Introducción

Programación en Paralelo

ICME Summer Workshop @ Santiago: Fundamentals of Data Science

Cindy Orozco

INSTITUTO DE INGENIERÍA MATEMÁTICA Y COMPUTACIONAL
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

Introducción Programación en Paralelo

Este curso es:

Un abrebocas de las soluciones clásicas para programar con múltiples unidades de procesamiento a la vez, mostrando la interrelación entre Hardware, paradigmas de programación y naturaleza del problema a solucionar.

Sin embargo no intenta presentar una descripción extensa de todos los tipos de paralelismo

Introducción Programación en Paralelo

Martes 9 Enero 2:00 – 3:30: Introducción a Arquitecturas en Paralelo

- ¿Qué es Programación en Paralelo? ¿Por qué es necesaria?
- Componentes: Memoria, Procesadores, Redes.
- Medida de desempeño

Martes 9 Enero 4:00 – 5:30: Sistemas de Memoria Compartida

- OpenMP: Implementación y errores comunes.

Miércoles 10 Enero 2:00 – 3:30: Sistemas de Memoria Distribuida

- Interfaz de Paso de Mensajes (MPI): Tipos de comunicación, requerimientos y errores comunes

Miércoles 10 Enero 4:00 – 5:30: Sesión Práctica: Solución ecuación de Poisson

- Comparación práctica entre paradigmas midiendo su desempeño.

Introducción Programación en Paralelo

- Cada clase terminará con una sesión práctica.
- Para esto se utilizará una máquina virtual con Linux y todo el software instalado.
- Toda la información del curso está en Github

https://github.com/cindyrella/Parallel_Computing

Introducción
Arquitecturas en Paralelo

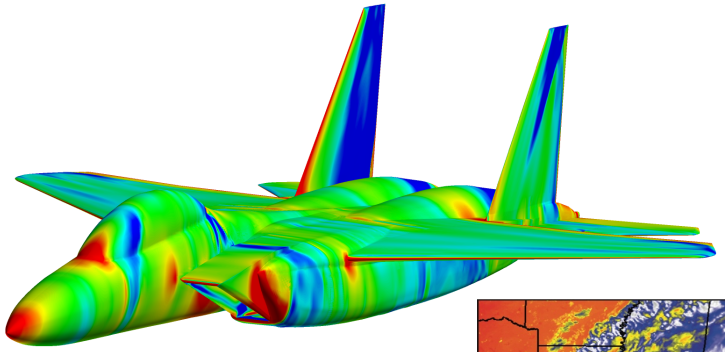
Programación en Paralelo

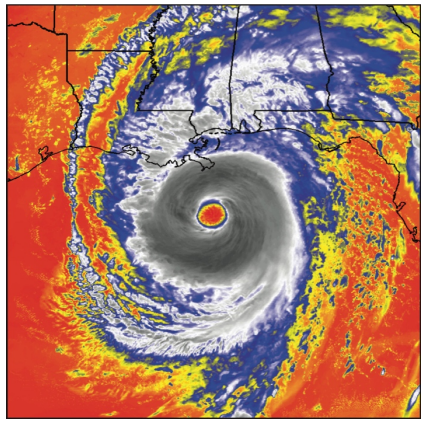
ICME Summer Workshop @ Santiago: Fundamentals of Data Science

Cindy Orozco

INSTITUTO DE INGENIERÍA MATEMÁTICA Y COMPUTACIONAL
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

Programación en Paralelo





Enorme Cantidad

Datos / Parámetros

Extremadamente Costosos

Algoritmos: Simulaciones / Análisis

Resultados

Programación en Paralelo

¿Qué es?

Uso de **múltiples procesadores** (CPUs, GPUs, …) para solucionar un problema

¿Por qué es útil?

Mayor cantidad de **operaciones** por segundo: múltiples unidades trabajando simultáneamente

*Reduce el **tiempo** total de ejecución*

Mayor **memoria** disponible: cada unidad cuenta con su memoria propia

*Incrementa el **tamaño** de problemas a solucionar*

SOLUCIÓN PROBLEMAS COMPLEJOS = SIMULACIÓN FIEL DE LA REALIDAD

Programación en Paralelo

Requiere una modificación del **código**, diferente para cada lenguaje

No soluciona problemas de un código serial **ineficiente**

Sin embargo

La comunicación entre múltiples unidades genera un **sobrecosto** (overhead)

El tiempo total se encuentra dominado por el trabajador más lento: **sincronizar** genera retrasos

No todos los **algoritmos** se pueden paralelizar

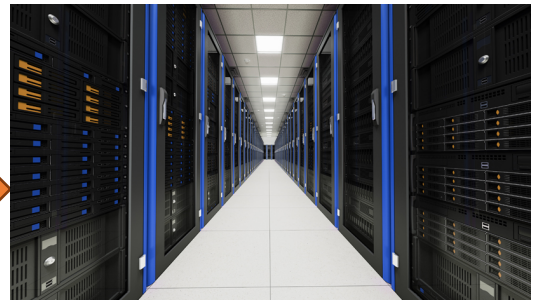
Debemos encontrar un equilibrio entre Hardware disponible y la aplicación

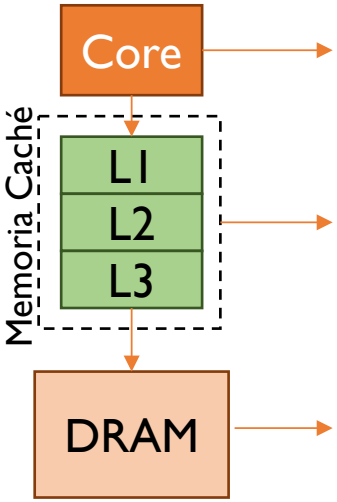
Hardware

¿… ?

Hardware

¿… ?



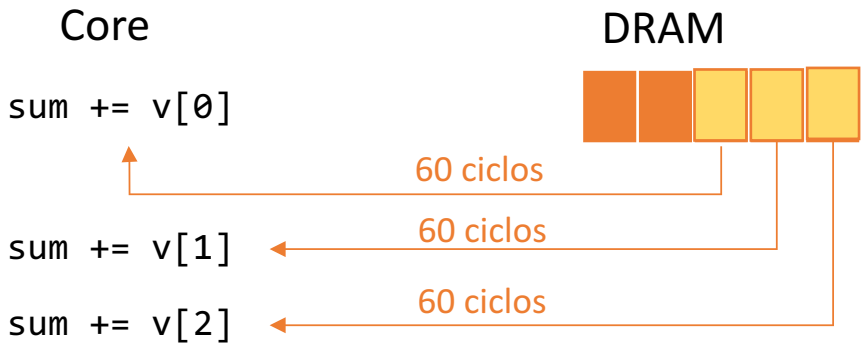


Procesamiento: 1-32 operaciones/ciclo

Memoria de fácil acceso : latencia = L1 2-3 ciclos
L2 8-10 ciclos

Memoria principal : latencia = 60-200 ciclos

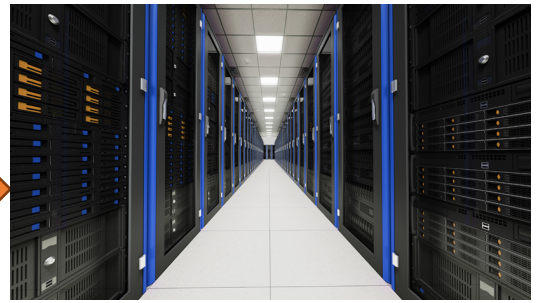
```
int sum = 0;
std::vector<int> v = {1,2,3,4,5};
…
for (int i = 0; i< v.size(); i++){
    sum += v[i];
}
```

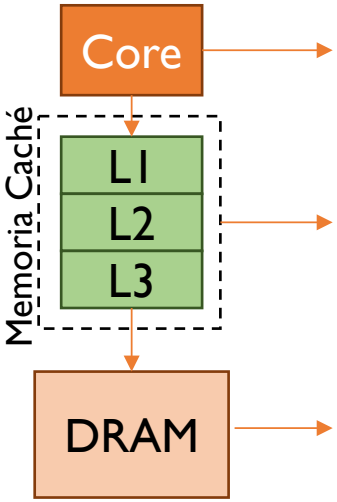


Latencia: tiempo necesario para establecer comunicación de 0 bytes

Hardware

¿… ?



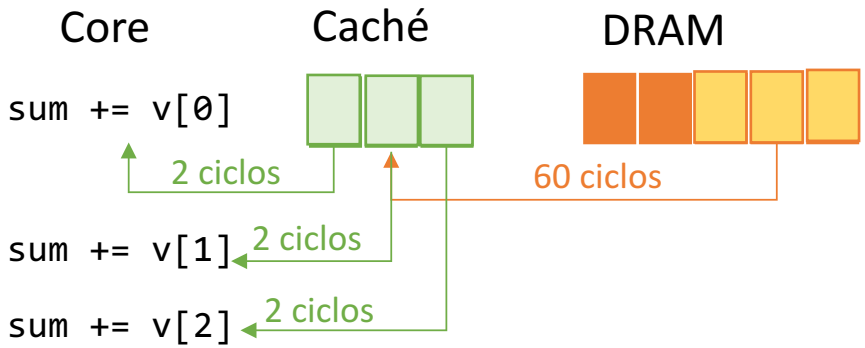


Procesamiento: 1-32 operaciones/ciclo

Memoria de fácil acceso : latencia = L1 2-3 ciclos
L2 8-10 ciclos

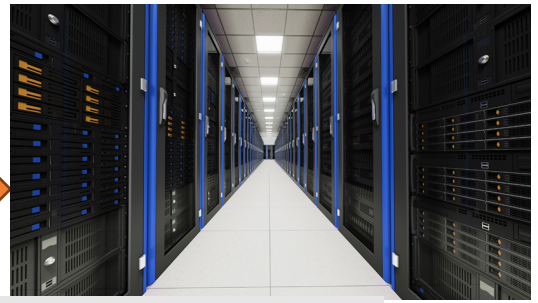
Memoria principal : latencia = 60-200 ciclos

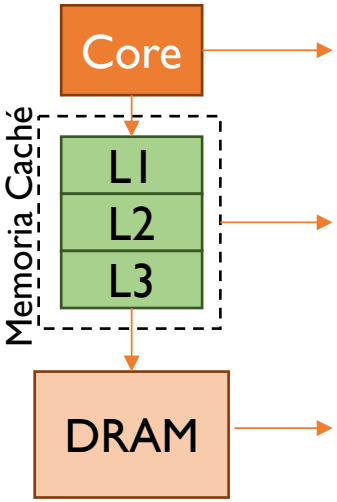
```
int sum = 0;
std::vector<int> v = {1,2,3,4,5};
…
for (int i = 0; i< v.size(); i++){
    sum += v[i];
}
```



Hardware

¿… ?



