EN transferencia asíncrona se requiere almacenamiento intermedio para evitar esperas. El proceso fuente continúa la ejecución en cuanto los datos se copien en un *buffer*. El destino espera en el *receive* bloqueante a que lleguen los datos, en caso de que estos no hayan llegado aún.



Secuencial	Paralelo	
	<u>F1</u>	<u>F2</u>
... A=valor; ... copia=A; A=valor; copia=A; ...

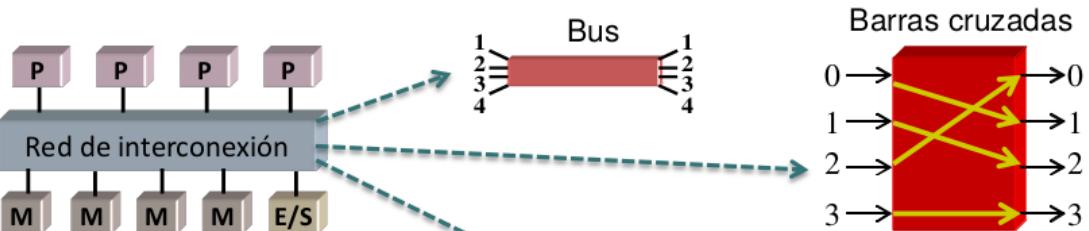
F1 es el flujo de control productor del dato (*envía el dato*)
F2 es el flujo de control consumidor del dato (*recibe el dato*)

Paralelo multicamputador (size = 4 byte)	
<u>F1</u>	<u>F2</u>
... send(F2, valor, 4); receive(F1,copia,4); ...

2.3.5.4 Incremento de escalabilidad en multiprocesadores y red de interconexión.

Como los SMP tienen escasa escalabilidad, se ha intentado incrementar la escalabilidad en multiprocesadores:

- Aumentar caché del procesador.
- Usar redes de menor latencia y mayor ancho de banda que un bus (jerarquía de buses, barras cruzadas, multietapa).
- Distribuir físicamente los módulos de memoria entre los procesadores (pero se sigue compartiendo espacio de direcciones).



- Incremento escalabilidad multiprocesadores:
 - Aumentar cache del procesador
 - Usar redes de menor latencia y mayor ancho de banda que un bus (jerarquía de buses, barras cruzadas, multietapa)
 - Distribuir físicamente los módulos de memoria entre los procesadores (pero se sigue compartiendo espacio de direcciones)

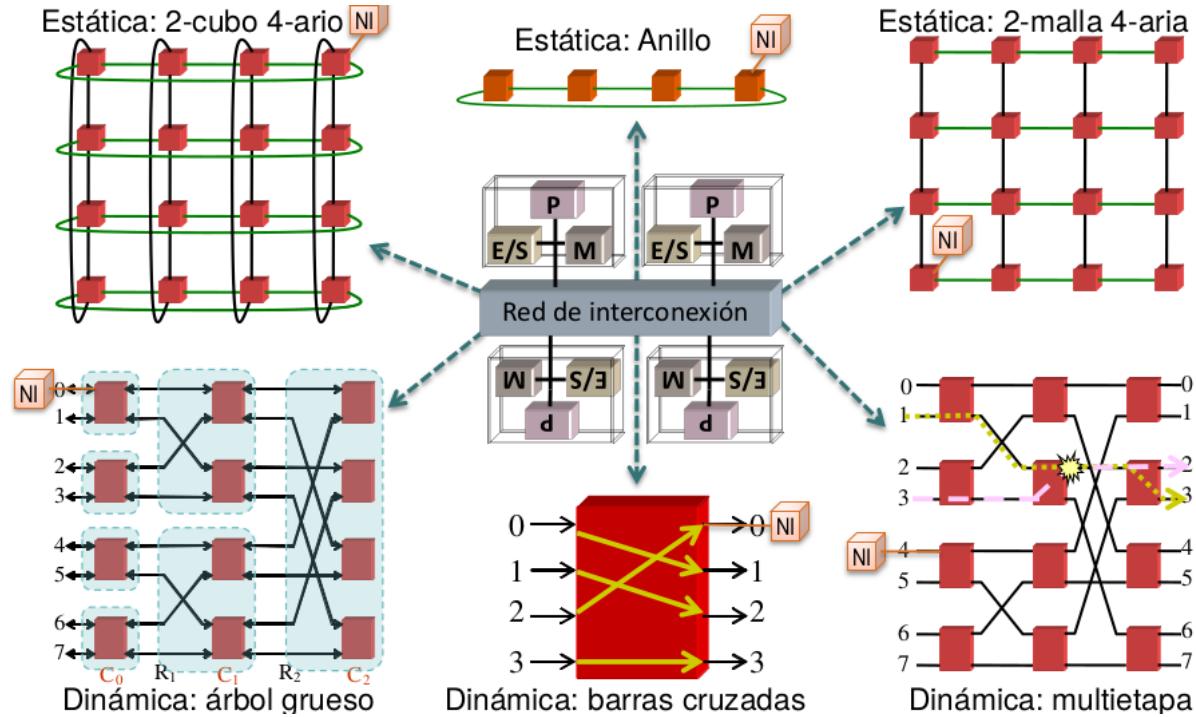
2.3.5.5 Clasificación completa de computadores según el sistema de memoria.

- Multiprocesadores.

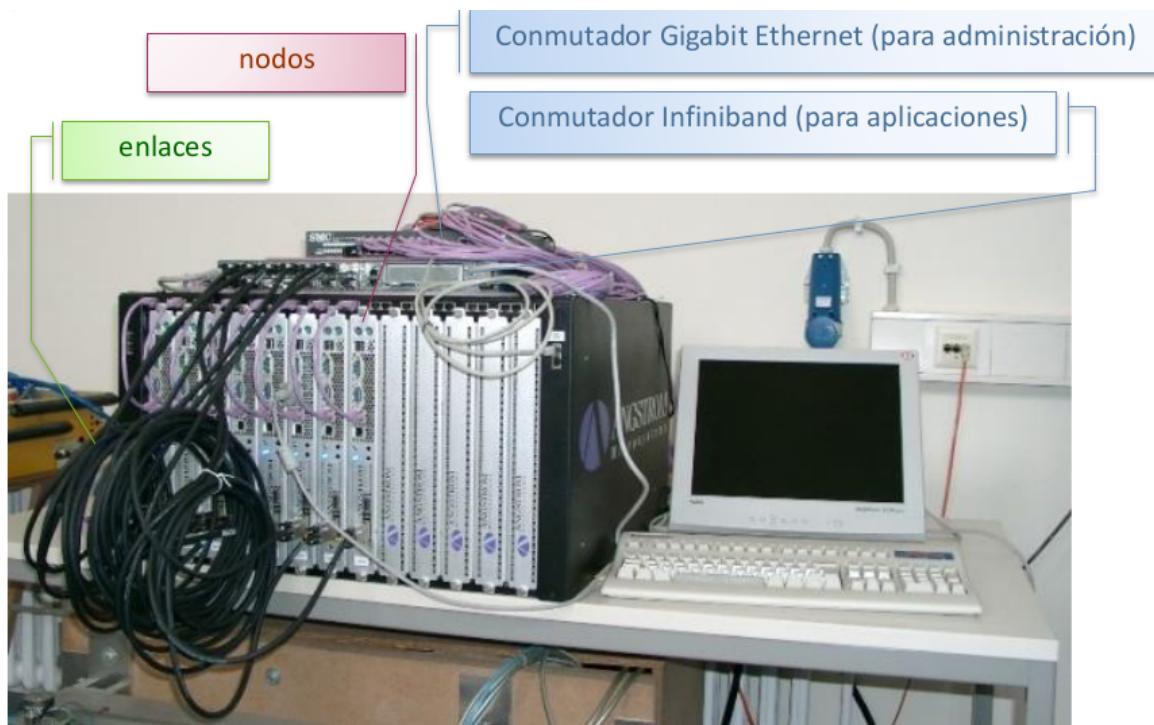
- UMA (*Uniform Memory Access*).
 - SMP.
- NUMA (*Non-Uniform Memory Access*).
 - NUMA. Arquitecturas con acceso a memoria no uniforme sin coherencia de caché entre nodos. No incorporan hardware para evitar problemas por incoherencias entre cachés de distintos nodos. Esto hace que los datos modificables compartidos no se puedan trasladar a caché de nodos remotos; hay que acceder a ellos individualmente a través de la red. Se puede hacer más tolerable la latencia utilizando precaptación (*prefetching*) de memoria y procesamiento multihebra.
 - CC-NUMA. Arquitecturas con acceso a memoria no uniforme y con caché coherente. Tienen hardware para mantener coherencia entre cachés de distintos nodos, que se encarga de las transferencias de datos compartidos entre nodos. El hardware para mantenimiento de coherencia supone un coste añadido e introduce un retardo que hace que estos sistemas escalen en menor grado que un NUMA.
 - COMA. Arquitecturas con acceso a memoria solo caché. La memoria local de los procesadores se gestiona como caché. El sistema de mantenimiento se encarga de llevar dinámicamente el código y los datos a los nodos donde se necesiten.

Multi-computadores Memoria no compartida	NORMA No Remote Memory Access	ej. cluster, red de computadores	Memoria físicamente distribuida	+ + Nivel de empaquetamiento y conexión
Multi-procesadores Memoria compartida Un único espacio de direcciones	NUMA Non-Uniform Memory Access	NUMA		
		CC-NUMA		
		COMA		
	UMA Uniform Memory Access	SMP Symmetric MultiProcessor	Memoria físicamente centralizada	- - Escalabilidad

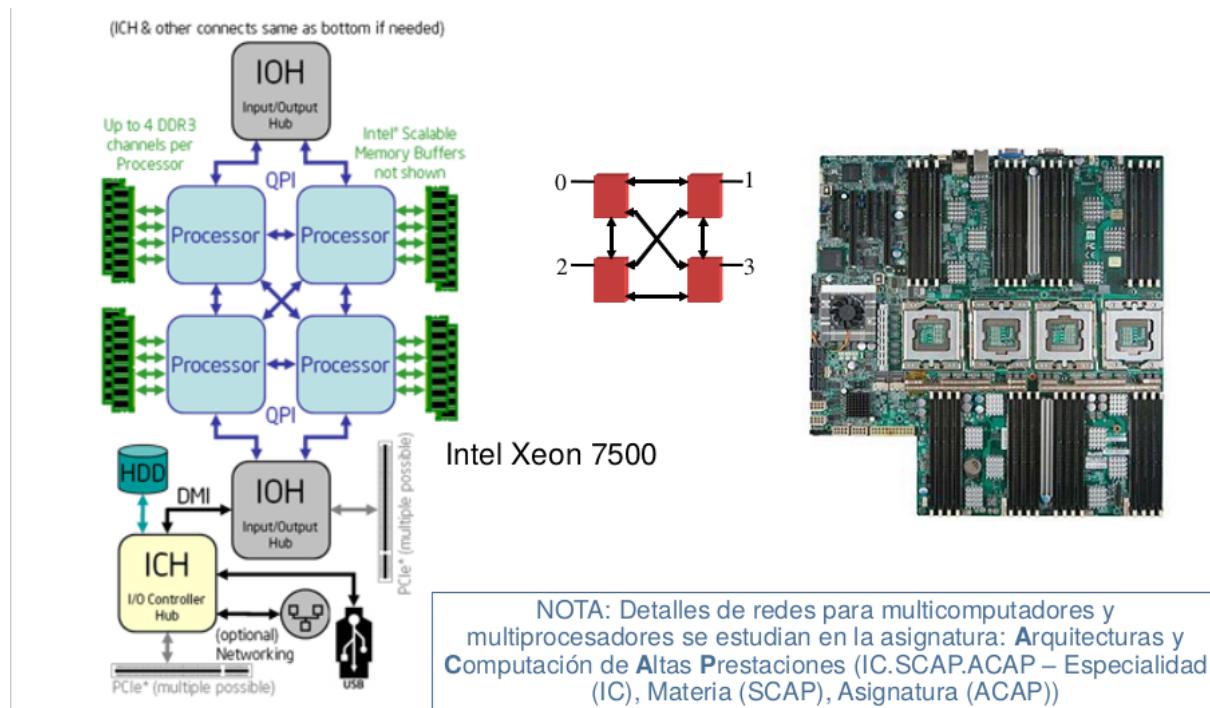
2.3.5.6 Red en sistemas con memoria físicamente distribuida (NI: Network Interface).



Ejemplo: Red (con conmutador o switch) de barras cruzadas.



Ejemplo: Placa CC-NUMA con red estática



2.3.6 Flujos de control (propuesta de clasificación de arquitecturas con múltiples flujos de control (o threads o flujos de instrucciones))

- **TLP (Thread Level Parallelism).** Ej. múltiples flujos de control concurrentemente o en paralelo.
 - **Implícito.** Flujos de control creados y gestionados por la arquitectura.
 - **Explícito.** Flujos de control creados y gestionados por el SO.

- **Con una instancia SO.** Multiprocesadores, multicores, cores multithread...
- **Con múltiples instancias SO.** Multicomputadores.

2.3.7 Nivel del paralelismo aprovechado (propuesta de clasificación)

- **Arquitecturas con DLP (*Data Level Parallelism*).** Ejecutan las operaciones de una instrucción concurrentemente o en paralelo: unidades funcionales vectoriales o SIMD.
- **Arquitecturas con ILP (*Instruction Level Parallelism*).** Ejecutan múltiples instrucciones concurrentemente o en paralelo: cores escalares segmentados, superescalares o VLIW/EPIC.
- **Arquitecturas con TLP (*Thread Level Parallelism*) explícito y una instancia de SO.** Ejecutan múltiples flujos de control concurrentemente o en paralelo.
 - **Cores** que modifican la arquitectura escalar segmentada, superescalar o VLIW/EPIC para ejecutar threads concurrentemente o en paralelo.
 - **Multiprocesadores:** ejecutan threads en paralelo en un computador con múltiples cores (incluye multicores).
- **Arquitecturas con TLP explícito y múltiples instancias SO.** Ejecutan múltiples flujos de control en paralelo.
 - **Multicomputadores:** ejecutan threads en paralelo en un sistema con múltiples computadores.

1.2.4 2.4 Nota histórica

- **DLP (*Data Level Parallelism*).** Unidades funcionales (o de ejecución) SIMD (o multimedia).
- **ILP (*Instruction Level Parallelism*).**
 - Procesadores/cores segmentados.
 - Procesadores con múltiples unidades funcionales.
 - Procesadores/cores superescalares
 - Procesadores/cores VLIW
- **TLP (*Thread Level Parallelism*).**
 - TLP explícito con una instancia de SO.
 - Multithread grano fino (FGMT).
 - Multithread grano grueso (CGMT).
 - Multithread simultánea (SMT).
 - Multiprocesadores en un chip (CMP) o multicores.
 - Multiprocesadores.
 - TPL explícito con múltiples instancias del SO (multicomputadores): IC.SCAP.

1.3 Lección 3. Evaluación de prestaciones.

1.3.1 3.1 Objetivos.

- Distinguir entre tiempo de CPU (sistema y usuario) de unix y tiempo de respuesta.
- Distinguir entre productividad y tiempo de respuesta.
- Obtener, de forma aproximada mediante cálculos, el tiempo de CPU, GFLOPS y los MIPS del código ejecutado en un núcleo de procesamiento.
- Explicar el concepto de ganancia en prestaciones.
- Aplicar la ley de Amdahl.

1.3.2 3.2 Medidas usuales para evaluar prestaciones.

3.2.1 Tiempo de respuesta.

- Real (*wall-clock time, elapsed time, real time*).
- $CPU\ time = user + sys$ (no incluye todo el tiempo).
- Con un flujo de control.
 - $elapsed \geq CPU\ time$.

```

1 time ./program.exe
2 elapsed 5.4
3 user 3.2
4 sys 1.0

```

- Con múltiples flujos de control
 - $elapsed < CPU\ time$, $elapsed \geq CPU\ time / n^{\circ}\ fluxos\ control$.

En el programa, **user 3.2** significa el tiempo de CPU de usuario (tiempo de ejecución en espacio de usuario). **sys 1.0** significa el tiempo de CPU de sistema (tiempo en el nivel del kernel del SO). Además, hay otro tiempo asociado a las esperas debidas a I/O o asociados a la ejecución de otros programas.

Comando time en Unix: 3.2u + 1.0s es el 78 % del tiempo transcurrido (5.4).

Alternativas para obtener tiempos:

Función	Fuente	Tipo	Resolución aprox. (microsegs)
time	SO (/usr/bin/time)	elapsed, user, system	10000
clock()	SO (time.h)	CPU	10000
gettimeofday()	SO (sys/time.h)	elapsed	1
clock_gettime() / clock_getres()	SO (time.h)	elapsed	0.001
omp_get_wtime() / omp_get_wtick()	OpenMP (omp.h)	elapsed	0.001
SYSTEM_CLOCK()	Fortran	elapsed	1

RISC: menos instrucciones que CISC
CISC tienen instrucciones de acceso a memoria de la ALU
Los RISC para acceder a memoria, solamente se pueden usar instrucciones dedicadas a memoria

```
for (i=0; i<N; i++) {
    v3[i]=v1[i]+v2[i];
}
```

$T_{CPU} = NI \times CPI \times T_{ciclo}$

$T_{CPU} = NI \times \frac{1}{IPC} \times T_{ciclo}$

$T_{CPU} = \frac{N \text{º ciclos}}{\text{código}} \times T_{ciclo}$

$T_{CPU} = \left[\sum_i NI_i \times CPI_i \right] \times T_{ciclo}$

$T_{CPU} = NI \times \left(\frac{\sum_i NI_i \times CPI_i}{NI} \right) \times T_{ciclo}$

$T_{CPU} = NI \times CPI \times T_{ciclo}$

$T_{ciclo} = 1/F$

Suma vectores de doubles

.L7:
... ; rax=0, rbx=8N
movsd v1(%rax), %xmm0
addsd v2(%rax), %xmm0
movsd %xmm0, v3(%rax)
addq \$8, %rax
cmpq %rbx, %rax
jne .L7
...

i	NI _i	CPI _i
movsd m,r	2N	4
movsd r,m	N	5
addsd m,r	N	1
addq i,r	N	1
cmp r,r	N	1
jne	N	1
		6N

Para $N = 10^3$ y $F = 100\text{MHz}$ ($\Rightarrow T_{ciclo} = 10^{-8}\text{seg./ciclo}$):

$$T_{CPU} \approx \left[6N \times \left(\frac{2N \times 4 + N \times 5 + 3N \times 1}{6N} \right) \right] \times T_{ciclo}$$

$$= 10^3 \times 16 \text{ ciclos/código} \times 10^{-8}\text{seg./ciclo}$$

$$= 16 \times 10^{-5}\text{seg./código}$$

$$TiempoDeCPU (T_{CPU}) = CiclosDelPrograma \cdot T_{CICLO} = \frac{CiclosDelPrograma}{FrecuenciaDeReloj}$$

$$CiclosporInstrucción (CPI) = \frac{CiclosDelPrograma}{NúmeroDeInstrucciones(NI)}$$

$$T_{CPU} = NI \cdot CPI \cdot T_{CICLO}$$

$$CiclosDelPrograma = \sum_{i=1}^n CPI_i \cdot I_i$$

$$CPI = \frac{\sum_{i=1}^n CPI_i \cdot I_i}{NI}$$

En el programa hay I_i instrucciones del tipo i ($i=1, \dots, n$).

Cada instrucción del tipo i consume CPI_i ciclos.

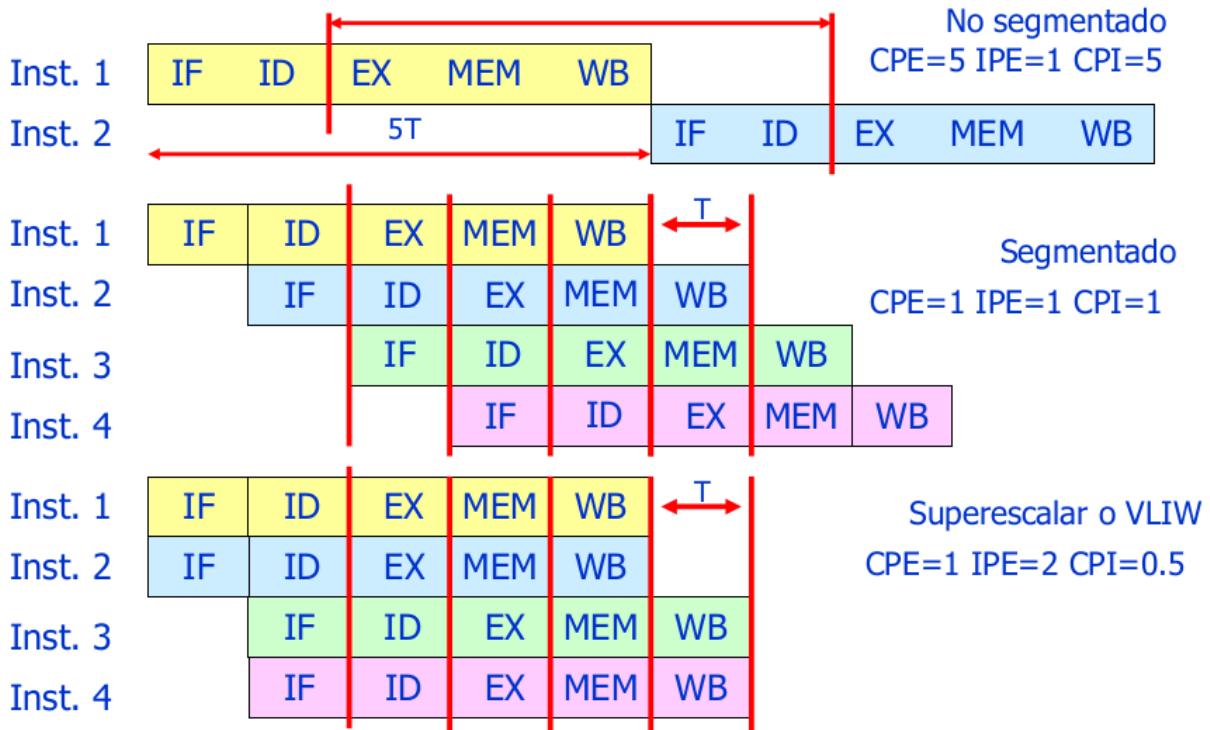
Hay n tipos de instrucciones distintos.

$$T_{CPU} = NI \cdot (CPE/IPE) \cdot T_{CICLO}$$

$$CPI = \frac{CPE}{IPE}$$

Hay procesadores que pueden lanzar para que empiecen a ejecutarse (emitir) varias instrucciones al mismo tiempo.

- **CPE:** Número mínimo de ciclos transcurridos entre los instantes en que el procesador puede emitir instrucciones
- **IPE:** Instrucciones que pueden emitirse (para empezar su ejecución) cada vez que se produce dicha emisión.

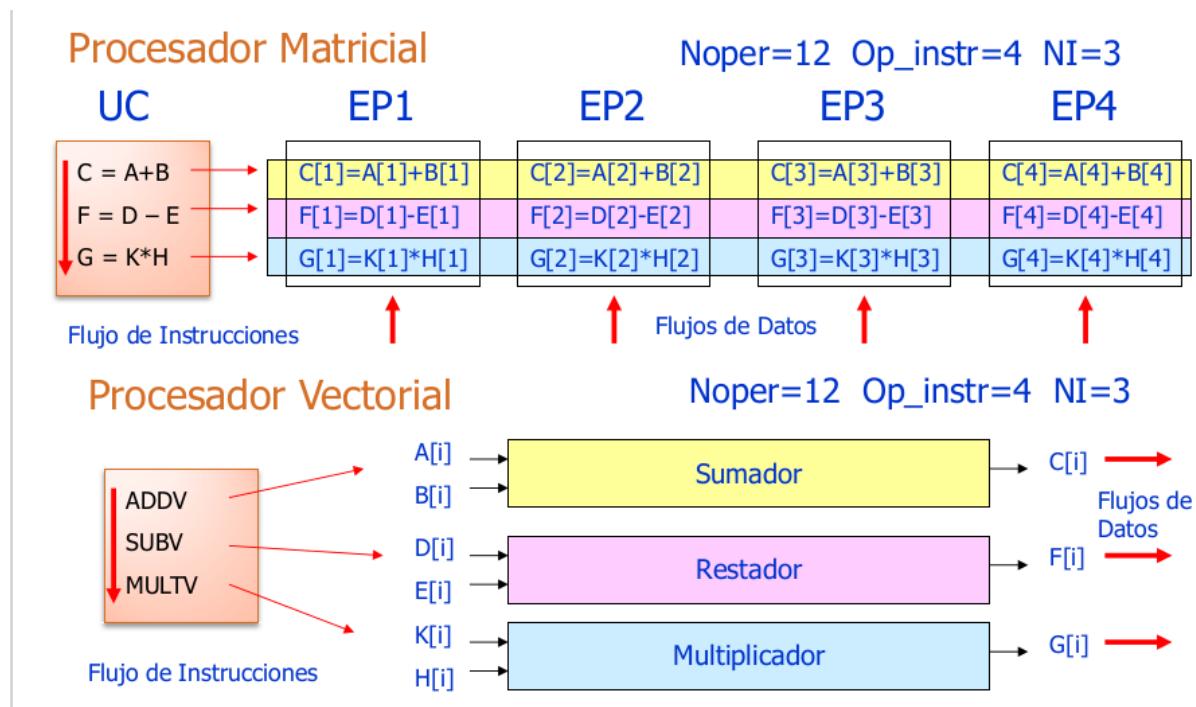


$$T_{CPU} = (Noper / OpInstr) \cdot CPI \cdot T_{CICLO}$$

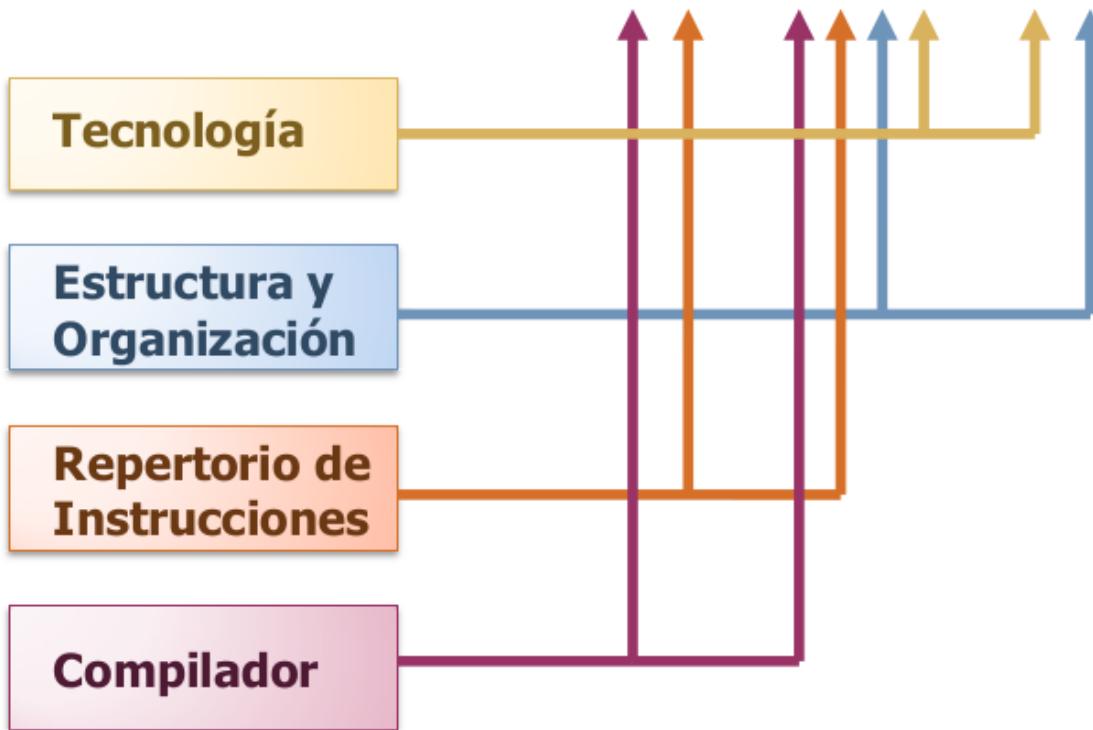
$$NI = Noper / OpInstr$$

Hay procesadores que pueden codificar varias operaciones en una instrucción.

- **Noper:** Número de operaciones que realiza el programa
- **Op_instr:** Número de operaciones que puede codificar una instrucción.



$$T_{CPU} = NI \times CPI \times T_{ciclo}$$



3.2.2 Productividad: MIPS, MFLOPS.

MIPS: millones de instrucciones por segundo.

$$MIPS = \frac{NI}{T_{CPU} \cdot 10^6} = \frac{F(frecuencia)}{CPI \cdot 10^6}$$

- Depende del repertorio de instrucciones (difícil la comparación de máquinas con repertorios distintos)
- Puede variar con el programa (no sirve para caracterizar la máquina)
- Puede variar inversamente con las prestaciones (mayor valor de MIPS corresponde a peores prestaciones)

MFLOPS: millones de operaciones en coma flotante por segundo.

$$MFLOPS = \frac{\text{OperacionesEnComaFlotante}}{T_{CPU} \cdot 10^6}$$

- No es una medida adecuada para todos los programas (sólo tiene en cuenta las operaciones en coma flotante del programa)
- El conjunto de operaciones en coma flotante no es constante en máquinas diferentes y la potencia de las operaciones en coma flotante no es igual para todas las operaciones (por ejemplo, con diferente precisión, no es igual una suma que una multiplicación..).
- Es necesaria una normalización de las instrucciones en coma flotante

MIPS y FLOPS

AC FTC

-02

```
;r12=&x,r13=&y,rax=0,rbp=N,xmm1=a
.L6:
    movsd (%r12,%rax,8), %xmm0
    mulsd %xmm1, %xmm0
    addsd (%r13,%rax,8), %xmm0
    movsd %xmm0, (%r13,%rax,8)
    addq $1, %rax
    cmpl %eax, %ebp
    jg .L6
T(N=226)=0.182 seg.
```

$$\begin{aligned} GIPS &= \frac{NI}{T_{CPU} \times 10^9} = \frac{N \times 7}{0.182 \times 10^9} \\ &= \frac{2^{26} \times 7}{0.182 \times 10^9} \approx 2.58 \text{ GIPS} \end{aligned}$$

$$\begin{aligned} GFLOPS &= \frac{n^o FP}{T_{CPU} \times 10^9} = \frac{N \times 2}{0.182 \times 10^9} \\ &= \frac{2^{26} \times 2}{0.182 \times 10^9} \approx 0.737 \text{ GFLOPS} \end{aligned}$$

-03

```
;r12=&x,r13=&y,rax=0,rbp=N/2,xmm1=a
.L7:
    movapd (%r12), %xmm0
    addq $1, %rax
    addq $16, %r12
    addq $16, %r13
    mulpd %xmm1, %xmm0
    addpd -16(%r13), %xmm0
    movaps %xmm0, -16(%r13)
    cmpl %ebp, %eax
    jb .L7
T(N=226)=0.178 seg.
```

$$\begin{aligned} GIPS &= \frac{NI}{T_{CPU} \times 10^9} = \frac{(N/2) \times 9}{0.178 \times 10^9} \\ &= \frac{2^{25} \times 9}{0.178 \times 10^9} \approx 1.7 \text{ GIPS} \end{aligned}$$

$$GFLOPS = \frac{2^{26} \times 2}{0.178 \times 10^9} \approx 0.754 \text{ GFLOPS}$$

1.3.3 3.3 Conjunto de programas de prueba (Benchmark).

