Infraestructuras Paralelas y Distribuidas
Clase 01 - Introducción a la programación en paralelo

John Sanabria - john.sanabria@correounivalle.edu.co

Temas a abordar

- Motivación a la programación en paralelo
- Qué son los patrones de programación en paralelo y por qué son importantes?
- Objetivo de la paralelización → reducir el *span*
- La ley de Amdahl, ley de Moore
- ¿Por qué aparecen los procesadores *multicore*? Las paredes en el diseño de procesadores
- La “memoria” nuestro cuello de botella

Introducción a Sistemas de Cómputo Paralelos - Basado en el libro Structured Parallel Programming

- Hoy sistemas paralelos de cómputo los encontramos desde el celular, pasando por los *laptops*, hasta los entornos de nube
- La paralelización automática funciona pero algunos detalles aún se les escapan. Programar en paralelo implica un **conocimiento importante** del hardware del computador

Veamos un poco de realidad

Vamonos de compras

- Visitemos una tienda en línea (Alkosto, Éxito)
- Búsquemos algunos celulares como: IPhone, Samsung, y otra marca.

Lectura - Paralelismo en todas partes

Lectura sobre la necesidad del paralelismo

Pensando en paralelo

- Programación serial es natural pero es inadecuada para las nuevas arquitecturas de cómputo. La programación serial es fácil porqué:
  - Es fácil de razonar sobre ella
  - Los programas son **generalmente** determinísticos
- La propuesta de **patrones estructurados paralelos** permite concentrarse en la solución del problema e ignorar los detalles de implementación. Ejemplo PThread vs OpenMP
- Los patrones paralelos reducen el **no-determinismo** que es común en la programación paralela. Ejemplo de no determinismo con `pthreads`

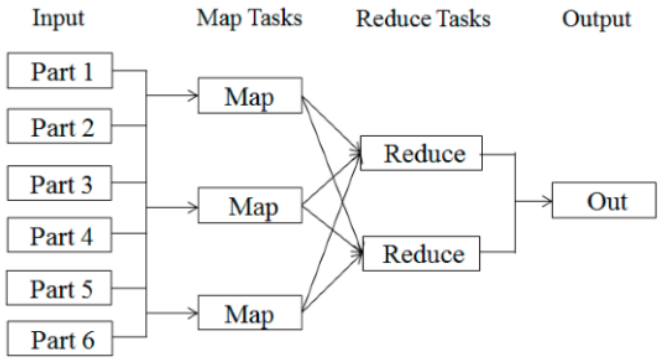


Figure 1: Búsqueda del máximo en paralelo

Trampas seriales

- El objetivo de un programador de aplicaciones paralelas es encontrar las partes del código sujetas a ser paralelizables
- Sin embargo, existen las **trampas seriales** las cuales son aquellas partes del código que parecieran deben ser seriales/secuenciales pero no lo son

¿Cuál de estos ciclos `for` se puede paralelizar?

```
for (int i = 0; i < MAX_VECTOR; i++) // inicializacion
  vector[i] = i;

for (int i = 0; i < MAX_VECTOR; i++) // operacion
  vector[i] = fibonacci(i);

printf("[");
for (int i = 0; i < MAX_VECTOR; i++) // impresion ordenada del vector
  printf(" %d ",vector[i]);
printf("]");
```

Escriba su programa en C - 40 minutos

- En el slide anterior usted vió tres ciclos.
- Escriba un programa en C que paralelice la mayor cantidad de ciclos y se comporte de forma correcta.
- ¿Qué observa de su solución?

Rendimiento

- Cuando se piensa en programas eficientes paralelos se piensa **reducir la cantidad de trabajo computacional** → *trampa serial*
- **El objetivo a la hora de paralelizar un algoritmo es minimizar su *span* o tiempo de ejecución**
- Al *span* del algoritmo se le conoce también como la **ruta crítica** y nos indica el tiempo más largo que toma llevar alguna de las computaciones ejecutadas en paralelo dentro de un programa
- Una forma de reducir el *span* es a través de la reducción de la comunicación o acceso a los datos

