Programación Paralela y Distribuida
Clase 02 - *Background*

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

Teoría del rendimiento o *performance*

- ¿Qué es *performance*?
  - Reducir el tiempo total de computación
  - Incrementar la tasa a la cuál una serie de resultados se alcanza
  - Reducir la cantidad de energía consumida durante la computación
- Una vez usted define que es *performance* usted comienza a trabajar en modificar su código para alcanzar ese *performance*
- **OJO** es importante el *performance* pero con resultados correctos

Teoría del rendimiento

- El rendimiento de un programa se puede medir o teóricamente o prácticamente a través de las implementaciones en software de este programa
  - Teóricamente es más económico a la hora de comparar versiones del programa
  - Teóricamente se pueden identificar posibles limitaciones en el programa
  - Una vez identificado el algoritmo teórico se debería pasar a la versión en software

Latency y *Throughput*

- *Latency* corresponde al tiempo que toma completar una tarea
- *Throughput* corresponde a la tasa a la cual una serie de tareas puede ser completada en una cierta cantidad de tiempo. Un término análogo es *bandwidth* de red
- El *pipelining* es un ejemplo de mejora en el *throughput* pero incremento del *latency* debido a las esperas que se tienen en cada etapa del *pipeline*
- Los **servidores web** deben tener *response times* bajos pero se deteriora el *throughput* de solicitudes dado que estas se **encolan** y se ocasiona una demora del procesamiento

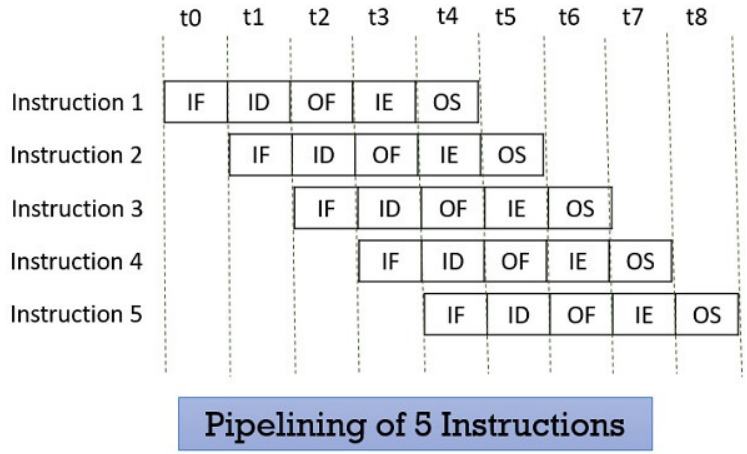

Figure 1: Pipelining de 5 etapas

Ley de Amdahl (1)

- La ley de Amdahl establece que el tiempo de ejecución de un programa está dividido en
  - Instrucciones ejecutadas serialmente, W_{ser}
  - Instrucciones ejecutadas en paralelo, W_{par}
- Si tenemos P *workers* los tiempos para ejecutar un algoritmo está dado por:

$$T_1 = W_{ser} + W_{par},$$
$$T_P = W_{ser} + \frac{W_{par}}{P}$$

- Insertando estas expresiones en la fórmula del *speedup* tenemos que:

$$S_P = \frac{W_{ser}+W_{par}}{W_{ser}+\frac{W_{par}}{P}}$$

- Sea f la fracción de código no paralelizable, entonces:

$$W_{ser} = fT_1$$
$$W_{par} = (1-f)T_P$$

- Se sustituye nuevamente en la ecuación del *speedup*

$$S_P \leq \frac{1}{f+\frac{1-f}{P}}$$

Ley de Amdahl (2)