- Propiedades exigidas a medidas de prestaciones:

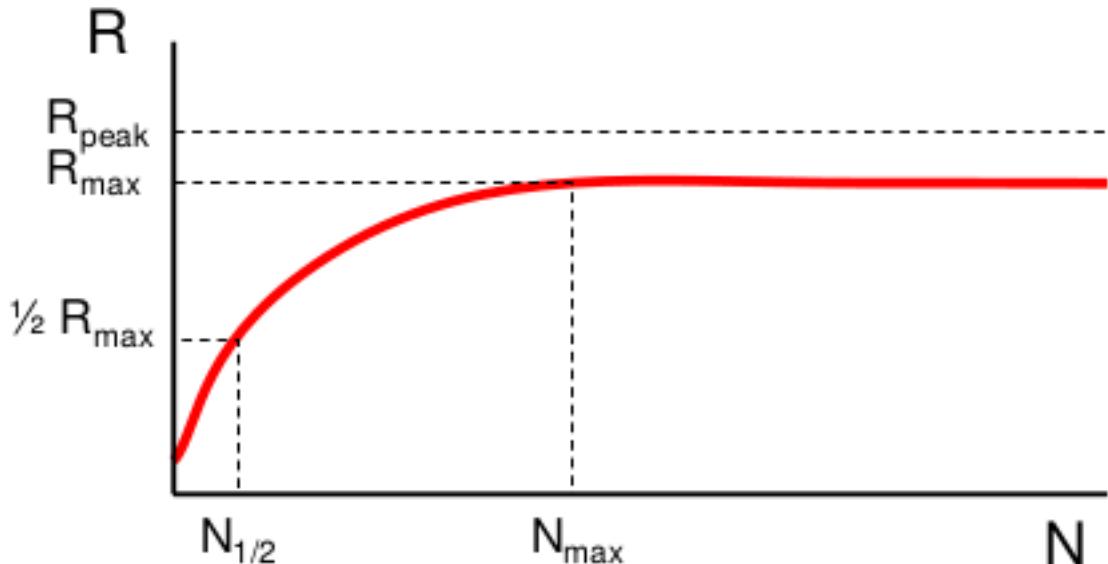
- Fiabilidad => Representativas, evaluar diferentes componentes del sistema y reproducibles.
 - Permitir comparar diferentes realizaciones de un sistema o diferentes sistemas => Aceptadas por todos los interesados (usuarios, fabricantes, vendedores).
- Interesados:
- Vendedores y fabricantes de hardware o software.
 - Investigadores de hardware o software.
 - Compradores de hardware o software.
- Tipos de Benchmarks:
- De bajo nivel o microbenchmark.
 - testping-pong, evaluación de las operaciones con enteros o con flotantes.
 - Kernels.
 - resolución de sistemas de ecuaciones, multiplicación de matrices, FFT, descomposición LU.
 - Sintéticos.
 - Dhrystone, Whetstone.
 - Programas reales.
 - SPEC CPU2006: enteros (gcc, gzip, perlbench).
 - Aplicaciones diseñadas.
 - Predicción de tiempo, simulación de terremotos.

1.3.4 3.3.1 LINPACK.

El núcleo de este programa es una rutina denominada DAXPY (Double precision- real Alpha X Plus Y) que multiplica un vector por una constante y los suma a otro vector. Las prestaciones obtenidas se escalan con el valor de N (tamaño del vector):

```
1 | for (i=0; i<N; i++)
2 |   y[i] = alpha*x[i] + y[i];
```

```
for (i=0 ; i<N ; i++)
y[i] = alpha*x[i] + y[i];
```



1.3.5 3.4 Ganancia en prestaciones.

3.4.1 Mejora o ganancia de prestaciones (speed-up o ganancia en velocidad).

Si en un computador se incrementan las prestaciones de un recurso haciendo que su velocidad sea p veces mayor (ejemplos: se utilizan p procesadores en lugar de uno, la ALU realiza las operaciones en un tiempo p veces menor...):

El incremento de velocidad que se consigue en la nueva situación con respecto a la previa (máquina base) se expresa mediante la ganancia de velocidad o speed-up, S_p

$$S_p = \frac{V_p}{V_1} = \frac{T_1}{T_p}$$

donde

V_1 : velocidad de la máquina base.

V_p : velocidad de la máquina mejorada (un factor p en uno de sus componentes).

T_1 : tiempo de ejecución en la máquina base.

T_p : tiempo de ejecución en la máquina mejorada.

Si se incrementan las prestaciones de un sistema, el incremento en prestaciones (velocidad) que se consigue en la nueva situación, p , con respecto a la previa (sistema base, b) se expresa mediante la ganancia en prestaciones o speed-up, S

$$S = \frac{V_p}{V_b} = \frac{T_b}{T_p}$$

$$S = \frac{T_{CPU}^b}{T_{CPU}^p} = \frac{NI^b \cdot CPI^b \cdot T_{CICLO}^b}{NI^p \cdot CPI^p \cdot T_{ciclo}^p}$$

donde

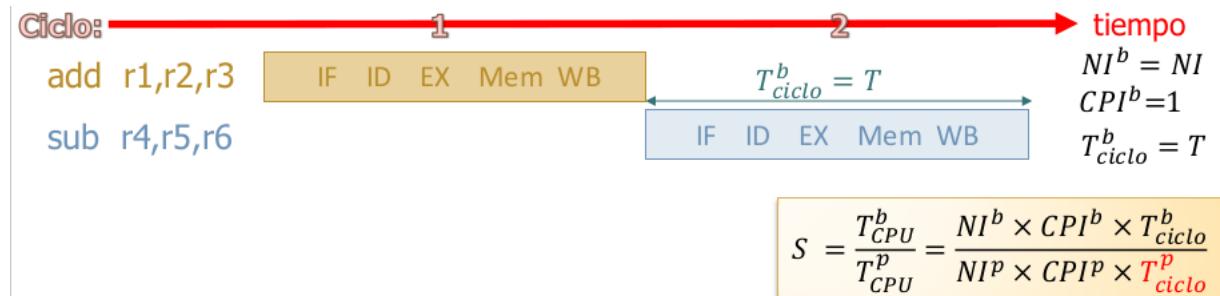
V_b : velocidad de la máquina base.

V_p : velocidad de la máquina mejorada (un factor p en uno de sus componentes).

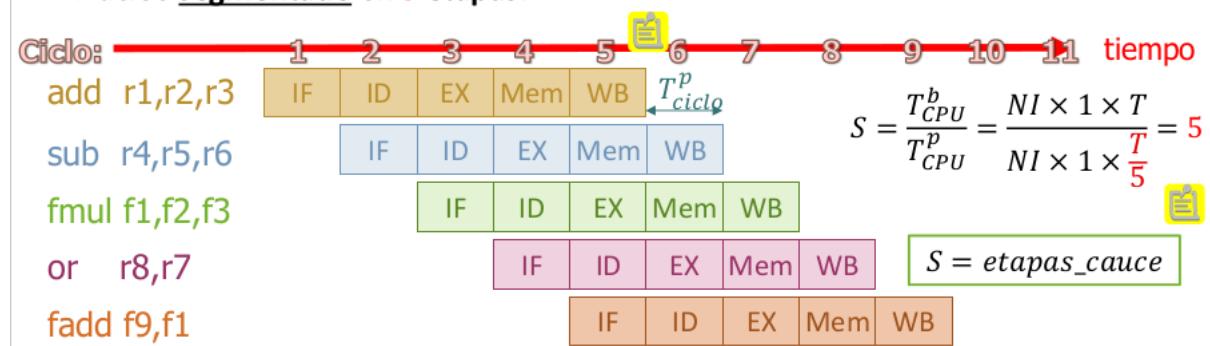
T_b : tiempo de ejecución en la máquina base.

T_p : tiempo de ejecución en la máquina mejorada.

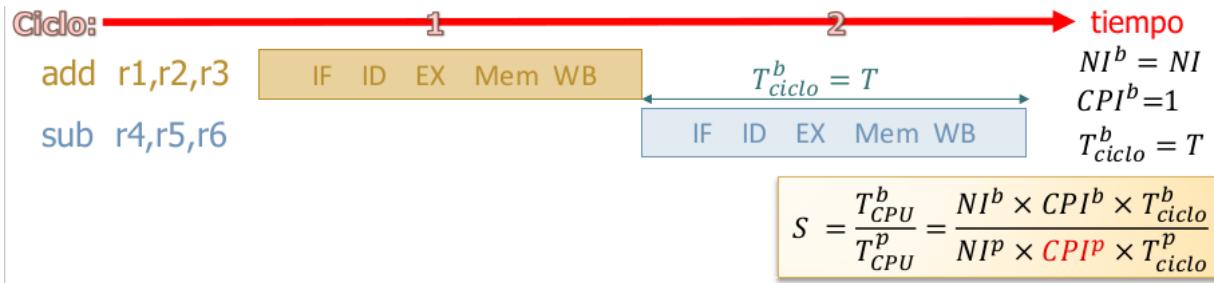
Mejora en un núcleo de procesamiento: segmentación.



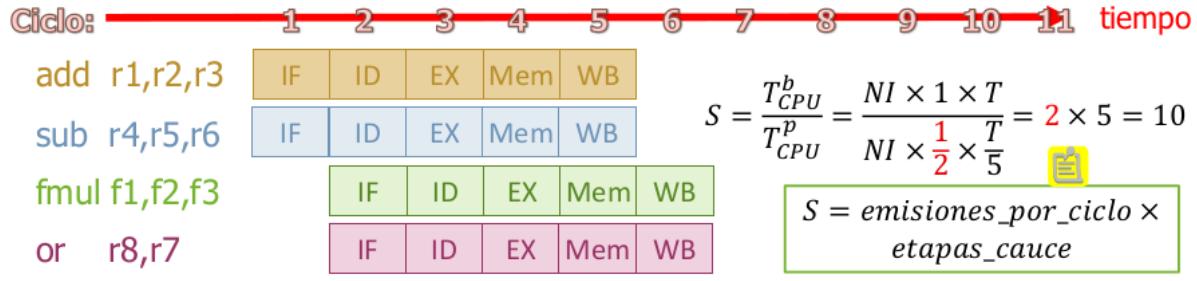
Núcleo segmentado en 5 etapas:



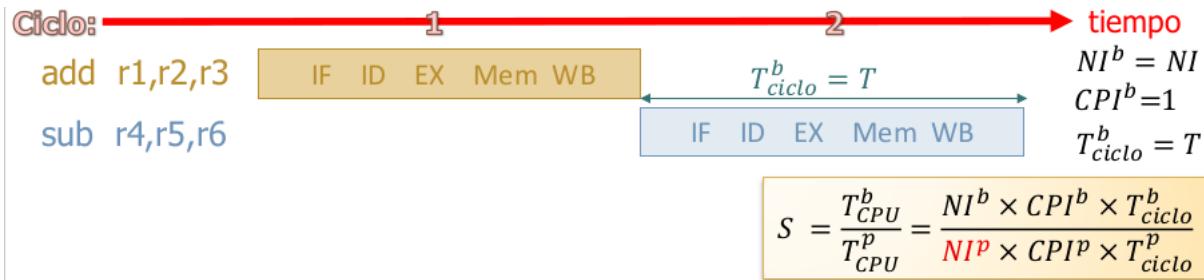
Mejora en un núcleo de procesamiento: operación superescalar.



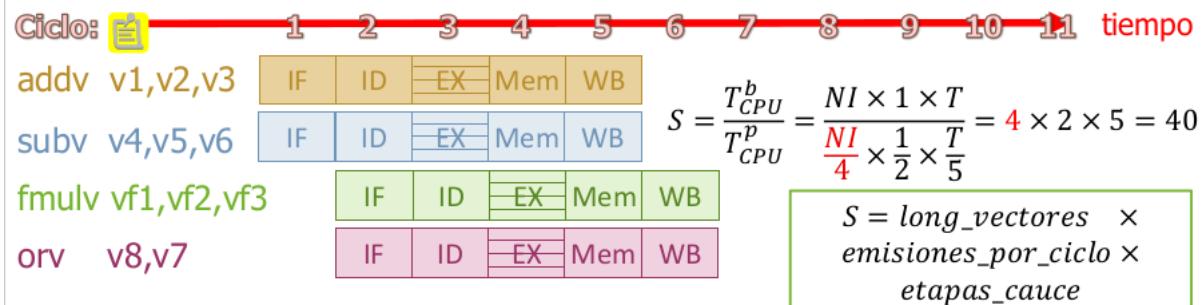
Núcleo superescalar con 2 emisiones por ciclo y 5 etapas:



Mejora en un núcleo de procesamiento: unidades funcionales SIMD.



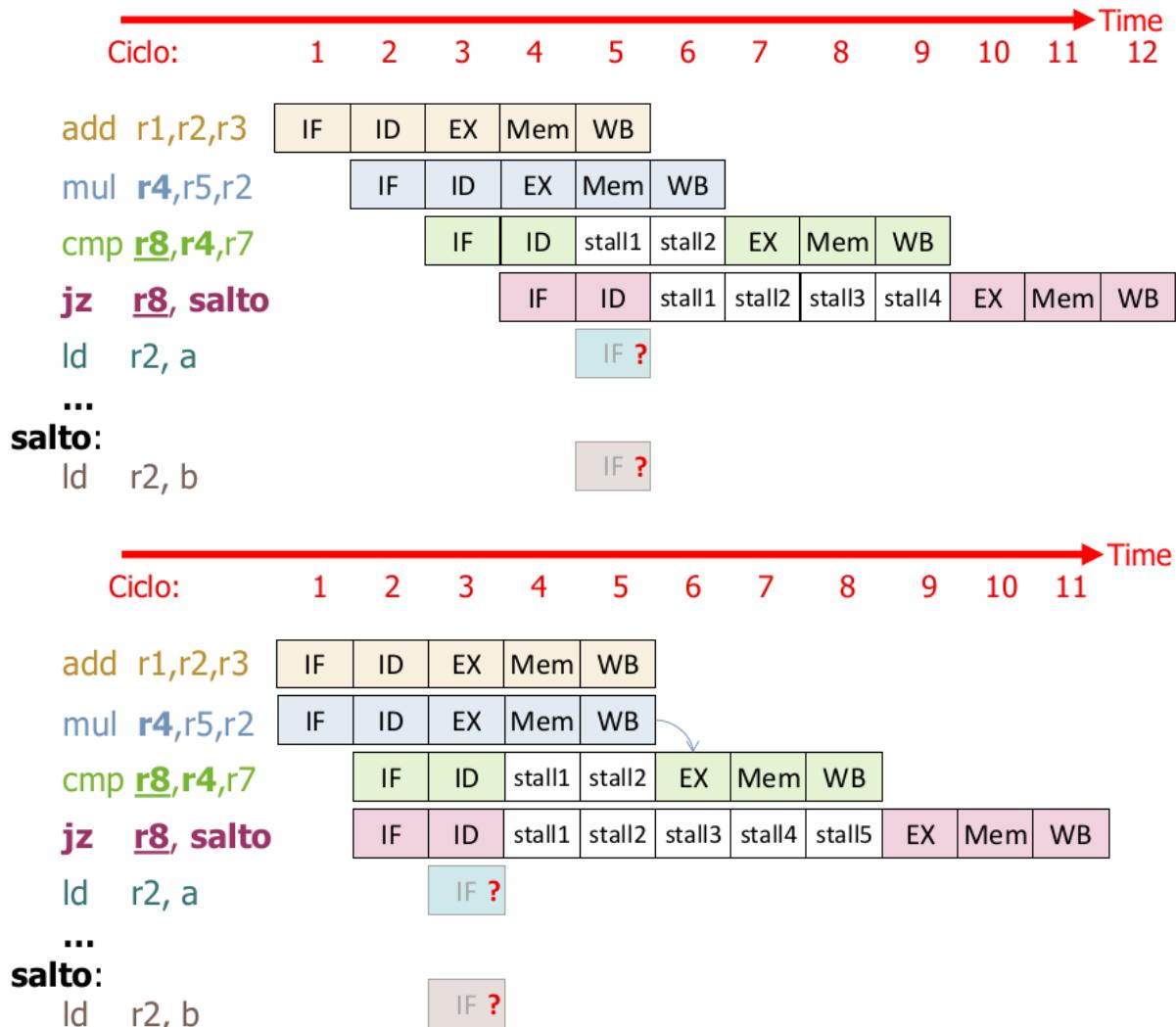
Núcleo superescalar con 2 emisiones por ciclo y 5 etapas, y unidades funcionales SIMD (vectoriales) que procesan **vectores de 4 componentes**:



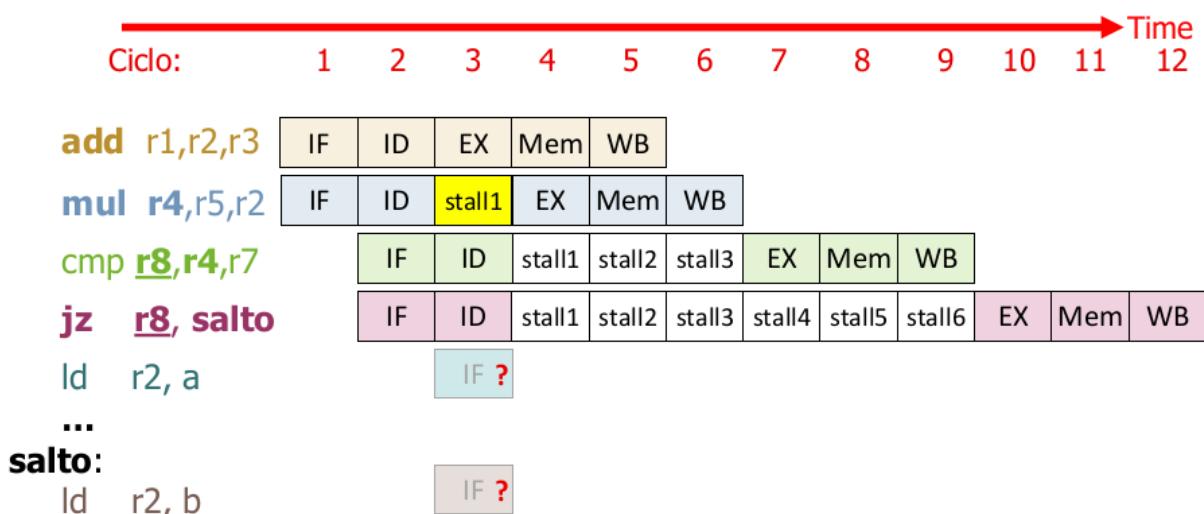
¿Qué impide que se pueda obtener la ganancia en velocidad pico?

- Riesgos:
 - Datos.
 - Control.
 - Estructurales.
- Accesos a memoria (debido a la jerarquía).

Riesgos de datos y control:



Riesgos de datos, control y estructural:



add y mul usan la misma unidad funcional

3.4.2 Ley de Amdahl.

La mejora de velocidad, S , que se puede obtener cuando se mejora un recurso de una máquina en un factor p está limitada por:

$$S = \frac{V_p}{V_b} = \frac{T_b}{T_p} \leq \frac{1}{f + \frac{1-f}{p}} = \frac{p}{1+f(p-1)}$$

Si $p \rightarrow \infty$, entonces $\frac{p}{1+f(p-1)} \rightarrow 1/f$.

Si $f \rightarrow 0$, entonces $\frac{p}{1+f(p-1)} \rightarrow p$.

donde f es la fracción del tiempo de ejecución en la máquina sin la mejora durante el que no se puede aplicar esa mejora.

Ejemplo: Si un programa pasa un 25 % de su tiempo de ejecución en una máquina realizando instrucciones de coma flotante, y se mejora la máquina haciendo que estas instrucciones se ejecuten en la mitad de tiempo, entonces $p=2$; $f=0.75$.

$$S \leq 2/(1+0.75)=1.14$$

$$S = \frac{T_b}{T_p} = \frac{1}{0.75+\frac{0.25}{2}} = 1.14$$

Hay que mejorar el caso más frecuente (lo que más se usa)

Ley enunciada por Amdahl en relación con la eficacia de los computadores paralelos: dado que en un programa hay código secuencial que no puede parallelizarse, los procesadores no se podrían utilizar eficazmente.

2 Tema 2. Programación paralela

2.1 Lección 4. Herramientas, estilos y estructuras en programación paralela.

2.1.1 Objetivos.

- Distinguir entre los diferentes tipos de herramientas de programación paralela: compiladores paralelos, lenguajes paralelos, API Directivas y API de funciones.
- Distinguir entre los diferentes tipos de comunicaciones colectivas.
- Diferenciar el estilo/paradigma de programación de paso de mensajes del de variables compartidas.
- Diferenciar entre OpenMP y MPI en cuanto a su estilo de programación y tipo de herramienta.
- Distinguir entre las estructuras de tareas/procesos/threads master-slave, cliente-servidor, descomposición de dominio, flujo de datos o segmentación, y divide y vencerás.

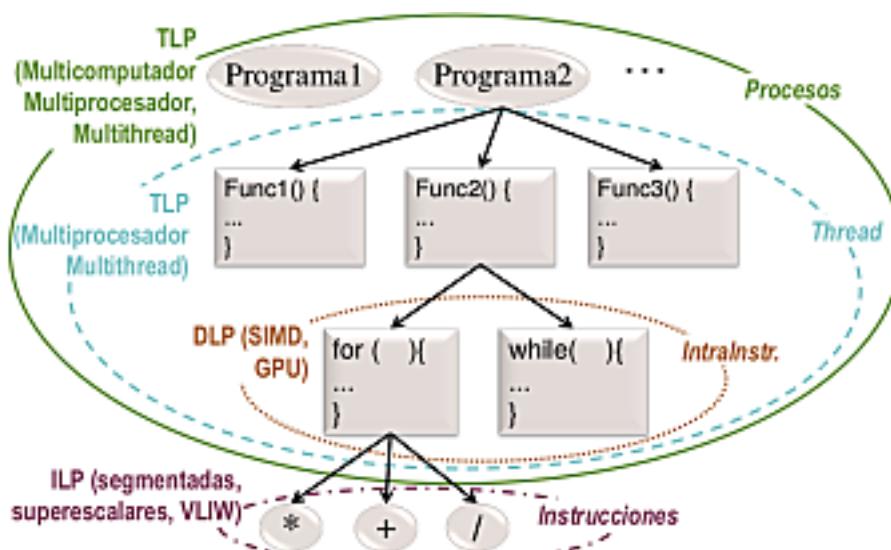
2.1.2 4.1 Problemas que plantea la programación paralela al programador. Punto de partida.

4.1.1 Problemas que plantea la programación paralela al programador.

Nuevos problemas, respecto a programación secuencial:

- **División** en unidades de cómputo independientes (tareas).
- **Agrupación/asignación** de tareas o carga de trabajo (códigos, datos) en procesos/threads.
- **Asignación** a procesadores/núcleos.
- **Sincronización y comunicación.**

Los debe abordar la herramienta de programación o el programador o SO.



4.1.2 Punto de partida.

Para obtener una versión paralela de una aplicación

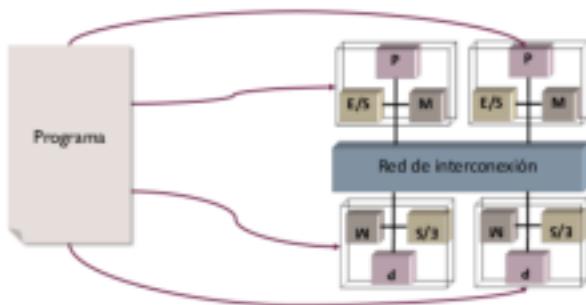
- se puede partir de una **versión secuencial** que resuelva el problema y buscar la parallelización sobre este. La versión paralela depende de la descripción del problema que se ha utilizado en la versión secuencial de partida. Ventaja: se puede saber el tiempo de ejecución real de las diferentes funciones o tareas, lo que facilita la distribución equilibrada de la carga de trabajo entre procesadores.
- se puede partir de la **definición de la aplicación**.

Se apoya en:

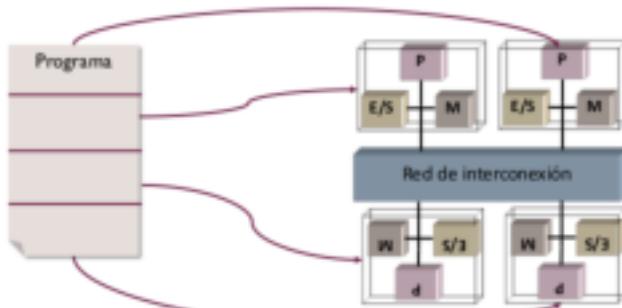
- **Programa** paralelo que resuelva un problema parecido.
- **Versiones** paralelas u optimizadas de bibliotecas de funciones: BLAS (*Basic Linear Algebra Subroutine*), LAPACK (*Linear Algebra PACKAGE*)...

4.1.3 Modos de programación MIMD.

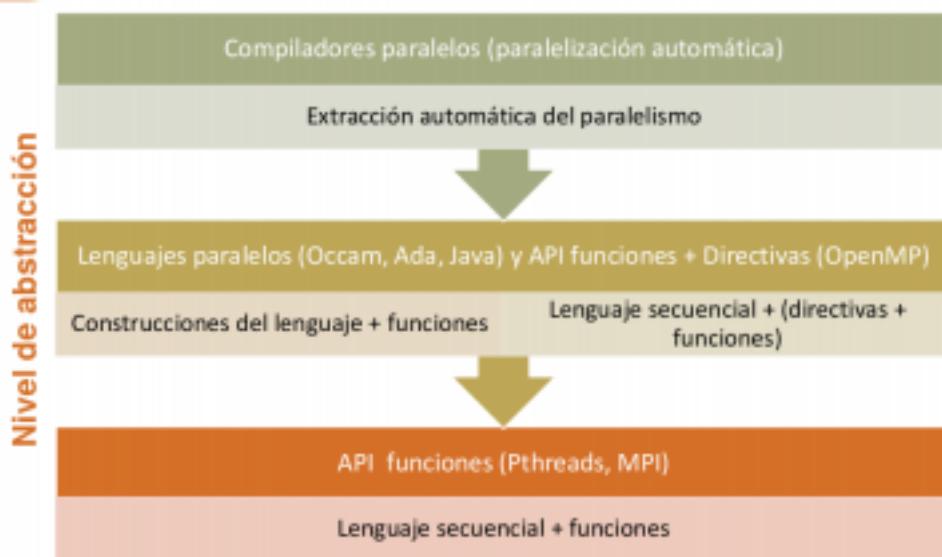
- **SPMD (Single-Program Multiple Data)**: parallelismo de datos. Todos los códigos que se ejecutan en paralelo se obtienen compilando el mismo programa. Cada copia trabaja con un conjunto de datos distintos y se ejecuta en un procesador diferente. Es recomendable en sistemas masivamente paralelos. Se usa en sistemas con memoria distribuida, en multiprocesadores y multicomputadores.



- **MPMD (Multiple-Program Multiple Data)**: parallelismo de tareas o funciones. Los códigos que se ejecutan en paralelo se obtienen compilando programas independientes. La aplicación a ejecutar (o el código secuencial inicial) se divide en unidades independientes. Cada unidad trabaja con un conjunto de datos y se asigna a un procesador distinto.



2.1.3 4.2 Herramientas para obtener código paralelo.



- Las **herramientas** permiten de forma implícita (lo hace la propia herramienta) o explícita (lo hace el programador):
 - Localizar **paralelismo** o descomponer en **tareas independientes** (*decomposition*).
 - Asignar las tareas, es decir, la **carga de trabajo** (código + datos), a procesos/threads (*scheduling*).
 - **Crear y terminar** procesos/threads (o enrolar y desenrolar en un grupo).
 - **Comunicar y sincronizar** procesos/threads.
- El programador, la herramienta o el SO se encarga de **asignar procesos/threads** a unidades de procesamiento (*mapping*).

Ejemplo: cálculo de PI con OpenMP/C.

```
#include <omp.h>
#define NUM_THREADS 4
main(int argc, char **argv) {
    double ancho,x, sum=0; int intervalos, i;
    intervalos = atoi(argv[1]);
    ancho = 1.0/(double) intervalos;
    omp_set_num_threads(NUM_THREADS); →Crear y Terminar
#pragma omp parallel →Comunicar y sincronizar
{
    #pragma omp for reduction(+:sum) private(x) \
                schedule(dynamic) →Asignar
    for (i=0;i< intervalos; i++) {
        x = (i+0.5)*ancho; sum = sum + 4.0/(1.0+x*x);
    }
    sum* = ancho;
}
```

Localizar y Asignar

reduction: tiene que sumar todas las variables `sum` y las guarda en `sum`.

Ejemplo: cálculo de PI en MPI/C. (Modificación del bucle `for` para repartir el trabajo entre los procesos)

```
#include <mpi.h>
main(int argc, char **argv) {
    double ancho,x,sum,lsum; int intervalos,i,nproc,iproc;
    if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1); → Enrolar
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
    intervalos=atoi(argv[1]);
    ancho=1.0/(double) intervalos; lsum=0;
    for (i=iproc; i<intervalos; i+=nproc) {
        x = (i+0.5)*ancho; lsum+= 4.0/(1.0+x*x);
    }
    lsum*= ancho; → Comunicar/sincronizar
    MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE,
               MPI_SUM,0,MPI_COMM_WORLD);
    MPI_Finalize(); → Desenrolar
}
```

4.2.1 Comunicaciones colectivas.

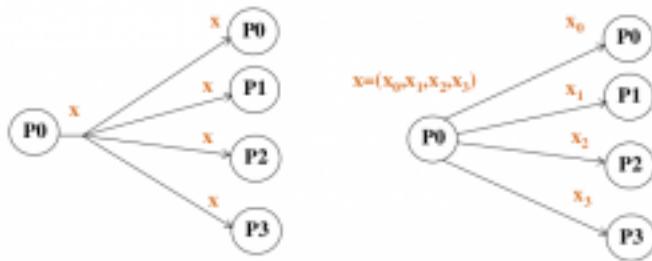


4.2.2 Comunicación uno-a-todos.

Un proceso envía y todos los procesos reciben. Hay variantes en las que el proceso que envía no forma parte del grupo y otras en las que reciben todos los del grupo excepto el que envía. Dentro de este grupo están:

- **Difusión**: todos los procesos reciben el mismo mensaje.
- **Dispersión (scatter)**: cada proceso receptor recibe un mensaje diferente.

Difusión (*broadcast*) Dispersión (*scatter*)

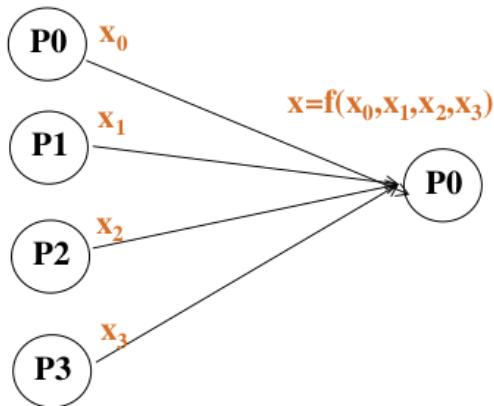


4.2.3 Comunicación todos-a-un.

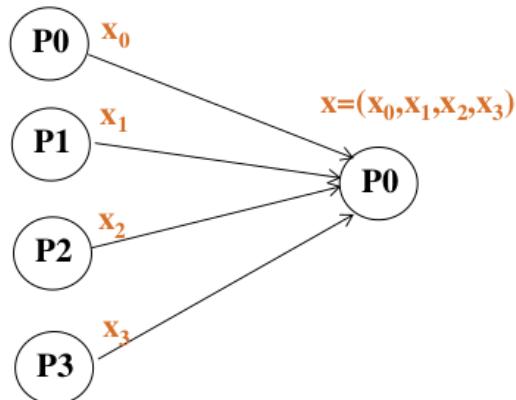
Todos los procesos en el grupo envían un mensaje a un único proceso:

- **Reducción:** los mensajes enviados por los procesos se combinan en un solo mensaje mediante un operador. La operación de combinación es usualmente commutativa y asociativa.
- **Acumulación (*gather*):** los mensajes se reciben de forma concatenada en el receptor (en una estructura vectorial). El orden en la que se concatenan depende del identificador del proceso.

Reducción



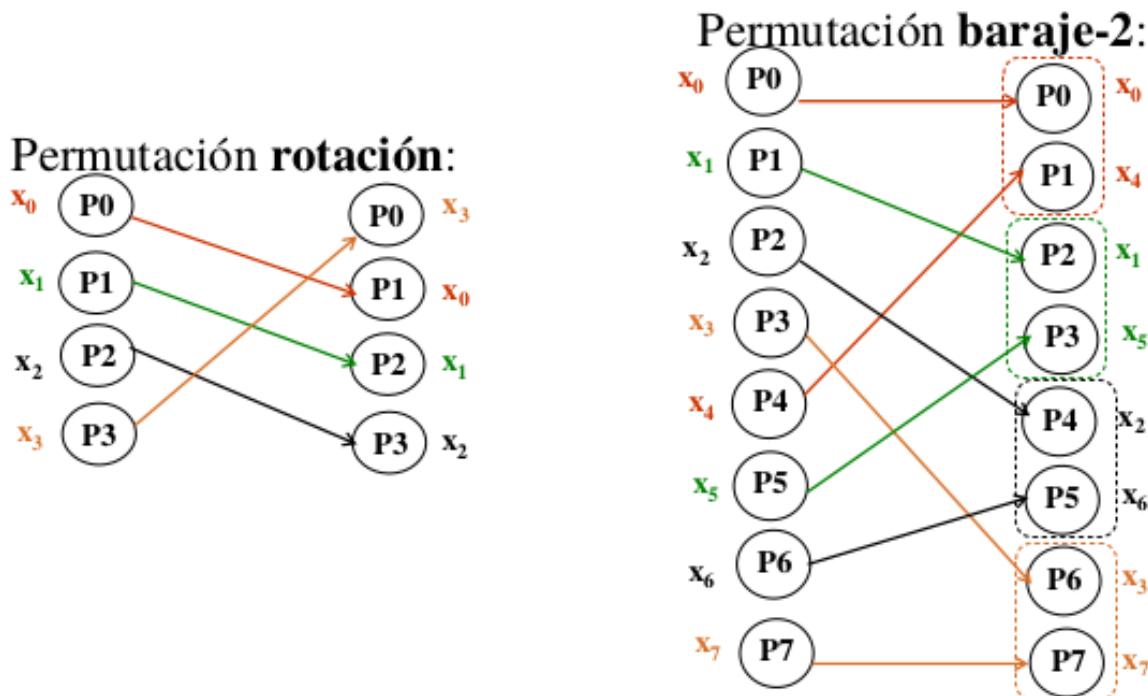
Acumulación (*gather*)



En la reducción, lo que envían todos los procesos se reduce a un único valor, aplicando conmutativa y asociativa. En acumulación se aplican los valores tal cual.