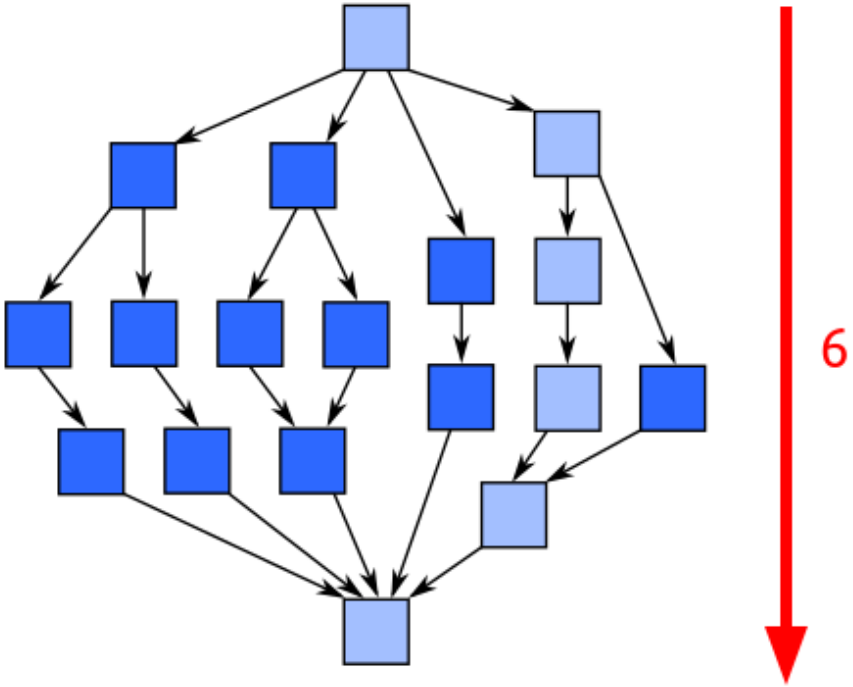


Figure 2: Ejemplo *Work-Span* con 18 tareas

Memoria compartida, localidad y caché (1)

- La **memoria compartida** y la **localidad** son conceptos que ayudan a reducir los tiempos de ejecución
- En este enlace se presentan conceptos relacionados a:
  - Localidad temporal
  - Localidad espacial

Memoria compartida, localidad y caché (2)

- Código:
  - En este enlace se lleva a cabo una práctica donde se evidencia el efecto de **fallos de acceso a memoria** si se recorre una matriz o por filas o columnas
  - En este otro enlace se muestra como los tiempos de acceso a una misma región de memoria disminuyen de manera sustancial los **tiempos de acceso a memoria**
    - En este último programa determine los números de fallos de página de los ciclos 2 y 3.

Algunas anotaciones respecto a `valgrind`

- Valgrind cuando se usa con la opción `--tool=cachegrind` brinda información respecto al uso de la caché
  - `I1` lectura de **instrucciones** en caché `L1`.
    - `I1mr` caché `L1` *read misses*
    - `ILmr` caché `LL` (*last level*) *read misses*
  - `Dr` lectura de **datos** en caché
    - `D1mr` caché `D1` *read misses*
    - `DLmr` caché `DL` (*last level*) *read misses*
  - `Dw` **escritura** de **datos** en caché
    - `D1mw` caché `D1` *write misses*
    - `DLmw` caché `DL` (*last level*) *write misses*
- Para obtener el detalle línea por línea de la ejecución de un programa `cg_annotate --auto=yes cachegrind.out.<pid>`

Rendimiento del caché

- Los accesos de la CPU a memoria se dividen en $Hit + Miss$
  - La tasa de acierto $HitRatio = h = \frac{Hit}{Hit+Miss}$
  - La tasa de fallo $MissRatio = 1 - HitRatio = \frac{Miss}{Hit+Miss}$
- El tiempo promedio de acceso a memoria (T_c tiempo de acceso a caché, T_m tiempo de acceso a RAM)
  - Cuando el acceso es simultáneo $T_{avg} = h * T_c + (1 - h) * T_m$
  - Cuando el acceso es jerárquico $T_{avg} = h * T_c + (1 - h) * (T_c + T_m)$

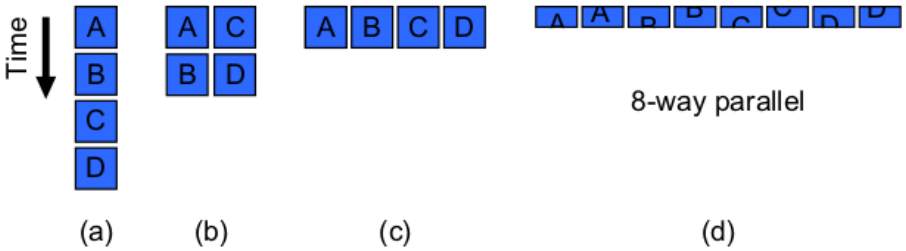
Un caso especial de paralelismo

- Suponga una tarea donde las tareas a ejecutar son totalmente independientes las unas de las otras
- El ***span*** de ese algoritmo está dado por el tiempo de aquella secuencia de instrucciones/tareas que tome más tiempo en ejecutarse