- La ley de Amdahl nos dice entonces que por mucho poder paralelo que se tenga a disposición, el algoritmo está limitado por el tiempo que toma ejecutar sus instrucciones secuenciales

$$S_\infty \longrightarrow \frac{1}{f}$$

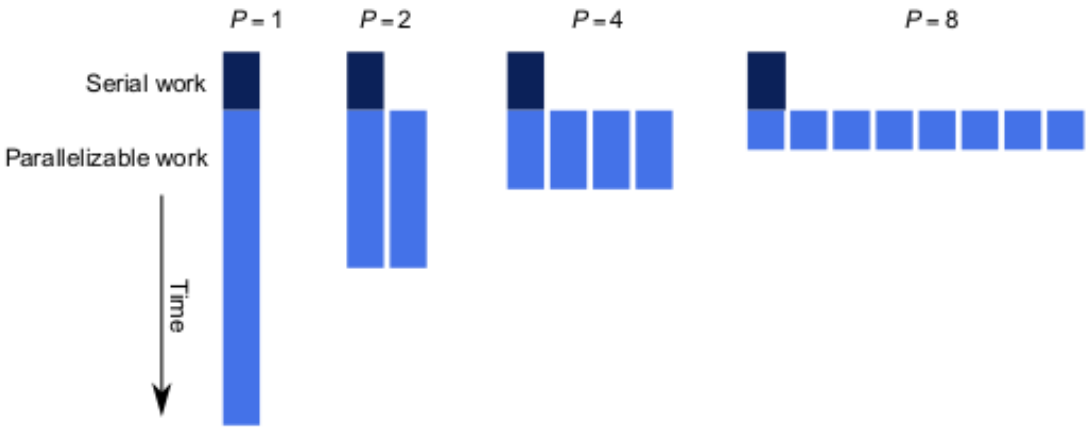

Figure 2: Serial + Paralelo

Speedup, eficiencia y escalabilidad (1)

- *Speedup* compara la *latencia* de ejecutar un programa con una unidad de hardware (*worker*) versus P unidades de hardware

$$speedup = S_P = \frac{T_1}{T_P}$$

- **Eficiencia** es el *speedup* dividido por el número de *workers*, P:

$$efficiency = \frac{S_P}{P} = \frac{T_1}{P \times T_P}$$

Speedup, eficiencia y escalabilidad (2)

Ejercicio numérico

Suponga que un **programa secuencial** toma 2.5 segundos en ser ejecutado. El programa se modifica de modo que al ser ejecutado sobre un computador con 8 núcleos alcanza un tiempo de 725 milésimas de segundo. ¿Cuál es la eficiencia de este programa?

Ejercicio numérico

Suponga un programa A del cual se sacaron una versión secuencial A y una versión paralela A_p que usaba 8 hilos de ejecución. Los tiempos de ejecución del programa fueron $T_A = 12$ y $T_{A_p} = 5$. ¿Qué porcentaje del código es secuencial y qué porcentaje es paralela?

Hint: El $Speedup = \frac{1}{1-f+f/p}$ siendo f la porción del código secuencial y p el número de *workers* involucrados en el procesamiento paralelo (*workers*: *threads*, *cores*)

Speedup, eficiencia y escalabilidad (3)

- La eficiencia ideal es 1 ¿Eso qué significa? ***Hint*** use la fórmula de eficiencia e identifique en que momento se logra obtener un valor de 1
- El término ***linear speedup*** es cuando un algoritmo corre P veces más rápido cuando este ejecuta sobre P procesadores
- ***Superlinear speedup*** es cuando se tiene una eficiencia superior a 1, es decir, superior al 100%
  - Un programa serial ineficiente (e.g. pobre uso de cache) se compara con un programa **afinado** y ejecutado en paralelo
  - En una búsqueda basada en árboles, un algoritmo puede llegar más rápido a identificar ramas del árbol que no llevan a ningún resultado, podar esas ramas y así evitar que otras instancias intenten ir por esas ramas

