



Fundamentos de la supercomputación paralela y distribuida

Paralelismo y Sistemas Distribuidos
Grado en ciencia e ingeniería de datos
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

Licencia



**Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)**

<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es>

HPC: High Performance Computing

- HPC se corresponde con un segmento específico en el contexto de la computación
- La arquitectura esta diseñada para aplicaciones con mucho paralelismo
 - Segmento de servidores
 - CPUS muy potentes
 - Un alto número de cores
 - Cada uno de ellos no a frecuencias muy altas
 - Normalmente con bastante memoria, especialmente si están dedicados a *machine learning*
 - Instrucciones vectoriales
 - Eficiencia energética

HPC: High Performance Computing

- Las aplicaciones suelen estar diseñadas para explotar al máximo estas arquitecturas
 - Alto número de procesos/threads
 - Con tamaños de datos y patrones de accesos ajustados a los diferentes niveles de cache para aprovechar al máximo las optimizaciones y minimizar conflictos
 - Comunicaciones optimizadas
 - ▶ Agrupando transferencias para minimizar interferencias
 - ▶ Intentando solapar comunicación y cálculo
 - Almacenamiento de datos optimizado
 - ▶ Solapando comunicación y acceso a disco
 - ▶ Usando sistemas de organización y acceso eficiente
 - ▶ Nuevos tipos de bases de datos

Arquitectura HPC

- Visión general de arquitecturas HPC
 - Nodo de computación: CPU y Memoria.
 - Red de interconexión
 - Dispositivos

Arquitectura HPC

- Concurrencia y paralelismo
 - Concurrencia es la capacidad de ejecutar un programa en paralelo
 - Los programas son secuenciales o concurrentes. Un programa concurrente puede ejecutarse en paralelo dependiendo de la arquitectura. Un programa secuencial no puede ejecutarse en paralelo, ya que está definido en el código.
- Un proceso/thread (programa en ejecución) se ejecuta de forma concurrente cuando comparte con el core con otros procesos/threads
- Un proceso/thread se ejecuta de forma paralela cuando tiene las cpus dedicadas

Concurrente/Paralelo

Ejecución paralela

Time(each CPU executes one process)

CPU0	Proc. 0
CPU1	Proc. 1
CPU2	Proc 2

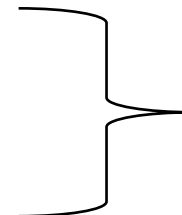
Ejecución concurrente

Time (CPU is shared among processes)

CPU0	Proc. 0	Proc. 1	Proc. 2	Proc. 0	Proc. 1	Proc. 2	Proc. 0	Proc. 1	Proc. 2	...
------	---------	---------	---------	---------	---------	---------	---------	---------	---------	-----

Concurrente/Paralelo

- La idea es sencilla, pero la implementación no tanto
- Los movimientos de threads entre CPUs es una de las principales fuentes de pérdida de rendimiento, incluso teniendo N procesos/threads y N cores a veces hay movimientos o asignaciones absurdas como poner 2 threads/procesos en un mismo core
 - Cada vez que un thread se mueve de core, hay una perdida importante de rendimiento
 - ▶ Coste del cambio de contexto
 - ▶ Pérdida de la optimización ofrecida por la cache
 - Si dos threads/procesos comparten core.... Este problema se multiplica porque es repetido periódicamente
- Hay varios "actores" responsables de la asignación/mapeo de threads a cores
 - El sistema de colas
 - Las librerías de paralelismo
 - El S.O.



Demasiado caos a veces

Métricas de evaluación

- Ejecución de aplicaciones paralelas
- Caracterización de una aplicación paralela como un grafo: Tareas y dependencias entre tareas
- Análisis de aplicaciones a partir de un grafo de tareas
 - Concepto de camino crítico
 - Tiempo secuencial
 - Tiempo paralelo
 - Tiempo infinito
- Concepto de overhead
- Concepto de balanceo de carga
- Ley de Amdhal:
 - Fracción secuencial y paralela
 - Límite speedup en función de Amdhal
- Aceleración de una aplicación: Speedup/Slowdown
- Escalabilidad: Weak/Strong scale
- Aceleración del sistema: Throughput

