Tema 09: Multiprocesamiento

Arquitectura de Computadoras

Ing. Nicolás Majorel Padilla (npadilla@herrera.unt.edu.ar)

http://microprocesadores.unt.edu.ar/arqcom/

Temas que veremos

- Multiprocesamiento.
- Performance en sistemas paralelos.
  - Aceleración, Limitaciones, Eficiencia, Escalabilidad.
- Taxonomía de Flynn.
  - Paralelismo a nivel Datos: SIMD.
  - Paralelismo a nivel *Threads*: *Multithreading*.
  - Paralelismo a nivel núcleo: *Multicores*.
  - Paralelismo a nivel aplicación: Clusters.
    - Evolución de los centros de datos.

Lectura recomendada

- Computer Organization and Design, RISC-V Edition (2da ed, 2021):
  - Sección 6.1: *Introduction*
  - Sección 6.2: *The Difficulty of Creating Parallel Processing Programs*
  - Sección 6.3: *SISD, MIMD, SIMD, SPMD and Vector*
  - Sección 6.4: *Hardware Multithreading*
  - Sección 6.5: *Multicore and Other Shared Memory Multiprocessors*
  - Sección 6.8: *Clusters, Warehouse Scale Computers, and Other Message-Passing Multiprocessors*
  - Sección 6.14: *Fallacies and Pitfalls*
  - Sección 6.15: *Concluding Remarks*

¿Dónde estamos parados?

- Comenzamos con un procesador con CPI = 1 y T largo.
- Le agregamos Pipelining, para mantener CPI = 1 y hacer T más pequeño.
- Con Superpipelining mantuvimos CPI = 1 y hicimos T mucho más pequeño.
- Finalmente, agregamos Paralelismo (ILP) para hacer que CPI sea menor que 1, manteniendo T muy pequeño.
- Muchísimas mejoras de Performance.
- Sin embargo…

¿Dónde estamos parados?

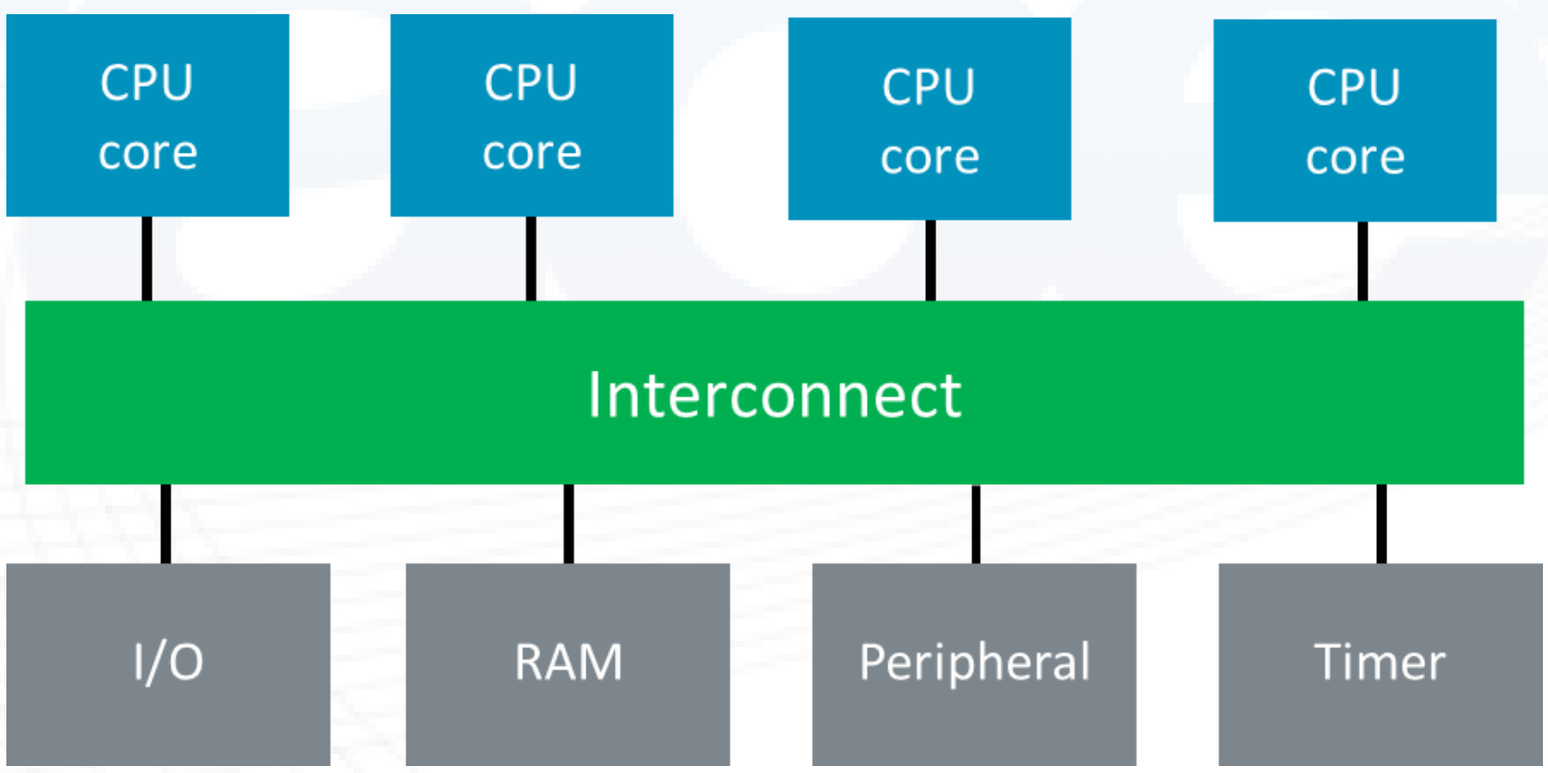
- Aumentamos mucho Hw, agregando complejidad y sobre todo área, lo que aumenta el costo (*Price*).
- Con Pipelining, conocimos los riesgos.
  - Hicimos mucho esfuerzo para evitarlos y minimizarlos.
- ILP está fuertemente limitado por las dependencias, en códigos mayormente secuenciales.
- La *Power Wall* y el fin del escalamiento de Dennard nos impiden seguir aumentando la frecuencia para mejorar la performance.

¿Dónde estamos parados?

- A pesar de todo lo anterior, la necesidad de seguir mejorando la performance es continua.
  - Es lo que mueve toda la industria.
  - *¿Qué más podemos hacer?*
- La *Power Wall* generó un cambio de paradigma que nos da la respuesta:
  - Buscar mejorar la performance mediante el uso de varios procesadores juntos (**multiprocesamiento**).
- Nuevamente aplicar Paralelismo (Gran Idea en Arquitectura de Computadoras), pero ahora a niveles diferentes.