Ejemplo

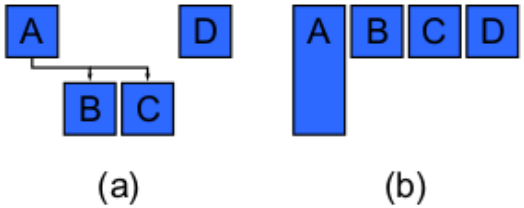
```
for (int i = 0; i < MAX_VECTOR; i++) // operacion
  vector[i] = fibonacci(i);
```

Varias tareas, diferentes tamaños e interdependencias (1)



a. Un solo procesador
b. Dos procesadores
c. Cuatro procesadores
d. 8 procesadores y tarea arbitrariamente fragmentable

Varias tareas, diferentes tamaños e interdependencias (2)



a. C y B no se pueden ejecutar hasta que A no termine
b. A toma más tiempo en ejecutarse que las demás. El tiempo de ejecución de las 4 tareas es el tiempo de ejecución de la **tarea que más tiempo toma**

Ejemplo

Visitar este Google Colab para ver una ejecución irregular

Dos desafíos de la programación en paralelo

- Descomposición de la computación
- Gestión de la comunicación

Construir una casa desde cero y la ley de Amdahl (1)

- Imagine los pasos que toma llevar a cabo construir una case desde cero

Figure 3: Diseñar

Figure 4: Construir

Construir una casa desde cero y la ley de Amdahl (2)

- Hay algunos pasos que por muchas manos que se involucren en la elaboración no se pueden paralelizar ejemplo:
  - No se puede construir la casa si no hay planos
  - Puedo tener 5 maestros de construcción y si los diseños no están listos estos no pueden hacer nada

Construir una casa desde cero y la ley de Amdahl (2)

- Gene Amdahl estableció una ley que define un límite inferior el cual una tarea por más elementos de procesamiento que se le involucren, no podrá reducir su tiempo de cómputo

"*the overall performance improvement gained by optimizing a single part of a system is* ***limited*** *by the fraction of time that* ***the improved part is actually used***"

$$S_{latency}(s) = \frac{T}{T(s)} = \frac{T}{(1-p)T + \frac{p}{s}T} = \frac{1}{(1-p) + \frac{p}{s}}$$

donde:

- s el factor de paralelización,
- p representa la proporción de código **paralelizable**

Ejemplos de uso de la ley de Amdahl (1)

- Suponga que el $30\%(0.3)$ de un programa está sujeto a speedup y la mejora en la ejecución es del doble ($s = 2$).

$$S_{latency} = \frac{1}{1-p+\frac{p}{s}} = \frac{1}{1-0.3+\frac{0.3}{2}}$$

- Suponga una tarea serial constituida de 4 partes consecutivas con porcentajes de tiempo de ejecución: $p_1 = 0.11$, $p_2 = 0.18$, $p_3 = 0.23$, y $p_4 = 0.48$. El *speedup* de cada tarea es $s_1 = 1$, $s_2 = 5$, $s_3 = 20$, y $s_4 = 1.6$.

$$S_{latency} = \frac{1}{\frac{p_1}{s_1}+\frac{p_2}{s_2}+\frac{p_3}{s_3}+\frac{p_4}{s_4}} = \frac{1}{\frac{0.11}{1}+\frac{0.18}{5}+\frac{0.23}{20}+\frac{0.48}{1.6}}$$

Ejemplos de uso de la ley de Amdahl (2)

- Suponga un programa que está constituido por dos partes A y B donde $T_A = 3$ y $T_B = 1$.
- Si la parte B se logra ejecutar 5 veces más rápido el *speedup* es:

$$S_{latency} = \frac{1}{1-0.25+\frac{0.25}{5}}$$

- Si la parte A se logra ejecutar 2 veces más rápido el *speedup* es:

$$S_{latency} = \frac{1}{1-0.75+\frac{0.75}{2}}$$

- **MORALEJA** se debe ser estratégico a la hora de decidir donde invertir tu esfuerzo para paralelizar

