

1. Clasificación del paralelismo implícito de una aplicación

El paralelismo implícito se clasifica según 3 aspectos:

-) Nivel de paralelismo en el código
-) Paralelismo de tareas - paralelismo de datos
-) Granularidad

En cada nivel de una aplicación, disponemos de una granularidad distinta:

- Programas: grano grueso
- Funciones: grano medio
- Bucles: grano medio-fino
- Operaciones: grano fino

Dependencias de datos

Sean B_1 y B_2 dos bloques de código. Para que haya dependencia de datos es necesario que:

- i) B_1 esté antes que B_2
- ii) B_1 y B_2 hagan referencia a la misma pos de memoria

Raw

```
...
a=b*c
...
//código que no usa a
d=a+c
...
```

WAW

```
...
a=b*c
...
//se lee a
a=d+e
...
//se lee a
```

WDR

```
...
b=a+1
...
a=d+e
...
//se lee a
```

Tenemos los niveles de paralelismo implícito dentro de una aplicación:

→ De tarea (relacionado con el nivel de función)

→ De datos (relacionado con el nivel de bucle y suelo verse en operaciones vectoriales)

2. Paralelismo explícito

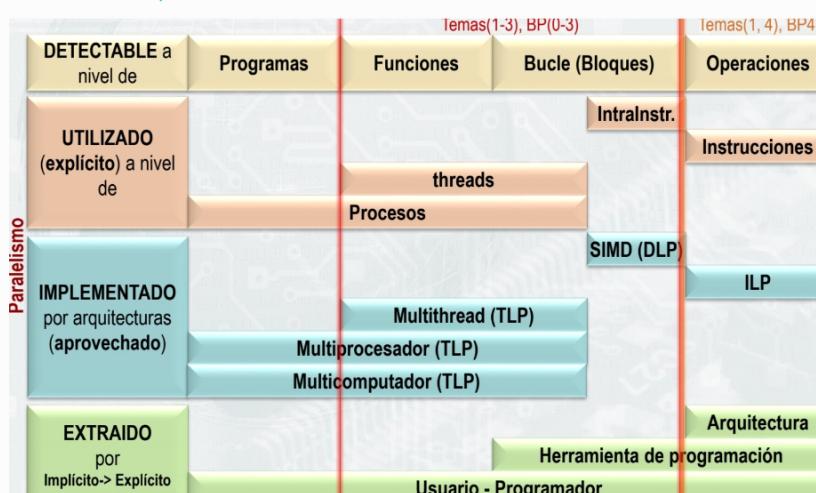
Este paralelismo se clasifica en tres niveles:

-) Instrucciones: La UC gestiona la ejecución de instrucciones de una unidad de procesamiento.
-) Hilo: Menor secuencia de instrucciones que se pueden ejecutar en paralelo o concurrentemente.
-) Proceso: Conjunto de hilos.

Hebras vs Procesos

	Procs	Contra
Hebras	Uso de memoria comp. por las hebras para comunicarse menor tiempo ejecución: menor gasto <ul style="list-style-type: none"> - En creación - En Destrucción - En Comunicación - En comunicación 	Cache hebra bien: <ul style="list-style-type: none"> - Pila - Regs, PC, SP más memoria ocupada
Procesos	Menos memoria o elementos duplicados	Uso de llamadas also para comunicación

Detección, utilización, implementación y extracción



3. Clasificación de arquitecturas paralelas

Computación paralela y computación distribuida

La computación paralela consiste en estructurar aspectos relacionados con desarrollo y ejecución de apps en sistemas de cómputo compuestos por múltiples cores que ejecutan una unidad autónoma.

La computación distribuida consiste en estructurar aspectos relacionados con desarrollo y ejecución de apps en un sistema distribuido, es decir, una colección de recursos autónomos en localizaciones distintas.

Hay dos tipos:

- Baja escala: Consiste en que los recursos son de dominio administrativo situados en distintas localizaciones físicas con conexión a través de infraestructura de red local.

- Grid: Consiste en que los recursos son de múltiples dominios administrativos situados en distintas localizaciones físicas con acceso a través de infraestructura de telecomunicaciones. Dentro de la red se tienen la computación cloud que consiste en utilizar un sistema cloud, es decir, un sistema que ofrece servicios de infraestructura, plataforma y software por los que se paga cuando se usa y a los que se accede mediante una interfaz.