4.2.4 Comunicación múltiple uno-a-uno.

Hay componentes del grupo que envían (escriben) un único mensaje y componentes que reciben (lean) un único mensaje. Si todos los componentes envían y reciben, se implementa una **permutación**.

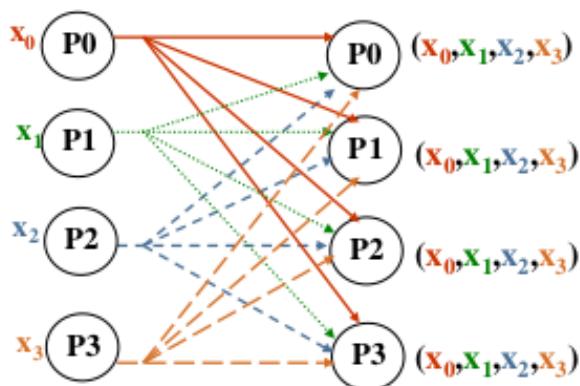


4.2.5 Comunicación todos-a-todos.

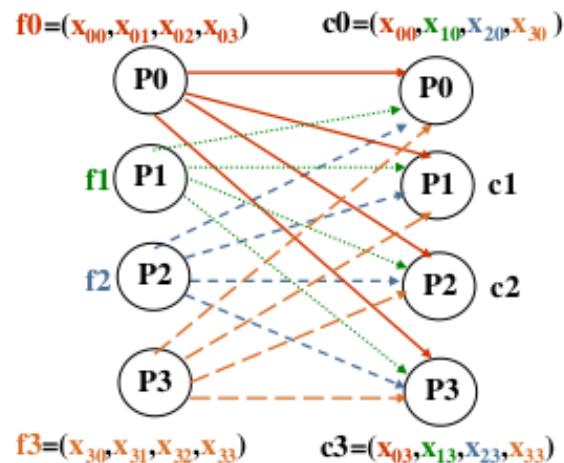
Todos los procesos del grupo ejecutan una comunicación uno-a-todos. Cada proceso recibe n mensajes, cada uno de un proceso diferente del grupo.

- **Todos difunden:** todos los procesos realizan una difusión. Usualmente las n transferencias recibidas por un proceso se concatenan en función del identificador del proceso que envía, de forma que todos los procesos reciben lo mismo.
- **Todos dispersan:** los procesadores concatenan diferentes transferencias. En la figura se ilustra la trasposición de una matriz 4×4 : el procesador P_i dispersa la fila i ($x_{i0}, x_{i1}, x_{i2}, x_{i3}$), tras la ejecución, el procesador P_i tendrá la columna i ($x_{0i}, x_{1i}, x_{2i}, x_{3i}$).

Todos Difunden (*all-broadcast*)
o chismorreo (*gossiping*)



Todos Dispersan (*all-scatter*)



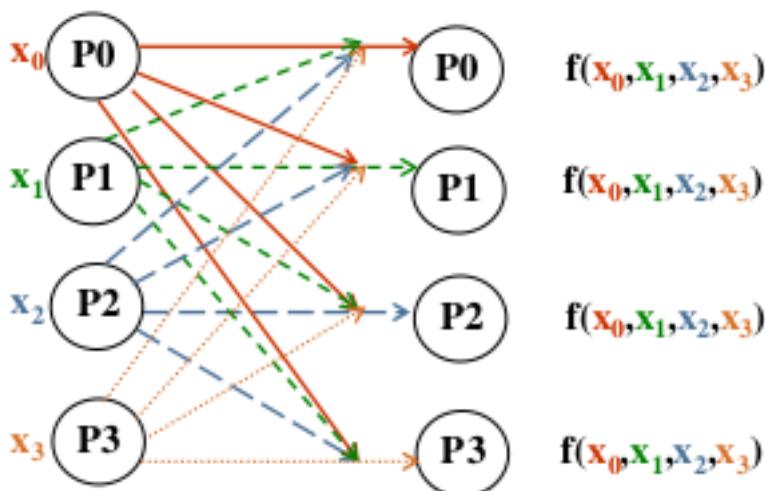
c = columna matriz

f = fila matriz

4.2.6 Servicios compuestos.

- Todos combinan o reducción y extensión: el resultado de aplicar una reducción se obtiene en todos los procesos porque la reducción se difunde una vez obtenida (reducción y extensión) o porque se realizan tantas reducciones como procesos (todos combinan).

Todos combinan



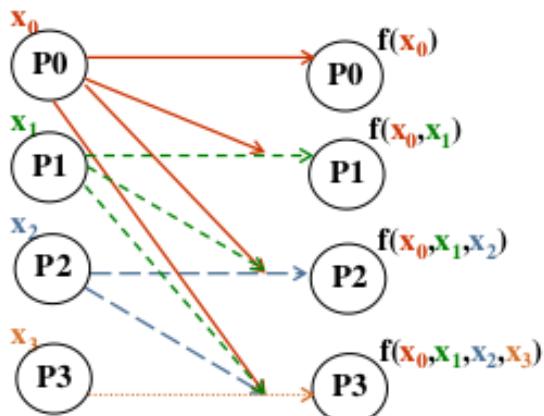
En la desviación típica se haría un todo reduce:

$$s = \sqrt{\frac{\sum_{i=1}^N (x_i - \text{media})^2}{N - 1}}$$

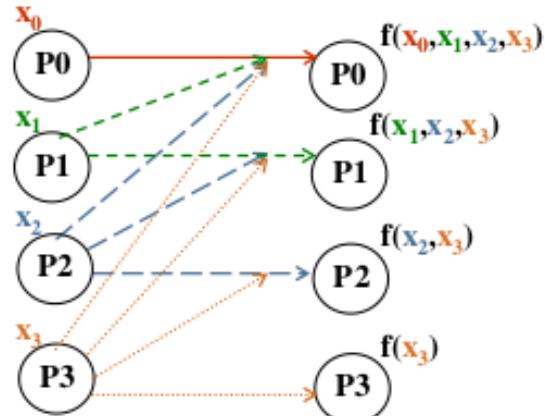
- **Recorrido (scan):** todos los procesos envían un mensaje, recibiendo cada uno de ellos el resultado de reducir un conjunto de estos mensajes.

- **Recorrido prefijo paralelo:** el proceso P_i recibe la reducción de los mensajes P_0, \dots, P_i .
- **Recorrido sufijo paralelo:** recibe la reducción de P_i, \dots, P_{n-1} .

Recorrido (scan) prefijo paralelo



Recorrido sufijo paralelo



Ejemplo: comunicación colectiva en OpenMP.

Uno-a-todos	Difusión (Seminario pract. 2)	<ul style="list-style-type: none"> ✓ Cláusula <code>firstprivate</code> (desde thread 0) ✓ Directiva <code>single</code> con cláusula <code>copyprivate</code> ✓ Directiva <code>threadprivate</code> y uso de cláusula <code>copyin</code> en directiva <code>parallel</code> (desde thread 0)
Todos-a-uno	Reducción (Seminario pract. 2)	<ul style="list-style-type: none"> ✓ Cláusula <code>reduction</code>
Servicios compuestos	Barreras (Seminario pract. 1)	<ul style="list-style-type: none"> ✓ Directiva <code>barrier</code>

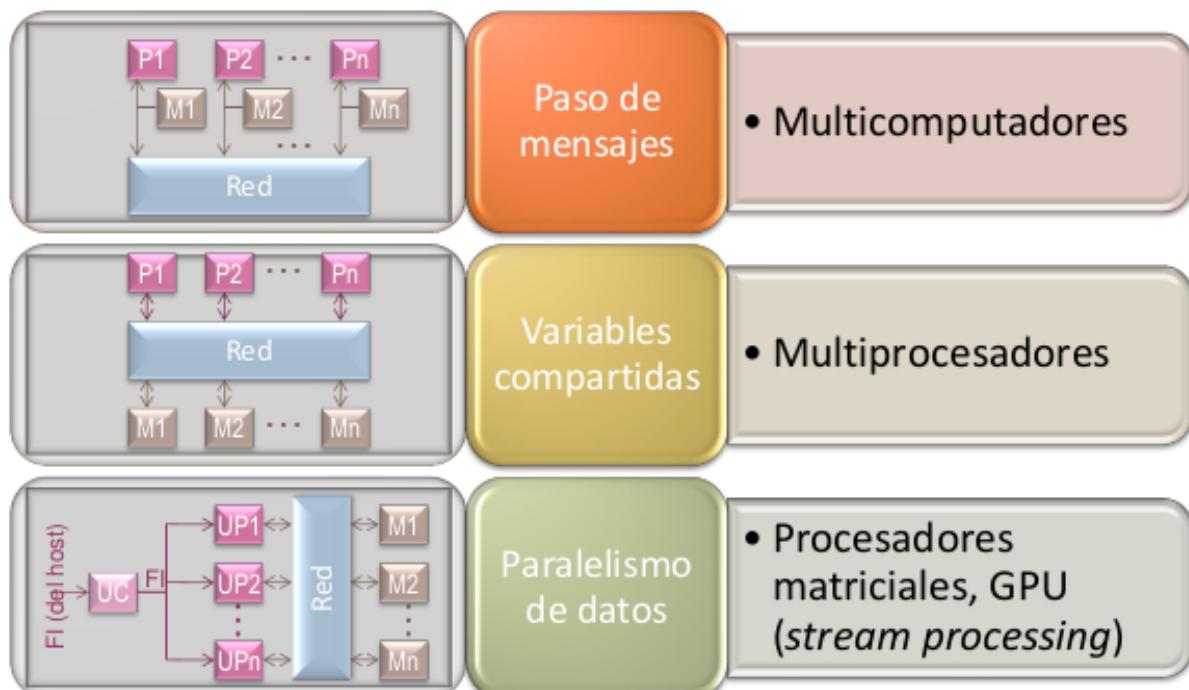
Ejemplo: comunicación en MPI.

Uno-a-uno	Asíncrona	<code>MPI_Send() / MPI_Receive()</code>
Uno-a-todos	Difusión	<code>MPI_Bcast()</code>
	Dispersión	<code>MPI_Scatter()</code>
Todos-a-uno	Reducción	<code>MPI_Reduce()</code>
	Acumulación	<code>MPI_Gather()</code>
Todos-a-todos	Todos difunden	<code>MPI_Allgather()</code>
Servicios compuestos	Todos combinan	<code>MPI_Allreduce()</code>
	Barreras	<code>MPI_Barrier()</code>
	Scan	<code>MPI_Scan</code>

2.1.4 4.3 Estilos/paradigmas de programación paralela.

4.3.1 Estilos de programación y arquitecturas paralelas.

- **Paso de mensajes:** también se puede usar en multiprocesadores.
- **Paralelismo de datos:** se corresponde con la arquitectura SIMD.



4.3.2 Estilos de programación y herramientas de programación.

- **Paso de mensajes** (*message passing*): cada procesador en el sistema tiene su espacio de direcciones propio. Los mensajes llevan datos de uno a otro espacio de direcciones y además se pueden aprovechar para sincronizar procesos. Los datos transferidos estarán duplicados en el sistema de memoria.
 - Lenguajes de programación: Ada, Occam.
 - API (Bibliotecas de funciones): MPI, PVM.
- **Variables compartidas** (*shared memory, shared variables*): se supone que los procesadores en el sistema comparten el mismo espacio de direcciones. Luego no necesitan transferir datos explícitamente, implícitamente se realiza la transferencia utilizando instrucciones del procesador de lectura y escritura en memoria. Para sincronizar, el programador utiliza primitivas que ofrece el software que se amparan en primitivas hardware para incrementar prestaciones.
 - Lenguajes de programación: Ada, Java.
 - API (directivas del compilador + funciones): OpenMP.
 - API (Bibliotecas de funciones): POSIX Threads, shmem, Intel TBB.
- **Paralelismo de datos** (*data parallelism*): las mismas operaciones se ejecutan en paralelo en múltiples unidades de procesamiento de forma que cada unidad aplica la operación a un conjunto de datos distinto. Solo soporta paralelismo a nivel de bucle. La sincronización está implícita. Dispone de construcciones para la distribución de datos entre los elementos de procesamiento.
 - Lenguajes de programación + funciones: HPF (High Performance Fortran), Fortran 95 (forall, operaciones con matrices/vectores), Nvidia CUDA.
 - API (directivas del compilador + funciones - stream processing): OpenACC.

2.1.5 4.4 Estructuras típicas de códigos paralelos.

4.4.1 Estructuras típicas de procesos/threads/tareas.

Estructuras típicas de procesos/threads en código paralelo:

- Descomposición de dominio o descomposición de datos cliente/servidor.
- Divide y vencerás o descomposición recursiva.
- Segmentación o flujo de datos.
- Master-Slave, o granja de tareas.

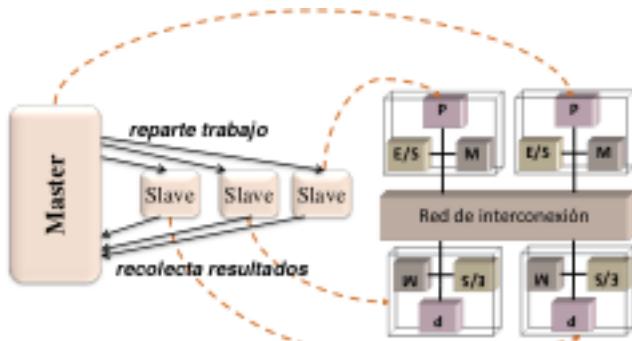
4.4.2 Master-Slave o granja de tareas.

Las tareas se representan con un círculo y los arcos representan flujo de datos.

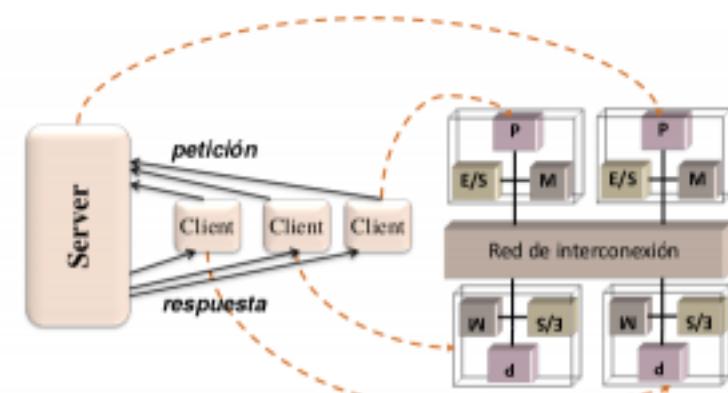
Tenemos un flujo de instrucciones que se encarga de repartir el trabajo entre esclavos y recolecta resultados. Los esclavos están ejecutando el mismo código. El máster puede hacer un trabajo distinto. Luego combinamos un MPMD con SPMD. La distribución de tareas entre los esclavos se puede realizar de forma dinámica o estática.

La escalabilidad del programa paralelo va a depender del número de esclavos y del camino de comunicación entre los esclavos y el dueño. Para incrementar la escalabilidad se puede dividir el dueño en múltiples dueños, cada uno con un conjunto diferente de esclavos.

Se puede llegar a esta estructura al paralelizar las iteraciones de un bucle



4.4.3 Cliente-Servidor.

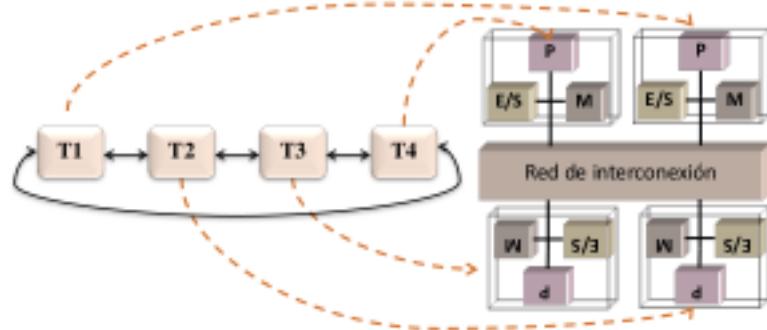


4.4.4 Descomposición de dominio o descomposición de datos.

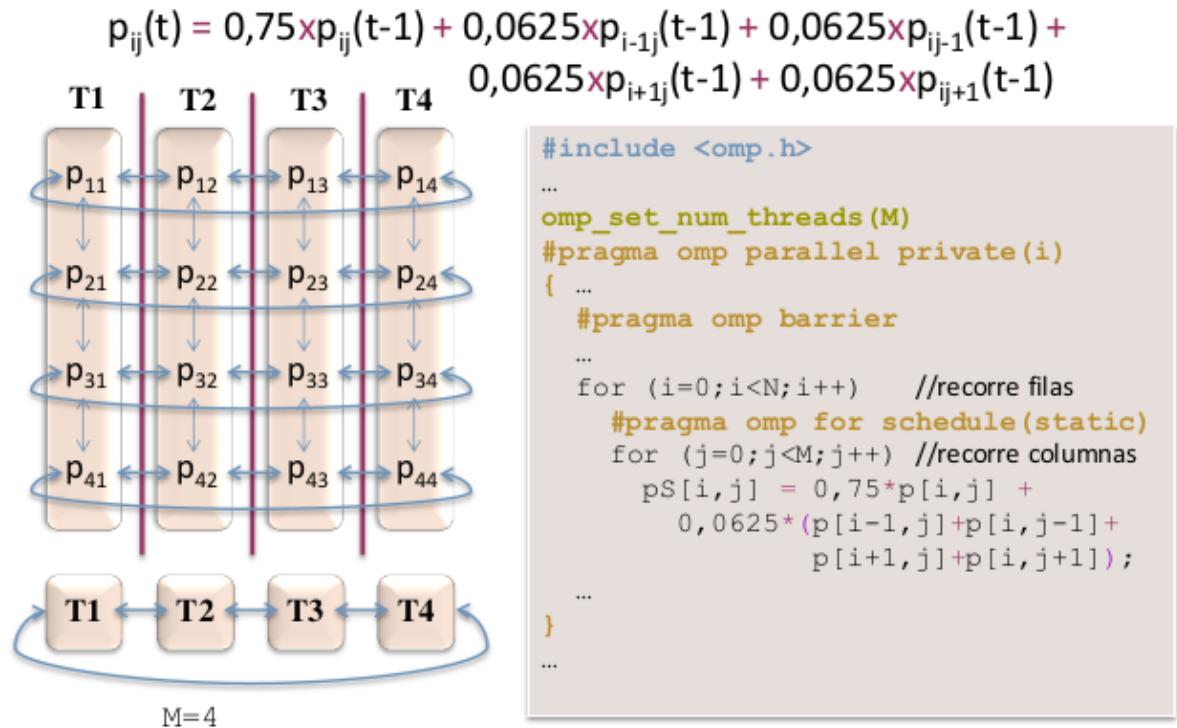
Es muy utilizada en problemas en los que se opera con grandes estructuras de datos. La estructura de datos de entrada o de salida o ambas se dividen en partes y se derivan las tareas paralelas, que realizan operaciones similares.

Los procesos pueden englobar varias tareas. Los diferentes procesos ejecutan típicamente el mismo código (SPMD), aunque cada uno trabaja sobre un conjunto de datos distintos. Puede haber comunicaciones entre los procesos.

- Se pueden representar con matrices
- Aplicación: inundaciones, software metereológico...



Ejemplo: filtrado imagen.



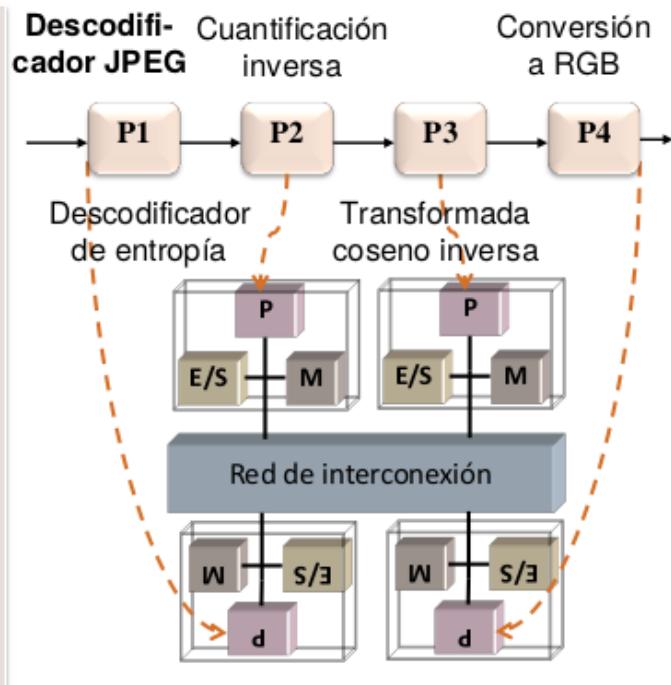
4.4.5 Estructura segmentada o de flujo de datos.

Aparece en problemas en los que se aplica a un flujo de datos en secuencia distintas funciones (paralelismo de tareas). La estructura de los procesos y de las tareas es la de un cauce segmentado. Cada proceso ejecuta por tanto distinto código (MPMD). Necesitamos que en la aplicación se aplique a una un flujo de entrada en secuencia una serie de operaciones, una detrás de otra. Ejemplo: MP3, MP4, multimedia...

En el caso de JPEG, los bloques se dividen en 8x8 bloques y se decodifica en el orden que indiquen las flechas. No puedo aplicar descomposición de dominio porque hay dependencia de bloques.

Podemos parallelizar una sola etapa, cuando se encuentren distintas estructuras en etapas.

```
#include <omp.h>
...
omp_set_num_threads(4);
...
#pragma omp parallel
{
    ...
    for (i=0;i<N;i++) {
        ...
        #pragma omp sections
        {
            #pragma omp section
            P1();
            #pragma omp section
            P2();
            #pragma omp section
            P3();
            #pragma omp section
            P4();
        }
        ...
    }
    ...
}
```



4.4.6 Divide y vencerás o descomposición recursiva.

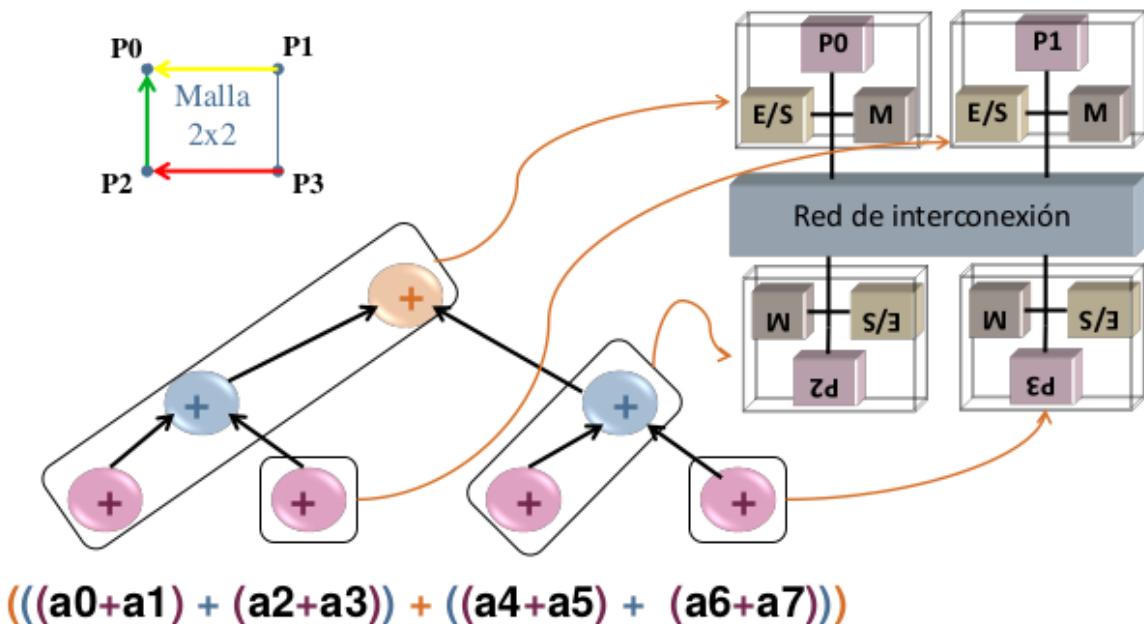
Se utiliza cuando un problema se puede dividir en dos o más subproblemas, de forma que cada uno se puede resolver independientemente, combinándose los resultados para obtener un resultado final.

Las tareas presentan una estructura en forma de árbol. No habrá interacciones entre las tareas que cuelgan del mismo parente. Puede haber paralelismo de tareas y de datos.

Los arcos representan flujo de datos. (flechas negras).

Agrupación/Asignación de tareas a flujos de instrucciones. (flechas negras).

En la imagen, usaríamos 4 flujos de datos como máximo porque el grado de paralelismo es 4. Las flechas naranjas representan la asignación de flujos de instrucciones.



2.2 Lección 5. Proceso de paralelización.

2.2.1 5.1 Objetivos.

- Programar en paralelo una aplicación sencilla.
- Distinguir entre asignación estática y dinámica (ventajas e inconvenientes).

2.2.2 5.2 Proceso de paralelización.

Los arcos en el grafo representan flujo de datos y de control, y los vértices, tareas.

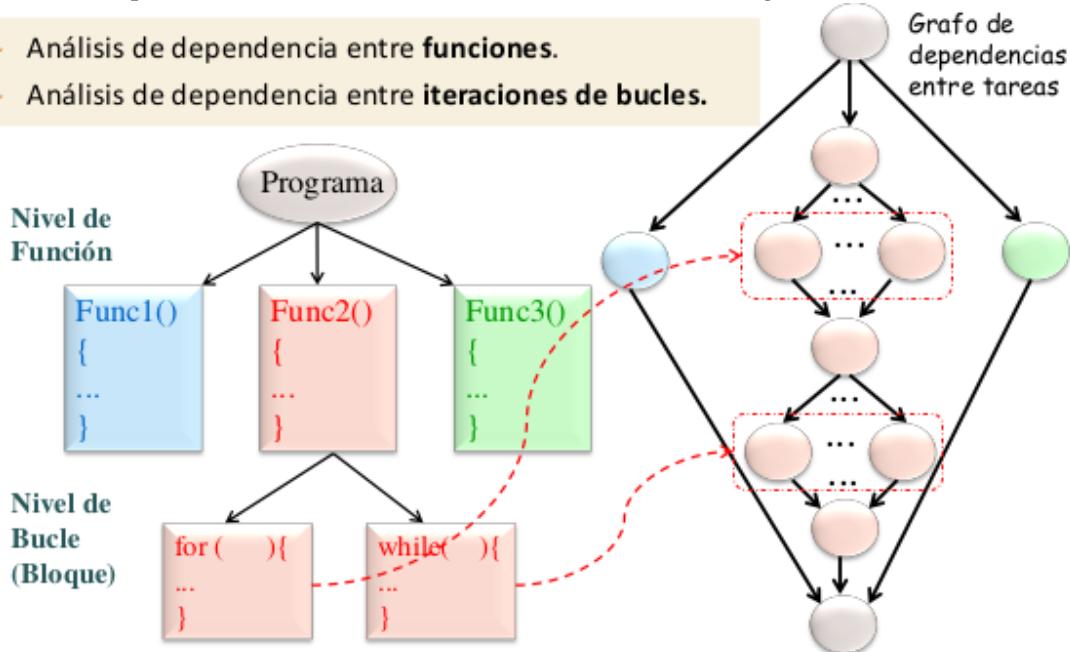
▪ Descomponer en tareas independientes.

- Análisis de dependencia entre **funciones**. Para extraer el paralelismo de tareas, tenemos que analizar las dependencias entre las funciones que pueden ser independientes o pueden hacerse independientes.
- Análisis de dependencia entre **iteraciones de bucles**. Analizando las iteraciones de los bucles dentro de una función, podemos encontrar si son o se pueden hacer independientes. Podemos detectar paralelismo de datos. Si hay varios bucles, se puede analizar la dependencia entre ellos para ver si se pueden ejecutar en paralelo las iteraciones de múltiples bucles.

En la siguiente imagen podemos ver la descomposición en tareas. Una de las funciones consta de dos bucles. Dado el grafo de dependencias entre tareas de la figura, se ha encontrado que son independientes las iteraciones del `for`, cada iteración es una tarea en el grafo. También son

independientes las iteraciones de `while`. Además, el grafo muestra que la salida del bucle `for` se usa en `while`, por eso las tareas de ambos ciclos se encuentran en el grafo a distintos nivel.

- Análisis de dependencia entre **funciones**.
- Análisis de dependencia entre **iteraciones de bucles**.

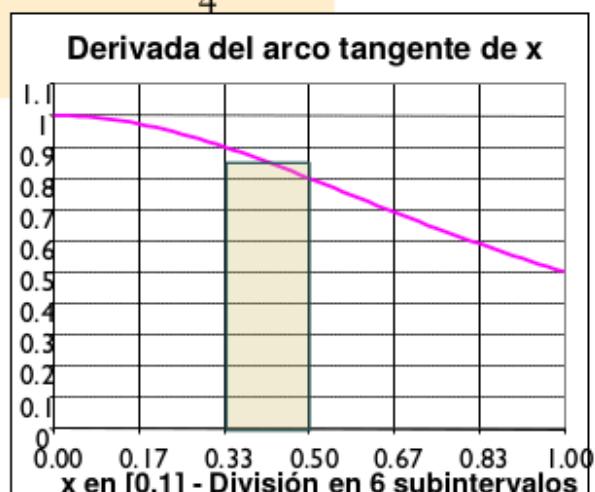


- Ejemplo de cálculo PI: descomposición en tareas independientes.

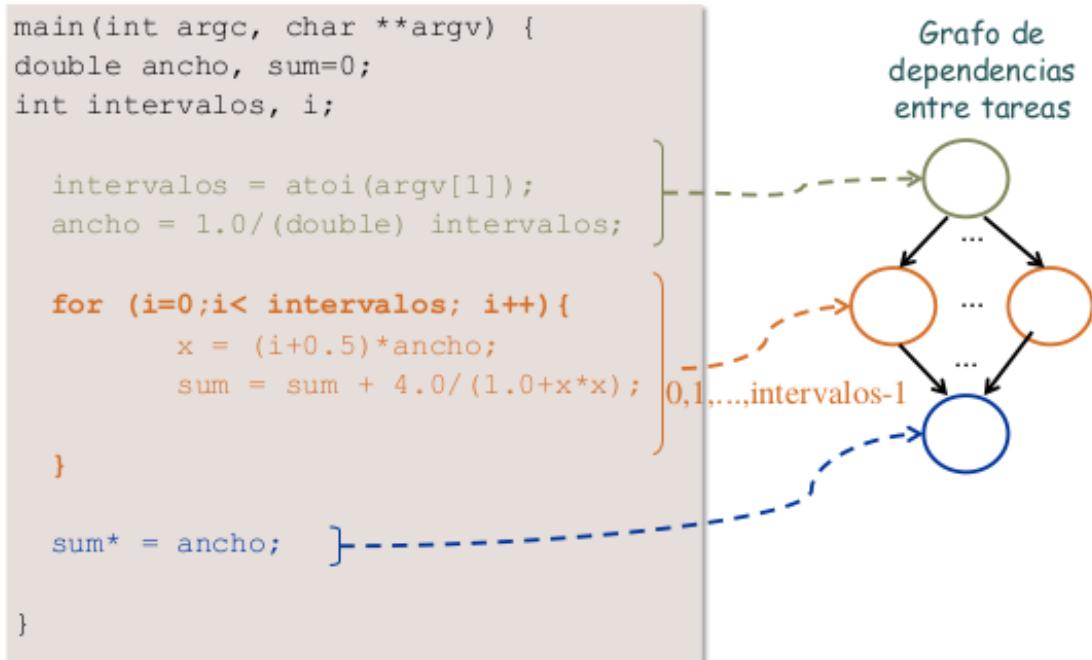
Se puede parallelizar pi fácilmente. Como la integral en el intervalo $[0,1]$ de la derivada del arco tangente de x es $\pi/4$:

$$\left. \begin{array}{l} \arctg'(x) = \frac{1}{1+x^2} \\ \arctg(1) = \frac{\pi}{4} \\ \arctg(0) = 0 \end{array} \right\} \Rightarrow \int_0^1 \frac{1}{1+x^2} dx = \arctg(x)|_0^1 = \frac{\pi}{4} - 0$$

- PI se puede calcular por integración numérica.

