

Teoría

Arquitectura de Computadores

2º DGIIM

María Aguado Martínez



UNIVERSIDAD
DE GRANADA



Este trabajo 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/

Teoría

Arquitectura de Computadores

2º DGIIM

María Aguado Martínez



UNIVERSIDAD
DE GRANADA

1531

Índice

| | |
|--|----------|
| I Tema 1: Arquitecturas Paralelas. Clasificación y prestaciones. | 6 |
| 1. Lección 1: Clasificación del paralelismo implícito en una aplicación | 6 |
| 1.1. Dependencias de datos | 7 |
| 1.2. Paralelismo implícito en una aplicación | 7 |
| 1.3. Unidades en ejecución en un computador a nivel de paralelismo explícito | 8 |
| 1.3.1. Instrucciones | 8 |
| 1.3.2. Thread o <i>light process</i> | 8 |
| 1.3.3. Proceso | 8 |
| 2. Lección 2: Clasificación de arquitecturas paralelas | 9 |
| 2.1. Terminología | 9 |
| 2.2. Computación paralela y distribuida | 11 |
| 2.2.1. Computación paralela | 11 |
| 2.2.2. Computación distribuida | 11 |
| 2.3. Clasificaciones de arquitecturas y sistemas paralelos | 12 |
| 2.3.1. Según el segmento del mercado | 12 |
| 2.3.2. Otras clasificaciones | 13 |
| 3. Lección 3: Evaluación de prestaciones | 17 |
| 3.1. Medidas usuales para evaluar prestaciones | 17 |
| 3.1.1. Tiempo de respuesta | 17 |
| 3.1.2. Tiempo de CPU | 18 |
| 3.1.3. Mejoras en las prestaciones | 18 |
| 3.2. Productividad | 19 |
| 3.3. Otras medidas | 20 |
| 3.4. Ganancia en prestaciones al realizar una mejora | 21 |
| 3.5. Mejora por segmentación | 21 |
| 3.5.1. Ganancia pico | 22 |

| | |
|--|-----------|
| 3.5.2. Ley de Amdahl | 22 |
| 3.6. Conjunto de programas <i>Benchmark</i> | 22 |
| 3.7. Tipos de benchmarks | 23 |
| II Tema 2: Programación paralela | 24 |
| 1. Lección 4: Herramientas, estilos y estructuras | 24 |
| 1.1. Problemas que plantea la programación paralela al programador | 24 |
| 1.2. Herramientas para la programación paralela | 24 |
| 1.2.1. Comunicaciones colectivas | 25 |
| 1.3. Estilos de programación paralela | 29 |
| 1.4. Estructuras típicas de códigos paralelos | 30 |
| 1.5. Buena asignación de tareas | 32 |
| 2. Lección 5: Proceso de paralelización | 32 |
| 2.1. Descomposición en tareas independientes | 32 |
| 2.2. Asignación de tareas a procesos y threads | 33 |
| 2.2.1. Granularidad | 33 |
| 2.2.2. Equilibrado de la carga | 33 |
| 2.2.3. Tipos de asignación | 34 |
| 2.2.4. Mapeo de procesos a unidades de procesamiento | 35 |
| 2.3. Redactar código paralelo | 35 |
| 2.3.1. Evaluación de prestaciones | 35 |
| 3. Lección 6: Evaluación de prestaciones | 35 |
| 3.1. Escalabilidad | 36 |
| 3.2. Ley de Amdahl | 37 |
| 3.3. Ganancia escalable | 38 |
| 3.3.1. Escalabilidad débil o Ley de Gustafson | 38 |
| 4. Ejercicios resueltos | 39 |
| 4.1. Ejercicio 2 | 39 |
| 4.1.1. Solución | 39 |

| | |
|---|-----------|
| 4.2. Ejercicio 4 | 39 |
| 4.2.1. Solución | 40 |
| III Tema 3: Arquitecturas con paralelismo a nivel de thread | 41 |
| 1. Lección 7: Arquitecturas TLP (Thread-Level-Paralelism): | 41 |
| 1.1. Clasificación de arquitecturas con TLP explícito y una instancia de SO | 41 |
| 1.2. MULTIPROCESADORES | 41 |
| 1.2.1. Clasificación | 41 |
| 1.2.2. Evolución UMA-NUMA en una placa | 42 |
| 1.3. MULTICORES | 42 |
| 1.4. CORES MULTITHREAD | 43 |
| 1.5. Hardware y arquitecturas TLP en un chip | 47 |
| 2. Lección 8: Coherencia en el sistema de memoria | 47 |
| 2.1. Sistema de memoria en multiprocesadores | 47 |
| 2.2. Concepto de coherencia en el sistema de memoria | 48 |
| 2.2.1. Métodos de actualización de Memoria Principal | 48 |
| 2.2.2. Alternativas de Propagación de Escrituras | 49 |
| 2.2.3. Requisitos para mantener coherencia | 49 |
| 2.2.4. Directorio | 50 |
| 2.3. Protocolos de mantenimiento de coherencia | 50 |
| 2.3.1. Protocolo MSI de espionaje | 51 |
| 2.3.2. Protocolo MESI de espionaje | 52 |
| 2.3.3. Protocolo MSI basado en directorios | 53 |
| 3. Lección 9: Consistencia del sistema de memoria | 54 |
| 3.1. Concepto de consistencia de memoria | 54 |
| 3.2. Consistencia secuencial | 55 |
| 3.2.1. Modelos de consistencia relajados | 56 |
| 3.2.2. Modelo que relaja W-R | 57 |
| 4. Lección 10: Sincronización | 57 |

| | |
|---|-----------|
| 4.1. Comunicación en multiprocesadores y necesidad de usar código de sincronización | 57 |
| 4.1.1. Comunicación uno-uno | 57 |
| 4.1.2. Comunicación colectiva y condiciones de carrera | 58 |
| 4.2. Soporte software y hardware de sincronización | 58 |
| 4.3. Cerrojos | 59 |
| 4.3.1. Definición y características | 59 |
| 4.3.2. Componentes en un código para sincronización | 59 |
| 4.3.3. Implementaciones | 60 |
| 4.4. Barreras | 61 |
| 4.4.1. Problema con barreras reutilizadas | 62 |
| 4.5. Apoyo hardware a primitivas software | 62 |
| 4.5.1. Instrucciones de lectura-modificación-escritura atómicas | 62 |
| 4.5.2. Cerrojos simples con instrucciones atómicas | 63 |
| 4.5.3. Consistencia de liberación | 64 |
| 4.5.4. Algoritmos eficientes con primitivas hardware | 64 |
| IV Tema 4. Arquitecturas con Paralelismo a nivel de Instrucción | 66 |
| 1. Riesgos y dependencias | 66 |
| 2. Microarquitecturas ILP superescalares | 66 |
| 2.1. Dependencias | 66 |
| 2.1.1. Ejemplo de dependencias | 67 |
| 2.2. Hardware en superescalares | 67 |
| 2.2.1. Etapas del cauce | 68 |
| 2.2.2. Emisión | 68 |
| 2.2.3. Consistencia del procesador y buffer de reorden | 70 |
| 2.2.4. Buffer de Reordenamiento (ROB) | 71 |
| 2.2.5. Ejecución especulativa | 72 |
| 3. Microarquitecturas VLIW | 73 |
| 3.1. Riesgos y dependencias | 73 |

| | |
|--|----|
| 3.2. Características generales | 74 |
| 3.3. Planificación Estática | 74 |
| 3.3.1. Tipos de planificación Estática | 75 |
| 3.4. Instrucciones de salto | 77 |

I | Tema 1: Arquitecturas Paralelas. Clasificación y prestaciones.

1 Lección 1: Clasificación del paralelismo implícito en una aplicación

En una aplicación podemos tener distintos niveles de paralelismo implícito:

| Niveles | Granularidad |
|----------------|------------------|
| Programas | Grano grueso |
| Funciones | Grano medio |
| Bucle(bloques) | Grano medio-fino |
| Operaciones | Grano fino |

■ Paralelismo 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.
- **Paralelismo 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 paralelismo de datos está relacionado con el paralelismo a nivel de bucle.
- **Paralelismo 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 paralelismo entre las funciones.

Además cabe destacar que podemos encontrar paralelismo en cada nivel **siempre que no haya dependencia entre ellos**.

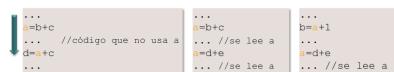
1.1 Dependencias de datos

Definición 1.1. Se dice que un bloque de código B_2 **depende de** B_1 si hacen referencia a una misma posición de memoria y B_1 aparece en la secuencia de código antes que B_2 . Es decir, que se accede desde B_1 a una posición de memoria a la que se accederá después desde B_2 .

Tenemos varios tipos de dependencias de datos:

1. **RAW**: Read after write, o *dependencia verdadera*.
2. **WAW**: Write after write, o *dependencia de salida*.
3. **WAR**: Write after read, o *antidependencia*.

Las dos últimas se pueden resolver si escogemos otra variable en la que escribir.



1.2 Paralelismo implícito en una aplicación

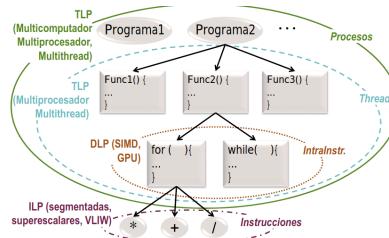
En primer lugar tenemos distintos tipos de paralelismos:

1. **Paralelismo de tareas (TLP)**
 - Se encuentra extrayendo la estructura lógica de funciones de una aplicación.
 - Está relacionado con el paralelismo *a nivel de función*.
2. **Paralelismo de datos (DLP)**
 - Se encuentra implícito en las operaciones con estructuras de datos, como operaciones **con vectores y matrices**.
 - Se puede extraer de la representación matemática de la aplicación.
 - Está relacionado con el paralelismo *a nivel de bucle*.

Además, podemos clasificar la **detección del paralelismo implícito**

- **Procesos**: mediante TLP (multiprocesador, multicomputador, multithread).
- **Thread**: mediante TLP (multiprocesador, multithread).
- **Intrainstrucciones- bucles**: mediante DLP (SIMD, GPU). 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.

- **Instrucciones:** mediante ILP (segmentadas, superescalares, VLIW).



1.3 Unidades en ejecución en un computador a nivel de paralelismo explícito

1.3.1 Instrucciones

La unidad de control de un core o procesador gestiona la ejecución de instrucciones por la unidad de procesamiento.

1.3.2 Thread o *light process*

Definición 1.2. Es la menor unidad de ejecución que gestiona el SO. Se dice también que es la menor secuencia de instrucciones que se pueden ejecutar en paralelo o concurrentemente.

- Un proceso puede constar de múltiples flujos de instrucciones (**threads**). Además, cada thread tiene:
 - Su propia pila.
 - Contenido de los registros, en particular el *contador de programa* y el *puntero de pila* (instruction pointer y stack pointer).
- Para comunicar threads de un proceso se usa la memoria que comparten.

1.3.3 Proceso

Definición 1.3. Es la mayor unidad de ejecución que gestiona el SO.

Algunas características son:

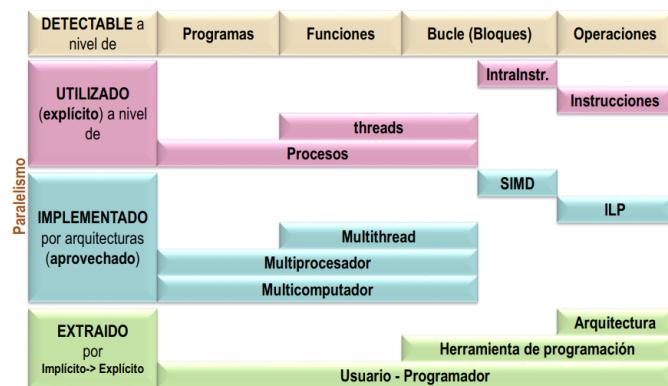
- Un proceso comprende uno o varios thread.
- Comprende el código del programa y todo lo que necesita para su ejecución:
 - Datos en pila, segmentos (variables globales y estáticas) y en heap.
 - Contenido de los registros.

- Tabla de páginas.
- Tabla de ficheros abiertos.
- Para comunicar procesos hay que usar llamadas al SO.

Resulta obvio que se va a tener menor granularidad para threads, pues el uso de threads implica:

1. Destrucción de threads en menor tiempo.
2. Comutación en menor tiempo.
3. Comunicación en menor tiempo.
4. Creación de threads en menor tiempo.

La siguiente tabla representa la detección, utilización, implementación y extracción del paralelismo en los distintos niveles:



2 Lección 2: Clasificación de arquitecturas paralelas

2.1 Terminología

Definición 2.1. El **núcleo** o **core** es un hardware que se encarga de captar de memoria y ejecutar un flujo de instrucciones. Se compone de una Unidad de Control (UC) y al menos una Unidad de Procesamiento (UP). También puede referirse sólo a la Unidad de Procesamiento, según el contexto.

Definición 2.2. Un **multicore** es un chip multiprocesador (CMP) que se compone de un encapsulado de varios dados de silicio con varios cores.

Definición 2.3. Un **chip de procesamiento** se puede referir a un encapsulado como el mencionado anteriormente o a un dado de silicio, donde tendremos los núcleos de procesamiento.

Definición 2.4. El **procesador** es un núcleo de procesamiento o un chip de procesamiento.

Intel empezó a llamarlo microprocesador cuando logró integrar el procesador en un único dado.

Definición 2.5. La **CPU** es la torre del computador de sobremesa, chip o núcleo de procesamiento. Se puede referir incluso a los procesadores físicos. Es un concepto muy ambiguo.

Definición 2.6. La **cache** es una memoria más rápida y de menor capacidad que la principal. Se puede dar a varios niveles, normalmente dos o tres niveles. Además se dirá que tiene una **latencia in-chip** si su latencia es de pocos o decenas de ciclos, o que tiene una **latencia off-chip** si su latencia es de decenas o cientos de ciclos.

Definición 2.7. La **Memoria Principal** es una memoria cuya latencia son cientos de ciclos de reloj y cuya productividad está limitada por el número de pines dedicados a la transferencia y por la frecuencia de los mismos.

Vemos que a medida que avanzamos en la siguiente lista las prestaciones y el coste aumentan pero disminuye el tiempo, tardan más:

- Registros
- Memoria local (en computadores empotrados)
- Caches
- Memoria Principal
- Memoria Virtual

Definición 2.8. La **Computación concurrente** se da cuando se ejecutan flujos de instrucciones compartiendo recursos multiplexados. Es decir, que usan los mismos recursos en distinto tiempo.

Definición 2.9. La **Computación paralela** se da cuando se ejecutan flujos de instrucciones a la vez cada uno en recursos distintos replicados.

2.2 Computación paralela y distribuida

2.2.1 Computación paralela

La **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).

2.2.2 Computación distribuida

La **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**.

Tenemos varios tipos de computación distribuida:

1. Computación distribuida baja escala

- Estudia los 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.

2. Computación distribuida a gran escala

a) 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**.

b) Computación cloud

- Comprende los aspectos relacionados con el desarrollo y ejecución de aplicaciones en un sistema cloud.
- Características de un 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.
 - 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 (multi-tenant).
 - Están conectados con métodos estándar independientes de la plataforma de acceso.

2.3 Clasificaciones de arquitecturas y sistemas paralelos

Podemos clasificar arquitecturas según varios criterios:

1. Comercial, según el segmento de mercado
2. Educación e investigación, incluyen clasificaciones según:
 - Flujos de instrucciones y flujos de datos.
 - Sistema de memoria.
 - Flujos de instrucciones (propuesta de clasificación de arquitecturas con múltiples flujos de instrucciones).
 - Nivel de paralelismo aprovechado.

2.3.1 Segundo el segmento del mercado

Según el volumen de venta podemos clasificarlos en:

- Supercomputadores
- Servidores de gama alta
- Servidores de gama media
- Servidores de gama baja
- PC/WS
- Mercado de computadores empotrados (*móviles, lavadoras, consolas,...*), tienen más restricciones que un computador externo, por ejemplo de consumo o de potencia.

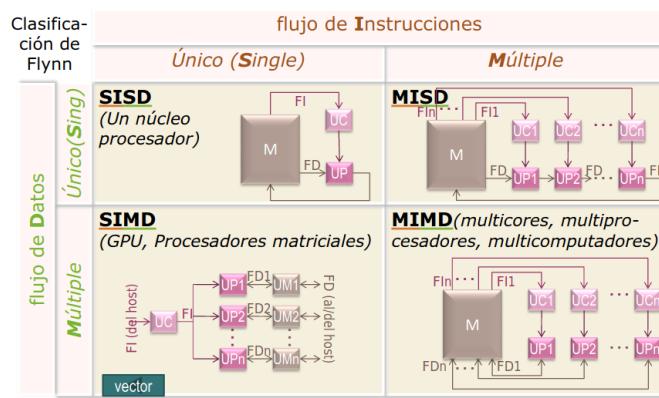
En general podemos hacer la siguiente división de computadores según el segmento:

1. Externo (*desktop, laptop, server, cluster*).
 - Sirven 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, etc.) y animación (películas animadas, efectos especiales, etc.), ...
 - Se clasifican en
 - WS/PC
 - Servidores básicos
 - Servidores de gama media

- Servidores de gama alta
 - Supercomputadores
2. Empotrado (oculto)
- Sirven para aplicaciones de propósito específico (videojuegos, teléfonos, coches, electrodomésticos,...)
 - Tienen restricciones como las ya mencionadas: potencia, precio, tamaño y tiempo.