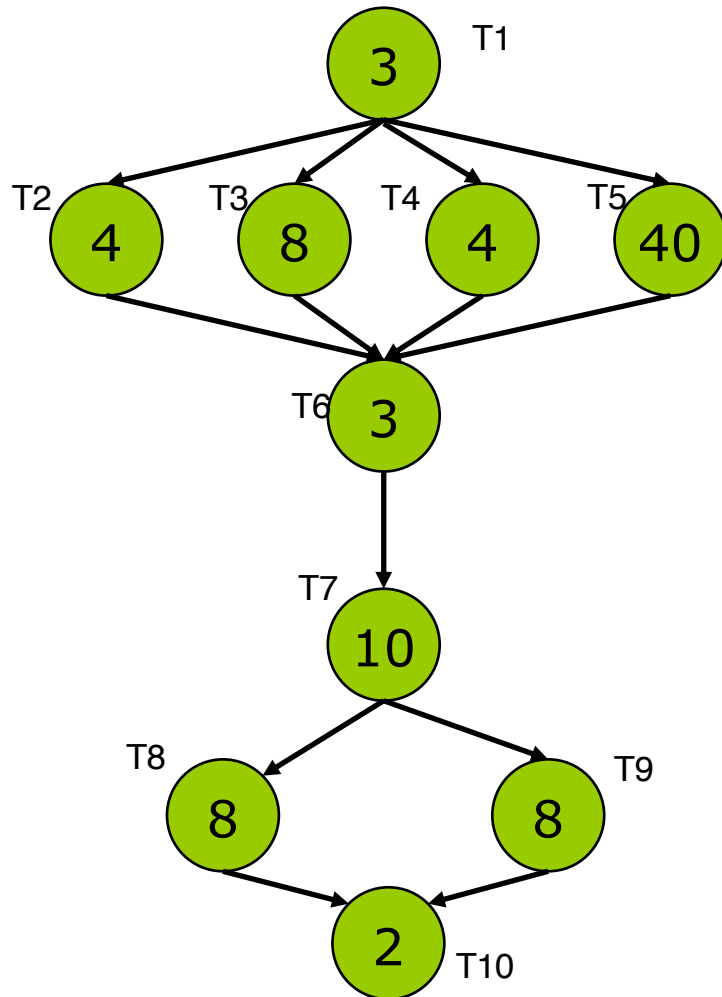
Análisis teórico de algoritmos paralelos

- La forma en la que paralelizamos los algoritmos determina el máximo paralelismo que podemos obtener
- Para hacer un análisis teórico, organizamos nuestro trabajo en tareas (*tasks*)
- Cada task es una secuencia de código indivisible
- Una task puede luego implementarse como un proceso/thread, o partes de un proceso/thread con sincronizaciones
 - ▶ Nos puede interesar crear menos y sincronizar para reducir costes, todo depende de los detalles
- Cuando el resultado de una tarea se utiliza en otra, decimos que hay una dependencia, ya que la segunda no puede empezar hasta que la primera termina

Grafos de tareas

- Representamos nuestro algoritmo o programa paralelo como un grafo acíclico
 - Las tasks son nodos
 - Las flechas reflejan dependencias
 - El peso de cada nodo es el tiempo de ejecución estimado de cada task

Ejemplo



■ Camino: es una lista de tasks que, manteniendo las dependencias, va del inicio al final

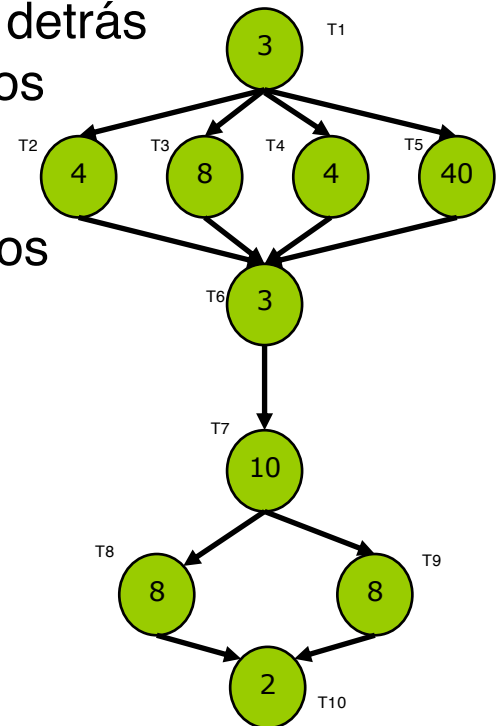
- ▶ Camino1: T1,T5,T6,T7,T9,T10
- ▶ Camino2: T1,T2,T6,T7,T9,T10
- ▶ Etc