Multiprocesamiento – Concepto

- En realidad, como concepto ya existía desde antes.
  - Los factores anteriores lo impulsaron,lo popularizaron y lo llevaron a nuestros bolsillos.
  - Se pasó de conectar varios procesadores en un sistema, a conectar varios núcleos en un mismo procesador.
  - El concepto de *System on a Chip* (SoC) mencionado en Tema 02.



Multiprocesamiento – Características

- Más procesadores…
  - Aumentan la performance (en principio).
  - Disminuyen el consumo de energía (en principio).
  - Aumentan la disponibilidad.
  - Disminuyen la eficiencia.
    - Generan problemas de sincronización y de coherencia de datos.
    - Generan dificultad para los programadores (no son transparentes).
- Esta última característica es la más importante.
  - ILP es transparente a los programadores.

Multiprocesamiento – Implicancias

- Cuatro estudiantes quieren hacer un TP de esta materia en paralelo, para hacerlo más rápido.
  - Inicialmente, deberían **dividir la tarea**.
    - De manera idea, en partes iguales (**balance de carga**).
    - Si no, algunos terminarán antes que los demás y tendrán que esperar (**sincronización**).
      - Se asume que es posible dividir el TP en partes iguales, lo cual no siempre es así.
      - Se asume también que todos los estudiantes son iguales, lo cual no siempre es así.
  - Luego, asignar una parte del TP a cada uno de ellos (***scheduling***).
  - Además, es probable que mientras cada uno hace su parte del TP pierdan algo de tiempo conversando entre ellos (***overhead* por comunicación**).

Multiprocesamiento – Implicancias

- Si los estudiantes fueran ocho, las implicancias anteriores se agravan.
- Todos los tipos de paralelismo que veremos tendrán que manejar estos detalles.
- **El mayor desafío del procesamiento paralelo es decidir cómo es mejor partir un problema en pedazos para que puedan ser procesados por separado**.
  - Esta dificultad da lugar a los distintos tipos de paralelismo que veremos.

Consideraciones sobre performance

- Un sistema es **escalable** si al incrementar los recursos (Hw), se logra una mejora proporcional en la performance.
- Si realizamos una tarea con *p* procesadores, *¿mejorará p veces?*
- **A (*p*) = t_ej(1 procesador) / t_ej(*p* procesadores)**
- Lo ideal sería obtener una aceleración **lineal**.

Consideraciones sobre performance

- Se desea sumar 8000 números independientes, cada suma demora 1 T, se cuenta con 8 procesadores.
- Separamos la tarea en pedazos independientes.
  - Cada procesador suma 1000 números.
  - Son necesarios 3 ciclos más para sumar los subtotales (coordinación).
- A(8) = 8000 / 1003 = 7,98
  - Aceleración casi ideal. Excelente caso.
  - No estamos considerando problemas de sincronización.
    - Que un procesador quede esperando datos de otro.

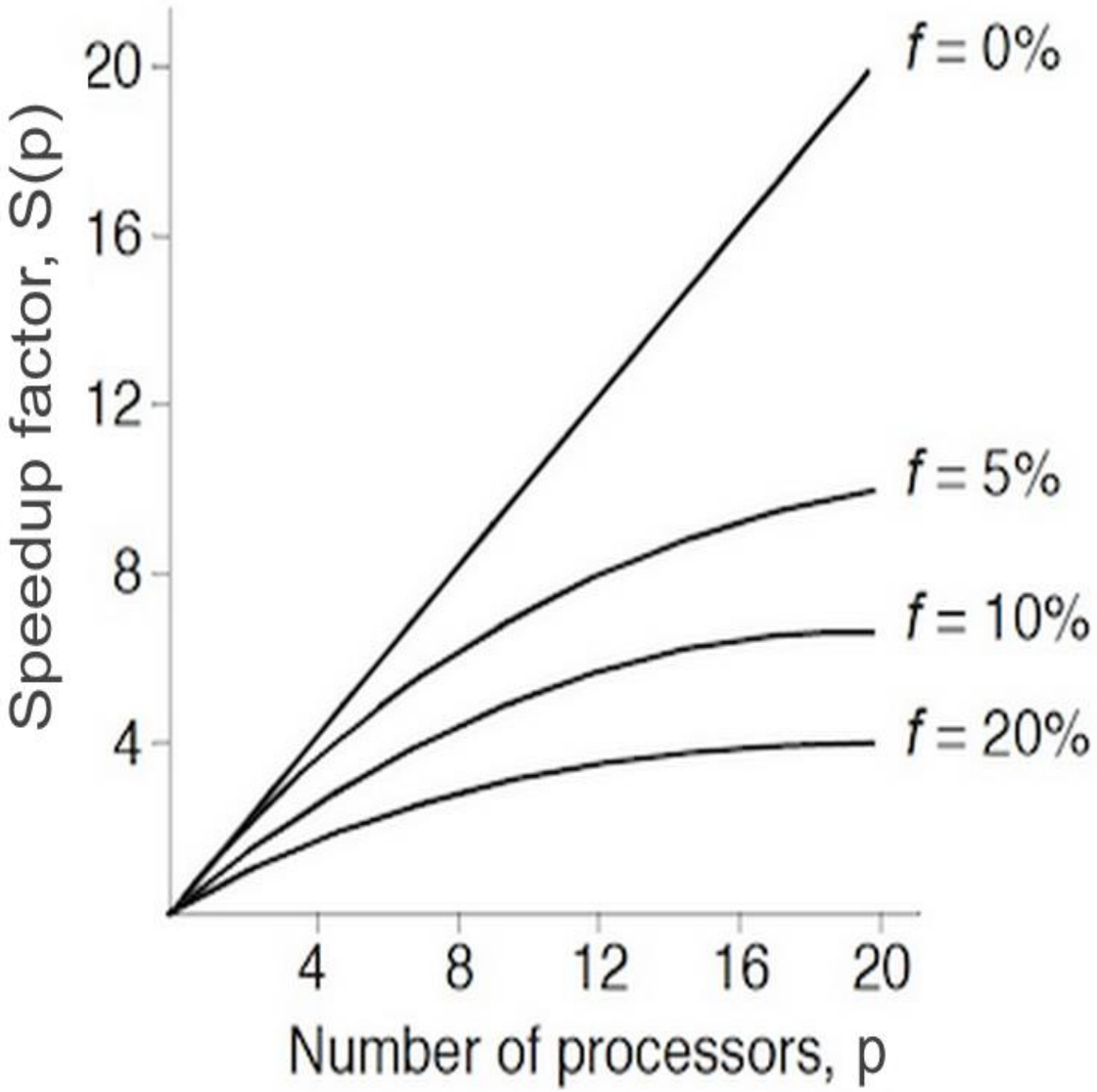
Consideraciones sobre performance

- El ejemplo anterior es bastante irreal.
  - Puede haber limitaciones por el uso de memoria, por el uso de I/O, por el medio de conexión.
- Ley de Amdahl, otra vez.
  - La mejora total de un sistema está limitada por la parte no modificada.
  - **A (*p*) = 1 / [1 – F + (F / *p*)]**

Consideraciones sobre performance

- Tenemos los mismos 8 procesadores anteriores, y queremos ejecutar una tarea que tiene un 20% de código secuencial, y el 80% restante es completamente paralelizable.
  - F = 0,8; p = 8.
  - A = 1 / [0,2 + (0,8 / 8)] = **3,33**
- Fuerte baja. Ni siquiera la mitad del ideal.
  - Y nada que ver con el Hw. Limitación exclusiva de Sw.

