UNIVERSIDAD DE GRANADA

Arquitectura de Computadores

Temario de la asignatura

Ismael Sallami Moreno

Recursos Ingeniería Informática y Ade

Licencia

Arquitectura de Computadores

Ismael Sallami Moreno

https://elblogdeismael.github.io

Índice general

Capítulo 1

Arquitecturas paralelas: clasificación y prestaciones

Tema 1 del programa. Por qué el paralelismo dejó de ser una opción, cómo se clasifican las máquinas que lo explotan y con qué medidas se juzga si el paralelismo está sirviendo de algo.

1.1 Por qué el paralelismo

Durante tres décadas el rendimiento de un procesador creció subiendo la frecuencia y añadiendo etapas al cauce. Ese camino se agotó alrededor de 2005, y por razones físicas, no de ingeniería.

La potencia dinámica de un circuito CMOS es

$$P = \alpha\, C\, V^2 f$$

con α la actividad de conmutación, C la capacidad, V la tensión y f la frecuencia. Subir la frecuencia exige subir la tensión para que las transiciones lleguen a tiempo, y la potencia crece con el **cuadrado** de la tensión. Hasta principios de siglo el escalado de Dennard compensaba: al reducir el tamaño del transistor se reducía también la tensión, y la densidad de potencia se mantenía. Cuando las corrientes de fuga crecieron, la tensión dejó de bajar y el escalado se rompió.

Tres muros, y los tres siguen en pie:

Muro	Qué impide
Potencia	disipar el calor de un chip a frecuencia mayor
Memoria	la latencia de la DRAM mejora mucho más despacio que el procesador
Paralelismo a nivel de instrucción	el paralelismo que un flujo secuencial ofrece está casi agotado

La respuesta de la industria fue la misma en todos los fabricantes: en vez de un procesador más rápido, varios procesadores. El paralelismo dejó de ser un asunto de supercomputación para pasar a ser la única forma de aprovechar el hardware, y eso traslada el problema al software.

1.2 Clasificación de Flynn

La clasificación clásica, por el número de flujos de instrucciones y de datos:

Clase	Instrucciones	Datos	Qué es
SISD	uno	uno	el procesador secuencial clásico
SIMD	uno	varios	una instrucción sobre muchos datos: vectorial, GPU
MISD	varios	uno	sin realizaciones comerciales; se cita por completitud
MIMD	varios	varios	multiprocesadores y multicomputadores

MIMD es la categoría dominante, y por sí sola no dice casi nada. Lo que la hace útil es subdividirla por el modelo de memoria.

1.2.1 Multiprocesadores y multicomputadores

	Multiprocesador	Multicomputador
Memoria	única y compartida	privada por nodo
Comunicación	escribir y leer variables	paso de mensajes
Sincronización	explícita, con cerrojos y barreras	implícita en el mensaje
Escalabilidad	limitada por la memoria compartida	alta
Programación	más sencilla, y con más errores sutiles	más laboriosa, y más explícita
Ejemplo	un procesador multinúcleo	un clúster

La diferencia no es de rendimiento sino de modelo mental. En memoria compartida la comunicación es invisible, y por eso las carreras de datos son fáciles de introducir; con paso de mensajes toda comunicación se escribe, y lo que se olvida produce un bloqueo, no un resultado incorrecto silencioso.

1.2.2 Modelos de acceso a memoria

Dentro de los multiprocesadores:

Modelo	Acceso	Consecuencia
UMA	uniforme: todos los procesadores tardan lo mismo	simple; no escala más allá de unas decenas
NUMA	no uniforme: la memoria local es más rápida que la remota	escala; obliga a colocar los datos donde se usan
COMA	la memoria actúa como caché y los datos migran	poco usado

NUMA es lo que hay en cualquier servidor de dos o más sockets, y también dentro de un solo chip con muchos núcleos. La consecuencia práctica es la política de **primer toque**: la página se asigna

en el nodo del hilo que la escribe primero, así que inicializar un vector entero desde un solo hilo deja toda la memoria en un nodo y estrangula al resto.

1.3 Prestaciones

1.3.1 Ganancia y eficiencia

Con T_1 el tiempo del mejor algoritmo secuencial y T_p el tiempo con p procesadores:

$$S(p) = \frac{T_1}{T_p} \qquad E(p) = \frac{S(p)}{p}$$

La ganancia ideal es p y la eficiencia ideal 1. En la práctica la eficiencia cae al aumentar p, porque las tres fuentes de pérdida crecen: el trabajo que no se puede paralelizar, la comunicación y la sincronización, y el desequilibrio de carga.

Una advertencia sobre el numerador: T_1 tiene que ser el tiempo del **mejor algoritmo secuencial**, no el del programa paralelo ejecutado con un procesador. Usar lo segundo produce ganancias infladas, porque el programa paralelo arrastra sobrecarga que el secuencial no tiene. Es la forma más común de presentar resultados engañosos en esta materia.

La **ganancia superlineal**, $S(p) > p$, existe y no viola nada: al repartir el problema entre más procesadores, la parte que toca a cada uno puede caber en su caché. Lo que ha mejorado no es el paralelismo sino la jerarquía de memoria.

1.3.2 Ley de Amdahl

Si una fracción f del tiempo es inherentemente secuencial:

$$S(p) = \frac{1}{f + \dfrac{1-f}{p}} \xrightarrow[p\to\infty]{} \frac{1}{f}$$

Con un 5 % secuencial, la ganancia máxima es 20 **por muchos procesadores que se añadan**. Es un resultado severo, y durante años se usó como argumento contra el paralelismo masivo.

1.3.3 Ley de Gustafson

Amdahl supone que el problema tiene tamaño fijo. Gustafson observó que en la práctica nadie compra una máquina mayor para resolver el mismo problema más rápido: la compra para resolver un problema mayor. Con el **tiempo** fijo en vez del tamaño:

$$S(p) = p - f\,(p-1)$$

que crece linealmente con p. Las dos leyes no se contradicen: responden a preguntas distintas. Amdahl mide **escalabilidad fuerte**, con el problema fijo; Gustafson, **escalabilidad débil**, con el trabajo por procesador fijo.

Distinguirlas es lo que permite leer una gráfica de escalabilidad: si el tamaño del problema crece con el número de procesadores, la curva mide otra cosa.

1.3.4 Otras medidas

- **Escalabilidad.** Un sistema es escalable si mantiene la eficiencia al crecer a la vez el número de procesadores y el tamaño del problema.
- **Granularidad.** La relación entre cómputo y comunicación. Gruesa cuando cada tarea hace mucho trabajo entre comunicaciones; fina en el caso contrario. La arquitectura determina qué granularidad compensa: en memoria compartida se puede bajar mucho más que en un clúster.
- **FLOPS y su límite.** Contar operaciones en coma flotante por segundo ignora si la máquina puede alimentarlas con datos. El **modelo del techo** cruza el pico aritmético con el ancho de banda de memoria, y sitúa cada núcleo de cómputo según su intensidad operacional —operaciones por byte leído—. Es lo que dice si un programa está limitado por cálculo o por memoria, que es la primera pregunta antes de optimizar.

1.4 Redes de interconexión

Lo que conecta procesadores y memoria. Se describen con cuatro parámetros:

Parámetro	Definición
Grado	enlaces por nodo
Diámetro	distancia máxima entre dos nodos
Ancho de bisección	enlaces que hay que cortar para partir la red en dos
Coste	número de enlaces

Topología	Diámetro	Grado	Comentario
Bus	1	1	no escala: el ancho de banda se reparte
Anillo	$p/2$	2	barato y con diámetro grande
Malla 2D	$2(\sqrt{p}-1)$	4	buena relación entre coste y diámetro
Toro	$\sqrt{p}$	4	malla con los extremos unidos
Hipercubo	$\log_2 p$	$\log_2 p$	diámetro mínimo, grado creciente
Crossbar	1	—	sin bloqueo y con coste $O(p^2)$

El ancho de bisección es el parámetro que predice el comportamiento con comunicaciones globales, como una transposición o una transformada. Un bus tiene bisección 1 y por eso una máquina con bus no pasa de unas pocas unidades.

1.5 Coherencia y consistencia

Dos problemas que se confunden y son distintos.

Coherencia responde a qué valor devuelve una lectura de **una** posición. Cada procesador tiene caché, así que puede haber varias copias del mismo dato; si uno escribe, los demás deben dejar de ver el valor viejo.

Consistencia responde a en qué orden se hacen visibles las escrituras a posiciones **distintas**. Es un contrato entre el hardware y el programador, y determina qué reordenamientos puede hacer la máquina.

1.5.1 Protocolos de coherencia

Dos familias:

- **Espionaje.** Todos los controladores de caché vigilan un medio común y reaccionan a lo que ven. Requiere difusión, así que sirve con pocos núcleos.
- **Basados en directorio.** Una estructura registra qué cachés tienen cada bloque, y las invalidaciones se envían solo a esas. Escala, y cuesta memoria para el directorio.

