

# Análisis del rendimiento

Metodología de la Programación Paralela

Jesús Sánchez Cuadrado (jesusc@um.es)

Curso 2020/21

# Introducción

- Los procesadores paralelos se usan para acelerar la resolución de problemas de alto coste computacional.
- La finalidad principal consiste en reducir el tiempo de ejecución:
  - Si el tiempo secuencial es  $t(n)$
  - Con  $p$  procesadores se intenta reducir el tiempo a  $t(n) / p$
- En esta sesión, ver algunas medidas que dan idea de la “bondad” de un algoritmo paralelo.

# Introducción

- El tiempo de ejecución de un programa secuencial depende de:
  - tamaño de la entrada
  - compilador
  - máquina
  - programador...
- Si nos olvidamos de valores constantes dependientes del sistema (por ejemplo, el coste de una operación aritmética en la máquina donde ejecutamos el programa) podemos considerar el tiempo en función del tamaño de la entrada:  $t(n)$ .
- Es el tiempo desde que empieza la ejecución del programa hasta que acaba.

# Modelado del tiempo de ejecución paralelo

- Aproximación del tiempo de ejecución
  - Teje = Suma de los tiempos de ejecución de los bloques de computación
  - Tcom = Suma de los tiempos usados en comunicación o sincronización
  - Toverhead = Tiempo de ejecución debido a la sobrecarga
  - Toverlapping = Tiempo en que la comunicación y la computación están solapados

$$t(n, p) = \underbrace{t_{eje}(n, p) + t_{com}(n, p)} + \underbrace{t_{overhead}(n, p) - t_{overlapping}(n, p)}$$

En la práctica usaremos esta fórmula como aproximación

Trataremos de:

- Minimizar el tiempo de overhead.
- Maximizar el solapamiento.

# Medidas de prestaciones

# Speed-up

- La mejora de un programa paralelo con respecto al secuencial

$$Speedup = \frac{Tiempo\ secuencial}{Tiempo\ paralelo} = \frac{T(n)}{T(n, p)}$$

- Tres tipos de costes:
  - Cálculos que se realizan de manera secuencial ( $T_{sec}$ )
  - Cálculos que se pueden realizar de manera paralela ( $T_{par}$ )
  - Sobrecarga por el paralelismo ( $T_{overhead}$ )

$$Speedup(n, p) \leq \frac{T_{sec}(n) + T_{par}(n)}{T_{sec}(n) + \frac{T_{par}(n)}{p} + T_{overhead}(n, p)}$$

# Speed-up

- ¿Qué programa secuencial elegir?
  - El del mejor algoritmo secuencial que resuelve el problema, pero cuál es el “mejor algoritmo secuencial” depende del tamaño de la entrada, distribución de los datos, máquina..., y puede no conocerse (Por ejemplo, ¿cuál es el coste de la multiplicación de matrices?)
  - El tiempo del mejor algoritmo secuencial conocido. Pero depende de los mismos parámetros. Para ordenación sería  $O(n \cdot \log(n))$ , pero en otros problemas no se conoce cuál es el mejor (e.g., multiplicación de matrices).
  - Tiempo de ejecución en sólo un procesador del algoritmo paralelo. Pero puede que el mejor esquema para paralelizar no sea bueno en secuencial.
  - El tiempo de ejecución de un algoritmo secuencial “razonablemente bueno”. Habrá que indicar cuál se usa.