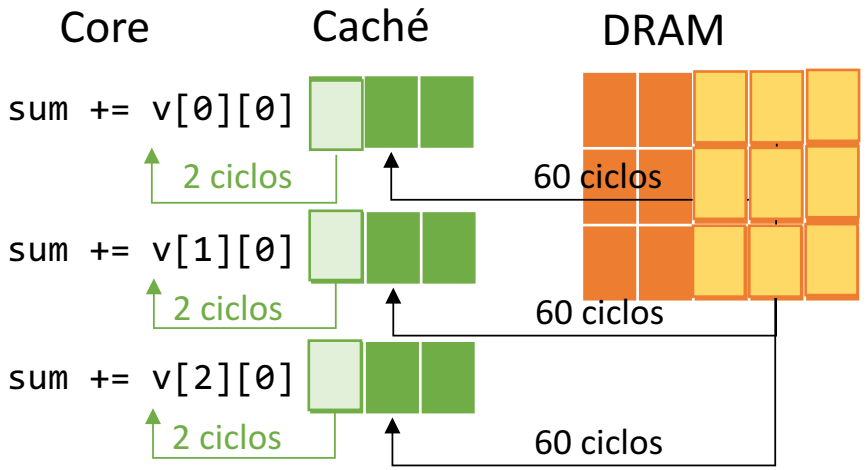


Procesamiento: 1-32 operaciones/ciclo

Memoria de fácil acceso : latencia = L1 2-3 ciclos
L2 8-10 ciclos

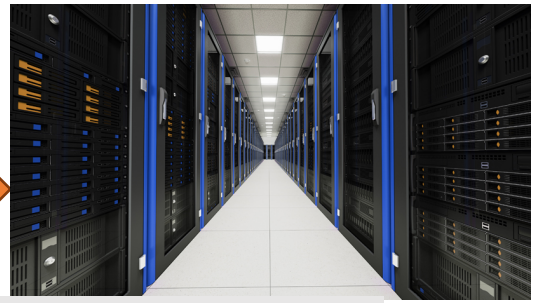
Memoria principal : latencia = 60-200 ciclos

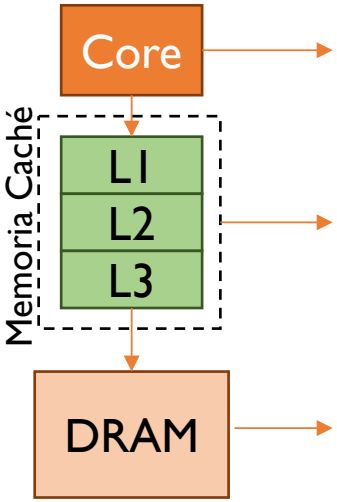
```
int sum = 0;
std::vector<std::vector<int>> v;
…
for (int j = 0; j<ncols; ++j){
    for (int i = 0; i<nrows; ++i)
        sum += v[i][j];
}
```



Hardware

¿… ?



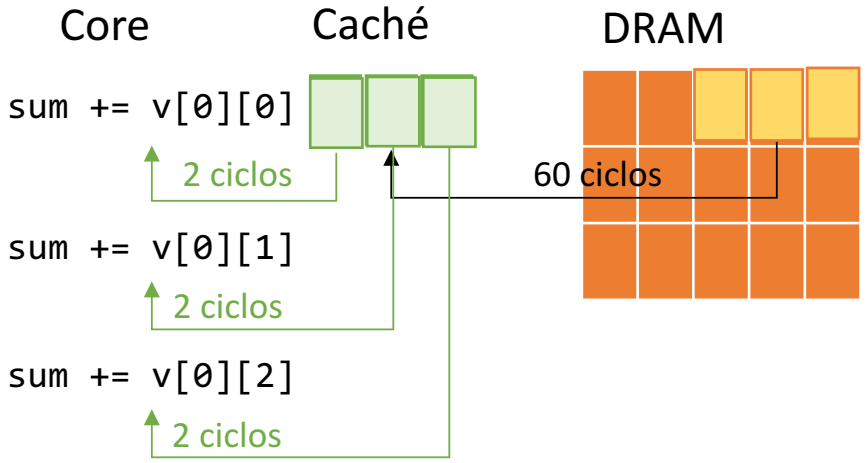


Procesamiento: 1-32 operaciones/ciclo

Memoria de fácil acceso : latencia = L1 2-3 ciclos
L2 8-10 ciclos

Memoria principal : latencia = 60-200 ciclos

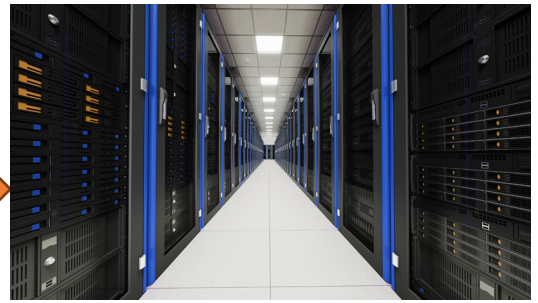
```
int sum = 0;
std::vector<std::vector<int>> v;
…
for (int i = 0; i<nrows; ++i){
    for (int j = 0; j<ncols; ++j)
        sum += v[i][j];
}
```

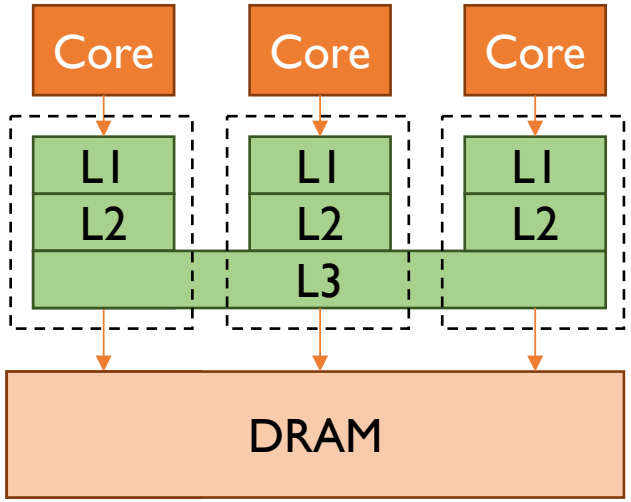


Adecuado uso de la memoria caché permite esconder la latencia en comunicación

Hardware

¿… ?

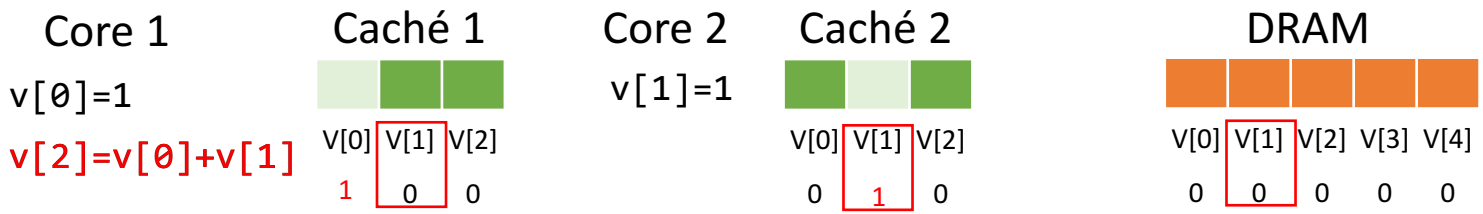




Cada procesador puede ejecutar una instrucción diferente:
Multithreading (multihilo)

Todos los core tienen igual acceso a la memoria principal:
Acceso Uniforme de Memoria (UMA)

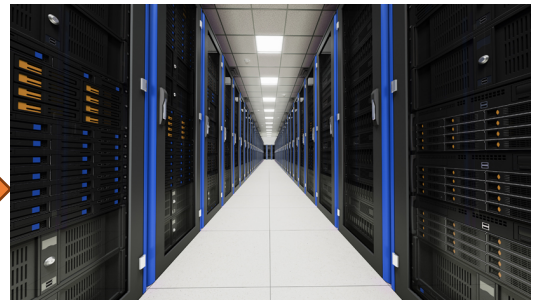
Cada core tiene diferente caché, es necesaria la **coherencia del caché**

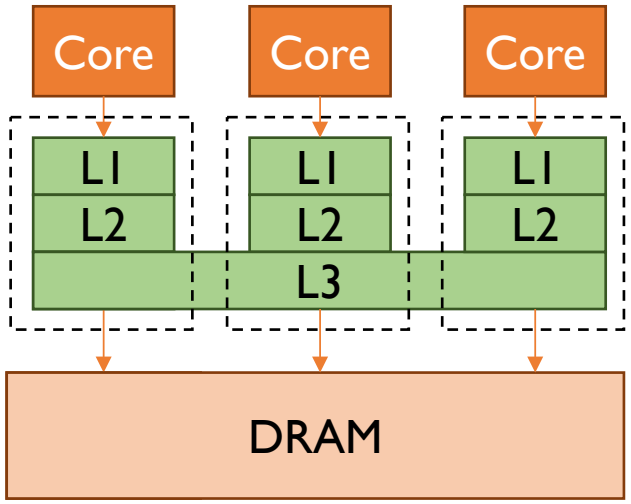


¿Qué valor de v[1] debemos tomar?

Hardware

¿… ?





