

Arquitectura de Computadoras y Sistemas Operativos

Cecilia Jarne
cecilia.jarne@unq.edu.ar
Twitter: [@ceciliajarne](https://twitter.com/ceciliajarne)

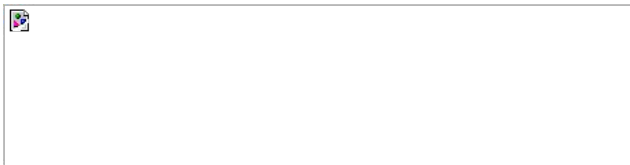
Arquitectura de Computadoras y Sistemas Operativos

TEMA 1. Arquitectura de Computadoras y Sistemas Operativos

Arquitectura Von Neuman.
Extensiones. Taxonomía de Flynn.
Arquitectura de un computador actual.
Arquitecturas híbridas.
Supercomputadoras. Computación científica y computación eficiente.

Objetivos

- Nos interesa tener conciencia de cómo se almacenan los datos, cómo se realizan las operaciones, cómo el sistema operativo administra el acceso a los recursos, de cómo se organiza la memoria, etc.
- Veamos un ejemplo:



<https://docs.python.org/3/tutorial/float.html>

0.1

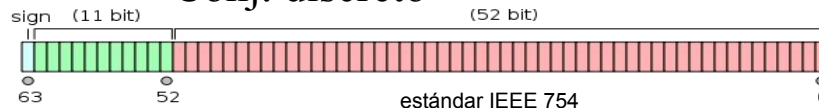


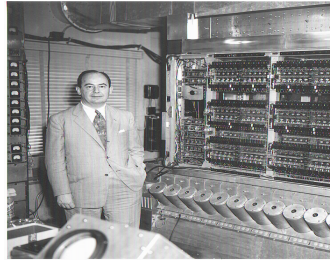
1/10



$3602879701896397 / 2^{55}$

- Representación binaria
- Representación de punto flotante
- Conj. discreto



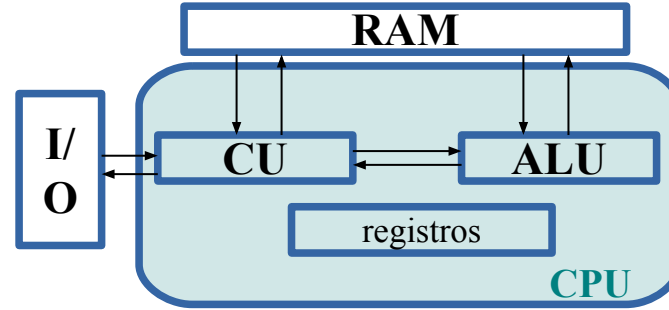


Von Neumann frente a la computadora IAS, 1952

<https://www.ias.edu/people/vonneumann/ecp>

Modelo de Von Neumann:

- Una **unidad de procesamiento** que contiene una unidad aritmético lógica y registros del procesador,
- Una **unidad de control** que contiene un registro de instrucciones y un contador de programa,
- Una **memoria** para almacenar tanto datos como instrucciones,
- Almacenamiento masivo externo,
- Mecanismos de **entrada y salida**

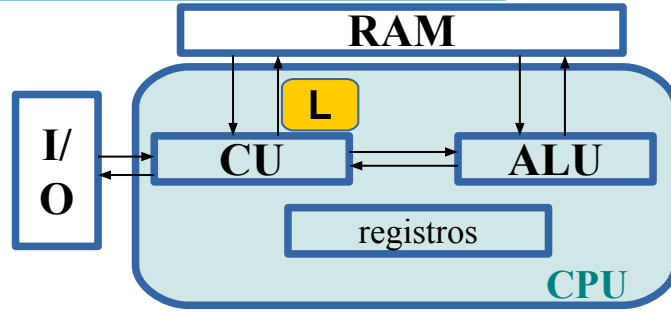
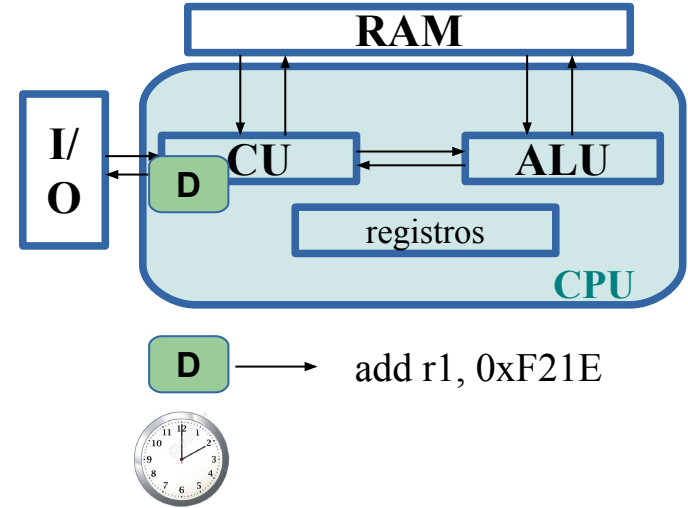
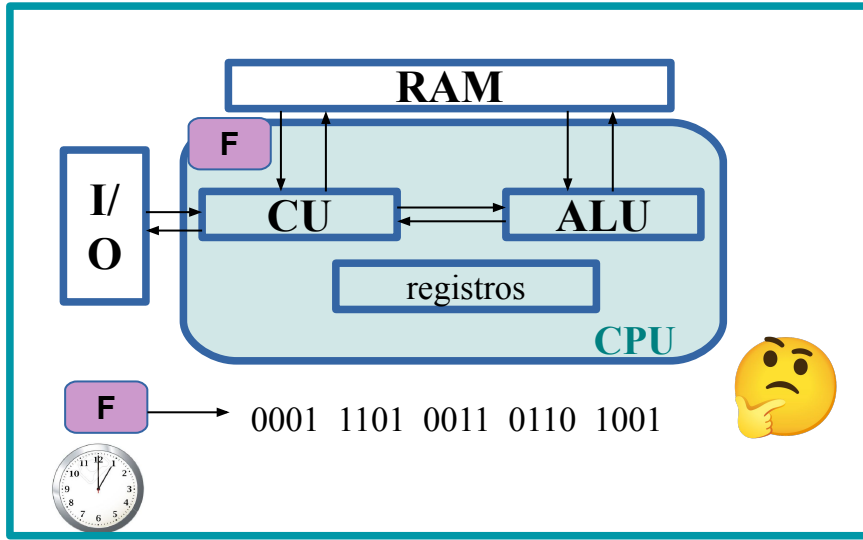


- Cada instrucción se resuelve en 5 pasos:
 - Fetch
 - Decode
 - Load
 - Execute
 - Store
- Cada paso se ejecuta en un ciclo de reloj