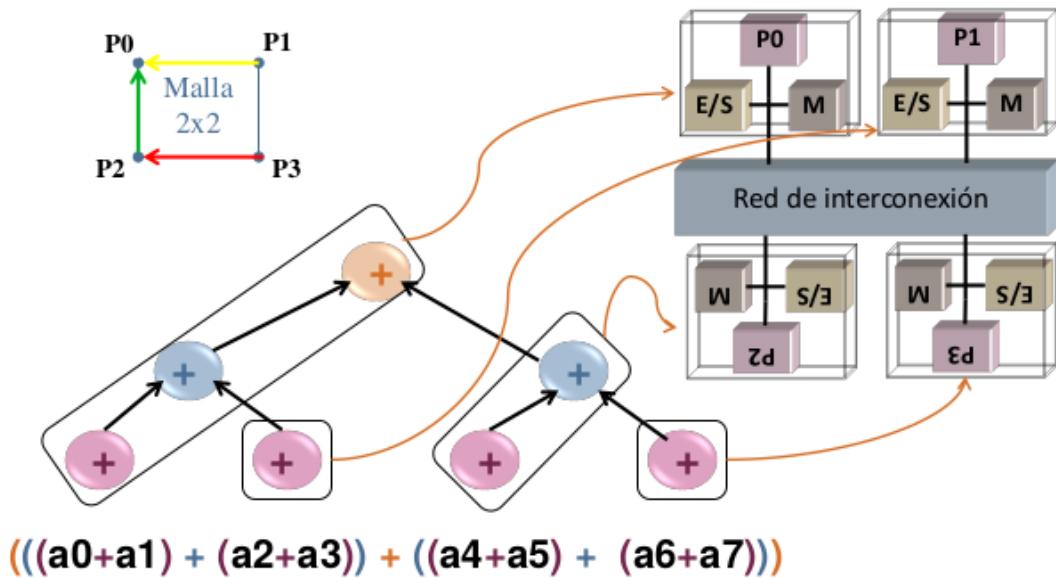


En la siguiente imagen se calcula π con una versión secuencial approximando el área en cada subintervalo utilizando rectángulos en los que la altura es el valor de la derivada del arco tangente en el punto medio.



- **Asignar (planificar+mapear) tareas a procesos y/o threads.** La asignación a procesos o hebras está ligada con la asignación a procesadores, incluso se puede realizar la asignación asociando los procesos (hebras) a procesadores concretos.

- Ejemplo: filtrado de imagen.



- Incluimos: agrupación de tareas en procesos/threads (scheduling) y mapeo a procesadores/cores (mapping).
- La **granularidad** de la carga de trabajo (tareas) asignada a los procesos/threads depende de:
 - número de cores o procesadores o elementos de procesamiento, cuanto mayor sea su número, menos tareas se asignarán a cada proceso (hebra).
 - tiempo de comunicación/sincronización frente a tiempo de cálculo.

- Para disminuir este tiempo, se pueden asignar más tareas a un proceso (hebra), así se reduce el número de interacciones entre tareas a través de la red.
- ¿Utilizar hebras o procesos? Depende de
 - **Arquitectura:**
 - ◊ Es más eficiente usar hebras en SMP y procesadores multihebra.
 - ◊ En arquitecturas mixtas se usan hebras y procesos, especialmente si el número de procesadores de un SMP es mayor que el número de nodos de un cluster.
 - **Sistema operativo:** debe ser multihebra.
 - **Herramientas de programación** para crear hebras y procesos.
- Se asignan hebras a las iteraciones de un **bucle** (paralelismo de datos).
- Se asignan procesos a **funciones** (paralelismo de tareas).
- **Equilibrado de la carga** (tareas = código + datos) o load balancing:
 - Objetivo: unos procesos/threads no deben esperar a otros.
- ¿De qué depende el equilibrado?
 - **La arquitectura:**
 - ◊ homogénea frente a la heterogénea.
 - ◊ uniforme frente a no uniforme.
 - La aplicación/descomposición.
- Tipos de asignación:
 - **Estática.**
 - ◊ Está determinado qué tarea va a realizar cada procesador o core.
 - ◊ Explícita: programador.
 - ◊ Implícita: herramienta de programación al generar el código ejecutable.
 - **Dinámica** (en tiempo de ejecución).
 - ◊ Permite que acabe una aplicación aunque falle algún procesador.
 - ◊ Consumo un tiempo adicional de comunicación y sincronización.
 - ◊ Distintas ejecuciones pueden asignar distintas tareas a un procesador o core.
 - ◊ Explícita: el programador.
 - ◊ Implícita: herramienta de programación al generar el código ejecutable.
- Mapeo de procesos/threads a unidades de procesamiento.
 - Normalmente se deja al SO el mapeo de threads (light process).
 - Lo puede hacer el entorno o sistema en tiempo de ejecución (runtime system de la herramienta de programación).
 - El programador puede influir.

- Códigos filtrado por imagen.

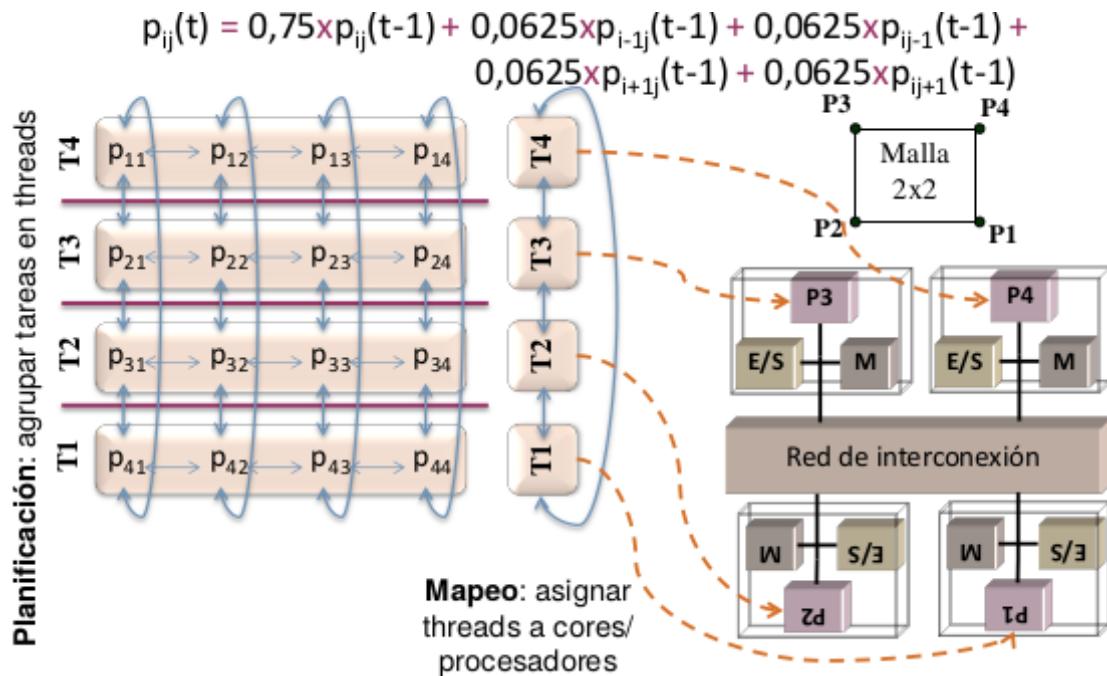
Descomposición por **columnas**.

```
1 #include <omp.h>
2 ...
3 omp_set_num_threads(M)
4 #pragma omp parallel private(i)
5 {
6     for (i=0;i<N;i++) {
7         #pragma omp for
8         for (j=0;j<M;j++) {
9             pS[i,j] = 0,75*p[i,j] + 0,0625*(p[i-1,j]+p[i,j-1]
10                + p[i+1,j]+ p[i,j+1]);
11         }
12     }
13 }
14 ...
```

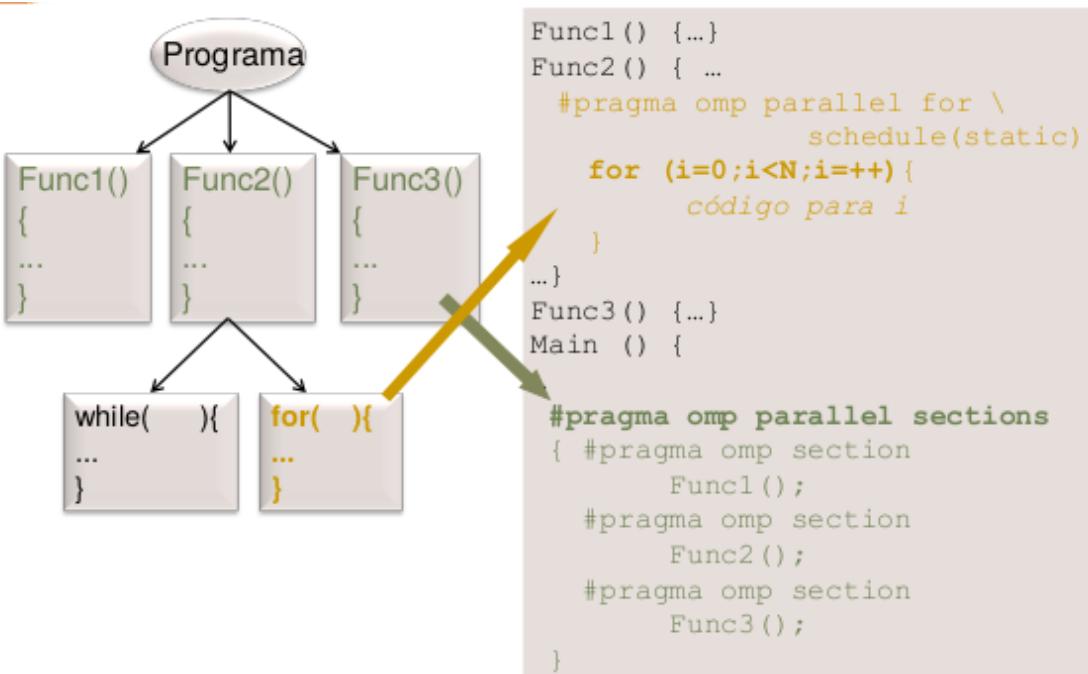
Descomposición por **filas**.

```
1 #include <omp.h>
2 ...
3 omp_set_num_threads(N)
4 #pragma omp parallel private(j)
5 {
6     #pragma omp for
7     for (i=0;i<N;i++) {
8         for (j=0;j<M;j++) {
9             pS[i,j] = 0,75*p[i,j] + 0,0625*(p[i-1,j]+p[i,j-1]+ p[i
10                +1,j]+ p[i,j+1]);
11         }
12     }
13 ...
```

- Ejemplo: filtrado de imagen.



- Ejemplo de asignación estática del parallelismo de tareas y datos con OpenMP.

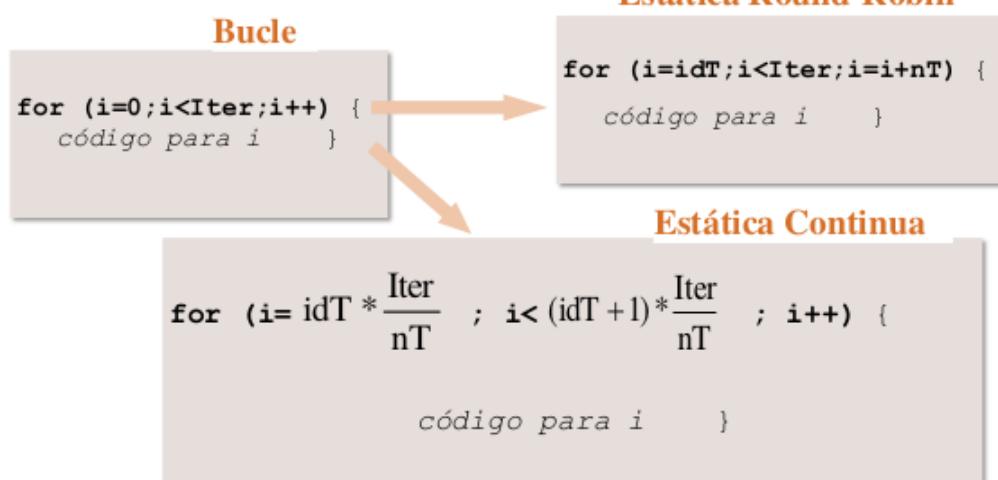


- Asignación estática.

- Asignación estática y explícita de las iteraciones de un bucle.

En la siguiente imagen se pueden ver dos alternativas que el programador puede utilizar explícitamente para asignar las iteraciones de un bucle a procesos (hebras) de forma estática. En la asignación turno rotatorio (*round-robin*), iteraciones consecutivas del bucle se asignan a procesos consecutivos (con identificador consecutivo). En el otro ejemplo se asignan iteraciones consecutivas al mismo proceso.

➤ Asignación **estática** y explícita de las iteraciones de un bucle:

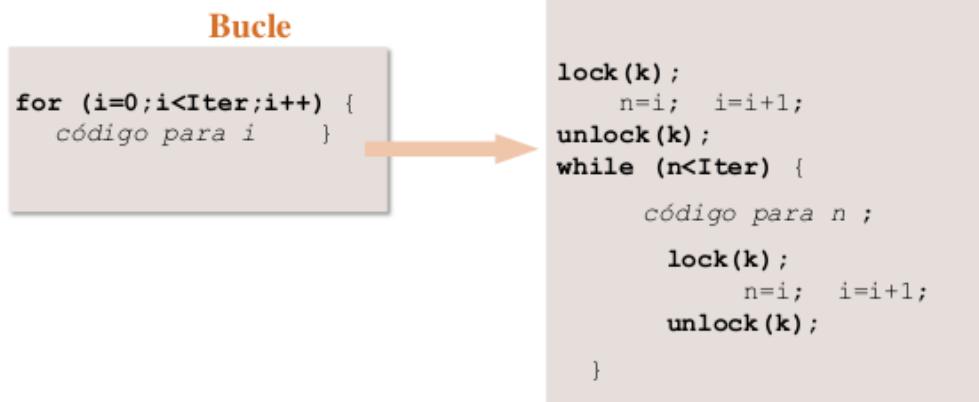


- Asignación dinámica.

- Asignación dinámica y explícita de las iteraciones de un bucle.

En la siguiente imagen se ve cómo el programador puede implementar explícitamente en el código una asignación dinámica para memoria compartida. Las iteraciones se reparten en orden. La variable `i` es compartida. Los procesos (hebras) consultan la variable `i` para “coger” la siguiente iteración que van a realizar, e incrementan su valor en uno para que el siguiente proceso no coja una iteración ya asignada. Para que una iteración no se ejecute dos veces, se utiliza un cerrojo `k` para excluir la lectura y modificación. Esquema: dueño-esclavo.

➤ Asignación **dinámica** y explícita de las iteraciones de un bucle:



➤ Dinámica e implícita

Ej. con OpenMP

- Dinámica e implícita.

- Asignación. Ejemplo: multiplicación matriz por vector.

$$c = A \bullet b; \quad c_i = \sum_{k=0}^{M-1} a_{ik} \bullet b_k = a_i^T \bullet b, \quad c(i) = \sum_{k=0}^{M-1} A(i,k) \bullet b(k), \quad i = 0, \dots, N-1$$

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}
a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}
a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}
a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}

•

b_1
b_2
b_3
b_4
b_5
b_6

 $\text{N}=8$
 $M=6$
 $=$

c_1
c_2
c_3
c_4
c_5
c_6
c_7
c_8

 0
 1
 2
 3

$c(1) = \sum_{k=0}^{M-1} A(1,k) \bullet b(k)$
 $c(2) = \sum_{k=0}^{M-1} A(2,k) \bullet b(k)$
 $c(3) = \sum_{k=0}^{M-1} A(3,k) \bullet b(k)$
 $c(4) = \sum_{k=0}^{M-1} A(4,k) \bullet b(k)$
 $c(5) = \sum_{k=0}^{M-1} A(5,k) \bullet b(k)$
 $c(6) = \sum_{k=0}^{M-1} A(6,k) \bullet b(k)$
 $c(7) = \sum_{k=0}^{M-1} A(7,k) \bullet b(k)$
 $c(8) = \sum_{k=0}^{M-1} A(8,k) \bullet b(k)$

■ Redactar código paralelo.

- El código depende de:
 - Estilo de programación (variables compartidas, paralelismo...).
 - Modo de programación (SPMD, MPMD...).
 - Punto de partida.
 - Herramienta software para el paralelismo.
 - Estructura.
- Se añaden o utilizan en el programa las funciones, directivas o construcciones del lenguaje que hagan falta para:
 - Crear y terminar procesos (hebras). Si se crean de forma estática, enrolar o desenrolar procesos en el grupo que va a cooperar en el cálculo.
 - Localizar paralelismo.
 - Asignar la carga de trabajo.
 - Comunicar y sincronizar.
- Ejemplo: cálculo de PI con OpenMP/C.

Con la cláusula de planificación, estamos haciendo la carga dinámica.

```

#include <omp.h>
#define NUM_THREADS 4
main(int argc, char **argv) {
    long double ancho,x, sum=0; int intervalos, i;
    intervalos = atoi(argv[1]);
    ancho = 1.0/(double) intervalos;
    omp_set_num_threads(NUM_THREADS); →Crear/Terminar
    #pragma omp parallel →Comunicar/sincronizar
    {
        #pragma omp for reduction(+:sum) private(x) \
                    schedule(dynamic) →Agrupar/Asignar
        for (i=0;i< intervalos; i++) {
            x = (i+0.5)*ancho; sum = sum + 4.0/(1.0+x*x);
        }
    }
    sum* = ancho;
}

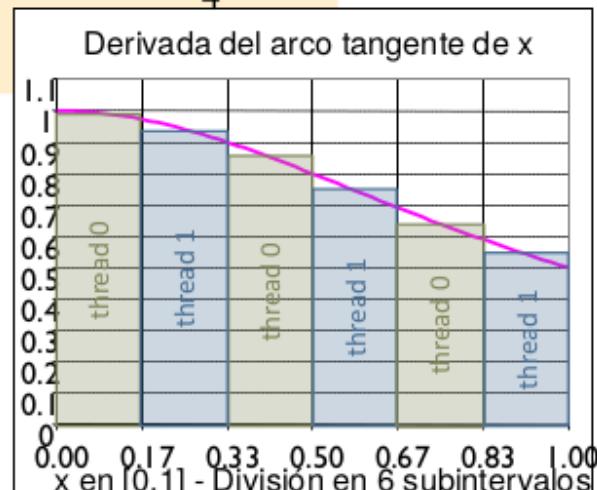
```

Localizar

- Asignación de tareas a 2 threads estática por turno rotatorio.

$$\left. \begin{array}{l} \arctg'(x) = \frac{1}{1+x^2} \\ \arctg(1) = \frac{\pi}{4} \\ \arctg(0) = 0 \end{array} \right\} \Rightarrow \int_0^1 \frac{1}{1+x^2} = \arctg(x)|_0^1 = \frac{\pi}{4} - 0$$

- PI se puede calcular por integración numérica.



- Ejemplo: cálculo de PI en MPI/C.
 - `MPI_Init()`: enrola el proceso que lo ejecuta dentro del mundo MPI, es decir, dentro del grupo de procesos denominado `MPI_COMM_WORLD`.
 - `MPI_Finalize()`: se debe llamar antes de que un proceso enrolado en MPI acabe su ejecución.
 - `MPI_Comm_size(MPI_COMM_WORLD, &nproc)`: pregunta a MPI el número de procesos enrolados en el grupo, se devuelve en `nproc`.
 - `MPI_Comm_rank(MPI_COMM_WORLD, &iproc)`: se devuelve al proceso su identificador, `iproc`, dentro del grupo.

```

#include <mpi.h>
main(int argc, char **argv) {
    double ancho,x,lsum,sum; int intervalos,i,nproc,iproc;
    if(MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc); →Enrolar
    MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
    intervalos=atoi(argv[1]); →Localizar-Agrupar
    ancho=1.0/(double) intervalos; lsum=0;
    for (i=iproc; i<intervalos; i+=nproc) {
        x = (i+0.5)*ancho; lsum+= 4.0/(1.0+x*x);
    }
    lsum*= ancho; →Comunicar/sincronizar
    MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE,
               MPI_SUM,0,MPI_COMM_WORLD);
    MPI_Finalize(); →Desenrolar
}

```

- Evaluar prestaciones. Para ver la bondad, es decir, para ver si hemos hecho un buen código.

2.3 Lección 6. Evaluación de prestaciones en procesamiento paralelo.

2.3.1 Objetivos.

- Obtener ganancia y escalabilidad en el contexto de procesamiento paralelo
- Aplicar la ley de Amdahl en el contexto de procesamiento paralelo
- Comparar la ley de Amdahl y ganancia escalable.

2.3.2 6.1 Ganancia de prestaciones y escalabilidad.

6.1.1 Evaluación de prestaciones.

- **Medidas usuales.**
 - Tiempo de respuesta.
 - Real (wall-clock time, elapsed time) (/usr/bin/time).
 - Usuario, sistema, CPU time = user + sys.
 - Productividad.
- **Escalabilidad.** Evolución del incremento (ganancia) en prestaciones (tiempo de respuesta o productividad) que se consigue en el sistema conforme se añaden recursos.
- **Eficiencia.**
 - Relación prestaciones/prestaciones máximas.
 - Rendimiento = prestaciones/no_recursos.
 - Otras: Prestaciones/consumo_potencia, prestaciones/área_ocupada.

6.1.2 Ganancia en prestaciones. Escalabilidad.

Ganancia de prestaciones:

$$S(p) = \frac{\text{Prestaciones}(p)}{\text{Prestaciones}(1)} = \frac{T_S}{T_P(p)}$$

$$T_p(p) = T_C(p) + T_O(p)$$

$\text{Prestaciones}(p)$: prestaciones para la aplicación en el sistema multiprocesador con p procesadores.

$\text{Prestaciones}(1)$: prestaciones obtenidas ejecutando la versión secuencial en un sistema uniprocesador.

T_S : tiempo de ejecución (respuesta) del programa secuencial. Para obtener T_S se debería escoger el mejor programa secuencial para la aplicación.

$T_P(p)$: tiempo de ejecución del programa paralelo con p procesadores.

$T_O(p)$: tiempo de penalización.

Ganancia en velocidad (Speedup)

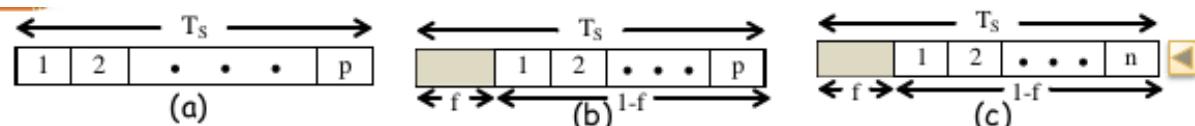
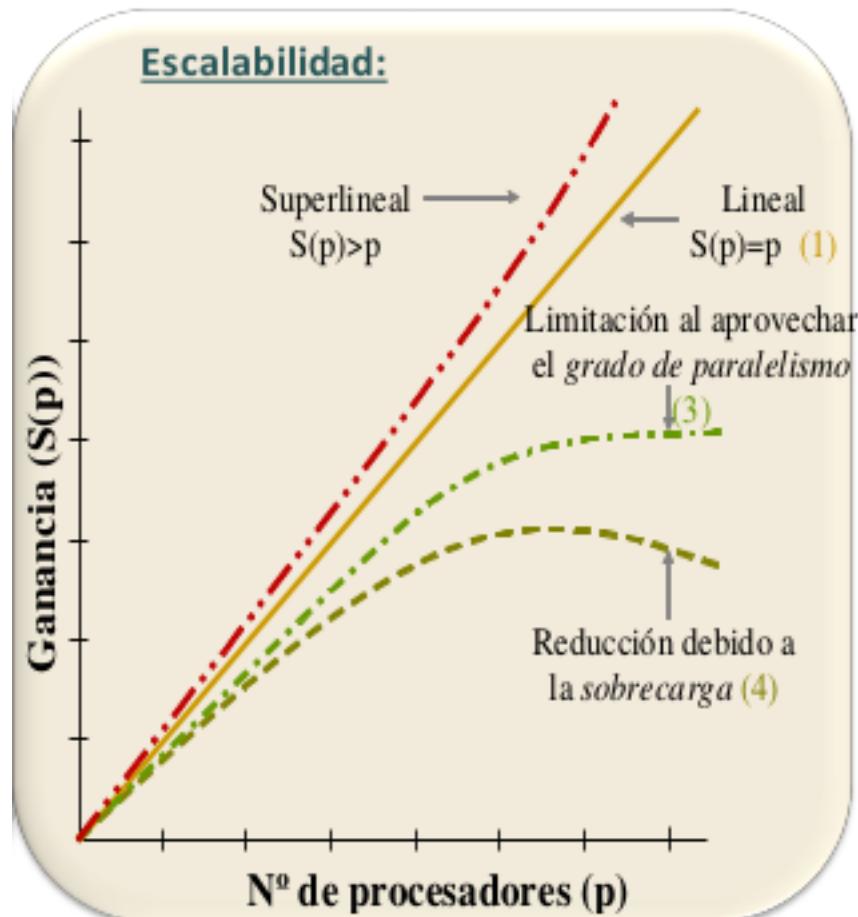
- Ganancia máxima de la eficiencia = 1
- Ganancia mínima de la eficiencia = $1/p$

$$T_p(p) = \frac{T_s}{p} \rightarrow S(p) = T_s / T_p(p) = \frac{T_s}{T_s/p} = p$$

- Sobrecarga (*overhead*):
 - Comunicación/sincronización.
 - Crear/terminar procesos/threads.
 - Cálculos o funciones no presentes en versión secuencial.
 - Falta de equilibrado.

$$E(p) = \frac{\text{Prest}(p)}{\text{PrestMax}(p)} = \frac{\text{Prest}(p)}{p \cdot \text{Prest}(1)} = \frac{S(p)}{p}$$

En la siguiente imagen, se mantiene constante el tamaño del problema (T_S).



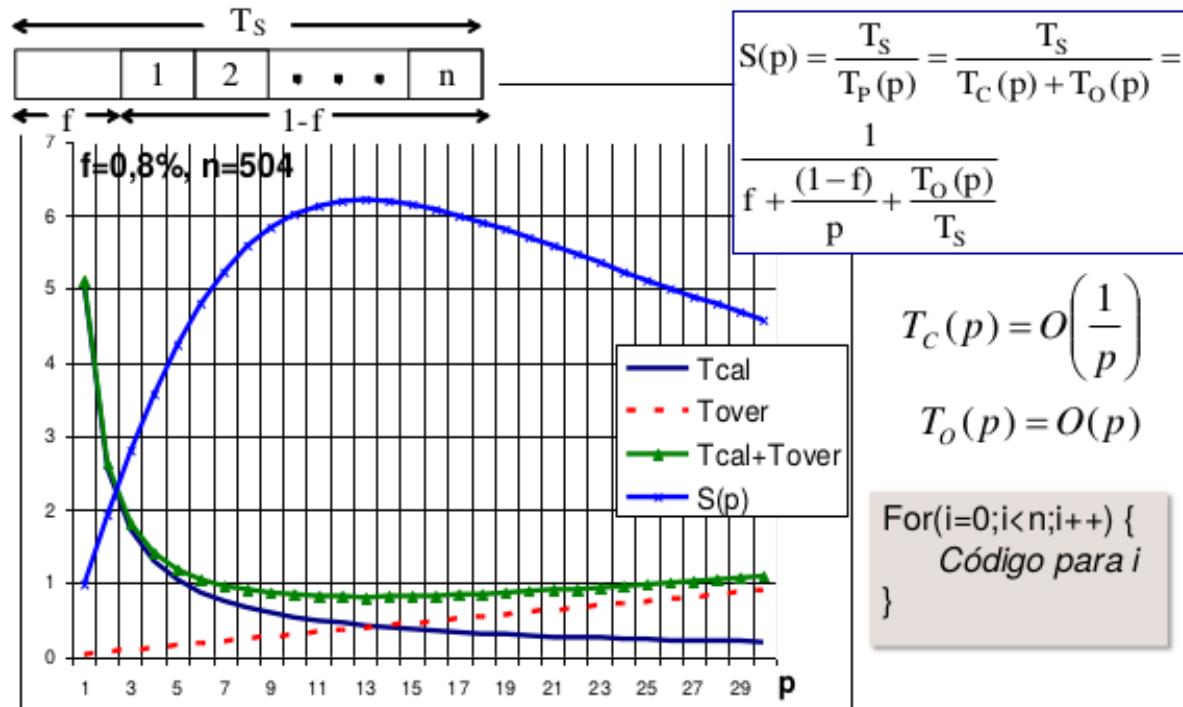
Modelo código	Fracción no paral. en T_S	Grado paralelismo	Overhead	Ganancia en función del número de procesadores p con T_S constante
(a)	0	ilimitado	0	$S(p) = \frac{T_S}{T_p(p)} = p$ Ganancia lineal (1)
(b)	f	ilimitado	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p \rightarrow \infty} \frac{1}{f}$ (2)
(c)	f	n	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p=n} \frac{1}{f + \frac{(1-f)}{n}}$ (3)
(b)	f	ilimitado	Incrementa linealmente con p	$S(p) = \frac{1}{f + \frac{(1-f)}{p} + \frac{T_O(p)}{T_S}} \xrightarrow{p \rightarrow \infty} 0$ (4)

- Número de procesadores óptimo:

$$S(p) = \frac{T_S}{T_P(p)} = \frac{T_S}{T_C(p) + T_O(p)} = \frac{1}{f + \frac{1-f}{p} + \frac{T_O(p)}{T_S}}$$

$$T_C(p) = O\left(\frac{1}{p}\right)$$

$$T_O(p) = O(p)$$

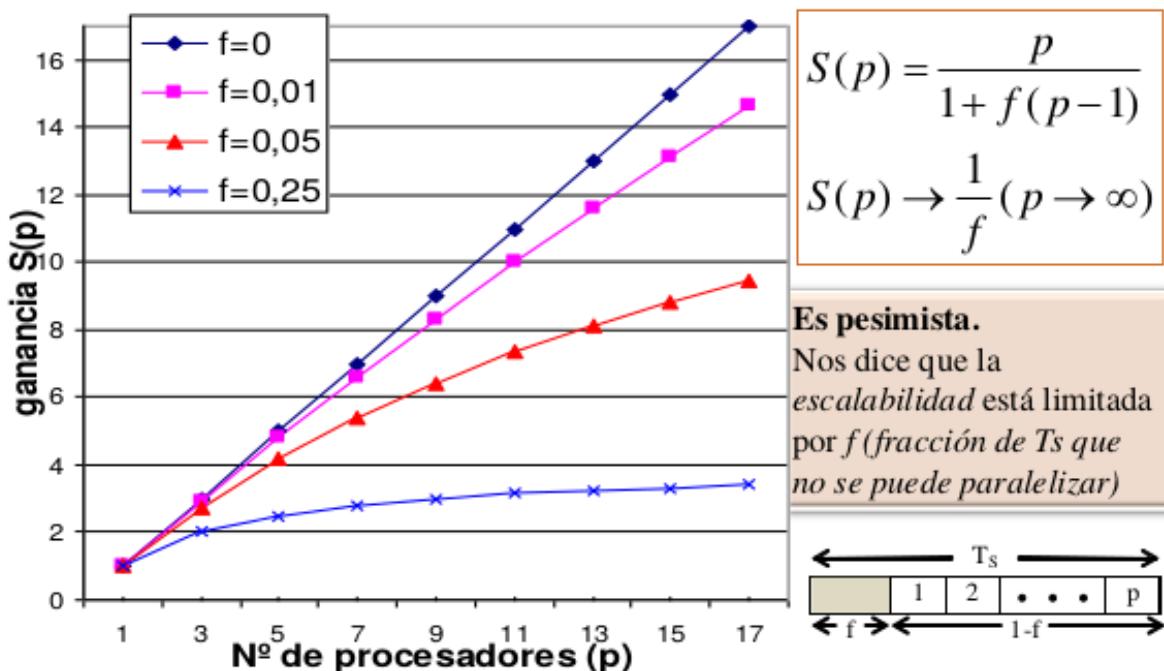


2.3.3 6.2 Ley de Amdahl.