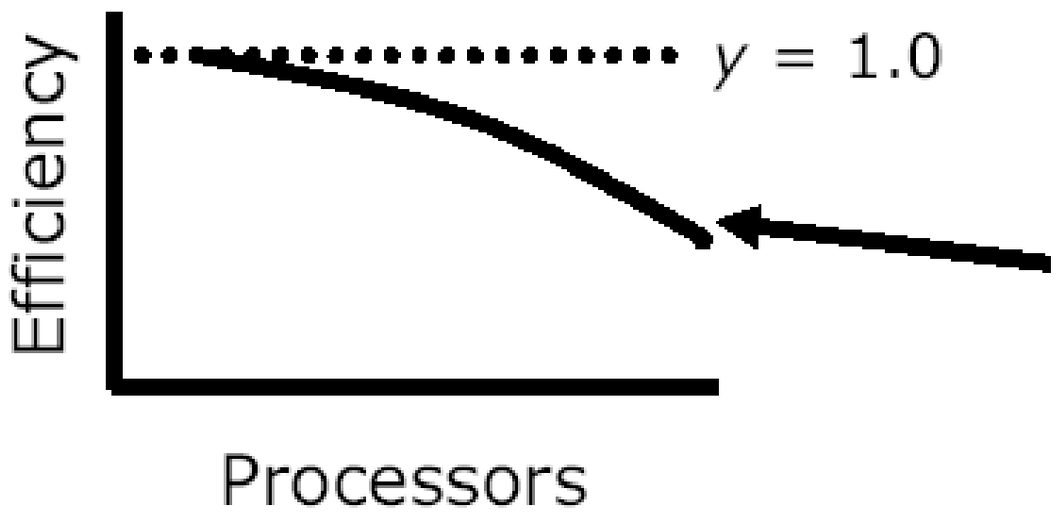
Conclusiones sobre Ley de Amdahl



- La fracción serial impone una fuerte limitación al paralelismo.
- Con muchos procesadores empieza a disminuir la aceleración.
  - Debido a problemas de comunicación y/o coordinación (congestión).
- Asume que el problema a resolver es de tamaño fijo.
  - La carga de trabajo de cada procesador va disminuyendo.
- Esto se conoce como **Escalabilidad fuerte** (*Strong Scaling*)

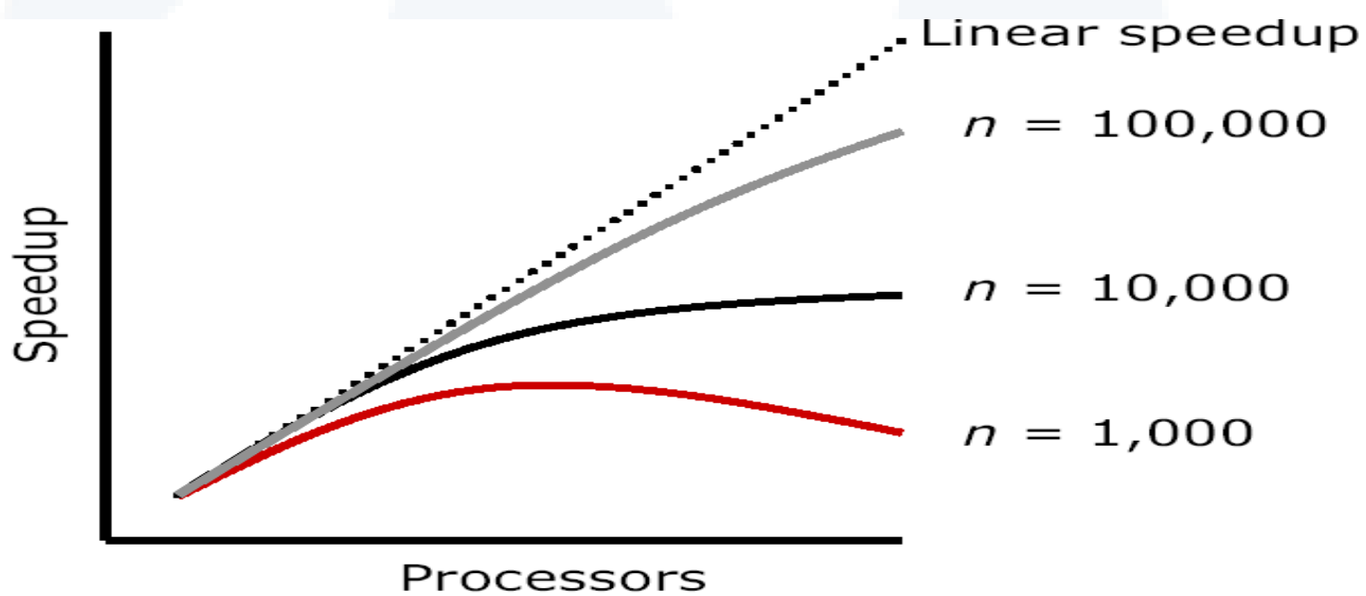
Eficiencia del paralelismo

- La aceleración no es la única métrica importante.
  - Ej: obtener una aceleración 2 usando 10 procesadores, *¿es un buen resultado?*
- La eficiencia es cuan bien se utilizan los recursos disponibles.
  - **E (*p*) = A (*p*) / *p***
  - Ej: si con 8 procesadores se obtiene una aceleración de 3,33, la eficiencia será del 41,6%.



Alternativa para mejorar Escalabilidad

- Aumentar los procesadores para realizar **una** tarea baja eficiencia, y está limitado por la parte serial.
- Entonces… les demos muchas tareas.
  - **Adaptar la cantidad de trabajo al paralelismo disponible.**
  - Cuanto mayor sea el conjunto de datos, más próxima será la aceleración al ideal.
    - La carga de trabajo de cada procesador se mantiene constante.
  - Esto se conoce como **Escalabilidad débil** (*Weak Scaling*).
  - No siempre es posible. Depende del perfil y del tamaño de la tarea.