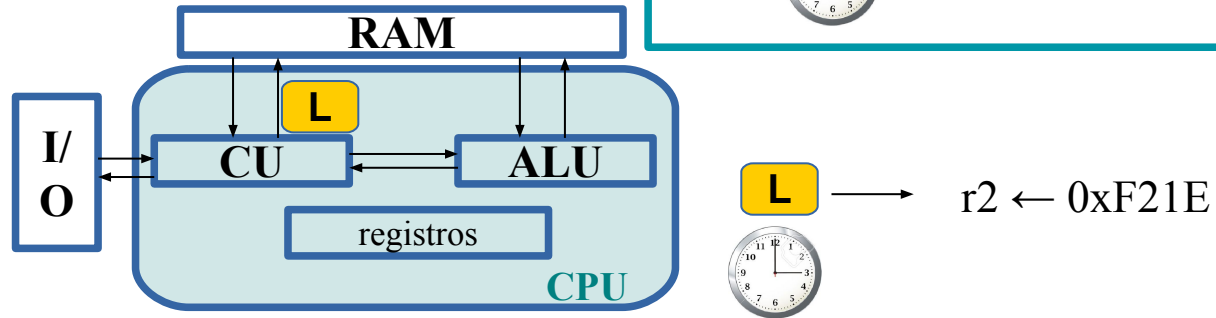
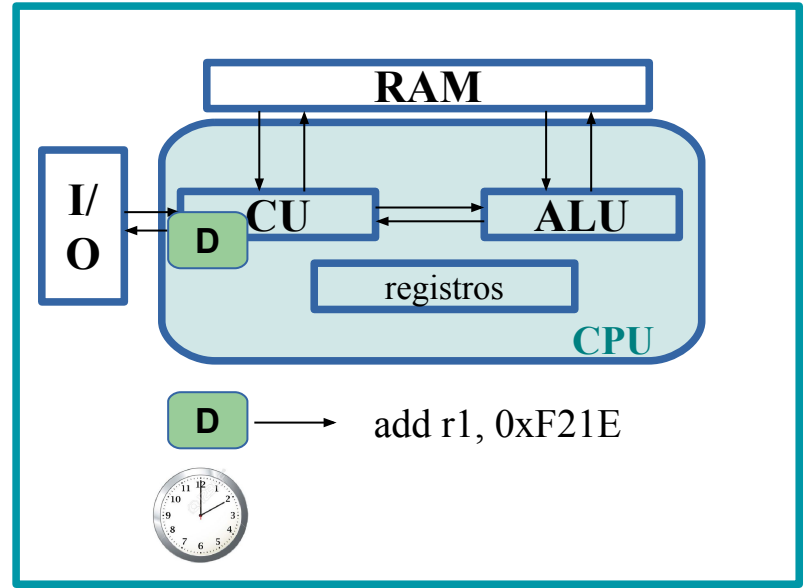
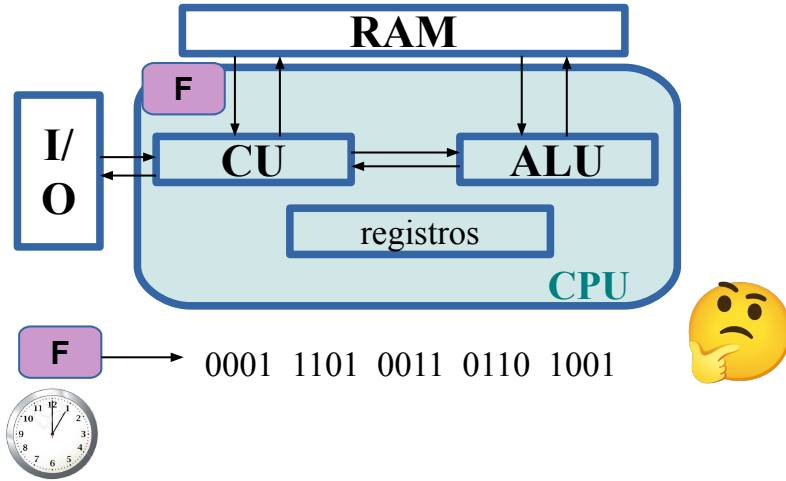


<https://goo.gl/m2cS1H>

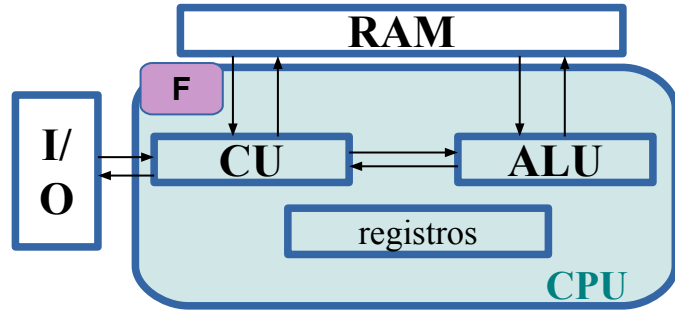
- Ejecutamos una instrucción:



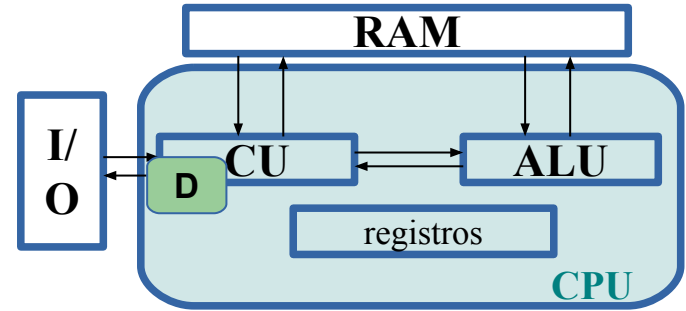

- Ejecutamos una instrucción:



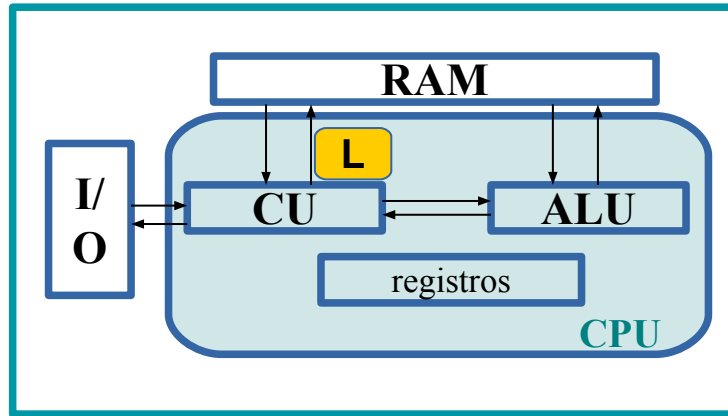

- Ejecutamos una instrucción:




F → 0001 1101 0011 0110 1001

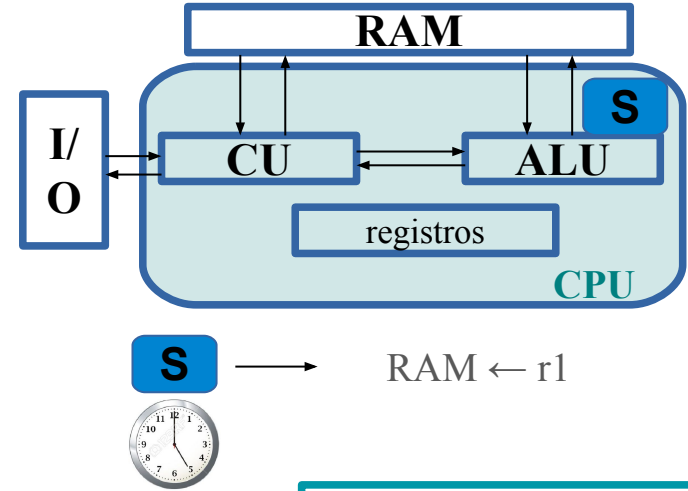
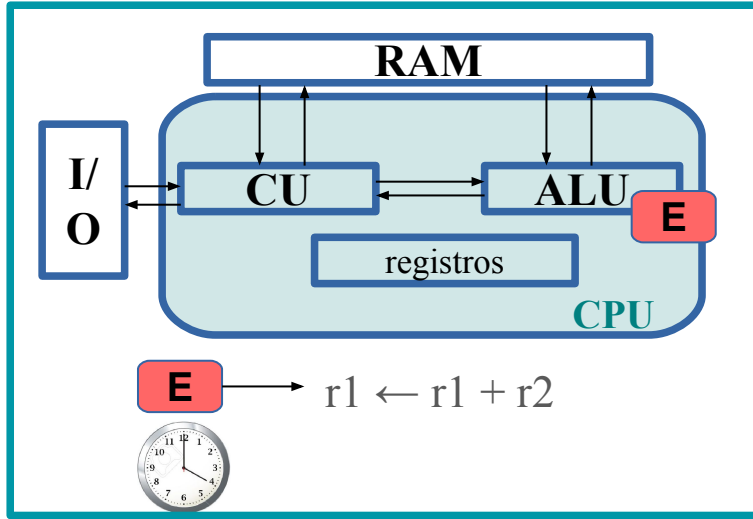