Cada procesador puede ejecutar una instrucción diferente:
Multithreading (multihilo)

Todos los core tienen igual acceso a la memoria principal:
Acceso Uniforme de Memoria (UMA)

Cada core tiene diferente caché, es necesaria la **coherencia del caché**

Pueden existir conflictos de memoria: **Race Condition** (Condición de carrera)

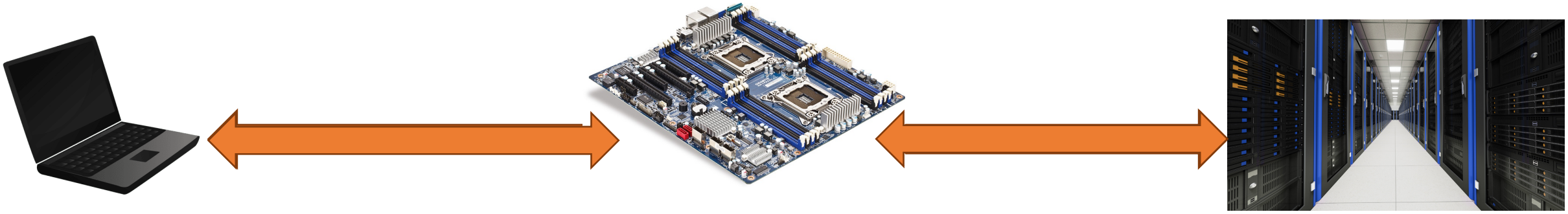
Core 1 Caché 1 Core 2 Caché 2 DRAM

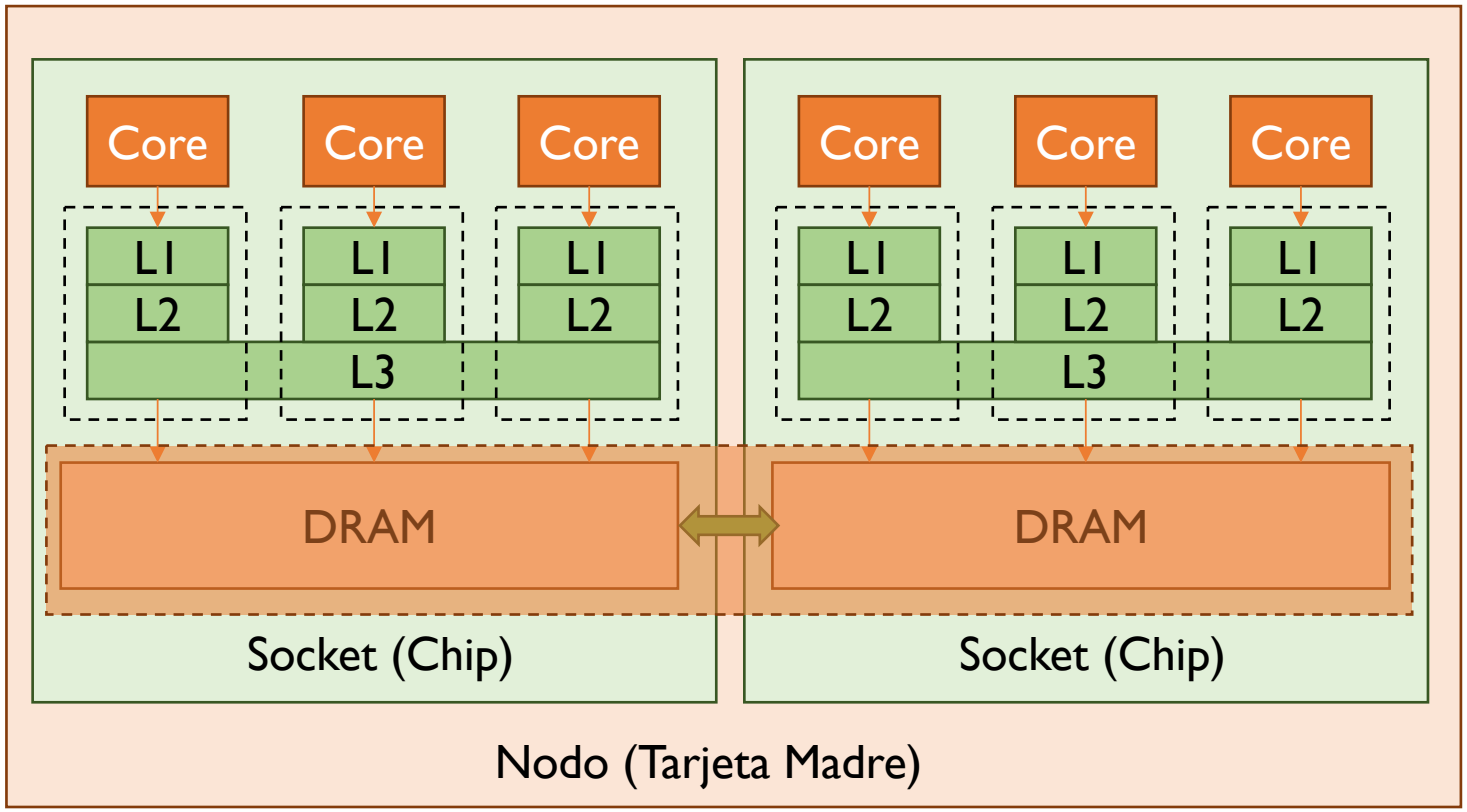
`v[1]=1`

`v[1]=2`

¿Qué valor de v[1] debemos tomar?

Hardware



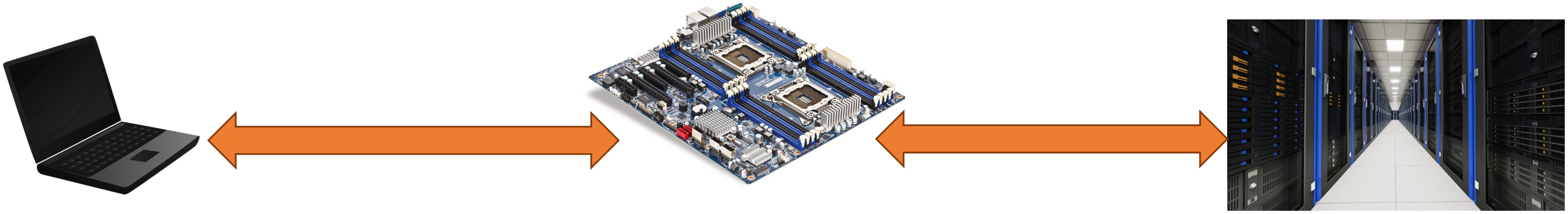


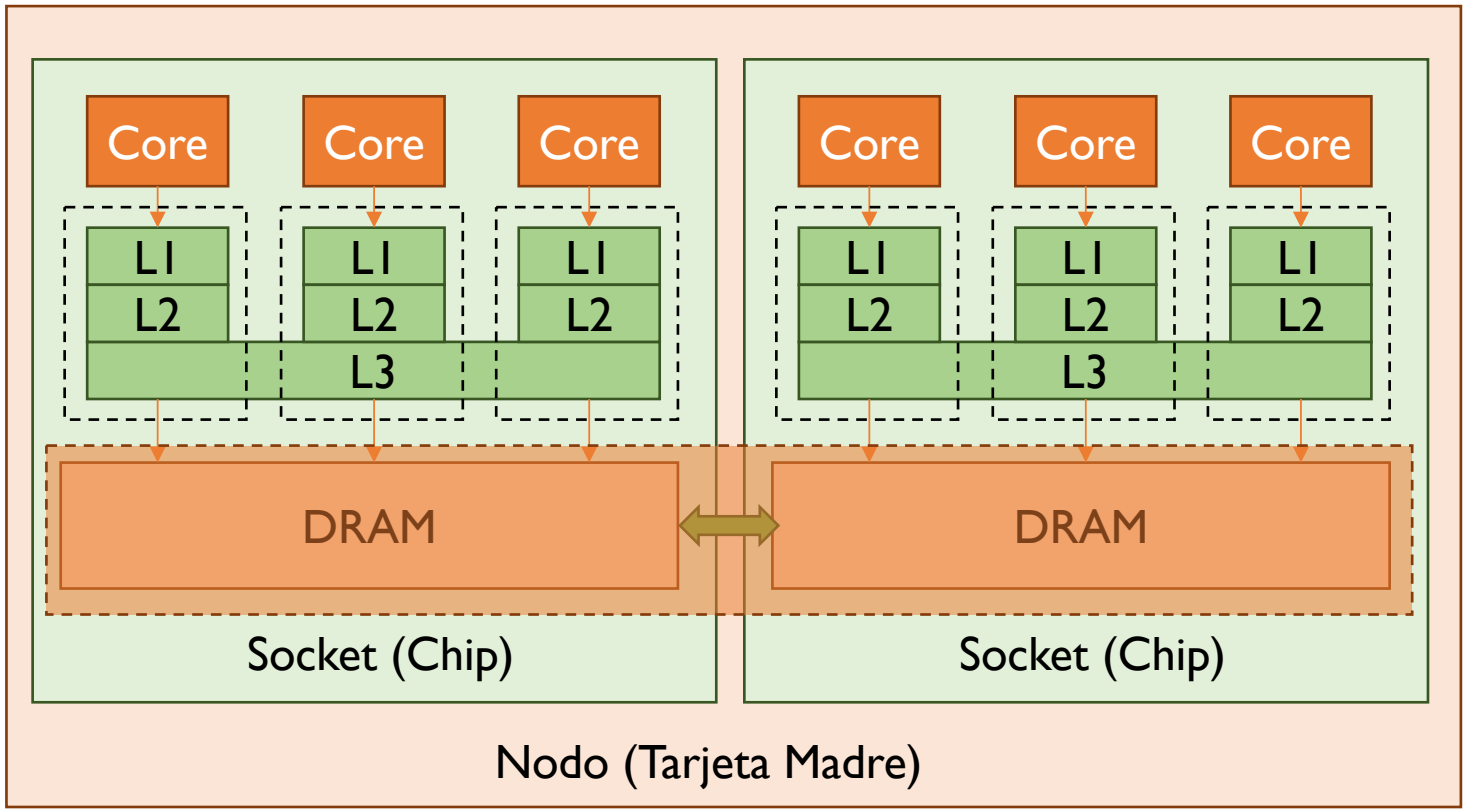
Todos los cores tienen acceso a toda la memoria DRAM. Pero **no uniformemente:**
Acceso no uniforme de memoria (**NUMA**)

La comunicación es interna en la tarjeta.

No existe restricción de acceso a cualquier dirección de memoria, pero la **latencia es diferente** dependiendo del socket.