■ Cada camino tiene como coste la suma de todas sus tasks

- ▶ Coste C1: $3+40+3+10+8+2=66$
- ▶ Coste C2: $3+4+3+10+8+2=30$

Ejemplo

- Si las tareas reflejan la mejor paralelización, el coste del camino **MÁS LENTO** es lo **MÁS RÁPIDO** que podrá ir la aplicación. Es lo que se conoce como **camino crítico**.
- En este ejemplo, el camino crítico es el camino 1 (66 segundos)
- Para evaluar los límites que impone el algoritmo, calcularemos
 - Tiempo secuencial o $T(1)$: suma del coste de todas las tareas. Sería el equivalente a ejecutar todas las tasks una detrás de otra en 1 CPU $\rightarrow 3+4+8+4+14\dots+2=92$ segundos
 - Tiempo infinito: coste del camino crítico
 - ▶ Aunque tengamos infinitos recursos, no podemos ir más rápido $\rightarrow 66$ segundos



Ejemplo

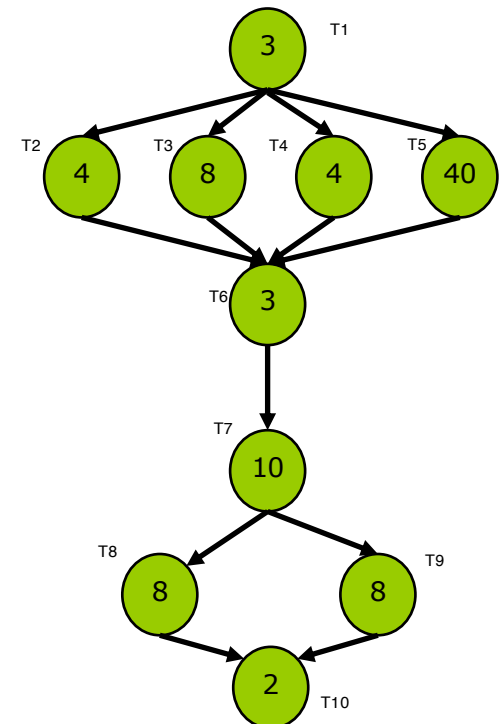
- Paralelismo: Máxima aceleración con infinitos recursos

- ▶ $Paralelismo = \frac{T(1)}{T_{\infty}} = 92/66 = 1.39 \text{ ☹}$

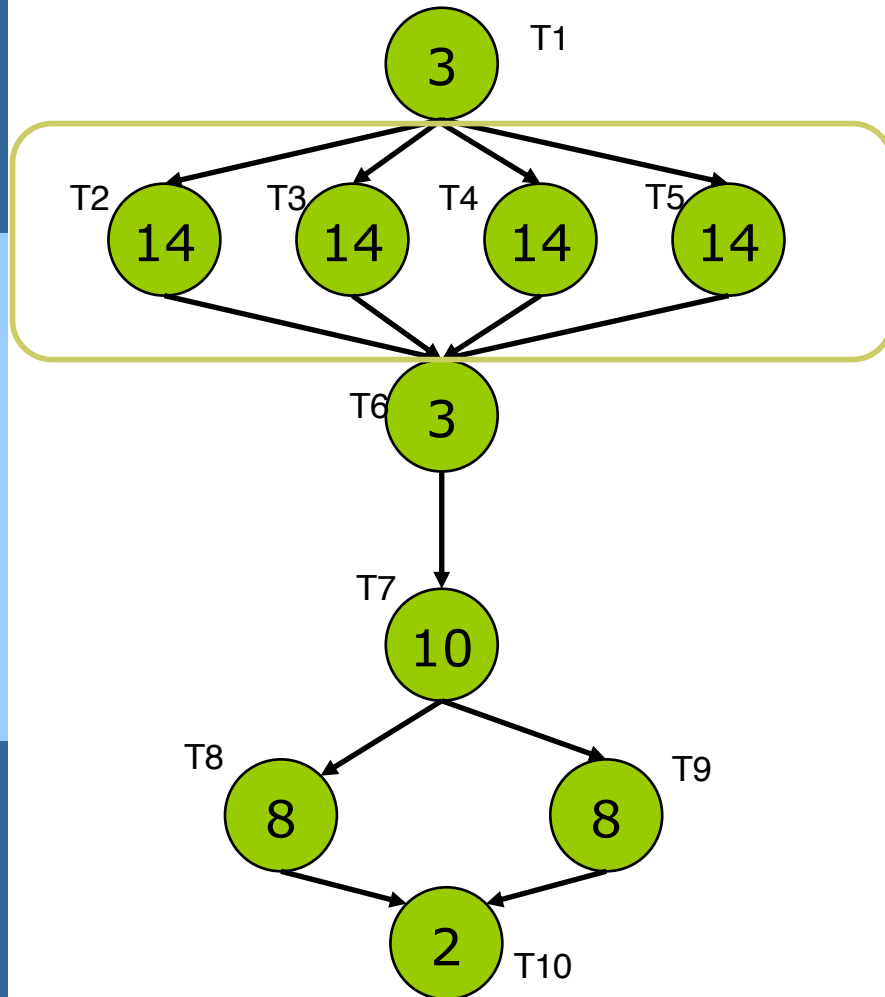
- Esto implica, que este algoritmo, aun teniendo infinitos cores, nunca llegaría ni a doblar su velocidad