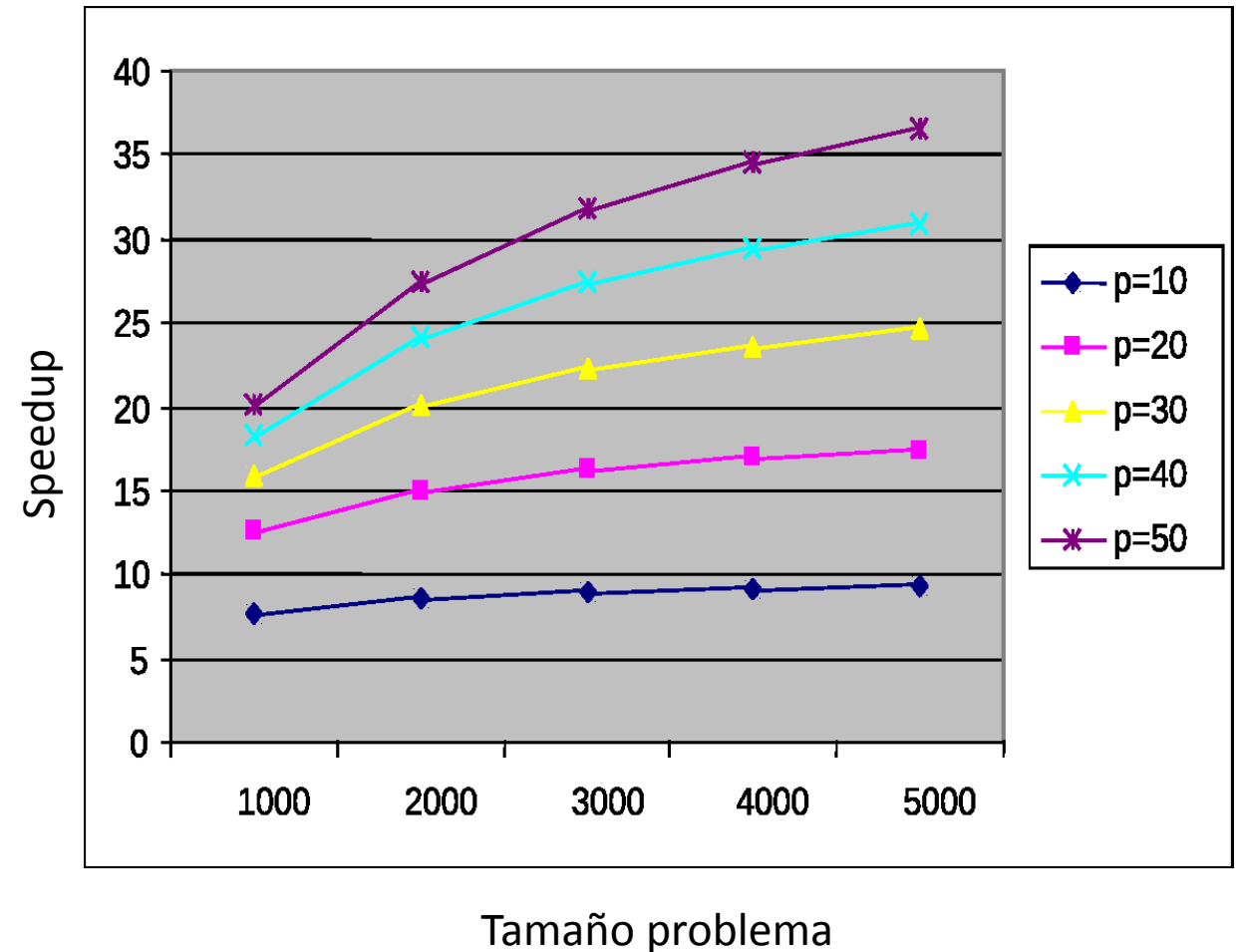
# Speed-up

- El speed-up será menor que el número de procesadores. Si no, podría hacerse un algoritmo secuencial mejor que el que se usa, simulando el algoritmo paralelo.
- En algunos casos puede haber speed-up superlineal por mejor gestión de la memoria al usar más procesadores.



# Speed-up

- Para un tamaño de problema fijo se aleja del valor óptimo al aumentar el número de procesadores.
- Para un número de procesadores fijo aumenta al aumentar el tamaño del problema.