El protocolo habitual es **MESI**, con cuatro estados por línea:

Estado	Significado	Copias en otras cachés
Modificada	modificada aquí, memoria desactualizada	ninguna
Exclusiva	igual que memoria, solo aquí	ninguna
Compartida	igual que memoria	puede haber
Inválida	no utilizable	—

El estado exclusivo evita una transacción de bus muy frecuente: si un procesador lee un bloque que nadie más tiene y luego escribe, no hace falta invalidar a nadie. Las variantes MOESI y MESIF añaden estados para que una caché sirva el dato a otra sin pasar por memoria.

Y el efecto que hay que conocer para programar: el **falso compartimiento**. Dos hilos escriben variables distintas que caen en la misma línea de caché. No comparten ningún dato, y aun así la línea viaja de un núcleo a otro en cada escritura y el programa se frena hasta varias veces. Se corrige separando las variables o rellenando hasta el tamaño de línea.

1.5.2 Modelos de consistencia

Modelo	Qué garantiza	Coste
Secuencial	el resultado es el de alguna intercalación de los programas, respetando el orden de cada uno	alto: impide casi toda reordenación
Consistencia total de almacenamientos	permite adelantar lecturas a escrituras pendientes	el de x86
Relajados	permiten más reordenaciones y exigen barreras explícitas	el de ARM y RISC-V

La consistencia secuencial es lo que un programador supone sin darse cuenta, y **ninguna máquina de propósito general la ofrece**, porque impide el búfer de escritura y la ejecución fuera de orden.

De ahí que un programa con carreras de datos pueda comportarse distinto en x86 y en ARM. Los lenguajes resuelven esto con su propio modelo de memoria —el de C11 y C++11— y con

operaciones atómicas que se traducen a las barreras que cada arquitectura necesite. La regla práctica: un programa sin carreras se comporta como si la consistencia fuera secuencial, y un programa con carreras no tiene comportamiento definido. Los dos problemas se desarrollan en Ortega Lopera et al., 2005 y en Hennessy et al., 2026.

Capítulo 2

Programación paralela

Tema 2 del programa. Cómo se descompone un problema, qué ofrecen los dos modelos de programación y qué errores aparecen que no existen en un programa secuencial.

2.1 Del problema al programa

Cuatro pasos, en este orden:

1. **Descomposición.** Partir el trabajo en tareas.
2. **Asignación.** Repartir las tareas entre procesos o hilos.
3. **Orquestación.** Comunicar y sincronizar.
4. **Correspondencia.** Situar los hilos en los procesadores físicos.

Los dos primeros determinan cuánto paralelismo hay; los dos últimos, cuánto se aprovecha.

2.1.1 Formas de descomponer

Descomposición	Se reparte	Ejemplo
De dominio	los datos	cada hilo procesa un trozo de la matriz
Funcional	las funciones	un hilo lee, otro comprime, otro escribe
Recursiva	el árbol de llamadas	divide y vencerás
Especulativa	se ejecutan alternativas antes de saber cuál vale	búsqueda con poda

La de dominio es la que más escala, porque el número de tareas crece con el tamaño del problema. La funcional está acotada por el número de funciones, y por eso sirve como complemento y no como estrategia principal.

2.1.2 Equilibrado de carga

Repartir por igual el número de tareas no reparte por igual el trabajo si las tareas cuestan distinto. Dos enfoques:

- **Estático.** El reparto se decide antes de ejecutar. Sin sobrecarga, y solo vale si el coste de cada tarea es previsible.
- **Dinámico.** Los hilos toman trabajo de una cola común según terminan. Se adapta al desequilibrio y cuesta sincronización sobre la cola.

El caso que lo ilustra es un bucle en el que el trabajo por iteración crece con el índice, como el

cálculo del triángulo inferior de una matriz. Un reparto en bloques contiguos deja al último hilo con mucho más trabajo; un reparto cíclico lo equilibra sin coste adicional.

El **robo de trabajo** es la forma refinada del reparto dinámico: cada hilo tiene su propia cola y, cuando se queda sin trabajo, roba del extremo opuesto de la cola de otro. Reduce la contención porque en el caso común nadie toca la cola ajena.

2.2 Memoria compartida: OpenMP

El modelo dominante dentro de un nodo. El programa arranca con un hilo y crea equipos de hilos en las regiones marcadas con directivas, sin reescribir la estructura del programa.

```
#pragma omp parallel for reduction(+:suma) schedule(static)
for (int i = 0; i < n; i++) {
    suma += v[i] * w[i];
}
```

2.2.1 Alcance de las variables

Es la fuente principal de errores, y la única decisión que el compilador no puede tomar:

Cláusula	Efecto
`shared`	una sola copia, visible a todos. Por omisión para las de fuera
`private`	una copia por hilo, **sin inicializar**
`firstprivate`	copia por hilo, inicializada con el valor de entrada
`lastprivate`	copia por hilo; al salir se conserva la de la última iteración
`reduction(op:var)`	copia privada por hilo y combinación final con `op`

`private` no inicializa. Un acumulador declarado `private` empieza con basura, y el programa da resultados distintos en cada ejecución sin fallar. `reduction` es lo que hay que usar para acumular, y además evita la carrera sobre la variable compartida.

Escribir `default(none)` obliga a declarar el alcance de cada variable. Es más trabajo y convierte el error silencioso en un error de compilación.

2.2.2 Reparto de iteraciones

Planificación	Reparto	Cuándo
`static`	bloques fijos, decididos antes	iteraciones de coste uniforme
`dynamic`	bloques bajo demanda	coste variable e impredecible
`guided`	bloques que decrecen	compromiso entre las dos
`auto`, `runtime`	lo decide el compilador o una variable de entorno	para experimentar

El tamaño de bloque importa por dos razones opuestas: pequeño equilibra mejor y aumenta la

sincronización sobre el contador compartido; grande hace lo contrario. Y con `static` y bloque 1 el reparto es cíclico, que es la solución al bucle triangular de más arriba.

2.2.3 Sincronización

Construcción	Qué hace
`barrier`	ningún hilo pasa hasta que llegan todos
`critical`	sección de exclusión mutua
`atomic`	actualización atómica de una variable; más barato que `critical`
`single`	lo ejecuta un hilo; los demás esperan al final
`master`	lo ejecuta el hilo 0, **sin barrera**
`nowait`	suprime la barrera implícita del final de un bucle

Las construcciones de reparto llevan barrera implícita al final. `nowait` la quita cuando el resultado no se necesita todavía, y es una de las optimizaciones más rentables cuando hay varios bucles seguidos independientes.

La diferencia entre `single` y `master` se olvida constantemente: `single` tiene barrera y `master` no. Usar `master` donde hacía falta `single` deja a los demás hilos avanzando sobre datos que aún no están.

2.2.4 Tareas

Para paralelismo irregular, donde el número de unidades de trabajo no se conoce de antemano:

```
#pragma omp parallel
#pragma omp single
{
    recorrer(raiz);          /* genera tareas al descender */
}

void recorrer(nodo *n) {
    if (!n) return;
    #pragma omp task
    recorrer(n->izq);
    #pragma omp task
    recorrer(n->der);
    procesar(n);
}
```

El patrón `parallel` + `single` es obligatorio: el equipo se crea con `parallel` y un solo hilo genera las tareas, que los demás ejecutan. Sin `single`, cada hilo generaría el árbol entero.

2.3 Paso de mensajes: MPI

El modelo para multicomputadores. Todos los procesos ejecutan el mismo programa y se distinguen por su identificador dentro del comunicador.

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &yo);
MPI_Comm_size(MPI_COMM_WORLD, &n);

if (yo == 0) {
    MPI_Send(datos, cuenta, MPI_DOUBLE, 1, ETIQUETA, MPI_COMM_WORLD);
} else if (yo == 1) {
    MPI_Recv(datos, cuenta, MPI_DOUBLE, 0, ETIQUETA, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
MPI_Finalize();
```

Clase	Operaciones	Comentario
Punto a punto bloqueante	`MPI_Send`, `MPI_Recv`	`MPI_Send` puede volver antes de que se reciba, según el tamaño
Punto a punto no bloqueante	`MPI_Isend`, `MPI_Irecv`, `MPI_Wait`	permiten solapar cómputo y comunicación
Colectivas	`MPI_Bcast`, `MPI_Scatter`, `MPI_Gather`, `MPI_Reduce`, `MPI_Allreduce`	implementadas con algoritmos en árbol; mejores que hacerlas a mano
Sincronización	`MPI_Barrier`	rara vez necesaria si las colectivas ya sincronizan

Dos trampas concretas:

- **`MPI_Send` bloqueante no garantiza que haya un receptor.** Con mensajes pequeños el sistema los copia a un búfer y devuelve; con mensajes grandes espera. Un programa en el que dos procesos se envían mutuamente antes de recibir funciona con datos pequeños y **se bloquea al crecer el tamaño**. La solución es `MPI_Sendrecv` o las versiones no bloqueantes.
- **Las colectivas las tienen que ejecutar todos los procesos del comunicador.** Una colectiva dentro de un `if` que solo cumple un proceso bloquea a todos.

2.3.1 Modelo híbrido

En un clúster de nodos multinúcleo lo natural es combinar: MPI entre nodos y OpenMP dentro de cada uno. Reduce el número de procesos MPI, y con él la memoria de los búferes de comunicación y el número de mensajes. Su dificultad es que el soporte de hilos de MPI hay que pedirlo explícitamente y no todos los niveles están disponibles en toda implementación.

2.4 Errores propios del paralelismo

2.4.1 Carrera de datos

Dos hilos acceden a la misma posición sin sincronización y al menos uno escribe. El resultado depende del orden real de ejecución.