Hardware





1) Memoria Compartida

- Multithreading (múltiples instrucciones)
- Multicore: UMA = Acceso Uniforme
- Multisocket: NUMA = Acceso No Uniforme => Diferente latencia
- Necesaria coherencia Caché
- Posible tener condiciones de carrera

Necesitamos paradigma para:

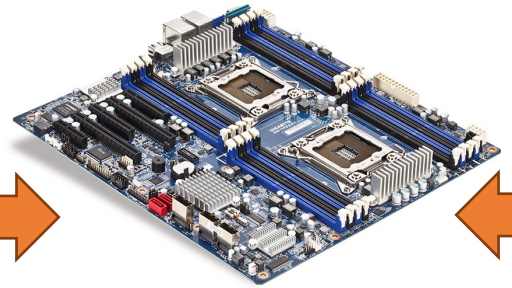
✓ *Distribuir funciones entre cores*

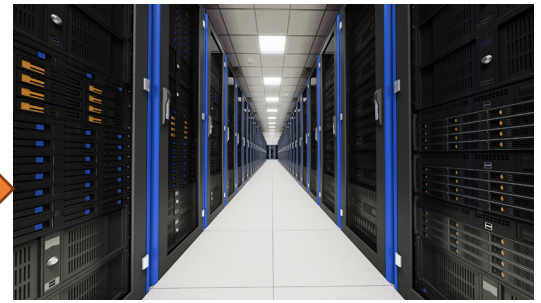
✓ *Evitar conflictos de memoria*

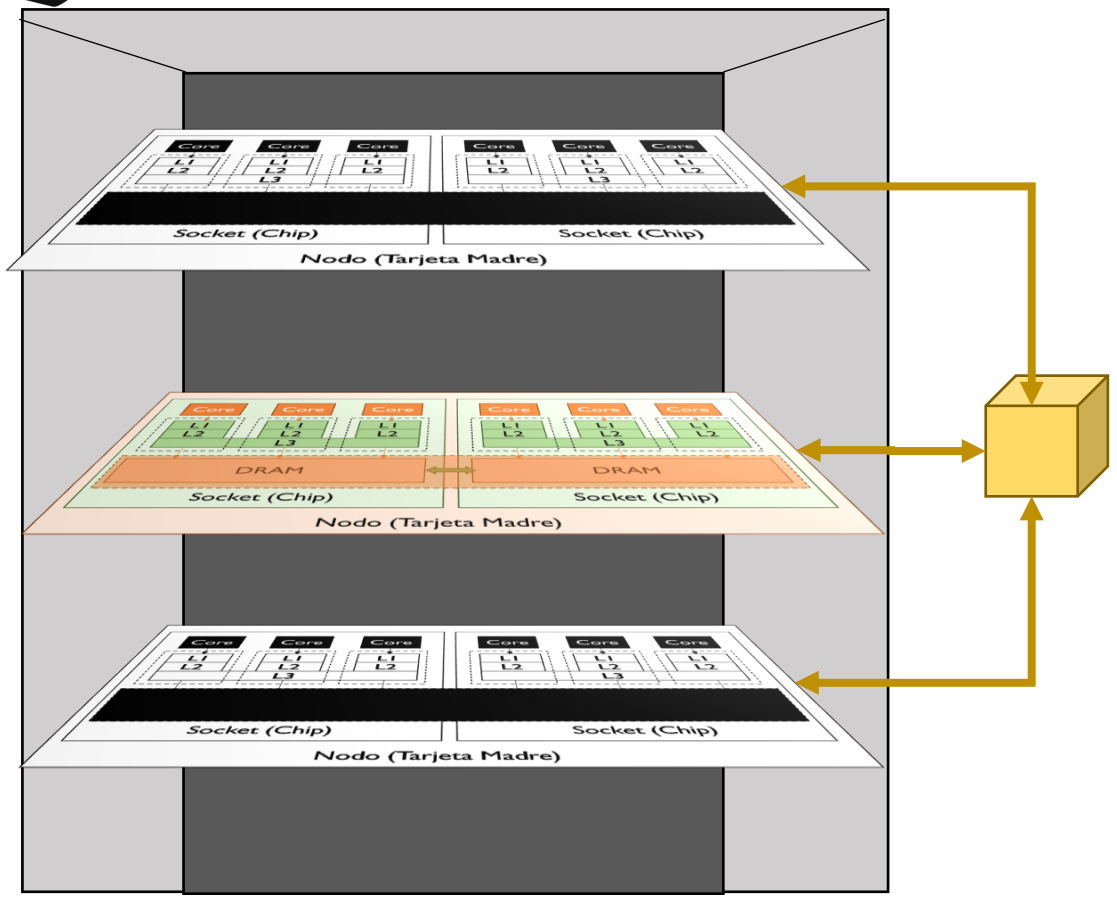
OpenMP

(Interfaz de Programación de aplicaciones)

Hardware



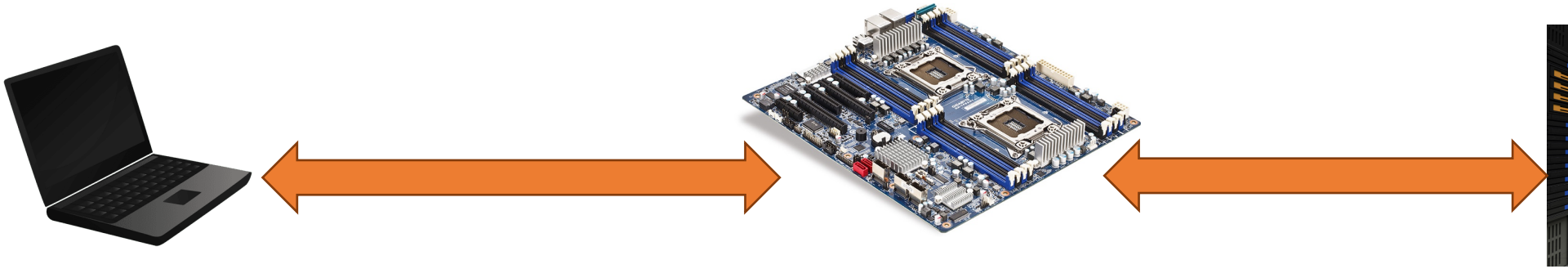


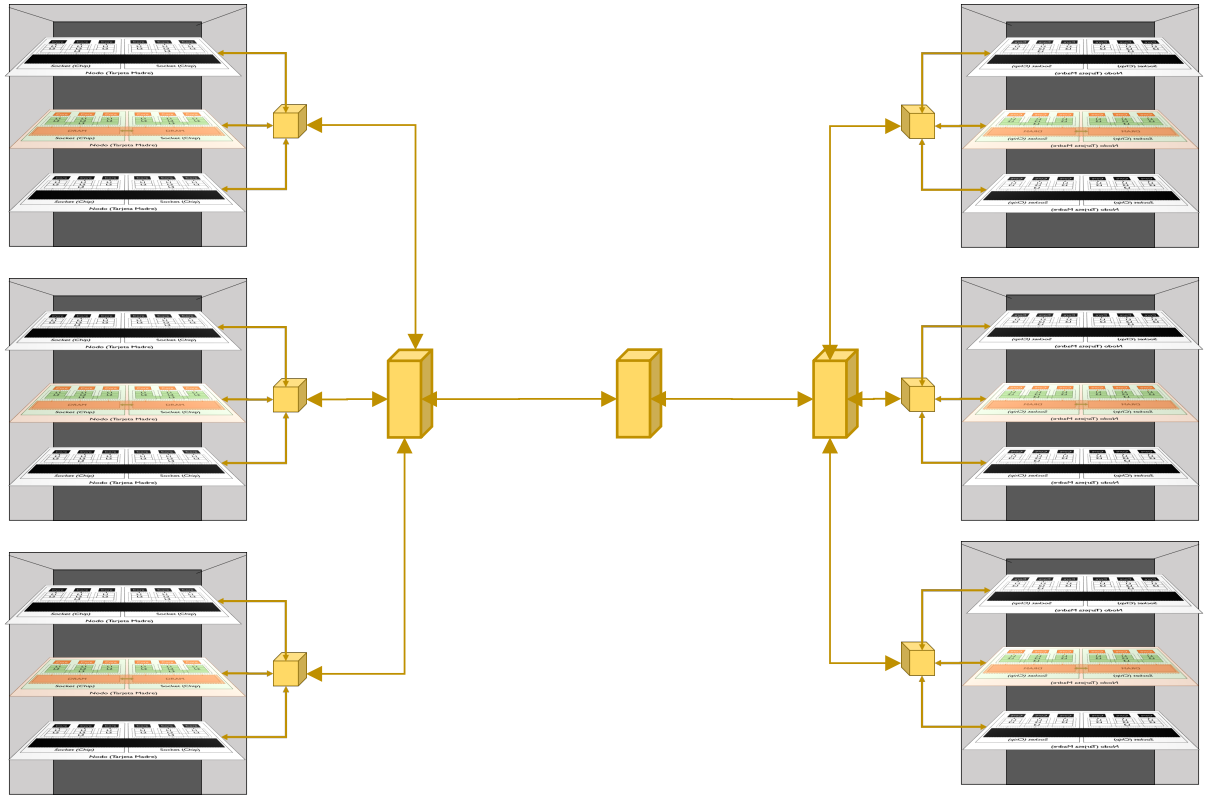


Múltiples Tarjetas madres son ensambladas en **rack** (torre)

Se conectan por red de comunicación usando **switch**

Hardware





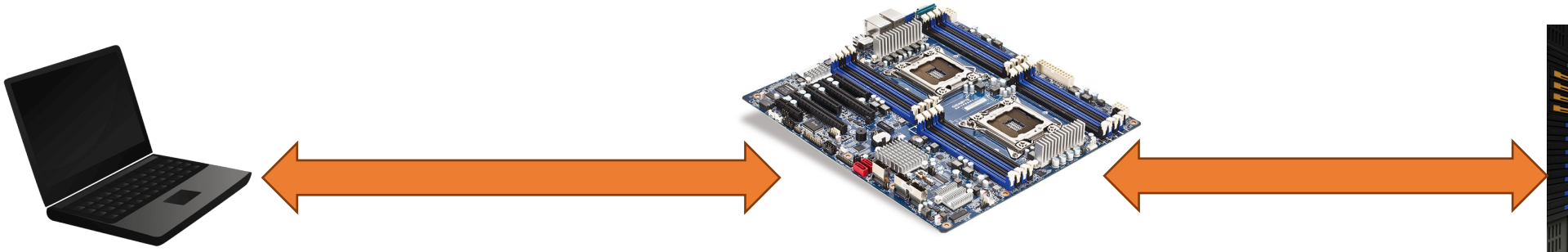
Múltiples Tarjetas madres son ensambladas en **rack** (torre)

