



8. Evaluación de Prestaciones

Índice

1. Objetivos
2. Motivación
3. Qué es el High Performance Computing
4. Un poco de historia
5. Aplicaciones
6. Clasificación de Arquitecturas y Criterios de Clasificación
7. Enfoques
8. Evaluación de Prestaciones
9. Limitación de prestaciones
10. Ley de Amdahl, Ley de Gustafson
11. Mejorando prestaciones



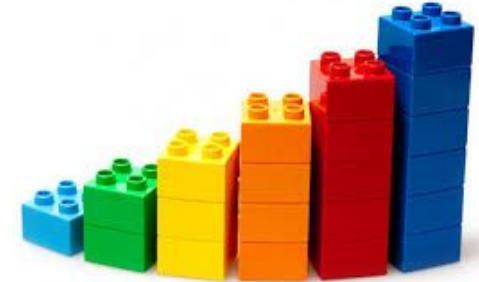
8.1 Medir tiempos

- No siempre interesa el tiempo desde el inicio hasta el final.
- Importa el **tiempo necesario para obtener una solución** a nuestro problema.
- Debemos **comparar con una versión del código secuencial** para obtener medidas de **ganancia o aceleración**.
- La ganancia está relacionada con el **tiempo secuencial y paralelo** de un algoritmo.
- La ganancia determina la **eficiencia**.
- Nuestro objetivo es siempre lograr la **máxima eficiencia**.



8.2 Escalabilidad

- Se dice que **un algoritmo es escalable**, si al aumentar el tamaño del problema que resuelve, la **eficiencia se mantiene constante**.
 - Si aumentamos el tamaño del problema, normalmente tendremos **más procesos** y la **eficiencia suele caer**.
- Nuestro objetivo será **buscar una ganancia lineal** (o súper-lineal si es posible) para lograr una **eficiencia constante**.





8.3 Latencia: wall-clock

- Debemos medir el **tiempo de ejecución necesario para llevar a cabo la tarea** que nos interesa.
- Por ejemplo, en una red, la latencia es la cantidad de tiempo que tarda un paquete en desplazarse de un origen a un destino.
- En nuestro caso **la latencia de un programa o tiempo de respuesta es el tiempo que tenemos que esperar para tener un resultado, contando solo el tiempo útil que el procesador ha estado trabajando en nuestro algoritmo.**





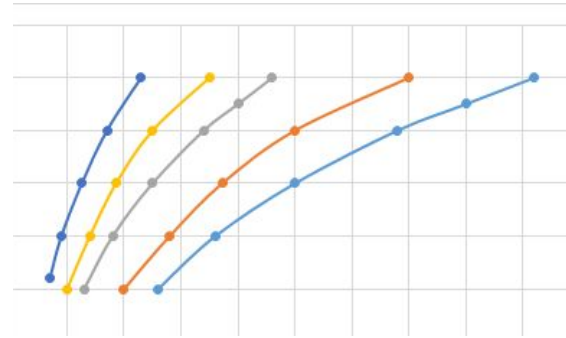
8.4 Energía consumida



- Cada vez se presta más **atención** también a **la energía consumida en relación al trabajo** realizado.
- A menor energía:
 - **Menor gasto en refrigeración** (y más tiempo de vida útil de los circuitos).
 - **Más sostenibilidad**
- El consumo energético en un supercomputador se mide a veces en el consumo por instrucción, el consumo por aplicación...

8.5 Rendimiento (Throughput)

- La potencia es la cantidad de trabajo que una máquina puede hacer por unidad de tiempo.
- Según se varíe la unidad de trabajo (instrucciones, procesos, hilos...), se generan diferentes medidas de rendimiento.
- La más conocida es **FLOPS**: Operaciones en coma flotante por unidad de tiempo. (Y sus múltiplos, como MFLOPS).
- Si se refieren a la **memoria**, será cantidad de **datos** (bytes, MB, GB) por segundo.
- Si se refieren a la **red** de interconexión, será la cantidad de **datos** que se pueden enviar por cantidad de tiempo.



8.5 Rendimiento (Throughput)



- El rendimiento se suele medir como la inversa de la latencia, pero esto solo es correcto si nos referimos a la misma cantidad de trabajo.
- Por ejemplo, en un proceso con una latencia de 1 segundo en completar 10^9 FLOPS, sí es correcto decir que su rendimiento es 10^9 FLOPS.
- Pero si hablamos de paquetes enviados a través de una red de interconexión, rendimiento y latencia no son inversamente proporcionales, pues afectan factores como la carga de la red.



8.6 Grado de concurrencia



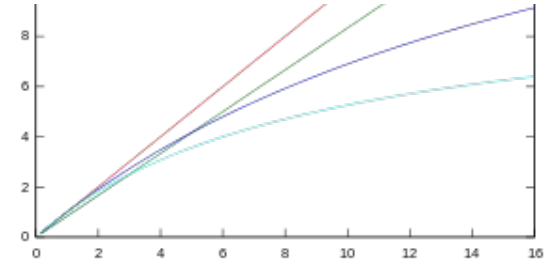
- Grado de concurrencia: **Cantidad de trabajo que se puede hacer de forma simultánea.**
- Las unidades de trabajo pueden ser: instrucciones, hilos, procesos...
- Aumentando la concurrencia se pueden mejorar otras métricas:
 - **Incrementa el rendimiento, haciendo más trabajo en el mismo tiempo.**
 - **Disminuye la latencia, terminando un trabajo en menos tiempo.**
 - **Ocultas las latencias** necesarias en puntos de sincronización, en sistemas desbalanceados... pues mientras que unos hilos están ejecutándose, otros pueden estar esperando.