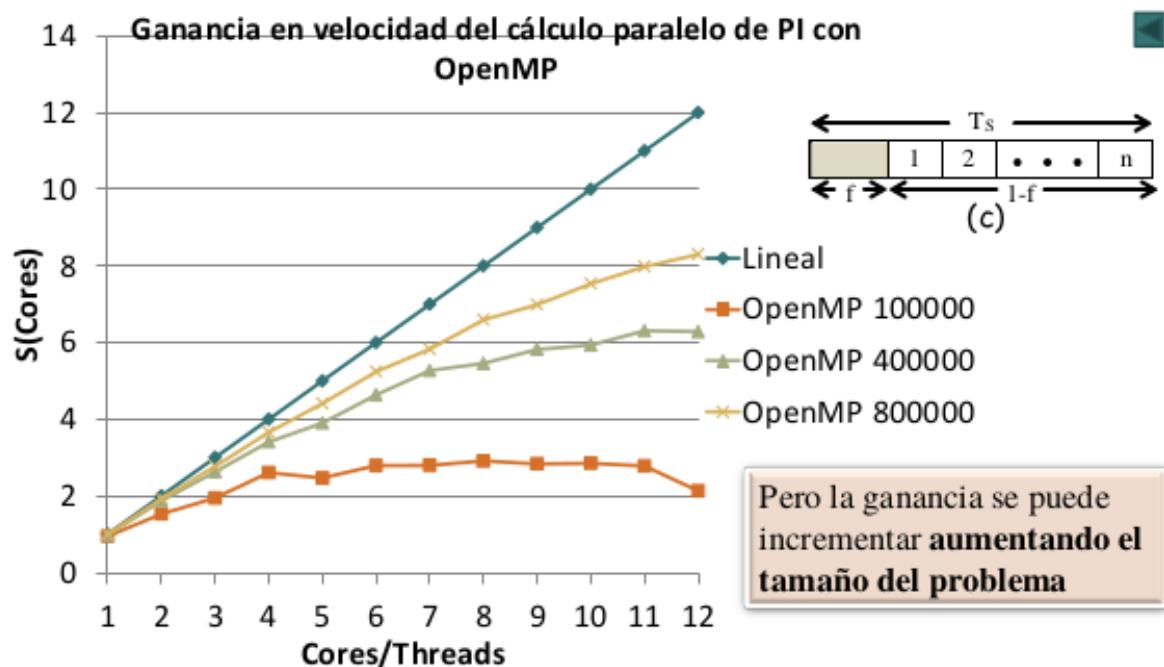
- Ley de Amdahl: la ganancia en prestaciones utilizando p procesadores está limitada por la fracción de código que no se puede parallelizar.

$$S(p) = \frac{T_S}{T_P(p)} \leq \frac{T_S}{f \cdot T_S + \frac{(1-f)T_S}{p}} = \frac{p}{1+f(p-1)} \rightarrow \frac{1}{f} (p \rightarrow \infty)$$

- S : incremento en velocidad que se consigue al aplicar una mejora. (parallelismo)
- p : Incremento en velocidad máximo que se puede conseguir si se aplica la mejora todo el tiempo. (número de procesadores)
- f : fracción de tiempo en el que no se puede aplicar la mejora. (fracción de t. no parallelizable)

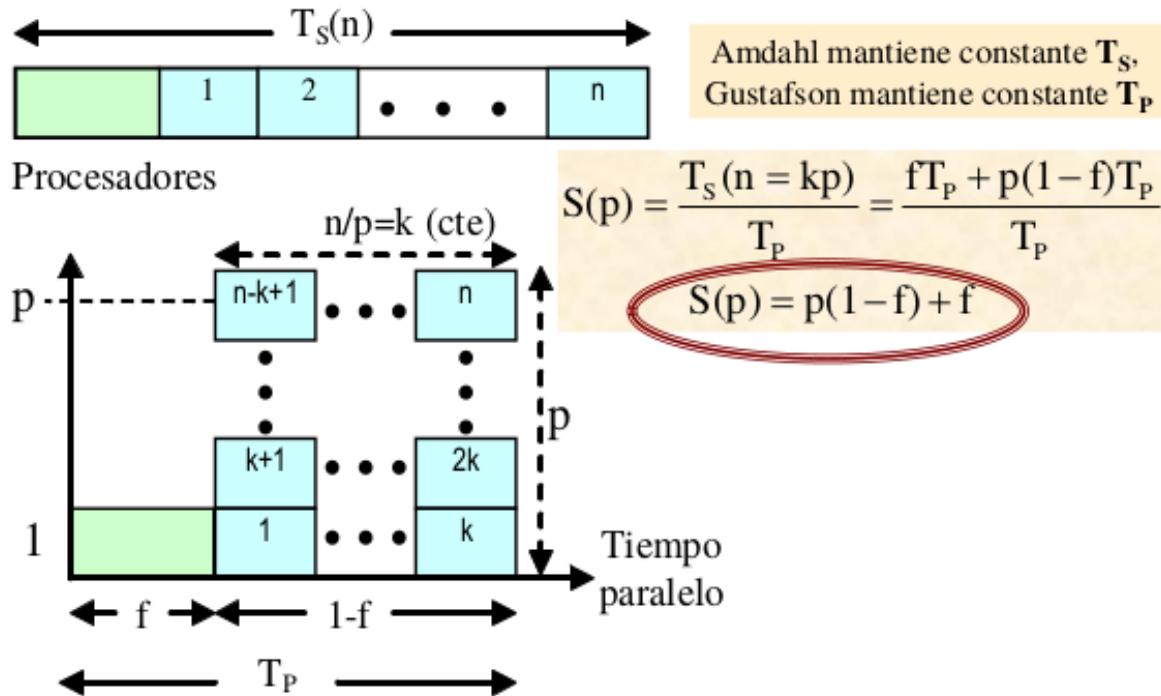


2.3.4 6.3 Ganancia escalable.



En la siguiente imagen, al incrementar el número de procesadores, se añade más trabajo y se incrementa el tamaño del problema (T_S), pero el tiempo paralelo se mantiene constante (T_p).

f : fracción de ejecución paralelo frente a la no paralelizable.



3 Tema 3. Arquitecturas con paralelismo a nivel de thread (TLP)

3.1 Lección 7. Arquitecturas TLP.

3.1.1 Objetivos.

- Distinguir entre cores multithread, multicores y multiprocesadores.
- Comparar entre cores multithread de grano fino, cores multithread de grano grueso y cores con multithread simultánea.

3.1.2 7.1 Clasificación y estructura de arquitecturas con TLP explícito y una instancia del SO.

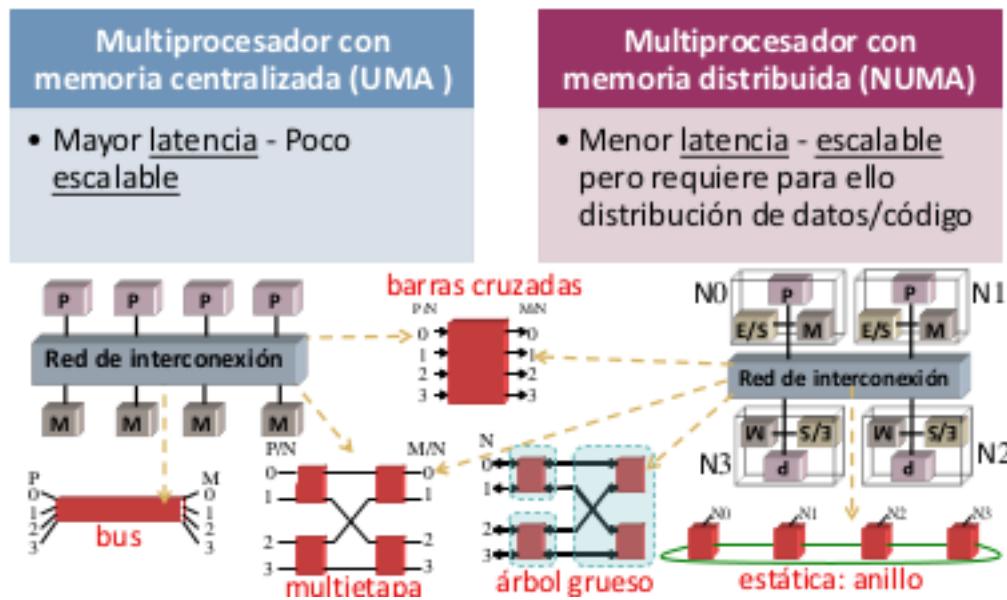
- **Multiprocesador.** Ejecutan varios threads en paralelo en un computador con varios cores/procesadores (cada thread en un core/procesador distinto). Diversos niveles de empaquetamiento: dado, encapsulado, placa, chasis y sistema.
- **Multicore o multiprocesador en un chip o CMP (Chip MultiProcessor).** Ejecutan varios threads en paralelo en un chip de procesamiento multicore (cada thread en un core distinto).
- **Core multithread.** Core que modifica su arquitectura ILP para ejecutar threads concurrentemente o en paralelo.

3.1.3 7.2 Multiprocesadores.

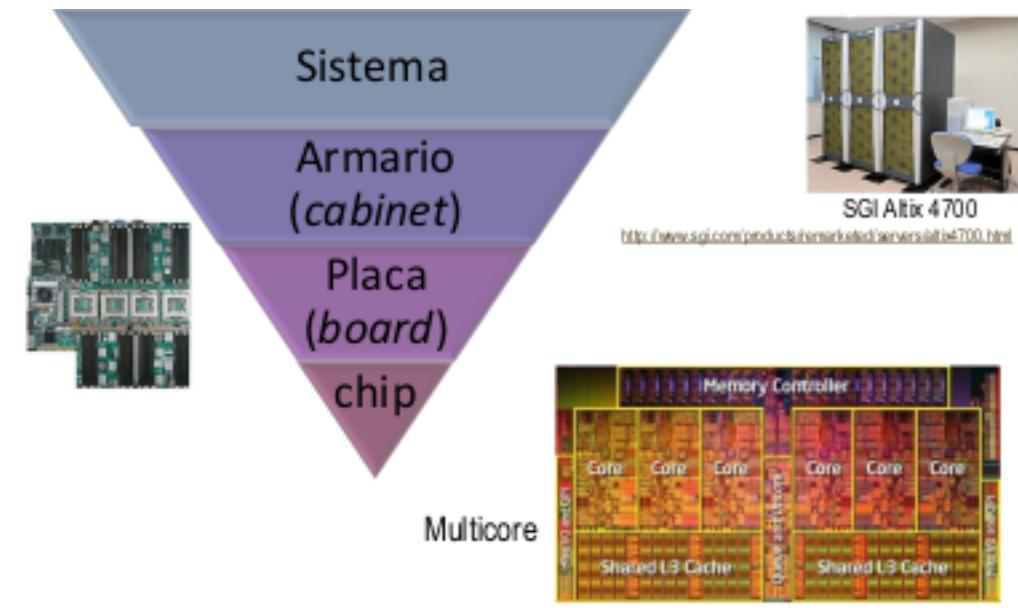
Ejecutan varios threads en paralelo en un computador con varios cores/procesadores (cada thread en un core/procesador distinto).

7.2.1 Criterio clasificación: sistema de memoria.

- **Multiprocesador con memoria centralizada (UMA).** Mayor latencia y poco escalable.
- **Multiprocesador con memoria distribuida (NUMA).** Menor latencia y escalable, pero requiere distribución de datos/código.



7.2.2 Criterio de clasificación: nivel de empaquet./conexión.



7.2.3 Multiprocesador en una placa: evolución de UMA a NUMA.

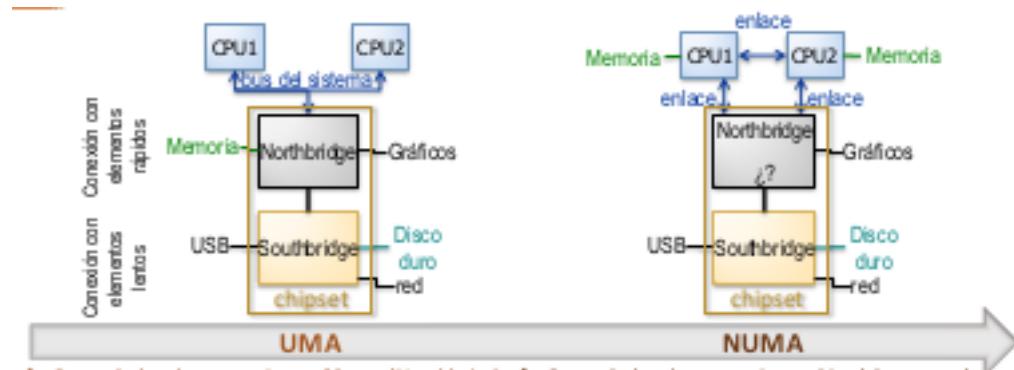
■ UMA.

- Controlador de memoria en chipset (*Northbridge chip*).
- Red: bus (medio compartido).

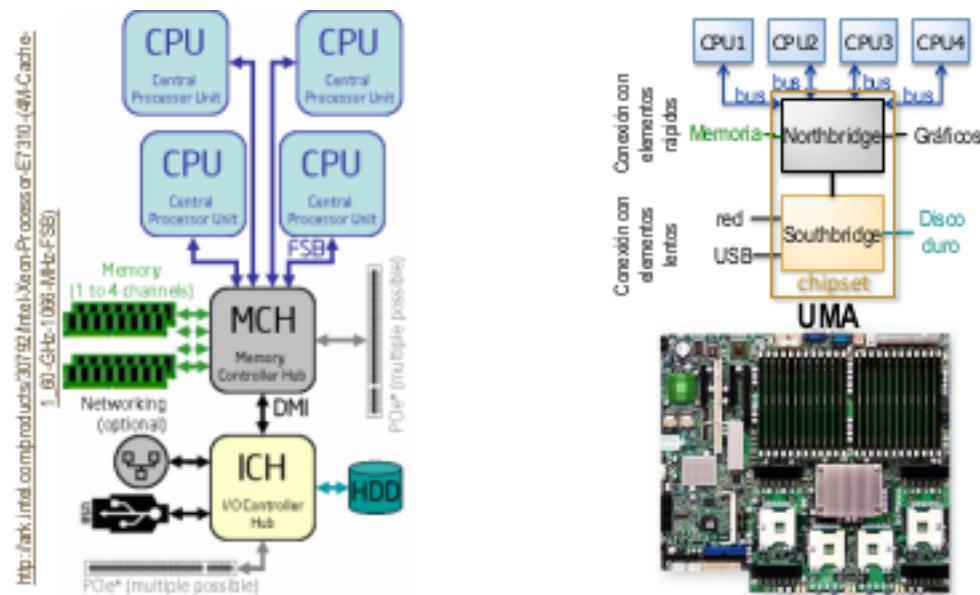
■ NUMA.

- Controlador de memoria en chip del procesador.
- Red: enlaces (conexiones punto a punto) y conmutadores (en el chip del procesador).
- Ejemplos en servidores:

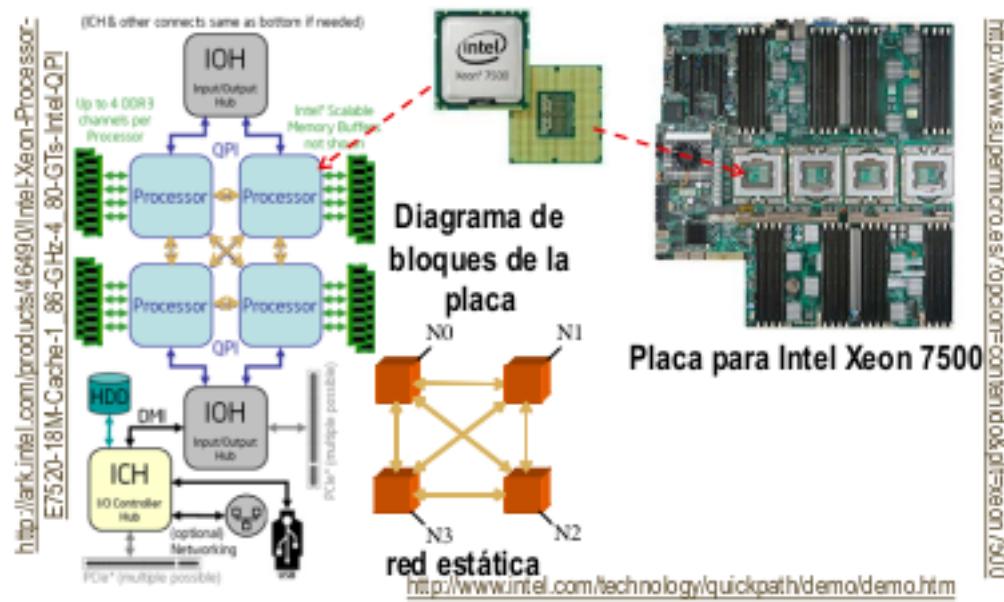
- AMD Opteron (2003): enlaces HyperTransport (2001).
- Intel (Nehalem) Xeon 7500 (2010): enlaces QPI (Quick Path Interconnect, 2008).



Multiprocesador en una placa: UMA con bus (Intel Xeon 7300):



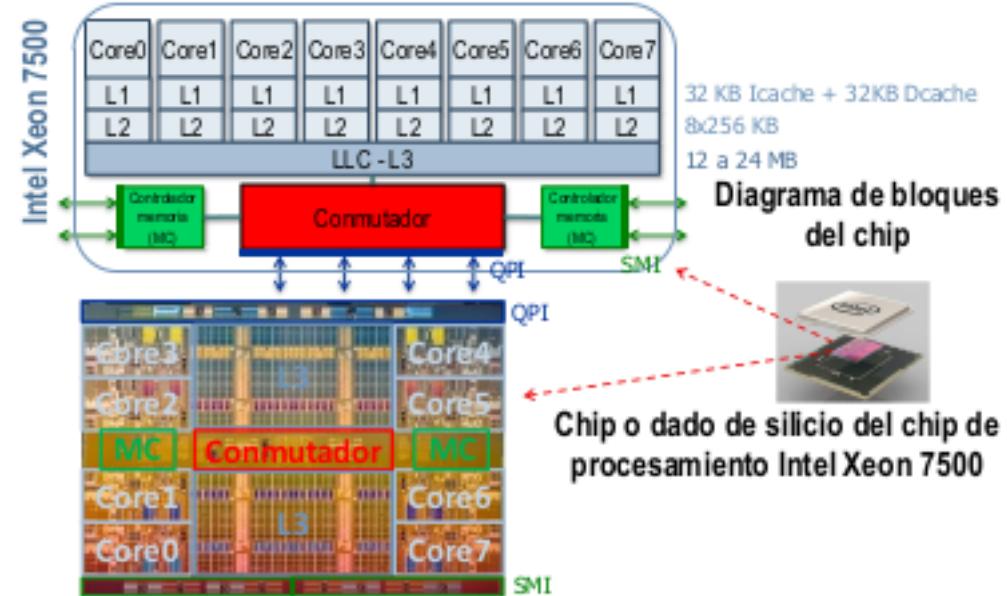
Multiprocesador en una placa: CC-NUMA con red estática (Intel Xeon 7500):



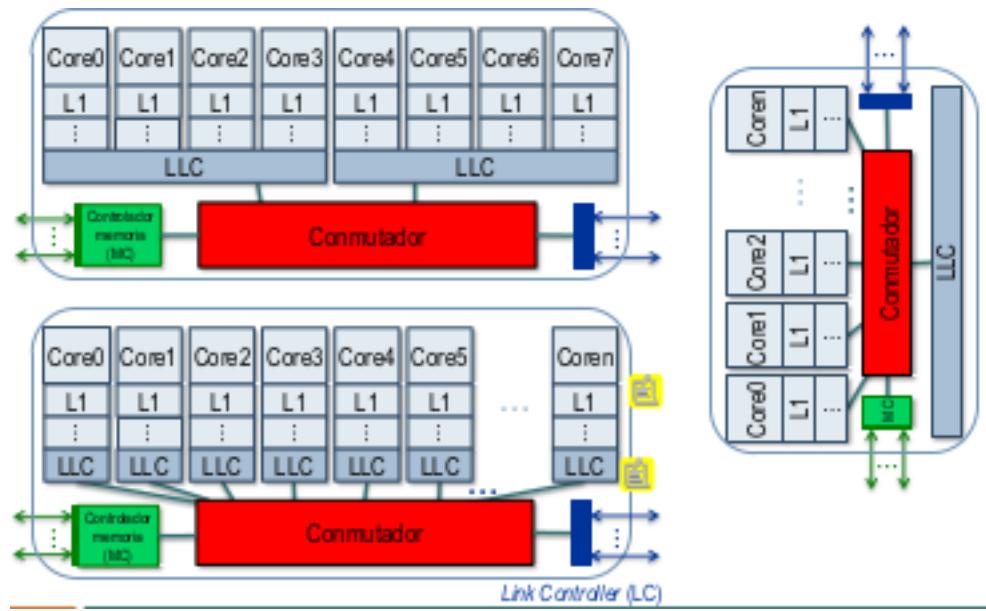
3.1.4 7.3 Multicores.

Ejecutan varios threads en paralelo en un chip de procesamiento multicore (cada thread en un core distinto).

Multiprocesador en un chip o Multicore o CMP (Chip MultiProcessor):



Multicore: otras posibles estructuras:



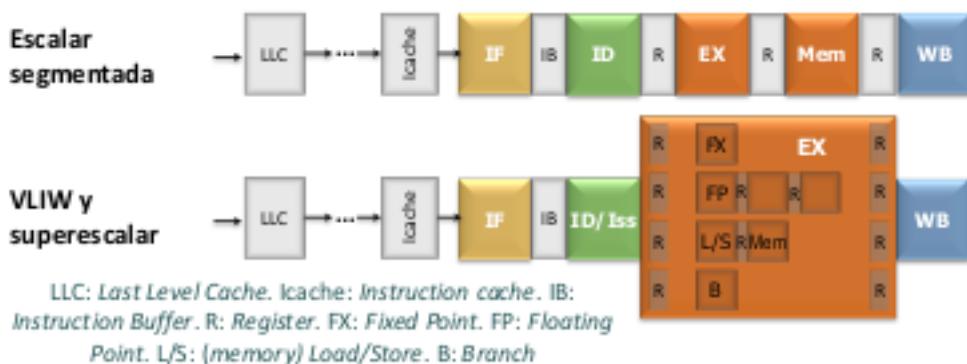
3.1.5 7.4 Cores Multithread.

Modifican su arquitectura ILP (segmentada, escalar o VLIW) para ejecutar threads concurrentemente o en paralelo.

7.4.1 Arquitecturas ILP.

Se modifica la arquitectura para el mismo nivel de instrucción para parallelizar para aprovechar el hardware de parallelización.

- Etapa de captación de instrucciones (*Instruction Fetch*).
- Etapa de decodificación de instrucciones y emisión a unidades funcionales (*Instruction Decode/Instruction Issue*).
- Etapas de ejecución (*Execution*). Etapa de acceso a memoria (*Memory*).
- Etapa de almacenamiento de resultados (*Write-Back*): capta el resultado del registro de arquitectura (para los programas en ensamblador).



- **Procesadores/cores segmentados.** Ejecutan instrucciones concurrentemente segmentando el uso de sus componentes.

- **Procesadores/cores VLIW (Very Large Instruction Word) y superescalares.** Ejecutan instrucciones concurrentemente (segmentación) y en paralelo (tienen múltiples unidades funcionales y emiten múltiples instrucciones en paralelo a unidades funcionales).

- **VLIW.**

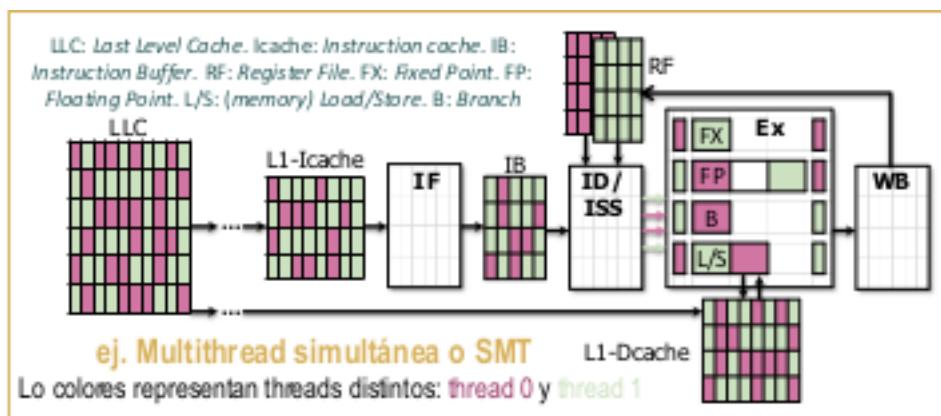
- Las instrucciones que se ejecutan en paralelo se captan juntas de memoria.
- Este conjunto de instrucciones conforman la palabra de instrucción muy larga a la que hace referencia la denominación VLIW.
- El hardware presupone que las instrucciones de una palabra son independientes: no tiene que encontrar instrucciones que pueden emitirse y ejecutarse en paralelo.

- **Superescalares.**

- Tiene que encontrar instrucciones que puedan emitirse y ejecutarse en paralelo (tiene hardware para extraer paralelismo a nivel de instrucción).

7.4.2 Modificación de la arquitectura ILP en Core Multithread (ej. SMT).

- Almacenamiento: se multiplexa, se reparte o comparte entre threads, o se replica.
 - Con SMT: repartir, compartir o replicar.
- Hardware dentro de etapas: se multiplexa, o se reparte o comparte entre threads.
 - Con SMT: unidades funcionales (etapa Ex) compartidas, resto etapas repartidas o compartidas; multiplexación es posible (p. ej. predicción de saltos y decodificación).



7.4.3 Clasificación de cores multithread.

- **Temporal Multithreading (TMT),**
 - Ejecutan varios threads concurrentemente en el mismo core.
 - La commutación entre threads la decide y controla el hardware.
 - Emite instrucciones de un único thread en un ciclo
- **Simultaneous MultiThreading (SMT) o multihilo simultáneo o horizontal multithread.**

- Ejecutan, en un core superescalar, varios threads en paralelo.
- Pueden emitir (para su ejecución) instrucciones de varios threads en un ciclo.



7.4.4 Clasificación de cores con TMT.

- **Fine-grain multithreading (FGMT) o interleaved multithreading.**
 - La conmutación entre threads la decide el hardware cada ciclo (coste 0).
 - por turno rotatorio (*round-robin*) o
 - por eventos de cierta latencia combinado con alguna técnica de planificación (ej. thread menos recientemente ejecutado)
 - ◊ Eventos: dependencia funcional, acceso a datos a cache L1, salto no predecible, una operación de cierta latencia (ej. div), ...
- **Coarse-grain multithreading (CGMT) o blocked multithreading.**
 - La conmutación entre threads la decide el hardware (coste de 0 a varios ciclos).
 - tras intervalos de tiempo prefijados (timeslice multithreading) o
 - por eventos de cierta latencia (switch-on-event multithreading).

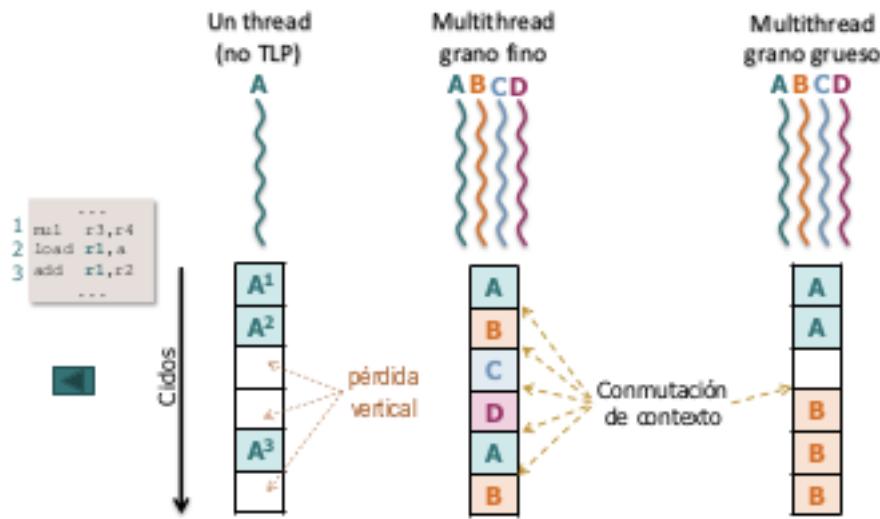


7.4.5 Clasificación de cores con CGMT con conmutación por eventos.

- **Estática.**
 - Conmutación.
 - Explícita: instrucciones explícitas para conmutación (instrucciones añadidas al repertorio).
 - Implícita: instrucciones de carga, almacenamiento, salto.
 - Ventaja/Inconveniente: coste cambio contexto bajo (0 o 1 ciclo) / cambios de contextos innecesarios.
- **Dinámica.**
 - Conmutación típicamente por fallo en la última cache dentro del chip de procesamiento (conmutación por fallo de cache), interrupción (conmutación por señal), ...
 - Ventaja/Inconveniente: reduce cambios de contexto innecesarios / mayor sobrecarga al cambiar de contexto.

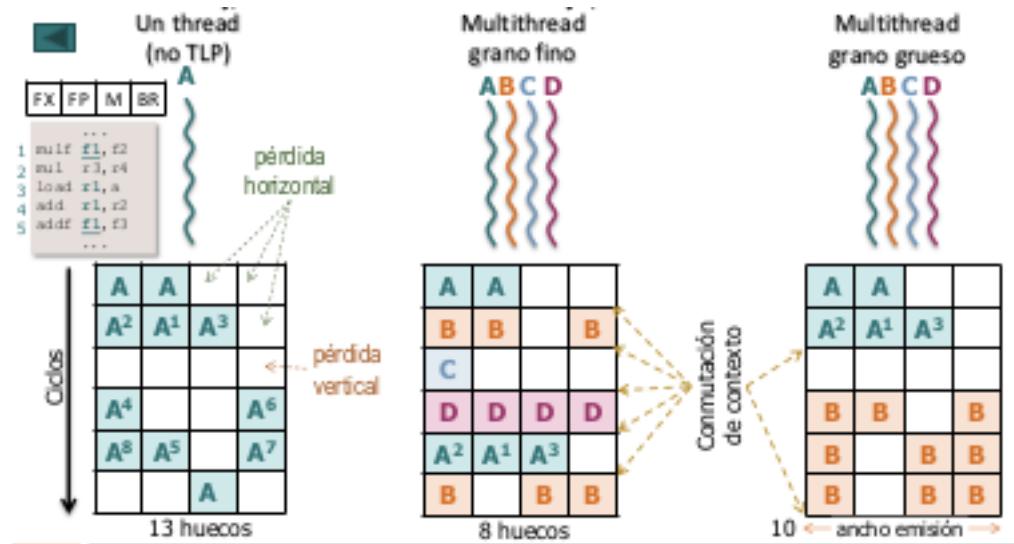
7.4.6 Alternativas en un core escalar segmentado.

En un core escalar se emite una instrucción cada ciclo de reloj.



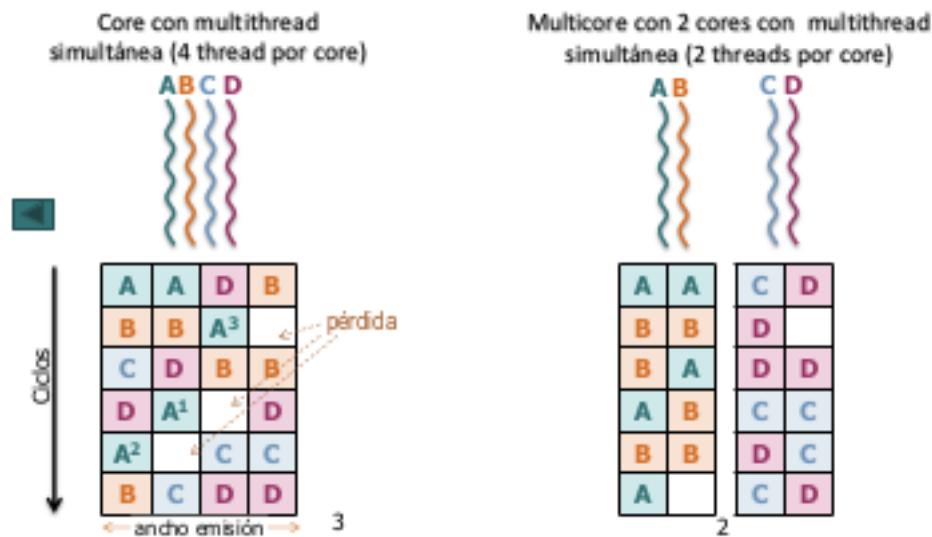
7.4.7 Alternativas en un core con emisión múltiple de instrucciones de un thread.

En un core superescalar o VLIW se emiten más de una instrucción cada ciclo de reloj; en las alternativas de abajo, de un único thread.



7.4.8 Core multithread simultánea y multicores.

En un multicore y en un core superescalar con SMT (Simultaneous MultiThread) se pueden emitir instrucciones de distintos threads cada ciclo de reloj.



3.1.6 7.5 Hardware y arquitecturas TLP en un chip.

Hardware	CGMT	FGMT	SMT	CMP
Registros de la arquitectura	replicado (al menos PC)	replicado	replicado	replicado
Almacenamiento	multiplexado	multiplexado, repartido, compartido o replicado	repartido, compartido o replicado	replicado
Otro hardware de las etapas del cauce	multiplexado	<u>Captación:</u> repartida o compartida; <u>Resto:</u> multiplexadas	<u>UF:</u> compartidas; <u>Resto:</u> repartidas o compartidas	replicado
Etiquetas para distinguir el thread de una instr.	Si	Si	Si	No
Hardware para comutar entre threads	Si	Si	No	No

3.2 Lección 8. Coherencia del sistema de memoria.

3.2.1 Objetivos.

- Comparar los métodos de actualización de memoria principal implementados en cache.
- Comparar las alternativas para propagar un escritura en protocolos de coherencia de cache.
- Explicar qué debe garantizar el sistema de memoria para evitar problemas por incoherencias.
- Describir las partes en las que se puede dividir el análisis o el diseño de protocolos de coherencia.
- Distinguir entre protocolos basados en directorios y protocolos de espionaje (snoopy).
- Explicar el protocolo de mantenimiento de coherencia de espionaje MSI.
- Explicar el protocolo de mantenimiento de coherencia de espionaje MESI.
- Explicar el protocolo de mantenimiento de coherencia MSI basado en directorios con difusión y sin difusión.