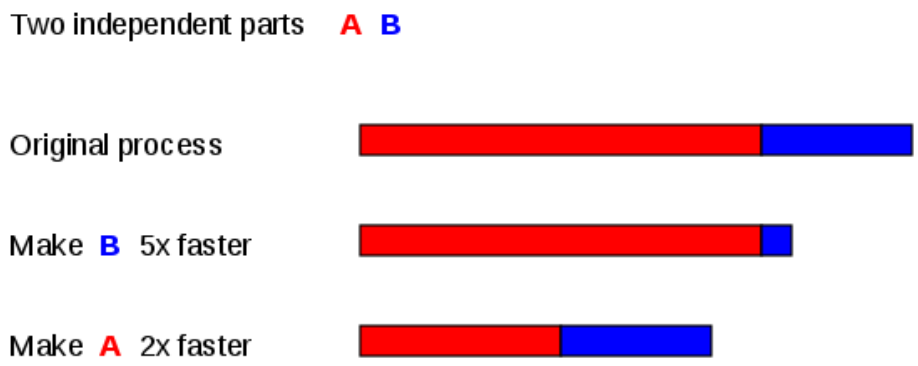


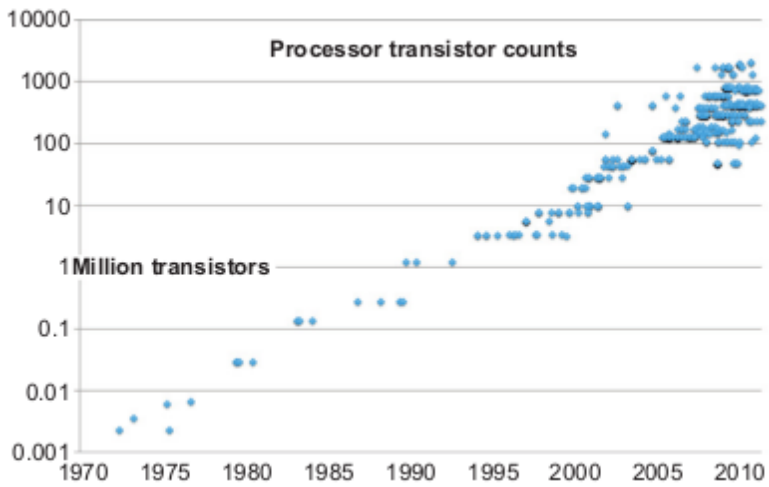
Figure 5: Programa constituido por A y B

Importante

- Elementos a tener en cuenta a la hora de paralelizar:
  - La cantidad total de **trabajo computacional** (hacer *profiling*)
  - El *span*
  - La cantidad total de **comunicación**

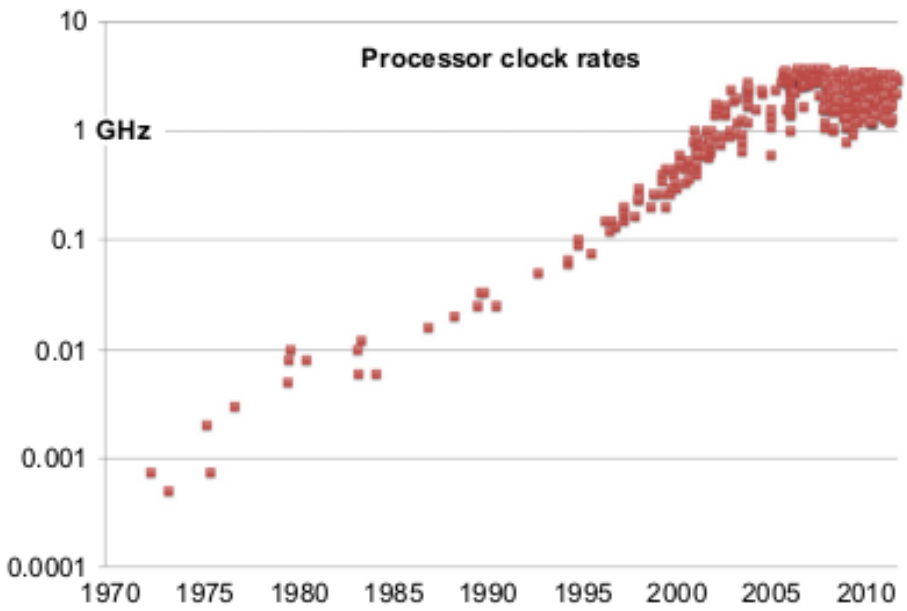
Nacimiento del paralelismo

Cantidad de transistores por procesador



- Ley de Moore establecía un crecimiento del doble en el número de transistores en los procesadores cada dos años
- Se experimentó un crecimiento de **6 ordenes de magnitud** en 40 años (10^3 transistores en 1971 → 10^9 transistores en 2011)