```
/* incorrecto */
#pragma omp parallel for
for (int i = 0; i < n; i++) suma += v[i];
```

`suma += v[i]` no es una operación: es leer, sumar y escribir. Dos hilos pueden leer el mismo valor y una de las dos actualizaciones se pierde. El resultado suele ser **casi** correcto, que es lo que hace difícil detectarlo: un error del 0,1 % pasa por ruido numérico.

Se corrige con `reduction(+:suma)`, no con `critical`, que serializaría el bucle entero.

2.4.2 Interbloqueo

Dos hilos esperan cada uno un recurso que el otro tiene. Las cuatro condiciones de Coffman se dan a la vez: exclusión mutua, retención y espera, ausencia de expropiación y espera circular. Romper cualquiera lo evita, y la más práctica es la última: **tomar siempre los cerrojos en el mismo orden global**.

2.4.3 Inanición y convoy

Un hilo no llega a progresar porque otros se le adelantan siempre, o todos se alinean detrás del mismo cerrojo y el programa avanza al ritmo de la sección crítica. Los dos se atacan reduciendo el tamaño de la sección crítica y sustituyendo el cerrojo global por varios de grano fino.

2.4.4 Sobrecarga

El paralelismo cuesta: crear hilos, sincronizar, comunicar. Un bucle de cien iteraciones triviales paralelizado es **más lento** que el secuencial. La cláusula `if` de OpenMP permite decidir en ejecución:

```
#pragma omp parallel for if(n > 10000)
```

2.4.5 Cómo se detectan

Herramienta	Qué encuentra
`valgrind --tool=helgrind`	carreras y errores de uso de cerrojos
`-fsanitize=thread`	carreras, con mucho menos sobrecoste
`perf stat`	ciclos, fallos de caché, instrucciones
`perf c2c`	falso compartimiento y contención de líneas de caché

Las dos primeras encuentran lo que las pruebas no: una carrera puede no manifestarse en mil ejecuciones y aparecer en la máquina del profesor. Ejecutar un programa paralelo y ver que da el resultado correcto **no es una comprobación**.

2.5 Patrones

Un repertorio pequeño cubre la mayor parte de los casos:

Patrón	Estructura	Ejemplo
Paralelismo de datos	la misma operación sobre muchos elementos	producto escalar
Descomposición geométrica	el dominio se divide y los bordes se intercambian	diferencias finitas
Divide y vencerás	recursión con tareas	ordenación por mezcla

Patrón	Estructura	Ejemplo
Cauce	etapas encadenadas, cada una en un hilo	procesado de flujo
Maestro y trabajadores	uno reparte, los demás ejecutan	trabajo irregular
Reducción	combinar resultados parciales con un operador asociativo	suma, máximo

La reducción exige que el operador sea asociativo, y **la suma en coma flotante no lo es**. Un mismo programa con distinto número de hilos puede dar resultados que difieren en los últimos dígitos. No es un error: es el orden de las operaciones. Cuando la reproducibilidad importa, hay que fijar el orden de combinación aunque cueste rendimiento. El desarrollo completo de OpenMP está en Chapman et al., 2008, y los patrones y su análisis en Rauber y Rünger, 2023 y Wilkinson y Allen, 2005.

Capítulo 3

Arquitecturas con paralelismo a nivel de hebra

Tema 3 del programa. Las máquinas que ejecutan varios flujos de control a la vez: multihilo dentro de un núcleo, multinúcleo dentro de un chip y multiprocesador dentro de una máquina.

3.1 De dónde sale el paralelismo de hebra

El paralelismo a nivel de instrucción del tema 4 extrae concurrencia de **un** flujo secuencial, y ahí hay un límite: las dependencias entre instrucciones próximas. Duplicar unidades funcionales deja de servir cuando no hay instrucciones independientes que ejecutar.

El paralelismo a nivel de hebra viene de otro sitio: de flujos que **por construcción** son independientes. Un programa con varios hilos, varios programas a la vez, o las peticiones concurrentes de un servidor. Es paralelismo explícito, y por eso el trabajo se traslada al programador.

3.2 Multihilo dentro de un núcleo

La idea: cuando un hilo se detiene —un fallo de caché, una dependencia larga—, las unidades funcionales quedan ociosas. Si el núcleo guarda el estado de varios hilos, puede ejecutar otro mientras tanto.

Lo que se duplica es el **estado arquitectónico**: contador de programa, banco de registros, registro de estado y tablas de traducción. Lo que se comparte son las unidades funcionales, las cachés y los predictores.

Variante	Cuándo cambia de hilo	Coste de un cambio
De grano grueso	ante una detención larga	algunos ciclos de vaciado
De grano fino	cada ciclo, por turno	ninguno
Simultáneo (SMT)	no cambia: emite instrucciones de varios hilos **en el mismo ciclo**	ninguno

El multihilo simultáneo es el que se implementa hoy, con dos hilos por núcleo en los procesadores de propósito general y hasta ocho en algunos servidores.

Tres consecuencias que conviene tener claras:

– **Dos hilos hardware no son dos núcleos.** Comparten unidades funcionales y cachés. La ganancia típica está entre el 15 % y el 30 %, no en el 100 %, y desaparece cuando los dos hilos compiten por el mismo recurso.

- **Puede empeorar.** Dos hilos con conjuntos de trabajo grandes se expulsan mutuamente de la caché. En cómputo intensivo con buena localidad, desactivar el SMT a veces mejora.
- **Es un canal lateral.** Compartir cachés y predictores permite que un hilo deduzca lo que hace el otro. Es la razón por la que en algunos entornos se desactiva por seguridad y por la que los planificadores evitan emparejar procesos de dominios distintos.

3.3 Multinúcleo

Varios núcleos completos en el mismo chip. Cada uno con su cauce, sus registros y su primer nivel de caché; los niveles superiores, compartidos.

3.3.1 Jerarquía

Nivel	Compartición típica	Latencia
L1 de instrucciones y datos	privado por núcleo	4 ciclos
L2	privado o por pareja	12 a 20 ciclos
L3	compartido por todo el chip	30 a 50 ciclos
Memoria principal	compartida	200 a 300 ciclos

El nivel compartido cumple dos funciones: aprovechar la capacidad cuando un núcleo la necesita más que otro, y servir de punto de coherencia. La contrapartida es la interferencia: un núcleo con un recorrido de memoria grande expulsa los datos de los demás.

3.3.2 Conexión interna

Con pocos núcleos basta un bus o un cruce de barras. Al crecer, ninguno de los dos escala, y se pasa a **anillo** —Intel durante años— o a **malla**, que es lo habitual con muchos núcleos. La consecuencia es que la latencia al nivel compartido deja de ser uniforme: depende de dónde esté el núcleo y dónde el segmento de caché.

3.3.3 Núcleos heterogéneos

Los diseños actuales combinan núcleos grandes, orientados a rendimiento monohilo, con núcleos pequeños y eficientes. La misma ISA, microarquitecturas distintas.

El problema se traslada al sistema operativo: colocar mal un hilo crítico en un núcleo pequeño arruina el rendimiento, y el planificador necesita información del hardware sobre qué está haciendo cada hilo. Es un caso claro de decisión que ya no se puede tomar solo en software.

3.4 Multiprocesadores

Varios chips en la misma máquina, con memoria compartida. En la práctica siempre NUMA: cada socket tiene sus canales de memoria, y acceder a la memoria del otro socket cuesta bastante más.

3.4.1 Consecuencias de NUMA

- **Política de primer toque.** La página se asigna en el nodo del hilo que la toca primero. Inicializar un vector grande desde un solo hilo deja toda la memoria en un nodo y estrangula

el ancho de banda. Se corrige inicializando en paralelo con el mismo reparto que luego usa el cálculo.
- **Afinidad.** Fijar los hilos a núcleos concretos evita que el planificador los mueva y pierdan su caché y su localidad de memoria. Se controla con `OMP_PLACES` y `OMP_PROC_BIND`, o con `numactl` desde fuera.
- **Medir antes de suponer.** `numactl --hardware` da las distancias entre nodos, y `numastat` cuenta los accesos remotos.

3.4.2 Coherencia a escala

Con muchos núcleos, el espionaje del bus deja de servir: la difusión de cada invalidación satura la red. Se pasa a **directorio**, que registra qué cachés tienen cada bloque y envía las invalidaciones solo a esas.

El coste es memoria: un directorio completo necesita un bit por caché y bloque, lo que crece con el cuadrado del sistema. De ahí los directorios dispersos, que guardan información solo de los bloques compartidos, y los jerárquicos.

3.5 Sincronización

La coherencia garantiza que las lecturas ven el último valor escrito. No garantiza atomicidad: un incremento sigue siendo leer, sumar y escribir.

3.5.1 Instrucciones atómicas

Instrucción	Semántica
`test-and-set`	escribe 1 y devuelve el valor anterior
`compare-and-swap`	si el valor es el esperado, lo sustituye; devuelve si tuvo éxito
`fetch-and-add`	suma y devuelve el valor anterior
`load-linked` / `store-conditional`	la escritura solo tiene efecto si nadie tocó la posición entre las dos

`compare-and-swap` es la primitiva universal: con ella se construye cualquier estructura sin cerrojos. La pareja de la última fila es la alternativa de las arquitecturas RISC, y tiene la ventaja de no bloquear la línea de caché durante la operación.

3.5.2 Cerrojos

Un cerrojo ingenuo con `test-and-set` en bucle es correcto y muy malo: cada intento escribe, y cada escritura invalida la línea en todos los núcleos que esperan.

El **cerrojo con espera de lectura** lo corrige: se lee en bucle hasta ver el cerrojo libre, y solo entonces se intenta la operación atómica. Mientras se lee, la línea está compartida y no genera tráfico.

Cerrojo	Propiedad
`test-and-set`	simple, mucho tráfico de coherencia
Con espera de lectura	mucho menos tráfico

Cerrojo	Propiedad
Con retroceso exponencial	reduce la contención al fallar
De turno (*ticket*)	justo: se atiende en orden de llegada
En cola (MCS)	cada hilo espera en su propia variable: sin tráfico entre núcleos

El cerrojo MCS es la respuesta correcta con muchos núcleos, porque convierte la espera en local. Es también la base del cerrojo con cola que usa el núcleo de Linux.

Y una decisión de política: **girar o dormir**. Girar desperdicia CPU pero evita el coste de un cambio de contexto; dormir lo contrario. Los cerrojos adaptativos giran un tiempo acotado y después duermen, que es lo que hacen las implementaciones de biblioteca.

3.5.3 Barreras

Ningún hilo pasa hasta que llegan todos. La implementación ingenua con un contador atómico serializa a todos sobre la misma línea; la **barrera en árbol** combina en $\log p$ pasos y escala.

Las barreras son caras y muchas son innecesarias. Quitar las implícitas con `nowait` cuando el resultado no se necesita todavía es una de las optimizaciones más rentables en OpenMP.

3.5.4 Estructuras sin cerrojos

Construidas sobre `compare-and-swap`, garantizan progreso sin exclusión mutua. Son más rápidas bajo contención alta y mucho más difíciles de escribir correctamente.

El **problema ABA** es su trampa característica: un valor cambia de A a B y vuelve a A entre la lectura y el `compare-and-swap`, que tiene éxito aunque el estado intermedio invalidara la operación. Se resuelve con contadores de versión junto al puntero, o con `load-linked` y `store-conditional`, que detectan la escritura intermedia aunque el valor coincida.

La regla práctica: usar las estructuras concurrentes de biblioteca y no escribir las propias salvo necesidad demostrada. El tratamiento de estas arquitecturas y de sus mecanismos de sincronización está en Anguita López y Ortega Lopera, 2016 y en Hennessy et al., 2026.

Capítulo 4

Arquitecturas con paralelismo a nivel de instrucción

Tema 4 del programa. Cómo se extrae paralelismo de un único flujo de instrucciones, qué lo limita y qué pueden hacer el compilador y el programador para ayudar.

4.1 El punto de partida

El cauce segmentado solapa instrucciones y su cota es un CPI de 1: una instrucción terminada por ciclo. Para bajar de ahí hay que **emitir varias instrucciones por ciclo**, y esa es la definición de una máquina superescalar. La medida deja de ser el CPI y pasa a ser su inverso, el IPC.

Lo que lo impide son las dependencias.

4.2 Dependencias

Tipo	Nombre	Descripción	¿Es real?
RAW	dependencia verdadera	una instrucción usa lo que otra produce	sí
WAR	antidependencia	una escribe donde otra aún no ha leído	no: solo comparten el nombre
WAW	dependencia de salida	dos escriben el mismo registro	no

Solo la primera transmite información. Las otras dos son artefactos de que el número de registros es finito, y de ahí que se llamen falsas.