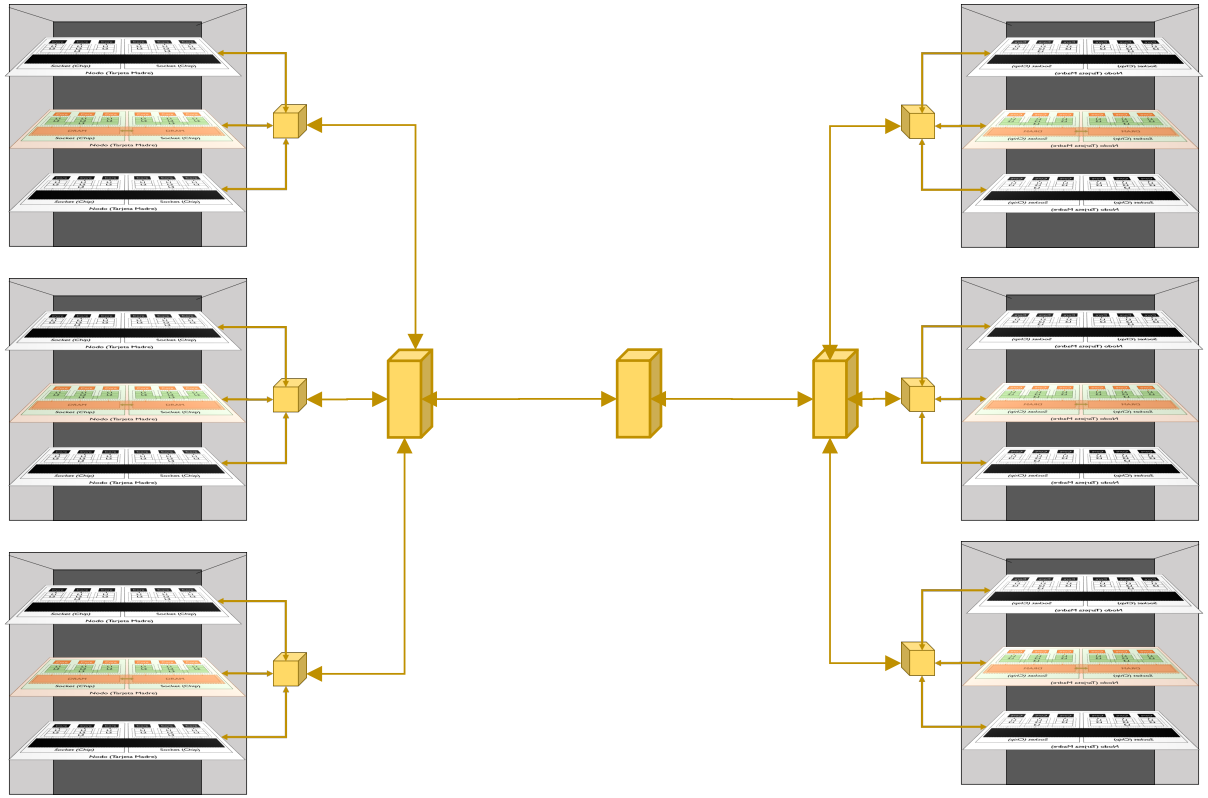
Se conectan por red de comunicación usando **switch**

El # swtiches entre nodos depende de **topología**: Fat tree, Hipercubo, Malla

La eficacia de comunicación depende de **protocolo**: Ethernet, Infiniband

Hardware





2) Memoria distribuida

- Nodos independientes: memoria DRAM propia, instrucciones propias
- Uso **explicito** de la red para saber de otros nodos

Necesitamos paradigma para:

✓ *Coordinar la comunicación de mensajes entre nodos*

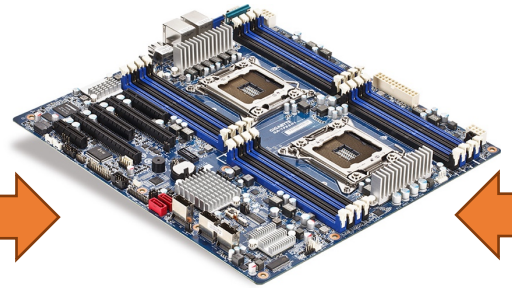
MPI

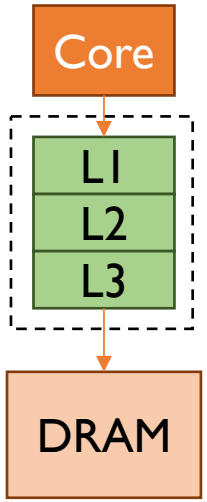
(Interfaz de Paso de Mensajes)

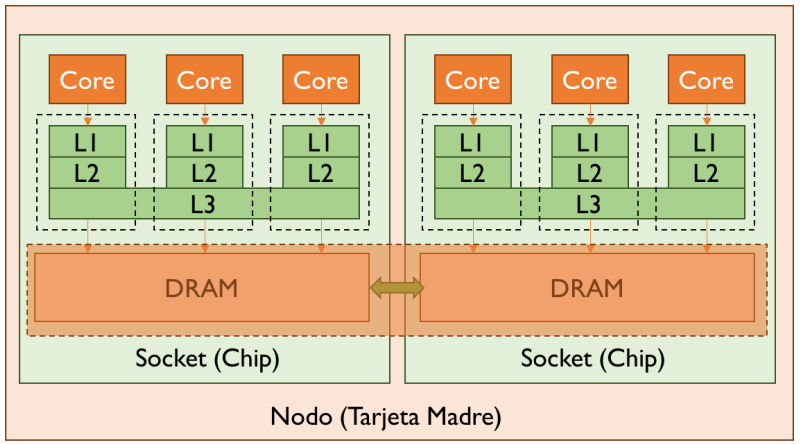
✓ *Coordinar las instrucciones entre nodos*

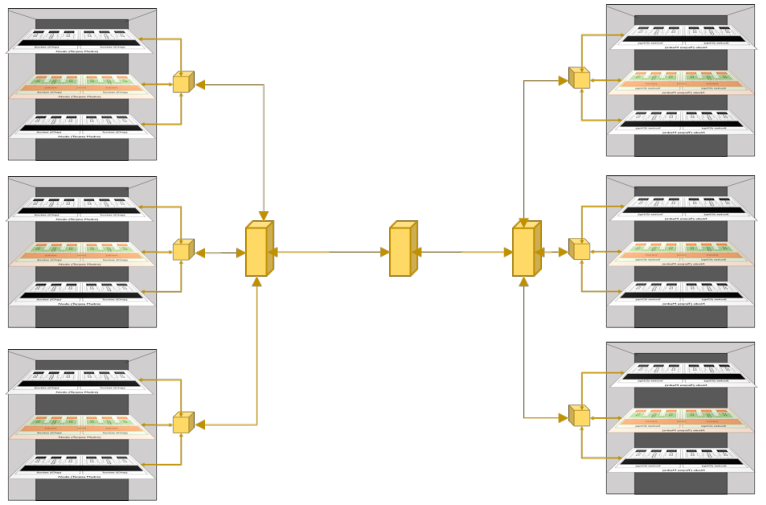
Planificador de Tareas (scheduler): SLURM

Hardware









0) Procesamiento Secuencial

1) Memoria Compartida

- ✓ Evitar conflictos memoria
- ✓ Distribuir Tareas

OpenMP

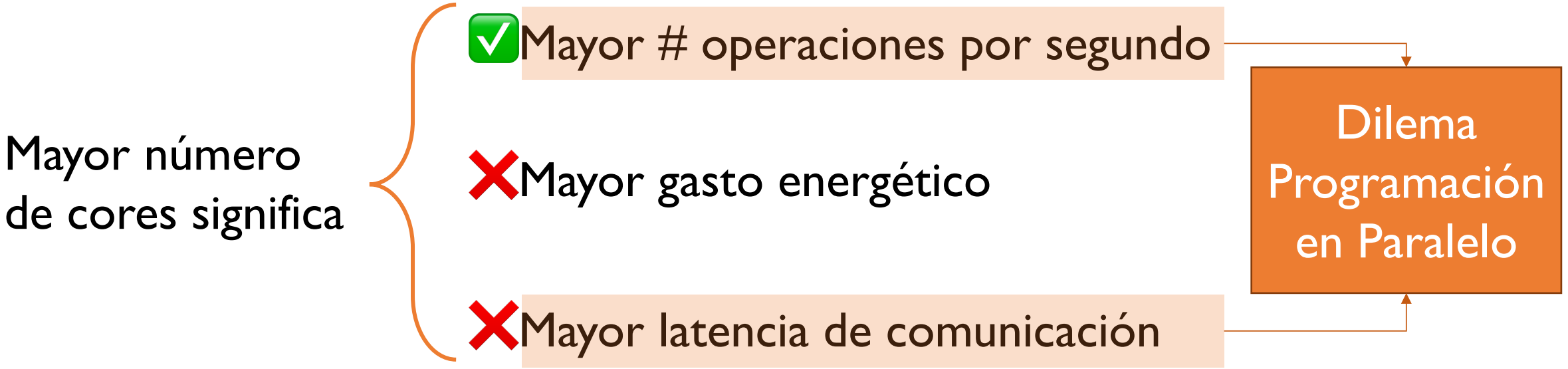
2) Memoria Distribuida

- ✓ Comunicación entre nodos — **MPI**
- ✓ Asignar Tareas — **Planificador de Tareas**

Hardware

Ya vimos que el **tipo de memoria** define **paradigma de programación** pero si tenemos múltiples arquitecturas, ¿Cómo escoger la mejor?

¿La mejor significa la que tiene mayor número de cores?