# Eficiencia

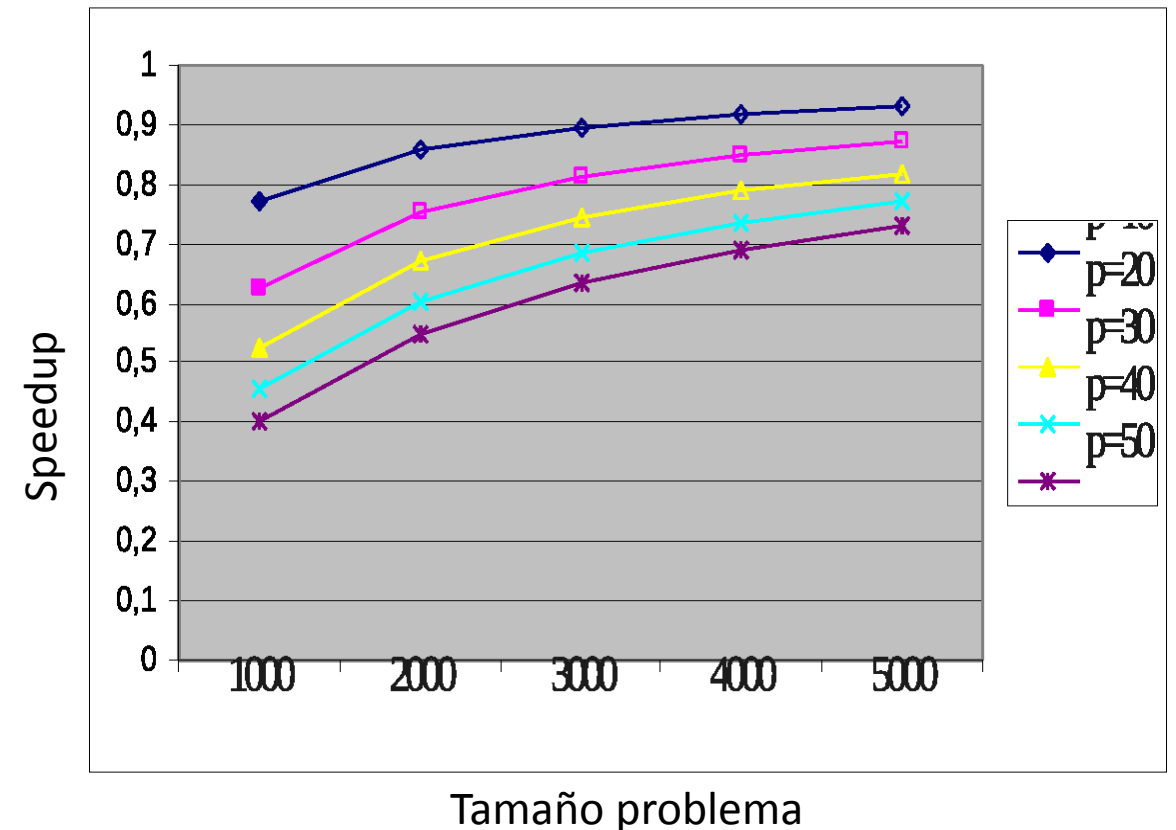
- Da una idea de la porción de tiempo que los procesadores se dedican a trabajo útil

$$Eficiencia(n, p) = \frac{Speedup(n, p)}{p} = \frac{t(n)}{p \cdot t(n, p)}$$

- Valor entre 0 y 1
  - Un valor de 1 es perfectamente eficiente
  - Empíricamente  $> 1$  si hay un speedup superlineal

# Eficiencia

- Para un tamaño de problema fijo se aleja del valor óptimo (1) al aumentar el número de procesadores.
- Para un número de procesadores fijo aumenta con el tamaño del problema.



# Ley de Amdahl

- El speedup que se puede conseguir está limitado por la fracción del programa que es inherentemente secuencial
- El tiempo de ejecución de un programa se puede clasificar en:
  - Tiempo dedicado a trabajo **inherentemente secuencial** (no se puede paralelizar)
  - Tiempo dedicado a trabajo **paralelizable** (podríamos paralelizarlo)
  - Sobrecarga añadida si realizamos la paralelización

$$Speedup(n, p) \leq \frac{Tsec(n) + Tpar(n)}{Tsec(n) + \frac{Tpar(n)}{p} + Sobrecarga} \leq \frac{Tsec(n) + Tpar(n)}{Tsec(n) + \frac{Tpar(n)}{p}}$$