8.7 Peak Performance



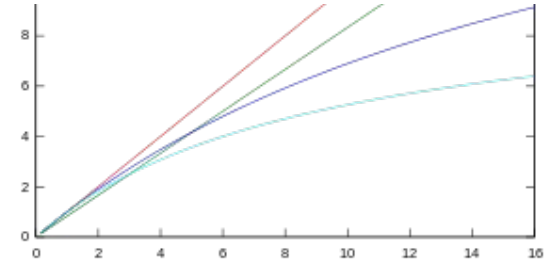
- El Rendimiento Pico o *Peak Performance* es la **cantidad máxima de trabajo** que puede realizar un sistema **por unidad de tiempo**, el máximo rendimiento.
- La diferencia entre ese máximo y el rendimiento de nuestro algoritmo, determina la **eficiencia**.
- Por ejemplo, la eficiencia en términos de *wall-clock* es la fracción de tiempo que un algoritmo ha estado realmente ejecutándose de forma útil. La eficiencia según el rendimiento de una máquina será una medida de cuánto tiempo útil ha estado ocupada.
- El rendimiento máximo **es siempre un límite superior (a veces inalcanzable)**.

8.8 Ganancia y escalabilidad



- En HPC, normalmente se dispone de **gran cantidad de recursos de computación**. Si el algoritmo es escalable, el aumento de recursos no reducirá la eficiencia.
- El incremento de prestaciones según los recursos de computación utilizados en una ejecución, se llama ganancia o aceleración (*Speedup*).
- En otros términos, la ganancia es la **relación entre la latencia antes de añadir un recurso y la latencia después de añadirlo**.
- La ganancia debe ser siempre **mayor que uno si el rendimiento aumenta**.

8.8 Ganancia y escalabilidad



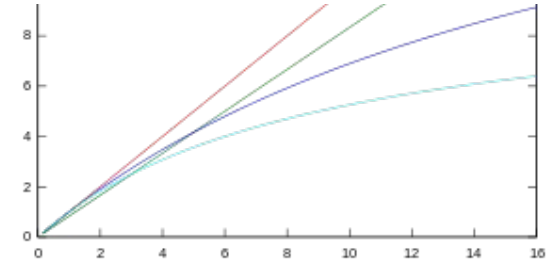
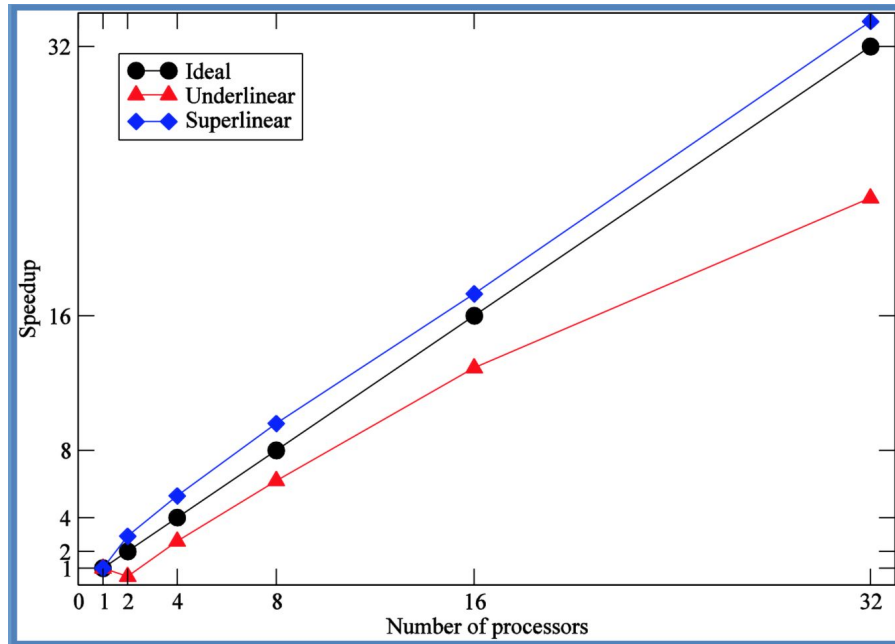
- La ganancia o aceleración (*speedup*) de un programa paralelo sobre N unidades de ejecución con respecto a su equivalente secuencial se calcula a partir de los tiempos de ejecución secuencial, T_p , y paralelo, T_s (en esa configuración):

$$S_N = \frac{T_s}{T_p}$$

- A partir de la aceleración, podemos calcular la eficiencia obtenida al usar N unidades de ejecución, E_N :

$$E_N = \frac{S_N}{N}$$

8.8 Ganancia y escalabilidad



¿Qué podría causar una aceleración superlineal?