D → add r1, 0xF21E



L → r2 ← 0xF21E

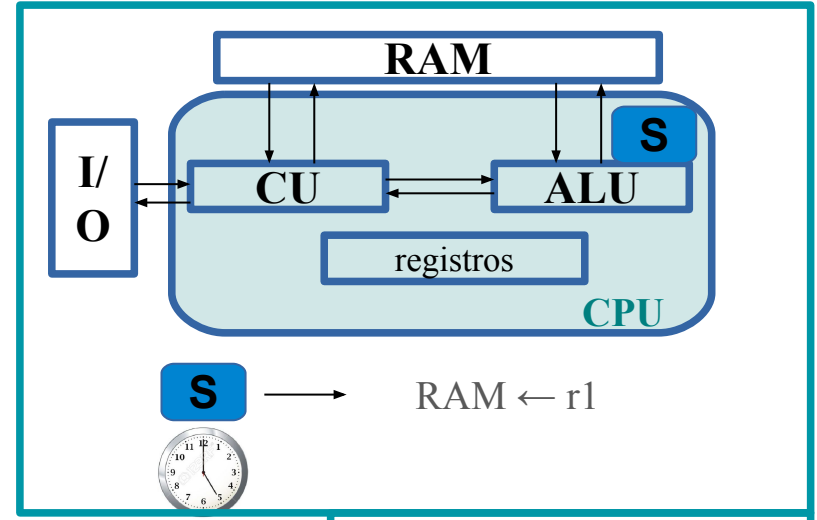
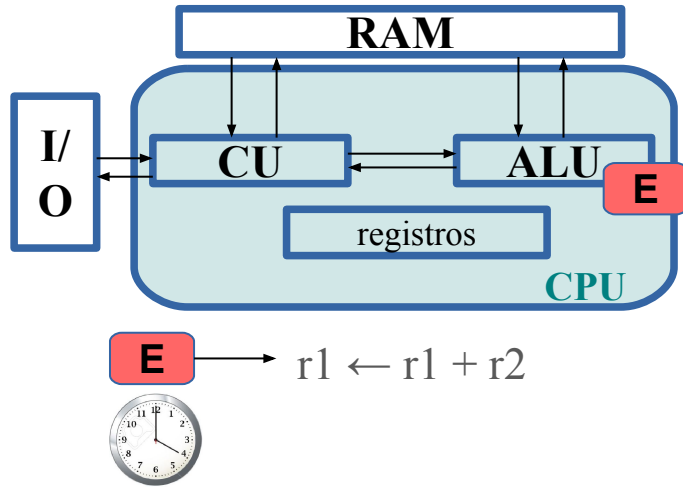




- Una instrucción secuencial involucra 5 ciclos de reloj



Este modelo es la base sobre la que se diseñan **todas(*) las computadoras** en la actualidad (incluyendo las super computadoras!). A este modelo se le fueron introduciendo diferentes **extensiones**.

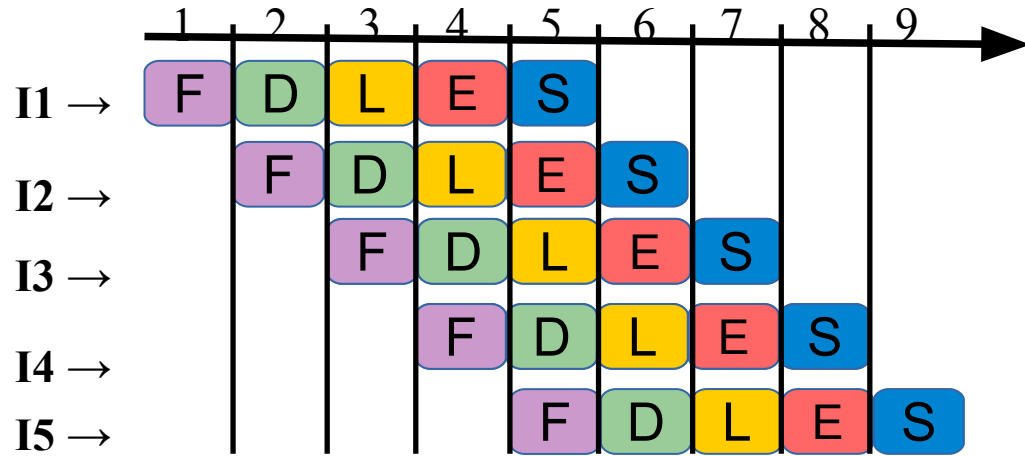


Este modelo es la base sobre la que se diseñan **todas(*) las computadoras** en la actualidad (incluyendo las super computadoras!). A este modelo se le fueron introduciendo diferentes **extensiones**.

- Una instrucción secuencial involucra 5 ciclos de reloj



- Extensiones del modelo Von Neumann



Pero cada porción de instrucción (FDLES) en cada instante de tiempo (ciclo de reloj) se realiza en una determinada parte del computador de manera independiente.

Paralelismo a nivel de instrucción.

Se pueden realizar porciones diferentes de dos o más instrucciones en un mismo instante de tiempo.

PIPELINING

Ahora en **9 ciclos** puedo terminar **5 instrucciones !**



Qué tan bueno es en realidad?

Veamos un ejemplo:

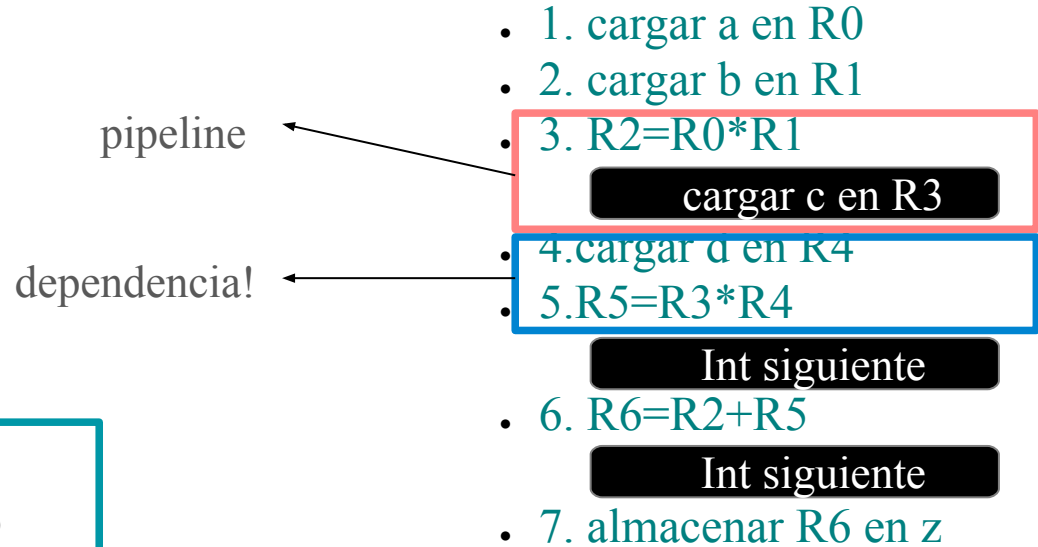
$$z = a * b + c * d$$

$$z1 = a * b$$

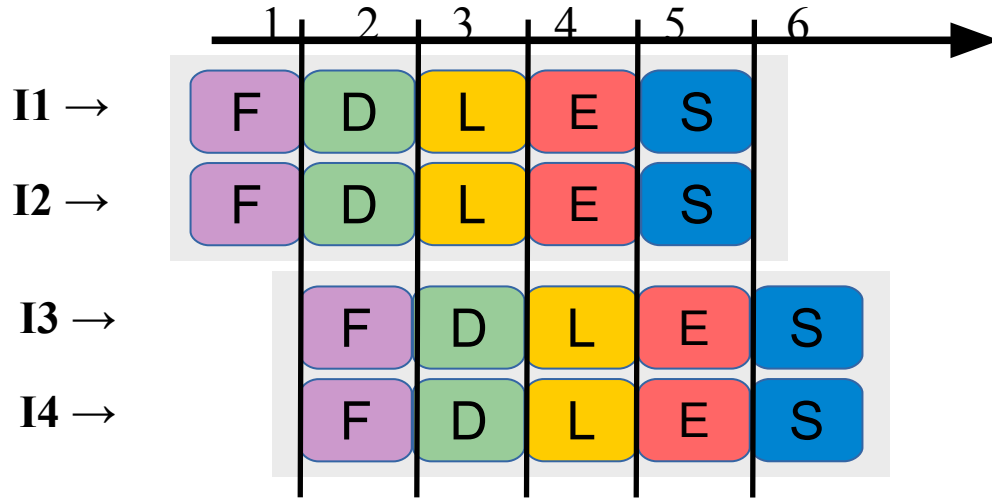
$$z2 = c * d$$

$$z = z1 + z2$$

- 3 operaciones
- De todos los pasos solo aproveché una sola vez la arquitectura
- Las dependencias son un problema



- Extensiones del modelo Von Neumann



Pipelining + superscaling

La arquitectura se modifica para permitir la ejecución de más de una instrucción al mismo tiempo.

Paralelismo a nivel de instrucción.

Sigue siendo procesamiento escalar.

SUPERSCALING

Qué tan bueno es en realidad?

Veamos un ejemplo:

$z = a * b + c * d$

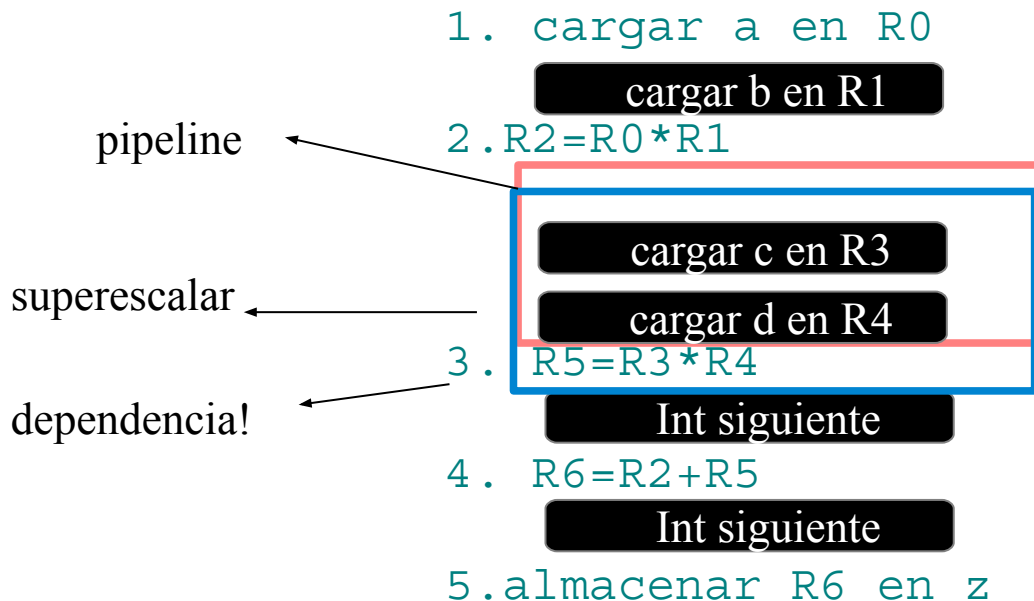
$z1 = a * b$

$z2 = c * d$

$z = z1 + z2$

- 3 operaciones
- De todos los pasos solo aproveché una sola vez la arquitectura
- Las dependencias son un problema

- Cómo podemos “atacar” el problema de las dependencias desde el punto de vista de la programación.



$x = y * z;$

$q = r + x * 2;$

$x = a + b;$

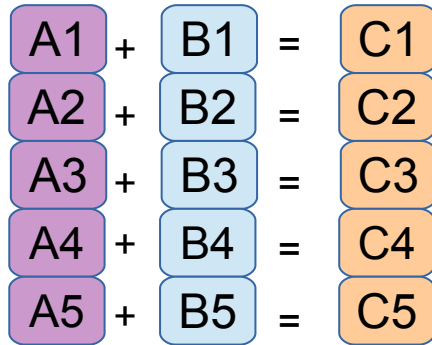
$x0 = y * z;$

$q = r + x0 * 2;$

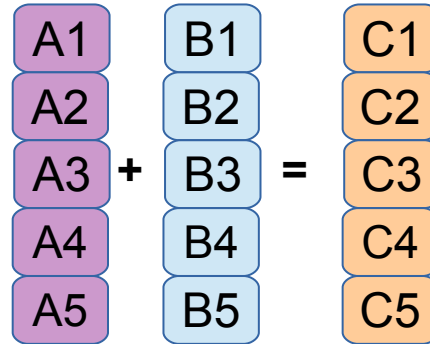
$x = a + b;$

- Extensiones del modelo Von Neumann

operaciones escalares



operaciones vectoriales



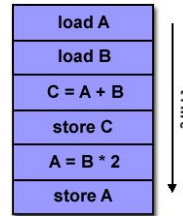
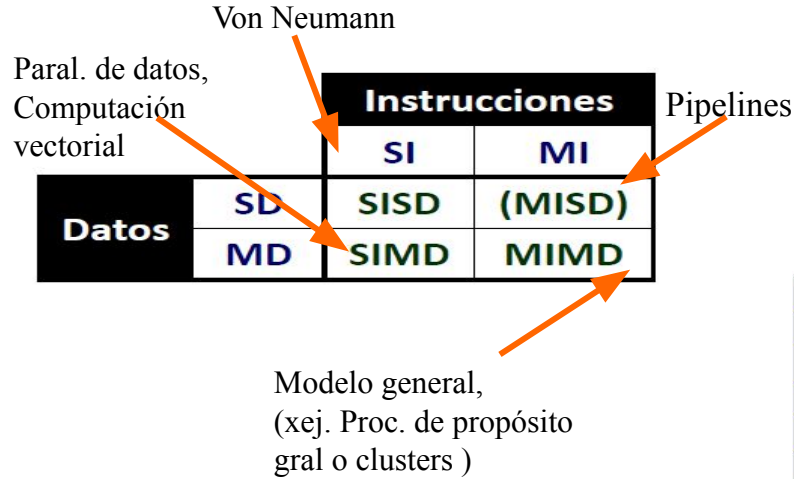
Capaz de ejecutar operaciones matemáticas sobre múltiples datos de forma simultánea. (registros vectoriales)

En general es adicional al pipelining superescalar.