2.3.2 Otras clasificaciones

1. CLASIFICACIÓN DE FLYNN

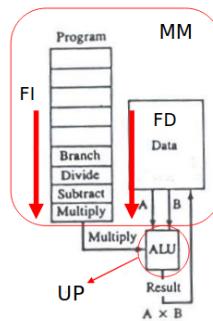


- Arquitecturas SISD, que corresponde a los computadores uniprocesador, se ejecuta todo secuencialmente. Consta de una Unidad de Procesamiento y una Unidad de Control, y sigue el ciclo de ejecución de instrucciones básico.

- Descripción estructural

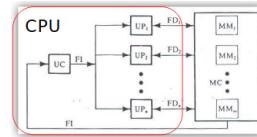


- Descripción funcional

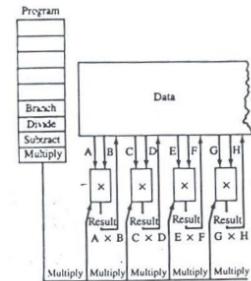


- Arquitecturas SIMD, que corresponde a los procesadores matriciales y vectoriales, y **aprovechan paralelismo de datos**. Disponen de varias unidades de Procesamiento y una Unidad de Control. Además, todas las unidades de procesamiento realizan operaciones parecidas, pero con datos distintos.

- Descripción estructural

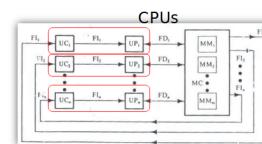


- Descripción funcional

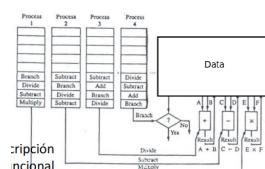


- Arquitecturas MIMD, que corresponde Multinúcleos, Multiprocesadores y Multicomputadores: Puede aprovechar, además, **paralelismo funcional**. Disponen de varias unidades de Procesamiento y varias Unidades de Control. Por tanto, cada unidad de control puede ejecutar flujos de instrucciones distintos.

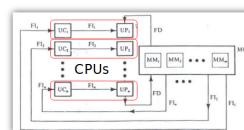
- Descripción estructural



- Descripción funcional

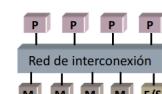


- Arquitecturas MISD, no existen computadores que funcionen según este modelo. Aunque se puede simular en un código este modelo para aplicaciones que procesan una secuencia o flujo de datos.



2. CLASIFICACIÓN POR EL SISTEMA DE MEMORIA (MIMD)

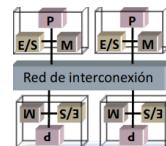
- **Multiprocesadores**: Todos los procesadores comparten el mismo espacio de direcciones y además, el programador NO necesita conocer dónde están almacenados los datos.



Si son de memoria centralizada se llamarán *SMP*, y tendrán las siguientes características:

- Mayor latencia

- Poca escalabilidad
- Comunicación implícita mediante variables compartidas, es decir que los datos no están duplicados en MP.
- **Sincronización:** Necesita implementar primitivas de sincronización.
- **Distribución de datos/código:** No necesita distribución de código y datos entre procesadores.
- **Programación:** La programación es más sencilla generalmente.
- **Multicomputadores:** Cada procesador tiene su espacio de direcciones propio y el programador necesita conocer dónde están almacenados los datos.



Presenta las siguientes características:

- Menor latencia
- Mayor escalabilidad
- Comunicación explícita mediante software para paso de mensajes. Los datos pueden estar duplicados en MP.
- **Sincronización:** mediante software de comunicación.
- **Distribución de datos/código:** Necesita distribución entre procesadores, mediante herramientas de programación sofisticadas.
- **Programación:** La programación es más difícil generalmente.

3. Comunicación uno a uno:

- En un multiprocesador la comunicación uno-uno se hace mediante la memoria compartida, por lo que se debe garantizar que el flujo de instrucciones consumidor del dato lea la variable compartida (A) cuando el productor haya escrito en la variable el dato.
- En un multicomputador se utiliza un *receive bloqueante* es decir que hasta que el nodo destino no reciba los datos del nodo fuente, no continuará con la ejecución.

4. Incremento de escalabilidad en multiprocesadores

Podemos incrementar la escalabilidad en multiprocesadores mediante:

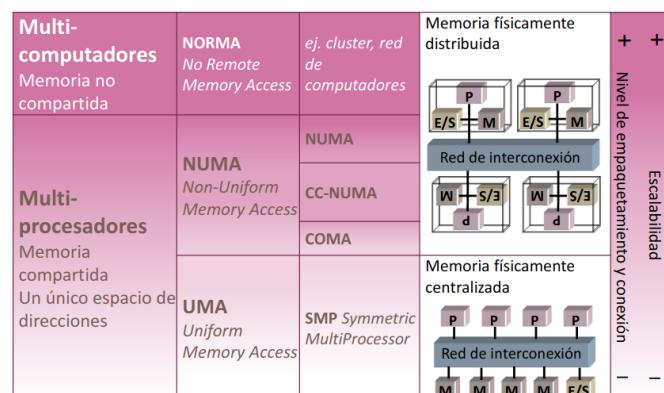
- Aumento de la cache del procesador.
- Uso de redes de menor latencia y mayor ancho de banda que un bus (jerarquía de buses, barras cruzadas, multietapa)
- Distribución física de los módulos de memoria entre los procesadores (pero se sigue compartiendo espacio de direcciones).

5. Clasificación completa

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.



6. CLASIFICACIÓN POR FLUJOS DE INSTRUCCIONES

Clasificamos computadores con múltiples flujos de instrucciones

- **TLP**: ej. múltiples flujos de instr. concurrente-mente o en paralelo
 - **implícito**, con flujos de instrucciones creados y gestionados por la arquitectura.
 - **explícito**, con flujos de instrucciones creados y gestionados por el SO.
 - **1 instancia del SO**, multiprocesadores, multicores, cores multithread.
 - **múltiples instancias del SO**, multicamputadores.

7. CLASIFICACIÓN POR NIVEL DE PARALELISMO APROVECHADO

- **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 VLI-W/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 VLI-W/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.

3 Lección 3: Evaluación de prestaciones

3.1 Medidas usuales para evaluar prestaciones

3.1.1 Tiempo de respuesta

El tiempo *real* de ejecución se compone de:

1. **Tiempo de CPU de usuario:** tiempo en ejecución en espacio de usuario.
2. **Tiempo de CPU de sistema:** tiempo en ejecución en el nivel del kernel del SO.
3. **Tiempo de espera:** tiempo asociado a las esperas debidas a E/S, o asociadas a la ejecución de otros programas.

Llamaremos *elapsed* a la suma de los tres, es decir el tiempo desde que se lanza la ejecución hasta que se termina. Consideramos que el tiempo de CPU es el tiempo de usuario más el del sistema, pero sin tener en cuenta las esperas. Por tanto, el tiempo de respuesta cambia según los flujos de instrucciones:

- Con un flujo de instrucciones, $elapsed \geq T_{CPU}$.
- Con varios flujos de instrucciones, $elapsed < T_{CPU}$ y además, $elapsed \geq \frac{T_{CPU}}{n_{flujoscontrol}}$.

Podemos obtener medidas de tiempo mediante:

| Función | Fuente | Tipo | Resolución aprox. (microsegundos) |
|---|--------------------|------------------------------|-----------------------------------|
| <code>time</code> | SO (/usr/bin/time) | <i>elapsed, user, system</i> | 10000 |
| <code>clock() /CLOCKS_PER_SEC</code> | SO (time.h) | <i>CPU</i> | 10000 |
| <code>gettimeofday()</code> | SO (sys/time.h) | <i>elapsed</i> | 1 |
| <code>clock_gettime() /clock_getres()</code> | SO (time.h) | <i>elapsed</i> | 0.001 |
| <code>omp_get_wtime() /omp_get_wtick()</code> | OpenMP (omp.h) | <i>elapsed</i> | 0.001 |
| <code>SYSTEM_CLOCK()</code> | Fortran | <i>elapsed</i> | 1 |

3.1.2 Tiempo de CPU

Vemos cómo estudiar el tiempo de CPU:

$$T_{CPU} = CiclosDelPrograma \cdot T_{CICLO} = \frac{CiclosDelPrograma}{FrecuenciaDeReloj} = NI * CPI * T_{ciclo}$$

donde

- $NI = num_instrucciones$
- $CPI = (ciclos_por_instrucción) = \frac{1}{IPC} = \frac{1}{Instrucciones_por_ciclo} = \frac{CiclosDelPrograma}{NúmeroDeInstrucciones(NI)}$

Además, puede que tengamos conjuntos de instrucciones que tardan ciclos distintos, es decir, con un CPI distinto, es ese caso:

$$T_{CPU} = \sum_i NI_i * CPI_i * T_{ciclo} = NI * \frac{\sum_i NI_i * CPI_i}{NI} * T_{ciclo}$$

A continuación hacemos algunas observaciones:

Hay procesadores que pueden emitir varias instrucciones al mismo tiempo, en ese caso:

- 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.

Y entonces tendríamos:

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

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

También hay procesadores que pueden codificar varias operaciones en una instrucción:

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

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

donde $NI = Noper/OpInstr$.

3.1.3 Mejoras en las prestaciones

Claramente, el tiempo de CPU depende de las especificaciones de cada arquitectura, en particular, vemos qué mejoras se pueden realizar, y dónde se verá reflejada la mejora:

- Tecnología:

- Disminuye T_{ciclo}
- Aumenta CPI
- Estructura y organización:
 - Disminuye NI
 - ALU y Unidades funcionales SIMD (operaciones con vectores)
 - Disminuye T_{ciclo} :
 - Segmentación
 - Disminuye CPI :
 - Computación superescalar
- Repertorio de instrucciones:
 - Modifica NI
 - Modifica CPI
- Compilador:
 - Modifica NI
 - Modifica CPI

Podemos pensar que si optimizamos al compilar el código (opciones -O) deberíamos obtener un tiempo de ejecución menor. Sin embargo, por ejemplo si compilamos con -O3 no siempre obtenemos un tiempo menor.

3.2 Productividad

Definición 3.1. La **productividad** es el número de entradas procesadas por unidad de tiempo.

La productividad la medimos mediante:

1. **MIPS** (millones de instrucciones por segundo):

$$MIPS = \frac{NI}{T_{CPU} * 10^6} = \frac{NI}{NI * CPI * T_{ciclo} * 10^6} = \frac{F}{CPI * 10^6}$$

- Depende del repertorio de instrucciones, lo que hace difícil la comparación de máquinas con repertorios distintos.
- Puede variar con el programa, no es una medida representativa de la máquina.
- Puede variar inversamente con las prestaciones (mayor valor de MIPS corresponde a peores prestaciones)

2. **MFLOPS** (millones de instrucciones punto flotante por segundo):

$$MFLOPS = \frac{n_{op_comaflotante}}{T_{CPU} * 10^6}$$

- 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.

Un programa puede tener más MIPS pero tener también más tiempo de ejecución, **no son una característica fiable**. Es mucho mejor comprobar los MFLOPS.

3.3 Otras medidas

Definición 3.2. La **funcionalidad** es el tipo de entradas diferentes que es capaz de procesar.

Definición 3.3. La **expansibilidad** es la posibilidad de ampliar la capacidad de procesamiento añadiendo bloques a la arquitectura existente.

Definición 3.4. La **escalabilidad** es la posibilidad de ampliar el sistema sin que esto suponga una devaluación de las prestaciones.

Definición 3.5. La **productividad** es el número de entradas procesadas por unidad de tiempo.

Definición 3.6. La **eficiencia** es el cociente prestaciones/coste.

Y vemos cómo se podrían aplicar estos conceptos a los elementos vistos del computador:

1. Memoria:

- *Entradas:* accesos a memoria.
- *Medida de prestaciones:*
 - Tiempo de respuesta = latencia.
 - Escalabilidad
 - Productividad= ancho de banda.

2. Computador:

- *Entradas:* programas de aplicaciones.

- *Medida de prestaciones:*
 - Tiempo de respuesta
 - Productividad
 - Funcionalidad: se mide mediante programas que procesa eficientemente.

3. Procesador:

- *Entradas:* instrucciones
- *Medida de prestaciones:*
 - Tiempo de respuesta = CPI, T_{ciclo} , CPI media
 - Productividad= mismas medidas que en tiempo de respuesta
 - Escalabilidad
 - Funcionalidad: se mide mediante instrucciones que procesa.

3.4 Ganancia en prestaciones al realizar una mejora

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 .

$$S = \frac{V_p}{V_b} = \frac{T_b}{T_p} = \frac{T_{CPU}^b}{T_{CPU}^p} = \frac{NI^b * CPI^b * T_{ciclo}^b}{NI^p * CPI^p * T_{ciclo}^p}$$

donde

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

3.5 Mejora por segmentación

En la mejora de un núcleo de procesamiento a través de la segmentación lo que cambia realmente es el tiempo de ciclo. Por ejemplo, si conseguimos que todas las instrucciones tarden lo mismo y tenemos un cauce de 5 etapas, entonces $T_{ciclo}^p = \frac{T_{ciclo}^b}{5}$.

En los ejemplos de las diapositivas vemos que además, podemos cambiar también el CPI pues se pueden emitir varias instrucciones en un ciclo.

Por último, si usamos unidades funcionales SIMD, vectoriales, que procesan vectores de p componentes, tendremos que $NI_p = \frac{NI_b}{p}$.

Si mezclamos todas las mejoras anteriores, tendremos $S = long_vectores * emisiones_ciclo * etapas_cauce$.

3.5.1 Ganancia pico

La ganancia pico es la ganancia ideal que se alcanzaría si no se tuviera un retraso por eventos como

- Riesgos, pueden ser
 - Datos
 - Control
 - Estructurales
- Accesos a memoria, debido a la jerarquía de memoria.

3.5.2 Ley de Amdahl

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

Así, tendremos:

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

y si $p \rightarrow \infty = \frac{1}{f}$ y $f \rightarrow 0 = p$, donde f es la fracción de tiempo de ejecución de la máquina sin la mejora durante el cual no se usa la mejora.

3.6 Conjunto de programas *Benchmark*

Son un conjunto de programas que se utilizan para medir las prestaciones de una arquitectura.

Las propiedades que se exige a las medidas de prestaciones son:

- Fiabilidad: Deben ser representativas, evaluar diferentes componentes del sistema y ser reproducibles.
- Permitir comparar diferentes realizaciones de un sistema o diferentes sistemas: deben ser aceptadas por todos los interesados (usuarios, fabricantes, vendedores).

Y los interesados en los programas benchmark son:

- Vendedores y fabricantes de hardware o software.
- Investigadores de hardware o software.
- Compradores de hardware o software.

3.7 Tipos de benchmarks

1. De bajo nivel o microbenchmark: test ping-pong, evaluación de las operaciones con enteros o con flotantes.
2. Kernels: resolución de sistemas de ecuaciones, multiplicación de matrices, FFT, descomposición LU.
3. Sintéticos: dhystone, Whetstone.
4. Programas reales, como SPEC CPU2017: enteros (gcc, perlbench).
5. Aplicaciones diseñadas: Predicción de tiempo, dinámica de fluidos, animación etc. (p. ej. SPEC2017).

II | Tema 2: Programación paralela

1 Lección 4: Herramientas, estilos y estructuras

1.1 Problemas que plantea la programación paralela al programador

Surgen nuevos problemas como por ejemplo:

- La división en unidades de cómputo independientes (=tareas).
- Agrupación y asignación de las tareas(=carga de trabajo) en procesos o threads, es decir la agrupación en flujos de instrucción.
- Asignación a procesadores o núcleos (hardware).
- Sincronización y comunicación (variables compartidas en el código).

Estos problemas los aborda la herramienta de programación o el SO.

¿Cómo se puede parallelizar?

El punto de partida es:

- Código secuencial, intentamos analizar qué parte tarda más para poder mejorarlo.
- Descripción o definición de la aplicación.
- Definir un programa paralelo parecido.
- Realizamos versiones paralelas u optimizadas de bibliotecas de funciones (*Blas* = Basic linear algebra subroutine o *LA PACK* = Linear Algebra Package).

Además, hay distintos **modos de programación MIMD**:

- SPMD (Single-Program Multiple Data): el paralelismo reside en que cada copia de mi programa ejecuta con datos distintos, que son los flujos que se asignan cada procesador. Imita al SIMD, es decir utiliza paralelismo de datos o de bucle.
- MPMD (Multiple program multiple data): Cada procesador ejecuta programas distintos, utiliza paralelismo de tareas o funciones.
- Modos combinados, mezcla entre ambas opciones.