- El efecto de la **caché** y la **RAM**
- El **algoritmo** (si la distribución de tareas permite evitar un número significativo de ellas, como en un Branch & Bound)



8.9 Profiling

- Es el proceso de **analizar un programa en ejecución** (análisis dinámico) **para medir** distintas magnitudes de interés sobre el mismo, como la cantidad de memoria usada, el número y tipo de instrucciones ejecutadas, o el tiempo usado por sus distintas funciones.
- Ejemplos de profiler son el **Intel VTune Profiler** o el Matlab Profiler.



8.9 Profiling

- Podemos distinguir los siguientes tipos de profilers:
 - **Según el resultado:**
 - Planos/Simples: Sólo dan el tiempo consumido por cada llamada
 - De grafo de llamadas: Además de tiempos, muestran el flujo de llamadas
 - Para conjunto de entradas: Estudian cómo varía el rendimiento de un programa para un conjunto de entradas.
 - **Según la granularidad** de los datos (tamaño de las muestras del profiler):
 - Basados en eventos: Registran la frecuencia de sucesos como interrupciones o llamadas a funciones
 - Estadísticos: En lugar de seguir toda la ejecución, presta atención en ciertos momentos y los extrapolan.



8.10 Tracing

- El *tracing* es un **tipo de registro de eventos para obtener información precisa y a bajo nivel sobre la ejecución de un programa**. Se presta especial atención a los mensajes que se manda durante la ejecución para detectar errores en el **orden de aparición en los eventos**.
- Mientras que el profiling se centra más en detectar problemas de rendimiento (cuellos de botella, funciones mal diseñadas, desbalanceo de cargas... => **tiempos**), el tracing se enfoca en identificar problemas en la lógica o diseño (**orden y desencadenantes**).
- Un ejemplo de Tracer es VampirTrace

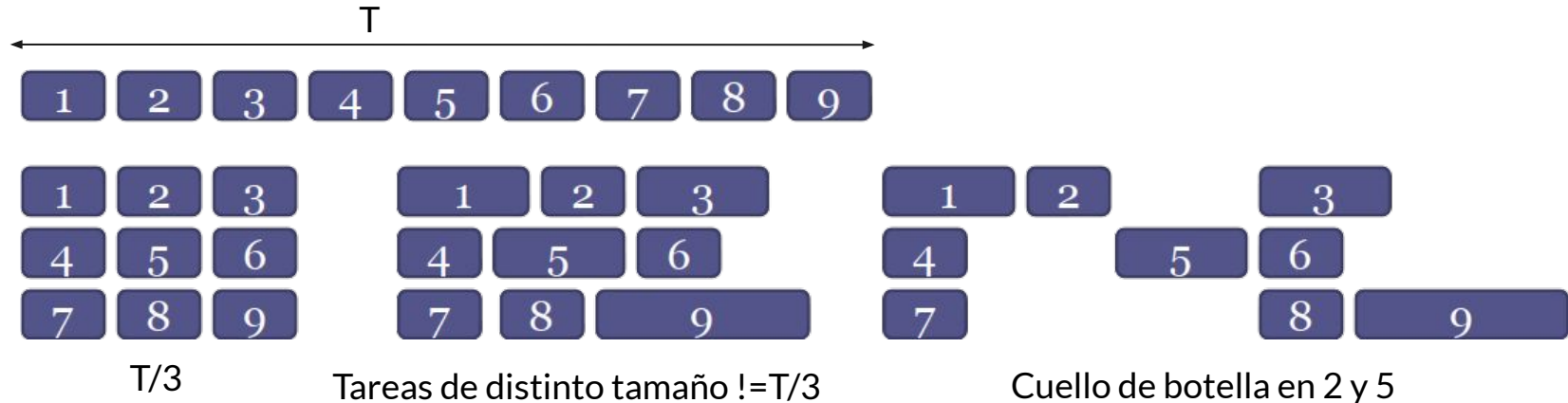


9. Limitación de prestaciones



9.1 Factores limitantes

Dado un trabajo T que tarda un tiempo TS en secuencial, y un conjunto de N unidades de procesamiento, al paralelizar T esperaríamos que el tiempo fuera TS / N . Sin embargo, esto difícilmente ocurrirá, pues al dividir el trabajo en tareas, no todas tardan lo mismo, y unas dependen de otras. Además, la propia paralelización implica tareas nuevas (*overhead*).