Ignoramos la sobrecarga

# Ley de Amdahl

- La fracción del programa que es inherentemente secuencial es:

$$f = \frac{T_{sec}(n)}{T_{sec}(n) + T_{par}(n)}$$

- La fórmula original del speedup puede verse como:

$$Speedup(n, p) \leq \frac{1}{f + \frac{1-f}{p}}$$

# Ley de Amdahl

- Desarrollo

$$T_{sec}(n) + T_{par}(n) = \frac{T_{sec}(n)}{f}$$

$$T_{par}(n) = \frac{T_{sec}(n)}{f} - T_{sec}(n)$$

$$Speedup(n, p) \leq \frac{T_{sec}(n) + T_{par}(n)}{T_{sec}(n) + \frac{T_{par}(n)}{p}} = \frac{\frac{T_{sec}(n)}{f}}{T_{sec}(n) + \frac{\frac{T_{sec}(n)}{f} - T_{sec}(n)}{p}}$$

$$\frac{\frac{\frac{T_{sec}(n)}{f}}{\frac{T_{sec}(n)}{f} - T_{sec}(n)}}{T_{sec}(n) + \frac{\frac{T_{sec}(n)}{f} - T_{sec}(n)}{p}} = \frac{\frac{T_{sec}(n)}{f}}{T_{sec}(n) + T_{sec}(n) \cdot \frac{\frac{1}{f} - 1}{p}} = \frac{\frac{1}{f}}{1 + \frac{\frac{1}{f} - 1}{p}} = \frac{1}{f + \frac{1-f}{p}}$$

# Ley de Amdahl

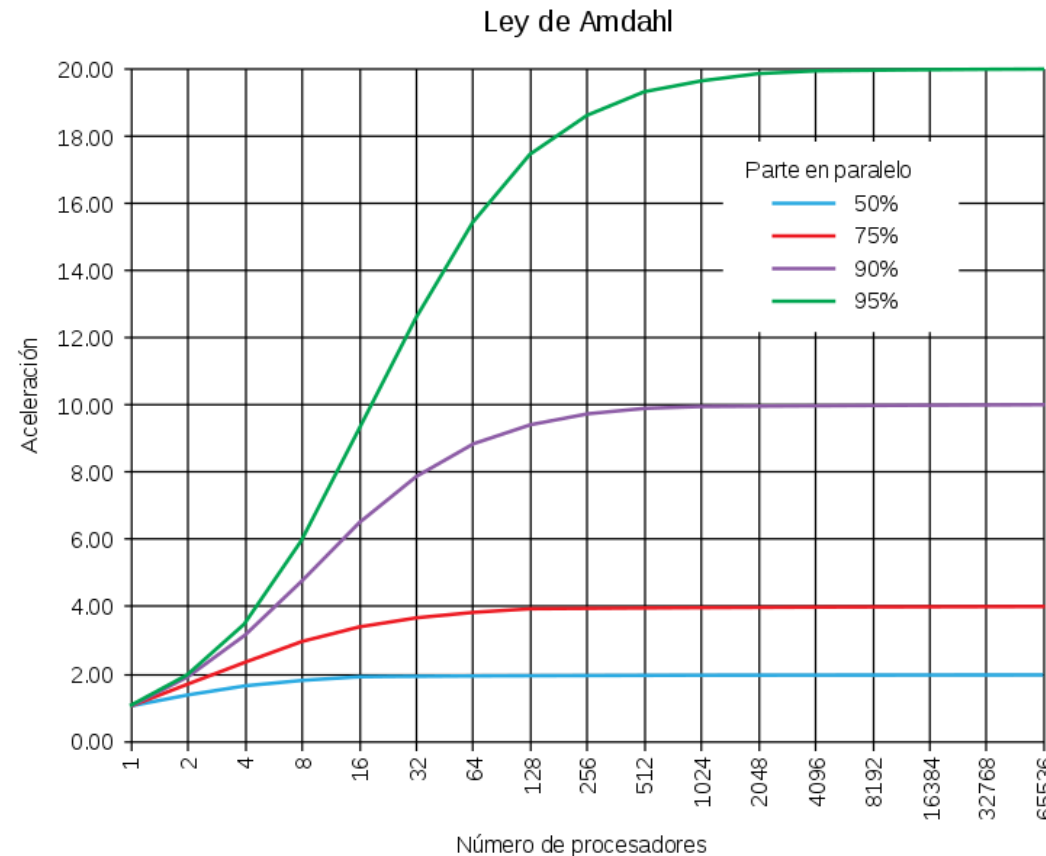
- Consecuencias
  - Ofrece una forma de determinar una cota para el speedup
  - Si no podemos paralelizar todo el programa, el speedup va a estar limitado
- Ejemplo:
  - Dado un programa. Se ejecutan diferentes *benchmarks* que revelan que el 90% del tiempo de ejecución es en partes que pueden ser paralelizadas, y el 10% en partes que no pueden serlo. ¿Cuál es el máximo speedup con 8 procesadores?