1.2 Herramientas para la programación paralela

Para empezar se dispone, a medida que se va disminuyendo el nivel de abstracción:

1. *Compiladores paralelos*: extraen la paralelización de forma automática.
2. *Lenguajes paralelos*: tienen construcciones particulares y bibliotecas de funciones que requieren un compilador exclusivo. Las API tienen un conjunto de directivas y funciones que se añaden a un compilador de un lenguaje secuencial. El paralelismo implícito es localizado por el programador.

- *API funciones + directivas (OPENMP)*, se puede parallelizar de dos maneras:
 - Construcciones del lenguaje específicas+ funciones.
 - Lenguaje secuencial + directivas+ funciones
- 3. *API y funciones (MPI)*, que utilizan lenguaje secuencial y funciones. Las API de funciones consisten en una biblioteca de funciones que se añaden a un compilador de un lenguaje secuencial habitual. La asignación de tareas la realiza el programador.

Entonces, en resumen, las herramientas permiten de forma implícita o explícita:

- Localizar paralelismo y descomponer en tareas independientes (**decomposition**).
- Asignar las tareas (= carga de trabajo, incluye código y datos) a procesos y threads(**scheduling**).
- Crear y terminar procesos y threads (**openmp**).
- Comunicar y sincronizar procesos y threads (el programador indica dónde).
- Asignar procesos a Unidad de Procesamiento (**mapping**), por parte del programador, la herramienta de programación o el SO, ver la lista de arriba.

En OpenMP, junto con otras herramientas, también disponemos de directivas o funciones colectivas, que facilitan el trabajo del programador.

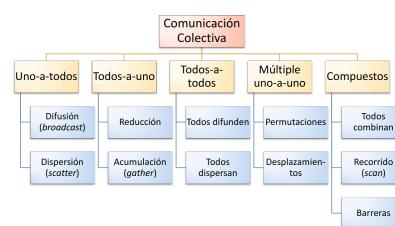
Bibliotecas de funciones

Cuando se trabaja con bibliotecas de funciones el cuerpo de los procesos se escribe en lenguaje secuencial y para la creación y gestión de los procesos, comunicación y sincronización se usan funciones de la biblioteca.

En este tema consideramos que la carga de trabajo, que incluye código y datos, es equivalente a las tareas.

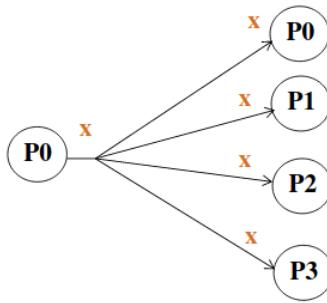
1.2.1 Comunicaciones colectivas

Las herramientas de programación paralela, además de comunicación entre dos procesos, pueden ofrecer comunicaciones en las que intervienen varios procesos. Estos esquemas pueden incrementar la eficiencia del programa paralelo.

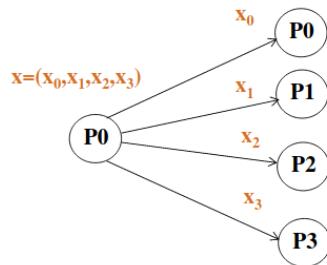


Comunicación uno-a-todos

- Difusión (broadcast):



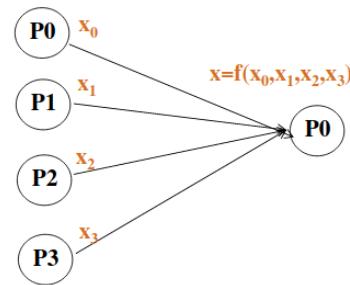
- Dispersión(scatter):



En el primero, a cada P le llega la misma información, y en el segundo, se divide la información y a cada P le llega una porción distinta.

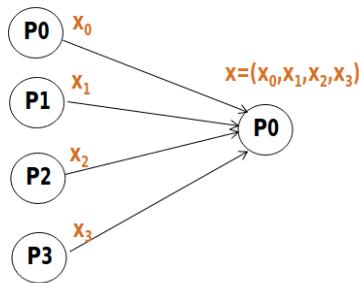
Comunicación todos-a-uno

- Reducción:



En reducción, se junta la información obtenida de cada procesador según una función predefinida f , obteniendo un único valor, la función f es conmutativa y asociativa.

- Acumulación(gather):



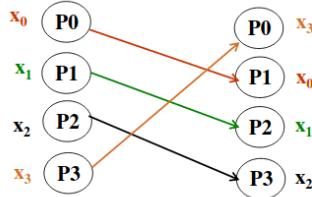
Los mensajes se reciben de forma concatenada en el receptor de una estructura vectorial.

Comunicación múltiple uno-a-uno

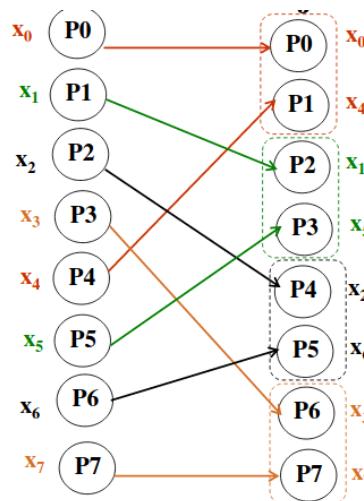
Si todos los componentes del grupo envían y reciben se llama permutación, si no se llama **desplazamiento**.

- Permutación rotación:

Permutación rotación:

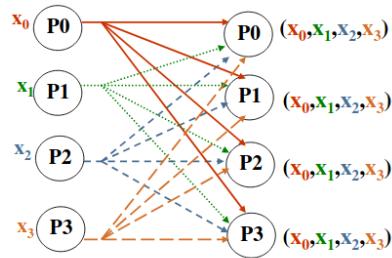


- Permutación baraje-2:



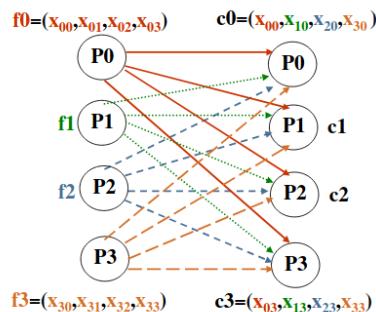
Comunicación todos-a-todos

- Todos difunden(**all-broadcast o gossiping**):



En all-broadcast cada procesador difunde a todos la misma información (difusión). Las n transferencias recibidas por un proceso se concatenan en función del identificador del proceso que las envía, de forma que todos los procesos reciben el mismo.

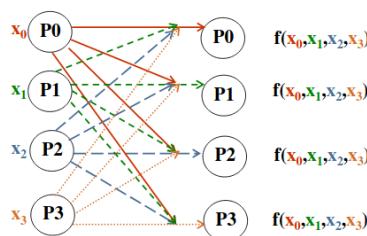
- Todos dispersan (**all-scatter**):



Los procesos concaténan diferentes transferencias.

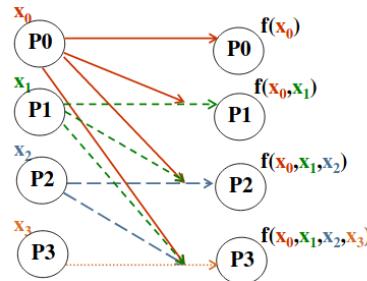
Servicios compuestos

- Todos combinan:

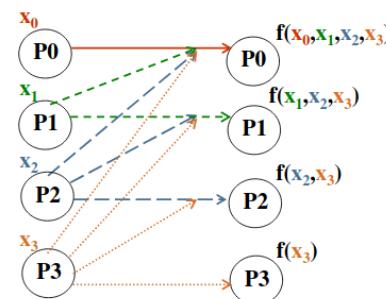


Cada procesador tiene una función predefinida que combina la información obtenida de los demás.

- Recorrido prefijo paralelo:



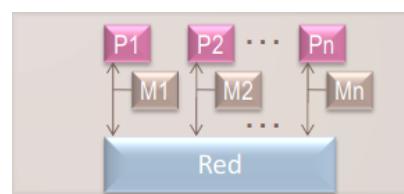
- Recorrido sufijo paralelo:



Las funciones colectivas son implementadas por la Interfaz de Paso de Mensajes (MPI).

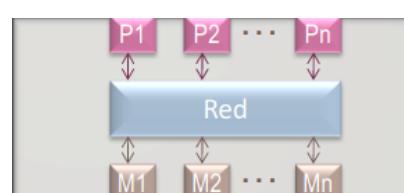
1.3 Estilos de programación paralela

1. **Paso de mensajes** (message passing): en multicomputadores, donde la memoria no se comparte. Ofrece herramientas para copiar datos de un espacio de memoria a otros.



Utiliza:

- Lenguajes de programación: Ada, Occam
 - API (bibliotecas de funciones): MPI, PVM
2. **Variables compartidas** (shared memory) en multiprocesadores, pues todos los procesadores tienen acceso a las zonas de memoria compartidas.

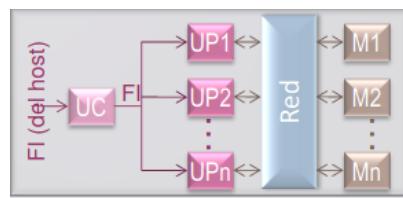


Utiliza:

- Lenguajes de programación: Java, Ada
- API(directivas+funciones): OPENMP
- API (bibliotecas de funciones): PosIx threads

Hay que tener en cuenta que la herramienta tiene que ofrecer herramientas de sincronización, pues las funciones comparten el mismo espacio de direcciones. Para ello se utilizan primitivas que ofrece el software que se amparan en instrucciones máquina para incrementar prestaciones.

3. **Paralelismo de datos** (data parallelism): en procesadores matriciales o en GPU, donde se pasa primero por la unidad de control. Solo soporta paralelismo a nivel de bucle. La sincronización está implícita. Dispone de construcciones para la distribución de datos entre los elementos de procesamiento.



Utiliza:

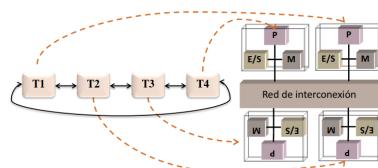
- Lenguajes de programación + funciones stream processing: Fortran, HPF,
- API(directivas+funciones): OpenACC (incluida en OpenMP), Nvidia CUDA

1.4 Estructuras típicas de códigos paralelos

Hay 5 estructuras típicas a la hora de parallelizar un código:

1. **Descomposición de dominio o descomposición de datos:** El trabajo a realizar por cada proceso se determina dividiendo las estructuras de datos de entrada y/o salida en tantos trozos como flujos.

Las tareas realizan tareas similares (útil en algoritmos con imágenes).

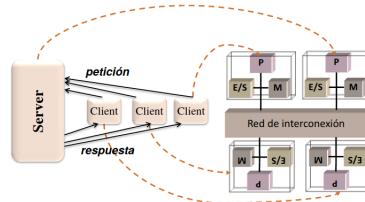


Por ejemplo, si tenemos 4 procesadores y 4 flujos de instrucciones, existen varias alternativas:

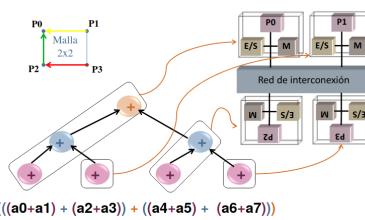
- Si dividimos la estructura de entrada, se asigna un Fl_i a todas las instrucciones que utilicen los datos del bloque i .

- Si se divide la estructura de salida, se asigna un FI_i a todas las instrucciones que hay que realizar para llegar a los datos del bloque i .

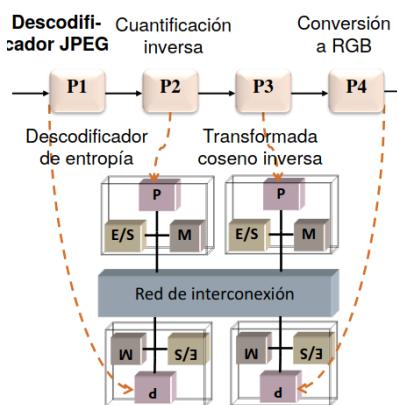
2. **Cliente/servidor:** El servidor se ejecuta en un procesador y los clientes en los demás. Cada cliente lanza una petición al servidor y al recibir la respuesta ejecuta el programa.



3. **Divide y vencerás:** cada iteración de la recursividad se realiza en un procesador.



4. **Segmentación o flujo de datos:** Se divide el cauce en etapas o segmentos, donde cada etapa se ejecuta en un procesador (en OMP, sections). Se utiliza cuando se aplica a un flujo de datos de entrada las mismas funciones en secuencia, haciendo que cada proceso ejecute distinto código.



La velocidad del trabajo depende de la función más lenta.

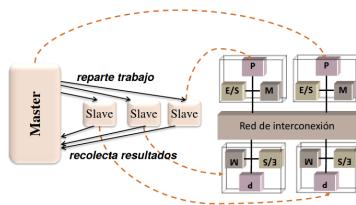
5. **Master-slave, granja de tareas:** Se distingue un proceso(master) que se encarga de mandar tareas a otros procesos(esclavos), que luego devuelven resultados de vuelta al master.

El master se ejecuta en un procesador, y es quien reparte trabajos entre los esclavos, que se ejecutan en los demás procesadores (1 procesador=1 esclavo). No hay comunicación entre esclavos.

A diferencia de en el de cliente, no se realizan peticiones, sino que es el master quien tiene todo el "poder".

- Se escribe un único trabajo, y cada esclavo ejecuta con datos distintos.

- El master controla la ociosidad de todos los procesadores, puede llegar a auto asignarse un trabajo.
- La asignación se puede hacer de forma estática o dinámica. Si es estática, se conoce la tarea de cada esclavo.



1.5 Buena asignación de tareas

1. Decidimos cuántos flujos de instrucciones diferentes queremos (**mapeo**). Siendo el número de FI (flujo de instrucciones) el número máximo de tareas que en un momento dado se podrían ejecutar en paralelo, es decir el **grado de paralelismo de la aplicación**.
2. Decidimos qué operaciones asignamos a cada FI (**planificación**). Teniendo en cuenta las herramientas de sincronización y comunicación, por ejemplo, si queremos una comunicación por red, los procesadores cuyos FI necesiten datos entre ellos estarán cercanos.

2 Lección 5: Proceso de paralelización

El proceso de paralelización tiene 4 pasos:

1. **Descomposición** en tareas independientes.
2. **Asignación** de tareas a procesos y threads.
3. **Redacción** de código paralelo.
4. **Evaluación** de prestaciones.

Nota. : En este tema usamos indistintamente flujo de instrucciones y proceso o núcleo. Un flujo de instrucciones realmente es un proceso o thread del sistema operativo.

Además, la carga de trabajo es el conjunto de tareas asignadas, incluyen código y datos.

2.1 Descomposición en tareas independientes.

Es importante analizar la dependencia entre funciones y entre iteraciones de bucles.

A partir de un código secuencial nos podemos situar en dos niveles de abstracción para extraer paralelismo:

- Nivel de función: Encontramos las funciones independientes y ejecutables en paralelo (paralelismo de tareas).
- Nivel de bucle/bloque: Analizamos las iteraciones de los bucles de una función, encontrando las independientes (parallelismo de datos).

2.2 Asignación de tareas a procesos y threads

Esta etapa incluye:

- **Scheduling** o agrupación de tareas en procesos.
- **Mapping** o el mapeo a los procesadores.

2.2.1 Granularidad

Definición 2.1. Definiremos la **granularidad** como el tamaño de los trozos de código que ejecutan los flujos de instrucciones entre comunicaciones, es decir, que se pueden ejecutar en paralelo.

La granularidad depende de:

- El número de cores o procesadores (elementos de procesamiento).
- El tiempo de comunicación y sincronización frente al tiempo de cálculo.

Además, conviene seguir ciertas indicaciones:

- No asignar más de un proceso o hebra a cada procesador.
- Se asignan iteraciones de bucle a hebras y funciones a procesos.

2.2.2 Equilibrado de la carga

Además, a la hora de asignar las tareas hay que controlar el **equilibrado de la carga**, donde lo conveniente es intentar disminuir al máximo el tiempo que los flujos de instrucciones (procesos o threads del SO), puedan permanecer ociosos. La situación ideal es que todos empiecen y acaben a la vez.

¿De qué depende el equilibrado?

1. **La arquitectura**, que puede ser:

- Heterogénea u homogénea: influye en el **tiempo de cálculo** de un programa paralelo.
 - Homogénea: los componentes de los nodos de cómputo son iguales, es decir que todos tardan lo mismo.
 - Heterogénea: cada componente puede tardar un tiempo, son las arquitecturas más comunes (procesadores de propósito general+GPU+FPGA).

- Uniforme o no uniforme: influye en el **tiempo de comunicación** de un programa paralelo.
 - Uniforme: todos los nodos de cómputo tardan el mismo tiempo en comunicarse.
 - No uniforme: la mayoría de los procesadores (incluso un multiprocesador a nivel de placa).

2. La aplicación y descomposición en tareas independientes:

Hay casos en los que el equilibrado se complica, por ejemplo hay aplicaciones en las que las tareas que se pueden ejecutar en paralelo entre puntos de sincronización del código no tardan el mismo tiempo, o aplicaciones en las que no se sabe las tareas que se pueden ejecutar en paralelo ni antes ni después de la ejecución.