9.1 Factores que limitan

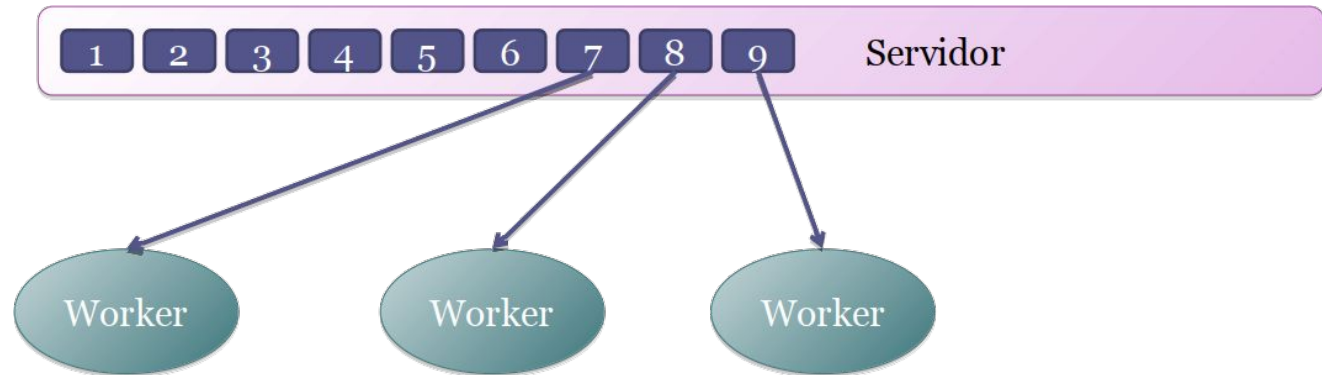
- Limitaciones por **dependencias** del algoritmo.
- Limitaciones por **sincronizaciones**.
- Limitaciones por cuellos de botella por el acceso a **recursos compartidos**.
- *Startup Overheads*.





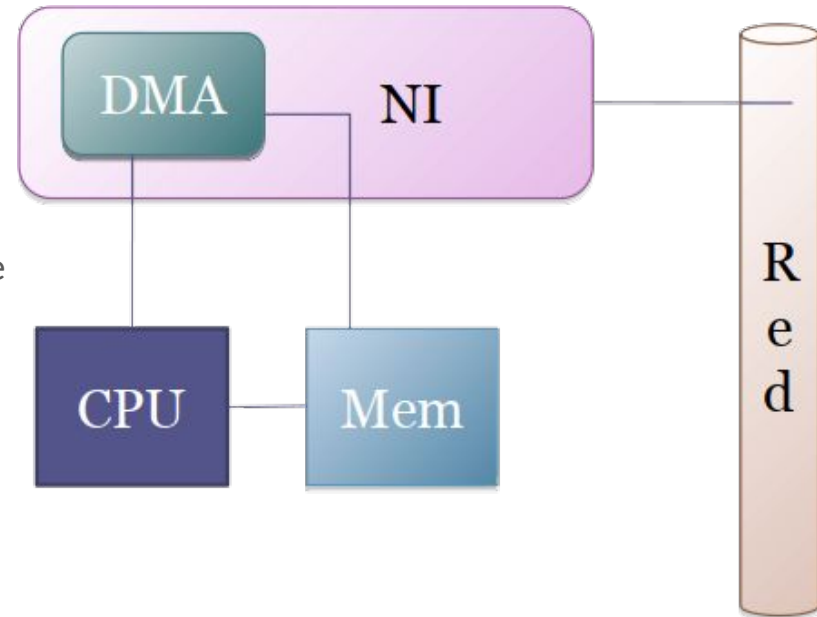
9.2 Limitaciones del algoritmo

- Ejemplo de reparto inicial de tareas entre unidades de procesamiento:

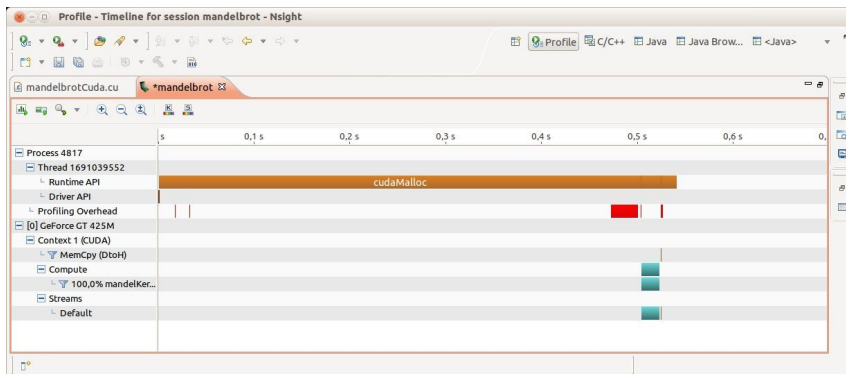


9.3 Acceso a recursos compartidos

- Ejemplo de comunicaciones por paso de mensajes con una sola interfaz de red:
- Estar accediendo siempre a la misma zona de la memoria (lo que implica sincronización y fallos de caché...)
- ...