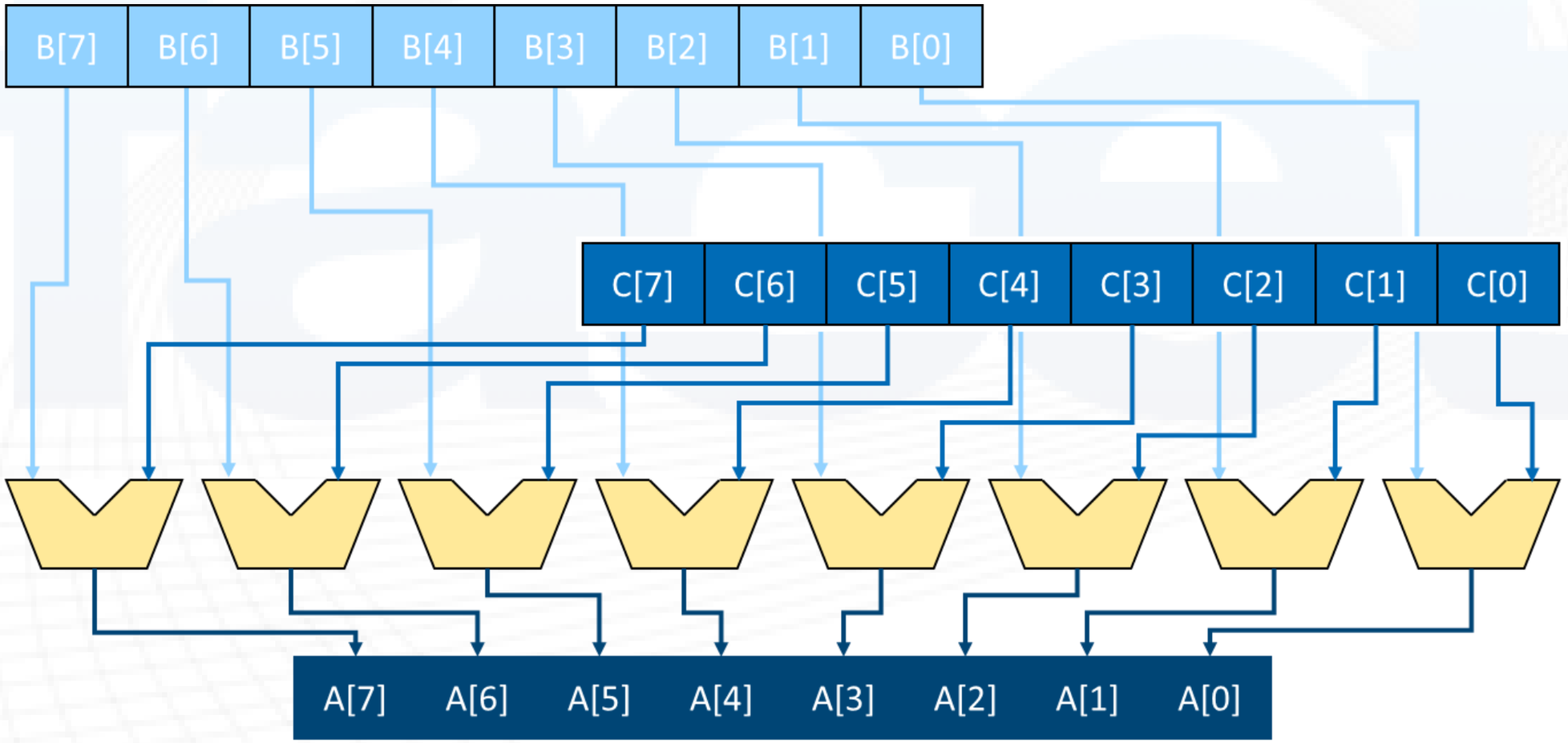


Una clasificación: Taxonomía de Flynn

- Hecha alrededor de 1960, propone categorías según la combinación entre cantidad de instrucciones y cantidad de datos.
- SISD**:** *Single Instruction, Single Data*.
  - Un procesador convencional que vimos hasta aquí. Todos los procesadores previos a 2004.
- **SIMD:** *Single Instruction, Multiple Data*.
  - Una única instrucción opera sobre un conjunto de datos.
- MISD**:** *Multiple Instruction, Single Data*.
  - No utilizado.
- **MIMD:** *Multiple Instruction, Multiple Data*.
  - Múltiples porciones de un mismo programa se ejecutan en distintos procesadores, con distintos datos.
  - Un procesador estándar desde 2004.
  - Dos variantes: *Multithreading* y *Multicore*.

SIMD – Concepto

- Ejecutan la misma instrucción de manera simultánea (en paralelo) sobre datos distintos.
- Pueden realizar una operación elemento a elemento sobre varios elementos de dos vectores en un solo ciclo
  - Ideales para trabajar con arrays en lazos for..



SIMD – Ventajas

- Explotan el **paralelismo a nivel datos (DLP)**.
  - Ideales cuando los datos son regulares.
- Bastante simples.
- La sincronización está implícita.
- Mercado inicial: aplicaciones multimedia.
  - Video maneja RGB: 3x8 bits + 8 bits de transparencia.
  - Audio: 24 bits/muestra.
- Mercado actual: multimedia + procesamiento digital de señales (DSP)
  - Manejo de sensores, control de servomotores, codecs de audio, reconocimiento de voz, códecs de video, robótica, etc.

SIMD – Desventajas

- La longitud de los registros es fija, **definida por el ISA**.
  - Se “divide” a los registros en “carriles” de datos más pequeños.
- Nuevas longitudes de registros implica agregar nuevas instrucciones.
  - Lo que a su vez implica dar nuevo soporte en los compiladores, SO, etc.
  - ¡Y reescribir/recompilar las aplicaciones!
  - Es un problema para los ISA de formatos de longitud fija.