Instrucciones vectoriales especiales (SSE,AVX,etc)

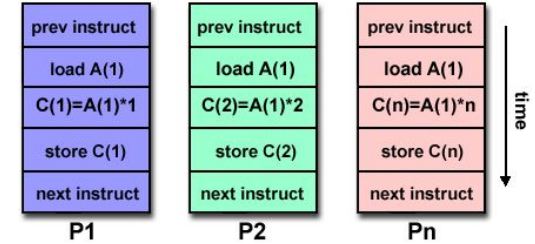
**OPERACIONES
VECTORIALES**

Clasificación de Arquitecturas

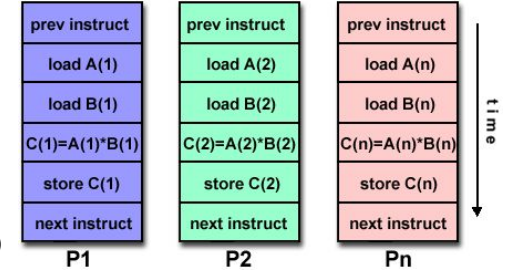


SISD

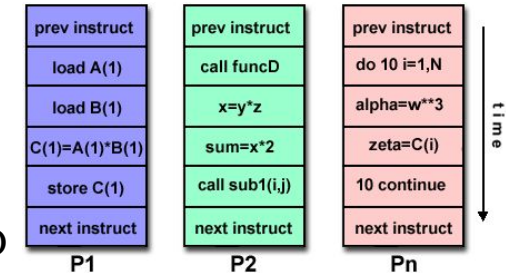
MISD



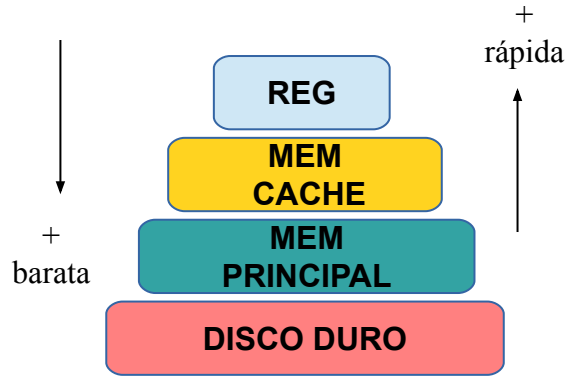
SIMD



MIMD



• Jerarquía de memoria



- **Registros:** cableados en el procesador
- **Cache:** memoria rápida, cercana al procesador y cara
- **Memoria ppl:** RAM, lenta y barata
- **Disco duro:** lentísimo y baratísimo

1 CPU cycle

0.3 ns

1 s

Level 1 cache access

0.9 ns

3 s

Level 2 cache access

2.8 ns

9 s

Level 3 cache access

12.9 ns

43 s

Main memory access

•120 ns

•6 min

Solid-state disk I/O

•50-150 μ s

•2-6 days

Rotational disk I/O

•1-10 ms

•1-12 months

Internet: SF to NYC

•40 ms

•4 years

Internet: SF to UK

•81 ms

•8 years

Internet: SF to Australia

•183 ms

•19 years

OS virtualization reboot

•4 s

•423 years

SCSI command time-out

•30 s

•3000 years

Hardware virtualization
reboot

•40 s

•4000 years

Physical system reboot

•5 m

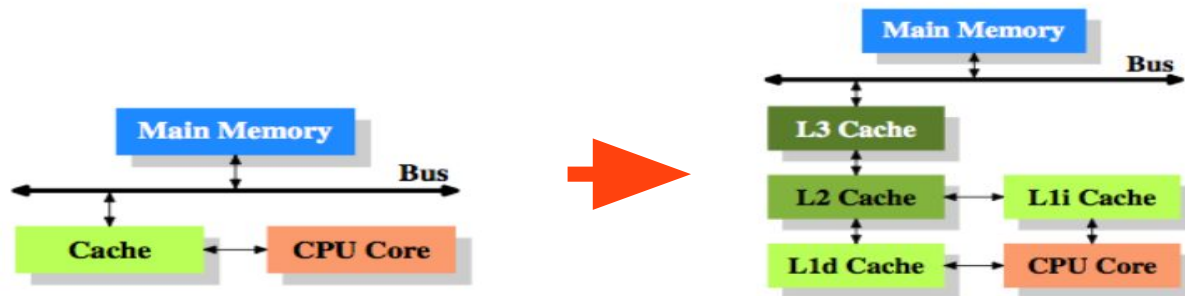
•32 millenia

<http://sgros.blogspot.com.ar/2014/08/memory-access-latencies.html>

• Memoria caché

- > costo
- > velocidad
- < capacidad

Los datos se transfieren a cache en bloques de un determinado tamaño → **cache lines**



```
maria@maria-UX21E: ~  
maria@maria-UX21E:~$ lscpu  
Arquitectura:          x86_64  
CPU op-mode(s):        32-bit, 64-bit  
Orden de bytes:         Little Endian  
CPU(s):                 4  
On-line CPU(s) list:    0-3  
Hilo(s) por núcleo:     2  
Núcleo(s) por zócalo: 2  
Socket(s):              1  
Nodo(s) NUMA:           1  
ID del vendedor:        GenuineIntel  
Familia de CPU:         6  
Modelo:                 42  
Stepping:               7  
CPU MHz:                 800.000  
BogoMIPS:                3190.05  
Virtualización:         VT-x  
caché L1d:              32K  
caché L1i:              32K  
caché L2:                256K  
caché L3:                3072K  
NUMA node0 CPU(s):      0-3
```

Diseñada de acuerdo al principio de localidad

espacial y temporal

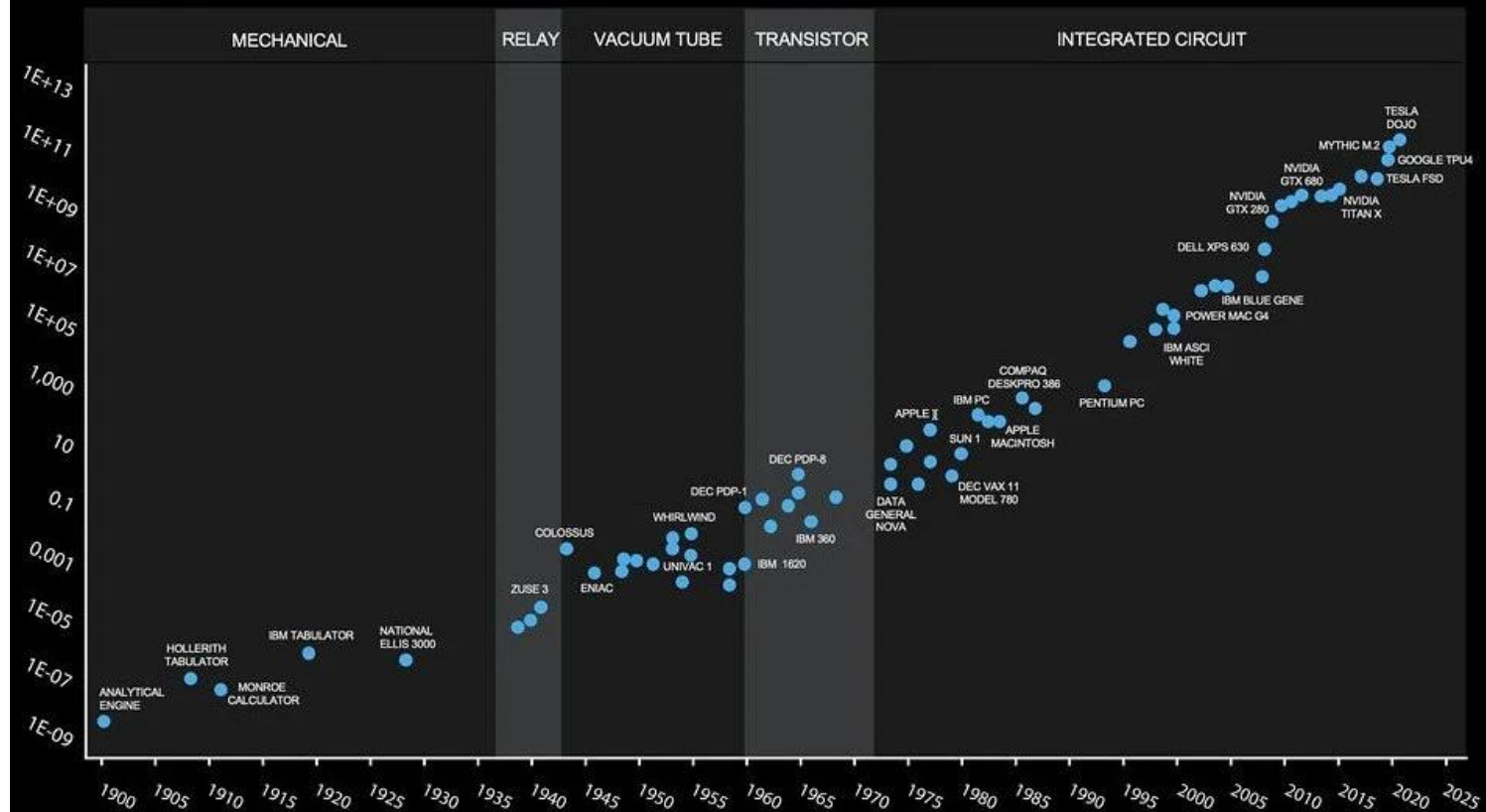
Por ejemplo, recorrer
un vector con un loop