$$Speedup(n, 8) \leq \frac{1}{f + \frac{1-f}{p}} = \frac{1}{0.1 + \frac{1-0.1}{8}} = 4.7$$

- La Ley de Amdahl, asume que queremos resolver un problema de tamaño fijo lo más rápido posible.

# Ley de Amdahl

- Si sólo podemos paralelizar una parte del programa, el speedup se va a ver severamente limitado.



\* Fuente: wikipedia



# Ley de Amdahl

¿Cómo podemos estimar las partes  
secuencia y paralelas de un programa?

# Escalabilidad

- Para un sistema paralelo interesa que las prestaciones se sigan manteniendo en cierta medida al aumentar el tamaño del sistema.
- No se puede seguir manteniendo las prestaciones con el tamaño del problema fijo y aumentando el número de elementos de proceso.
- Se trata de mantener las prestaciones al aumentar el tamaño del sistema pero aumentando también el tamaño del problema. Los sistemas (físicos o lógicos) que mantienen las prestaciones en esas condiciones se dice que son escalables.
- La idea es, para un determinado programa, ser capaces de determinar si en el futuro, cuando aumente el número de elementos de proceso y se quiera resolver problemas mayores, se seguirán manteniendo las prestaciones.

# Speedup de tiempo fijo

- La Ley de Amhdal asume un tamaño fijo del problema
- La Ley de Gustafson-Barsis asume que el tiempo de ejecución es fijo y se desea determinar cuánto puede escalar el tamaño del problema
- Útil cuando estamos interesados en aumentar la precisión del cálculo al aumentar el número de procesadores

$$Speedup(n) \leq p + \alpha(1 - p) = p - \alpha(p - 1)$$

# Ley de Gustafson-Barsis

- Tiempo de ejecución de un programa paralelo puede descomponerse en  $T = a + b$  donde
  - $a$  = tiempo de ejecución secuencial
  - $b$  = tiempo de ejecución en paralelo en cada procesador
- El tiempo de ejecución de ese programa si lo ejecutáramos en secuencial
  - $a + p \cdot b$
  - La fracción secuencial se mantiene constante (con respecto al programa paralelo)

$$Speedup = \frac{a + p \cdot b}{a + b} = \alpha + p \cdot (1 - \alpha) = p - \alpha \cdot (p - 1)$$

Porción secuencial  $\longrightarrow \alpha = \frac{a}{a + b}$

# Speedup de tiempo fijo

- Ejemplo
  - Una aplicación se ejecuta en 64 procesadores y tarda 220 segundos en ejecutarse. El 5% del tiempo se dedica a ejecución secuencial.
  - El speedup escalado es:

$$s = 64 + (1 - 64) \cdot (0.05) = 64 - 3.15 = 60.85$$

# Estudio experimental

- Diseño de experimentos que permitan obtener conclusiones:
  - Determinar qué tipo de conclusiones se quiere obtener: en cuanto a speed-up, escalabilidad...
  - Definir el rango de los experimentos: número de elementos de proceso y tamaño del problema.
  - Comparar los resultados experimentales con los teóricos. Si no coinciden, revisar los estudios teóricos o la forma en que se han hecho los experimentos.
- Pueden consistir en:
  - Estudiar el programa completo.
  - Estudiar el comportamiento en los distintos elementos de proceso: balanceo de la carga, sincronización...
  - Dividir el programa en partes para estudiarlas por separado.