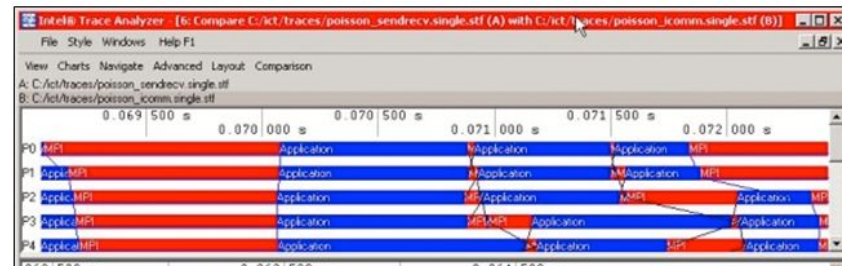


9.4 Startup Overheads



Profiling para CUDA

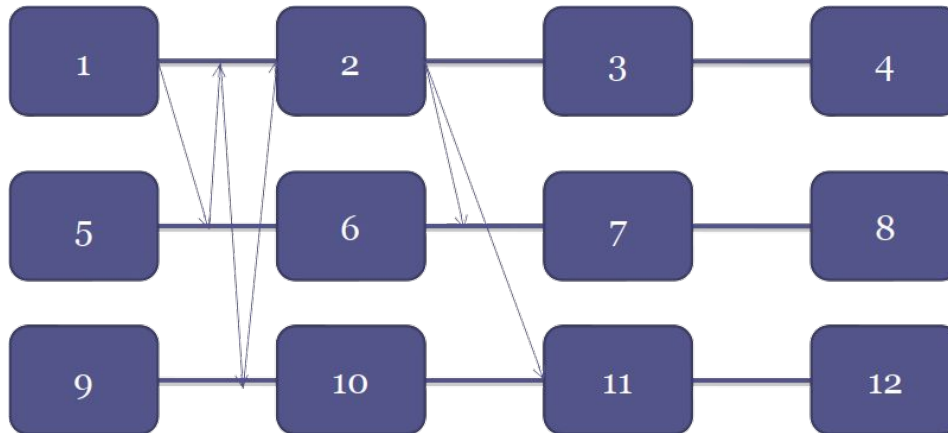
Profiling para MPI





9.5 Comunicaciones

- A cierto nivel, las comunicaciones siempre son secuenciales:





10. Ley de Amdahl y Ley de Gustafson

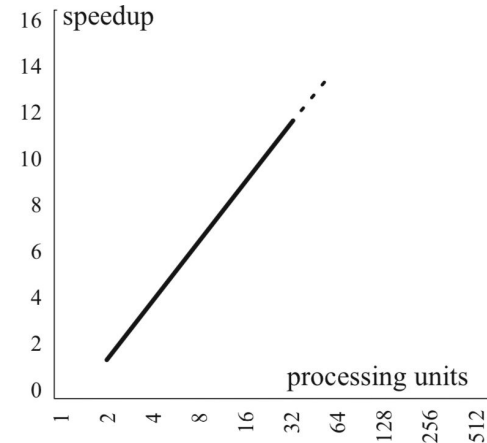




10.1 Ley de Amdahl

- Si doblamos el número de procesadores disponibles para un trabajo dado, esperamos que el tiempo sea la mitad. Si volvemos a doblar los procesadores, confiamos en que el tiempo anterior vuelva a ser la mitad del anterior... Es decir, **esperamos una aceleración lineal** con el número de procesadores:

+ Lo que esperamos:

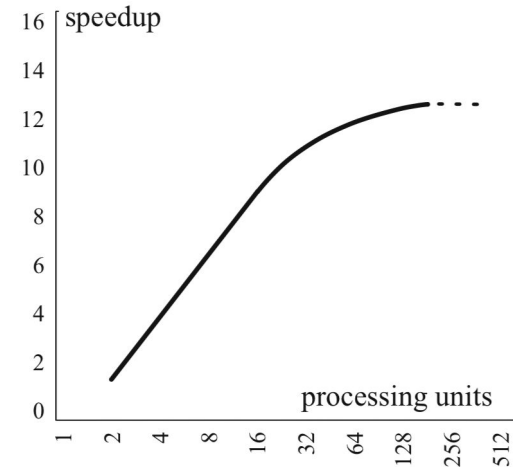
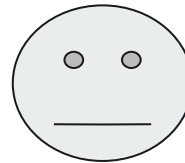


10.1 Ley de Amdahl

- Sin embargo, **pocas veces veremos ese comportamiento**. En su lugar, veremos una aceleración casi lineal con pocos procesadores y **plana a partir de cierto número**:

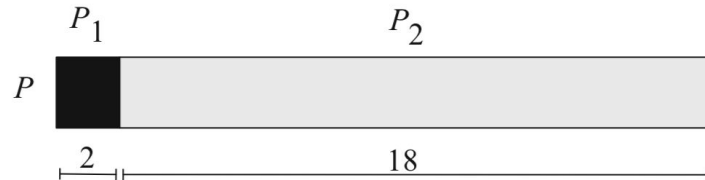
¿Por qué?

+ Lo que obtenemos:



10.1 Ley de Amdahl

- Imaginemos un programa P con 2 partes, P_1 y P_2 tal que $P = P_1 \cup P_2$
 - P_1 es una tarea secuencial pura, como analizar un directorio y hacer un listado de archivos. (2 min)
 - P_2 es una tarea paralelizable, como aplicar una función sobre cada archivo. (18 min)
- En secuencial, el tiempo de P puede mostrarse como la suma del tiempo de cada parte:





10.1 Ley de Amdahl

- Sólo P_2 puede beneficiarse de añadir procesadores, por lo que, aunque ese coste fuera prácticamente cero (logrando muy buena aceleración), **el tiempo de P nunca va a ser menor que el de la parte secuencial de P_1 .**
- Además, la propia P_2 no tiene por qué ser totalmente paralelizable, manteniendo entonces una parte secuencial.

=> La aceleración a la que podemos aspirar al mejorar una parte de un sistema está limitada por el impacto temporal de dicha parte mejorada. Por tanto, nuestra situación sería bastante peor si los tiempos de P_1 y P_2 fueran al revés (y deberíamos cambiar el enfoque).