Por ejemplo, un llamado a
una función dentro de un
loop

Cada vez que se realiza una operación LOAD/STRORE
puede ocurrir:

cache miss o cache hit

122 YEARS OF MOORE'S LAW

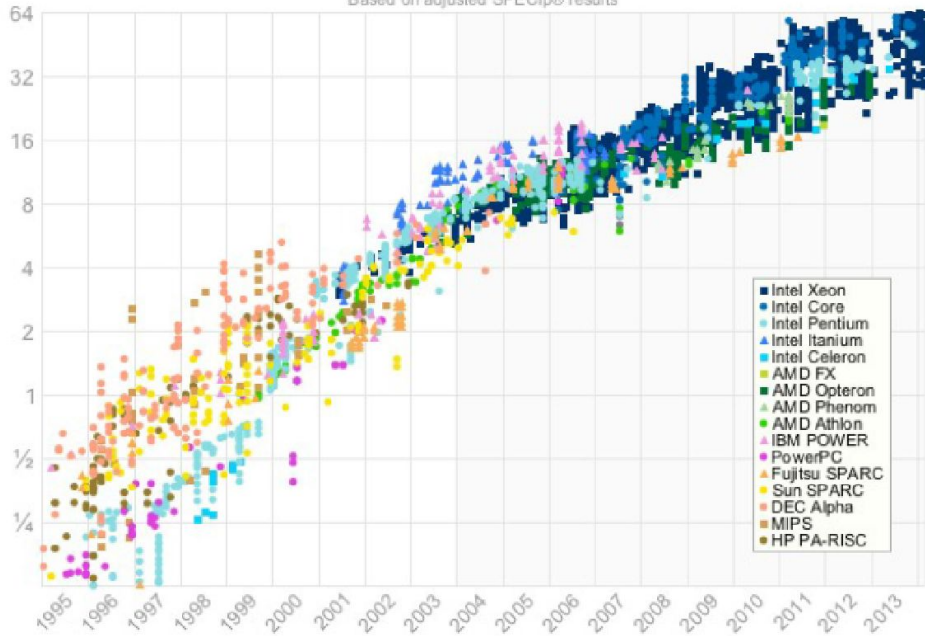


Ley de Moore:

Expresa que aproximadamente cada año se duplica el número de transistores en **un microprocesador**.

Single-Threaded Floating-Point Performance

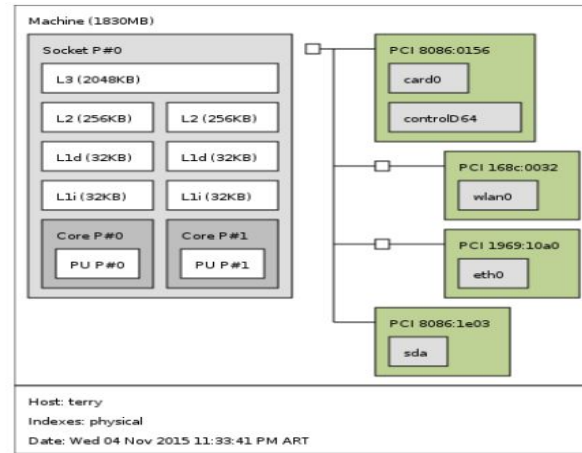
Based on adjusted SPECfp® results



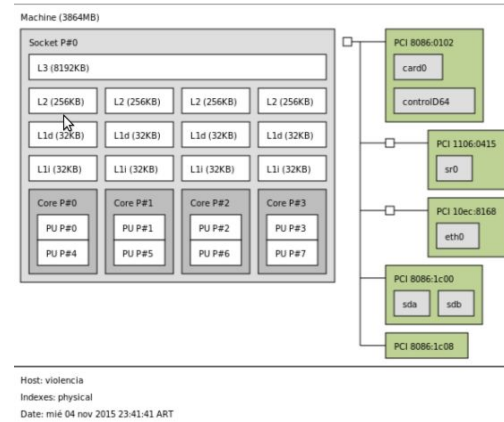
Ley de Moore:

Expresa que aproximadamente cada año se duplica el número de transistores en **un microprocesador**.