Modelo *Work-Span*

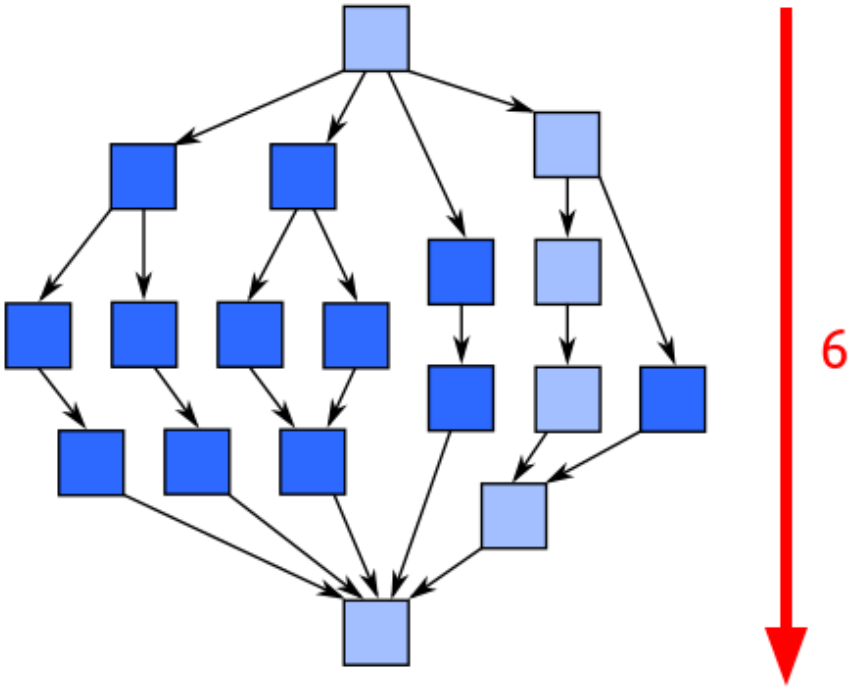

Figure 3: Ejemplo *Work-Span* con 18 tareas

- El modelo *Work-Span* deriva su nombre del número total de tareas a ejecutar (*work*) y del tiempo que toma llevar a cabo esas tareas en una máquina infinitamente paralela (*span*)
- Cuando el número de procesadores es 1 (P = 1) entonces procesar toma T_1. Cuando el num. de procesadores es ∞ entonces procesar tomar T_∞
- El ejemplo gráfico indica que el T_1 o el *work* de esa aplicación es 18 y su T_∞ o su *span* es 6. El *span* se conoce también como el *depth* del grafo

Model *Work-Span* (2)

- En este modelo es **imposible** tener un *speedup* **superlinear**

$$S_p = \frac{T_1}{T_P} \leq \frac{T_1}{\frac{T_1}{P}} = P$$

- ☹ Adicionar más procesadores no mejorará el rendimiento pues hay un límite dado por la cantidad de tareas que se pueden ejecutar en paralelo

$$S_p = \frac{T_1}{T_P} \leq \frac{T_1}{T_\infty}$$

- ➡ $speedup \leq \frac{work}{span}$. En el gráfico anterior el $speedup = \frac{T_1}{T_\infty} = \frac{18}{6} = 3$
- *Work-Span* **no considera** los costos de acceso a memoria y comunicación. A un modelo que **sí** los considere se llama ***burdened-span***

La Fórmula de Little

- La fórmula de Little nos relaciona el *throughput* y la *latencia* con la **concurrencia**
- Considere **se desea** tener un sistema capaz de procesar R ítems por unidad de tiempo y le toma al sistema procesar cada ítem L unidades de tiempo. La concurrencia del sistema está dado por

$$C = R \times L$$

- **Concurrencia** tiene que ver con el número total de tareas que están **en progreso** al mismo tiempo, mientras que el **paralelismo** se encarga de **ejecutar** las tareas al mismo tiempo
- **Concurrencia** es un término más general que **paralelismo** y se puede entender también como un **paralelismo simulado** o *pseudo-paralelismo*

La Fórmula de Little (2)