10.1 Ley de Amdahl

- En general, podemos ver **todo programa P** como **divisible en dos grandes partes P_1 y P_2** , siendo la primera totalmente **secuencial** (no “ve” más de un procesador), y la segunda susceptible de **paralelizarse**. El tiempo de ejecución en paralelo (y la consiguiente aceleración) se ve **limitado por P_1** y las **partes secuenciales** que quedan en P_2 .
- En este contexto, vamos a intentar calcular la aceleración formalmente sobre un programa $P = P_1 \cup P_2$. Su tiempo de ejecución secuencial, T_{seq} , viene dado por:

$$T_{\text{seq}}(P) = T_{\text{seq}}(P_1) + T_{\text{seq}}(P_2)$$

10.1 Ley de Amdahl

- Si empleamos más de un procesador para ejecutar P , su parte P_2 se verá acelerada por un factor $s > 1$ (mientras que P_1 seguirá igual). De esta forma, el tiempo de ejecución paralelo de P , T_{par} , será:

$$T_{\text{par}}(P) = T_{\text{seq}}(P_1) + T_{\text{par}}(P_2) = T_{\text{seq}}(P_1) + \frac{1}{s} T_{\text{seq}}(P_2)$$

- La aceleración de P , $S(P)$, puede calcularse entonces como:

$$S(P) = \frac{T_{\text{seq}}(P)}{T_{\text{par}}(P)} = \frac{T_{\text{seq}}(P_1) + T_{\text{seq}}(P_2)}{T_{\text{seq}}(P_1) + \frac{1}{s} T_{\text{seq}}(P_2)}$$



10.1 Ley de Amdahl

- Llegados a este punto, **podemos normalizar el tiempo según el tiempo secuencial de P** como $T_{seq}(P) = T_{seq}(P_1) + T_{seq}(P_2) = 1$ (podemos verlo también como un **100%**).
- Además, podemos expresar la parte paralelizable, P_2 , en términos de P_1 , como su complementaria: $P_2 = 1 - P_1$. Teniendo esto en cuenta sobre la expresión de aceleración previa:

$$S(P) = \frac{1}{T_{seq}(P_1) + \frac{1 - T_{seq}(P_1)}{s}}$$

10.1 Ley de Amdahl

- Podemos además reescribir la expresión anterior llamando seq al tiempo de la parte secuencial pura, $seq = T_{seq}(P_1)$ y N al número de procesadores empleado ($s=N$) y con el que asumimos una aceleración s . Esta es la formulación que se suele ver:

$$S(P) = \frac{1}{seq + \frac{1 - seq}{N}}$$

Ley de Amdahl, 1967

- De aquí se deduce la parte secuencial limita la aceleración aunque $N \rightarrow \infty$: $S(P) < \frac{1}{seq}$



10.1 Ley de Amdahl

- Por consiguiente, la aceleración tiende a $1/\text{seq}$ para infinitos procesadores y llega un punto en el que se deja de obtener aceleración (y **la eficiencia tiende a cero**).
- Sólo en el caso ideal cuando $\text{seq}=0$ se tiene que $S = N$ (y $E = 1$).
- Por ejemplo, supongamos que el 70% de un programa puede acelerarse mediante paralelización y disponemos de 16 procesadores:

$$S = \frac{1}{0.3 + \frac{0.7}{16}} = 2.91$$

Para 32 procesadores, $S = 3.11$. Para 64 procesadores, $S = 3.22$. Para 128, $S = 3.27 \dots$:-(



10.1 Ley de Amdahl

- Es importante tener en cuenta que únicamente estamos hablando de aumentar procesadores, por lo que la carga de trabajo (el coste computacional de P) se asume fija en este modelo.
- En este contexto, se dice que un algoritmo es fuertemente escalable (escalabilidad fuerte) si se puede acelerar mediante la adición de procesadores.



10.2 Ley de Gustafson

- La Ley de Amdahl da una visión pesimista (exigente) de la utilidad del procesamiento paralelo, pues se centra en cómo la parte secuencial fija limita la máxima aceleración.
- Sin embargo, **no considera** que, al disponer de más recursos computacionales, podamos plantearnos abordar problemas más grandes (**variar la carga**), situación que puede darse:
 - Una estimación de Pi indefinidamente precisa
 - Un algoritmo genético con una población más grande...
 - ...
- Metáfora de la conducción: https://es.wikipedia.org/wiki/Ley_de_Gustafson

10.2 Ley de Gustafson

- De hecho, muchas veces nos plantearemos **fijar el tiempo** más que el **tamaño computacional** del problema (entrada):
 - *¿Cuánto tiempo podemos permitirnos esperar un resultado lo más preciso posible?*
- Pensemos en una situación alternativa a la anterior en la que **fijamos el tiempo** y **pretendemos maximizar el trabajo que hacemos** con N procesadores. Descomponiendo el proceso P en su parte secuencial, **seq**, y la paralelizable, **par**:
 - El tiempo en paralelo será: $seq + par = 1$ (Normalización)
 - Sin paralelismo, el trabajo realizado sería $seq + par * N$

$$S(P) = \frac{T_{seq}}{T_{par}} = \frac{seq + par * N}{1}$$



10.2 Ley de Gustafson

- En este contexto de carga variable y tiempo fijo se define la **Ley de Gustafson**:

$$S(P) = \text{seq} + N^*_{\text{par}} = \text{seq} + N^*(1 - \text{seq}) \quad \text{Ley de Gustafson, 1988}$$

- Por tanto, aunque el código tenga una parte secuencial, **la aceleración** no tiene por qué estar limitada por ella, y **puede ser proporcional a N**.
- Se dice que un algoritmo es **débilmente escalable** (escalabilidad débil) si se puede aumentar su aceleración incrementando el tamaño del problema.