```
addq    %rax, %rbx        # 1
movq    %rbx, %rcx        # 2  RAW con 1: real
movq    $5, %rax          # 3  WAR con 1: falsa
addq    %rdx, %rcx        # 4  WAW con 2: falsa
```

4.2.1 Renombrado de registros

El hardware mantiene muchos más registros físicos que arquitectónicos y asocia cada escritura a uno nuevo. Las instrucciones 3 y 4 del ejemplo escriben en registros físicos distintos, así que dejan de tener conflicto con las anteriores.

Con el renombrado desaparecen WAR y WAW por completo, y solo quedan las dependencias verdaderas. Es el mecanismo que hace posible la ejecución fuera de orden.

4.2.2 Dependencias de memoria

Más difíciles que las de registro, porque el nombre no está en la instrucción sino en la dirección, que se calcula. Saber si dos accesos se refieren al mismo sitio —la **desambiguación de memoria**— exige comparar direcciones ya calculadas.

Los procesadores lo resuelven especulando: suponen que una carga no depende de un almacenamiento pendiente y ejecutan; si luego resulta que sí, deshacen. El predictor de dependencias de memoria es una pieza más del hardware especulativo.

4.3 Ejecución fuera de orden

Las instrucciones se ejecutan cuando sus operandos están listos, no en el orden del programa. El esquema:

Etapa	Orden
Búsqueda y decodificación	en orden
Renombrado y emisión a las estaciones de reserva	en orden
Ejecución	**fuera de orden**, según disponibilidad
Escritura del resultado	fuera de orden
Retirada	**en orden**, desde el búfer de reordenación

El **búfer de reordenación** es la pieza que hace compatible el desorden con la corrección: los resultados no son definitivos hasta que la instrucción se retira, y las retiradas ocurren en orden de programa. Eso da dos cosas a la vez: excepciones precisas y la posibilidad de descartar el trabajo especulativo.

El algoritmo de Tomasulo, de 1967, ya contenía el renombrado y las estaciones de reserva; el búfer de reordenación se añadió después precisamente para las excepciones precisas.

4.4 Especulación

Un salto tarda ciclos en resolverse. En vez de esperar, el procesador predice y sigue ejecutando por el camino predicho. Si acierta, no ha perdido nada; si falla, descarta lo especulado y vuelve.

Con cauces de veinte etapas y capacidad para cientos de instrucciones en vuelo, la penalización de un fallo de predicción está en 15 a 20 ciclos. Con un acierto del 95 % y saltos cada cinco instrucciones, eso son unos 0,2 ciclos perdidos por instrucción, que sobre un IPC de 4 es una pérdida notable. De ahí que los predictores actuales superen el 99 % en código regular.

Y de ahí también el efecto lateral que se descubrió en 2018: **la ejecución especulativa deja huella en la caché aunque el resultado se descarte**. Meltdown y Spectre miden esa huella para deducir datos que el programa nunca llegó a leer arquitectónicamente. Las mitigaciones cuestan rendimiento en cada transición al núcleo, y son la razón por la que el paso 5 de la tabla —retirada en orden— no basta para aislar.

4.5 VLIW

La alternativa a resolverlo todo en hardware: que sea el **compilador** quien agrupe instrucciones independientes en una palabra larga, y el hardware las emita sin comprobar nada.

	Superescalar	VLIW
Quién planifica	el hardware, en ejecución	el compilador, al compilar
Complejidad del hardware	alta	baja
Información disponible	latencias reales, incluidos los fallos de caché	solo la estática
Compatibilidad binaria	se mantiene entre generaciones	se rompe al cambiar el número de unidades
Consumo	mayor	menor

VLIW fracasó en el mercado de propósito general —Itanium es el caso conocido— por dos razones. La primera es que el compilador no puede saber si un acceso fallará en caché, y esa es justo la latencia que más varía. La segunda es la compatibilidad: el código se compila para un número concreto de unidades funcionales, así que cambiar el diseño obliga a recompilar todo.

Donde sí sobrevive es en procesadores de señal y aceleradores, donde el código se recompila con el hardware y los patrones de acceso son predecibles.

4.6 Técnicas del compilador

Lo que el compilador hace para aumentar el paralelismo disponible.

4.6.1 Planificación de instrucciones

Reordenar para separar una instrucción de la que depende de ella, y llenar el hueco con trabajo independiente. Es lo que evita las burbujas del riesgo *load-use*.

4.6.2 Desenrollado de bucles

Replicar el cuerpo del bucle reduce el número de saltos y, sobre todo, expone instrucciones independientes que el planificador puede entrelazar:

```
/* original */
for (int i = 0; i < n; i++) s += v[i];

/* desenrollado por cuatro, con acumuladores independientes */
double s0=0, s1=0, s2=0, s3=0;
int i;
for (i = 0; i + 3 < n; i += 4) {
    s0 += v[i];   s1 += v[i+1];
    s2 += v[i+2]; s3 += v[i+3];
}
for (; i < n; i++) s0 += v[i];
double s = (s0 + s1) + (s2 + s3);
```

Los cuatro acumuladores son lo esencial, y no el desenrollado en sí. Con un único acumulador, cada suma depende de la anterior y la cadena de dependencias impone la latencia de la suma en coma flotante —cuatro o cinco ciclos— por elemento. Con cuatro cadenas independientes, la unidad segmentada trabaja a pleno rendimiento.

Y el precio, que hay que decir: **el resultado cambia**, porque la suma en coma flotante no es asociativa. Por eso el compilador no lo hace solo salvo que se le autorice con `-ffast-math`, y por

eso esa opción no debe activarse a la ligera.

4.6.3 Segmentación software

Solapar iteraciones distintas del bucle: mientras se calcula la iteración i, se cargan los datos de la $i+1$ y se escriben los de la $i-1$. Consigue lo que el desenrollado, con menos código.

4.6.4 Predicación

Sustituir un salto por instrucciones condicionales que se ejecutan siempre y solo tienen efecto si se cumple la condición. En x86 es `cmov`. Elimina el riesgo de control, y a cambio ejecuta las dos ramas: solo compensa si la predicción del salto sería mala y las ramas son cortas.

4.6.5 Vectorización automática

El compilador convierte un bucle en instrucciones SIMD. Se comprueba con `gcc -O3 -fopt-info-vec` o `-fopt-info-vec-missed`, que dice **por qué** no vectorizó. Las causas habituales:

Obstáculo	Cómo se resuelve
Posible solapamiento entre punteros	`restrict`
Dependencia entre iteraciones	reescribir el bucle
Datos no alineados	atributos de alineamiento
Llamadas a función en el cuerpo	poner en línea
Salidas condicionales del bucle	reestructurar

4.7 Qué puede hacer el programador

El tema tiene una parte práctica clara: el código que se escribe determina cuánto ILP hay disponible.

- **Romper las cadenas de dependencias** con acumuladores múltiples, como arriba.
- **Evitar los saltos impredecibles.** Un `if` cuyo resultado alterna al azar cuesta la penalización completa cada vez. Ordenar los datos antes de recorrerlos puede hacer que un bucle con `if` se vuelva varias veces más rápido, sin cambiar una línea del bucle.
- **Marcar con `restrict`** los punteros que no se solapan, para que el compilador pueda reordenar los accesos.
- **Preferir vectores de estructuras o estructuras de vectores** según el patrón de acceso: recorrer un solo campo de un millón de estructuras trae a la caché todos los demás campos, y esa es una pérdida directa de ancho de banda.
- **Poner en línea** las funciones pequeñas del bucle interno, o el compilador no puede planificar a través de la llamada.
- **Medir.** `perf stat` da ciclos, instrucciones, IPC y fallos de predicción; un cambio que reduce instrucciones y baja el IPC puede ser peor.

La regla que resume el tema: el hardware ya explota todo el paralelismo que encuentra, así que la aportación del programador es **quitar los obstáculos** que le impiden encontrarlo, no intentar planificar en su lugar. El catálogo completo de estas técnicas está en Gerber et al., 2006 y en Fog, 2004, y su análisis cuantitativo en Hennessy et al., 2026.

Capítulo 5

Arquitecturas con paralelismo de datos

Tema 5 del programa. Las máquinas que aplican la misma operación a muchos datos a la vez: extensiones vectoriales, procesadores vectoriales y unidades gráficas de propósito general.

5.1 La idea

Muchos cálculos aplican la misma operación a todos los elementos de un vector. Ejecutarlos con instrucciones escalares desperdicia trabajo: cada elemento paga la búsqueda y la decodificación de su propia instrucción.

Una instrucción SIMD hace lo mismo con un solo trabajo de control. La ganancia es doble: menos instrucciones que buscar y decodificar, y unidades funcionales replicadas que operan en paralelo.

El precio es la rigidez. Todos los elementos hacen lo mismo, así que un dato que requiera un tratamiento distinto rompe el esquema, y ahí es donde aparecen las máscaras y la divergencia.

5.2 Extensiones SIMD

Registros anchos que se interpretan como varios elementos:

Extensión	Anchura	Elementos `double`	Elementos `float`
SSE	128 bits	2	4
AVX, AVX2	256 bits	4	8
AVX-512	512 bits	8	16
NEON (ARM)	128 bits	2	4
SVE (ARM)	variable	según implementación	—

La anchura fija de las tres primeras es una decisión de ISA con coste: cada ampliación exige instrucciones nuevas y recompilar. SVE y RVV, la extensión vectorial de RISC-V, adoptan **longitud de vector agnóstica**: el programa consulta la longitud en ejecución, así que el mismo binario aprovecha una implementación más ancha sin recompilar. Es la vuelta a la idea de los procesadores vectoriales clásicos.

5.2.1 Cómo se usan

Cuatro vías, de más a menos recomendable:

1. **Vectorización automática.** El compilador lo hace con `-O3`. Se comprueba con `-fopt-info-vec` y `-fopt-info-vec-missed`.

2. **Directivas.** `#pragma omp simd` indica que el bucle es vectorizable, y `#pragma omp simd reduction(+:s)` que la acumulación se puede reasociar.
3. **Intrínsecas.** Funciones que se corresponden con instrucciones concretas. Portables entre compiladores, no entre arquitecturas.
4. **Ensamblador.** Solo cuando nada más funciona.

```c
#include <immintrin.h>

double producto(const double *restrict a, const double *restrict b, int n) {
    __m256d acc = _mm256_setzero_pd();
    int i;
    for (i = 0; i + 3 < n; i += 4) {
        __m256d va = _mm256_loadu_pd(a + i);
        __m256d vb = _mm256_loadu_pd(b + i);
        acc = _mm256_fmadd_pd(va, vb, acc);
    }
    double t[4];
    _mm256_storeu_pd(t, acc);
    double s = t[0] + t[1] + t[2] + t[3];
    for (; i < n; i++) s += a[i] * b[i];
    return s;
}
```

Dos cosas de este fragmento:

- El **bucle de cola** para los elementos que no completan un vector no es opcional. Olvidarlo es un error de resultado, no de compilación.
- `_mm256_fmadd_pd` es una **multiplicación y suma fusionadas**: una sola instrucción, un solo redondeo. Es más rápida y **más precisa** que multiplicar y sumar por separado, y por eso puede dar un resultado distinto al de la versión escalar.

5.2.2 Obstáculos

Obstáculo	Efecto	Solución
Solapamiento de punteros	el compilador no puede reordenar	`restrict`
Dependencia entre iteraciones	impide vectorizar	reescribir
Acceso con zancada o indirecto	carga elemento a elemento	reorganizar los datos
Datos no alineados	carga más lenta	alinear a 32 o 64 bytes
Ramas dentro del bucle	fuerza ejecución con máscara	simplificar

El **acceso indirecto** —`v[idx[i]]`— es el que más limita en la práctica. Las instrucciones de recolección existen y son mucho más lentas que una carga contigua, así que la vectorización aporta poco. La solución no es una instrucción mejor sino cambiar la disposición de los datos.

5.2.3 Disposición de los datos

Disposición	Estructura	Acceso a un campo
Vector de estructuras	`struct P { float x, y, z; } p[N];`	con zancada 3; trae los otros campos a la caché
Estructura de vectores	`struct { float x[N], y[N], z[N]; } p;`	contiguo y vectorizable

Recorrer un solo campo con el primer esquema desperdicia dos tercios del ancho de banda. Con el segundo, cada campo es un vector contiguo. Es la transformación de mayor efecto en código numérico, y no cambia ni una operación aritmética.

5.3 Procesadores vectoriales

Los que dieron origen a la idea, en los supercomputadores de los años setenta y ochenta. Registros vectoriales de longitud grande, un registro de longitud que dice cuántos elementos son válidos, y unidades funcionales profundamente segmentadas.

Sus dos mecanismos característicos siguen siendo relevantes:

- **Encadenamiento.** El resultado de una operación vectorial alimenta a la siguiente elemento a elemento, sin esperar a que el vector entero esté listo.
- **Registro de máscara.** Un bit por elemento indica cuáles participan. Es lo que permite vectorizar un bucle con un `if` dentro: se calculan todos y solo se escriben los que la máscara habilita.

Frente a las extensiones SIMD, un procesador vectorial gestiona la longitud en ejecución y no en el repertorio, lo que evita el bucle de cola. AVX-512 recuperó las máscaras, y SVE y RVV, la longitud variable: el diseño vectorial clásico ha vuelto por partes.

5.4 GPU

Una unidad gráfica de propósito general lleva el paralelismo de datos al extremo: miles de hilos ligeros ejecutando el mismo código sobre datos distintos.

5.4.1 Modelo de ejecución

El modelo se llama SIMT, hilos únicos con instrucción múltiple. Los hilos se agrupan en unidades de 32 o 64 que ejecutan **la misma instrucción a la vez**. El programador ve hilos independientes; el hardware ejecuta grupos en bloque.

Concepto	Qué es
Hilo	la unidad lógica, con sus registros
Grupo (*warp*)	32 hilos que avanzan juntos
Bloque	conjunto de grupos con memoria compartida y barreras
Malla	todos los bloques del lanzamiento

5.4.2 Divergencia

Si dentro de un grupo unos hilos toman una rama y otros la contraria, el hardware **ejecuta las dos ramas en serie**, desactivando en cada una los hilos que no le corresponden. El grupo tarda la

suma de las dos.