Alguna solución para facilitar el equilibrado es:

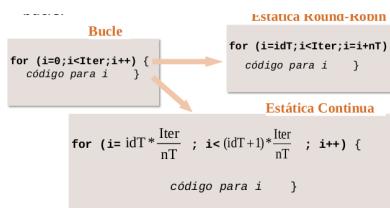
- En la aplicación todas las tareas que se pueden ejecutar en paralelo entre puntos de sincronización tardan lo mismo.
- La arquitectura es homogénea y uniforme.
- Si lo que se complica es el equilibrado estático, se puede usar una asignación en tiempo de ejecución

2.2.3 Tipos de asignación

La asignación puede ser:

- **Estática**, donde la asignación se realiza en tiempo de compilación o al escribir el programa:

- La asignación no cambia entre ejecuciones.
- La implementación es sencilla, no supone añadir mucho código extra.



- **Dinámica**, en este caso la asignación se va realizando durante la ejecución. Se va asignando trabajo a los flujos conforme terminan la tarea anterior.

- Facilita el equilibrado en arquitecturas heterogéneas y en las que las tareas suponen distinto tiempo.
 - Es la única implementación si no se conoce en ningún momento las tareas de la aplicación.
- La asignación puede cambiar entre ejecuciones.
- Su implementación requiere añadir código que penaliza el tiempo de ejecución (añade sobrecarga).

Dinámica

```
lock(k);
    n=i;  i=i+1;
unlock(k);
while (n<Iter) {
    código para n ;
    lock(k);
    n=i;  i=i+1;
    unlock(k);
}
```

Ambas asignaciones pueden ser explícitas, donde la realiza el programador, o implícitas, donde la realiza la herramienta de programación al generar el ejecutable.

2.2.4 Mapeo de procesos a unidades de procesamiento

El mapeo normalmente se deja al SO, aunque lo puede realizar el entorno o sistema en tiempo de ejecución, e incluso puede ser influido por el programador.

2.3 Redactar código paralelo

El código va a depender del estilo de programación que se utilice (Lección 4.3), del modo de programación (Lección 4.1.2), del punto de partida (Lección 4.1.1), de la herramienta que usemos (Lección 4.2) y de la estructura (Lección 4.4).

2.3.1 Evaluación de prestaciones

Se encarga de comprobar si se han obtenido las prestaciones mínimas.

3 Lección 6: Evaluación de prestaciones

En programación paralela se usa, para evaluar las prestaciones de un código paralelo:

- **Tiempo de respuesta:**
 - Real (wall-clock y elapsed time).
 - Usuario, sistema y CPU.
- Productividad

- **Escalabilidad**

- **Eficiencia**, donde se tiene en cuenta:

- La relación en prestaciones/prestaciones máximas.
- El **rendimiento**=prestaciones/nº de recursos
- Otras medidas, como prestaciones/consumo_potencia o prestaciones/área_ocupada.

3.1 Escalabilidad

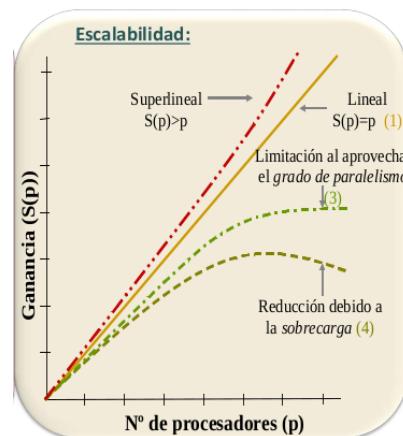
La escalabilidad define en qué medida aumentan las prestaciones de un código paralelo al añadir procesadores.

El incremento en prestaciones que se obtiene al usar p procesadores se llama **speed-up**, y es: $S(p) = \frac{T_{secuencial}}{T_{paralelo}}$
El tiempo de ejecución en paralelo ($T_{paralelo} = TC + TO$) depende de:

- **Tiempo de cálculo paralelo (TC)**, que es el tiempo que supone la ejecución en paralelo de las tareas del código secuencial.
- **Tiempo de sobrecarga (TO)** introducido por la paralelización. A su vez, este depende de:
 - *Tiempo de creación/destrucción* de los flujos de instrucciones (depende de p).
 - *Tiempo de comunicación y sincronización*, puede depender de p .
 - *Tiempo de cálculos y funciones no presentes en el código secuencial*.
 - *Falta de equilibrado*.

Además, tanto TC como TO pueden depender del tamaño del problema que se está resolviendo.

GANANCIA MÁXIMA



Compararemos en qué medida aumentan las prestaciones a medida que aumentamos el número de procesadores.

| | | | | Diagrama | Modelo código | Facción no paralelizable en T_s | Grado de 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 | | | | | | $S(p) = \frac{1}{f + (1-f)} \xrightarrow{p \rightarrow \infty} \frac{1}{f}$ (2) |
| (c) | f | n | 0 | | | | | | $S(p) = \frac{1}{f + (1-f)} \xrightarrow{p=n} \frac{1}{f + (1-f)} = \frac{1}{n}$ (3) |
| (d) | f | ilimitado | Incrementa linealmente con p | | | | | | $S(p) = \frac{1}{f + (1-f) + \frac{T_s}{T_s}} \xrightarrow{p \rightarrow \infty} 0$ (4) |

- **Caso ideal (a):** esperamos obtener un tiempo de ejecución $\frac{TS}{p}$ con p procesadores para un tiempo de ejecución secuencial TS .

Es decir, se reparte el código entre los procesadores disponibles, independientemente de p (paralelismo ilimitado). En este reparto, se asigna a cada procesador la misma cantidad de trabajo.

- La escalabilidad es una línea recta.

Además, hay otros casos a considerar, en primer lugar, la ganancia puede dejar de crecer a partir de un número de procesadores, debido a que puede llegar un momento en el que no se podrá repartir el trabajo entre más procesadores. (3)

O incluso la ganancia puede decrecer debido a que la sobrecarga se puede incrementar con el número de procesadores (4).

- **Caso b:** no todo el código se puede paralelizar (fracción f), y por tanto, la escalabilidad será:

$$S(p) = \frac{TS}{TS * f + \frac{TS(1-f)}{p}} = \frac{1}{f + \frac{1-f}{p}}$$

Entonces la ganancia en prestación máxima va a estar limitada por la fracción del código no paralelizable. En esta porción de código, el tiempo ($f * TS$), permanecerá constante.

Por tanto, el tiempo de ejecución paralelo nunca podrá ser inferior al tiempo que supone la ejecución de la parte no paralelizable y, por tanto, la ganancia en prestaciones nunca podrá ser superior a $1/f$ ($= TS/f \times TS$).

3.2 Ley de Amdahl

La **ley de Amdahl** dice que la ganancia en prestaciones que se puede conseguir aplicando paralelismo a un código secuencial está limitada por la fracción no paralelizable del mismo.

$$S(p) = \frac{T_s}{T_p(p)} \leq \frac{T_s}{f * T_s + \frac{(1-f) * T_s}{p}} = \frac{p}{1 + f(p-1)} \xrightarrow{p \rightarrow \infty} \frac{1}{f}$$

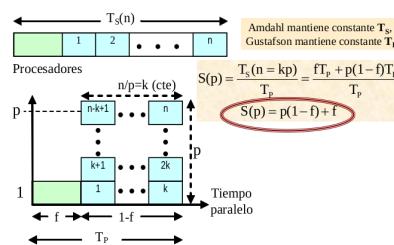
➤ S : Incremento en velocidad que se consigue al aplicar una mejora. (paralelismo)
 ➤ p : Incremento en velocidad máximo que se puede conseguir si se aplica la mejora todo el tiempo. (número de procesadores)
 ➤ f : fracción de tiempo en el que no se puede aplicar la mejora. (fracción de t. no paralelizable)

Algo importante a destacar es que en la práctica, la fracción de código que limita la escalabilidad se puede disminuir aumentando el tamaño del problema.

Además, la ganancia disminuye conforme aumenta la fracción de código no paralelizable.

3.3 Ganancia escalable

Si el objetivo es mejorar la calidad de los resultados aumentando el tamaño del problema, se puede obtener una ganancia que no está limitada y que depende de p .



En la gráfica se representa el tiempo paralelo TP en el eje x y el número de procesadores que se usa para diferentes trozos del código paralelo en el eje y. En este caso, en $f \times TP$ se usa 1 procesador (código no paralelizable) y, en $(1-f) \times TS$, p procesadores.

Cada procesador ejecuta $k=n/p$ tareas independientemente del valor de p ; es decir, conforme se incrementa p se incrementa también el tamaño del problema para mantener constante la carga de trabajo asignada a cada uno de los procesadores.

3.3.1 Escalabilidad débil o Ley de Gustafson

Definición 3.1. Estudiar la **escalabilidad débil** (ganancia escalable) de un código paralelo supone estudiar cómo evoluciona la ganancia o el tiempo conforme se incrementa p *manteniendo constante el tamaño del problema* que resuelve cada procesador, es decir, se mantiene el tamaño n .

Se pretende ver si el tiempo de ejecución paralelo se mantiene constante:

- Si aumenta, será menos escalable (escalabilidad débil) el código paralelo cuanto mayor sea la pendiente con la que aumenta el tiempo.

$$S(p) = \frac{T_s(n=kp)}{T_p} = \frac{f \cdot T_p + p(1-f)T_p}{T_p}$$

Definición 3.2. En el caso de un estudio de **escalabilidad fuerte** (Ley de Amdahl) se representa cómo evoluciona la ganancia (a veces se usa el tiempo paralelo) conforme se incrementa p manteniendo constante el tamaño del problema que se resuelve (TS). Cuanto más cercana se mantenga la ganancia a la ganancia lineal mejor código paralelo será.

En resumen, en la escalabilidad débil se mantiene constante T_p , obteniendo T_s a partir de T_p , mientras que en la escalabilidad fuerte mantenemos constante T_s , y obtenemos T_p a partir de T_s .

Nota. El significado y el valor de f en la ley de Amdahl es distinto del f que se utiliza en la ley de Gustafson.

EFICIENCIA: Para evaluar en qué medida las prestaciones que ofrece un sistema para un código paralelo se acercan a las prestaciones máximas que idealmente debería ofrecer dado los procesadores disponibles en el mismo, se usa la siguiente expresión de eficiencia:

$$E(p) = \frac{\text{Prestaciones}(p)}{p * \text{Prestaciones}(1)} = \frac{S(p)}{p}$$

4 Ejercicios resueltos

4.1 Ejercicio 2

Un programa tarda 20 s en ejecutarse en un procesador P1, y requiere 30 s en otro procesador P2. Si se dispone de los dos procesadores para la ejecución del programa (despreciamos sobrecarga):

1. ¿Qué tiempo tarda en ejecutarse el programa si la carga de trabajo se distribuye por igual entre los procesadores P1 y P2?
2. Qué distribución de carga entre los dos procesadores P1 y P2 permite el menor tiempo de ejecución utilizando los dos procesadores en paralelo? ¿Cuál es este tiempo?

4.1.1 Solución

1. Si la carga de trabajo es la mitad, tendremos $T_p(f, 1-f) = T_p(\frac{1}{2}, \frac{1}{2})$, donde f es la proporción del programa asignado a P_1 . Entonces si a P_1 le asignamos la mitad del trabajo, tardará la mitad: $T_1 = 10s$ y $T_2 = 15s$. Como se ejecuta en paralelo, el tiempo es el más lento de los dos, en este caso, $T_p(\frac{1}{2}, \frac{1}{2}) = 15s$.
2. Tenemos que equilibrar para que los procesadores no estén ociosos, es decir que empiecen y acaben a la vez, entonces: $T^{p_1}(f) = T^{p_2}(f)$, lo que conlleva: $20f = 30(1-f)$, es decir $f = \frac{3}{5}$ y entonces con esta repartición tendríamos:

$$T_p(\frac{3}{5}, \frac{2}{5}) = 12s$$

4.2 Ejercicio 4

Un 25% de un programa no se puede paralelizar, el resto se puede distribuir por igual entre cualquier número de procesadores. ¿Cuál es el máximo valor de ganancia de velocidad que se

podría conseguir al paralelizarlo en pprocesadores, y con infinitos? ¿A partir de cuál número de procesadores se podrían conseguir ganancias mayores o iguales que 2?

4.2.1 Solución

Utilizamos la Ley de Ahmdal:

$$S(p) = \frac{T_s}{T_p(p)} = \frac{T_s}{0.25T_s + \frac{0.75T_s}{p}} = \frac{1}{0.25 + \frac{0.75}{p}}$$

y si p tiende a infinito, $S(p) = \frac{1}{0.25} = 4$

Si queremos asegurarnos que $S(p) > 2$, hacemos $S(p) = 2$:

$$\frac{1}{0.25 + \frac{0.75}{p}} = 2$$

entonces despejando $p = \frac{3/2}{1/2} = 3$ procesadores.

III | Tema 3: Arquitecturas con paralelismo a nivel de thread

1 Lección 7: Arquitecturas TLP (Thread-Level-Paralelism):

1.1 Clasificación de arquitecturas con TLP explícito y una instancia de SO

Este tipo de arquitecturas se clasifican en:

- **Multiprocesador**, ejecutan varios thread en paralelo en un computador con varios cores/procesadores, asignando cada thread a un core distinto.
 - Incluyen distintos niveles de empaquetamiento (dato, encapsulado, placa, chasis, etc.)
 - Incluyen, por tanto los multicores en un chip (CMP-chip multiprocessor).
- **Multicore o multiprocesador en un chip(CMP)**, ejecutan varios threads en paralelo en un chip de procesamiento multicore.
- **Core multithread**, core que modifica su arquitectura ILP para ejecutar threads concurrentemente o en paralelo.

1.2 MULTIPROCESADORES

1.2.1 Clasificación

Como ya vimos en el Tema 1, se pueden clasificar en:

Según el sistema de memoria

- Multiprocesador con memoria centralizada (UMA), donde todos los procesadores comparten el espacio de direcciones de memoria. Y las transmisiones se realizan por interconexión.
 - Tiene mayor latencia y es poco escalable.
- Multiprocesador con memoria distribuida (NUMA), cada procesador tiene su propio espacio de memoria.
 - La latencia es menor y es más escalable, aunque se necesita distribución de datos y código.

Según el nivel de empaquetamiento (de pequeño a mayor)

- Chip
- Placa/board

- Armario/cabinet
- Sistema

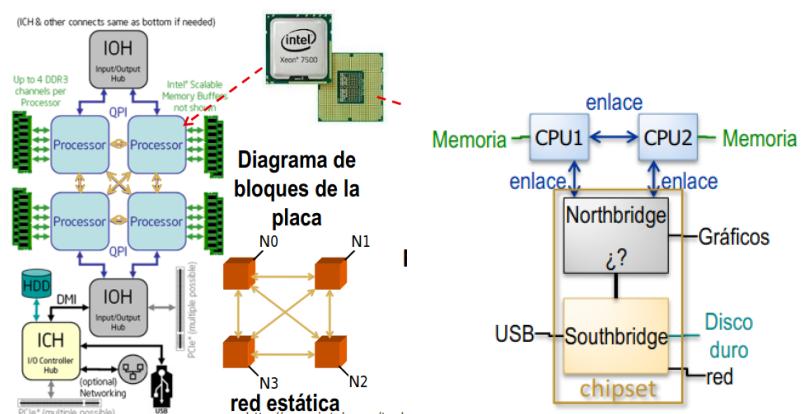
1.2.2 Evolución UMA-NUMA en una placa

■ UMA:

- El controlador de memoria (quién gestiona todas las operaciones, el cerebro), se encuentra en un **chipset**.
- Conexión y comunicación (red): mediante bus compartido.
- En un ciclo sólo hay un acceso a memoria, por tanto sólo será posible la paralelización si se accede a módulos distintos.

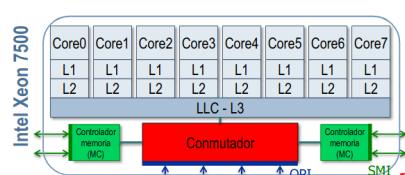
■ NUMA:

- El controlador de memoria se encuentra en un **chip del procesador**
- La comunicación (Red) es por medio de
 - Enlaces: conexiones punto a punto que permiten que un procesador pueda acceder a los módulos de memoria que no estén en su mismo nodo.
 - Conmutadores: en el chip del procesador
- Cada procesador puede acceder tanto a su módulo de memoria como al de los demás



1.3 MULTICORES

Ejecutan threads en paralelo en un **chip de procesamiento multicore**. Vemos qué es (ejemplo de estructura):



En un multicore se combinan varios microprocesadores independientes. Para acceder a otros chips de memoria desde uno de los cores, se utiliza el conmutador, y de estar conectados, se utiliza el controlador.

Realmente son multiprocesadores implementados en un dado de silicio. Normalmente tienen una caché de último nivel que comparten todos los núcleos de procesamiento, y luego cada uno tiene su propia caché (niveles restantes).