# Estudio experimental

- Se pueden estimar los valores constantes en las fórmulas teóricas:
  - $t_s$  y  $t_w$  se pueden estimar con un ping-pong, o pueden tener valores distintos para distintas operaciones de comunicación.
- El coste de la computación se puede estimar ejecutando parte del programa en un procesador.
- Ajuste del modelo teórico con datos experimentales:
  - Obtener valores constantes del estudio teórico con ajuste por mínimos cuadrados con resultados experimentales.
  - Estudio asintótico: comportamiento teniendo en cuenta el término de mayor orden de la fórmula teórica, compararlo con la gráfica experimental.

# Estudio experimental

- Configuración hardware
- Configuración software (ej., opciones de compilación)
- Speedup

- Fijar entrada

| Hilos      | Tiempo | Speedup |
|------------|--------|---------|
| Secuencial | 20.50  | -       |
| 2          | 10.25  | 2.00    |
| 3          | 7.50   | 2.73    |
| 4          | 6.5    | 3.15    |

Pueden ser necesarias varias ejecuciones (ej., JVM)

- Escalabilidad

- Fijar hilos

| Entrada | Tiempo | Speedup |
|---------|--------|---------|
| 1000    | 7.50   | 3.15    |
| 2000    | 6.30   | 3.25    |
| 3000    | 5.95   | 3.45    |



# Causas de reducción de las prestaciones

# Posibles causas de reducción de las prestaciones

- Contención de memoria
  - En memoria compartida el acceso a datos comunes o que están en el mismo bloque de memoria produce contención. Si los datos son de lectura puede no haber ese problema.
  - En el ejemplo los procesadores escriben en zonas distintas. No hay problema de coherencia, pero puede haber de contención si hay datos en los mismos bloques de memoria (false sharing).
- Código secuencial
  - Puede haber parte imposible de paralelizar, como la I/O.
  - Por ejemplo, la inicialización de variables. Además, si los datos están inicialmente en un procesador hay que difundirlos.

# Posibles causas de reducción de las prestaciones

- Computación extra
  - Si se obtiene un programa paralelo a partir de secuencial, son necesarias computaciones adicionales por:
    - uso de variables de control,
    - comprobación de identificadores de proceso,
    - cálculos adicionales o comunicaciones para obtener datos
    - calculados por otro procesador...
- Tiempo de sincronización
  - Cuando un proceso tiene que esperar a que estén disponibles datos procesados por otro.

# Posibles causas de reducción de las prestaciones

- Desbalanceo de la carga
  - El volumen total de la computación no se distribuye por igual entre todos los procesadores
  - En el ejemplo, en el primer paso trabajan todos pero en los pasos siguientes va disminuyendo el número de procesadores que trabajan,
  - También es posible que partamos de una entrada que no se puede balancear, por ejemplo si tenemos 12 datos para 8 procesadores,
  - O que el volumen de datos varíe a lo largo de la ejecución, con lo que se necesitaría balanceo dinámico.

# Herramientas de análisis

# Proceso de análisis

1. Construir primera versión
2. Preparar datos de prueba
  - Tamaños diferentes
  - Más pequeños al principio, más grandes para estudiar escalabilidad
3. Ejecutar pruebas de rendimiento
  - ¿Es adecuado el rendimiento?
  - Identificar cuellos de botella – Usar herramientas
  - Refactorizar el programa
  - Volver al punto 3

# Herramientas de análisis – Profilers

- Un **profiler** es una herramienta de análisis que captura eventos sobre la ejecución del programa para analizar posibles cuellos de botella
- Ejemplos de profilers
  - gprof
  - perf
  - callgrind / KCacheGrind
  - VisualVM

# Profilers

- Monitorización a nivel hardware
  - Uso de los contadores de los procesadores
  - Preciso pero la información es global
  - Poco configurable
  - Ejemplo: perf
- Muestreo (Sampling)
  - Se interrumpe el procesador cada cierto tiempo para consultar su estado
  - Predice qué instrucciones se ejecutan más frecuentemente
  - Información estadística, menos precisa
- Instrumentación
  - El código (fuente o binario) se modifica para insertar rutinas que capturan los eventos de interés
  - Ejemplo: gprof



# False sharing

- Memorias cache
  - Cache hit: cuando la CPU necesita un valor de memoria y el valor está en la cache, lo puede recuperar rápidamente
  - Cache miss: Cuando la CPU necesita un valor pero no está en la cache y debe ir a buscarlo a memoria
  - Los memoria cache se organiza en líneas o bloques, típicamente de 64 bytes