Básicamente Von
Neumann

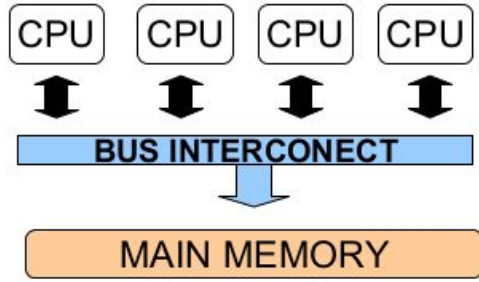


Intel(R) Celeron(R) CPU 1007U



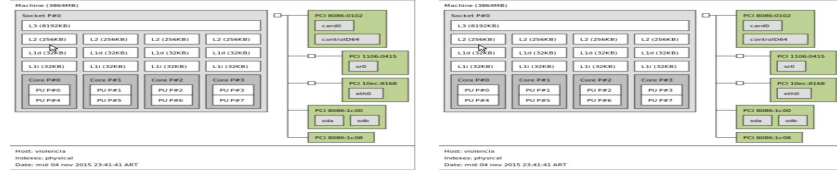
Intel(R) Core(TM) i7-2600 CPU

Symmetric Multiprocessors



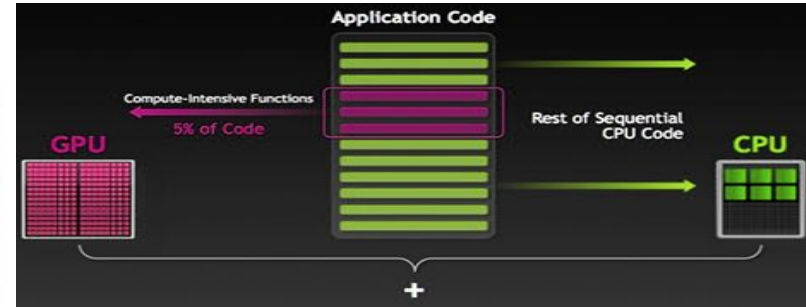
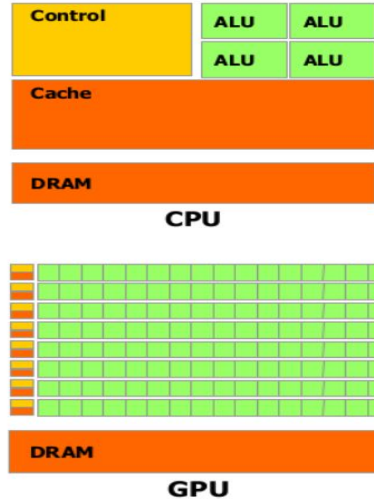
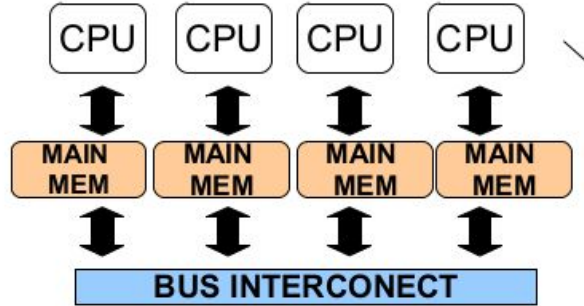
• Pero en realidad ...

Symmetric Multiprocessors - NonUniform Memory Access



Unidad de procesamiento gráficos (GPUs)

NonUniform Memory Access



Programación en Entornos Paralelos

Cómo impacta en el diseño de software?

- Paralelismo masivo
- Complejidad creciente
- Menos eficacia para software viejo
- Poca previsibilidad

Hay que pensar en el hardware al momento de codificar!

Podemos pensar en una serie de pasos:

- Usar aproximaciones cuando sea posible
- Desarrollar algoritmos más eficientes
- Utilizar estructuras de datos apropiadas
- Obtener hardware más veloz
- Usar/escribir software optimizado para el hardware que tenemos disponible. Aprovechar bibliotecas ya

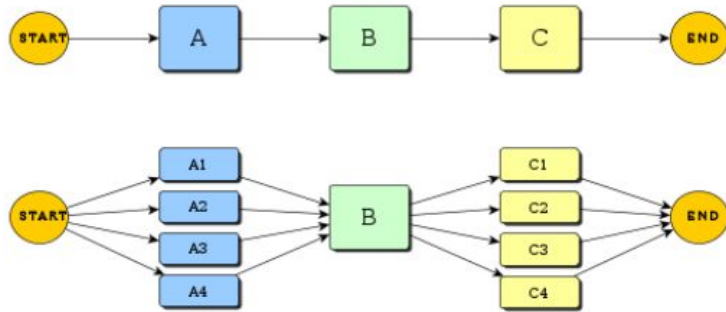
optimizadas como BLAS, LAPACK

- Paralelizar

Programación en Paralelo

Cecilia Jarne
cecilia.jarne@unq.edu.ar
Twitter: [@ceciliajarne](#)

Programación en Entornos Paralelos



Programacion paralela



Programación en Entornos Paralelos

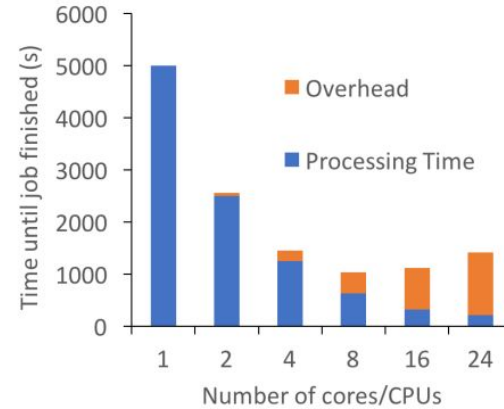
Cómo se debe pensar la programación en HPC?

- Programación optimizada
- Tener en cuenta el hardware subyacente (x ejemplo jerarquía de memoria)

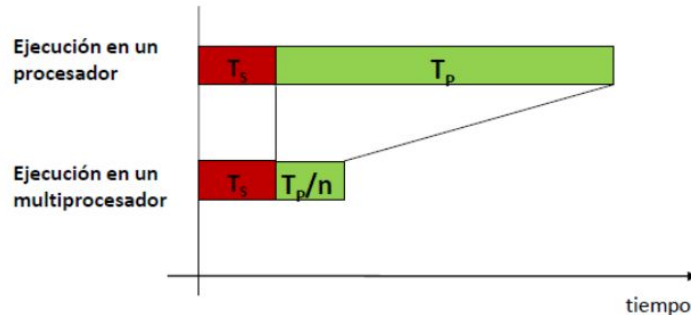
Computación de alta performance

- Tener en cuenta el tiempo invertido en la programación
- Más difícil el debugging y el profiling
- Portabilidad
- Ejecución no determinista
- No es posible evaluar la eficiencia algorítmica
- Qué lenguajes de programación son recomendables? Porque?
- La programación paralela no siempre es la solución y no es fácil obtener resultados óptimos

- T_{proc} : depende la complejidad y dimensión del problema + características de las unidades de procesamiento (hw, heterogeneidad, no dedicación, etc).
- T_{com} : Depende de la localidad de procesos y datos (comunicación inter e intraprosesor, canal de comunicación)
- T_{idle} (ocioso): x el no determinismo de la ejecución. Es necesario hacer balance de carga (mejorar algoritmos)



$$T = T_{proc} + T_{com} + T_{idle}$$



Ley de Amdahl (1967): La parte serial de un programa determina una cota inferior para el tiempo de ejecución, aún cuando se utilicen al máximo técnicas de paralelismo.

>num de proc → problema >

Programación en Entornos Paralelos

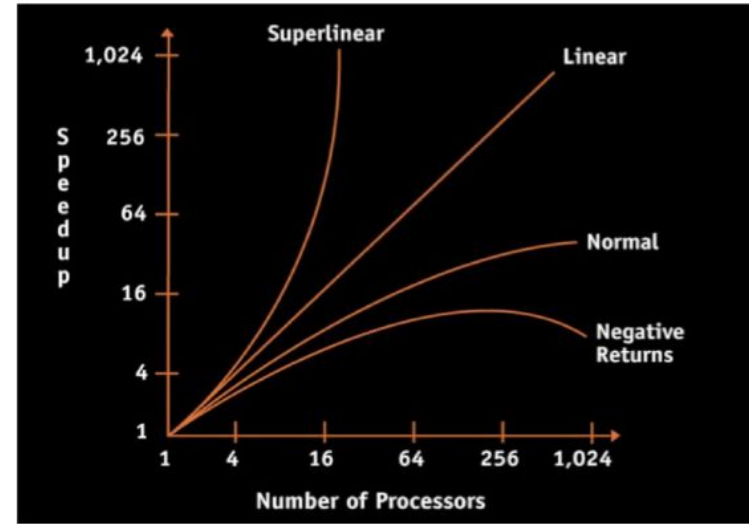
Cómo sabemos que tan bueno (rápido) es nuestro programa paralelo?

- Speedup: $Sp = T1/Tp$ ($T1=1\text{proc}$; $Tp=p\text{proc}$)
- $Sp=p \rightarrow$ speedup lineal (Si uso p procesadores tengo una mejora de factor p)
- Sp compara un prog consigo mismo para diferente número de procesadores.
- La parte serial del programa es un límite para el speedup (errores de paralelización, mal balanceo, costo de comunicación, costo de conexión, tiempo de sincronización)
- Sea S = sección en serie, $1-P=S$ sección en paralelo. Cuál es el máximo speedup para N procesadores?

$$\lim \rightarrow \text{speedup} = \frac{1}{(1-P) + \frac{P}{N}}$$

$$\text{Eficiencia} = Sp/p = T1/(p \cdot Tp)$$

Performance es un compromiso: El grado de paralelismo obtenido y el overhead. Para ello se usan técnicas de scheduling y de balance de carga



Lo que queremos: linear speedup, =100% de eficiencia.