Rendimiento: Medición

Idealmente, si tenemos una aplicación que dura T_{serial} en un solo procesador, al dividirla entre p procesadores durará T_{serial}/p

✓ Speedup $$S_p = \frac{T_{serial}}{T_{paralelo}}$$

Ganancia total $$v.s. \frac{T_{serial}}{T_{serial}/p} = p$$

✓ Eficiencia $$E_p = \frac{S_p}{p}$$

Ganancia por procesador $$v.s. \frac{p}{p} = 1$$

Rendimiento: Ley de Amdahl

$$T_{paralelo} \approx T_{serial}\left(\frac{f}{p} + 1 - f\right)$$

$$f = Fracción\ Paralelizable$$

$$1 - f = Fracción\ No\ Paralelizable$$

$$p = Num\ Procesadores$$

Usualmente no el 100% de una aplicación es paralelizable

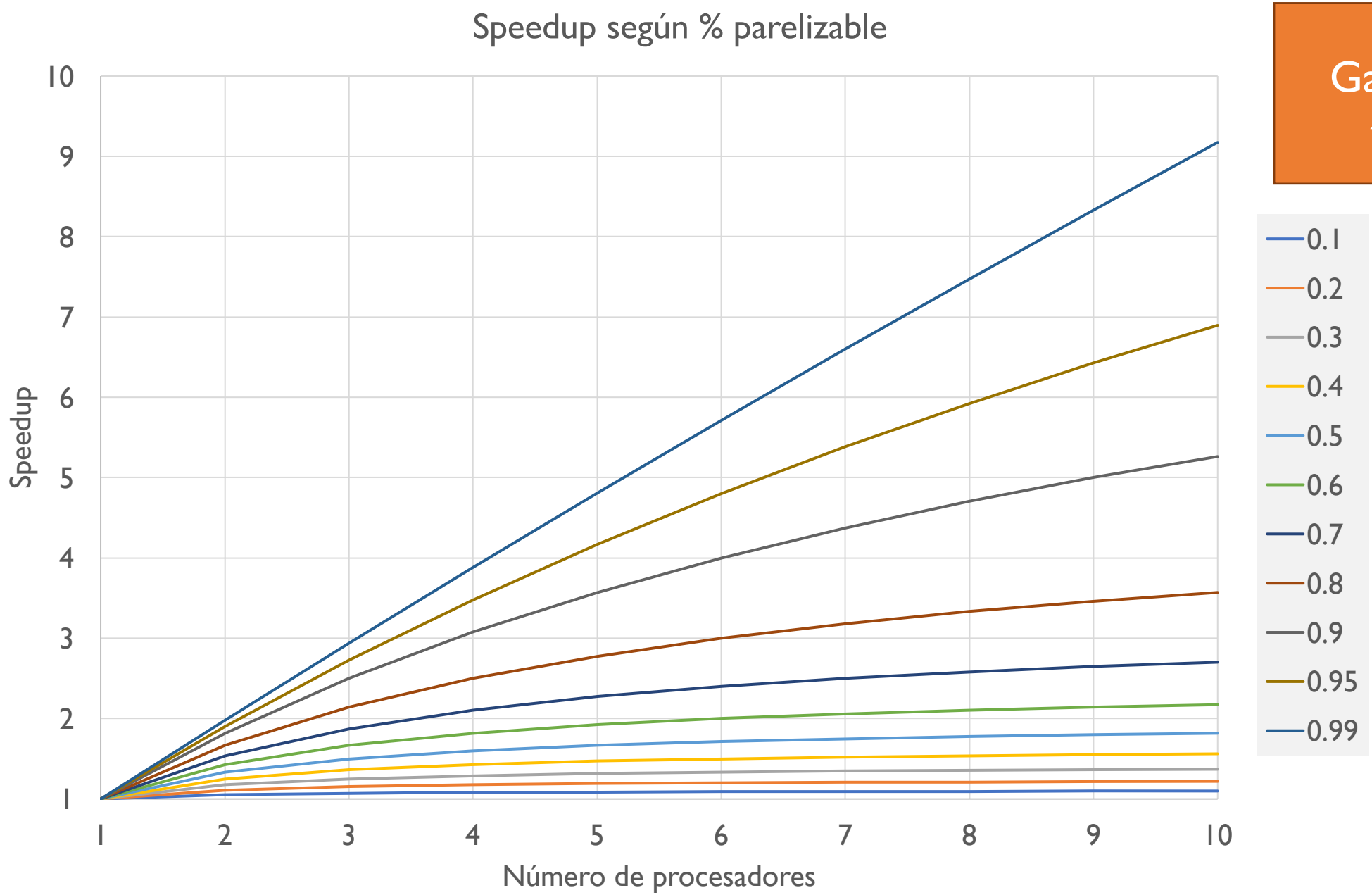
Rendimiento: Ley de Ware

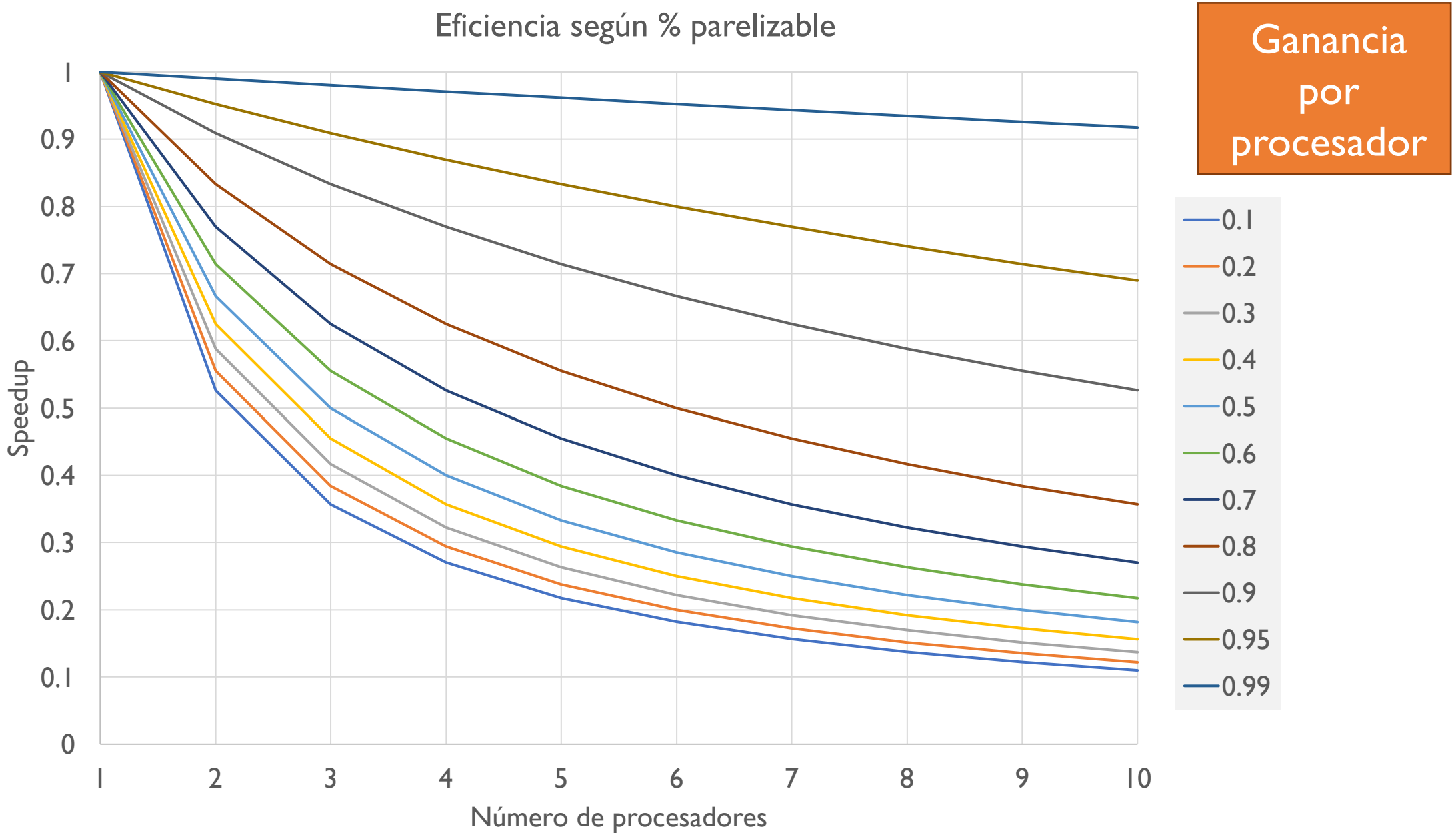
$$T_{paralelo} \approx T_{serial}\left(\frac{f}{p} + 1 - f\right)$$

✓ Speedup $S_p = \frac{T_{serial}}{T_{paralelo}} = \frac{p}{f + (1-f)p}$ $v.s.$ p