- La concurrencia se puede usar para mejorar el ***throughput*** cuando las tareas llevan a cabo operaciones con **alta latencia**
- Suponga que un *core* ejecuta una operación por *click* del reloj pero cada operación espera por un acceso a memoria que toma 3 *click* del reloj ➡ para ocultar la latencia se pueden ejecutar $C_{tiempo/inst} = R_{tiempo/acceso} \times L_{acceso/inst} = 1 \times 3$ operaciones a la vez
  - **IMPORTANTE** en este caso las operaciones deben ser independientes entre ellas
- Este principio de concurrencia es el que usa el paquete ***multithreading*** de Python
- **ENTONCES** paralelizar para ocultar la latencia y maximizar el *throughput* requiere **sobre descomponer** un problema y generar **extra concurrencia** por cada unidad de procesamiento

Riesgos de la programación en paralelo

- Condiciones de competencia (*race conditions*)

Table 2.2 Two tasks race to update shared variable `X`. Interleaving can cause one of the updates to be lost.

Task A	Task B
`a = X;`	`b = X;`
`a += 1;`	`b += 2;`
`X = a;`	`X = b;`

Figure 4: X podría quedar o con el valor de ‘a’ o de ‘b’

- Exclusión mutua y candados (*locks*)

Table 2.5 Mutex eliminates the race in Table 2.2. The mutex `M` serializes updates of `X`, so neither update corrupts the other one.

Task A	Task B
`extern tbb::mutex M;`	`extern tbb::mutex M;`
`M.lock();`	`M.lock();`
`a = X;`	`b = X;`
`a += 1;`	`b += 2;`
`X = a;`	`X = b;`
`M.unlock();`	`M.unlock();`

Figure 5: `mutex` para garantizar acceso ordenado

Riesgos de la programación en paralelo (2)

- ***Deadlock*** se produce cuando dos procesos se bloquean mutuamente sin permitir el avance de ninguno de los dos

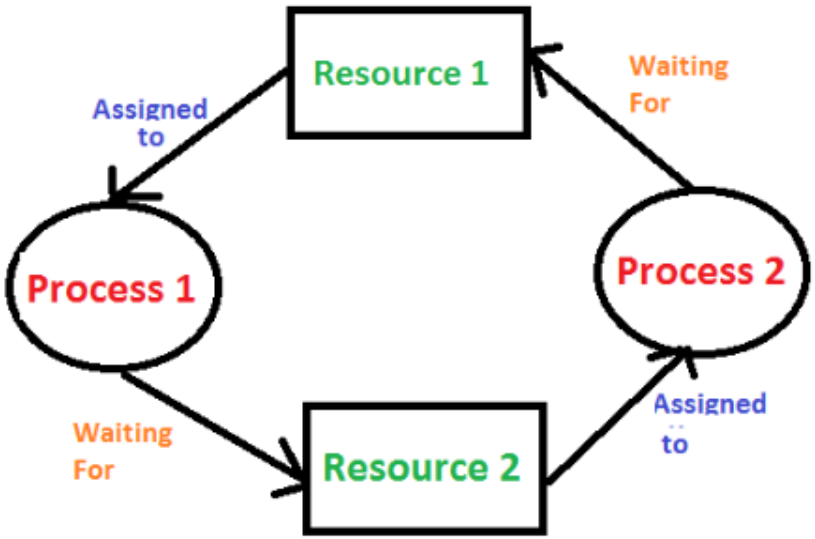

Figure 6: Ambos procesos requieren R1 y R2

- ***Strangled Scaling*** (“estrangulación” del escalamiento) se produce cuando se usan candados en zonas de código amplio, se debe tener bloqueos más limitados en el código (***fine-grain locking***) . En general, **deben evitarse al máximo los candados**
- **Falta de localidad**
  - **Localidad temporal** se accede a **la misma posición de memoria** en el futuro cercano
  - **Localidad espacial** se accede a **posiciones de memoria cercanas** en el futuro cercano

Riesgos de la programación en paralelo (3)

- **Desbalance en la carga** es la carga desigual entre *trabajadores*. Si las **tareas se pueden descomponer arbitrariamente** entonces con su división se puede balancear la carga