En la imagen, por ejemplo, tenemos 3 niveles de caché, dos de ellos independientes y el tercero compartido. En este caso se podría usar L3 para comunicación y sincronización. Es decir, el paso entre L2 y L3 se realizaría como en un multiprocesador UMA. Este esquema es, de todas las alternativas vistas en clase, la que supone una mejora en las prestaciones, pues un core no tiene que acceder a memoria para que otro core (que no comparta memoria) escriba un dato ya que todos los cores comparten L3.

1.4 CORES MULTITHREAD

En este tipo de cores se modifica la arquitectura original ILP para poder dar soporte a TLP. Estos cores buscan minimizar el tiempo donde se encuentran ociosos durante la etapa del cauce (por motivos de dependencia p.e.). Esta ociosidad se suele resolver añadiendo threads. Observamos que **coinciden**: n^o de threads a añadir con el n^o m醙imo de instrucciones `nop` +1.

Antes de nada, realizamos un repaso sobre las arquitecturas ILP.

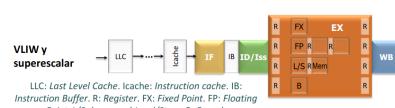
Arquitecturas ILP

Hay dos tipos de arquitecturas ILP:

- **Procesadores segmentados**, ejecutan instrucciones concurrentemente haciendo uso de la segmentación de sus componentes, es decir, dividir el flujo de instrucciones en etapas.



- **Procesadores VLIW y superescalares**, ejecutan instrucciones concurrentemente usando segmentación, y en paralelo. Es decir, tienen múltiples unidades funcionales y emiten múltiples instrucciones en paralelo a unidades funcionales. Hay dos tipos:
 - VLIW: las instrucciones que se ejecutan en paralelo se captan juntas de memoria, formando una palabra de instrucción más larga.
 - El hardware presupone que las instrucciones de una palabra son independientes (no filtra).
 - Superescalares:
 - Tienen que encontrar instrucciones que se puedan emitir y ejecutar en paralelo.
 - El hardware extrae el paralelismo a nivel de instrucción.



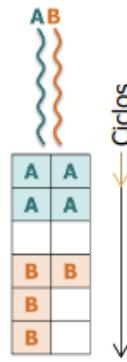
Modificación ILP a Core Multithread

Hay dos aspectos a modificar:

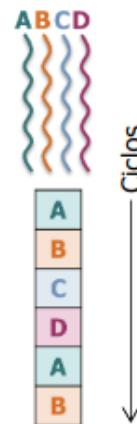
- Almacenamiento: se multiplexa, reparte o comparte entre threads, o se replica.
 - En SMT: repartir, compartir o replicar
- Hardware entre etapas: se multiplexa, reparte o comparte entre threads.
 - Crea un incremento en las prestaciones, aunque lo que se replica siempre es el fichero de registros de la arquitectura.
 - En SMT:
 - Las unidades funcionales (etapa EX) compartidas
 - Resto de etapas repartidas o compartidas
 - No se pueden multiplexar, debido a que en un SMT se pueden emitir a la vez instrucciones de distintos threads en paralelo. Si se multiplexara se ejecutarían concurrentemente.
 - Por ejemplo, se usa en predicción de saltos y decodificación.

Clasificación de Cores Multithread

1. **Temporal Multithreading (TMT):** Ejecutan varios threads concurrentemente en el mismo core.



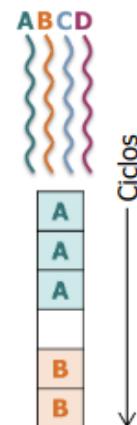
- La conmutación entre threads la decide y controla el hardware.
- En un ciclo sólo emite instrucciones de un único thread.
- **Clasificación:**
 - a) **Fine-grain multithreading (FGMT)** la conmutación entre threads la decide el hardware en **cada ciclo**, mediante implementaciones sencillas.



Puede ser:

- Por turno rotatorio.
- Eventos de cierta latencia junto con alguna técnica de planificación.
 - Eventos: dependencia, operación con latencia,...

b) Coarse-grain multithreading (CGMT), la conmutación entre threads la decide el hardware, con un coste 0 a varios ciclos:



- Tras intervalos de tiempo t prefijados (*timeslice multithreading*).
- Eventos de cierta latencia (*switch-on-event multithreading*).

Clasificación:

a) Estática:

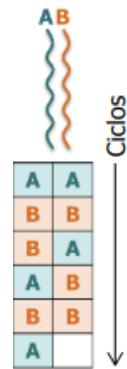
- Comutación, puede ser **explícita** (instrucciones añadidas al repertorio explícitas para conmutación), o **implícita** (instrucciones de carga, almacenamiento, salto).
- Ventaja: Coste de cambio de contexto mínimo (0-1 ciclo)
- Inconveniente: Cambios de contexto innecesarios.

b) Dinámica:

- Comutación según eventos como fallo en la última cache dentro del chip de procesamiento, interrupción (comutación por señal), ...
- Ventaja: Reduce cambios de contexto innecesarios.

- Inconveniente: Mayor sobrecarga al cambiar de contexto, pues añade hardware para realizarlo.

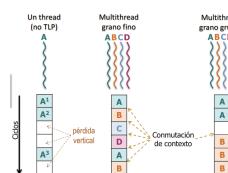
2. **Simultaneous MultiThreading (SMT):** Ejecutan en un core superescalar varios threads en paralelo.



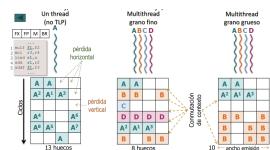
- Pueden emitir instrucciones de varios threads en un ciclo.

Alternativas

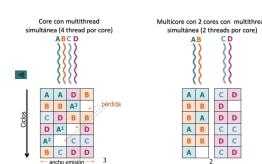
En un core escalar segmentado, solamente se emite una instrucción en cada ciclo de reloj, pero se puede implementar FGMT o CGMT:



En un core con emisión múltiple de instrucciones de un thread, es decir superescalar o VLIW. En este tipo de core se emiten más de una instrucción por cada ciclo de reloj, pero siempre del mismo thread, vemos las alternativas:



En un core multithread, es decir, un multicore o un core superescalar con SMT (Simultaneous MultiThread), se pueden emitir instrucciones de distintos threads cada ciclo de reloj:



1.5 Hardware y arquitecturas TLP en un chip

Vemos como afecta la paralelización a nivel de thread al hardware del chip, según la alternativa que se escoja:

| 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 | Captación: repartida o compartida; Resto: multiplexadas | <u>UF</u> : compartidas; <u>Resto</u> : repartidas o compartidas | replicado |
| Etiquetas para distinguir el thread de una instr. | Sí | Sí | Sí | No |
| Hardware para conmutar entre threads | Sí | Sí | No | No |

Las prestaciones en la tabla anterior concuerdan con la complejidad del hardware.

2 Lección 8: Coherencia en el sistema de memoria

Los computadores que implementan mantenimiento de coherencia son:

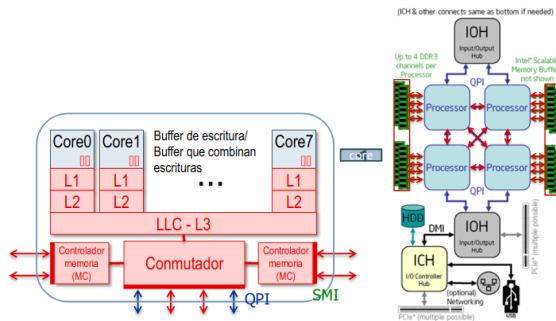
- Multiprocesadores:
- NUMA: los CC-NUMA y COMA
- UMA: los SMP

2.1 Sistema de memoria en multiprocesadores

El Sistema de memoria de los multiprocesadores incluye:

- Cachés de todos los nodos
- Memoria principal
- Controladores de memoria
- Buffers
 - De escritura/almacenamiento
 - Combinados entre escrituras-almacenamientos, etc
- Red de interconexión, o el medio de comunicación entre los componentes anteriores.

Por tanto, la comunicación de los datos entre los procesadores la realiza el *sistema de memoria*. Podemos tener copias de una dirección de MP en distintos puntos de nuestro sistema de memoria (p.e. distintos niveles de caché). Si estas copias no tienen el mismo contenido se produce lo que llamamos **incoherencia**, que a su vez produciría que la devolución de esa dirección de memoria no sea lo último que se ha leído.



2.2 Concepto de coherencia en el sistema de memoria

La incoherencia surge cuando se realiza una modificación sobre un dato(modificable) en uno de los nodos, y no se actualiza el resto de copias de ese dato dentro de los nodos del multiprocesador.

| CLASES DE E.D | EVENTOS | FALTA DE COHERENCIA |
|--------------------------------|--------------------------------------|---|
| Datos modificables | E/S | Cache-MP |
| Datos modificables compartidos | Fallo de Cache | Cache-MP |
| Datos modificables privados | Del thread/proceso al fallo de cache | Cache-MP, aunque también entre caches de distinto nivel |
| Datos modificables compartidos | Lectura de cache no autorizada | Cache-cache, entre caches del mismo nivel |

2.2.1 Métodos de actualización de Memoria Principal

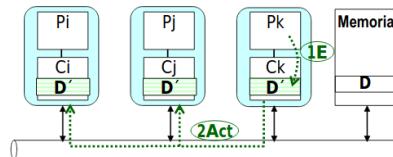
También valen para actualización entre último nivel de caché-MP y entre niveles distintos de caché.

- Escritura inmediata **write-through**: cada vez que un procesador escribe en su caché escribe también en memoria principal.
 - Por los principios de localidad temporal y espacial sería más rentable si se escribiera el bloque entero en MP.
 - No se admiten faltas de coherencia
- Posescritura (**write-back**): Se actualiza la memoria principal reescribiendo el bloque completo cuando dicho bloque sale de la caché.
 - Disminuye los accesos a caché por los principios de localidad temporal y espacial. Además, es más rentable escribir el bloque entero después de las modificaciones.
 - El controlador de la caché tiene un bit con información sobre si un bloque ha sido modificado o no. En esta alternativa sí que se permiten fallos de caché, así que habría que **añadir hardware** para detectar dicho bit.

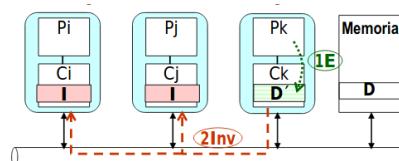
Cuando se comparten las cachés se utiliza posescritura, y en caso contrario, escritura inmediata.

2.2.2 Alternativas de Propagación de Escrituras

- Escritura con actualización (**write-update**): Cada vez que un procesador escribe en una dirección en su caché, se escribe en las copias de esa dirección en las demás cachés.



- Escritura con invalidación (**write-invalidate**): Antes que un procesador modifique una dirección en su caché se invalidan las copias del bloque en las demás cachés.
 - Reduce el tráfico, especialmente si los datos los comparten pocos procesadores.



Aún así, aunque se usen mecanismos de propagación, no se puede garantizar la coherencia.

2.2.3 Requisitos para mantener coherencia

1. Propagar las escrituras en una dirección (se ocupa el controlador de caché)

- La escritura en una dirección debe hacerse visible en un tiempo finito a otros procesadores.
- CONEXIÓN:
 - **BUS**: los paquetes de actualización/invalidación son visibles a todos los controladores de caché conectados al bus (muy sencillo de implementar).
 - **Difusión**: los paquetes de actualización se envían a todas las cachés, causando desaprovechamiento del ancho de banda.
 - Para mayor escalabilidad:
 - Filtrar los envíos: sólo se envían paquetes a las cachés con copias del bloque.
 - Mantener en una tabla (**directorio**) información sobre qué caché tiene copia de cada bloque.

2. Serializar las escrituras en una dirección

- Las escrituras en una dirección deben verse en el mismo orden por todos los procesadores.
- CONEXIÓN
 - **BUS**: el orden en el que los paquetes aparecen en el bus determina el orden en el que se ven por todos los nodos.
 - **Otros**, el orden en el que las peticiones de escritura llegan a *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.

- **Difusión**, en los ejemplos de las diapositivas se utiliza actualización para propagar las escrituras y el orden de llegada al home es el orden real para todos. Al ser actualización se envía a todos los nodos, para que todos acaben con el mismo contenido: el primero que llega al home confirma y escribe en todos.
- **Sin difusión con directorio distribuido**: en la caché hay copia del bloque y de quién tiene copia del bloque. Por tanto, las peticiones *solo se envían al home*. El home se encargará de mandar la actualización a quien tenga copia del bloque.
 - Mantiene el orden de llegada al home.

2.2.4 Directorio

Ya hemos visto que el directorio almacena información sobre el almacenamiento y modificación de cada bloque de memoria.

Alternativas de implementación

- Centralizado
 - Se comparte por todos los nodos.
 - Tiene información sobre los bloques de todos los módulos de memoria.
 - Distribuido
 - Las filas se dividen entre los nodos.
 - Cada nodo tiene un directorio con información sobre los bloques de su módulo de memoria.
-

2.3 Protocolos de mantenimiento de coherencia

Para diseñar un protocolo de mantenimiento de coherencia en el sistema de memoria se tienen que determinar:

- **Política de actualización de MP** (inmediata, posescritura o mixta).
- **Política de coherencia entre cachés** (invalidación, actualización, mixta).
- **El comportamiento**:
 - Definir los estados posibles de los bloques de caché y memoria.
 - Definir transferencias a generar entre eventos, es decir, qué paquetes se generan entre los eventos, qué nodos intervienen y el orden entre ellos. Los eventos pueden ser:
 - Lecturas/escrituras del procesador del nodo.
 - Llegada de paquetes de otro nodo.
 - Definir transiciones de estado para bloques de caché y memoria (cuando se pasa de un estado a otro).

Así, tenemos tres protocolos principales

1. Protocolos de espionaje (**snoopy**)
 - Se usa en buses y en sistemas con una difusión eficiente (nº de nodos pequeño o porque implementa difusión).

- Rentable si el número de nodos es pequeño.
2. Protocolos basados en **directorios**
 - Para redes sin difusión.
 - Para redes escalables, multietapa y estáticas.
 3. Esquemas **jerárquico**
 - Para redes jerárquicas, con jerarquía de buses, redes escalables, o redes escalables-buses.

Todos los que vamos a ver utilizan posescritura y escritura con invalidación.

2.3.1 Protocolo MSI de espionaje

Este protocolo busca minimizar el número de estados usando posescritura e invalidación.

ESTADOS

1. ESTADOS EN CACHÉ: mínimo

- Modificado (M), es la única copia válida en todo el sistema.
- Compartido(C,S): está válido, también válido en memoria y puede que haya copia válida en otras cachés.
- Inválido (I): se ha invalidado o no está físicamente.

Si se pide un bloque inválido, puede ser que esté físicamente pero no está actualizado, por tanto no se puede devolver.

2. ESTADOS EN MEMORIA

- Válido: puede haber copia válida en una o varias cachés.
- Inválido: habrá copia válida en una caché.

La respuesta de la memoria muchas veces se inhibe por el controlador de caché. Por ejemplo, si una caché ve una petición de lectura de un bloque de memoria que tiene, y que no está en la memoria, tendrá que inhibir el mensaje de la memoria (bloque inválido) y devolver dicho bloque (respuesta con bloque).

TRANSFERENCIAS

| Nombre | Descripción | Evento que la causa |
|----------|-------------------------------|--|
| PtLec | Petición de lectura de bloque | Lectura con fallo de caché del procesador del nodo. |
| PtLecEx | Petición de acceso exclusivo | Escritura del procesador en un bloque compartido o inválido (INVALIDACIÓN) |
| PtPEsc | Petición de posescritura | Por el reemplazo del bloque modificado. |
| RpBloque | Respuesta con bloque | Al tener en estado "M. el bloque solicitado por PtLec o PtLecEx |

Los dos primeros eventos son entre el nodo origen y el procesador, el tercero es entre el nodo origen y el controlador de cache y el último es entre el nodo origen y los demás nodos.

Hay que tener en cuenta que PtEx no existe en buses porque en un ciclo de bus siempre se devuelven un bloque, por tanto, sólo se usaría PtLexEx. Además, en MSI no se suele almacenar el estado del bloque en memoria.

Sin embargo, este protocolo no resulta del todo eficiente, pues en un código secuencial, en un solo procesador, a cada escritura que se quiera hacer se generarán PtLecEx, sin ser necesarios.

| 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 |
| Compart. (S) | Reemplazo | Genera paquete posescritura (PtPEsc) | Inválido |
| | PrLec | | Compartido |
| | PrEsc | Genera paquete PtLecEx (PtEx) | Modificado |
| Inválido (I) | PtLec | | Compartido |
| | PtLecEx | Invalida copia local | Inválido |
| | PrLec/PrEsc | Genera paquete PtLec | Modificado |
| | PrEsc | Genera paquete PtLecEx | Modificado |
| | PtLec/PtLecEx | | Inválido |

2.3.2 Protocolo MESI de espionaje

Este protocolo también utiliza posescritura e invalidación.