✓ Eficiencia $E_p = \frac{S_p}{p} = \frac{1}{f + (1-f)p}$ $v.s.$ 1

Ganancia total



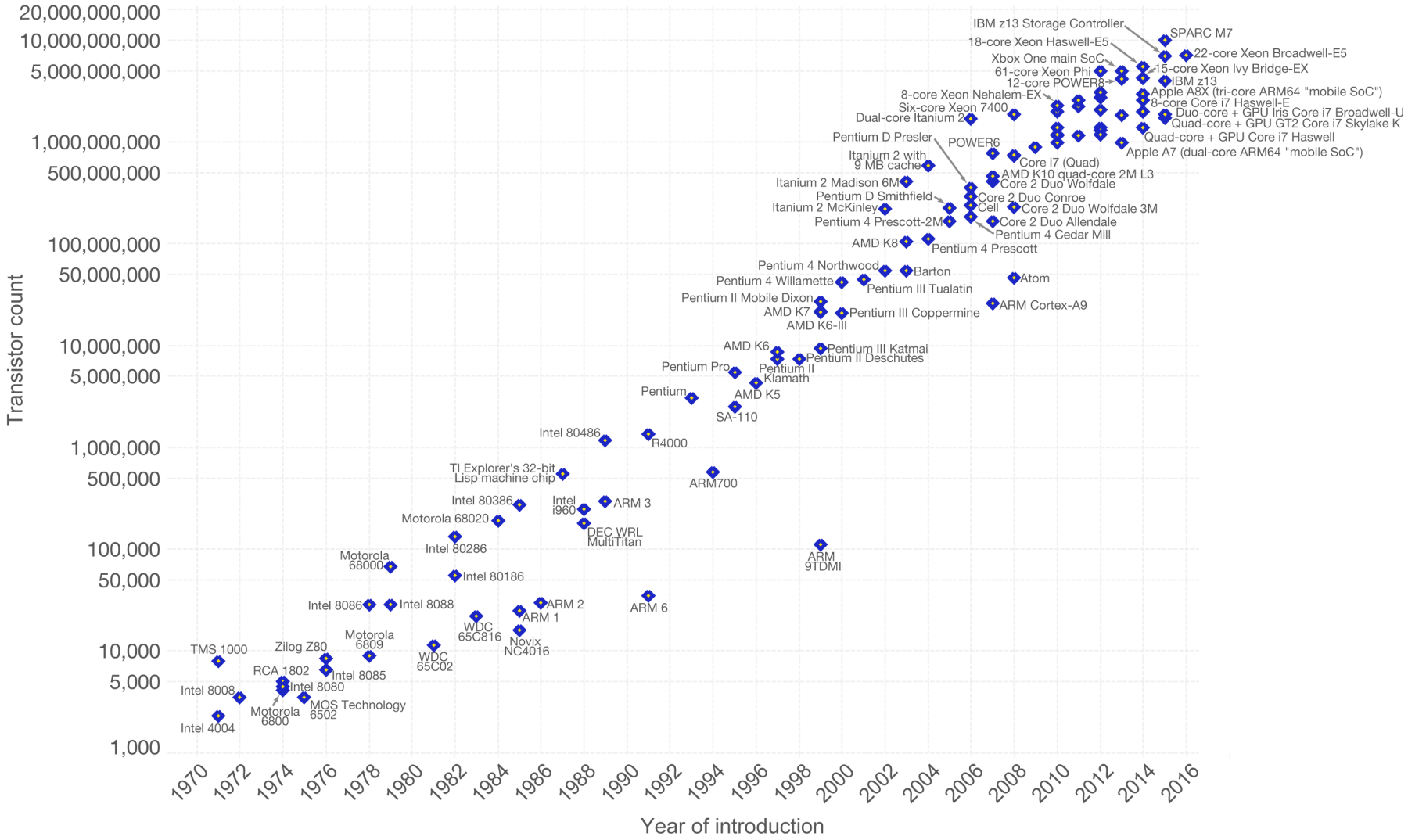


Ganancia por procesador

Aunque estos modelos son teóricos y sólo analizan el procesamiento, nos dan una idea clara que no necesariamente a más procesadores, mejor rendimiento.

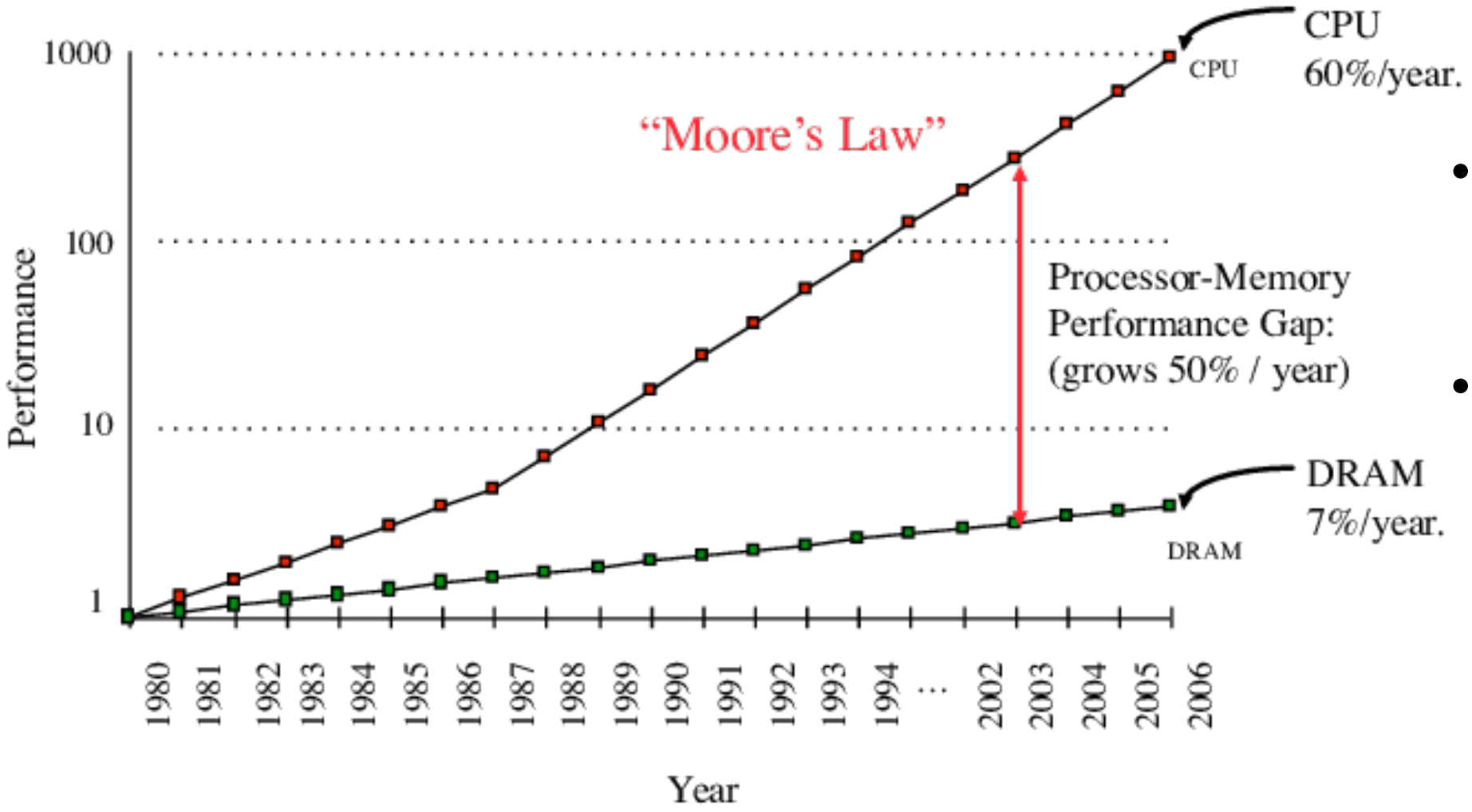
Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Ley de Moore



- Procesamiento incrementa exponencialmente
- Comunicación incrementa linealmente.

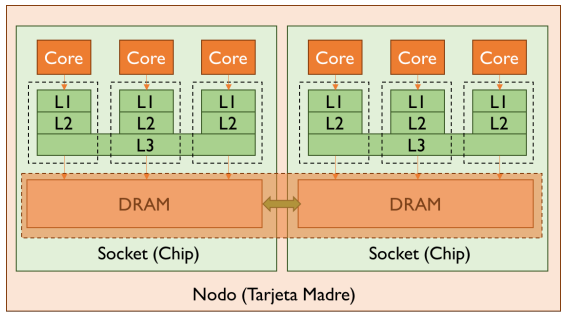
Comunicación

✓Latencia Costo de comunicación de 0 Bytes

✓Ancho de Banda Tasa de comunicación por segundo

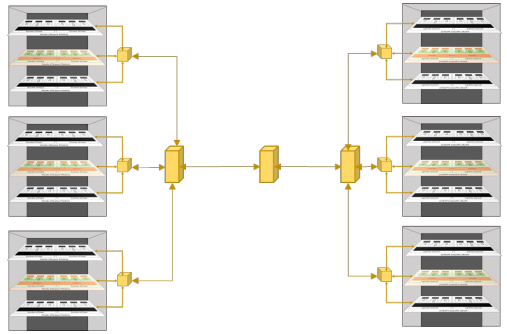
Granularidad

Relación entre comunicación y computación



Fina

Pequeñas subrutinas del algoritmo pueden dividirse en múltiples procesadores

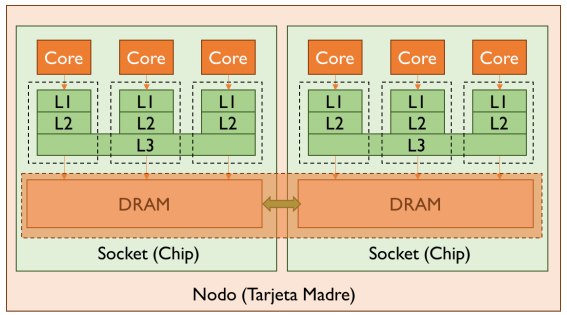


Gruesa

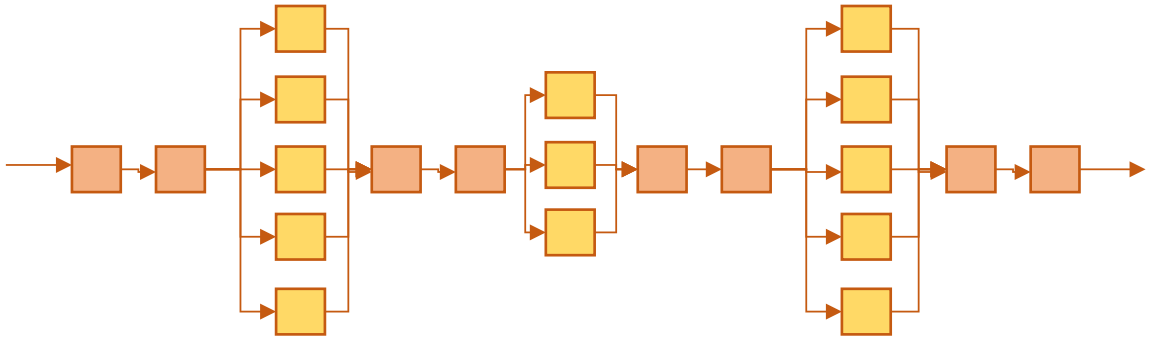
La mayoría de cálculos pueden ser ejecutados en paralelo, con pequeños intercambios de comunicación.