Frecuencia de reloj en el procesador



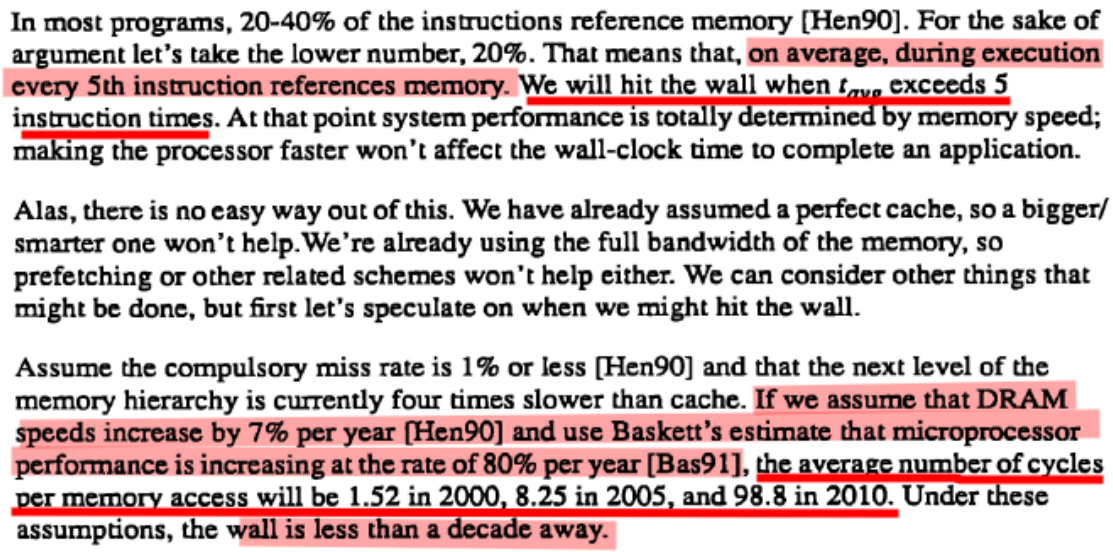
- Las tasas de reloj venían incrementando (1973 - 2003) pero ahora se han estabilizado alrededor de los 3 GHz
- La frecuencia de reloj se “estabilizó” debido a:
  - *Power wall* consumo exagerado de electricidad de parte del procesador
  - *Memory wall* discrepancias entre las velocidades del reloj y las velocidades de memoria
  - *Instruction Level Parallelism (ILP) wall* límites al paralelismo de bajo nivel

Power Wall

- La tasa de consumo de energía no crece a la misma tasa del reloj del computador

Memory Wall

- Las tasas de acceso a memoria no crecen tan rápido como la tasa de procesamiento de la CPU (**cuello de botella**)
- Han habido avances pero no son suficientes (e.g. *double data rate* (DDR) *signaling*)
- Limitaciones: **bandwidth** (tasa de transmisión) y la **latencia** (demora *natural* de la transmisión de la información)
- **Los algoritmos deben reducir al máximo los accesos a memoria**



In most programs, 20-40% of the instructions reference memory [Hen90]. For the sake of argument let's take the lower number, 20%. That means that, on average, during execution every 5th instruction references memory. We will hit the wall when t_{avg} exceeds 5 instruction times. At that point system performance is totally determined by memory speed; making the processor faster won't affect the wall-clock time to complete an application.

Alas, there is no easy way out of this. We have already assumed a perfect cache, so a bigger/ smarter one won't help.We're already using the full bandwidth of the memory, so prefetching or other related schemes won't help either. We can consider other things that might be done, but first let's speculate on when we might hit the wall.

Assume the compulsory miss rate is 1% or less [Hen90] and that the next level of the memory hierarchy is currently four times slower than cache. If we assume that DRAM speeds increase by 7% per year [Hen90] and use Baskett's estimate that microprocessor performance is increasing at the rate of 80% per year [Bas91], the average number of cycles per memory access will be 1.52 in 2000, 8.25 in 2005, and 98.8 in 2010. Under these assumptions, the wall is less than a decade away.