10.3 Corolario

- **Ambas leyes son complementarias.**
- Debe aplicarse una u otra **según el contexto** en el que nos encontremos:
 - No siempre tiene sentido optar por un problema más grande.
 - A veces necesitamos reducir el tiempo que esperamos “a toda costa” (latencia).
- Otras veces sí: tenemos que hacer lo máximo posible en un tiempo fijado...

10.3 Corolario

- De hecho, ambas leyes se pueden **anar** usando una variable auxiliar, α ($\alpha > 0$):

$$S(P) = \frac{seq + parN^\alpha}{seq + parN^{\alpha-1}} = \frac{seq + (1 - seq)N^\alpha}{seq + (1 - seq)N^{\alpha-1}}$$

Si $\alpha=0$: ☐ Ley de Amdahl

$$\frac{seq + parN^0}{seq + \frac{par}{N}} = \frac{1}{seq + \frac{(1 - seq)}{N}}$$

Si $0 < \alpha < 1$ y $N \gg 1$:

$$S(P) = 1 + \frac{par}{seq} N^\alpha$$

Barrera de Amdahl superada:
S(P) proporcional a N

Si $\alpha=1$: ☐ Ley de Gustafson


$$\frac{seq + (1 - seq)N}{seq + (1 - seq)N^0} = seq + (1 - seq)N$$



10.3 Corolario

- En cualquier caso, hay que tener en cuenta que se definen en **condiciones ideales** (como igualar el tiempo paralelo al secuencial dividido por el número de procesadores). Sin embargo, al **paralelizar se esperan**:

- Coste de creación/inicialización
- Comunicación
- Sincronización
- Desbalanceo de carga




OVERHEAD, que se reduce maximizando la relación cómputo/comunicación, evitando sincronismo y balanceando la carga



10.4 Ejercicio

Un programa se ejecuta en 40 segundos en un sistema con múltiples procesadores. Un 20% del tiempo ha usado 4 de los procesadores y un 60% del tiempo ha utilizado 3. El 20% restante su ejecución ha sido en un único procesador. Asumimos que la carga se balancea de forma equitativa en todo momento y obviamos el *overhead*. ¿Cuánto tiempo tardaría en ejecutarse en un solo procesador? ¿Qué aceleración estamos obteniendo? ¿Y qué eficiencia?

11 Mejorando prestaciones

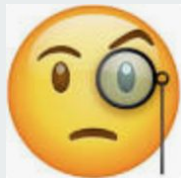


Cosas básicas que deberíamos hacer y no siempre hacemos

- Compilar con opciones de optimización -O2, -O3...
- No utilizar variables innecesarias o tenerlas declaradas desde el principio para usarlas solo en una parte pequeña del código.
- Las variables globales admiten tamaños mayores que las dinámicas o las locales
- Usar operaciones vectoriales
- Desenrollar bucles
- Generar iteraciones en bucles independientes

11 Mejorando prestaciones

Cosas básicas que deberíamos hacer y no siempre hacemos



En cualquier caso, no olvidemos estas dos máximas:

- *First, make it work. Then, make it fast.*
- *Premature optimization is the root of evil!*

- Utilizar funciones inline (Inlining)
- Alineación de variables en memoria (Aliasing)
- Utilización de registros para datos muy utilizados (register)
- Utilizar variables de precisión adecuada
- Optimización de accesos a memoria evitando fallos de página

11 Mejorando prestaciones

No repetir trabajo

```
1 Logical:: FLAG
2 FLAG = false
3 Do i=1, N{
4     If(complex(A(i)) < Threshold){
5         FLAG = true
6     }
7 }
```

```
1 Logical:: FLAG
2 FLAG = false
3 Do i=1, N{
4     If(complex(A(i)) < Threshold){
5         FLAG = true
6         Exit()
7     }
8 }
```

11 Mejorando prestaciones

Optimizar/aproximar cálculos, y evitar llamadas a función

Problema

```
Y= Pow(x,2);

Integer::iL,iR,iU,iO,iS,iN
Double precision::edez,tt
Loop
...
edez=iL+iR+iU+iO+iS+iN
BF=0.5d0*(1.d0+THNH(edez/tt));
```

Solución

```
Y= x*x;

Double precision, dimension(-6:6)::tanh_table
Integer::iL,iR,iU,iO,iS,iN
Double precision::tt
do i=-6,6
    tanh_table(i) = 0.5d0*(1.d0+THNH(dble(i)/tt));
enddo
Loop
...
BF=tanh_table(iL+iR+iU+iO+iS+iN);
```

- Si el conjunto de variables activas es mayor que la caché de mayor nivel => fallos de página
- Si nos basta 1 byte, no usar variables que ocupen 4 (especialmente si la máquina direcciona así)...
- Si es un cuadrado, podemos hacerlo simplemente multiplicando, y evitamos una función externa
- Senos, cosenos, tangentes... según el caso podemos usar aproximaciones (p.e.j. polinomios de Taylor) y/o pre-calcular y almacenar el rango que necesitamos para ahorrar cálculos.