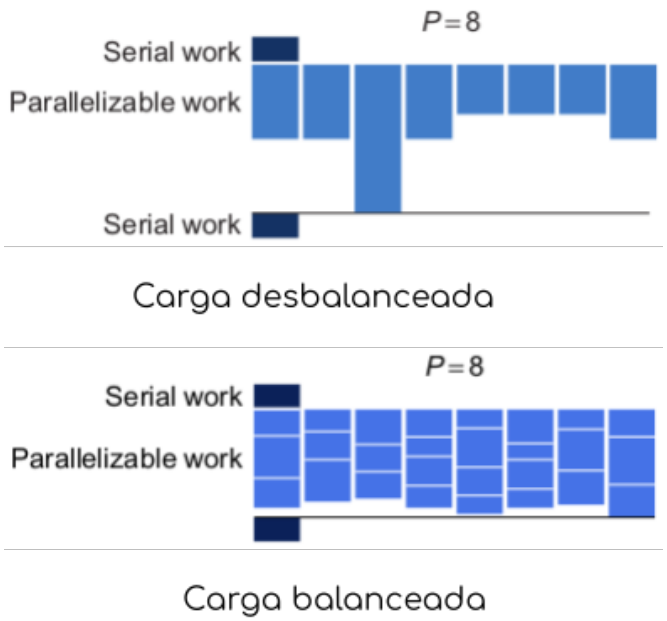

Figure 7: Descomposición de tareas para alcanzar el balance

- ***Overhead*** incrementar el número de tareas a procesar implica un mayor sobrecosto a la hora de gestionar y sincronizar tal cantidad de tareas. **Se debe encontrar un balance** de modo que se reduzca el coste en tiempo de la gestión en relación a la ganancia a la hora de procesar las tareas

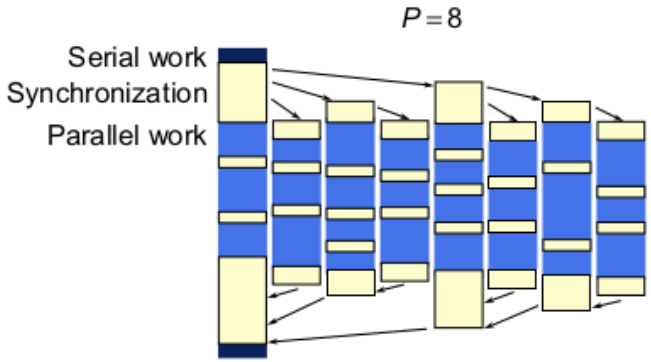

Figure 8: Más tareas demandan más costos de gestión y sincronización

Resumen

- La arquitectura del computador afecta de manera significativa el tiempo de ejecución de los programas
- La ley de Amdahl nos da unos límites teóricos respecto al rendimiento de una aplicación
- El modelo *work-span* nos permite conocer unos límites, uno bajo(*work*, serial) y otro alto(*span*, paralelo), respecto al rendimiento de un programa

Resumen (2)

- Se enumeraron algunos riesgos en la programación en paralelo que pueden limitar la *escalabilidad* de las aplicaciones
  - Hacer que el paralelismo escale con los datos
  - Mantener el *span* corto
  - Sobre-descomponer las tareas de modo que haya una holgura en la paralelización
  - Minimizar la sincronización (evitar *locks*)
  - Usar el concepto de **localidad** para disminuir el tráfico a la memoria
  - Aprovechar el paralelismo vectorial y de hilos de ser posible

NOTA

- El reordenamiento de operaciones puede causar en un algoritmo una mejora en su rendimiento. Sin embargo, ese reordenamiento puede traer diferentes resultados

```
a = 1e20
b = 1e-20
c = 1e-20
d = a + b - a
e = b + c - b
f = d + e

# Order 1
# (a + b - a) + (b + c - b) + a = (b) + (c) + a
f_order1 = d + e + a

# Order 2
# b + c - a + a + (a + b - a) + (b + c - b) = b + c + b + c
f_order2 = b + c - a + a + d + e
```