SIMD – Ejemplo en RV32

- En RISC-V, hay una extensión no estándar (extensión P).
- Supongamos que queremos calcular la suma de dos vectores de 40 elementos de tamaño un byte (C = A + B)

```
li t0, 40                   # Inicializar contador
loop_scalar:
    lbu t1, 0(a0)           # Carga A[i]
    lbu t2, 0(a1)           # Carga B[i]
    add t3, t1, t2          # Suma
    sb t3, 0(a2)            # Guarda C[i]

    addi a0, a0, 1          # Actualizar punteros
    addi a1, a1, 1
    addi a2, a2, 1
    addi t0, t0, -1         # Decrementar contador
    bnez t0, loop_scalar
```

SIMD – Ejemplo en RV32

- La extensión P agrega una instrucción **padd.b**, que suma el contenido de dos registros, pero considerándolo como 4 elementos de un byte.

```
li t0, 10                   # Inicializar contador
loop_simd:
    lw t1, 0(a0)            # Carga 4 elementos
    lw t2, 0(a1)            # Carga 4 elementos
    padd.b t3, t1, t2       # SUMA SIMD
    sw  t3, 0(a2)           # Guarda los 4 elementos

    addi a0, a0, 4          # Actualizar punteros
    addi a1, a1, 4
    addi a2, a2, 4
    addi t0, t0, -1         # Decrementar contador
    bnez t0, loop_simd      # Si t0 != 0, repetir
```

SIMD – Ejemplo en RV32

- *¿Qué aceleración se obtiene con el código anterior?*

- Ventajas:
  - Se reduce la cantidad de iteraciones del lazo.
    - Porque ahora se pueden **cargar** y **procesar** varios elementos en paralelo.
  - No requiere agregar registros.
  - Se aprovechan mejor los accesos a memoria.
- Desventajas:
  - La cantidad de elementos de los vectores debería ser múltiplo de la cantidad de operaciones que se puedan hacer en paralelo.
  - Los elementos de los vectores deben estar contiguos en memoria y alineados.
  - Limitado por el tamaño de los registros (32 o 64 bits).

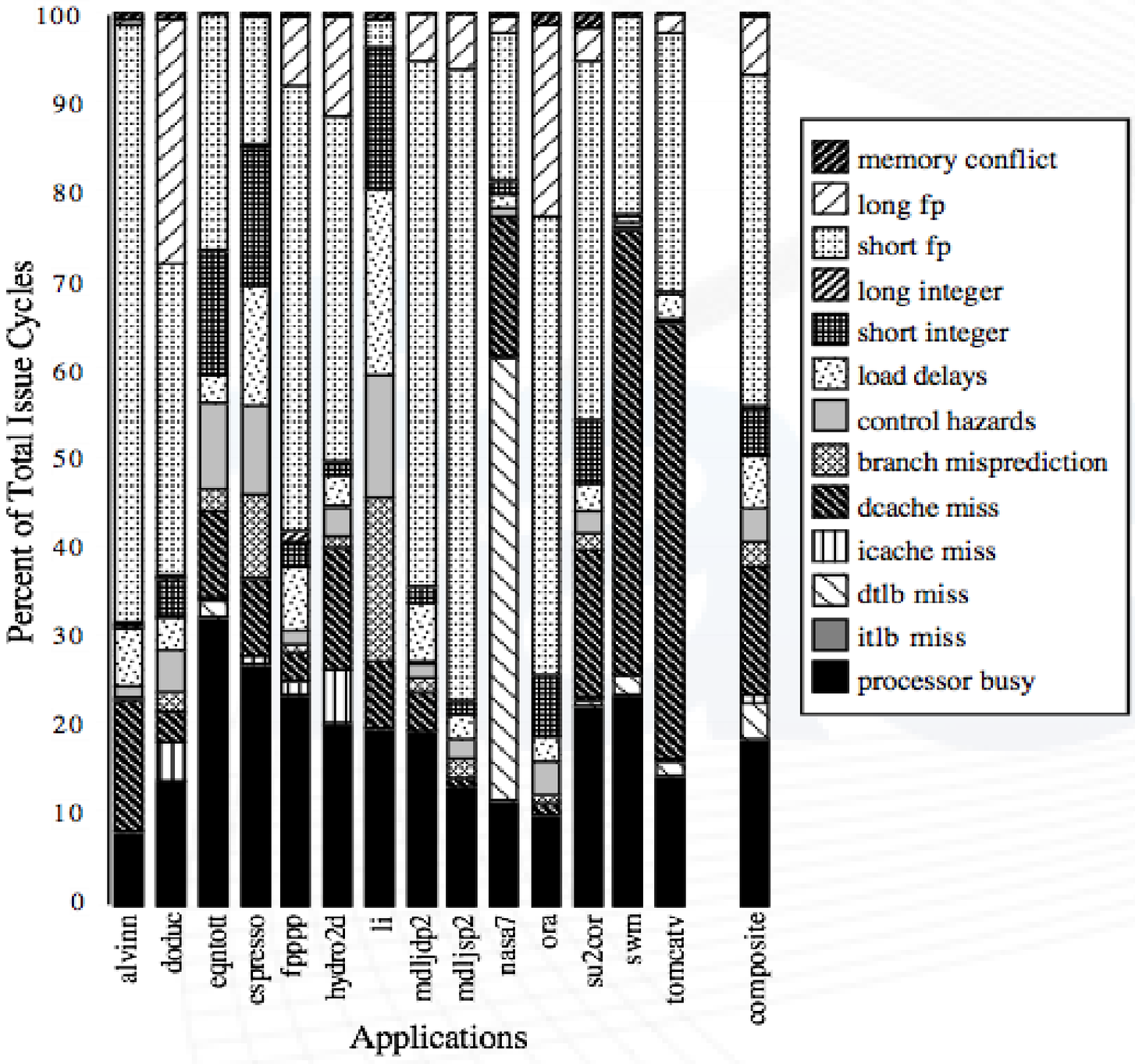
SIMD – Ejemplos en otros ISAs

- En las arquitecturas ARM, existe la extensión NEON.
  - Sí añade un banco de registros adicional, más ancho (32 registros de 128 bits).
  - vmla.f32 v2, v0, v1 # v2 = v2 + (v0 * v1)
    - Considera el contenido de los registros como 4 números de 32 bits, los cuales multiplica y acumula en un tercer registro.
- Muy popular en la arquitectura x86.
  - Extensiones MMX (64 bits), SSE (128 bits), SSE2, AVX (256 bits), AVX2, AVX512 (512 bits).
  - Cada extensión añade registros e instrucciones.
  - Por ejemplo, en AVX2:
    - vpbroadcastd ymm0, [eax]
    - Carga 8 veces el valor de 32 bits del registro eax en el registro ymm0.

SIMD – Variantes

- Dentro de esta categoría podrían entrar también **arquitecturas vectoriales** y los **procesadores gráficos (GPU)**.
  - Distintos en esencia, pero también explotan paralelismo a nivel datos.
- El ISA de RISC-V provee una extensión estándar para operaciones vectoriales (extensión V).
  - No posee registros de longitud fija, sino que la longitud la define la implementación del fabricante y no el ISA.
    - Permite varias implementaciones diferentes de Hw, pero simplificando el Sw, porque no necesita agregar instrucciones.
      - Una misma instrucción puede sumar dos vectores, independientemente de su tamaño.
    - Su configuración es dinámica, en tiempo de ejecución.
      - No se modifica el ISA, ni las aplicaciones, ni los compiladores, SO, etc.
- Imitada por ARM en sus ISAs más reciente, llamada SVE (*Scalable Vector Extension*) y su sucesora SVE2.
- Estos temas irían en “Arquitectura 2” ☺

Multithreading – Motivación



- Un análisis de ILP para un procesador superescalar que puede emitir 8 instrucciones por ciclo mostró que la mayoría del tiempo, el procesador está parado sin hacer nada.
- No se puede cambiar de tarea.
- Surgió el concepto de hilos de ejecución (*threads*).