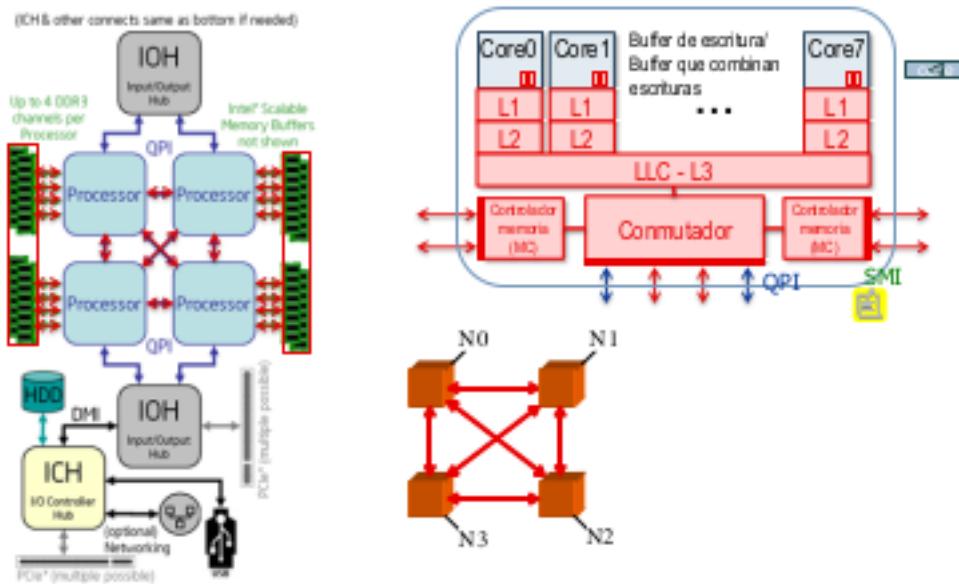
Computadores que implementan en hardware mantenimiento de coherencia:

Multi-computadores Memoria no compartida	NORMA No Remote Memory Access	nivel de sistema (<i>Cluster</i> , armario, chasis (<i>blade server</i>))	 Memoria físicamente distribuida	+ Escalabilidad
	NUMA Non-Uniform Memory Access	NUMA (nivel de sistema, n. armario/chasis, n. placa) CC-NUMA (nivel de armario: SGI Altix; nivel de placa)		
Multi-procesadores Memoria compartida Un único espacio de direcciones	UMA Uniform Memory Access	COMA	 Memoria físicamente centralizada	- Escalabilidad
		Coherencia por hardware SMP System-on-a-chip MultiProcessor (nivel de placa; nivel de chip: multicore como Intel Core i7, i5, i3)		

3.2.2 8.1 Sistema de memoria en multiprocesadores.

- El sistema de memoria incluye:
 - Caches de todos los nodos.
 - Memoria principal.
 - Controladores.
 - Buffers:
 - Buffer de escritura/almacenamiento.
 - Buffer que combinan escrituras/almacenamientos, etc.
 - Medio de comunicación de todos estos componentes (red de interconexión).
- La comunicación de datos entre procesadores la realiza el sistema de memoria.

- La lectura de una dirección debe devolver lo último que se ha escrito (desde el punto de vista de todos los componentes del sistema).



3.2.3 8.2 Concepto de coherencia en el sistema de memoria: situaciones de incoherencia y requisitos para evitar problemas en estos casos.

8.2.1 Incoherencia en el sistema de memoria.

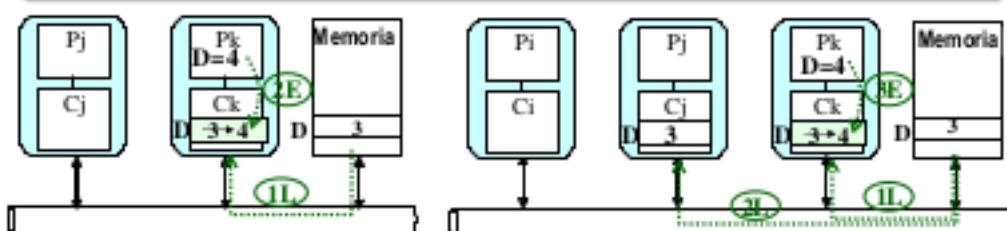
Si en el sistema de memoria las copias de una dirección no tienen el mismo contenido, tendremos una **incoherencia** en el sistema de memoria. Esto puede provocar problemas si los componentes del sistema se comunican a través de posiciones de memoria (escribiendo y leyendo en posiciones de memoria). No se podrá conseguir llevar a cabo satisfactoriamente la comunicación si el componente que recibe no lee a través de la dirección de memoria a la que accede lo último que se ha escrito.

En la imagen de abajo a la izquierda, el procesador P_k lee de la posición de memoria D (1L), lo que provoca que el bloque de memoria donde se encuentra esta dirección se copie a su caché. Tras la transferencia, el contenido de la posición de memoria D en la caché es 3, coincide con el contenido de la dirección en memoria principal. Si P_k escribe un nuevo valor, 4, en la dirección D (2E), escribirá en la copia de la dirección que tienen en su caché. Tendremos entonces una incoherencia en el sistema de memoria, ya que en la posición D no tiene el mismo contenido en memoria principal y en la caché. También pueden dar problemas las situaciones de incoherencia entre caché y memoria principal para datos modificables privados de un proceso si el SO permite que los procesos migren de un procesador a otro. Supongamos que el proceso P_k que acaba de modificar D en su caché, emigra al procesador P_j . Si en P_j el proceso accede a D , leería de memoria un contenido no actualizado.

En la imagen de abajo a la derecha, dos procesos que se ejecutan en procesadores distintos, P_k y P_j , acceden a una dirección D que comparten y que además pueden modificar. Primero leen (1L y 2L) el contenido de la dirección de memoria D , lo que provoca los correspondientes fallos de caché y la transferencia del bloque de memoria donde se encuentra esta dirección a la caché de los dos procesadores. Uno de ellos P_k , escribe en D , pasando el contenido de D en su caché a 4. Por lo que se produce una falta de coherencia en el sistema de memoria entre la caché de P_j y P_k , además de una falta de coherencia con la

memoria principal. Si un dispositivo de E/S escribe en la posición D , modificará la copia de memoria. Entonces ninguna de las tres copias de D en el sistema tendrá el mismo contenido. La falta de coherencia entre cachés provocada al modificar un dato compartido, se hace patente si el proceso P_j vuelve a leer el contenido de D , ya que el valor que obtiene, es el contenido no actualizado de su caché.

Clases de estructuras de datos	Eventos que ponen de manifiesto faltas de coherencia	Tipos de Falta de coherencia
Datos modificables	E/S	Cache-MP
Datos modificables compartidos	Fallo de cache	Cach e-MP
Datos modificables privados	Emigra thread/proceso → Fallo cache	Cach e-MP
Datos modificables compartidos	Lectura de cache no actualizada	Cach e-Cache

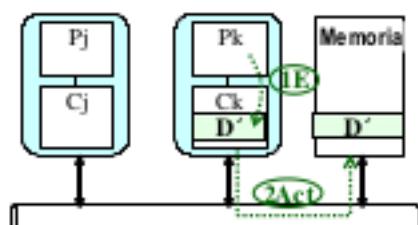


8.2.2 Métodos de actualización de memoria principal implementados en caches.

- **Escritura inmediata (write-through).** Cada vez que un procesador escribe en su cache escribe también en memoria principal. Cada escritura supone la utilización de la red y para transferir datos aislados, con el consiguiente desaprovechamiento del ancho de banda de la red. La situación del tráfico empeora en sistemas con múltiples procesadores, ya que puede haber varios procesadores escribiendo simultáneamente.

Como en una aplicación se escribe varias veces en un bloque (a las variables contiguas en el código del programa se les asigna posiciones consecutivas en memoria), en la misma dirección o en distintas direcciones, se incrementarían entonces las prestaciones si el bloque se transfiere por la red una vez realizadas todas las modificaciones (se disminuye el número de accesos, y cuando se accede, se aprovecha en mayor medida el ancho de banda de la red).

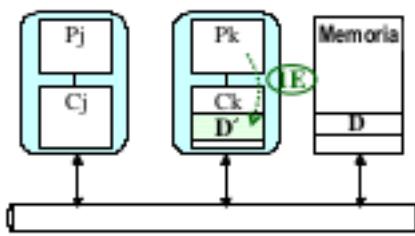
Con escritura inmediata se opta por no provocar incoherencia entre caché y memoria principal cuando se escribe en caché; pero no se evita la incoherencia entre cachés, o entre caché y memoria cuando escribe en memoria principal algún componente.



- **Posescritura (write-back).** Se actualiza memoria principal escribiendo todo el bloque cuando se

desaloja de la caché. Cuando un procesador modifica una dirección solo se escribe en la caché del procesador; el dato no se transfiere a memoria principal, por lo que se puede escribir varias veces en un bloque sin acceder a memoria principal. La actualización de memoria se realiza posteriormente, cuando el bloque que contiene la dirección modificada se elimina de caché a fin de dejar espacio para otro bloque. Se debe mantener información en el directorio caché sobre los bloques de memoria modificados en la caché.

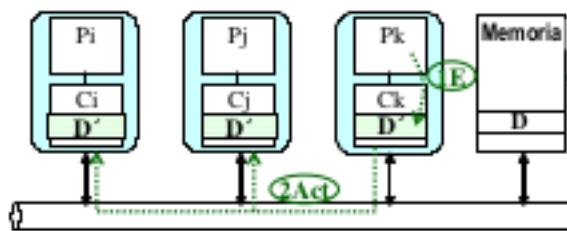
Se permite que aparezcan incoherencias entre caché y memoria también cuando se escribe en caché. La situación empeora con posescritura frente a escritura inmediata.



8.2.3 Alternativas para propagar una escritura en protocolos de coherencia de cache.

La falta de coherencia entre cachés y los problemas debidos a la incoherencia entre memoria principal y caché se pueden solventar con hardware específico.

- **Escritura con actualización (write-update).** Cada vez que un procesador escribe en una dirección en su cache se escribe en las copias de esa dirección en otras caches.

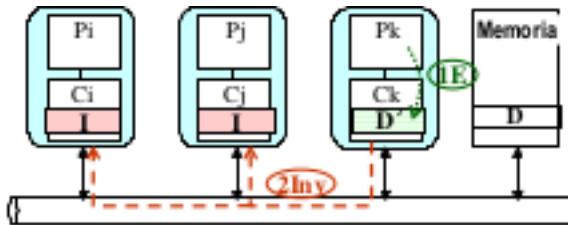


Para reducir tráfico, sobre todo si los datos están compartidos por pocos procesadores.

- **Escritura con invalidación (write-invalidate).** Antes que un procesador modifique una dirección en su cache se invalidan las copias del bloque de la dirección en otras caches. El procesador que va a modificar una dirección, primero obtiene acceso exclusivo al bloque que la contiene. Cuando otro procesador lea la dirección, su caché falla, provocando que tenga que acceder a memoria principal, consiguiendo así el dato actualizado. El acceso exclusivo pretende asegurar que no hay otras copias del bloque en cachés de otros procesadores que se puedan leer o escribir cuando se está realizando la escritura. Invalidar es más rápido que actualizar, ya que solo se debe transferir la dirección (bloque) en la que se va a escribir, mientras que al actualizar, se deben transferir además los datos a escribir. Usando invalidación solo se permite compartir un bloque de memoria mientras se lee del bloque.

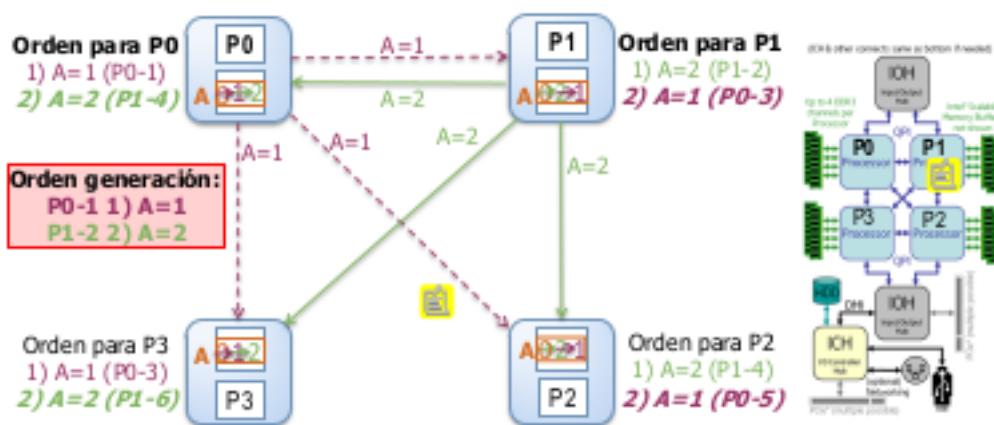
Si se escribe varias veces sucesivas en un bloque sin que otro procesador lea, usando invalidación se puede reducir el acceso a la red (transferencias) frente a la política de actualización, ya que una

vez invalidado un bloque, nuevas modificaciones del bloque por parte del mismo procesador no originan transferencias. Por contra, si se escribe para que a continuación otro u otros procesadores lean lo escrito, podría ser más eficiente actualizar, ya que al invalidar las copias en otras cachés cuando se escribe, las lecturas posteriores de esa dirección por otros procesadores provocan fallos de caché, lo que origina tráfico en la red.



8.2.4 Situación de incoherencia aunque se propagan las escrituras (usa difusión).

- Contenido inicial de las copias de la dirección A en calabaza. P0 escribe en A un 1 y, después, P1 escribe en A un 2.
- Se utiliza actualización para propagar las escrituras (las propagación se nota con flechas).
- Llegan en distinto orden las escrituras debido al distinto tiempo de propagación (se está suponiendo que los Proc. están ubicados en la placa tal y como aparecen en el dibujo). En cursiva se puede ver el contenido de las copias de la dirección A tras las dos escrituras.
 - Se da una situación de incoherencia aunque se propagan las escrituras: P0 y P3 acaban con 2 en A y P1 y P2 con 1.



8.2.5 Requisitos del sistema de memoria para evitar problemas por incoherencia.

- Propagar** las escrituras en una dirección.
 - La escritura en una dirección debe hacerse visible en un tiempo finito a otros procesadores.
 - Componentes conectados con un bus:
 - Los paquetes de actualización/invalidación son visibles a todos los nodos conectados al bus (controladores de cache).

■ **Serializar** las escrituras en una dirección.

- Las escrituras en una dirección deben verse en el mismo orden por todos los procesadores (el sistema de memoria debe parecer que realiza en serie las operaciones de escritura en la misma dirección).
- Componentes conectados con un bus:
 - El orden en que los paquetes aparecen en el bus determina el orden en que se ven por todos los nodos.



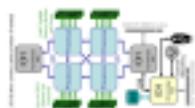
8.2.6 Requisitos del sistema de memoria para evitar problemas por incoherencia: la red no es un bus.

■ **Propagar** escrituras en una dirección

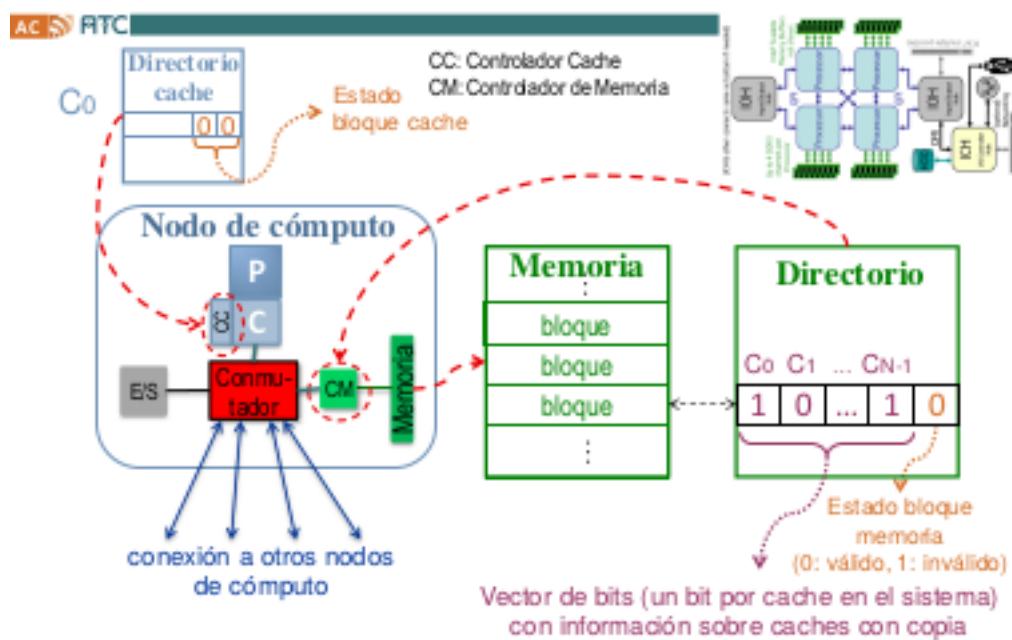
- Usando difusión:
 - Los paquetes de actualización/invalidación se envían a todas las cachés.
- Para conseguir mayor escalabilidad:
 - Se debería enviar paquetes de actualización/invalidación sólo a caches (nodos) con copia del bloque.
 - Mantener en un directorio, para cada bloque, los nodos con copia del mismo.

■ **Serializar** escrituras en una dirección.

- El orden en el que las peticiones de escritura llegan a su home (nodo que tiene en MP la dirección) o al directorio centralizado sirve para serializar en sistemas de comunicación que garantizan el orden en las transferencias entre dos puntos.



8.2.7 Directorio de memoria principal.



8.2.8 Alternativas para implementar el directorio.

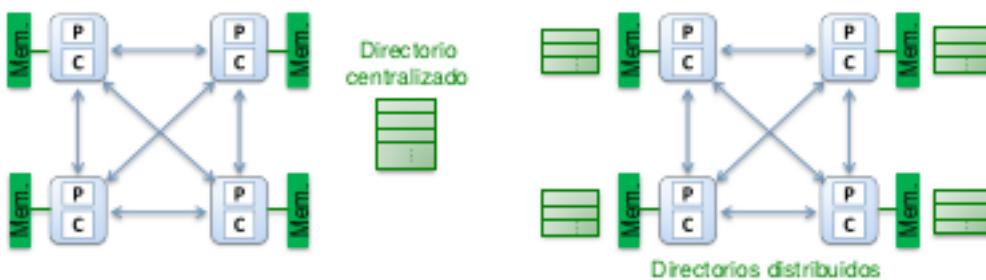
■ Centralizado.

- Compartido por todos los nodos.
- Contiene información de los bloques de todos los módulos de memoria.

■ Distribuido.

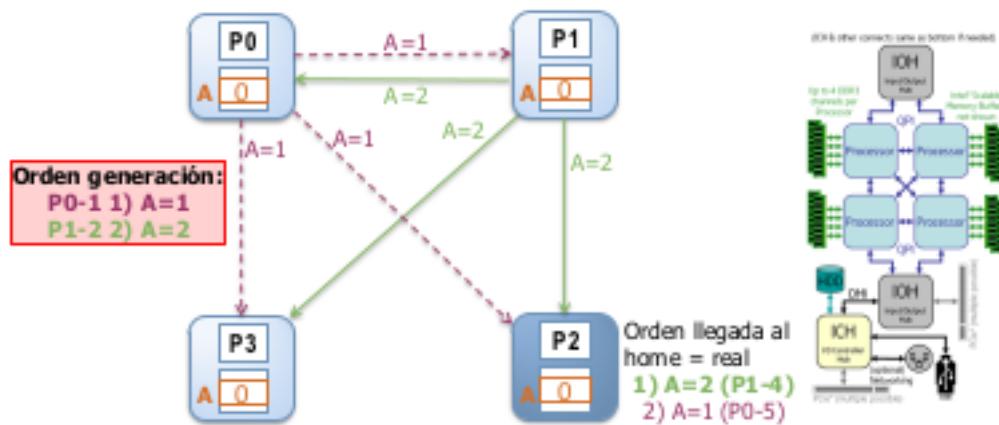
- Las filas se distribuyen entre los nodos.
- Típicamente el directorio de un nodo contiene información de los bloques de sus módulos de memoria.

Centralizado	Distribuido
<ul style="list-style-type: none"> • Compartido por todos los nodos • Contiene información de los bloques de todos los módulos de memoria 	<ul style="list-style-type: none"> • Las filas se distribuyen entre los nodos • Tipicamente el directorio de un nodo contiene información de los bloques de sus módulos de memoria

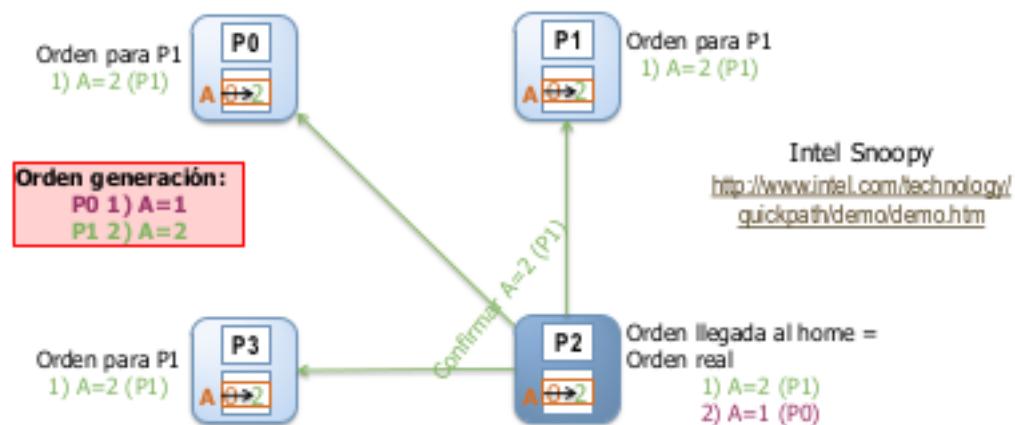


8.2.9 Serialización de las escrituras por el home. Usando difusión.

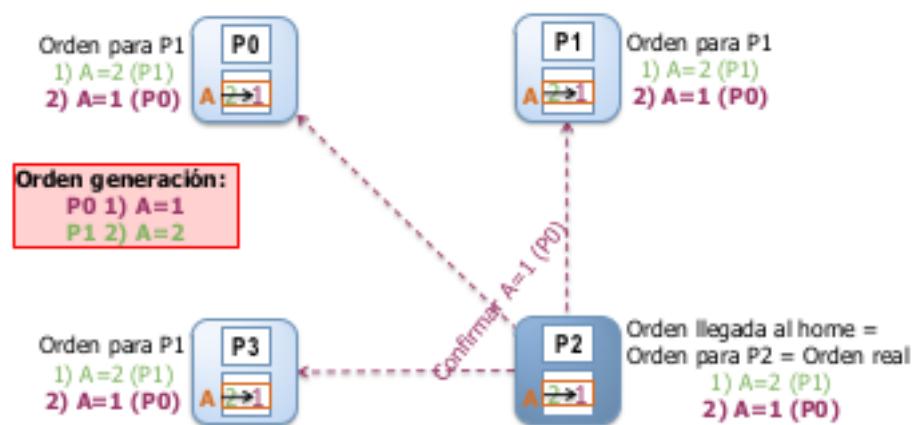
- Contenido inicial de las copias de la dirección A en calabaza. P0 escribe en A un 1 y, después, P1 escribe en A un 2.
- Se utiliza actualización para propagar las escrituras (las propagación se nota con flechas).
- El orden de llegada al home es el orden real para todos.



- Contenido inicial de las copias de la dirección A en calabaza.
- Se utiliza actualización para propagar las escrituras (las propagación se nota con flechas).

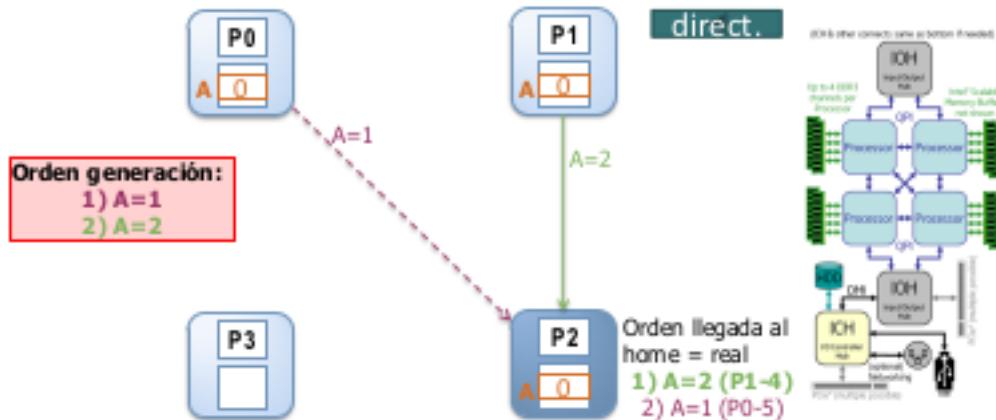


- Se utiliza actualización para propagar las escrituras (las propagación se nota con flechas).

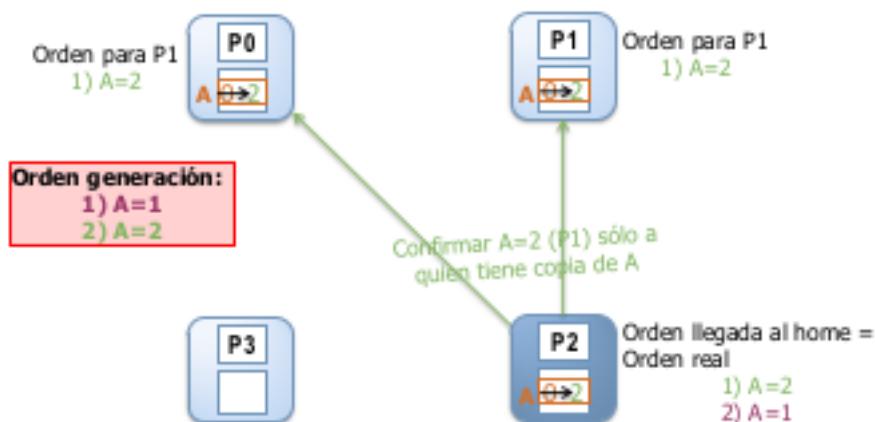


8.2.10 Serialización de las escrituras por el home. Sin difusión y con directorio distribuido.

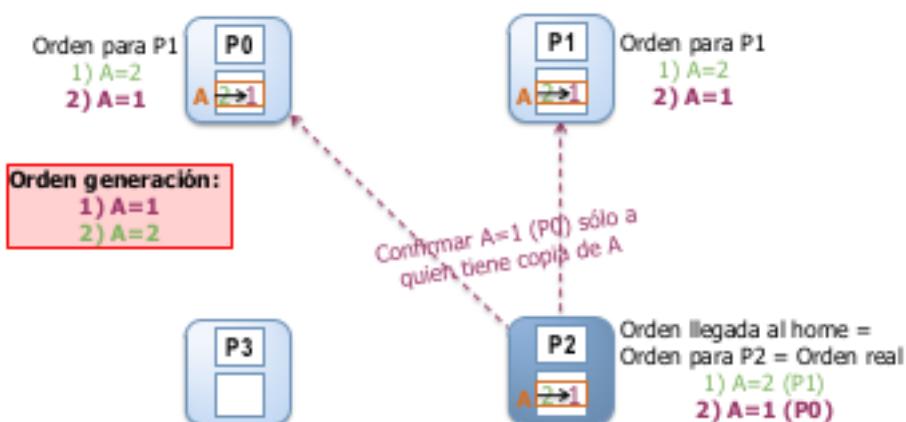
- Contenido inicial de las copias de la dirección A en calabaza. P0 escribe en A un 1 y, después, P1 escribe en A un 2.
- Se utiliza actualización para propagar las escrituras (las propagación se nota con flechas).
- El orden de llegada al home es el orden real para todos.



- Contenido inicial de las copias de la dirección A en calabaza.
- Se utiliza actualización para propagar las escrituras (las propagación se nota con flechas).

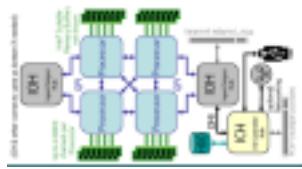


- Se utiliza actualización para propagar las escrituras (las propagación se nota con flechas).



3.2.4 8.3 Protocolos de mantenimiento de coherencia: clasificación y diseño.

8.3.1 Clasificación de protocolos para mantener coherencia en el sistema de memoria.

- Protocolos de espionaje (snoopy).
 - Para buses, y en general sistemas con una difusión eficiente (bien porque el número de nodos es pequeño o porque la red implementa difusión).
- Protocolos basados en directorios.
 - Para redes sin difusión o escalables (multietapa y estáticas).
- Esquemas jerárquicos.
 - Para redes jerárquicas: jerarquía de buses, jerarquía de redes escalables, redes escalables-buses.

8.3.2 Facetas de diseño lógico en protocolos para coherencia.

- Política de actualización de MP:
 - escritura inmediata, posescritura, mixta.
- Política de coherencia entre caches:
 - escritura con invalidación, escritura con actualización, mixta
- Describir comportamiento:
 - Definir posibles estados de los bloques en cache, y en memoria.
 - Definir transferencias (indicando nodos que intervienen y orden entre ellas) a generar ante eventos:
 - lecturas/escrituras del procesador del nodo.
 - como consecuencia de la llegada de paquetes de otros nodos.
 - Definir transiciones de estados para un bloque en cache, y en memoria.

3.2.5 8.4 Protocolo MSI de espionaje.

8.4.2 Protocolo de espionaje de tres estados (MSI) – posescritura e invalidación.

- Estados de un bloque en caché:

- **Modificado (M)**: es la única copia del bloque válida en todo el sistema. La caché debe proporcionar el bloque si observa al espiar el bus que algún componente lo solicita y debe invalidarla si algún otro nodo solicita una copia exclusiva del bloque para su modificación. El procesador tiene el uso exclusivo del bloque, de forma que puede disponer de él para leer y escribir sin informar al resto del sistema. La caché debe atender a las peticiones a través del bus.
- **Compartido (C,S)**: está válido, también válido en memoria y puede que haya copia válida en otras caches. La caché debe invalidar su copia si observa al espiar el bus que algún otro nodo solicita una copia exclusiva del bloque para su modificación.
- **Inválido (I)**: no está físicamente o se ha invalidado por el controlador como consecuencia de la escritura en el bloque en otra caché. El procesador tiene que acceder al bloque a través de la red.

Se pueden ordenar en función del grado de propiedad o disponibilidad del bloque por parte del procesador al que pertenece la caché (orden creciente): inválido, compartido y modificado.

■ **Estados** de un bloque en memoria (en realidad se evita almacenar esta información):

- **Válido**: puede haber copia válida en una o varias caches.
- **Inválido**: habrá copia válida en una cache.

■ **Transferencias** generadas por un nodo con caché (tipos de paquetes):

- **Petición de lectura de un bloque (PtLec)**: por lectura con fallo de caché del procesador del nodo (**PrLec**). El controlador de la caché inicia la transacción de lectura poniendo en el bus la dirección a la que se desea acceder. El sistema de memoria (memoria principal u otra caché) proporcionará el bloque donde se encuentra la dirección solicitada.
- **Petición de acceso exclusivo (PtLecEx)**: por escritura del procesador (**PrEsc**) en bloque compartido o inválido. El controlador de la caché genera el paquete que indicará la dirección en la que se desea escribir. El resto de cachés con copias válidas del bloque invalidan sus copias. También se invalida el bloque en memoria si se encuentra en estado válido. El procesador puede solicitar una confirmación de este paquete.
- **Petición de posescritura (PtPEsc)**: por el reemplazo del bloque modificado (el procesador del nodo no espera respuesta). El bloque a reemplazar generado por la circuitería de reemplazo debe transferirse a memoria principal si está en estado modificado (ya que la memoria no tiene el bloque actualizado). El controlador de la caché genera la transferencia poniendo en el bus la dirección del bloque a escribir en memoria y el contenido del propio bloque. El procesador no conoce este hecho y no espera ninguna respuesta.
- **Respuesta con bloque (RpBloque)**: al tener en estado modificado el bloque solicitado por una **PtLec** o **PtLecEx** recibida.