- ▶ No tiene muchas tareas que podamos hacer a la vez
 - ▶ El peso de las tareas no está equilibrado

- Además, este es un análisis teórico para evaluar el potencial de nuestro algoritmo, aquí no están incluidos los costes de crear tareas, sincronizar, etc.



Ejemplo



■ Si equilibramos el peso conseguimos una mejora, aunque seguimos limitados por el número de tareas paralelas

- $T(1) = \text{no cambia!} = 92$
- $T(\text{inf}) = 3 + 14 + 3 + 10 + 8 + 2 = 40$
- $\text{Paralelismo} = 92/40 = 2.3$

Análisis de aplicaciones

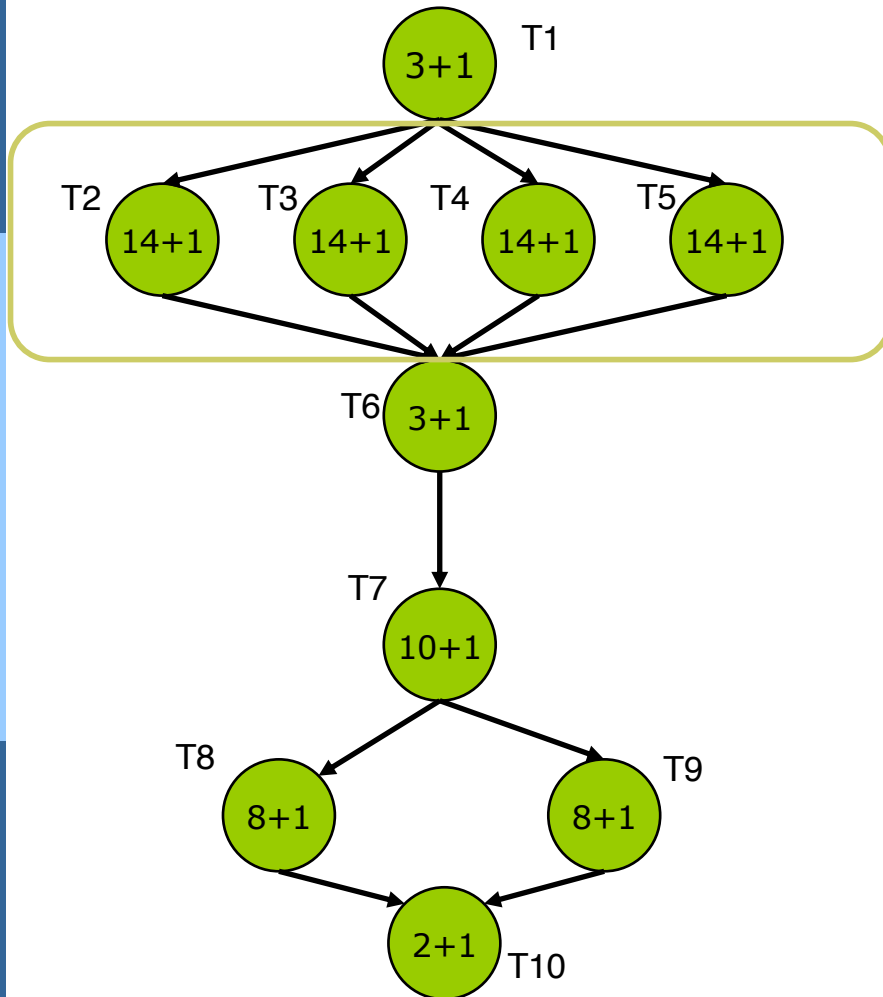
- Un grafo de tareas debería reflejar el potencial paralelismo de un algoritmo, sin tener en cuenta donde se va a ejecutar, el modelo de programación, etc
- Sin embargo, la realidad es que los detalles influyen de forma determinante en la viabilidad de la paralelización
- El modelo de paralelización elegido influye directamente en la estrategia de a elegir
 - MPI: el programa estará distribuido en diferentes nodos donde la transferencia de datos es “muy” costosa → muy importante minimizar el número de comunicaciones, sincronizaciones
 - OpenMP: El programa se ejecutará con varios threads en el mismo nodo, compartiendo memoria → importante minimizar interferencias en los accesos a cache