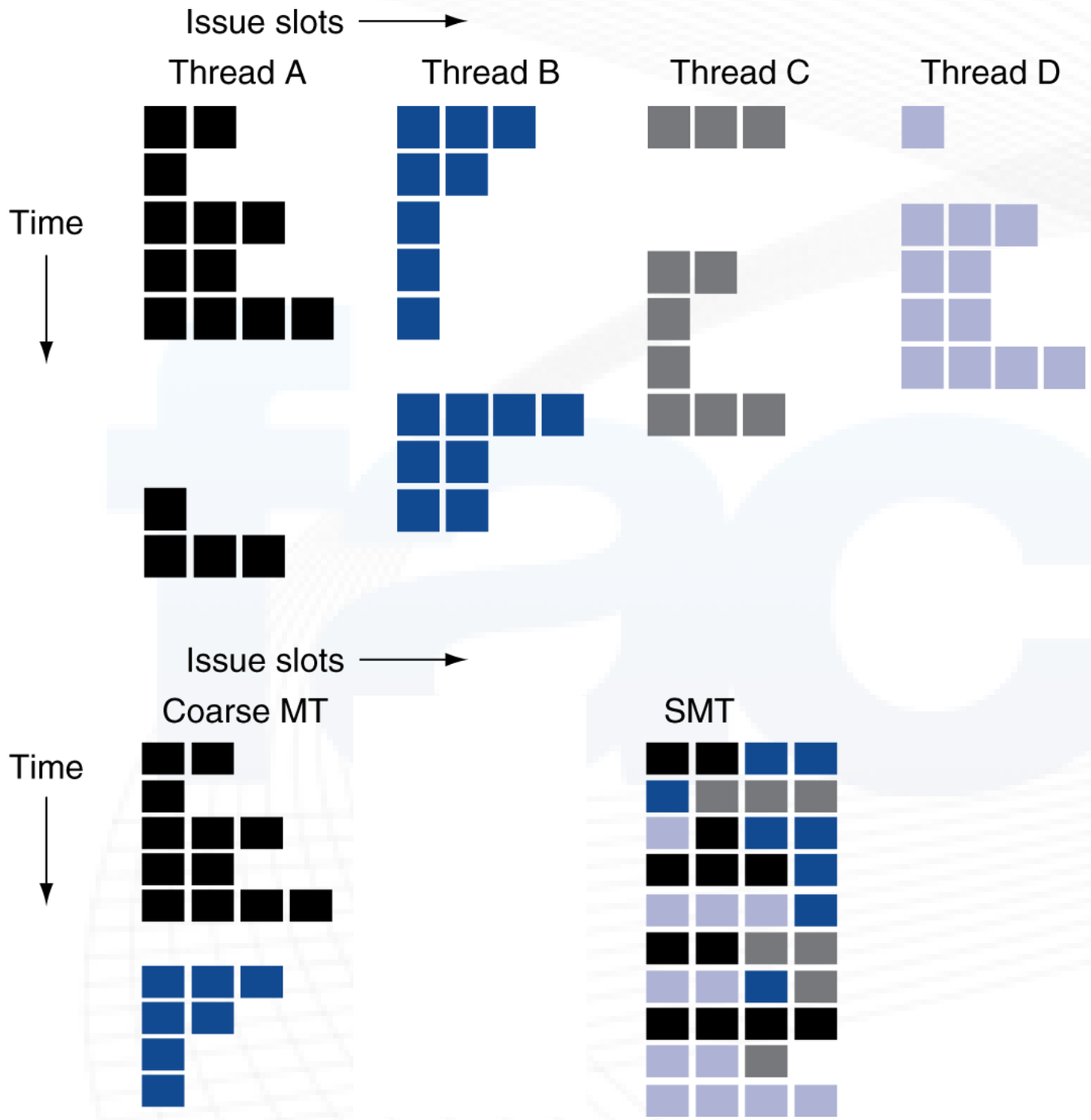
Multithreading – Definiciones

- Los hilos de ejecución son funciones **independientes** de un mismo programa.
  - Comparten los recursos del programa.
  - Cada *thread* tiene su propio PC y su propio estado (banco de registros, etc).
  - Es mucho más rápido cambiar de *thread* que de proceso.
    - El cambio suele ser instantáneo.
- Ejemplos:
  - Sonido estéreo.
  - Corrector ortográfico instantáneo.
- Si las instrucciones provienen de distintos *threads* de un mismo programa, ya se sabe que son independientes.
  - O sea, que no tienen dependencias.

Multithreading – Variantes

- Un único procesador que soporta múltiples *threads* de ejecución.
  - Cada *thread* es considerado como un procesador virtual.
- ***Coarse-grained multithreading***: Cuando la ejecución de un *thread* se ve detenida por alguna parada, se cambia a otro *thread*.
  - La parada queda enmascarada, la ejecución continúa (pero de otro *thread*).
  - La clave es que el cambio de *thread* sea más rápido que la resolución de la parada.
- ***Multithreading* simultáneo (SMT)**: aprovecha las características de un procesador superescalar con emisión dinámica de múltiples instrucciones, y busca emitir siempre instrucciones de diferentes *threads* para llenar las unidades funcionales disponibles.

Multithreading – Variantes



- Cuatro *threads* diferentes, que se ejecutan en un procesador superescalar *4-wide*.
- Las dependencias hacen que no siempre se puedan emitir 4 instrucciones en cada ciclo (ILP).
- SMT aprovecha que las instrucciones de distintos *threads* no tienen dependencias entre sí.
  - Y las ejecuta en simultáneo.

Multithreading – Ventajas

- Es una forma de explotar el **paralelismo a nivel *threads* (TLP)**.
- Caso de uso ideal: un *thread* principal más un *thread* pequeño que ejecute en paralelo, y cuando haya mucha comunicación entre ambos.
- 2 *threads* en SMT generan una mejora promedio del 20-25%
  - Con muy poco agregado de recursos: duplicar el PC, el banco de registros, y algunos registros de estado.
    - Aproximadamente un aumento del 5% del área y un aumento del 2% del consumo de energía.
  - En *threads* con bajo ILP, es posible obtener una aceleración casi lineal.