# False sharing

- Coherencia espacial
  - Si el programa usa una dirección de memoria ahora, posiblemente necesita pronto los valores de las direcciones de memoria cercanas
- Coherencia temporal
  - Si el programa está usando una dirección de memoria ahora, posiblemente la volverá a utilizar pronto

Si se cumplen estas reglas => Cache hit ++  
Si no => Cache miss ++

# False sharing

cache-miss-row.c

cache-miss-col.c

- ¿Cuál va más rápido?
  - Complejidad igual –  $O(n^2)$

```
double array[NUM];

for( int i = 0; i < NUM; i++ ){
    for( int j = 0; j < NUM; j++ ){
        sum += array[ i ][ j ];
        // acceso por filas
    }
}
```

```
double array[NUM];

for( int i = 0; i < NUM; i++ ){
    for( int j = 0; j < NUM; j++ ){
        sum += array[ j ][ i ];
        // acceso por columnas
    }
}
```

# False sharing

- Herramientas para analizar fallos de cache

```
$ valgrind --tool=cachegrind ./mi_programa
```

```
$ sudo perf stat -d ./mi_programa
```

\* <http://www.brendangregg.com/perf.html>

# False sharing

- Cachegrind
  - Simula cómo interacciona el programa con la memoria cache y con el predictor de saltos
  - Asume una arquitectura con dos niveles de cache, L1 y LL.
    - Si la arquitectura tiene más niveles sólo se analiza el último
  - Por desgracia no funciona bien con programas multi-hilo
    - Usar perf
- I => Instruction
- D => Data

<http://valgrind.org/docs/manual/cg-manual.html>

### cache-miss-row.c

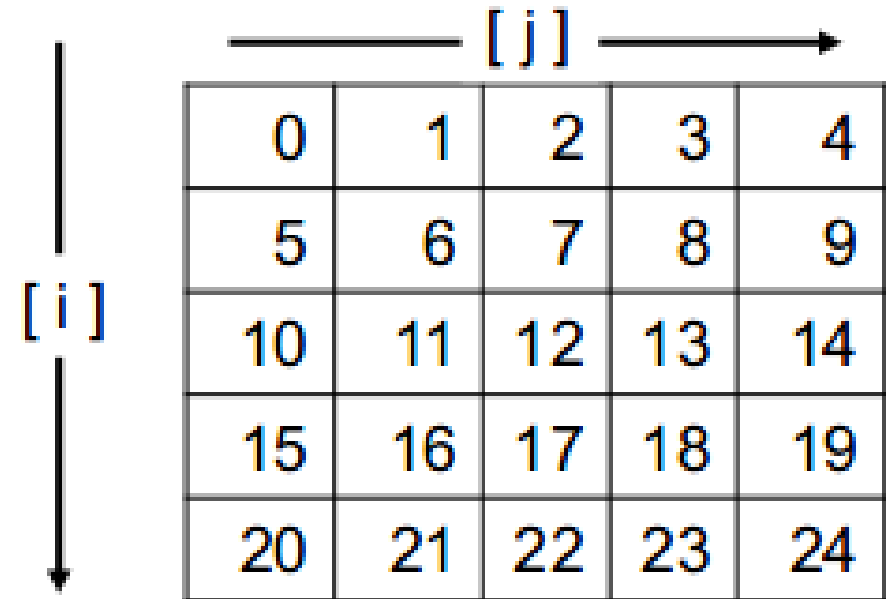
```
==23156== D    refs:          700,133,994  (600,101,603 rd    + 100,032,391 wr)
==23156== D1   misses:         6,254,292  (  6,253,488 rd    +           804 wr)
==23156== LLd misses:         6,253,340  (  6,252,605 rd    +           735 wr)
==23156== D1   miss rate:           0.9% (           1.0%    +           0.0% )
==23156== LLd miss rate:           0.9% (           1.0%    +           0.0% )
```