Lo que obtenemos: sublinear speedup, <100% de eficiencia.

A veces, solo a veces: superlinear speedup >100% de eficiencia.

Programación en Entornos Paralelos

Cómo empezamos a programar en paralelo?

- Lograr un compromiso entre:
- Grado de paralelismo
- Overhead (sincro y comunicación)
- Cómo? Schedulling + balance de carga
- Cómo diseñamos/implementamos un algoritmo paralelo?
- Modelos de programación: memoria compartida o memoria distribuida
- Cómo dividir el problema?
- Particionar datos o funcionalidades?
- Qué lenguajes de programación usamos?

Programación en Entornos Paralelos

MPI (Message Passing Interface)

Ventajas:

- Excelente para paralelización en clústeres o sistemas con memoria distribuida.
- Permite comunicación eficiente entre múltiples nodos.
- Ofrece gran control sobre la distribución de trabajo y la comunicación entre procesos.

Desventajas:

- Es más complejo de programar que OpenMP (requiere manejar la comunicación explícita entre procesos).
- Escalabilidad limitada por la latencia en la comunicación entre nodos.
- Requiere un diseño cuidadoso para evitar deadlocks o problemas de sincronización.

MPI (Message Passing Interface)

- **Descarga:** Una de las implementaciones más utilizadas de MPI es Open MPI.

Link Open MPI: <https://www.open-mpi.org/>

- **Lenguajes:** C, C++, Fortran, Python (a través de bindings como `mpi4py`)

<https://mpi4py.readthedocs.io/en/stable/>



Programación en Entornos Paralelos

```
#include <stdio.h>

int main() {
    printf("Hola Mundo\n");
    return 0;
}
```

C básico: Solo imprime un mensaje, y no utiliza paralelismo.

MPI en C: Ejecuta el código en varios procesos paralelos, cada uno con su propio rango, imprimiendo desde cada uno.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv); // Inicializa el entorno MPI

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size); // Obtiene el
    número total de procesos

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Obtiene el
    rango de cada proceso

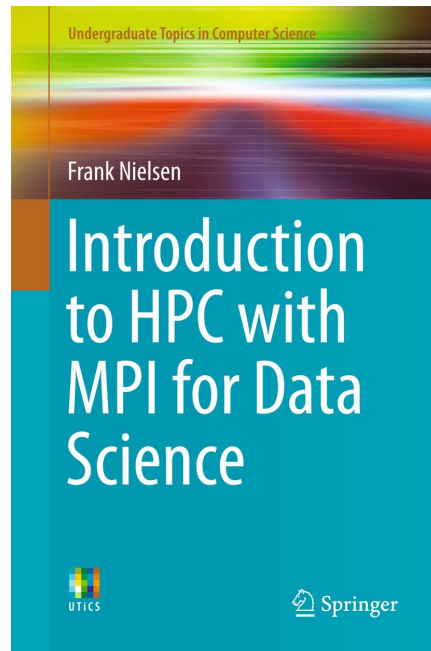
    printf("Hola Mundo desde el procesador %d de %d\n", world_rank,
    world_size);

    MPI_Finalize(); // Finaliza el entorno MPI
    return 0;
}
```

Programación en Entornos Paralelos

Para compilar y ejecutar este programa en C con MPI:

```
mpicc -o hola_mundo_mpi hola_mundo_mpi.c  
mpirun -np 4 ./hola_mundo_mpi
```



<https://www.skillsoft.com/book/introduction-to-hpc-with-mpi-for-data-science-4a2797c0-f418-11e6-bb2f-0242c0a80b05>

Programación en Entornos Paralelos

```
print("Hola Mundo")
```

```
from mpi4py import MPI
```

```
comm = MPI.COMM_WORLD # Inicializa el comunicador global  
size = comm.Get_size() # Obtiene el número total de procesos  
rank = comm.Get_rank() # Obtiene el rango de cada proceso
```

```
print(f"Hola Mundo desde el procesador {rank} de {size}")
```

Python básico: Similar al C básico, solo imprime el mensaje una vez.

MPI en Python: Al igual que en C con MPI, ejecuta el código en varios procesos, mostrando un mensaje desde cada proceso.

```
mpirun -np 4 python hola_mundo_mpi.py
```

Programación en Entornos Paralelos

CUDA (Compute Unified Device Architecture)

Ventajas:

- Aprovecha la gran capacidad de cómputo de las GPUs (tarjetas gráficas) para tareas altamente paralelizables.
- Ideal para aplicaciones científicas y de inteligencia artificial que requieran procesamiento masivo de datos.
- Buen soporte para bibliotecas optimizadas, como cuBLAS o cuDNN.

Desventajas:

- Depende de hardware específico (tarjetas NVIDIA).
- El desarrollo en CUDA puede ser más complicado que en CPU, ya que requiere conocimientos específicos de programación en GPU.
- Transferir datos entre CPU y GPU puede ser costoso en términos de tiempo.

CUDA (Compute Unified Device Architecture)

- **Descarga:** CUDA Toolkit se puede descargar desde el sitio oficial de NVIDIA.

Link CUDA Toolkit: <https://developer.nvidia.com/cuda-toolkit>

- **Lenguajes:** C, C++, Fortran, Python (a través de librerías como PyCUDA)



Programación en Entornos Paralelos

OpenMP (Open Multi-Processing)

Ventajas:

- Fácil de usar y de integrar en código C, C++, y Fortran con directivas de compilador.
- Ideal para paralelizar bucles y secciones de código ya existentes con mínimos cambios.
- Buen soporte para programación de memoria compartida (multithreading).
- Disponible en la mayoría de compiladores modernos.

Desventajas:

- Escalabilidad limitada en sistemas con gran cantidad de núcleos (debido a la memoria compartida).
- Falta de portabilidad en arquitecturas de memoria distribuida (ej. clústeres).
- No adecuado para tareas con comunicación intensa entre procesos.

Programación en Entornos Paralelos

OpenMP

- **Descarga:** OpenMP no se descarga como un software independiente, sino que viene integrado en los compiladores compatibles, como GCC, Clang, Intel Compiler, entre otros. Asegúrate de que el compilador que estás usando tenga soporte para OpenMP.
 - **Link GCC (GNU Compiler Collection):** <https://gcc.gnu.org/>
 - **Link Clang:** <https://clang.llvm.org/>
 - **Link Intel Compiler:**
<https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html>
- **Lenguajes:** C, C++, Fortran



Python Multiprocessing

Ventajas:

- Facilita la creación de procesos paralelos en Python, especialmente en sistemas con múltiples núcleos.
- Permite la ejecución en paralelo de diferentes tareas sin las restricciones del Global Interpreter Lock (GIL) de Python.
- Es de fácil uso y tiene una interfaz similar a la programación en serie.

Desventajas:

- La sobrecarga de comunicación entre procesos puede hacer que no sea adecuado para tareas muy ligeras.
- No siempre escala bien en aplicaciones con una gran cantidad de procesos o hilos debido al uso de memoria compartida.
- Los tiempos de inicio y sincronización de procesos pueden afectar el rendimiento en tareas cortas o sencillas.

Python Multiprocessing

- **Descarga:** Python ya incluye el módulo `multiprocessing` en su biblioteca estándar, por lo que no es necesario descargar nada adicional si tienes Python instalado.
 - **Link Python:** <https://www.python.org/downloads/>
- **Lenguajes:** Python

A no confundir con Multi Threading:

MULTI PROCESSING VS MULTI THREADING