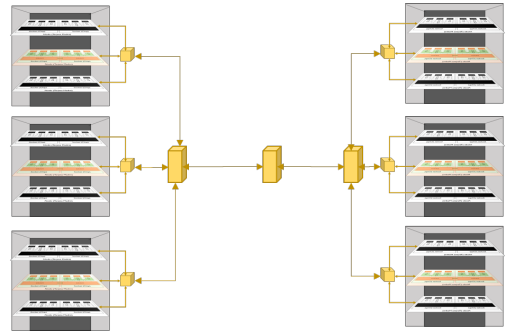
Granularidad

Relación entre comunicación y computación

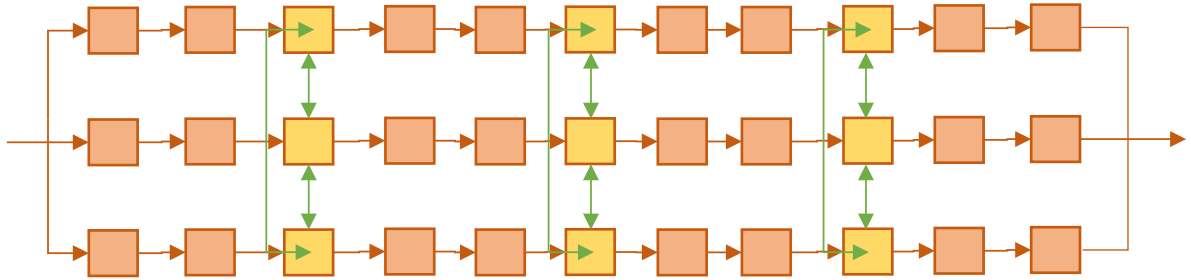


Fina





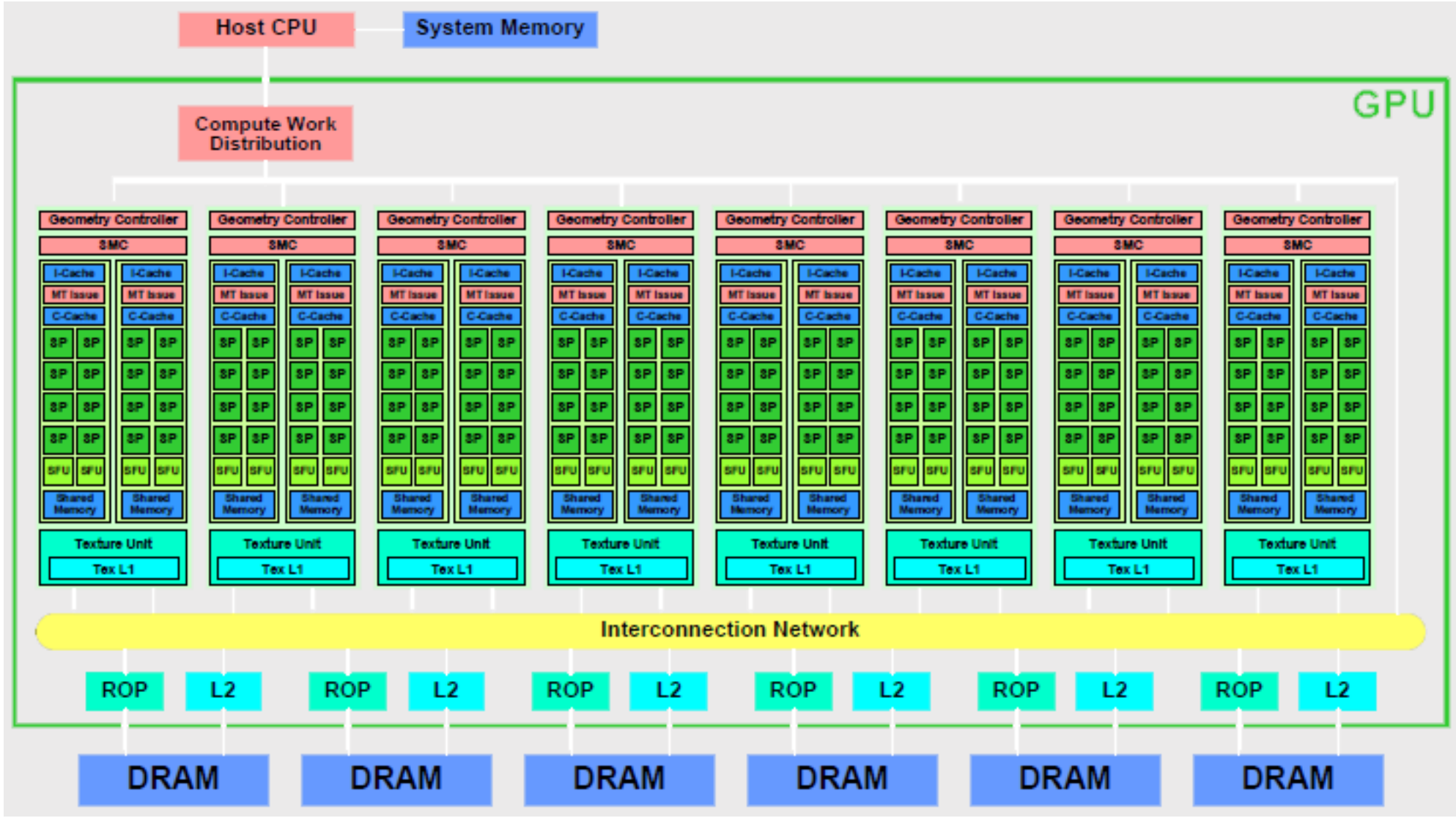
Gruesa



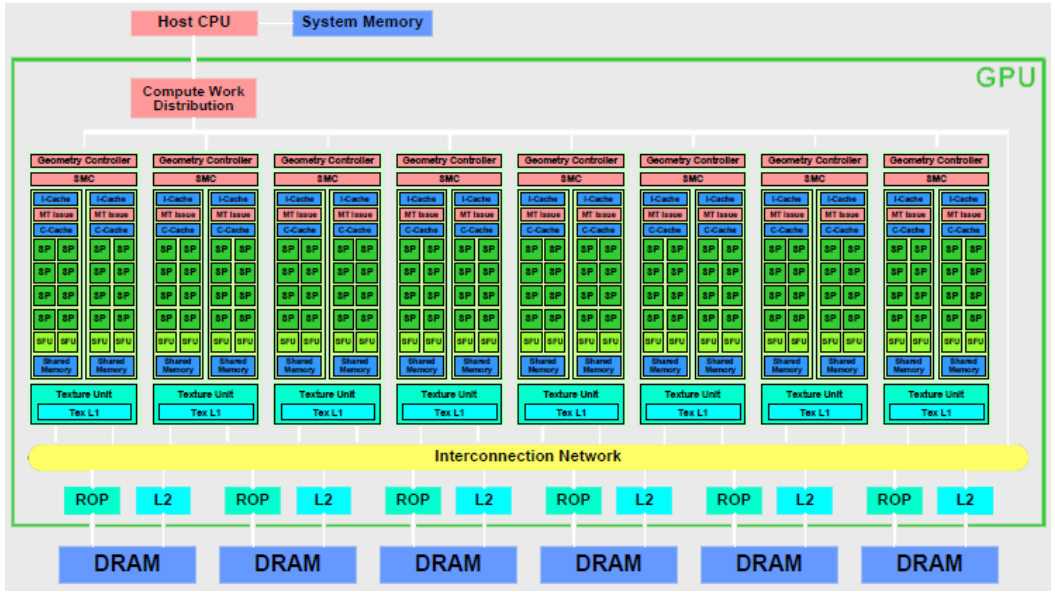
Ideas Importantes

1. Paralelismo depende de: **procesamiento** y **comunicación**
2. A mayor número de procesadores, mayor número de operaciones posibles por segundo, pero mayor costo de comunicación
3. El acceso de memoria define paradigma de programación: Memoria Compartida (multi-core, multi-socket) y Memoria Distribuida (multi-nodo), y esto está altamente relacionado con la **granularidad**.
4. El **porcentaje paralelizable** afecta fuertemente la eficiencia del sistema.

Bono: GPUs y demás aceleradores



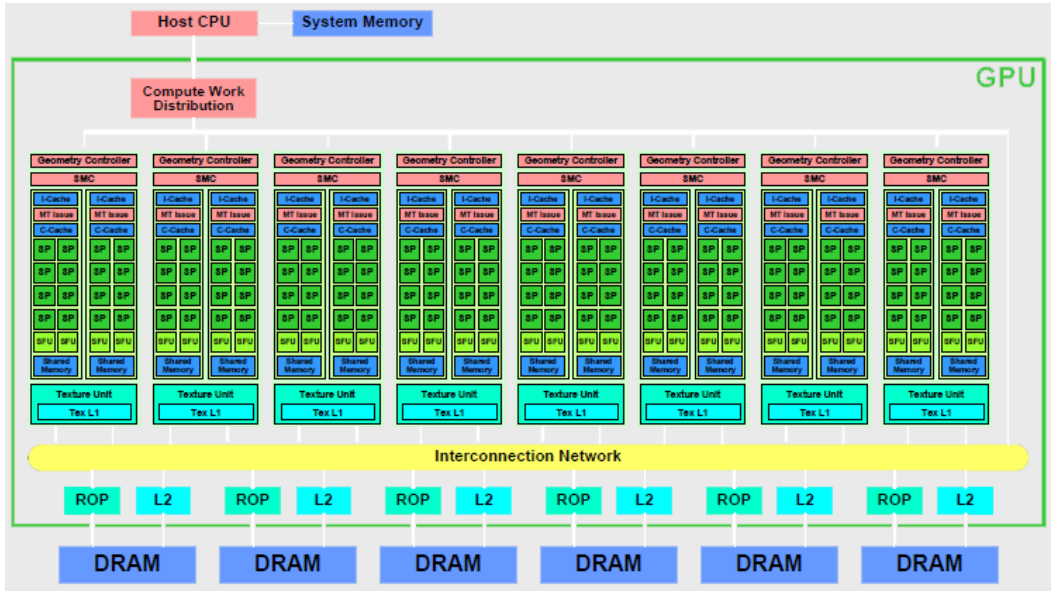
Bono: GPUs y demás aceleradores



- ✓En los últimos años las tarjetas gráficas (GPU) han empezado a ser usadas como instrumentos de cálculo
- ✓ Mientras los CPU tienen decenas de cores, los GPU tienen miles de cores, i.e. miles/millones de operaciones por ciclo.
- ✓Sin embargo, su memoria por core es pequeña, por lo tanto, comunicación host - device es un parámetro relevante

Objetivo: Maximizar número de operaciones aplicadas a un conjunto de datos
=> algoritmos que usan ***batch***

Bono: GPUs y demás aceleradores



- CUDA es el lenguaje para controlar devices (GPUs) desde el host (CPU):
  - ✓Manejo de memoria
  - ✓Asignación de tareas
- Existen soluciones más amigables
  - ✓Software se adapta a la disponibilidad del sistema. Ej: TensorFlow
  - ✓OpenAcc es una extensión de OpenMP que incluye directivas para asignar tareas a distintos GPUs y ”escode” los tránsitos de memoria.