```
if (idx % 2 == 0) { camino_a(); } else { camino_b(); }
```

Ese código divide cada grupo por la mitad y duplica el tiempo. Reorganizar los datos para que los hilos de un mismo grupo tomen la misma rama es la optimización característica de GPU, y no tiene equivalente en CPU.

5.4.3 Jerarquía de memoria

Memoria	Ámbito	Latencia
Registros	hilo	mínima
Compartida	bloque	baja, gestionada por el programador
Global	toda la malla	alta
Constante y de texturas	toda la malla, solo lectura	con caché propia

La memoria compartida es una caché **explícita**: la gestiona el programa, no el hardware. Cargar en ella un bloque de datos que se va a reutilizar es el patrón central de la programación de GPU, y es el mismo bloqueo por *tiles* que se aplica a la multiplicación de matrices en CPU, con la diferencia de que aquí es obligatorio escribirlo.

Y la otra optimización crítica: la **coalescencia**. Si los hilos consecutivos de un grupo acceden a posiciones consecutivas, el hardware combina los accesos en una sola transacción. Si acceden de forma dispersa, hace una por hilo. La diferencia puede ser de un factor de treinta y dos.

5.4.4 Cuándo compensa una GPU

A favor	En contra
Miles de elementos con el mismo tratamiento	poco paralelismo
Alta intensidad aritmética	mucho control de flujo divergente
Accesos regulares y contiguos	accesos irregulares
El dato se queda en la GPU varias fases	transferencias frecuentes por PCIe

La última fila decide más casos de los que parece: la transferencia entre memoria principal y memoria de GPU puede costar más que el cálculo. Un núcleo de cómputo diez veces más rápido que la CPU no gana nada si hay que copiar los datos de ida y vuelta cada iteración.

5.5 Comparación

	SIMD en CPU	GPU
Paralelismo	decenas de elementos	miles de hilos
Latencia de una operación	baja	alta, se oculta con más hilos
Control de flujo	penaliza	penaliza mucho más
Transferencia de datos	ninguna	por PCIe

	SIMD en CPU	GPU
Programación	directivas o intrínsecas	CUDA, OpenCL, SYCL, OpenMP con descarga

Las dos son complementarias, no alternativas. Un programa bien optimizado usa SIMD dentro de cada hilo, varios hilos por núcleo y descarga a la GPU las fases que lo justifican. El análisis de estas arquitecturas está en Hennessy et al., 2026, y su tratamiento en el contexto de la asignatura en Anguita López y Ortega Lopera, 2016.

Capítulo 6

Problemas resueltos

Los cálculos que la asignatura pide, resueltos paso a paso. Cubren las prestaciones del tema 1, la sincronización del tema 3, el ILP del tema 4 y la vectorización del tema 5.

6.1 Ganancia, eficiencia y Amdahl

6.1.1 Problema 1

Un programa tarda 100 s en un procesador. El 80 % de ese tiempo es paralelizable. Calcular la ganancia y la eficiencia con 4, 16 y 64 procesadores, y la cota superior.

Solución. Con $f = 0{,}2$ de fracción secuencial:

$$S(p) = \frac{1}{0{,}2 + \dfrac{0{,}8}{p}}$$

p	T_p (s)	$S(p)$	$E(p)$
1	100,0	1,00	1,00
4	40,0	2,50	0,63
16	25,0	4,00	0,25
64	21,3	4,71	0,07
∞	20,0	**5,00**	0

Pasar de 16 a 64 procesadores —cuadruplicar la máquina— gana un 18 %. La eficiencia cae al 7 %: **59 de los 64 procesadores no aportan nada**. El número que gobierna todo es el 20 % secuencial, y reducirlo al 10 % duplicaría la cota.

6.1.2 Problema 2

Un programa alcanza $S = 3{,}2$ con 4 procesadores. Estimar la fracción secuencial y predecir la ganancia con 32.

Solución. Despejando f de la ley de Amdahl:

$$f = \frac{\dfrac{p}{S} - 1}{p - 1} = \frac{\dfrac{4}{3{,}2} - 1}{3} = \frac{0{,}25}{3} = 0{,}083$$

Con $f = 0{,}083$ y $p = 32$:

$$S(32) = \frac{1}{0{,}083 + \dfrac{0{,}917}{32}} = \frac{1}{0{,}1117} = 8{,}95$$

Esta forma de estimar f a partir de una medida se llama **fracción serie de Karp-Flatt**, y su utilidad real es aplicarla a varios valores de p: si f sale constante, la pérdida es la fracción secuencial; si crece con p, hay sobrecarga de comunicación o sincronización que Amdahl no modela.

6.1.3 Problema 3

Comparar escalabilidad fuerte y débil sobre el mismo programa, con $f = 0{,}05$ y 64 procesadores.

Solución.

Fuerte, con Amdahl:

$$S = \frac{1}{0{,}05 + \dfrac{0{,}95}{64}} = 15{,}4$$

Débil, con Gustafson:

$$S = p - f(p-1) = 64 - 0{,}05 \cdot 63 = 60{,}9$$

El mismo programa y la misma máquina dan 15,4 o 60,9 según qué se mantenga fijo. No hay contradicción: la primera resuelve el mismo problema más rápido y la segunda resuelve un problema 64 veces mayor en el mismo tiempo. Publicar la segunda cifra sin decir cuál se midió es la forma más común de exagerar resultados en esta materia.

6.2 Prestaciones y memoria

6.2.1 Problema 4

Un procesador tiene CPI de ejecución 1,2. El 30 % de las instrucciones acceden a memoria. La caché falla el 4 % de las veces y la penalización es de 150 ciclos. Calcular el CPI real y qué fracción del tiempo se pierde en memoria.

Solución. Cada instrucción hace un acceso para buscarse a sí misma más 0,3 de datos, es decir 1,3 accesos:

$$\text{CPI} = 1{,}2 + 1{,}3 \cdot 0{,}04 \cdot 150 = 1{,}2 + 7{,}8 = 9{,}0$$

La memoria aporta 7,8 de los 9,0 ciclos: el **87 %** del tiempo. Bajar la tasa de fallos del 4 % al 1 % deja el CPI en 3,15, casi tres veces mejor, sin tocar el procesador.

6.2.2 Problema 5

Una máquina NUMA de dos nodos tiene latencia local de 90 ns y remota de 140 ns. Un programa hace el 100 % de sus accesos en el nodo local si se inicializa bien, y el 50 % remotos si toda la memoria acaba en un nodo. Calcular la diferencia.

Solución.

- Bien colocado: 90 ns por acceso.

– Mal colocado: $0{,}5 \cdot 90 + 0{,}5 \cdot 140 = 115$ ns.

Un 28 % más lento, y ni una línea del cálculo ha cambiado: solo **quién tocó la memoria primero**. Es el efecto de la política de primer toque del tema 3, y se corrige inicializando en paralelo con el mismo reparto que usa el cálculo.

6.2.3 Problema 6

Ocho hilos acumulan en `double parcial[8]`, un elemento cada uno. Las líneas de caché son de 64 bytes. Explicar el problema y calcular el relleno necesario.

Solución. Ocho `double` son 64 bytes: **el vector entero cabe en una sola línea**. Cada escritura de cualquier hilo invalida la línea en los otros siete, así que la línea viaja continuamente entre núcleos aunque no se comparta ningún dato. Es falso compartimiento.

Con `double` de 8 bytes y línea de 64, hacen falta $64 - 8 = 56$ bytes de relleno por elemento:

```
struct { double v; char relleno[56]; } parcial[8];
```

El vector pasa de 64 a 512 bytes y cada hilo escribe en su propia línea. La alternativa sin relleno es que cada hilo acumule en una variable local y solo escriba en el vector al final, que es lo que `reduction` hace por dentro.

6.3 Sincronización

6.3.1 Problema 7

Comparar el tráfico de coherencia de un cerrojo `test-and-set` con el de un cerrojo con espera de lectura, con p hilos esperando.

Solución. Con `test-and-set` puro, cada hilo ejecuta una operación de **escritura** en cada intento. Toda escritura invalida la línea en los otros $p - 1$, así que en cada vuelta hay del orden de p transacciones de coherencia, y $O(p^2)$ mientras se espera.

Con espera de lectura, los hilos leen en bucle: la línea queda en estado compartido y las lecturas no generan tráfico. Solo cuando el cerrojo se libera intentan la operación atómica, y ahí hay una ráfaga de $O(p)$.

Un cerrojo MCS baja incluso eso a $O(1)$, porque cada hilo espera sobre una variable propia y el que sale despierta solo al siguiente de la cola.

6.3.2 Problema 8

Un programa tiene una sección crítica de 200 ns dentro de un bucle donde cada iteración cuesta 2 µs. Calcular el número máximo de hilos que pueden trabajar sin que el cerrojo sea el cuello de botella.

Solución. La sección crítica se ejecuta en serie, así que el sistema completo no puede hacer más de una cada 200 ns. Cada hilo pide entrar una vez por iteración, es decir cada 2 µs:

$$p_{max} = \frac{2000 \text{ ns}}{200 \text{ ns}} = 10$$

Con más de diez hilos, el cerrojo satura y añadir hilos no aporta nada. Es Amdahl otra vez, con la sección crítica en el papel de fracción secuencial: aquí $f = 0{,}1$ y la cota es 10.

6.4 Paralelismo a nivel de instrucción

6.4.1 Problema 9

Una suma en coma flotante tiene latencia 4 ciclos y la unidad está segmentada con iniciación cada ciclo. Calcular el tiempo de un bucle de acumulación de n elementos con uno y con cuatro acumuladores.

Solución.

Con un acumulador, cada suma depende de la anterior, así que hay que esperar la latencia completa: $4n$ ciclos.

Con cuatro acumuladores independientes, las cuatro cadenas se entrelazan y la unidad acepta una suma por ciclo: n ciclos, más los 4 del vaciado final.

Ganancia: 4. El número de acumuladores que compensa es exactamente la latencia dividida por el intervalo de iniciación; añadir más no mejora y consume registros.

6.4.2 Problema 10

Un cauce de 18 etapas tiene una penalización de 17 ciclos por fallo de predicción. Los saltos son el 20 % de las instrucciones. Calcular el CPI añadido con acierto del 90 % y del 99 %.

Solución.

$$\mathrm{CPI}_{extra} = 0{,}20 \cdot (1 - h) \cdot 17$$

Acierto	CPI añadido
90 %	$0{,}20 \cdot 0{,}10 \cdot 17 = 0{,}34$
99 %	$0{,}20 \cdot 0{,}01 \cdot 17 = 0{,}034$

Sobre un CPI base de 0,25 —un superescalar de cuatro vías— el primer caso **duplica con creces** el tiempo de ejecución y el segundo lo empeora un 14 %. Es el argumento cuantitativo de por qué los predictores actuales son tan elaborados, y de por qué ordenar los datos antes de un bucle con `if` puede acelerarlo varias veces.

6.5 Vectorización

6.5.1 Problema 11

Un bucle procesa $n = 1000$ elementos `float`. Calcular cuántas iteraciones vectoriales y cuántas de cola hacen falta con SSE, AVX2 y AVX-512.

Solución. Elementos `float` por registro: 4, 8 y 16.

Extensión	Iteraciones vectoriales	Cola
SSE	$\lfloor 1000/4 \rfloor = 250$	0
AVX2	$\lfloor 1000/8 \rfloor = 125$	0
AVX-512	$\lfloor 1000/16 \rfloor = 62$	8

Solo AVX-512 deja cola. Con $n = 1003$ las tres la dejarían, y omitir ese bucle final no da error de compilación: da un resultado incorrecto en los últimos elementos, que es peor.

6.5.2 Problema 12

Un producto escalar lee dos vectores de `double` y hace una multiplicación y una suma por elemento. La máquina alcanza 100 GFLOPS y 25 GB/s. Determinar si el programa está limitado por cálculo o por memoria.

Solución. Por elemento: 16 bytes leídos y 2 operaciones. La intensidad operacional es

$$I = \frac{2 \text{ ops}}{16 \text{ bytes}} = 0{,}125 \text{ ops/byte}$$

El rendimiento que el ancho de banda permite:

$$25 \text{ GB/s} \times 0{,}125 \text{ ops/byte} = 3{,}1 \text{ GFLOPS}$$

Muy por debajo de los 100 GFLOPS de pico, así que el programa está **limitado por memoria**. Vectorizarlo no lo acelerará: las unidades ya esperan datos. La intensidad de corte, donde los dos límites se cruzan, es $100/25 = 4$ ops/byte, y un producto escalar está treinta veces por debajo.

Una multiplicación de matrices por bloques, en cambio, reutiliza cada dato B veces y su intensidad crece con el tamaño del bloque: ahí sí compensa vectorizar.

6.6 Coherencia

6.6.1 Problema 13

Dos procesadores con protocolo MESI. P1 lee X, luego P2 lee X, luego P1 escribe X, luego P2 lee X. Indicar el estado de la línea en cada caché tras cada paso.

Solución.

Paso	Caché de P1	Caché de P2	Transacción
P1 lee X	Exclusiva	—	fallo de lectura; nadie más la tiene
P2 lee X	Compartida	Compartida	P1 detecta la petición y degrada su estado
P1 escribe X	**Modificada**	Inválida	P1 invalida la copia de P2
P2 lee X	Compartida	Compartida	P1 sirve el dato y actualiza memoria

El estado exclusivo del primer paso es lo que evita una transacción en el tercero **cuando no hay lector intermedio**: si P2 no hubiera leído, P1 pasaría de exclusiva a modificada en silencio, sin tocar el bus. Ese caso, un procesador que lee y luego escribe un dato que nadie más usa, es el más frecuente de todos, y es la razón de que el estado exista.

Estos problemas siguen el planteamiento de Anguita López y Ortega Lopera, 2016; el marco cuantitativo es el de Hennessy et al., 2026 y Ortega Lopera et al., 2005.

Capítulo 7

Temario práctico

Las tres prácticas del programa: programar con memoria compartida, optimizar para una microarquitectura con paralelismo a nivel de instrucción, y medir si algo de lo anterior ha servido.

7.1 Práctica 1. Programación paralela con memoria compartida

7.1.1 Puesta en marcha

```
gcc -fopenmp -O2 -o programa programa.c
export OMP_NUM_THREADS=8
./programa
```

Sin `-fopenmp` el compilador **ignora las directivas** y produce un programa secuencial correcto. Es la primera comprobación cuando una versión paralela no acelera nada: puede que ni siquiera sea paralela.

Variables de entorno que hacen falta en las medidas:

Variable	Para qué
`OMP_NUM_THREADS`	número de hilos
`OMP_PROC_BIND`	fija los hilos a núcleos
`OMP_PLACES`	qué núcleos, y cómo se agrupan
`OMP_SCHEDULE`	planificación cuando el código usa `runtime`
`OMP_DISPLAY_ENV`	imprime la configuración efectiva al arrancar

`OMP_PROC_BIND=close` con `OMP_PLACES=cores` es el punto de partida razonable en una máquina NUMA: sin fijación, el planificador mueve los hilos y cada migración pierde la caché y la localidad de memoria del tema 3.

7.1.2 Distribución del trabajo

```
#pragma omp parallel for schedule(static) reduction(+:suma)
for (int i = 0; i < n; i++) {
    suma += f(v[i]);
}
```

El ejercicio de la práctica es comparar planificaciones sobre un bucle con coste por iteración desigual. Con un bucle triangular:

```
for (int i = 0; i < n; i++)
```

```
    for (int j = i; j < n; j++)
        m[i][j] = calcular(i, j);
```

`schedule(static)` reparte bloques contiguos, así que el hilo que recibe las primeras filas hace mucho más trabajo y los demás esperan en la barrera final. `schedule(static,1)` reparte cíclicamente y equilibra sin coste adicional; `schedule(dynamic)` equilibra mejor todavía y paga sincronización sobre el contador. Medir los tres es lo que convierte la tabla del tema 2 en un resultado.

7.1.3 Comunicación y sincronización

```
#pragma omp parallel
{
    int yo = omp_get_thread_num();
    parcial[yo] = trabajar(yo);

    #pragma omp barrier

    #pragma omp single
    combinar(parcial);
}
```

Los errores que la práctica enseña a reconocer:

Error	Síntoma
Acumular sobre una variable `shared` sin `reduction`	resultado ligeramente distinto en cada ejecución
Declarar `private` un acumulador	empieza con basura; `firstprivate` lo inicializa
`master` donde hacía falta `single`	no hay barrera y los demás avanzan sobre datos incompletos
`critical` alrededor del cuerpo entero	el programa se serializa y va más lento que el secuencial
Barreras dentro de una construcción de reparto	comportamiento indefinido: no todos los hilos las alcanzan

La última no es teórica. Poner un `#pragma omp barrier` dentro de un `for` paralelo es ilegal precisamente porque los hilos no ejecutan el mismo número de iteraciones.

7.1.4 Falso compartimiento, medido

El experimento que cierra la práctica, porque conecta el tema 1 con un número:

```
/* con falso compartimiento: las ocho posiciones caen en una o dos lineas */
double parcial[8];

/* sin el: cada hilo escribe en su propia linea de 64 bytes */
struct { double v; char relleno[56]; } parcial[8];
```

El segundo puede ser varias veces más rápido con ocho hilos, sin cambiar una sola operación aritmética. `perf c2c` señala exactamente qué línea de caché está viajando entre núcleos.

7.1.5 Comprobar la corrección

```
gcc -fopenmp -fsanitize=thread -g -o prog prog.c && ./prog
valgrind --tool=helgrind ./prog
```

Que el programa dé el resultado correcto **no demuestra nada**: una carrera puede no manifestarse en mil ejecuciones. Estas dos herramientas la encuentran aunque no se produzca, porque analizan los accesos y no el resultado.

7.2 Práctica 2. Optimización de código para microarquitecturas ILP

Partir de una versión ingenua y transformarla paso a paso, midiendo cada cambio.

7.2.1 El caso de referencia

```
/* version 0: una cadena de dependencias de longitud n */
double suma(const double *v, int n) {
    double s = 0.0;
    for (int i = 0; i < n; i++) s += v[i];
    return s;
}
```

Cada suma depende de la anterior, así que el bucle avanza al ritmo de la latencia de la suma en coma flotante, unos cuatro ciclos por elemento, con la unidad segmentada casi vacía.

```
/* version 1: cuatro cadenas independientes */
double suma4(const double *v, int n) {
    double s0=0, s1=0, s2=0, s3=0;
    int i;
    for (i = 0; i + 3 < n; i += 4) {
        s0 += v[i];   s1 += v[i+1];
        s2 += v[i+2]; s3 += v[i+3];
    }
    for (; i < n; i++) s0 += v[i];
    return (s0 + s1) + (s2 + s3);
}
```

Lo que importa son los acumuladores independientes, no el desenrollado. Y hay que decir el precio: **el resultado cambia en los últimos dígitos**, porque la suma en coma flotante no es asociativa. Por eso el compilador no lo hace solo sin `-ffast-math`.

7.2.2 Ramas impredecibles

```
for (int i = 0; i < n; i++)
    if (v[i] > umbral) s += v[i];
```

Con datos aleatorios el predictor falla la mitad de las veces y cada fallo cuesta la penalización completa. Dos formas de arreglarlo:

- **Ordenar los datos antes.** El mismo bucle se vuelve varias veces más rápido sin tocar una línea.
- **Escribirlo sin rama**, con una expresión que el compilador convierta en `cmov`, o con máscaras

SIMD.

Se comprueba con `perf stat -e branch-misses`, que es el contador que lo delata.

7.2.3 Comprobar qué hizo el compilador

```
gcc -O3 -march=native -fopt-info-vec-missed -S prog.c
objdump -d prog.o | less
```

`-fopt-info-vec-missed` dice **por qué** no vectorizó un bucle, que es más útil que saber que no lo hizo. Las causas habituales son el posible solapamiento de punteros, que se resuelve con `restrict`, y las dependencias entre iteraciones, que exigen reescribir.

7.2.4 El orden de los cambios

La práctica insiste en un orden concreto, y tiene motivo:

1. Elegir el algoritmo. Ninguna micro-optimización compensa una complejidad peor.
2. Arreglar el acceso a memoria: recorrido por filas, bloqueo, disposición de las estructuras.
3. Dejar que el compilador trabaje: `-O3`, `restrict`, funciones en línea.
4. Romper cadenas de dependencias y quitar ramas impredecibles.
5. Solo entonces, intrínsecas o ensamblador.

Saltarse los tres primeros para escribir intrínsecas es el error clásico: se optimiza en detalle un código limitado por el ancho de banda de memoria, donde el paralelismo aritmético no cambia nada.

7.3 Práctica 3. Evaluación de prestaciones

7.3.1 Cómo se mide

```
#include <omp.h>
double t0 = omp_get_wtime();
trabajo();
double t1 = omp_get_wtime();
printf("%.6f s\n", t1 - t0);
```

Reglas que la práctica exige y que casi todo el mundo incumple al principio:

- **Reloj monótono, no el de pared del sistema.** `omp_get_wtime` y `clock_gettime(CLOCK_MONOTONIC)` valen; la hora del día da saltos al ajustarse.
- **Descartar la primera ejecución.** Carga las cachés y las páginas.
- **Repetir y quedarse con la mediana o el mínimo**, no con la media: un pico del sistema contamina la media y no la mediana.
- **Fijar la frecuencia** o al menos anotarla. Con escalado dinámico, la misma medida da resultados distintos según la temperatura del equipo.
- **Anotar la máquina**: `lscpu`, `numactl --hardware`, versión del compilador y opciones exactas. Una medida sin esos datos no es reproducible.

7.3.2 Contadores hardware

```
perf stat -e cycles,instructions,cache-misses,branch-misses ./prog
perf stat -e L1-dcache-load-misses,LLC-load-misses ./prog
perf record ./prog && perf report
```

Contador	Qué diagnostica
`instructions / cycles`	el IPC: cuánto ILP se está aprovechando
`cache-misses`	si el problema es la jerarquía de memoria
`branch-misses`	si el problema son las ramas
`LLC-load-misses`	si se está yendo a memoria principal

Un IPC bajo con pocos fallos de caché apunta a cadenas de dependencias; un IPC bajo con muchos fallos, a memoria. Es el diagnóstico que decide cuál de las dos prácticas anteriores aplicar.

7.3.3 Escalabilidad

Las dos curvas que hay que producir, y no confundir:

- **Escalabilidad fuerte.** Problema de tamaño fijo, número de hilos creciente. Se representa $S(p)$ frente a p y se compara con la recta ideal. La ley de Amdahl del tema 1 predice dónde se dobla la curva, y ajustar f a los datos da la fracción secuencial real del programa.
- **Escalabilidad débil.** Trabajo por hilo fijo, problema y número de hilos creciendo juntos. Lo ideal es una línea horizontal en el tiempo.

Presentar una curva de escalabilidad débil llamándola fuerte es la forma habitual de inflar resultados, y es lo que la práctica enseña a detectar.

Y la trampa del denominador: T_1 **tiene que ser el mejor programa secuencial**, no la versión paralela con un hilo. Esta última arrastra la sobrecarga de OpenMP, así que usarla infla la ganancia sistemáticamente.

7.3.4 Modelo del techo

Cruzar el pico aritmético de la máquina con su ancho de banda de memoria sitúa cada núcleo de cómputo según su **intensidad operacional**, en operaciones por byte leído. El punto de corte dice si el programa está limitado por cálculo o por memoria, y por tanto qué tiene sentido optimizar.

Un producto escalar tiene intensidad baja —dos cargas por multiplicación y suma— y está limitado por memoria: vectorizarlo mejora poco. Una multiplicación de matrices por bloques tiene intensidad alta y sí se beneficia. Es el mismo razonamiento del tema 5 sobre cuándo compensa una GPU, con números en lugar de intuición.

7.3.5 El informe

Lo que la práctica pide entregar, y que es el hábito que queda:

1. La máquina, con sus datos.
2. El compilador y las opciones exactas.
3. Tamaño del problema y número de repeticiones.
4. Tiempos con su dispersión, no un solo número.
5. Las curvas de ganancia y eficiencia, y qué predice Amdahl.
6. La explicación de por qué la curva se dobla donde se dobla.

El punto 6 es el que distingue una medida de un resultado. Los detalles de estas prácticas y sus problemas resueltos están en Anguita López y Ortega Lopera, 2016, la parte de OpenMP en Chapman et al., 2008 y la metodología de medida en Rauber y Rünger, 2023.

Bibliografía

Anguita López, M., & Ortega Lopera, J. (2016). *Fundamentos y Problemas de Arquitectura de Computadores*. Editorial Técnica Avicam.

Chapman, B., Jost, G., & van der Pas, R. (2008). *Using OpenMP. Portable Shared Memory Parallel Programming*. MIT Press.

Fog, A. (2004). How to Optimize for the Pentium Family of Microprocessors.

Gerber, R., Bik, A. J. C., Smith, K. B., & Tian, X. (2006). *The Software Optimization Cookbook* (2.ª ed.). Intel Press.

Hennessy, J. L., Patterson, D. A., & Kozyrakis, C. (2026). *Computer Architecture: A Quantitative Approach* (7.ª ed.). Elsevier.

Ortega Lopera, J., Anguita López, M., & Prieto Espinosa, A. (2005). *Arquitectura de Computadores*. Thomson.

Rauber, T., & Rünger, G. (2023). *Parallel Programming: For Multicore and Cluster Systems*. Springer.

Wilkinson, B., & Allen, M. (2005). *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers* (2.ª ed.). Pearson Prentice Hall.