11 Mejorando prestaciones

Almacenar cálculos recurrentes

```
Do i=1,N  
    A(i)=A(i)+s+r*sin(x)  
enddo
```

```
tmp=s+r*sin(x)  
Do i=1,N  
    A(i)=A(i)+tmp  
enddo
```

11 Mejorando prestaciones



```
do j=1,N
  do i=1,N
    if(i>=j) then
      sign=1;
    else if(i<j)then
      sign=-1;
    else
      sign=0;
    endif
    C(j)=C(j)+sign*A(i,j)*B(i)
  enddo
enddo
```

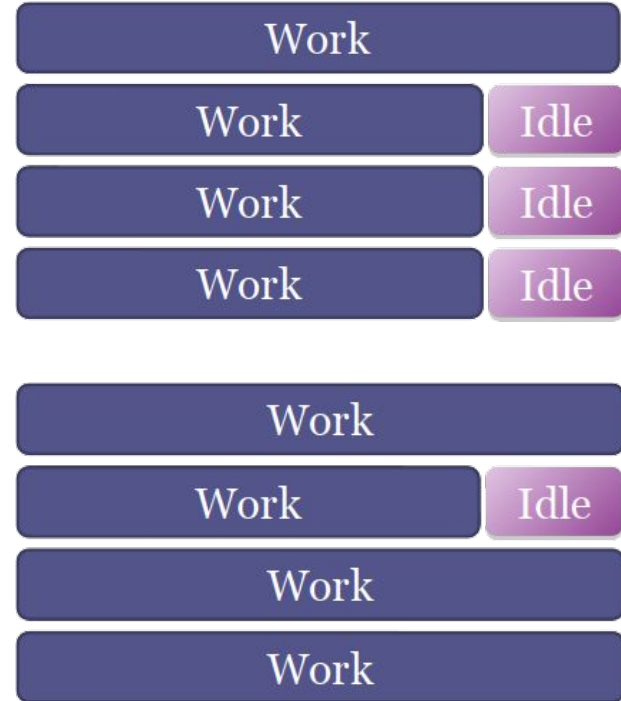
```
do j=1,N
  do i=j+1,N
    C(j)=C(j)+A(i,j)*B(i)
  enddo
enddo
do j=1,N
  do i=1,j-1
    C(j)=C(j)-A(i,j)*B(i)
  enddo
enddo
```

11 Mejorando prestaciones

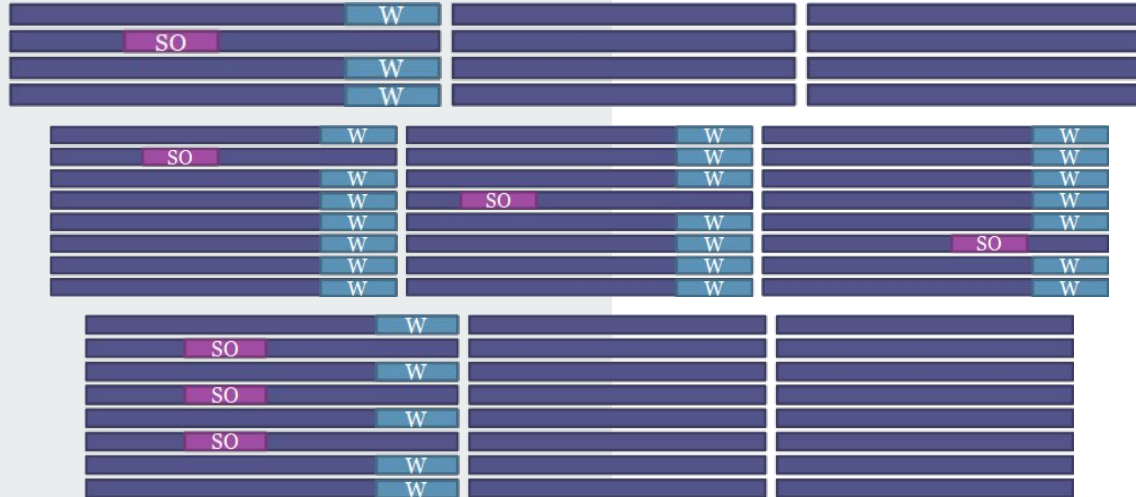


Profiling y Tracing para buscar

- Cuellos de botella
- Pérdidas de tiempo en sincronización
- Sobrecarga en las comunicaciones
- ...



11 Mejorando prestaciones



- Si la cantidad de trabajo a paralelizar es dinámica, lo que puede dar lugar a desbalances
- La granularidad puede ser inadecuada
- Las unidades funcionales específicas, como multiplicadores, están ocupadas
- Acciones del SO (Jitter)



Gracias.