Análisis de aplicaciones

- Otros conceptos importantes a conocer y tener en cuenta....
- Overhead
 - Llamamos overhead a cualquier tipo de tiempo de ejecución adicional al propio algoritmo, normalmente generado por las librerías que usamos de gestión del paralelismo, por problemas de compartición de cpu, de memoria, sincronizaciones, etc
 - Ejemplo:
 - ▶ Si coste de crear una tarea fuera 1ms y tenemos 1.000.000 de tareas → 1000 segundos de overhead por creación de tareas!
 - Algunos de estos costes se dan “en paralelo”, por lo que no influyen de forma lineal en el tiempo de ejecución



Ejemplo: coste creación 1 seg



■ Sin overhead

- $T(1) = \text{no cambia!} = 92$
- $T(\text{inf}) = 3 + 14 + 3 + 10 + 8 + 2 = 40$
- $\text{Paralelismo} = 92 / 40 = 2.3$

■ Con overhead

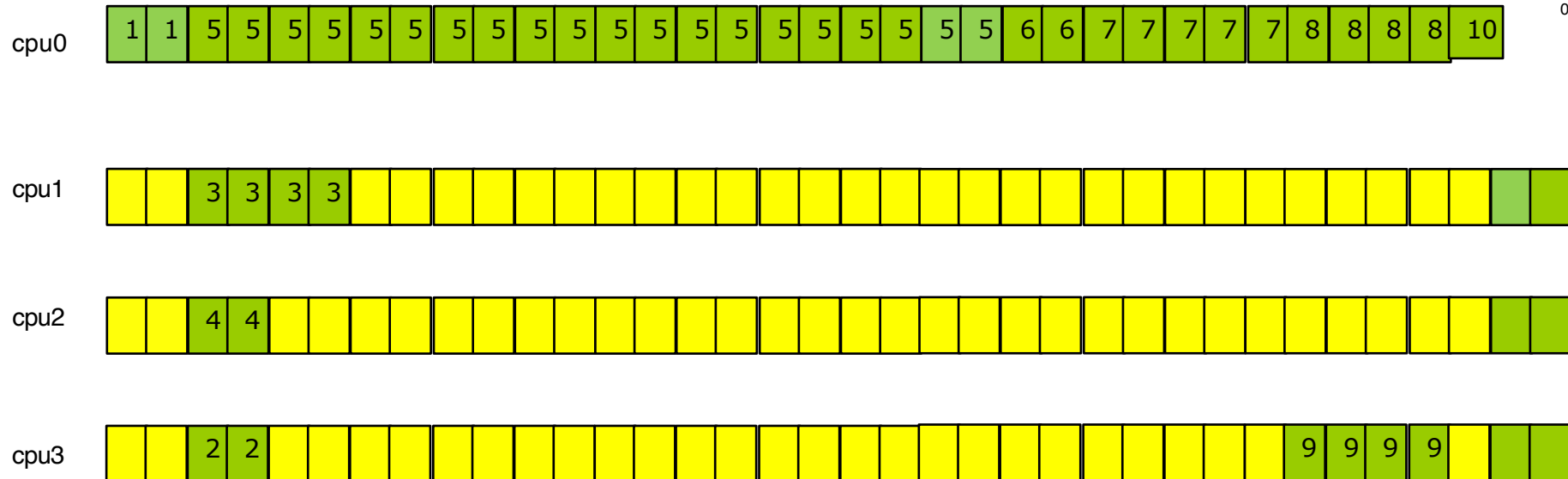
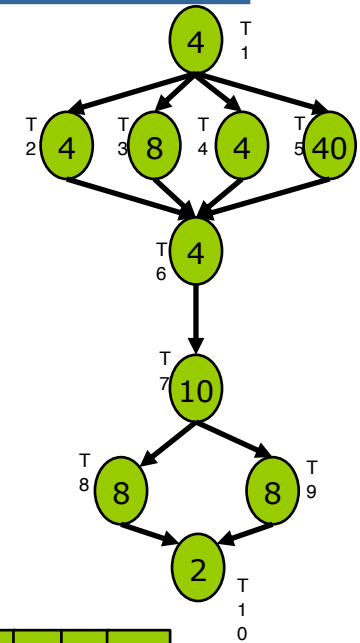
- Tiempo sin paralelizar = 92 seg
- Overhead = 10 seg
- Tiempo paralelo 1 cpu = $92 + 10 = 102$ seg
- $T(\text{inf}) = 4 + 15 + 4 + 11 + 9 + 3 = 46$ (vs. 40)
- $\text{Paralelismo} = 92 / 46 = 2$