Figure 6: Wulf W.A., McKee, S.A.; Hitting the memory wall..., December 1994

ILP Wall

- El hardware es naturalmente paralelo y los procesadores son capaces de ejecutar en paralelo instrucciones que son independientes → **superscalar instruction issue**
- Técnicas para aprovechar la naturaleza paralela de los procesadores
  - **speculative execution** es acerca de ejecutar instrucciones que se suponen se ejecutarán en el futuro
  - **pipelining** es otra técnica para paralelizar y consiste en dividir (hasta en 10 etapas) la ejecución de una instrucción enlace 1, enlace 2

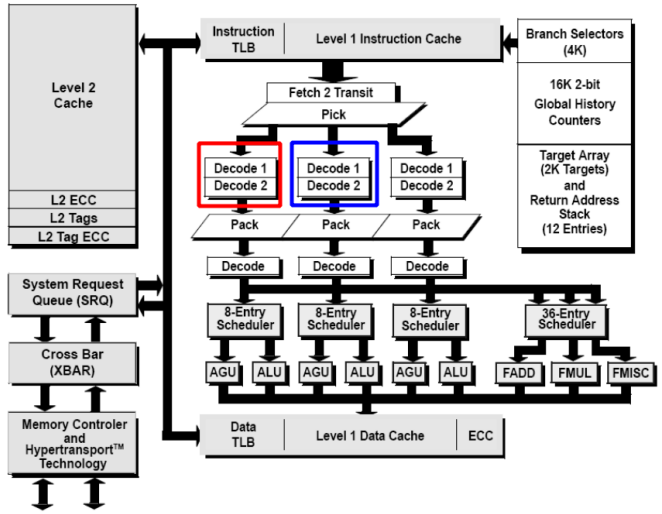


Figure 7: AMD Athlon 64 Processor

```
if (cond) {
  inst_cond_1
  inst_cond_2
  inst_cond_3
} else {
  inst_else_1
  inst_else_2
  inst_else_3
}
```