ESTADOS

1. ESTADOS EN CACHÉ

- **Modificado (M)**, es la única copia válida en todo el sistema.
- **Exclusivo (E)**, es la única copia del bloque válida en cachés, la memoria también está actualizada.
- **Compartido (C,S)**: está válido, también válido en memoria y en **al menos** otra cache.
- **Inválido (I)**: se ha invalidado o no está físicamente.

Si se pide un bloque inválido, puede ser que esté físicamente pero no está actualizado, por tanto no se puede devolver.

2. ESTADOS EN MEMORIA

- **Válido**: puede haber copia válida en una o varias cachés.
- **Inválido**: habrá copia válida en una caché.

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

2.3.3 Protocolo MSI basado en directorios

1. Sin difusión Recordamos que los estados en caché son Modificado(M), Compartido(C) e inválido(I), y los estados en bloque de memoria son Válido e Inválido.

Existen cinco tipos de nodos en MSI con directorios:

- 1- Solicitante(S)
- 2- Origen(O), es el que tiene información en su directorio sobre el bloque que se está tratando.
- 3- Modificado(M)
- 4- Propietario(P), tiene un bloque válido.
- 5- Compartidor(C).

- Tipos de paquetes
 - **Petición**
 - De solicitante a origen:
 - PtLec, lectura de un bloque.
 - PtLexEx, lectura con acceso exclusivo.
 - PtEx, petición de acceso exclusivo sin lectura.
 - PtPEsc, posescritura.
 - **Reenvío de petición**
 - Origen a nodos con copia (P, M , C):
 - RvInv, invalidación.
 - RvLec, lectura.
 - RvLexEx, lectura exclusiva.
 - **Respuesta**
 - Nodo propietario a origen
 - RPBloque, respuesta con bloque
 - RPInv, respuesta confirmando invalidación sin bloque.
 - RPBloqueInv, respuesta confirmando invalidación con bloque.
 - Nodo origen a solicitante
 - RPBloque, respuesta con bloque
 - RPInv, respuesta confirmando invalidación sin bloque.
 - RPBloqueInv, respuesta confirmando invalidación con bloque.

2. Con difusión

Los estados, tanto en caché como en memoria, son los mismos. La principal diferencia es que las peticiones se realizan por difusión, es decir, no van directamente al nodo origen, sino que se envían a los demás nodos también.

Aún así, las respuestas se envían al nodo origen, que es quién las gestiona.

Vemos las transferencias.

- Tipos de paquetes:

Los tipos de nodos tampoco cambian.

- **Difusión de petición**

- De solicitante a origen y propietario:
 - PtLec, lectura de un bloque
 - PtLecEx, lectura con acceso exclusivo.
 - PtEx, acceso exclusivo sin lectura
- De solicitante a origen
 - PtPEsc, posescritura

- **Respuesta**

- De propietario a origen
 - RPBloque, respuesta con bloque
 - RPInv, respuesta confirmando invalidación sin bloque.
 - RPBloqueInv, respuesta confirmando invalidación con bloque.
- Origen a solicitante
 - RPBloque, respuesta con bloque
 - RPInv, respuesta confirmando invalidación sin bloque.
 - RPBloqueInv, respuesta confirmando invalidación con bloque.

Nota. Varias cositas:

- Los UMA son más comunes en el mercado, aunque los NUMA van ganando terreno.
- Los protocolos con difusión tienen menos ancho de banda, es decir menor productividad.
- Los protocolos sin difusión aprovechan mejor el ancho de banda.
- En el peor caso, un acceso a memoria en difusión genera 3 paquetes y sin difusión genera 4, por tanto hay mayor latencia sin difusión.

3 Lección 9: Consistencia del sistema de memoria

3.1 Concepto de consistencia de memoria

Los multiprocesadores, como ya hemos visto, implementan **coherencia** en el hardware, es decir, van a ver el mismo orden-serie en los accesos a *una dirección de memoria*.

Sin embargo, diferenciamos este concepto con el de **consistencia**, que nos indica en qué orden van a ver los multiprocesadores todos los accesos a memoria.

La consistencia de memoria especifica las restricciones en el orden en el cual las operaciones de memoria deben parecer haberse realizado (operaciones a las mismas o distintas direcciones de memoria y emitidas por el mismo o distinto procesador).

Podemos diferenciar, según si estamos hablando de procesadores o de flujos de instrucciones, de **consistencia software** o **consistencia hardware**.

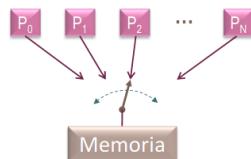
3.2 Consistencia secuencial

La **consistencia secuencial** es el modelo de consistencia que espera el programador de las herramientas de alto nivel. Es decir, es lo que se espera por lógica que ocurra.

La consistencia secuencial plantea varios requisitos:

- Todas las operaciones de un único procesador(thread) parezcan ejecutarse en el orden descrito por el programa de entrada al procesador. (**orden del programa**)
 - Todos los procesadores ven los accesos a memoria en el mismo orden en el que aparecen en el código que se está ejecutando.
- Todas las operaciones de memoria parecen ser ejecutadas una cada vez (**ejecución atómica**), es decir que todos los procesadores ven el MISMO acceso a memoria.

Este último requisito hace que se presente el sistema de memoria como una memoria global conectada a todos los procesadores a través de un conmutador central.



Veamos un ejemplo:

| | | | |
|--|---|--|--|
| Inicialmente $k1=k2=0$ P1 $k1=1;$ if ($k2=0$) { Sección crítica }; | | P2 $k2=1;$ if ($k1=0$) { 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 $copia$? |

- CASO 1: se espera que o bien entre P1 o bien entre P2 a la ejecución del bucle, pero que no entren los dos.
- CASO 2: se espera que, o bien $reg1=1$ o bien nada
- CASO 3: P2 se quedaría esperando hasta que se actualice el valor de A.
- Supuestamente este último código podría valer como un código de sincronización, siempre que el sistema en el que se ejecute tenga consistencia secuencial.

Sin embargo, actualmente ningún multiprocesador implementa consistencia secuencial, y por eso se recurre a un modelo más relajado con los requisitos.

3.2.1 Modelos de consistencia relajados

Son modelos de consistencia que, aunque se basan en el secuencial, difieren con él en los requisitos para poder incrementar las prestaciones.

Básicamente empiezan a permitir desordenar los accesos a memoria de *distintas direcciones*.

■ Orden del programa

- Hay modelos que permiten que se relaje en el código ejecutado el orden entre dos accesos a distintas direcciones (W-R, W-W, R-RW).

■ Atomicidad

- 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 la especificación de:

- Los órdenes de acceso a memoria que no garantiza el sistema de memoria.
 - Órdenes de un mismo procesador
 - Atomicidad en las escrituras
- Mecanismos que ofrece el hardware para garantizar un orden cuando sea necesario.

Un procesador puede leer anticipadamente o bien su propia escritura o bien la escritura de otro procesador, es decir, puede acceder a lo que haya escrito algún procesador *antes de que lo vean todos los demás*, y es algo que permite el **hardware**.

Además, se pueden añadir **instrucciones máquina** que hacen que el orden se mantenga cuando es código de comunicación:

- LDA: se ocupa de la adquisición y liberación de la memoria.
- DMB (u MEMBAR) actúa como una barrera de memoria, es decir hasta que no se hacen todos los accesos anteriores no se hacen los otros.
- I-m-e: garantiza la lectura-modificación-escritura atómica, añade orden también. Es decir, mientras se hagan esas instrucciones, ningún otro procesador va a leer, modificar, etc.

| Modelo | Orden del programa relajado W→R W→W R→RW | Orden global Lec. anticipada prosia 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 | I-m-e, STBAR (instrucción Store BARrier) |
| Sparc-RMO | Si | Si | MEMBAR (instrucción Memory BARrier) |
| PowerPC | Si | Si | SYNC, ISYNC (instrucciones SYNC/ORIZATION) |
| Itanium | Si | Si | LD, ACQ, ST, REL, MP (Acquisition Load, Release Store, Memory Fence) y cmpxchq\$, acq y otras I-m-e |
| ARMv7 | Si | Si | DMB (Data Memory Barrier) |
| ARMv8 | Si | Si | LDN (LDAR, STL, STLR (Load-Acquire, Store-Release), LDN), LDNE (LDAR, STL, STLR (Load-Release, Store-Release), LDNE), LDSE (LDAR, STL, STLR (Load-Release, Store-Release), LDSE), LDSE2 (LDAR, STL, STLR (Load-Release, Store-Release), LDSE2), DMB |

Volvemos al ejemplo anterior:

| | | |
|--|--|--|
| Inicialmente k1=k2=0 P1 k1=1; if (k2=0) { Sección crítica }; | P2 k2=1; if (k1=0) { Sección crítica }; | NO se comporta como SC los que relajan el orden W→R |
| P1 A=1; 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 |
| 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 |

- CASO 1:
- CASO 2:
- P1 actualiza su caché y propaga la actualización
- P2 llega al if y como ya ha llegado la propagación de P1, entra a hacer escritura.
- P2 actualiza las cachés con copia de B

3.2.2 Modelo que relaja W-R

Estos modelos permiten que una lectura pueda adelantar a una escritura previa en el orden del programa, **evitando dependencias RAW**.

Características:

- Es implementado por sistemas con buffer FIFO de escritura para los procesadores. El buffer evita que las escrituras retarden la ejecución del código bloqueando lecturas posteriores.
- Permiten que el procesador pueda leer una dirección directamente del buffer. Por ejemplo cuando se lee antes que otros procesadores una escritura propia.
- Puede incluir **instrucciones de serialización**, que garantizan un orden correcto.
- Hay sistemas en los que se permite que un procesador pueda hacer lectura anticipada (acceso no atómico).
- Si queremos asegurar el acceso atómico se usarían instrucciones como la de Intel anterior.

Modelo que relaja W-W y W-R

Además de tener lo mencionado anteriormente, que relaja W-R, tiene las siguientes características:

- Tiene buffer de escritura que permite que lecturas adelanten a escrituras.
- Permiten que el hw solape escrituras a memoria a distintas direcciones, para que puedan llegar a la memoria principal o a cachés de todos los procesadores fuera del orden del programa.
- Se proporciona hardware que garantiza los dos órdenes. Por ejemplo sistemas Sun Sparc.

4 Lección 10: Sincronización

4.1 Comunicación en multiprocesadores y necesidad de usar código de sincronización

4.1.1 Comunicación uno-uno

Empecemos por la comunicación más básica que podemos utilizar: la comunicación uno a uno.

Esta comunicación debe ser la comunicación mínima garantizada y se da entre un generador, que es quien envía el mensaje, y un consumidor, que es quien lo recibe.

En este tipo de comunicación se tiene que garantizar que :

- El consumidor lea la **variable compartida** cuando el proceso que envía haya escrito en la variable el dato.
- Si reutilizamos la variable se debe garantizar que no se envía un nuevo dato hasta que se haya leído el anterior.

Para poder asegurar lo anterior usamos **código de sincronización**.

Sin embargo, hay veces que el código de sincronización no es suficiente para asegurar la comunicación correcta, y dependiendo del modelo de consistencia del sistema habría que añadir instrucciones que garantizan el orden (DMB).

4.1.2 Comunicación colectiva y condiciones de carrera

En la comunicación colectiva que vemos a continuación se reparten las iteraciones del bucle de manera implícita entre los distintos threads. Además, usamos variables privadas a cada thread que asegure que obtiene la suma correcta (*sump*).

| Secuencial | Paralela (<i>sum=0</i>) |
|--|--|
| <pre>for (i=0 ; i<n ; i++) { sum = sum + a[i]; } printf(sum);</pre> | <pre>for (i=ithread ; i<n ; i+=nthread) { sump = sump + a[i]; } sum = sum + sump; /* SC, sum compart. */ if (ithread==0) printf(sum);</pre> |

Si analizamos dicho código llegamos a la conclusión de que si varios thread leen el mismo valor de *sum*, y le suman su correspondiente *sump*, el valor final de *sum* será el del thread más lento, y debería contener la suma de los dos thread. $R_S^i - R_S^j - (W_S^i \mid W_S^j)$.

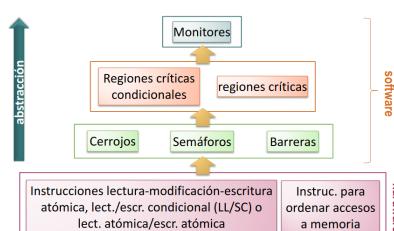
Esta situación se denomina **condición de carrera**, y el código afectado se denomina **sección crítica**. En este ejemplo resulta en que el valor final de *sum* no incluiría todos los *sump*. Por tanto, se debería cumplir lo siguiente:

- La lectura-modificación de *sum* se debe hacer en exclusión mutua.
 - Usamos cerrojos
- El proceso 0 no tiene que imprimir hasta que todos los *sump* se hayan acumulado.
 - Usamos barreras

En resumen, las **secciones críticas** serán secuencias de instrucciones con una o varias direcciones compartidas (variables compartidas) que se deben acceder en exclusión mutua.

4.2 Soporte software y hardware de sincronización

El soporte necesario para la sincronización se representa por el siguiente esquema:



Sobre el esquema haremos varias anotaciones:

- Los cerros, semáforos y barreras se implementan mediante instrucciones máquina.
- El soporte hardware es quién se encarga de hacer la sincronización eficiente.
- Conforme aumentamos la abstracción, menos conocimientos son necesarios para implementar sincronización, pero el resultado serán programas más ineficientes.

4.3 Cerros

4.3.1 Definición y características

Los cerros son un soporte software que permiten sincronizar mediante dos operaciones:

- **lock () o cierre del cerrojo:** adquieren el derecho a acceder a una sección crítica mediante la apertura o cierre del cerrojo k:
 - Si varios procesos intentan el cierre a la vez, sólo uno de ellos lo consigue y el resto pasa a una etapa de espera.
 - Todos los procesos que accedan a `lock ()` con el cerrojo cerrado se esperarán.
- **unlock () o apertura del cerrojo:** libera a uno de los threads que esperan el acceso a una sección crítica (éste adquiere el cerrojo):
 - Si no hay threads en espera permite que el siguiente thread que ejecute `lock ()` lo adquiera sin espera.

Básicamente, los cerros aseguran que solamente hay un FI (Flujo de Instrucciones) en la sección crítica, mediante una variable k, que indicará el estado del cerrojo (k=0 abierto, k=1 cerrado).

| Secuencial | Paralela |
|---|---|
| <pre>for (i=0 ; i<n ; i++) { sum = sum + a[i]; }</pre> | <pre>for (i=ithread ; i<n ; i=ithread) { sump = sum + a[i]; } lock(k); sum = sum + sump; /* SC, sum compart. */ unlock(k);</pre> |

4.3.2 Componentes en un código para sincronización

Método de adquisición

Es el método por el cual un thread trata de adquirir el derecho a pasar a utilizar unas direcciones compartidas. En otras palabras, cómo actualiza k para que los demás no puedan verlo.

- Utilizando lectura-modificación-escritura atómicas.
- Utilizando LL/SC, instrucción compuesta por:
 - Instrucción de lectura enlazada (linked load LL)
 - Operación de escritura condicionada (Store Conditional) a que entre LL y SC ningún otro SC haya leído.
 - En esta operación se permite el acceso de otros FI pero no se llevará a cabo la escritura

Método de espera

Es el método por el cual un thread espera a adquirir el derecho a pasar a utilizar unas direcciones compartidas. La espera se puede implementar mediante:

- **Espera ocupada:** los threads esperan en un bucle while hasta que se abre el cerrojo.
- **Suspensión del proceso o del thread,** el procesador conmuta a otro thread y éste se pasa a una cola del SO.

Si la sección crítica es corta, se suele implementar una espera ocupada, pues la suspensión del proceso conlleva una llamada al sistema operativo y debemos asegurarnos de que no la hacemos para nada.

Método de liberación

Es el método utilizado por un thread para liberar a uno (cerrojo) o varios threads en espera (barrera).

4.3.3 Implementaciones

Cerrojo Simple

Se implementa con una variable compartida k que toma dos valores: abierto(0), cerrado(1).

- La apertura del cerrojo (`unlock()`) se realiza mediante una **operación indivisible** que escribe un 0.
- El cierre del cerrojo (`lock()`) primero lo lee y luego escribe un 1.
- El resultado de la lectura:
 - Si el cerrojo estaba cerrado el thread espera hasta que otro thread ejecute `unlock()`.
 - Si estaba abierto adquiere el derecho a pasar a la sección crítica y cierra el cerrojo (`k=1`).

Claramente la operación que incluye la lectura y escritura de k (pasar de 0 a 1) debe ser atómica, de lo contrario, varios threads podrían pasar a la sección crítica.

Cerrojo Simple II

Se debe añadir lo necesario para asegurar que la lectura y escritura de 1 en el cerrojo sea en **exclusión mutua**, es decir, que sea una **operación atómica**, que mientras un FI hace el R-W de k, ningún otro FI pueda acceder a k.

```

lock (k)
lock(k) {
  while (leer-asignar_1-escibir(k) == 1) {};
} /* k compartida */

unlock (k)
unlock(k) {
  k = 0;
} /* k compartida */

```

En el caso de que haya más de un flujo de instrucciones, el primero que realice la operación atómica será el único que lea un 0, los demás leerán un 1 y por tanto se quedan en espera.