Análisis de aplicaciones


■ Balanceo de carga

- Consiste en equilibrar al máximo la cantidad de trabajo de tareas que se ejecutan a la vez para reducir al máximo las esperas de las tareas que dependen de ellas (tiempo muerto)
- En el ejemplo original, ¿que pasaría con 4 cpus?
 - Cambiamos el tiempo de T1 y T6 a 4 para simplificar el dibujo

Análisis de aplicaciones



Cada cuadro son 2 seg

 Tiempo idle (sin ejecutar nada útil)

Análisis de aplicaciones

■ Granularidad de las tareas

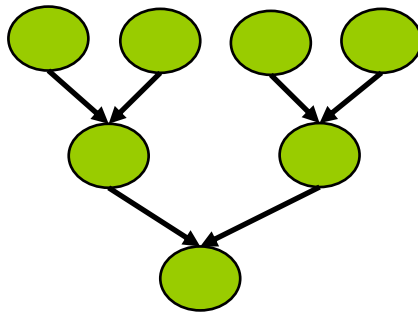
- Se entiende como granularidad de las tareas el tamaño (en tiempo de ejecución de las tareas)
- El tamaño de las tareas influye directamente en
 - ▶ El número de tareas → overhead de creación
 - ▶ La cantidad y tamaño de mensajes: sincronización y transferencia de datos
 - ▶ La cantidad de paralelismo que podemos extraer

■ Normalmente se habla de

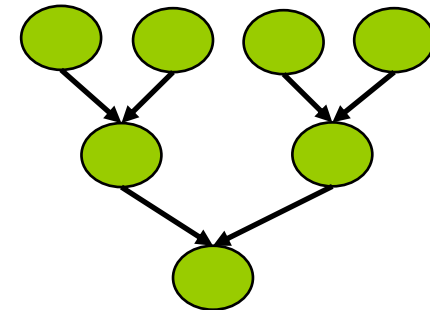
- Fine grained: Tareas pequeñas
 - ▶ Ventaja: Nos permite generar mas paralelismo. Nos permite balancear mejor la carga
- Coarse grained: Tareas grandes
 - ▶ Ventaja: Menos overheads en general.

Granularidad: ejemplo

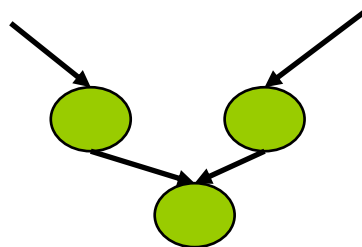
- Hacemos un diseño para sumar los elementos de un vector en paralelo
- Si el tamaño del vector es 4096 elementos, y hacemos un diseño en árbol, tenemos.



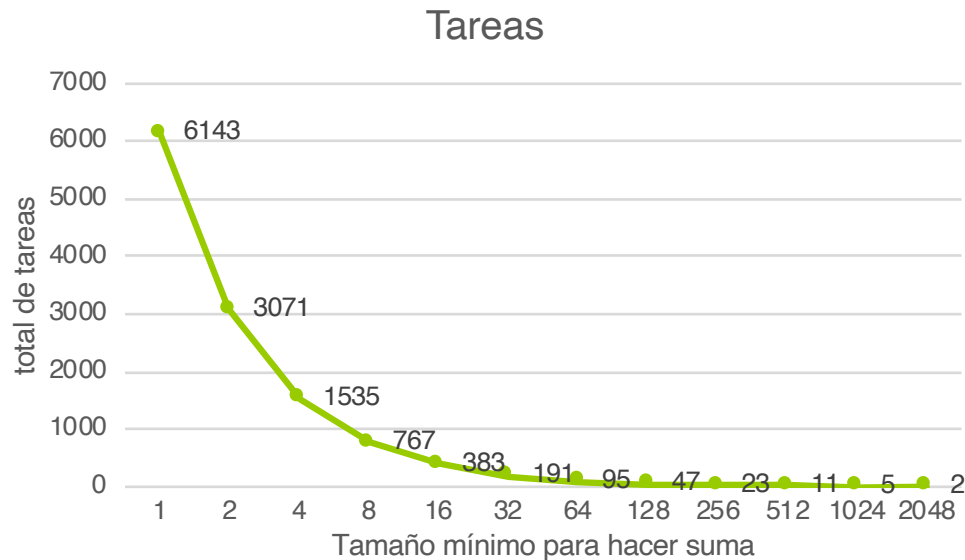
....