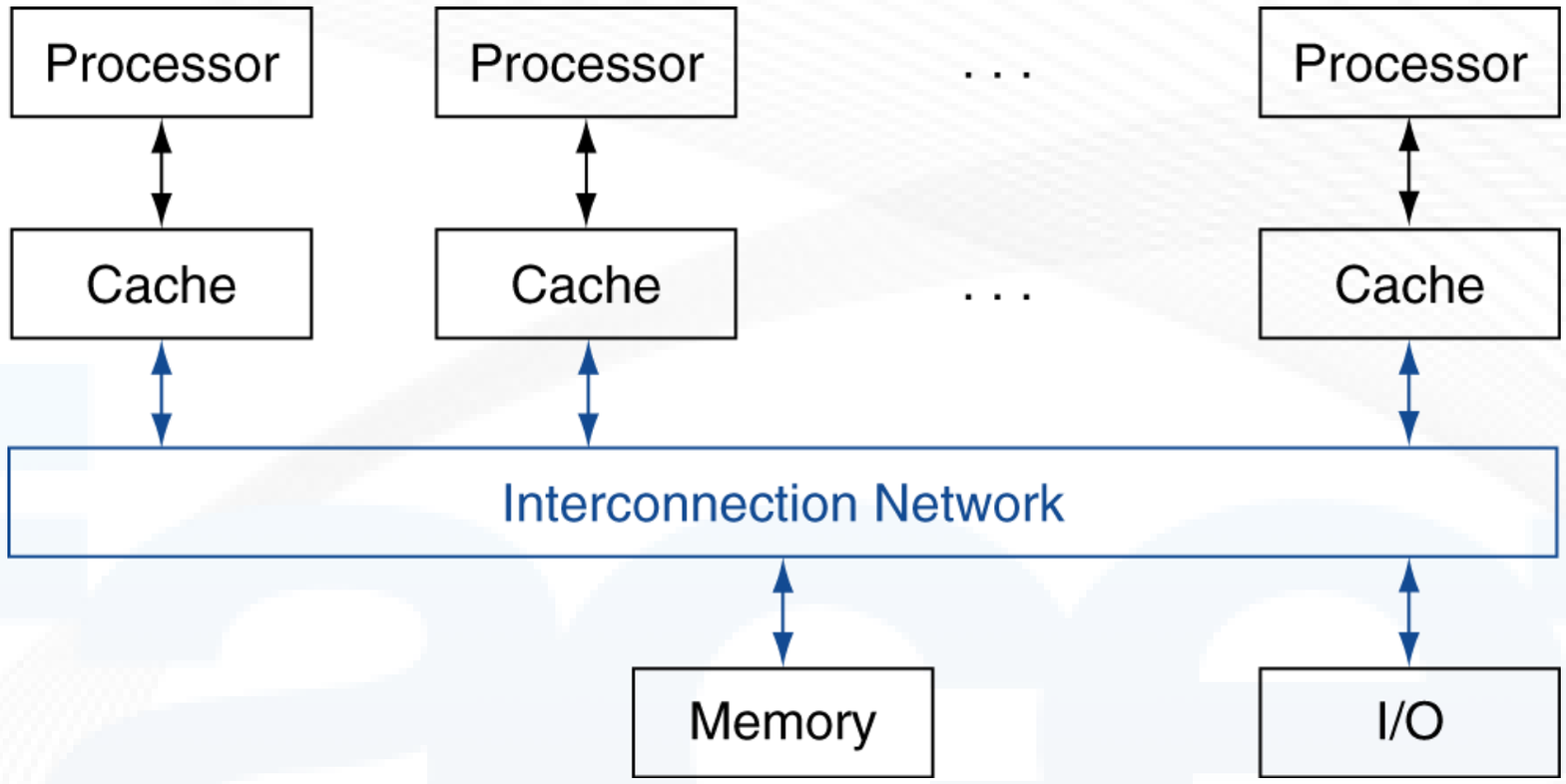
Multithreading – Desventajas

- Limitación 1: **software**
  - Procesadores con muchos *threads* por núcleo no mostraron una mejora que valga la pena.
  - Los programas (Sw) deben ser reescritos para usar los *threads*.
    - Por lo tanto, no usan muchos *threads*.
  - La ejecución de programas que poseen solamente un único *thread* puede empeorar, por el *overhead* necesario para dar soporte.
- Conclusión: **usar pocos *threads* por núcleo.**
- Limitación 2: **seguridad**
  - Se descubrió que el soporte para SMT representa una seria vulnerabilidad de seguridad.
  - Porque implica compartir muchos recursos: memorias de datos e instrucciones, buffers, estructuras para predicción de saltos, etc.
  - Más detalles en Tema 17.

Multicore – Concepto

- Varios procesadores trabajan en conjunto ejecutando distintas partes de un mismo programa.
- La manera más común y simple de que los procesadores trabajen en paralelo, es que compartan información a través de la memoria.
  - Todos los procesadores comparten un único espacio de direcciones.
  - Todos los datos de memoria están disponibles para todos los procesadores en todo momento.
- También conocido como **multiprocesamiento con acceso a memoria compartido** (*Shared Memory multiProcessors*, SMP).
  - Prácticamente la mayoría de los procesadores desde 2004.