Cerrojos con OpenMP

Utilizan las directivas de OpenMP vistas en prácticas:

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

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).

En este tipo de cerrojos se añaden dos contadores:

- Contador de adquisición: `contadores.adq`
- Contador de liberación: `contadores.lib`

Y funciona de la siguiente manera:

- Cuando un thread llega a la sección del cerrojo:
- Coge número, es decir, almacena en una variable local el valor de `contadores.adq` en el momento de su llegada.
- Incrementa `contadores.adq` en uno.
- Mientras no coincida su número con el número de `contadores.lib`, espera.
- Cuando un thread termina de ejecutar la sección crítica:
- Incrementa `contadores.lib` para que el siguiente número pueda pasar.

Como ya he mencionado antes, es importante mantener el acceso a los contadores en exclusión mutua, teniendo en cuenta el orden de los accesos que proporcione el modelo de consistencia del sistema en el que estemos trabajando.

4.4 Barreras

Las barreras son zonas de código en las que los FI se quedan esperando hasta que llegan todos.

Normalmente constan de un **contador** y de una **bandera**. El primero informa de cuántos thread han llegado a la barrera, y la segunda *se levantará* cuando el contador tenga el valor necesario, es decir, cuando todos los thread hayan llegado.

Claramente la consulta y modificación de la bandera (en el caso del primer thread que llegue, poner la bandera a 0, y en el caso del último thread que llegue poner el contador a 0 y la bandera a 1, los demás thread solamente tienen que incrementar el contador), se tiene que hacer en **exclusión mutua**, y por tanto se tendrá que tener en cuenta el modelo de consistencia del sistema.

```
Barrera(id, num_threads) {
    if (bar[id].cont==0) bar[id].bandera=0;
    cont_local = ++bar[id].cont;
    if (cont_local ==num_threads) {
        bar[id].cont=0;
        bar[id].bandera=1;
    }
    else espera mientras bar[id].bandera==0;
}
```

- Acceso Ex. Mutua.

- Implementar **espera**. Si **espera ocupada**:

→ **while (bar[id].bandera==0) {};**

4.4.1 Problema con barreras reutilizadas

Sin embargo, surge un problema con las actualizaciones mencionadas anteriormente, y es que si bien hacen que las banderas se puedan reutilizar, puede darse el siguiente escenario:

- Los thread entran uno a uno en el bucle de espera.
- Antes de que llegue el último thread, uno de los que estaban esperando (FI_i) es suspendido por el SO.
- FI_i pasa a ejecutar la rutina de tratamiento de interrupción.
- Llega el último thread y actualiza los valores:
 - Contador 0
 - Bandera 1
- Los thread continúan sus ejecuciones hasta llegar a una **segunda barrera**
- Van entrando y mientras tanto el FI_i vuelve a su ejecución anterior y por tanto reinicia la ejecución dentro del bucle de espera.

Esto hace que en un momento dado como la bandera es 0 en ambas barreras, el FI_i se quede esperando en la barrera anterior y todos los demás threads esperen en la segunda barrera esperando a que FI_i llegue, pero no puede.

Solución:

La solución para la situación descrita anteriormente pasa por mantener el uso de la bandera anterior, es decir si la barrera anterior se ha abierto con $b=1$, en la siguiente barrera se esperará mientras $b=1$. Por tanto, en cada barrera se espera en un valor distinto que se almacena en una variable local.

```

Barrera sense-reversing
Barrera(id, num_procesos) {
  bandera_local = !(bandera_local); //se complementa bandera local
  lock[bar[id].cerrojo];
  cont_local = ++bar[id].cont; //cont_local es privada
  unlock[bar[id].cerrojo];
  if (cont_local == num_procesos) {
    bar[id].cont = 0; //se hace 0 el cont. de la barrera
    bar[id].bandera = bandera_local; //para liberar thread en espera
  }
  else while (bar[id].bandera != bandera_local) {}; //espera ocupada
}
  
```

4.5 Apoyo hardware a primitivas software

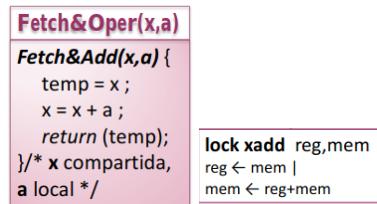
4.5.1 Instrucciones de lectura-modificación-escritura atómicas

Tenemos tres operaciones básicas que se realizan en exclusión mutua:

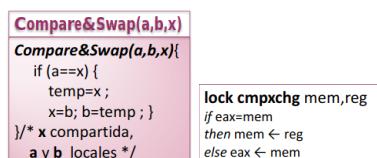
- **Test&Set (x)**: hace una comprobación del contenido de una variable al mismo tiempo que varía su contenido en caso que la comprobación se realizó con el resultado verdadero.

| |
|---|
| Test&Set (x) |
| <pre> Test&Set (x) { temp = x ; x = 1 ; return (temp) ; } /* x compartida */ </pre> |
| mov reg,1 xchg reg,mem reg ↔ mem |

- **Fetch&Oper (x, a)**: incrementa de manera atómica el valor pasado como parámetro x con a y devuelve el valor original de x.
- El lock de la implementación ensamblador garantiza la exclusión mutua si en esa escritura se realiza alguna lectura o escritura.



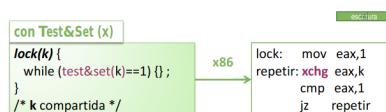
- **Compare&Swap (a, b, x)**: compara los contenidos de a y x y si son el mismo intercambia b y x.
- En la implementación ensamblador que vemos hay una pequeña diferencia, y es que se llama como operando implícito eax y en el caso de que coincidan los valores al intercambiar se carga el valor de la memoria a eax en vez de al registro.



4.5.2 Cerrojos simples con instrucciones atómicas

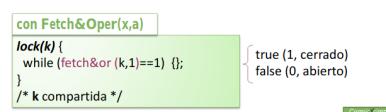
Test&Set

Vemos que el software de bajo nivel está escrito en verde y las primitivas de los procesadores están escritas en morado. Básicamente garantizamos que el acceso a la variable de cierre k se haga en exclusión mutua, mediante la comprobación y posible escritura con test&set.



Fetch&Or

Hace el or con k y 1 de manera atómica, para que si k=0, el or dará 1 y se quedará esperando y si k=1, es decir que no hay ningún FI en la sección crítica del cerrojo, el or sacará 0 y actualizará k y podrá continuar la ejecución.

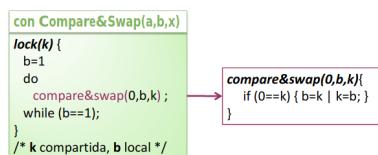


Compare&Swap

Con esta instrucción atómica aseguramos que si $a(0) = k$, entonces k empieza a valer b , por tanto solo se pasa a la sección crítica si $k=0$. Bastaría con poner como condición de espera $b=1$.

En otras palabras, se llama a compare&swap con $(0,1,k)$:

- Si $k=0$, se intercambiarán valores y por tanto $k=b=1$ y $b=0$
- Si $k=1$, no se intercambian valores y $b=1$, $k=1$, **cerrojo cerrado**.



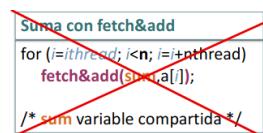
4.5.3 Consistencia de liberación

Podríamos poner todos los ejemplos que hay en las diapositivas, pero en realidad lo único que hace falta saber de esta sección es que hay procesadores (ARM) que por su consistencia en el sistema de memoria, donde una lectura puede adelantar a una escritura, no garantizan que no se abre el cerrojo hasta que no acabe todo el código de la sección crítica.

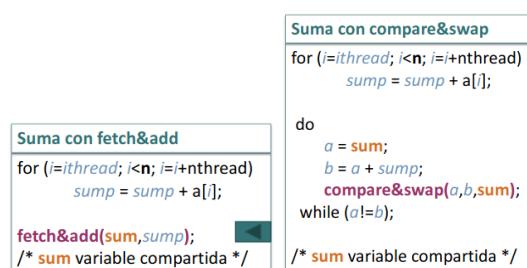
4.5.4 Algoritmos eficientes con primitivas hardware

Usar primitivas hardware hace que los algoritmos con cerrojo sean más eficientes primero porque no hay código de lock(), y segundo porque no tenemos que hacer uso de variables locales.

En el primer caso vemos que aunque se use una instrucción atómica, no hay paralelismo:



En el segundo y tercer ejemplo se hace la suma de elementos de un vector con fetch&add() y con compare&swap():



Por último, vemos la diferencia en OpenMP entre las directivas Atomic y Critical:

- Atomic: utiliza primitivas hardware como las vistas anteriormente
- Critical: utiliza los cerrojos más ineficientes

Por tanto, deberíamos usar **atomic**.

Nota. Varias cositas otra vez:

- El lock() siempre se tiene que hacer con instrucciones máquina.
- El unlock() se puede hacer sin instrucciones máquina (si estamos en un modelo que solo relaja W-R).
- Si el modelo relaja W-W, tenemos un problema con el unlock.

IV | Tema 4. Arquitecturas con Paralelismo a nivel de Instrucción

Nota. Este tema se ha reducido este año, así que no fiarse mucho.

Recordamos que las arquitecturas con ILP ejecutan múltiples instrucciones concurrentemente o en paralelo, y son por ejemplo **cores escalares segmentados, superescalares o VLIW/EPIC**. En este tema veremos las características de dichas arquitecturas y cómo reducir su tiempo de ejecución.

Existen dos tipos de microarquitectura de núcleos ILP:

- Superescalar
- VLIW

1 Riesgos y dependencias

Tenemos que tener el cuenta que a la hora de ejecutar concurrentemente o en paralelo varias instrucciones, nos podemos encontrar con riesgos que hacen que se tengan que incluir instrucciones `nop` y que retrasan lo que sería una ejecución ideal.

Estas dependencias se dividen en tres tipos:

1. Dependencias de **datos**, que a su vez incluyen:
 - Dependencias WAW y WAR.
 - Dependencias RAW
2. Dependencias de **control** (saltos).
3. Dependencias **estructurales** (se dan cuando dos instrucciones necesitan la misma unidad funcional).

Veremos a continuación de qué manera se pueden reducir o evitar estas dependencias dentro de cada tipo de arquitectura ILP.

2 Microarquitecturas ILP superescalares

2.1 Dependencias

En una arquitectura superescalar, las dependencias las gestiona el **hardware**. Que se encarga de:

- Extraer paralelismo, y por tanto eliminar dependencias **WAW** y **WAR**. Se puede conseguir mediante:

- El uso de registros distintos a los de la arquitectura.
- Evitar problemas por dependencias que no se han podido eliminar. En este caso estamos tratando:
 - Dependencias **RAW**, usando predicción con antelación, podemos incluso evitar la *penalización por salto*.
 - Dependencias de **control**
 - Dependencias **estructurales**

Una observación dentro de estas arquitecturas es que **no introducen instrucciones `nop`**, pues el hardware se encarga de eliminar y evitar las dependencias.

2.1.1 Ejemplo de dependencias

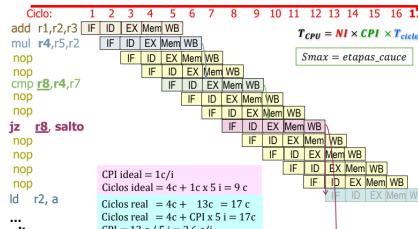
Por ejemplo, en la figura vemos que para ejecutar 5 instrucciones, la situación ideal sería:

$$CPI(\text{ciclos}/\text{instrucción}) = 1 \text{ y}$$

$$Num_{ciclos} = 4 + (1 + 5\text{instrucciones}) = 9\text{ciclos}$$

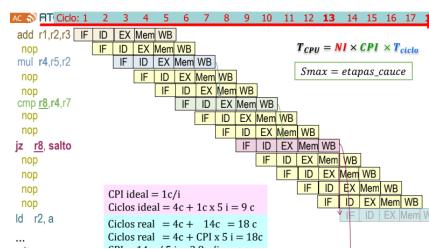
Sin embargo, después de las instrucciones `nop` añadidas tendremos:

- **Riesgo de datos y control:**



- Ciclos_real = $4c * 13c = 17c = 4c + CPI * 5i = 17c$, por tanto, vemos que $CPI = 17c/5i = 2.6\text{ciclos}/\text{instrucción}$

- **Riesgo estructural**



- Ciclos_real = $4c * 14c = 18c = 4c + CPI * 5i = 18c$, por tanto, vemos que $CPI = 14c/5i = 2.8\text{ciclos}/\text{instrucción}$.

2.2 Hardware en superescalares

Sabemos que los superescalares tienen una arquitectura segmentada. Por tanto, primero vamos a ver qué pinta tiene el cauce segmentado:

2.2.1 Etapas del cauce

Las instrucciones se captan desde la caché de primer nivel (L1), que a su vez está dividida en una sección de código y otra de datos.

Sabemos que el cauce se divide en distintas etapas:

1. Captación(IF) desde la caché L1.

- Detección de saltos y predicción de saltos incondicionales
- Hardware utilizado:
 - Hardware de precaptación (se puede añadir incluso una etapa que haga esto).
 - Tabla de saltos, que realiza la ejecución especulativa para reducir la penalización por saltos.

2. Decodificación(ID)- Emisión (Iss)- Envío a Unidades Funcionales

■ Características:

- Se tiene que evitar problemas por dependencias RAW y estructurales.
- Se renombran los registros, para evitar dependencias WAW Y WAR
- Ejecución especulativa de instrucciones.
- Captura de operandos desde registros se puede hacer desde registros de la arquitectura, o registros de renombrado.
- La estación de reserva (*cola* para una unidad funcional) puede ser **centralizada** o **independiente**, según cómo se repartan las unidades funcionales.
- La decodificación se realiza de forma ordenada. Sin embargo, a partir del envío de instrucciones a las UF, se procesan de forma desordenada.

■ Hardware utilizado:

- Ventana de instrucciones
- Buffer de renombrado
- Buffer de reorden, en muchos casos actúa como buffer de renombrado.

3. Ejecución (EX)

4. WB

- Termina el procesamiento actualizando los registros de la arquitectura **en el orden del programa**, por tanto, depende del modelo de consistencia que se tenga, lo ideal para mantener el orden sería implementar consistencia secuencial del procesador, aunque ya sabemos que no es posible.

2.2.2 Emisión

Renombramiento de registros

Es una técnica que se utiliza para evitar el efecto de las dependencias WAR (Antidependencias en la emisión desordenada) y WAW, (Dependencias de Salida, en la ejecución desordenada). Consiste en asignar cada escritura a un registro físico distinto para poder ejecutar las instrucciones en paralelo. Se puede implementar de dos formas:

- *Implementación estática*, en tiempo de compilación.

- *Implementación dinámica*, en tiempo de ejecución
 - Supone añadir circuitería adicional y registros extra.

El **hardware** que se utiliza para el renombramiento de registros se llama **Buffers de Renombrado**, cuyas características son:

1. **Tipos**, pueden ser separados o mezclados con los registros de la arquitectura.
2. **Número** de Buffers.
3. **Mecanismos para acceder**, pueden ser asociativos o indexados.
4. **Velocidad del renombrado**, o el máximo número de nombres asignados por ciclo que admite el procesador.

Algoritmo de Tomasulo

El Algoritmo de Tomasulo es una forma de implementar el renombramiento de registros, y se usa en la mayoría de arquitecturas hoy en día. En primer lugar veremos sus componentes:

- **Banco de registros de la arquitectura**
- **Estaciones de reserva**, para las unidades funcionales.
- **Banco de registros de renombrado**, dentro del buffer de renombrado.

Campos de las estaciones de reserva

Las estaciones de reserva contienen los siguientes campos:

- OC: operación a realizar
- OS1: operando 1
- VS1: estado del operando 1 (1=válido, 0=inválido)
- OS2: operando 2
- VS2: estado del operando 2 (1=válido, 0=inválido)
- Rdestino: registro *del buffer de renombrado* donde se almacenará el valor obtenido después de realizar la operación.

Cuando un operando no está en estado válido inicialmente, es decir que depende del resultado de otra operación, se almacena como operando el valor del registro de renombrado que tendrá el valor resultante de la operación. De esta manera, cuando dicha operación haya acabado, la estación de reserva puede **espiar** el bus y buscar el valor al buffer de renombrado.

Campos del buffer de renombrado

- #: número de registro (dentro del banco de registros de renombrado) que se va a usar para renombrar.
- EV: Si el registro se está usando para renombrar algo o no(1/0).
- Dest: registro (dentro del banco de registros de la arquitectura, R_Ac) que se quiere renombrar.

- Valid: si la instrucción que provocó el renombrado ha acabado y tenemos un valor útil o no (1/0).
- Ult: si es el último renombrado del registro.

El buffer de renombrado se comparte entre las unidades funcionales y las estaciones de reserva.