Cada tarea solo suma
2 números



Granularidad:ejemplo



- $N=4096$
- Si creamos 1 tarea cada 2 elementos, tenemos 6143 tareas usando una implementación recursive
- Si usamos un mínimo de 256 elementos, creamos 23 tareas con el mismo algoritmo

Análisis de aplicaciones

■ Speedup(p)

- Speedup es la aceleración que consigue una aplicación, se expresa en función del número de procesadores (cores)
- $Speedup(p) = \frac{Tiempo(1)}{Tiempo(p)}$
- Para ser justos, el tiempo(1) debería ser el tiempo sin paralelizar, pero mucha veces ese Código no está disponible o no es igual que el paralelizado, por lo que usamos el tiempo con 1 core.
- El tiempo con P cores depende de la asignación de tareas a cores
- Para calcularlo a nivel teórico, hemos de hacerlo con una propuesta de asignación de tareas-cores específica
- En el laboratorio ejecutaremos nuestros programas con los P cores y mediremos el tiempo

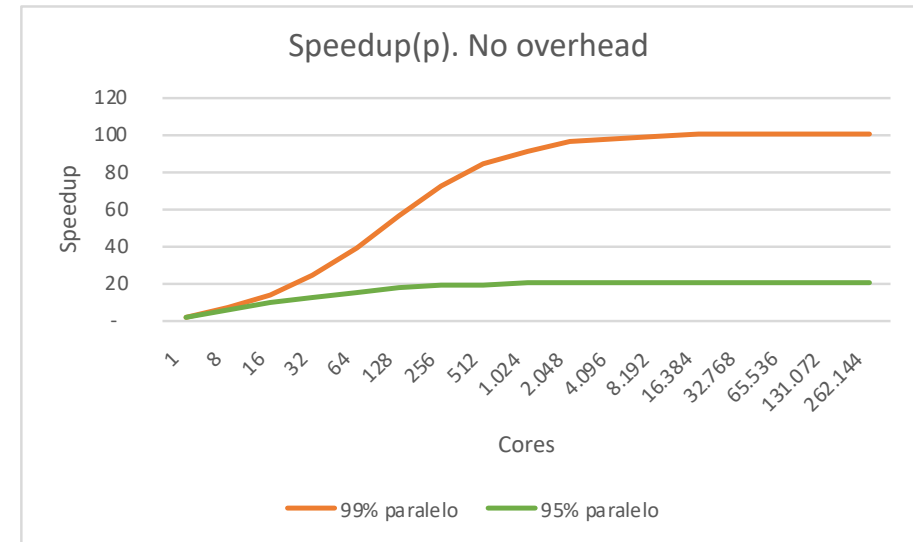
Análisis de aplicaciones

■ Eficiencia

- La eficiencia nos da una medida de como aprovechamos los recursos que tenemos
- $Eficiencia(p) = \frac{Speedup(p)}{p}$
- Sacar un speedup=100 no está mal, pero si es con 10.000 cores ☹ , no están muy bien aprovechados
- Cualquier factor que genere overhead normalmente es proporcional al número de core, por lo que la eficiencia suele caer al aumentar el numero de cores

Speedup

- Patrones de speedup habituales
- Caso 1, **ideal**: no hay overhead, se muestra el efecto de no tener 100% del código paralelizable
 - 99% o 95% código perfectamente paralelizable
 - Caso 99%, speedup máximo de 100 con 262K cores
 - Ejemplo calculado sobre caso sintético de $T(1)=100.000$ segundos
 - ▶ $100.000 \cdot 0,01 = 1000$ segundos que no es posible acelerar
 - ▶ $S_{\infty} = \frac{100000}{1000} = 100$
 - Eficiencia con 262144 cores=0,00038☹
 - Eficiencia con 256 cores=0,28