Multicore – Características



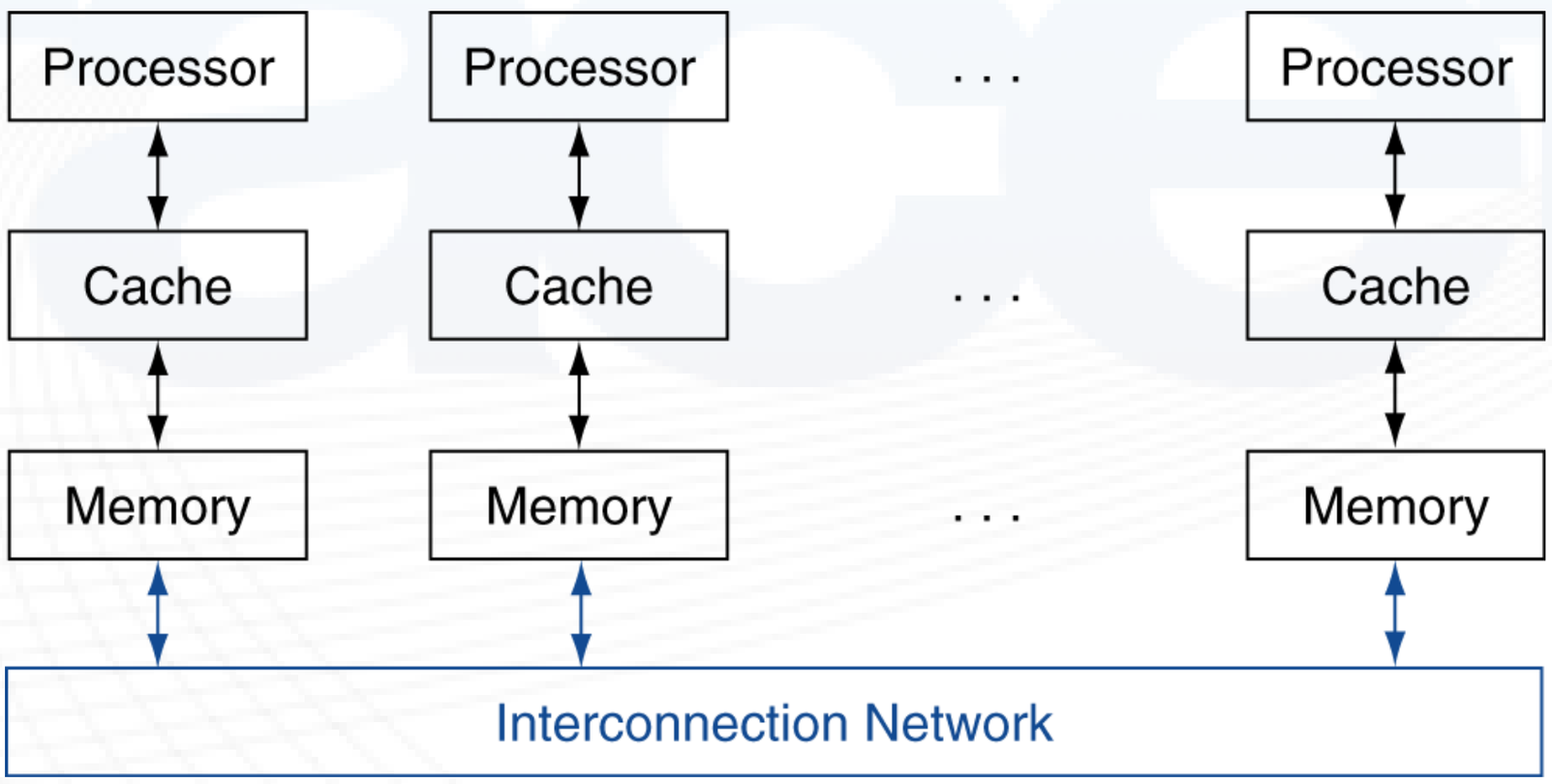
- Los distintos procesadores se conectan a la memoria compartida a través de alguna red de interconexión.
- Pueden ser núcleos dentro de un mismo chip, o procesadores dentro de un mismo servidor.
- Es una forma de explotar el **paralelismo a nivel núcleos**.

Multicore – Características

- Como la memoria es compartida, es bastante simple de programar: a través de instrucciones Ld/St.
  - Se necesitan operaciones atómicas en memoria para sincronización.
  - Operaciones que no se pueden interrumpir.
  - Fundamentales para Sistemas Operativos.
  - En el ISA de RISC-V, están incluidas en la extensión A. *¿Se acuerdan?*

Clusters – Concepto

- La progresión natural de *multicore*, es que en vez de que sean varios procesadores trabajando en conjunto, sean varias computadoras completas.
- En este caso, las redes de interconexión comunican nodos completos procesador-memoria a través del sistema de I/O.



Clusters – Concepto

- El acceso a memoria ya no es compartido.
  - Los procesadores tienen cada uno un espacio de direccionamiento privado.
- Es necesario que los programas incorporen un mecanismo explícito para compartir datos, que se denomina interfaz para **paso de mensajes** (*Message Passing Interface,* MPI).
  - Las aplicaciones empaquetan los datos que quieren transmitir en forma de mensajes.
  - La comunicación es explícita, mediante rutinas para enviar *send*(Pi, Dk) y recibir *wait*(Pj,Dk).
  - Hay librerías que implementan estas rutinas para muchos lenguajes de programación, e iniciativas abiertas como OpenMPI.

Clusters – Concepto

- Las redes de comunicación suelen ser de bajo costo (Ethernet).
- Muy populares para ejecutar aplicaciones que aprovechan **paralelismo a nivel aplicación (ALP)**.
  - Búsqueda web, servidor de mail, servidor de archivos, bases de datos.
- Poseen una alta disponibilidad, a diferencia de SMP.
- Son muy fáciles de escalar.

Computadoras de gran escala

- La popularidad de varios servicios relacionados a Internet produjo que algunos clusters sean extremadamente grandes.
  - Reciben el nombre de ***Warehouse-Scale Computers*** **(WSC)**.
- Su tamaño incrementa la complejidad.
  - Aproximadamente 100.000 servidores.
  - Necesitan espacio físico, energía para alimentarlos y mantenerlos a baja temperatura.
  - Tolerancia a fallos es muy crítica.
- Proveen *Software as a Service* (SaaS).
  - También conocido como la popular “nube” (*cloud computing*).
  - Millones de pequeños pedidos independientes simultáneos que pueden ser paralelizados.
  - Su métrica más importante es la latencia, vista desde el usuario final.

Evolución de los *Datacenters*

- Los clusters fueron populares en los centros de datos desde mediados de 1990 hasta mediados de 2000.
  - Salas de servidores dedicadas, con equipos HP, Dell, Sun.
- Desde mediados de 2000 hasta hoy, son populares los centros de datos de gran escala (WSC).
  - Proveedores de servicios *cloud*: Google, Amazon, Microsoft, Meta.
- Actualmente (2026) se está hablando del término ***Global-Scale Computing***, totalmente novedoso.
  - Centros de datos masivos, distribuidos globalmente pero coordinados.
  - Proveedores de servicios de IA: Google, OpenAI.

¿Cómo podemos explotar paralelismo?

- Reescribiendo programas con algoritmos/lenguajes que usen paralelismo.
  - Concentrarse en la parte más lenta (Amdahl).
  - Concentrarse en cómo particionar los datos, y cómo hacer la sincronización.
- Distintos enfoques, no excluyentes:
  - Paralelismo a nivel datos.
  - Paralelismo a nivel *thread*.
  - Paralelismo a nivel núcleos.
  - Paralelismo a nivel aplicación.
- No escalar nunca paralelismo sin escalar datos.

Resumen final

- Multiprocesamiento como técnica para seguir aumentando la performance.
  - Superar limitaciones de ILP y *Power Wall*.
- Problemas: no es transparente, dificultad para dividir la tarea, comunicación, sincronización.
  - Performance limitada por software (Amdahl).
  - No tiene sentido aumentar paralelismo sin escalar los datos.
- SIMD para aprovechar paralelismo a nivel datos.
  - Multimedia, procesamiento digital de señales.
- *Multithreading* simultáneo para aprovechar paralelismo a nivel *threads*.

Resumen final

- *Multicores* con acceso a memoria compartido, para aprovechar el paralelismo a nivel núcleos.
- Clusters que se comunican mediante paso de mensajes, para aprovechar el paralelismo a nivel aplicación.
  - Interconexión de computadoras completas, mediante redes de bajo costo, con alta disponibilidad.
  - Escalaron a WSC que proveen Cloud Computing.