### cache-miss-col.c

```
==26047== D    refs:          700,133,998  (600,101,605 rd    + 100,032,393 wr)
==26047== D1   misses:       100,004,290  (100,003,482 rd    +           808 wr)
==26047== LLd misses:         6,263,341  (  6,262,604 rd    +           737 wr)
==26047== D1   miss rate:        14.3% (        16.7%    +           0.0% )
==26047== LLd miss rate:           0.9% (           1.0%    +           0.0% )
```

# False sharing

- Al recorrer por columnas estamos “sacando” de la memoria cache líneas que vamos a necesitar en otras iteraciones
- Es mejor recorrer en la dirección de las líneas de cache

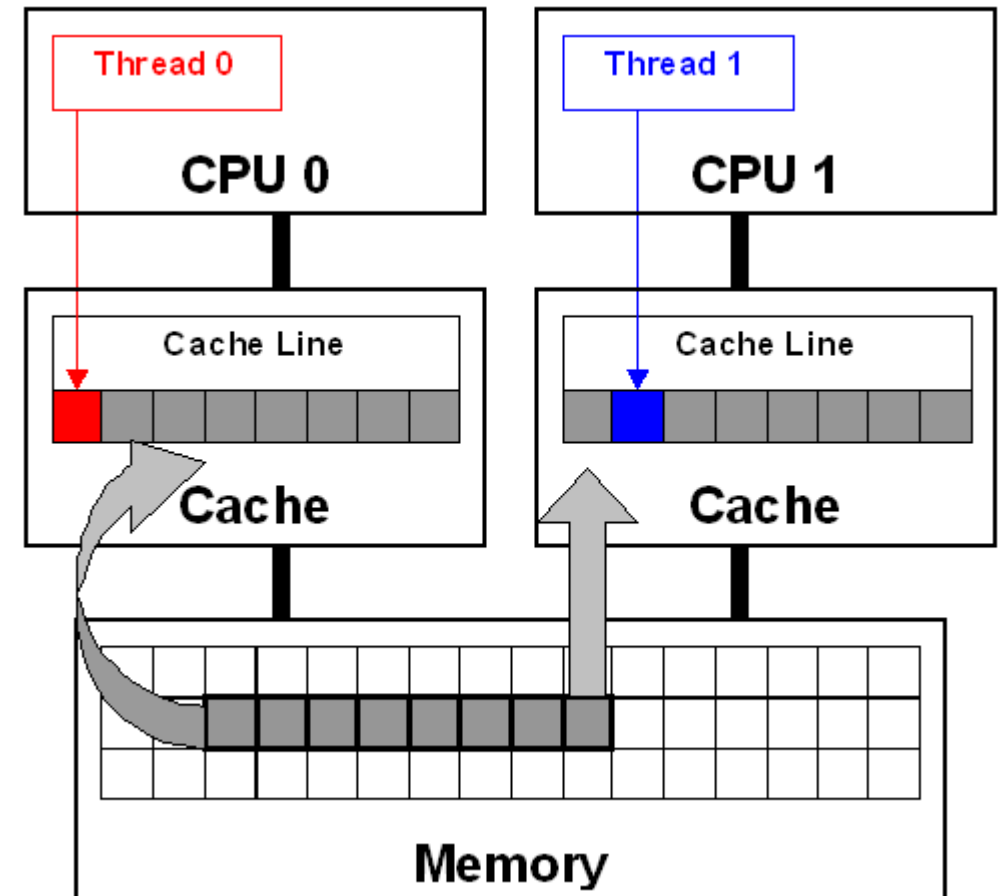


A 5x5 grid representing a memory array. The rows are indexed by  $[i]$  (vertical arrow pointing down) and the columns by  $[j]$  (horizontal arrow pointing right). The elements are numbered 0 to 24 in row-major order.

|    |    |    |    |    |
|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  |
| 5  | 6  | 7  | 8  | 9  |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 |

# False sharing

- El problema es aún peor cuando tenemos varios hilos
- Al escribir en una posición de memoria se invalida la línea entera
- Se puede producir un efecto ping-pong





# Referencias

- Basada en las transparencias originales de Domingo Giménez Cánovas
- Introducción a la Programación Paralela, Domingo Giménez Cánovas
  - Disponible en la biblioteca
- Parallel Programming in C with MPI and OpenMP. Michael Quinn.