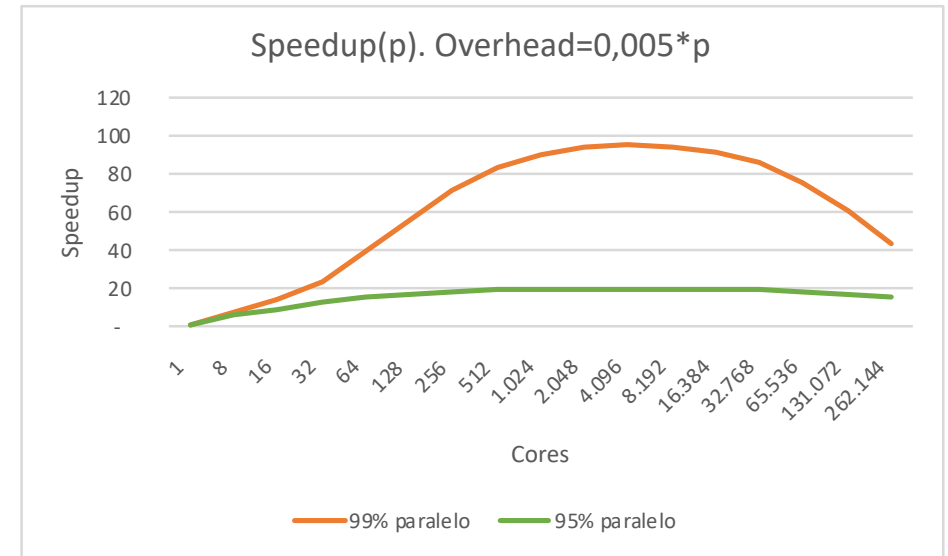
Speedup

■ Patrones de speedup habituales

- Caso 2, **realista**: hay overhead proporcional al número de cores.
- El impacto del overhead es una deceleración.
 - ▶ $T(1024)=1101$ seg.
 - ▶ $T(2048)=1058$ seg.
 - ▶ $T(32762)=1166$ seg.

■ Muy importante tenerlo en cuenta !!

■ En los casos en que el programa pierde velocidad, ya se habla de slowdown



La ley de Amdhal

- La ley de Amdhal ofrece un límite teórico (una intuición) sobre la máxima aceleración que podemos obtener con un programa en función de **que porcentaje se puede paralelizar y que porcentaje no**
- Es el modelo de speedup utilizado en el ejemplo anterior
- El modelo nos indica el máximo speedup teórico en función de la “parallel fraction”, ϕ)
- Idea: Si tenemos un $x\% = \phi$ de nuestro programa que podemos acelerar, $T(1) \cdot (1 - \phi)$ será el mínimo tiempo de ejecución de nuestro programa, que corresponde con el caso (no real) de acelerar el resto infinitamente.
 - $T(p) = \text{tiempo_no_acelerado} + \text{Tiempo_acelerado}(p)$
 - $\text{Tiempo_no_acelerado} = T(1) \cdot (1 - \phi)$
 - $\text{Tiempo_acelerado}(p) = \frac{\phi \cdot T(1)}{p}$

La ley de Amdhal

- En general:

- $Speedup(p) = \frac{T(1)}{(1-\phi)*T(1) + \frac{T(1)*\phi}{p}} \rightarrow Speedup(p) = \frac{1}{(1-\phi) + \frac{\phi}{p}}$

- En el caso de querer estimar el máximo speedup, $p=\infty$

- $Speedup(\infty) = \frac{1}{(1-\phi)}$

Objetivos de la paralelización

■ *Strong scale*: Ejecutar un trabajo fijo lo más rápido posible

- Las métricas que se utilizan son el Tiempo de ejecución, speedup y la eficiencia
- La mayoría de usuarios solo se centran en el tiempo, es lo que se llama “*time to solution*”, sin embargo, conviene ser conscientes del gasto que generamos
- La cantidad de trabajo que tenemos en este caso es constante, y vamos aumentando el número de recursos para ir más rápido
- Suele estar limitado por los factores que hemos comentado, pero es el objetivo tradicional del entorno HPC
- 1 ejecución N (\gg) cores

■ *Weak scale*: Ejecutar más trabajo en el mismo tiempo

- La métrica que se utiliza es el throughput=número de trabajos por unidad de tiempo
- Esta opción es muy interesante cuando tenemos poca eficiencia por motivos de overhead
- Habitualmente
- N ejecuciones con M(\ll) cores

Resumen: Límites en el paralelismo

- Factores que limitan el paralelismo, relacionados con:
 - El propio algoritmo (dependencias, Fracción paralela)
 - La configuración concreta de la aplicación (input)
 - El entorno de ejecución (herramientas, librerías, etc),
 - El hardware
- Cantidad de trabajo a realizar
- *Overheads*
 - Creación de trabajo
 - Transferencia de datos
 - Sincronizaciones
- Cantidad de recursos disponibles
- Equilibrio en la distribución de datos (Load balance)