Emisión de operaciones

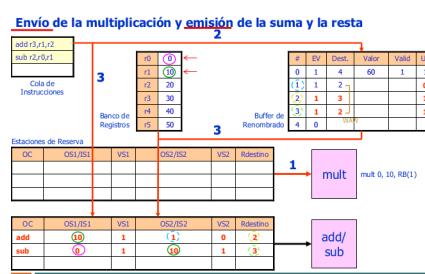
Las operaciones se pueden emitir a la estación de reserva de manera ordenada o desordenada (permite más paralelismo). Pero solo se podrán emitir en paralelo instrucciones sin dependencias.

Además, tenemos que tener en cuenta que en las arquitecturas actuales, operaciones como la suma o la resta tardan menos que la multiplicación. Y las instrucciones store se ejecutan **en cuanto** disponen del operando a almacenar.

Cuando se finalice la ejecución de las instrucciones se pasará a los registros de la arquitectura el valor del último renombrado que se ha hecho.

En la figura estamos ejecutando:

```
1 mult r2, r0, r1
2 add r3, r1, r2
3 sub r2, r0, r1
```



2.2.3 Consistencia del procesador y buffer de reorden

Hemos visto los modelos de consistencia de memoria en el tema anterior, recordamos:

- **Consistencia de memoria:**
 - Débil/relajado: los accesos a memoria (Load/Store) se pueden hacer de manera desordenada siempre que no afecten a las dependencias.
 - Deben detectarse y resolverse las dependencias de acceso a memoria.
- **Consistencia del orden de los accesos a memoria**
 - Fuerte: los accesos a memoria deben realizarse estrictamente en el orden del programa
 - Se consigue mediante el ROB.

De la misma manera, podemos analizar el concepto de consistencia en el procesador:

- **Consistencia de procesador:**
 - Débil/relajado: las instrucciones se pueden completar de manera desordenada siempre que no afecten a las dependencias.

- Deben detectarse y resolverse las dependencias de acceso a memoria.
- **Consistencia en el orden en el que se completan las instrucciones**
 - Fuerte: las instrucciones deben completarse estrictamente en el orden del programa
 - Se consigue mediante el ROB.

Además, podemos ver que si no dejamos que se relaje el orden (W->R), es decir que si no dejamos que las lecturas adelanten a escrituras, no habrá un aprovechamiento muy grande del paralelismo. En resumen, para que se pueda aprovechar el paralelismo es esencial:

- Relajación del orden W-R (lectura adelanta).
- Uso del renombrado de registros.
- Usar ejecución especulativa: se supone que una instrucción no plantea problemas (dependencias, saltos, etc), y se ejecuta. Si al ejecutarla se comprueba que sí que los planteaba, no se actualizan los registros de la arquitectura después. De esta manera, es como si no se hubiera ejecutado.

Todo lo mencionado anteriormente lo podemos implementar a la vez en una estructura hardware llamada **Buffer de reordenamiento**.

2.2.4 Buffer de Reordenamiento (ROB)

Es una estructura hardware implementada como una cola FIFO, en la que se van almacenando las instrucciones que se decodifican (emiten) según el orden del programa. Veamos algunas de sus características:

- El puntero de cabecera apunta a la siguiente posición libre, y el **puntero de cola** a la siguiente instrucción a retirar.
- Las instrucciones se introducen en el ROB en orden de programa estricto y pueden estar marcadas como:
 - Emitidas **i**, están en la estación de reserva y uno de sus operandos no está disponible.
 - En ejecución **x**, están en la estación de reserva con los operandos disponibles, se ejecuta.
 - Finalizada **f**.
- Una instrucción sólo se puede retirar y pasar a modificar los registros de la arquitectura **si se ha terminado su ejecución** y la de todas las instrucciones que la preceden. Además, si la instrucción se ejecutaba con especulación, se retirará si ya se ha resuelto la especulación.
- La consistencia se mantiene porque sólo las instrucciones que se retiran del ROB se completan y se retiran en orden estricto del programa.

- #: Número de instrucción dentro del ROB
- codop: Operación a realizar
- N^o inst.: número de instrucción dentro del programa
- Reg. Destino: Registro destino
- Unidad: Unidad funcional que se ocupará de la instrucción
- Resultado: Resultado de la ejecución.
- Ok: Si ha finalizado la ejecución

- Marca: Estado de la instrucción (i,x,f)

El ejemplo completo está en las diapos, pero el ROB tendrá esta forma:

| # | codop | Nº Inst. | Reg. Dest. | Unidad | Resultado | ok | marca |
|---|-------|----------|------------|----------|-----------|----|-------|
| 3 | mult | 8 | r1 | int_mult | - | 0 | x |
| 4 | st | 8 | - | store | - | 0 | i |
| 5 | add | 9 | r2 | int_add | - | 0 | x |
| 6 | xor | 10 | r1 | int_alu | - | 0 | i |

2.2.5 Ejecución especulativa

La ejecución especulativa, como ya hemos visto, consiste en suponer que la instrucción se ejecuta sin problemas (instrucciones con salto, dependencias, ...). Sin embargo, tenemos que analizar la ejecución especulativa en instrucciones con salto.

¿Cómo se detectan los saltos en la captación? Cuando se actualiza el PC en la instrucción anterior (por tanto PC apuntará a la instrucción de salto), se consulta la tabla de salto con la dirección de la instrucción de PC. Dicha tabla tiene una entrada por instrucción con salto, y contiene los siguientes campos:

- Dirección de la instrucción con salto
- Dirección del objetivo del salto
- Bits de historial, con información sobre ejecuciones anteriores de esta instrucción.

De esta manera, la primera vez que se ejecute la instrucción de salto se crea la entrada en la tabla de saltos y se incluye una predicción. Y así, cada vez que se ejecute la instrucción, se podrá tomar una decisión basada en los bits de historial que **elimina la penalización por completo**.

La decisión se llama *predicción de salto* y puede seguir uno de los esquemas siguientes:

- **Predicción Fija**, donde se toma siempre la misma decisión.
 - *Taken*, si se salta.
 - *Not taken*, si no se salta.
- **Predicción Verdadera**, la decisión se toma mediante:
 - *Predicción Estática* (se suele usar en bucles, donde *casi siempre* volvemos a ejecutar el código de bucle) o según los atributos de la instrucción de salto:
 - Código de operación
 - Desplazamiento
 - Decisión del compilador
 - *Predicción Dinámica*, según el resultado de ejecuciones pasadas de la instrucción (información almacenada en los bits de historial). Por ejemplo:
 - *Predicción con 1 bit de historia*, donde hay dos estados posibles: **Tomado(1)** y **No tomado(0)**, según si el salto se toma o no. Las flechas indican las transiciones de estado dependiendo de lo que se produce al ejecutarse la instrucción. Siempre que se hace una predicción fallida se cambia de estado.
 - *Predicción con 2 bits de historia*, donde existen cuatro estados posibles:

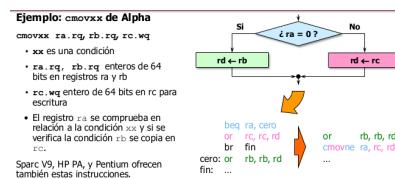
- **Tomado Fuerte** (11)
 - **Tomado Débil** (10)
 - **No Tomado Débil** (01)
 - **No Tomado Fuerte** (00)

La primera vez que se ejecuta un salto se inicializa con predicción estática. Las flechas indican las transiciones de estado dependiendo de lo que se produce al ejecutarse la instrucción.

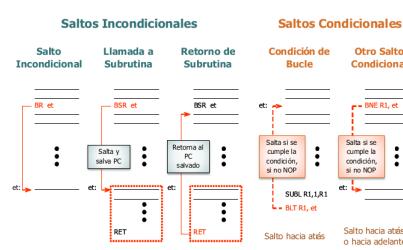
- *Predictión con 3 bits de historia* (o número impar de bits), donde cada entrada guarda las últimas ejecuciones del salto. Se salta o no según el bit mayoritario (mayoría de 1=salto).
 - Para actualizar el valor, se desplaza un bit a la izquierda el valor actual y se introduce en la izquierda la decisión tomada (1 salto, 0 no salto).

Además, podemos reducir el número de instrucciones de salto **includiendo en el repertorio máquina** instrucciones con operaciones condicionales. Es decir, que realizan la instrucción si se cumple una condición, por eso disponen de dos partes:

1. Condición
 2. Operación



Podemos clasificar los saltos según la figura siguiente:



3 Microarquitecturas VLIW

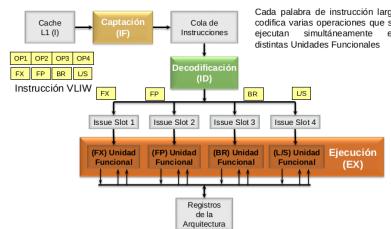
3.1 Riesgos y dependencias

Ya hemos visto que en las microarquitecturas superescalares los riesgos y las dependencias se resolvían mediante el hardware. Sin embargo en las arquitecturas VLIW, de menor potencia (usadas en sistemas empotrados), no se dispone de hardware para esa labor, por tanto, se encarga **el software** (compilador, programador,...) de:

- Extraer paralelismo mediante la reducción de dependencias:
 - WAR, WAW y RAW
 - Control, por la eliminación de las instrucciones de control, como por ejemplo saltos, y utilización de set o cmov.
 - Estructurales, se reduce uso de unidades funcionales para carga o almacenamiento (set, lea).
- Evitar problemas causados por las dependencias no eliminadas, mediante la inclusión de instrucciones nop.
 - Hay que tener en cuenta que a menor número de instrucciones nop, mayor aprovechamiento de la arquitectura y por tanto mayor reducción de las dependencias.

3.2 Características generales

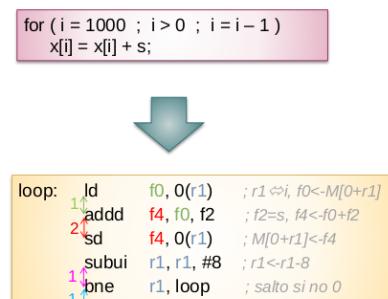
En las microarquitecturas VLIW se captan a la vez las instrucciones que se van a captar en paralelo. El *compilador* o el *programador* es quién se encarga de organizar las instrucciones en palabras según el número de unidades funcionales disponibles. En el ejemplo siguiente, vemos que tenemos 4 unidades funcionales, y por tanto se tienen que captar de 4 en 4 instrucciones.



3.3 Planificación Estática

Comparación con escalar

En el código que se muestra a continuación existen dependencias RAW entre cada dos instrucciones consecutivas, y además tenemos una instrucción de salto que controla el final del bucle. Parece que no se puede aprovechar mucho ILP:



Sin embargo, si suponemos que hay suficientes unidades funcionales para la suma, y que **cuando hay dependencias de tipo RAW, los retardos introducidos son los siguientes:**

| Instr. que produce el resultado | Instr. que usa el resultado | Espera |
|---------------------------------|-----------------------------|--------|
| Operación FP | Operación FP | 3 |
| Operación ALU | Store | 2 |
| Load | Operación FP | 1 |
| Load | Store | 0 |

Podemos comparar las siguientes opciones para implementar el bucle y comprobaremos si son beneficiosas en una arquitectura VLIW:

| | Opción 1 (Sin desenrollar) | Opción 2 (Sin desenrollar) | Ciclos |
|------|------------------------------|------------------------------|--------|
| loop | ld f0,0(r1) ; f4< M[r1] | ld f0,0(r1) | 1 |
| | nop | subi r1, r1, #8 | 2 |
| | addd f4, f0, f2 | addd f4, f0, f2 | 3 |
| | nop | 1. nop | 4 |
| | nop | bne r1, loop | 5 |
| | sd f4, 8(r1) ; M[8(r1)] < f4 | sd f4, 8(r1) ; M[8(r1)] < f4 | 6 |
| | subi r1, r1, #8 | | 7 |
| | nop | | 8 |
| | subi r1, r1, #8 | | 9 |
| | bne r1, loop | | 10 |
| | nop | | |

Y si organizamos las instrucciones según las unidades funcionales, tal y como lo hemos descrito en el apartado anterior (rellenando con nop si no disponemos de instrucción para esa unidad funcional), tendremos:

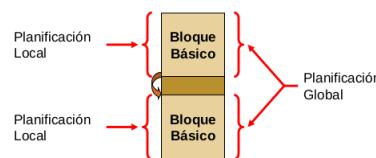
| | Instrucción FX/BR | Instrucción FP | Load/Store | Ciclos |
|------|-------------------|--------------------|--------------|--------|
| loop | subui r1, r1, #8 | nop | ld f0, 0(r1) | 1 |
| | nop | nop | nop | 2 |
| | nop | 1. addd f4, f0, f2 | nop | 3 |
| | nop | 2. nop | nop | 4 |
| | bne r1, loop | 3. nop | nop | 5 |
| | nop | 4. sd f4, 8(r1) | | 6 |

En este caso, la arquitectura VLIW no ganaría nada frente a la opción 2 del bucle sin desenrollar. Además, se necesitarían 18 palabras en el código VLIW frente a las 6 en una arquitectura escalares.

3.3.1 Tipos de planificación Estática

Existen dos tipos de planificación estática:

1. **Planificación local**, que actúa sobre un bloque básico, mediante:
 - Desenrollado de bucles
 - Planificación de las instrucciones del cuerpo aumentado del bucle.
2. **Planificación global**, que actúa considerando bloques de código entre instrucciones de salto.



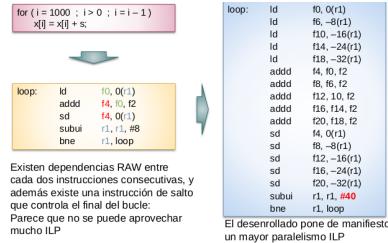
Planificación Estática Local

Vemos cómo actúa la planificación Estática Local en los dos ámbitos descritos anteriormente:

1. Desenrollado de bucles:

Al desenrollar un bucle se crean bloques básicos más largos, que hacen que sea más fácil la planificación local de sus sentencias. Además de disponer de más sentencias, éstas suelen ser independientes, ya que operan sobre diferentes datos.

En el ejemplo anterior tendríamos:



Y analizamos el código ensamblador desde una perspectiva VLIW (separar por unidades funcionales):

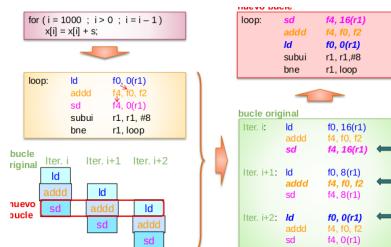
| | Instrucción Entera | Instrucción FP | Load/Store | Ciclo |
|--------|--------------------|----------------|-----------------|--------|
| loop | | | ld r0, 0(r1) | 1 |
| | | | ld r6, 0(r1) | 2 |
| | addd r4, 0, r2 | | ld r10, -16(r1) | 3 |
| | addd r8, 6, r2 | | ld r14, -24(r1) | 4 |
| | addd r12, 10, r2 | | ld r18, -32(r1) | 5 |
| | addd r16, 14, r2 | | sd r4, 0(r1) | 6 |
| | addd r20, 18, r2 | | sd r8, -8(r1) | 7 |
| | subui r1, 1, #40 | | sd r12, 24(r1) | 8 |
| | bne r1, loop | | sd r16, 16(r1) | 9 |
| | | | sd r20, 8(r1) | 10 |
| Slot 1 | | Slot 2 | | Slot 3 |

Se tardarían $10 \times (1000/5) = 2000$ ciclos, frente a los $10 \times 1000 = 10000$ ó $6 \times 1000 = 6000$ ciclos del bucle sin desenrollar.

2. Segmentación software:

Se reorganizan los bucles de forma que cada iteración del código transformado contiene instrucciones tomadas de distintas iteraciones del bloque original. De esta forma se separan las instrucciones dependientes en el bucle original entre diferentes iteraciones del bucle nuevo.

De nuevo, vemos el código anterior bajo esta perspectiva:



Y en este caso:

| | Instrucción Entera | Instrucción FP | Load/Store | Ciclo |
|--------|--------------------|-----------------|---------------|--------|
| loop | | addd r4, 10, r2 | sd r4, 16(r1) | 1 |
| | | | ld r0, 0(r1) | 2 |
| | subui r1, r1, #8 | | | 3 |
| | bne r1, loop | | | 4 |
| | | | | 5 |
| Slot 1 | | Slot 2 | | Slot 3 |

Se tardarían $5 \times 1000 = 5000$ ciclos (algunos más si se consideran las instrucciones previas al inicio del bucle con segmentación software y las posteriores al final del bucle). Si se desenrolla este bucle ya segmentado, se pueden mejorar mucho más las prestaciones.

Y otra implementación podría ser:

| | Instrucción Entera | Instrucción FP | Load/Store | Ciclo |
|------|--------------------|----------------|--------------|-------|
| loop | subui r1,r1,#8 | addd f4,f0,f2 | sd f4,16(r1) | 1 |
| | | | | 2 |
| | bne r1,loop | | ld f0,8(r1) | 3 |
| | | | | 4 |
| | | | | 5 |

Donde se tardarían 4000 ciclos.

3.4 Instrucciones de salto

Ya hemos visto que en VLIW las decisiones de saltar y no saltar se podían modificar mediante instrucciones máquina condicionales que introduce el compilador y que evitan ejecuciones especulativas.