F1

F2

Figure 8: Ejecución especulativa

Pipelining en gráficas

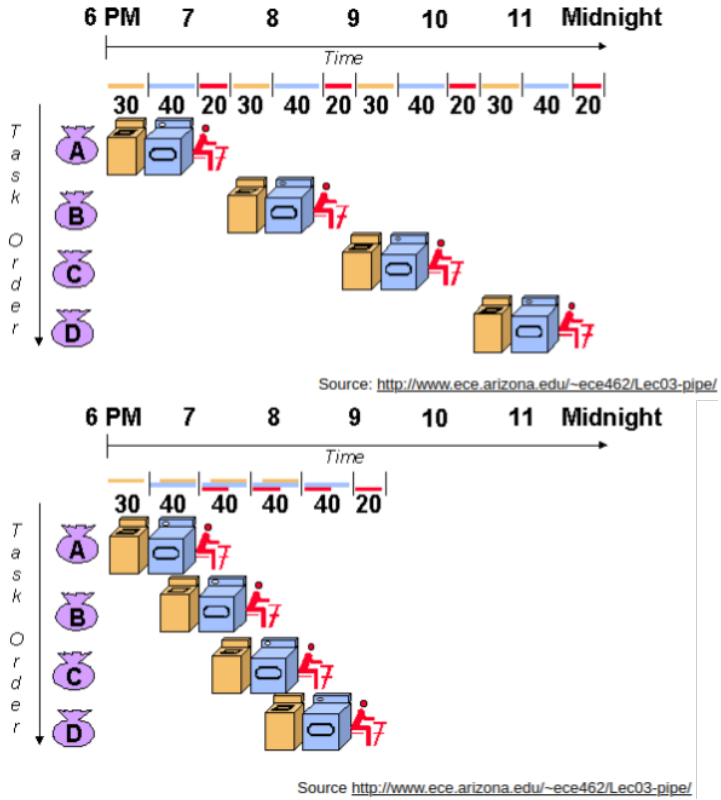


Figure 9: Etapas del lavado de ropa

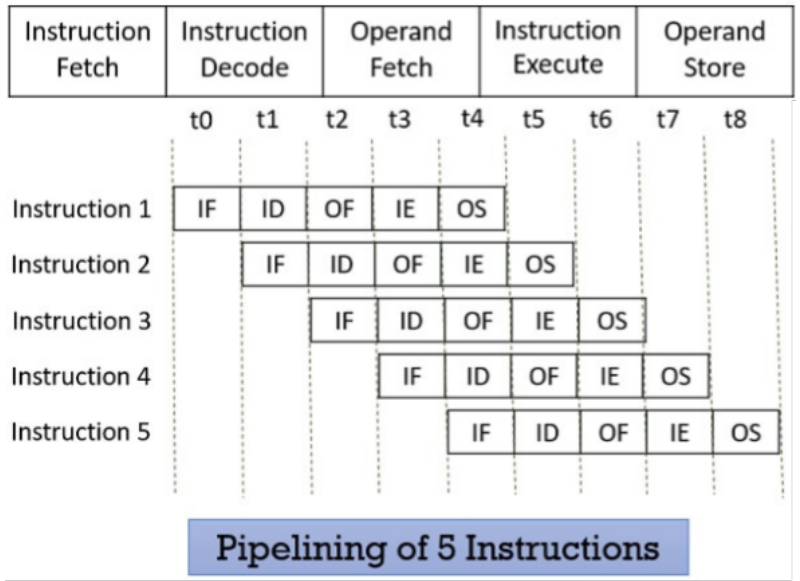


Figure 10: Etapas en la ejecución de instrucciones

- Ya se ha llegado al límite en el cual estas y otras técnicas pueden aprovechar la naturaleza paralela de los computadores, Listado de microarquitecturas de Intel

Palabras finales respecto a los **Walls**

- La velocidad de un algoritmo no puede depender de una **alta tasa de reloj** ya que genera “grandes” consumos de energía
- Paralelismo interno del hardware ya ha llegado a sus límites (a.k.a. ILP)
- Se deben escribir programas explícitamente paralelos PERO que **reduzcan sus accesos a memoria**
- Los procesadores tienen cada vez más rutinas paralelizadas pero **los programadores deben usarlas** (e.g. operaciones super-escalares)
  Enlace

The free lunch is over

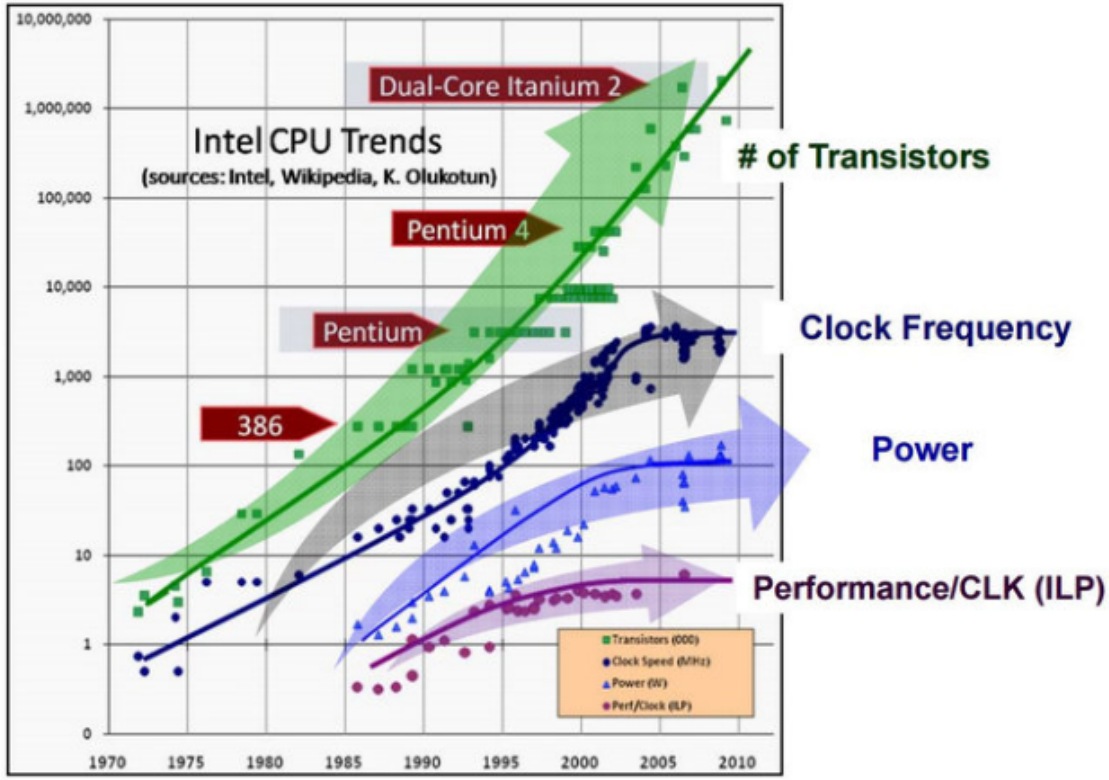


Figure 11: The free lunch of CMOS scaling is over

Los patrones de software en paralelismo

- Los patrones nacen de estructurar las “soluciones mejor conocidas” a ciertos problemas y se empaquetan para ser usadas en aplicaciones [*paralelas*] eficientes
- Los patrones ofrecen un **vocabulario** que permite la fácil comunicación de algoritmos y nuevos diseños de aplicaciones
- Los patrones nos permiten crear **soluciones de alto nivel** a problemas complejos