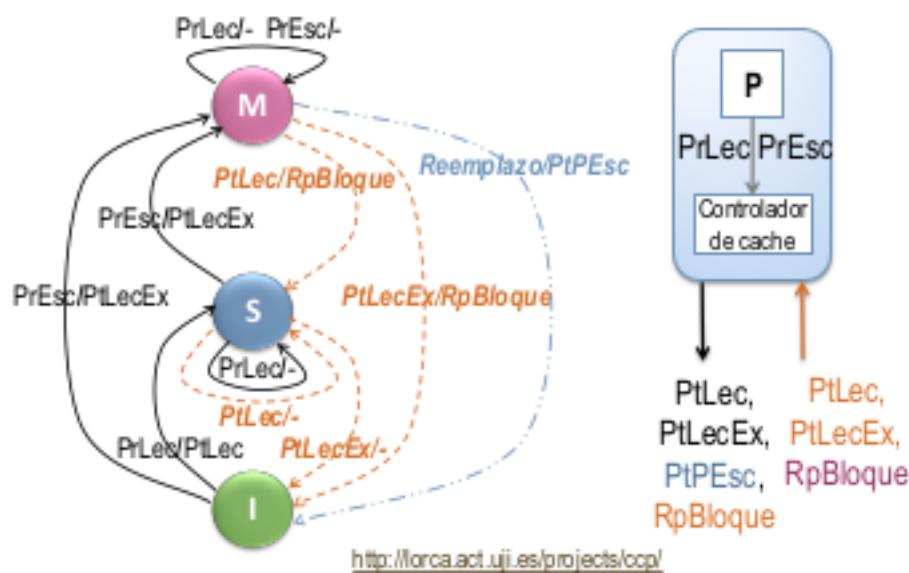
PtLec, PtLecEx

PtPEsc,

o RpBloque

8.4.3 Diagrama MSI de transiciones de estados.

Las líneas discontinuas son transiciones provocadas por paquetes del bus; las líneas continuas son transiciones provocadas por acciones del procesador de la caché. Las líneas de han etiquetado con el evento que provoca la transición y el paquete que generan (evento/paquete).



EST. ACT.	EVENTO	ACCIÓN	SIGUIENTE
Modificado (M)	PrLec/PrEsc		Modificado
	PtLec	Genera paquete respuesta (RpBloque)	Compartido
	PtLecEx	Genera paquete respuesta (RpBloque) Invalida copia local	Inválido
	Reemplazo	Genera paquete posescritura (PtPEsc)	Inválido
Compart. (S)	PrLec		Compartido
	PrEsc <small>modificar</small>	Genera paquete PtLecEx (PtEx)	Modificado
	PtLec		Compartido
	PtLecEx <small>inválido</small>	Invalida copia local	Inválido
Inválido (I)	PrLec	Genera paquete PtLec	Compartido
	PrEsc	Genera paquete PtLecEx	Modificado
	PtLec/PtLecEx		Inválido

8.4.4 Tabla de descripción de MSI.

■ **Fallo de lectura:** el procesador lee (PrLec) y el bloque no está en caché (estado inválido). El controlador de caché del procesador que lee difunde un paquete de petición de lectura de un bloque de memoria (PtLec); el estado del bloque en la caché después de la lectura será de compartido. La copia del bloque en otras cachés también tendrá estado compartido, y en la memoria, válido. El paquete PtLec provoca los siguientes efectos en otras cachés:

- Si el bloque se encuentra en otra caché en estado modificado, deposita el bloque en el bus (respuesta RpBloque) y pasa a estado compartido. La memoria también recoge el bloque del bus pasando a estado válido.
- Si el bloque está compartido, la memoria proporciona el bloque a la caché que lo solicita. El bloque sigue en estado compartido.

■ **Fallo de escritura al no estar el bloque en la caché:** el procesador escribe (PrEsc) y el bloque no está en caché. El controlador de caché del procesador que escribe difunde un paquete de petición de acceso exclusivo al bloque (PtLecEx); el estado del bloque en la caché después de la escritura se promociona a modificarlo. El paquete PtLecEx provoca los siguientes efectos:

- Si la memoria tiene el bloque válido, lo deposita en el bus y pasa a estado inválido.
- Si una caché tiene el bloque en estado modificado, deposita el bloque en el bus y pasa a estado inválido.
- Si una caché tiene en estado compartido, pasa a estado inválido.

■ **Acierto de escritura en bloque compartido:** el procesador escribe (PrEsc) y el bloque está en caché en estado compartido. Si el bloque está en la caché en estado compartido, antes de escribir se debe obtener acceso exclusivo al bloque, para lo cual el controlador de caché difunde un paquete de petición de acceso exclusivo al bloque (PtLecEx); el estado del bloque en la caché después de la escritura se promociona modificando (el acierto de escritura en bloque compartido se trata como el fallo de escritura). El paquete PtLecEx generado provoca los siguientes efectos:

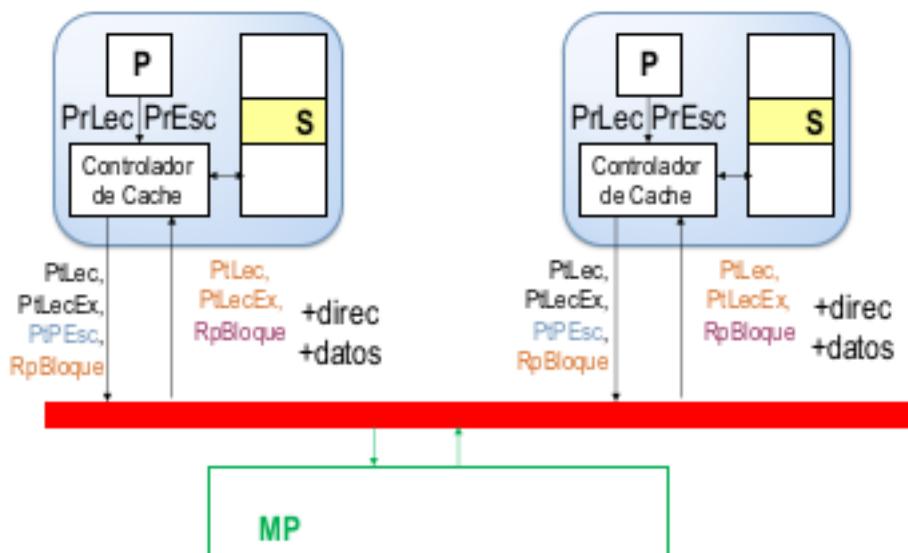
- La memoria deposita el bloque en el bus, ya que lo tiene válido (el bloque depositado se puede ignorar), y pasa a estado inválido. Para evitar que la memoria deposite en este

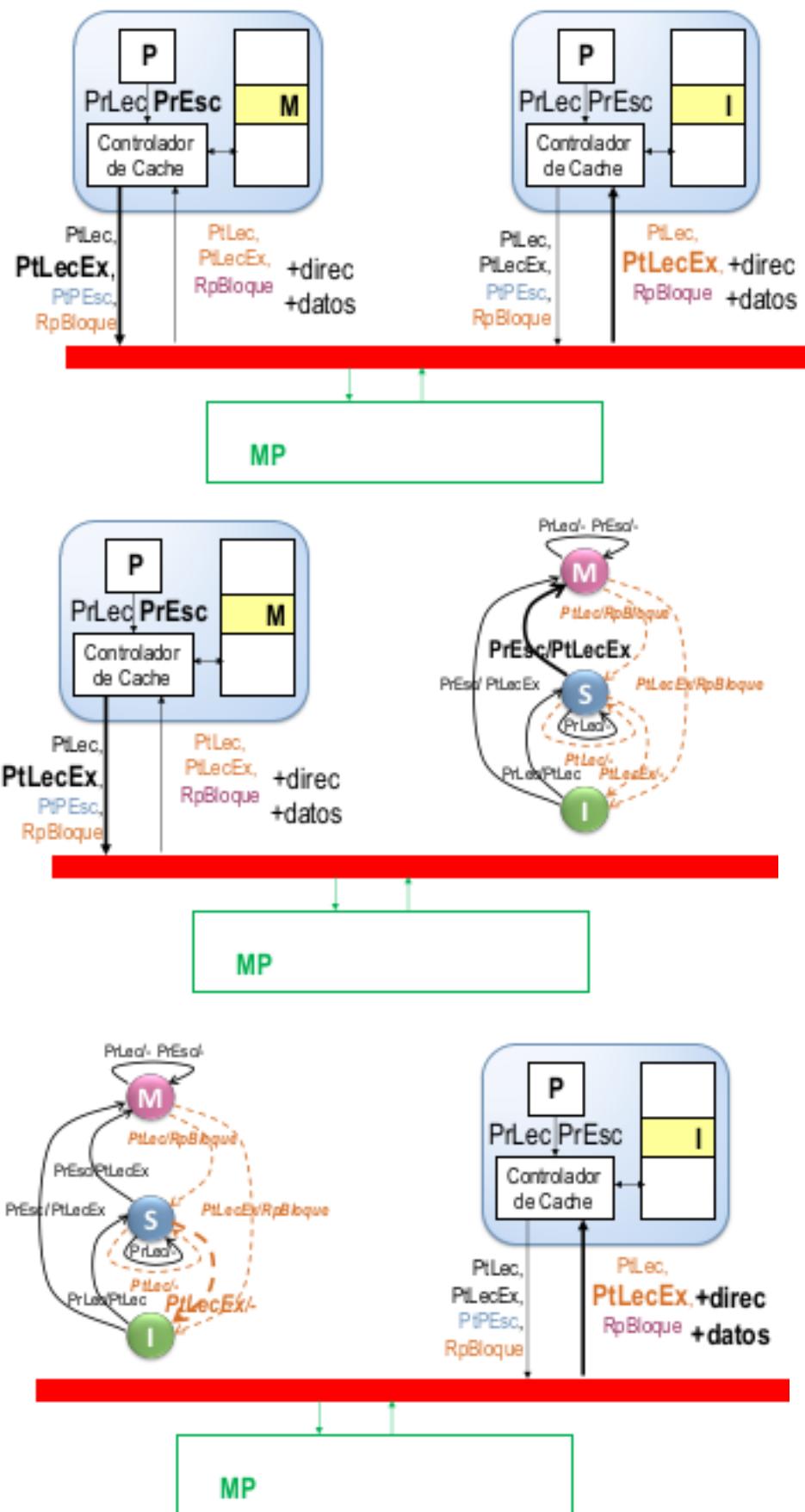
caso el bloque en el bus y reducir así el tráfico, se debe añadir un nuevo paquete al sistema para esta situación en la que el procesador escribe y el bloque está en estado compartido. Sería un paquete de petición de acceso exclusivo, pero sin lectura (PtEx).

2. El bloque no puede estar en estado modificado en otra caché.
3. Si una caché tiene el bloque en estado compartido, pasa a estado inválido.

- **Acierto de escritura en bloque modificado:** el procesador escribe (PrEsc) y el bloque está en caché en estado modificado. Como el nodo ya tiene la propiedad exclusiva del bloque, no se genera ningún paquete. El bloque sigue en estado modificado.
- **Acierto de lectura:** el procesador lee (PrLec) y el bloque se encuentra actualizado en la caché. En este caso el bloque se mantiene en el mismo estado. No genera ningún paquete.
- **Reemplazo:** fallo en el acceso del procesador a otro bloque, y la política de reemplazo selecciona este bloque para hacer sitio al nuevo. Si el bloque reemplazado se encuentra en estado modificado, el controlador de caché del procesador que escribe difunde un paquete de posescritura en memoria (PtPEsc), el estado del bloque en la caché pasa a ser de inválido (no está presente físicamente). Hay que tener en cuenta que se utiliza posescritura como política de actualización de memoria principal. Este paquete PtPEsc provoca que el bloque se transfiera a memoria pasando el estado del bloque en memoria a válido.

8.4.5 Ejemplo MSI.





3.2.6 8.5 Protocolo MESI de espionaje.

8.5.1 Protocolo de espionaje de cuatro estados (MESI) – posescritura e invalidación.

- Estados de un bloque en cache:

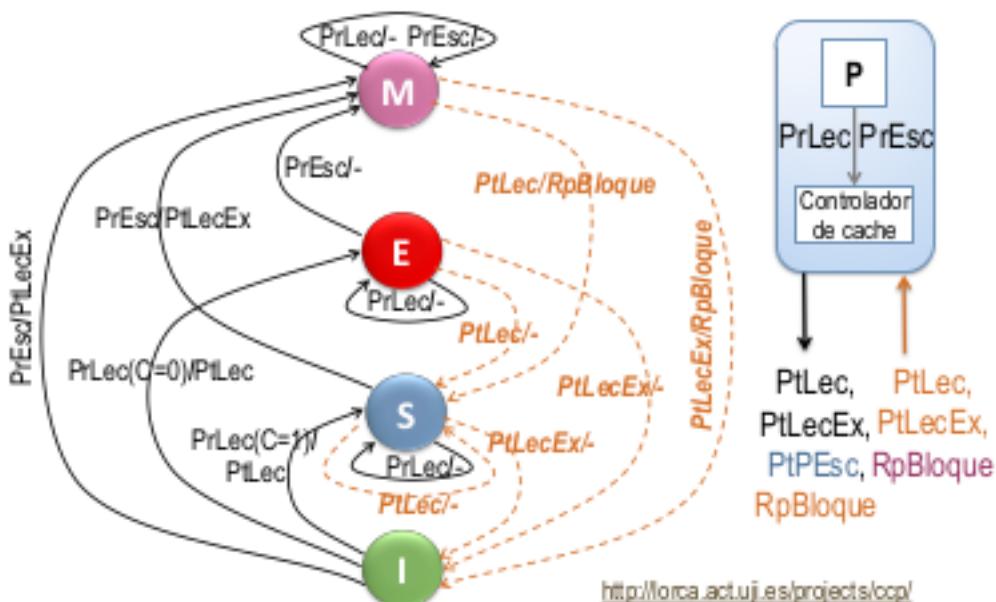
- **Modificado (M)**: es la única copia del bloque válida en todo el sistema, el resto de cachés y la memoria tienen una copia no actualizada. La caché debe proporcionar el bloque si observa al espiar el bus que algún componente lo solicita, y debe invalidarla si algún otro nodo solicita una copia exclusiva del bloque para su modificación.
- **Exclusivo (E)**: es la única copia del bloque válida en caches, la memoria también está actualizada. La caché debe invalidar su copia si observa al espiar el bus que algún otro nodo solicita una copia exclusiva del bloque para su modificación.
- **Compartido (C,Shared)**: es válido en esta caché, también válido en memoria y en al menos otra caché. La caché debe invalidar su copia si observa al espiar el bus que algún otro nodo solicita una copia exclusiva del bloque para su modificación.
- **Inválido (I)**: se ha invalidado o no está físicamente.

- Estados de un bloque en memoria (en realidad se evita almacenar esta información):

- **Válido**: puede haber copia válida en una o varias caches.
- **Inválido**: habrá copia valida en una cache.

8.5.2 Diagrama MESI de transiciones de estados.

Las líneas discontinuas son transiciones provocadas por paquetes del bus, las líneas continuas por acciones del procesador de la caché. Las líneas se han etiquetado con el evento que provoca la transición y el paquete que generan (evento/paquete).



8.5.3 Tabla de descripción de MESI.

Modificado (M)	PtLec/PtEsc		Modificado
	PtLec	Genera RpBloque	Compartido
	PtLecEx	Genera RpBloque. Invalida copia local	Inválido
	Reemplazo	Genera PtPEsc	Inválido
Exclusivo (E)	PtLec		Exclusivo
	PtEsc		Modificado
	PtLec		Compartido
	PtLecEx	Invalida copia local	Inválido
Compartido (S)	PtLec/PtLec		Compartido
	PtEsc	Genera PtLecEx	Modificado
	PtLecEx	Invalida copia local	Inválido
Inválido (I)	PtLec (C=1)	Genera PtLec	Compartido
	PtLec (C=0)	Genera PtLec	Exclusivo
	PtEsc	Genera PtLecEx	Modificado
	PtLec/PtLecEx		Inválido

- **Fallo de lectura:** el procesador lee una dirección del bloque (PtLec) y este no está en caché. El controlador de caché del procesador que lee difunde un paquete de petición de lectura de un bloque de memoria (PtLec). El estado del bloque en la caché después de la lectura pasará a ser compartido si hay copias del bloque en otras cachés, y pasará a estado exclusivo si no hay copias del bloque en otras cachés. El controlador de la caché que solicita el bloque, requiere conocer si el bloque solicitado se encuentra en otras cachés. Se puede añadir una línea OR cableada que informe si hay cachés con copias del bloque solicitado. La actualización de esta OR por parte de los controladores de caché se realizaría en la fase de direccionamiento. La memoria dispondrá del bloque válido al terminar el proceso. El paquete PtLec provoca los siguientes efectos en otras cachés:

1. Si el bloque se encuentra en otra caché en estado modificado, deposita el bloque en el bus y pasa a estado compartido. La memoria también coge el bloque del bus.
2. Si el bloque está compartido, sigue en estado compartido. El bloque que llega a la caché del nodo que lo solicita procede de memoria.

Hay implementaciones en las que, si el bloque solicitado está disponible en estado válido en alguna caché, en lugar de obtener el bloque de memoria, se proporciona por parte de alguna de estas cachés con bloque válido. Hay que añadir entonces hardware, que decida qué caché con bloque válido va a generar la respuesta con el bloque. Esta alternativa es interesante en sistemas con memoria físicamente distribuida (protocolos basados en directorios), ya que se puede proporcionar el bloque desde el nodo más cercano al que lo solicita. - **Fallo de escritura al no estar el bloque en caché:** el procesador escribe (PtEsc) y el bloque no está en caché. El controlador de caché del procesador que escribe difunde un paquete de petición de acceso exclusivo al bloque (PtLecEx). El estado del bloque en la caché después de la escritura será de modificado. El paquete PtLecEx provoca los siguientes efectos: 1. Si una caché tiene el bloque en estado modificado, bloquea la lectura de memoria y deposita el bloque en el bus. El bloque pasa en esta caché a estado inválido. 2. Si una caché tiene el bloque en estado compartido, pasa a estado inválido. - **Acierto de escritura en bloque compartido:** el procesador escribe (PtEsc) y

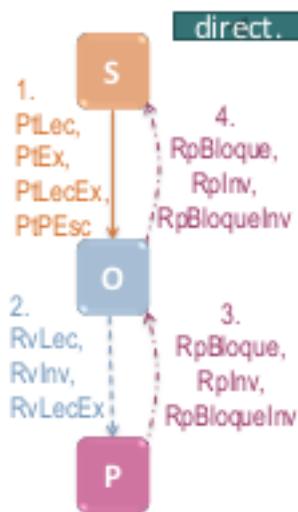
hay una copia del bloque en su caché y en la caché de algún otro procesador. El controlador de caché del procesador que escribe difunde un paquete de petición de acceso exclusivo al bloque (PtLecEx). El estado del bloque en la caché después de la escritura será de modificado. El paquete PtLecEx provoca los siguientes efectos: 1. La memoria puede depositar el bloque en el bus, pero se puede ignorar (la caché que ha generado la petición PtLecEx tiene el bloque válido). Para evitar que se utilice el bus de datos innecesariamente se puede añadir al diseño otro paquete distinto que pida acceso exclusivo pero no la lectura de los datos (PtEx). 2. El bloque no puede estar en estado modificado en otra caché. 3. Si una caché tiene bloque en estado compartido, pasa a estado inválido. - **Acierto de escritura en bloque modificado:** el procesador escribe (PrEsc) y el bloque está en caché en estado modificado. Como no hay otro nodo con copia válida del bloque en su caché no es necesario generar un paquete. El bloque continúa en estado modificado. - **Acierto de lectura:** el procesador lee (PrLec) y el bloque se encuentra en la caché actualizado. El bloque se mantiene en el mismo estado. No genera ningún paquete. - **Reemplazo:** fallo en el acceso del procesador a otro bloque y la política de reemplazo selecciona este bloque para hacer sitio al nuevo bloque. El controlador de caché del procesador que escribe difunde un paquete de posescritura en memoria (PtPEsc) si el bloque reemplazado se encuentra en estado modificado. El bloque pasa a estado inválido (no está presente físicamente). Este paquete PtPEsc provoca que el bloque se transfiera a la memoria pasando el estado del bloque en memoria a válido (si se almacena información de estado en memoria).

3.2.7 8.6 Protocolo MSI basado en directorios con o sin difusión.

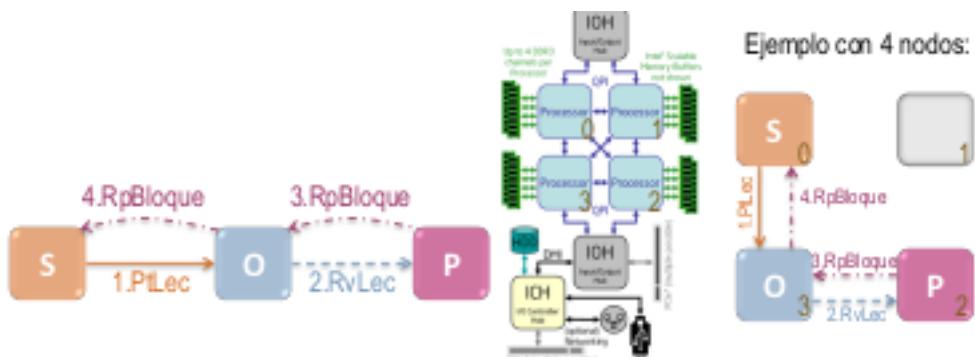
8.6.1 MSI con directorios (sin difusión).

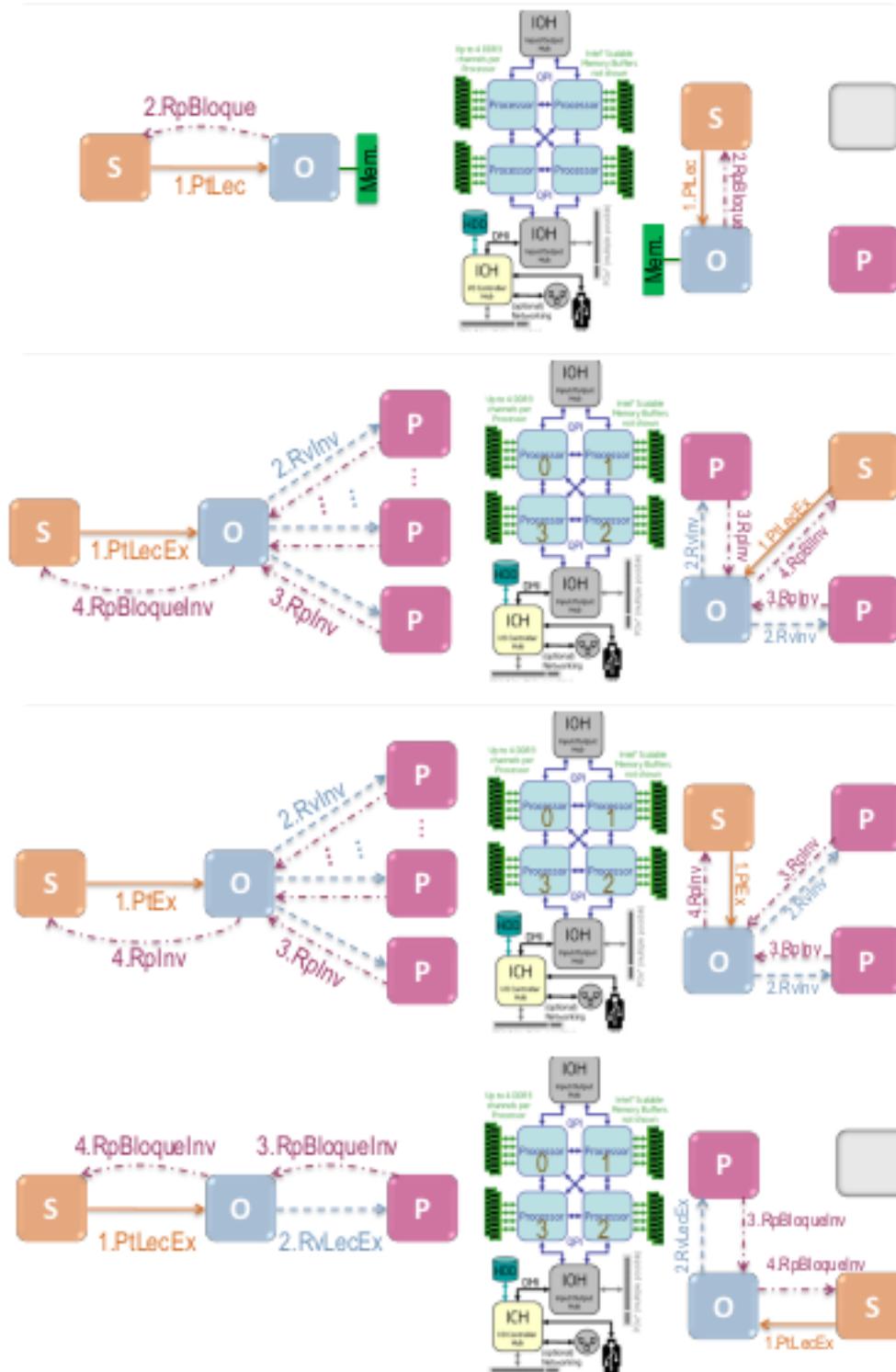
- Estados de un bloque en cache:
 - Modificado (M), Compartido (C), Inválido (I).
- Estados de un bloque en MP:
 - Válido e inválido.
- Transferencias (tipos de paquetes):
 - Tipos de nodos:
 - **Solicitante (S):** contiene el procesador que ha emitido una petición sobre el bloque.
 - **Origen (O):** el bloque tiene como origen el módulo de memoria de este nodo; la entrada del directorio para el bloque se encuentra en este nodo.
 - **Modificado (M):** único nodo que tiene una copia del bloque en su caché en estado modificado (dirty), no hay otro nodo con copia válida en caché y la memoria principal tampoco tiene copia válida. El nodo modificado y origen pueden coincidir.
 - **Propietario (P):** nodo con copia válida del bloque en su caché, y que por tanto puede suministrar el bloque. Puede ser el nodo origen, modificado, exclusivo o compartidor.
 - **Compartidor (C):** nodo con una copia válida del bloque en caché; forma parte de un grupo que comparte el bloque (en caché).

- Petición de nodo S a O:** lectura de un bloque (PtLec), lectura con acceso exclusivo (PtLecEx), petición de acceso exclusivo sin lectura (PtEx), posescritura (PtPEsc).
- Reenvío de petición de nodo O a nodos con copia (P, M, C):** invalidación (RvInv), lectura (RvLec, RvLecEx).
- Respuesta de**
 - nodo P a O:** respuesta con bloque (RpBloque), resp. con o sin bloque confirmando inv. (RpInv, RpBloqueInv).
 - nodo O a S:** resp. con bloque (RpBloque), resp. con o sin bloque confirmando fin inv. (RpInv, RpBloqueInv).



Estado Inicial	Evento	Estado final
D) Inválido	Fallo de lectura	D) Válido
S) Inválido		S) Compartido
P) Modificado		P) Compartido
Acceso remoto		



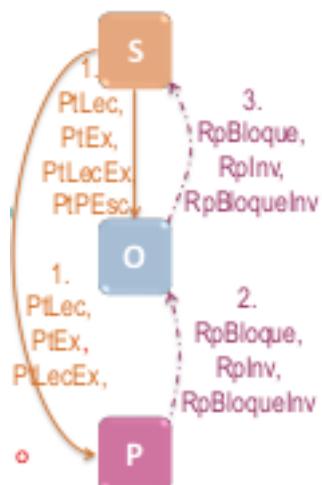


8.6.2 MSI con directorios (con difusión).

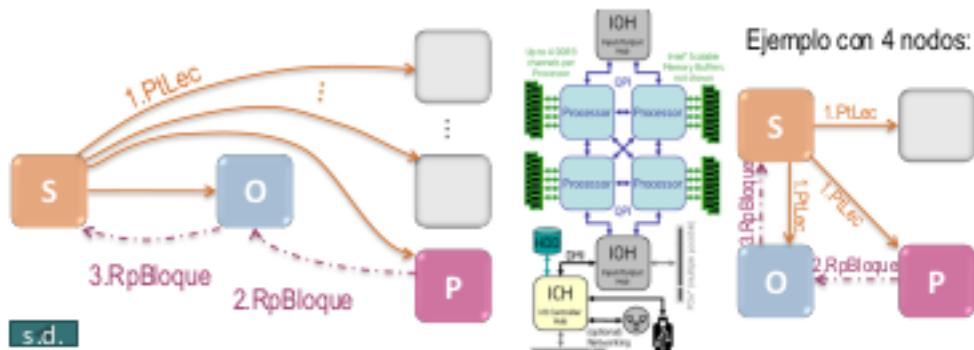
- Estados de un bloque en **cache**:
 - Modificado (M), Compartido (C), Inválido (I).
 - Estados de un bloque en **MP**:
 - Válido e inválido.

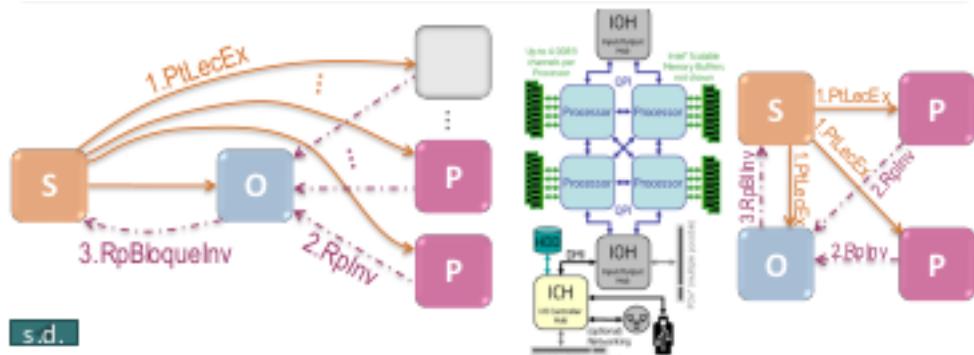
■ Transferencias (tipos de paquetes):

- **Tipos de nodos:** solicitante (S), origen (O), modificado (M), propietario (P) y compartidor (C).
- **Difusión** de petición del nodo S a
 - O y P: lectura de un bloque (PtLec), lectura con acceso exclusivo (PtLecEx), petición de acceso exclusivo sin lectura (PtEx).
 - O: posescritura (PtPEsc).
- **Respuesta de**
 - **nodo P a O:** respuesta con bloque (RpBloque), resp. con o sin bloque confirmando inv. (RpInv, RpBloqueInv).
 - **nodo O a S:** resp. con bloque (RpBloque), resp. con o sin bloque confirmando fin inv. (RpInv, RpBloqueInv).



Estado Inicial	Evento	Estado final
D) Inválido	Fallo de lectura	D) Válido
S) Inválido		S) Compartido
P) Modificado		P) Compartido
Acceso remoto		





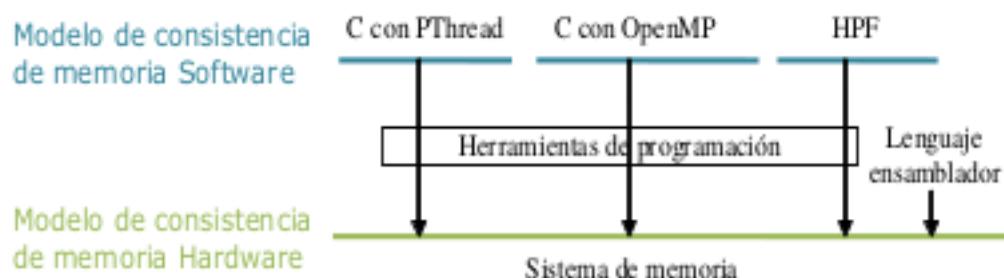
3.3 Lección 9. Consistencia del sistema de memoria.

3.3.1 Objetivos.

- Explicar el concepto de consistencia.
- Distinguir entre coherencia y consistencia.
- Distinguir entre el modelo de consistencia secuencial y los modelos relajados.
- Distinguir entre los diferentes modelos de consistencia relajados.

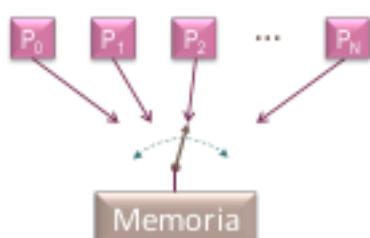
3.3.2 9.1 Concepto de consistencia de memoria.

- Especifica (las restricciones en) el **orden** en el cual las **operaciones de memoria** (lectura, escritura) deben parecer haberse **realizado** (operaciones a las mismas o distintas direcciones y emitidas por el mismo o distinto proceso/procesador).
- La coherencia sólo abarca operaciones realizadas por múltiples componentes (proceso/procesador) en una misma dirección.



3.3.3 9.2 Consistencia secuencial (SC).

- SC es el modelo de consistencia que espera el programador de las herramientas de alto nivel.
- SC requiere que:
 - Todas las operaciones de un único procesador (thread) parezcan ejecutarse en el orden descrito por el programa de entrada al procesador (**orden del programa**).
 - Todas las operaciones de memoria parezcan ser ejecutadas una cada vez (**ejecución atómica**) -> serialización global.
- SC presenta el sistema de memoria a los programadores como una memoria global conectada a todos los procesadores a través un conmutador central.



Inicialmente $k_1=k_2=0$	P1 $k_1=1;$ if ($k_2=0$) { Sección crítica };	P2 $k_2=1;$ if ($k_1=0$) { Sección crítica };	① ¿Qué espera el programador?
P1 $A=1;$	Inicialmente P2 if ($A=1$) $B=1;$	$A=B=0$ P3 if ($B=1$) $reg1=A;$	② ¿Qué espera el programador que se almacene en $reg1$ si llega a ejecutarse $reg1=A$?
Inicialmente $A=0, k=0$	P1 $A=1;$ $k=1;$	P2 while ($k=0$) {}; $copia=A;$	③ ¿Qué espera el programador que se almacene en $copia$? sin sincronización

- 1) (Orden del programa) El programador espera que solo un proceso pueda entrar en la sección crítica. Para ello no se debería permitir que las lecturas puedan adelantar escrituras, se debe mandar el orden W -> R.