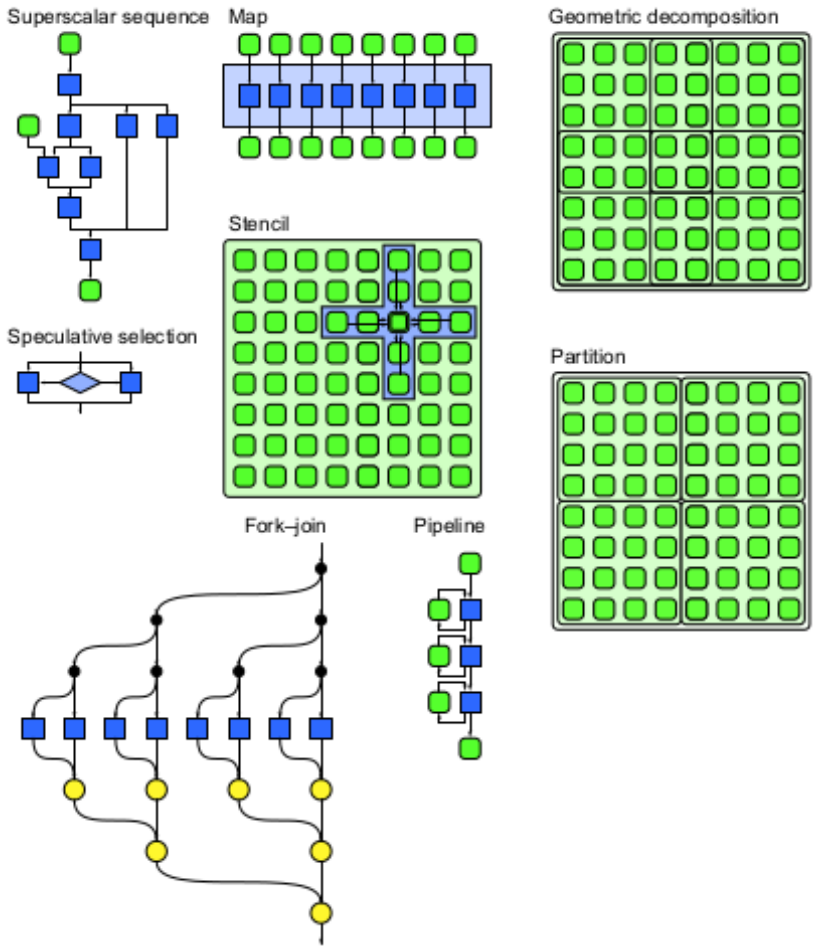


Figure 12: Algunos patrones paralelos

¿Qué significa **soluciones de alto nivel**?

- Las soluciones no se enfocan tanto en las minucias de programación (e.g. librerías de *threads*) sino en la algorítmia de la solución
- Las soluciones en este nivel **no se enfocan demasiado** en los detalles del hardware
- El objetivo a la hora de crear aplicaciones paralelas es enfocarse en soluciones de software **escalables** capaces de responder en entornos de dos, cuatro o decenas de núcleos (e.g. 32 o 64) con poca o ninguna modificación

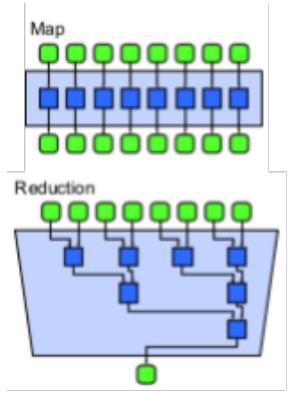


Figure 13: Patrón MapReduce

Necesidad por paralelización explícita

- Algunas veces los compiladores no son buenos identificando oportunidades para la paralelización o pueden sugerir erróneamente zonas para paralelizar

Paralelización automática no es tan sencilla

- Suma de dos vectores, **paralelizable?**

```
void addme(int n,
  double a[n],
  double b[n],
  double c[n]) {
  int i;
  for (i = 0; i < n; i++)
    a[i] = b[i] + c[i];
}
```

Paralelización automática no es tan sencilla

- Invocación de la función addme

```
double a[10];
a[0] = 1;
addme(9, a + 1, a, a);
```