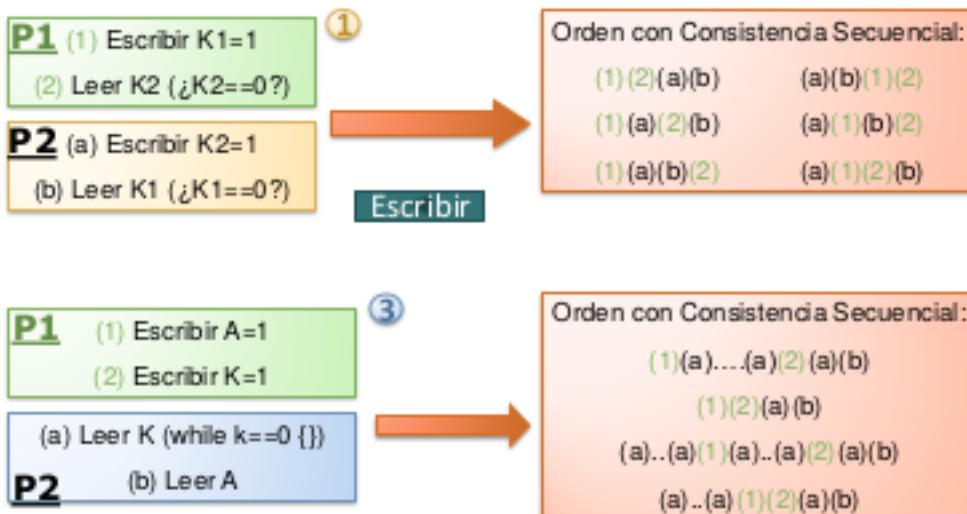
Si hay consistencia secuencial, esperaríamos que entraría uno de ellos, no a la vez. Si no, podrían entrar los dos a la vez. Se utiliza para obtener un acceso en exclusión mutua a una sección crítica (cuando no se dispone de primitivas atómicas de lectura-modificación-escritura); se basa en que k_1 y k_2 no pueden ser 0 al mismo tiempo (aunque tiene un problema de interbloqueo). Para que pueda actuar correctamente, no se puede permitir en un procesador que las lecturas adelanten a escrituras.

- 2) (Atomicidad) Los procesos comparten las variables A y B, que están inicialmente a 0. El programador espera que al finalizar la ejecución $reg1$ contenga 1. Supongamos que P2 lee 1 de A y entonces escribe en B, y supongamos que P3 lee 1 de B y a continuación lee de A. La ejecución atómica de las operaciones de acceso a memoria asegura que el valor escrito en A por P1 se ve en todo el sistema (por todos los procesadores) al mismo tiempo. Entonces, puesto que P3 ve la escritura en B de P2 después de que P2 vea la escritura en A de P1, se garantiza que P3 lee en $reg1$ lo que P1 escribe en A.

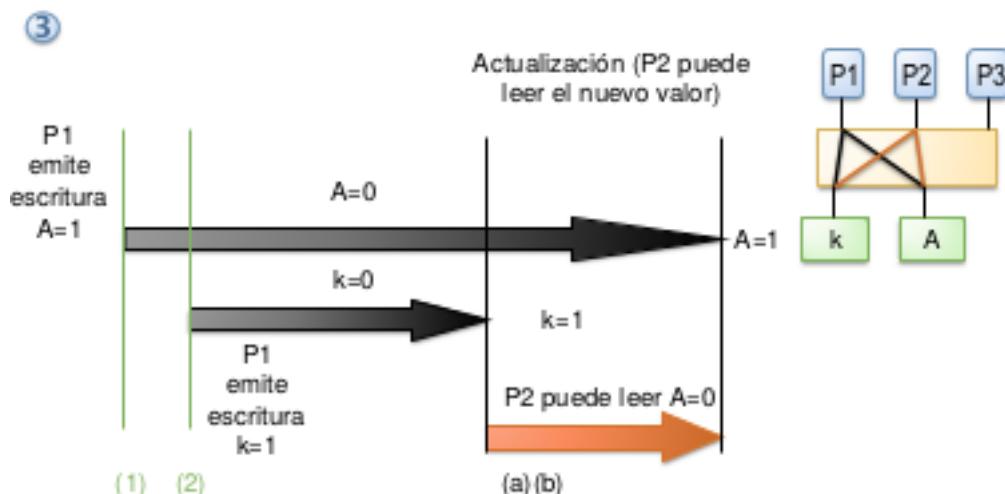
Para garantizar un funcionamiento correcto, la escritura de una dirección debe ser vista al mismo tiempo por todos los procesos. En caso contrario, podría ocurrir que P3 viera un valor de B de 1 y un valor de A de 0.

Ejemplo de consistencia secuencial:

Se cumple que se ejecuta (1) antes de (2) y (a) antes que (b).



¿Qué puede ocurrir en el computador?



3.3.4 9.3 Modelos de consistencia relajados.

- Difieren en cuanto a los requisitos para garantizar SC que relajan (los relajan para incrementar prestaciones):
 - **Orden del programa:**
 - Hay modelos que permiten que se relaje en el código ejecutado en un procesador el orden entre dos acceso a distintas direcciones (W→R, W→W, R→RW).
 - **Atomicidad (orden global):**
 - Hay modelos que permiten que un procesador pueda ver el valor escrito por otro antes de que este valor sea visible al resto de los procesadores del sistema.
- Los modelos relajados comprenden:
 - Los órdenes de acceso a memoria que no garantiza el sistema de memoria (tanto órdenes de un mismo procesador como atomicidad en las escrituras).
 - Mecanismos que ofrece el hardware para garantizar un orden cuando sea necesario.

Ejemplo de modelos de consistencia hardware relajados.

Modelo	Orden del programa relajado W→R W→W R→RW			Orden global Lec. anticipada propia de otro	Instrucciones para garantizar los órdenes relajados por el modelo
Sparc-TSO, x86-TSO	Si			Si	I-m-e (instruc. lectura-modificación-escritura atómica)
Sparc-PSO	Si	Si		Si	I-m-e, STBAR (instrucción Store BARrier)
Sparc-RMO	Si	Si	Si	Si	MEMBAR (instrucción MEMory BARrier)
PowerPC	Si	Si	Si	Si	SYNC, ISYNC (instrucciones SYNChronization)
Itanium	Si	Si	Si	Si	LD . ACQ, ST . REL, MF (ACQuisition Load, RELease Store, Memory Fence) y cmpxchgb . acq y otras I-m-e
ARMv7	Si	Si	Si	Si	DMB (Data Memory Barrier)
ARMv8	Si	Si	Si	Si	LDA LDAR, STL STLR (Load-Acquire, Store-reLease 32b 64b), LDAEX LDAXR, STLEX STLXR (Load-Acquire eXclusive, Store-reLease eXclusive 32b 64b), DMB

Consistencia secuencial:

Inicialmente $k_1=k_2=0$		P1 $k_1=1;$ if ($k_2=0$) { Sección crítica };	P2 $k_2=1;$ if ($k_1=0$) { Sección crítica };	NO se comporta como SC los que relajan el orden W→R ①
P1 $A=1;$		Inicialmente P2 if ($A=1$) $B=1;$	$A=B=0$ P3 if ($B=1$) $reg1=A;$	NO se comporta como SC los que no garantizan atomicidad ②
Inicialmente $A=0$		P1 $A=1;$ $k=1;$	P2 while ($k=0$) {}; copia= A ;	NO se comporta como SC los que relajan el orden W→W o R→R ③

9.3.1 Modelo que relaja W→R.

- Permiten que una lectura pueda adelantar a una escritura previa en el orden del programa; pero evita dependencias RAW.
 - Lo implementan los sistemas con buffer (FIFO) de escritura para los procesadores (el buffer evita que las escrituras retarden la ejecución del código bloqueando lecturas posteriores).
 - Generalmente permiten que el procesador pueda leer una dirección directamente del buffer (leer antes que otros procesadores una escritura propia)
- Para garantizar un orden correcto se pueden utilizar instrucciones de serialización.
- Hay sistemas en los que se permite que un procesador pueda leer la escritura de otro antes que el resto de procesadores (acceso no atómico).

- Para garantizar acceso atómico se puede utilizar instrucciones de lectura-modificación-escritura atómicas.

9.3.2 Modelo que relaja W->R y W->W.

- Tiene buffer de escritura que permite que lecturas adelanten a escrituras en el buffer.
- Permiten que el hardware solape escrituras a memoria a distintas direcciones, de forma que pueden llegar a la memoria principal o a caches de todos procesadores fuera del orden del programa.
- En sistemas con este modelo se proporciona hardware para garantizar los dos órdenes. Los sistemas con Sun Sparc implementa un modelo de este tipo.
- Este modelo no se comporta como SC en el siguiente ejemplo:

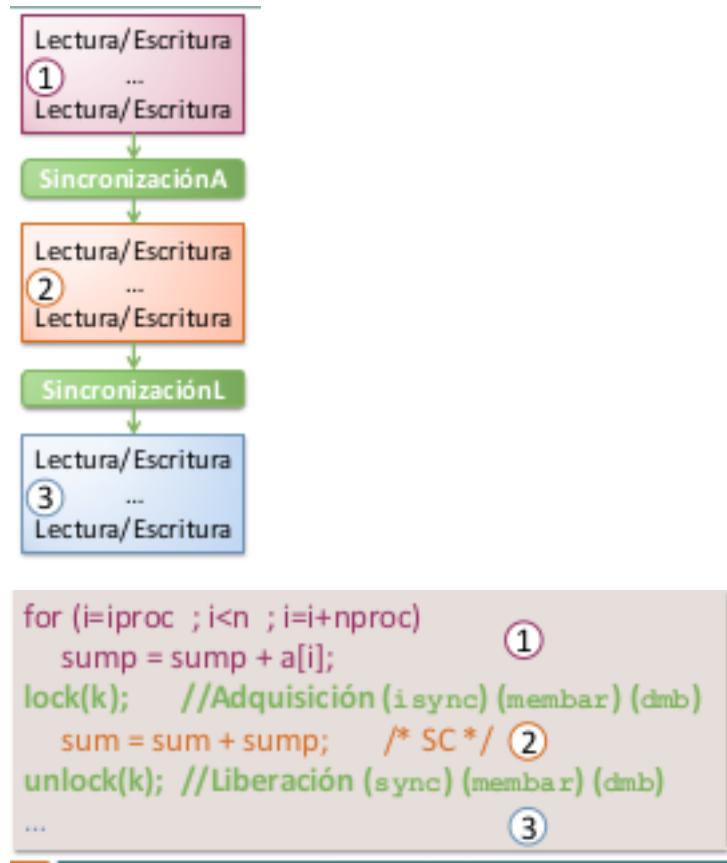
```

1 P1
2 A=1;
3 k=1;
4
5 P2
6 while (k=0){}
7 copia=A;
```

9.3.3 Modelo de ordenación débil.

- Se basa en mantener el orden entre accesos solo en los puntos de sincronización del código. Tiene en cuenta que cuando se necesita coordinar el acceso a una variable compartida (para permitir la comunicación de datos entre procesos) se añade código extra de sincronización.
- Relaja W->R, W->W y R->RW.
- Si S es una operación de sincronización (liberación o adquisición), ofrece hardware para garantizar el orden:
 - Una operación etiquetada como de sincronización se debe completar antes que las operaciones de acceso a memoria posteriores en el orden del programa, **S->WR**.
 - Las operaciones de acceso a memoria anteriores en el orden del programa a una operación etiquetada como de sincronización, se deben completar antes que las operaciones de acceso a memoria posteriores en el orden del programa, **WR->S**.
- PowerPC implementa un modelo basado en ordenación débil.

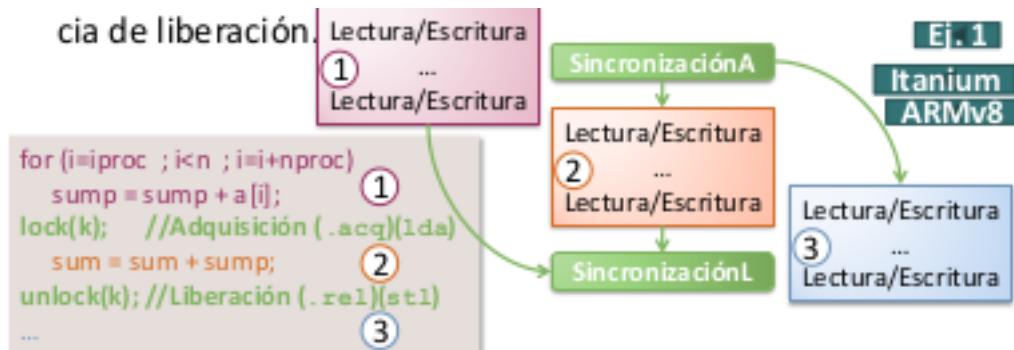
En la siguiente imagen vemos el modelo de ordenación débil. Las lecturas y escrituras (bloques 1, 2 y 3) se deben completar antes que una operación de sincronización posterior en el orden del programa, y viceversa.



9.3.4 Consistencia de liberación.

- Tiene en cuenta adicionalmente que hay dos tipos de código utilizado en el proceso de sincronización. Por una parte se añade código que el proceso ejecuta para ganar el acceso a variables o recursos compartidos. Si hay varios procesos implicados en el acceso, este código permite que solo uno de ellos gane el acceso. Por otra parte, se añade código que permite a un proceso dar permiso a otro para que pueda acceder a las variables o recursos compartidos. Luego distingue entre dos tipos de operaciones de sincronización: adquisición y liberación.
- El código utilizado para adquirir acceso se basa en leer una variable compartida o en una operación de lectura-modificación-escritura.
- Relaja W->R, W->W y R->RW.
- Si SA es una operación de adquisición y SL de liberación, ofrece hardware para garantizar el orden:
 - SA->WR y WR->SL.
- Sistemas con Itanium implementan un modelo de consistencia de liberación.

En la siguiente imagen vemos que la primera de las operaciones de sincronización es de adquisición y la segunda de liberación, se pueden solapar accesos de lectura y escritura de los bloques 1 y 2, y también se pueden solapar accesos de lectura y escritura de los bloques 2 y 3.



3.4 Lección 10. Sincronización.

3.4.1 Objetivos.

- Explicar por qué es necesaria la sincronización en multiprocesadores.
- Describir las primitivas para sincronización que ofrece el hardware.
- Implementar cerros simples, cerros con etiqueta y barreras a partir de instrucciones máquina de sincronización y ordenación de accesos a memoria.

3.4.2 10.1 Comunicación en multiprocesadores y necesidad de usar código de sincronización.

10.1.1 Comunicación uno-a-uno. Necesidad de sincronización. Se debe garantizar que el proceso que recibe lea la variable compartida cuando el proceso que envía haya escrito en la variable el dato a enviar. Si se reutiliza la variable para comunicación, se debe garantizar que no se envía un nuevo dato en la variable hasta que no se haya leído el anterior. En definitiva, se necesita algún mecanismo que asegure que solo un proceso puede estar accediendo en un momento dado a una dirección compartida (acceso en exclusión mútua). Una **región o sección crítica** es una secuencia de instrucciones con una o varias direcciones compartidas que se deben acceder en exclusión mútua.

En el siguiente código se ha utilizado para sincronizar una variable compartida `k` como bandera (*flag*). Se persigue que el proceso P2 acceda al nuevo valor de A, 1, cuando el proceso P1 haga 1 la bandera `k`.

```

1 // Paralela (inicialmente K=0)
2 P1
3 ...
4 A = 1;
5 K = 1;
6 ...
7
8 P2
9 ...
10 while (K == 0){};
11 copia = A;
12 ...

```

10.1.2 Comunicación colectiva. Hay que coordinar el acceso de múltiples procesos a una variable compartida, de forma que escriban uno detrás de otro (sin interferencias entre ellos) o lean cuando tengan disponibles los resultados definitivos en la memoria compartida. Puede haber uno o varios procesos que envían y/o uno o varios procesos que reciben. Se pueden combinar varios envíos en las variables antes de realizar la lectura por parte del o los procesos que reciben. No obstante, debe garantizarse que los procesos acceden a las variables compartidas sin interferir unos con otros, es decir, se necesita un acceso en exclusión mutua. Además, debe garantizarse que no se accede al resultado hasta que todos los procesos involucrados hayan ejecutado la sección crítica.

- Ejemplo de comunicación colectiva: suma de n números:

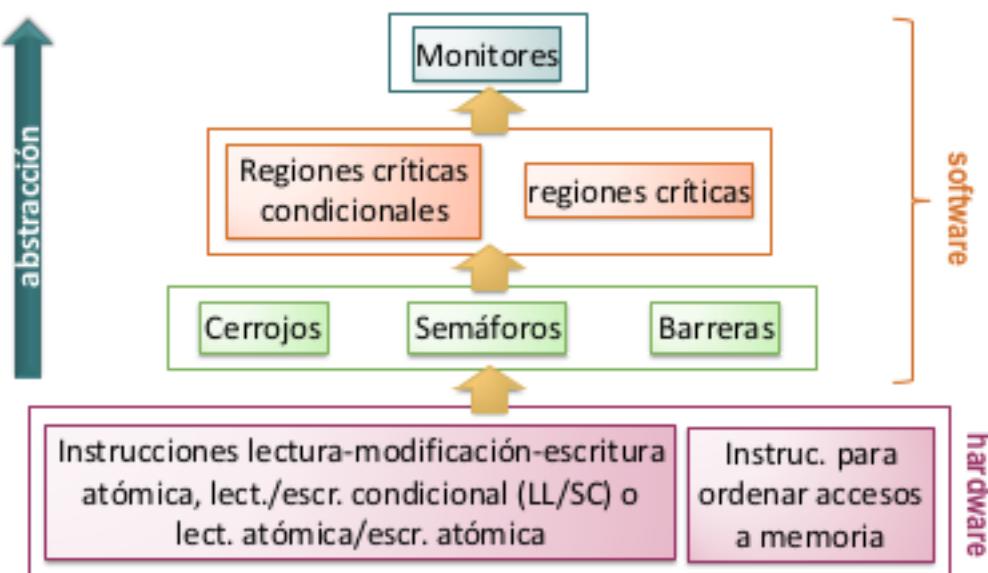
- La lectura-modificación-escritura de `sum` se debería hacer en exclusión mutua (es una sección crítica) => cerrojos.
- Sección crítica: Secuencia de instrucciones con una o varias direcciones compartidas (variables) que se deben acceder en exclusión mutua.
- El proceso 0 no debería imprimir hasta que no hayan acumulado `sum` en `sum` todos los procesos => barreras.

```

1 // Secuencial
2 for (i=0; i<n; i++)
3     sum = sum + a[i];
4 printf(sum);
5
6
7 // Paralela (sum=0)
8 for (i=ithread; i<n; i=i+nthread)
9     sump = sump + a[i];
10 sum = sum + sump;    // SC, sum compart
11 if (ithread == 0)
12     printf(sum);
13
14 // sump es compartida
15 // No se imprime siempre el mismo resultado, necesitaríamos cerrojos pero
   aun así se imprimiría mal porque lo imprime el thread 0. Habría que
   poner una barrera también y ya no habría más problemas.

```

3.4.3 10.2 Soporte software y hardware para sincronización.



3.4.4 10.3 Cerrojos.

- Permiten sincronizar mediante dos operaciones:
 - **Cierre del cerrojo o `lock(k)`**: intenta **adquirir** el derecho a acceder a una sección crítica (cerrando o adquiriendo el cerrojo `k`).
 - Si varios procesos intentan la **adquisición** (cierre) a la vez, sólo uno de ellos lo debe conseguir, el resto debe pasar a una **etapa de espera**.
 - Todos los procesos que ejecuten `lock()` con el cerrojo cerrado deben quedar **esperando**.
 - **Apertura del cerrojo o `unlock(k)`**: **libera** a uno de los threads que esperan el acceso a una sección crítica (éste adquiere el cerrojo).
 - Si no hay threads en **espera**, permitirá que el siguiente thread que ejecute la función `lock()` adquiera el cerrojo `k` sin espera.
- Cerrojos en ejemplo suma:
 - Alternativas para implementar la espera:
 - Espera ocupada.
 - Suspensión del proceso o thread, éste queda esperando en una cola, el procesador commuta a otro proceso-thread.

```

1 Secuencial
2 for (i=0; i<n; i++)
3     sum = sum + a[i];
4
5 Paralela
6 for (i=ithread; i<n; i=i+nthread)
7     sump = sump + a[i];
8 lock(k);
9 sum = sum + sump;    // SC, sum compart
10 unlock(k);

```

- Componentes en un código para sincronización.
 - Método de **adquisición**.
 - Método por el que un thread trata de adquirir el derecho a pasar a utilizar unas direcciones compartidas. Ej.:
 - ◊ Utilizando lectura-modificación-escritura atómicas: Intel x86, Intel Itanium, Sun Sparc.
 - ◊ Utilizando LL/SC (Load Linked / Store Conditional): IBM Power/PowerPC, ARMv7, ARMv8.
 - Método de **espera**.
 - Método por el que un thread espera a adquirir el derecho a pasar a utilizar unas direcciones compartidas:

- ◊ Espera ocupada (busy-waiting).
- ◊ Bloqueo.
- Método de **liberación**. - Método utilizado por un thread para liberar a uno (cerrojo) o varios (barrera) threads en espera.

3.4.5 10.3.1 Cerrojos simples.

- Se implementa con una variable compartida k que toma dos valores: abierto (0), cerrado (1).
- **Apertura** del cerrojo, `unlock(k)`: abre el cerrojo escribiendo un 0 (operación **indivisible**).
- Cierre del cerrojo, `lock(k)`: Lee el cerrojo y lo cierra escribiendo un 1.
 - Resultado de la lectura:
 - si el cerrojo estaba **cerrado** el thread espera hasta que otro thread ejecute `unlock(k)`,
 - si estaba **abierto** adquiere el derecho a pasar a la sección crítica.
 - `leer-asignar_1-escribir` en el cerrojo debe ser **indivisible (atómica)**.
- Se debe añadir lo necesario para garantizar el acceso en exclusión mutua a k y el orden imprescindible en los accesos a memoria.

```

1 lock(k) {
2     while (leer-asignar_1-escribir(k) == 1) {} ;
3 } // k compartida
4
5 unlock(k) {
6     k = 0 ;
7 } // k compartida

```

- Cerrojos en OpenMP:

Descripción	Función de la biblioteca OpenMP
Iniciar (estado <code>unlock</code>)	<code>omp_init_lock(&k)</code>
Destruir un cerrojo	<code>omp_destroy_lock(&k)</code>
Cerrar el cerrojo <code>lock(k)</code>	<code>omp_set_lock(&k)</code>
Abrir el cerrojo <code>unlock(k)</code>	<code>omp_unset_lock(&k)</code>
Cierre del cerrojo pero sin bloqueo (devuelve 1 si estaba cerrado y 0 si está abierto)	<code>omp_test_lock(&k)</code>

3.4.6 10.3.2 Cerrojos con etiqueta.

Fijan un orden FIFO en la adquisición del cerrojo (se debe añadir lo necesario para garantizar el acceso en exclusión mutua al contador de adquisición y el orden imprescindible en los accesos a memoria):

```

1 // lock (contadores)
2 contador_local_adq = contadores.adq; //contadores.adq inicialmente es 0
3 contadores.adq = (contadores.adq+1) mod max_flujos;
4 while (contador_local_adq <> contadores.lib){} // <> es !=
5
6 // unlock (contadores)
7 contadores.lib = (contadores.lib + 1) mod max_flujos;

```

3.4.7 10.3.3 Barreras.

```

1 //Thread 0 (lo mismo para 1, 2, 3)
2 main (){
3     ...
4     Barrera(g,4)
5     ...
6 }

```

El siguiente algoritmo para barreras presenta problemas si los procesos reutilizan la misma barrera, por ejemplo dentro de un bucle, ya que se puede dar la siguiente situación:

1. En la primera utilización de la barrera el SO reemplaza algún proceso P_i que está esperando en la bandera a ser liberado.
2. A continuación llega el último proceso a la barrera activando la variable bandera.
3. Los procesos que están en espera, excepto P_i que está suspendido, verán la bandera a 1 y saldrán del bucle de espera ocupada, abandonando la barrera.
4. Un proceso P_j llega a la segunda función que utiliza la misma barrera desactivando la bandera.
5. P_i vuelve a ejecutarse, pero encuentra la bandera cerrada por lo que se queda en bucle. P_i no supera nunca la primera ejecución de la barrera, y el resto de procesos no supera la segunda ejecución, también se quedarán en el bucle de espera. Ocurre que en la segunda utilización de la barrera el contador no va a poder alcanzar un valor igual al número de procesos, ya que uno de ellos se ha quedado en la primera llamada a la barrera.

```

1 Barrera (id, num_threads){
2     if (bar[id].cont == 0)
3         bar[id].bandera = 0;           // Acceso Ex. Mutua
4
5     cont_local = ++bar[id].cont;    // Accesp Ex. Mutua
6
7     if (cont_local == num_threads){
8         bar[id].cont = 0;
9         bar[id].bandera = 1;
10    }
11    else
12        espera mientras bar[id].bandera = 0; // Implementar espera. Si
13            espera ocupada: while (bar[id].bandera == 0){}

```

13 }

- **Barreras sin problema de reutilización.** Se modifica el código para que cada vez que se reutilice la bandera, los procesos para salir esperen una condición distinta. Se va a cambiar la condición para la liberación entre usos consecutivos de la barrera. Si en la última utilización han esperado para salir a que la bandera sea 1, en la siguiente van a esperar para salir que esta sea 0, y en la siguiente 1.

```

1 // Barrera sense-reversing
2 Barrera(id, num_procesos) {
3     bandera_local = !(bandera_local) //se complementa bandera local
4     lock(bar[id].cerrojo);
5     cont_local = ++bar[id].cont      //cont_local es privada
6     unlock(bar[id].cerrojo);
7
8     if (cont_local == num_procesos) {
9         bar[id].cont = 0;           //se hace 0 el cont. de la barrera
10        bar[id].bandera = bandera_local; //para liberar thread en espera
11    }
12    else
13        while (bar[id].bandera != bandera_local) {}; //espera ocupada
14 }
```

3.4.8 10.3.4 Apoyo hardware a primitivas software.

```

1 // Test&Set (x)
2 Text&Set(x){
3     temp = x;
4     x = 1;
5     return (temp);
6 }
7 // x compartida
```

Se traduce en:

```

1 mov reg, 1
2 xchg reg, mem
3 reg <-> mem
4 # no hace falta poner lock

1 // Fetch&Oper(x,a)
2 Fetch&Add(x,a) {
3     temp = x ;
4     x = x + a ;
```

```

5         return (temp);
6     }
7 // x compartida, a local
8

```

Se traduce en:

```

1 lock xadd reg,mem
2 reg <- mem |
3 mem <- reg+mem

1 // Compare&Swap(a,b,x)
2 Compare&Swap(a,b,x){
3     if (a==x) {
4         temp=x;
5         x=b;
6         b=temp;
7     }
8 }
9 // x compartida, a y b locales

```

Se traduce en:

```

1 lock cmpxchg mem,reg
2 if eax=mem
3 then mem <- reg
4 else eax <- mem

```

```

1 // Con Test&Set (x)
2 lock (k){
3     while (test&set(k) == 1){};
4 }
5 // k compartida

```

Se traduce en:

```

1 lock:    mov    eax,1
2 repetir:  xchg   eax,k
3           cmp    eax,1
4           jz     repetir

1 // Con Fetch&Oper(x,a)
2 lock (k){
3     while (fetch&or (k,1) == 1){};      // true (1, cerrado)
4                               // false (0, abierto)

```

```

5 }
6 // k compartida

```

```

1 // Con Compare&Swap(a,b,x)
2 lock (k){
3     b = 1
4     do                                // compare&swap(0,b,k){
5         compare&swap(0,b,k);          // if (0 == k) { b=k | k=b; }
6         while (b == 1);             }
7 }
8 // k compartida, b local

```

```

1 // Compare&Swap
2 lock:                                //lock(M[lock])
3     mov ar.ccv = 0                     // cmpxchg compara con ar.ccv
4                               // que es un registro de propósito
5                               // específico
5     mov r2 = 1                        // cmpxchg utilizará r2 para poner el
6     cerrojo a 1                      // cerrojo a 1
6 spin:                                 // se implementa espera ocupada
7     ld8 r1 = [lock];;                // carga el valor actual del cerrojo
8     en r1
8     cmp.eq p1,p0 = r1, r2;          // si r1=r2 entonces cerrojo está a 1
9     y se hace p1=1
9     (p1) br.cond.spnt spin ;;      // si p1=1 se repite el ciclo; spnt
10    indica que se usa una           // predicción estática para el salto
10    // de 'no tomar'
11    cmpxchg8.acq r1 = [lock], r2 ;; //intento de adquisición escribiendo 1
12                               // IF [lock]=ar.ccv THEN [lock]<-r2;
12                               // siempre r1<-[lock]
13    cmp.eq p1, p0 = r1, r2          // si r1!=r2 (r1=0) => cer. era 0 y se
14    hace p1=0
14    (p1) br.cond.spnt spin ;;      // si p1=1 se ejecuta el salto
15 // Consistencia
16 unlock:                                //unlock(M[lock])
17     st8.rel [lock] = r0 ;;          //liberar asignando un 0, en Itanium
17     r0 siempre es 0

```

```

1 //Test&Set
2 lock:                      #lock(M[r3])
3     li      r4,1           #para cerrar el cerrojo
4 bucle: lwarx   r5,0,r3    #carga y reserva: r5<-M[r3]
5     cmpwi   r5,0           #si está cerrado (a 1)
6     bne-    bucle         #esperar en el bucle, en caso contrario
7     stwcx.  r4,0,r3       #poner a 1 (r4=1): M[r3] <- r4
8     bne-    bucle         #el thread repite si ha perdido la reserva
9     isync               #accede a datos compartidos cuando sale del
10    bucle
11
12 // Consistencia
13 unlock:                   # unlock(M[r3])
14     sync                  #espera hasta que terminen los accesos
15     anteriores
16     li      r1,0
17     stw    r1,0(r3)      #abre el cerrojo

```

```

1 // Test&Set
2 lock:                      #lock(M[r1])
3     mov     r0,#1           #Para posteriormente cerrar el cerrojo asigna
4             a r0 un 1
5 bucle: ldrex   r5,[r1]    #Lee cerrojo
6     cmp     r5,#0           #Comprueba si el cerrojo es 0 (abierto)
7     strexeq r5,r0,[r1]    #Si el cerrojo está abierto intenta escribir (
8             un 1)
9     cmpeq   r5,#0           #Comprueba si ha tenido éxito la escritura
10    bne     bucle          #Vuelve a intentarlo si no ha tenido éxito
11    dmb                 #Para asegurar que se ha adquirido el cerrojo
12             antes de ...
13
14             #realizar los accesos a memoria que hay despué
15             s del lock
16
17 //Consistencia
18 unlock:                   # unlock(M[r1])
19     dmb                  #Espera a que terminen los accesos anteriores
20
21     ...
22
23     mov     r0, #0           #antes de abrir el cerrojo
24     str     r0, [r1]        #abre el cerrojo

```

1 //Test&Set

```

2 lock:                      #lock(M[r1])
3     mov      r0,#1          #Para posteriormente cerrar el cerrojo asigna
4         a r0 un 1
5 bucle: ldaex   r5,[r1]    #Lee cerrojo con ordenación de adquisición
6         cmp      r5,#0      #Comprueba si el cerrojo es 0 (abierto)
7         strexeq r5,r0,[r1]  #Si el cerrojo está abierto intenta escribir (
8             un 1)
9         cmpeq   r5,#0      #Comprueba si ha tenido éxito la escritura
10        bne     bucle      #Vuelve a intentarlo si no ha tenido éxito
11 //Consistencia de liberación
12 unlock:                   #unlock(M[r1])
13     mov      r0,#0
14     stl      r0,[r1]      #Abre el cerrojo usando almacenamiento con
15         liberación

```

```

1 // NO
2 // Suma con fetch&add
3 for (i=ithread; i<n; i=i+nthread)
4     fetch&add(sum,a[i]);
5 //sum variable compartida

1 // Suma con fetch&add
2 for (i=ithread; i<n; i=i+nthread)
3     sump = sump + a[i];
4 fetch&add(sum,sump);
5 // sum variable compartida

1 // Suma con compare&swap
2 for (i=ithread; i<n; i=i+nthread)
3     sump = sump + a[i];
4 do
5     a = sum;
6     b = a + sump;
7     compare&swap(a,b,sum);
8 while (a!=b);
9 // sum variable compartida

```

4 Bibliografía

10.3.4.8 Algoritmos eficientes con primitivas hardware. Ortega, M. Anguita, A. Prieto. Arquitectura de Computadores, Thomson, 2005.