Costos de recursos vitales que:

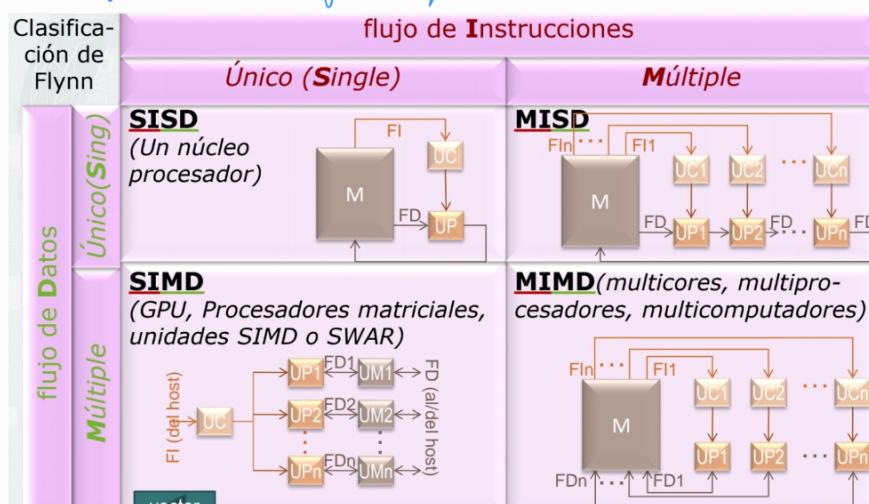
- Son una abstracción de los físicos
- Parecen ilimitados porque hay sistemas de liberación y reserva automáticos,
- Acceso rápido
- Conectados de modo tal que estén independientes de las plataformas de acceso.

Criterios de clasificación

→ Comercial. Se divide según el segmento en la pirámide de compra en externos y empotrados.

→ Educacion, investigación:

- Clasificación de Flynn (Hijos de inst y de datos)



- Seguir el sistema de memoria (FISMIS)

Multiprocesadores	Multicomputadores
<ul style="list-style-type: none"> - Mismo espacio de direcciones - No es necesario conocer la ubicación de los datos <p>Si memoria centralizada: - - - - -</p> <p>en multiprocesadores</p> <ul style="list-style-type: none"> - Mejor latencia - Mejor estabilidad - Comunicación duplicada por variables compartidas - No duplicación de datos 	<ul style="list-style-type: none"> - Cada procesador tiene su espacio de direcciones - Si es necesario conocer la ubicación de los datos <p>- - - - -</p> <ul style="list-style-type: none"> - Menor latencia - Mayor estabilidad - Comunicación explícita por paso de mensajes - Duplicación y copia de datos

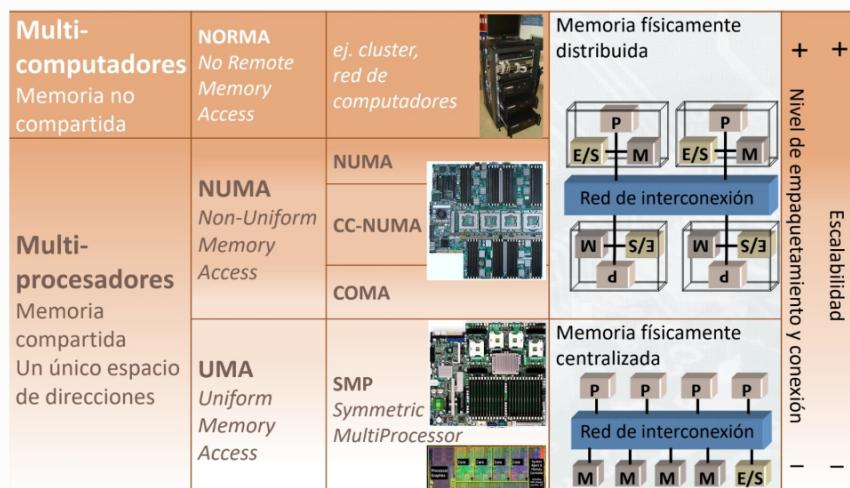
<ul style="list-style-type: none"> - Implementar primitives de sincronización - No es necesario conocer la ubicación de los datos - Programación más sencilla <p><u>En consec.</u></p> <ul style="list-style-type: none"> - Requiere garantizar que el productor haya escrito antes del consumidor 	<ul style="list-style-type: none"> - Sincronización incorporada por software de comunicación - Sí necesita distribución de código y datos entre los procesadores - Programación más difícil. <ul style="list-style-type: none"> - No hay problema por software de comunicación
--	--

Incremento de escalabilidad en multiprocesadores y red de interconexión

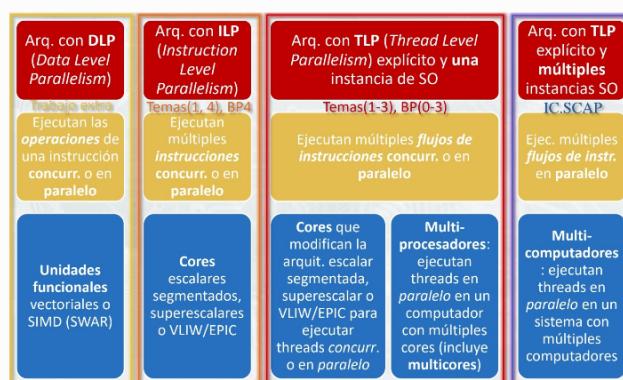
Para conseguir dicho incremento debemos:

- Aumentar la caché del procesador
- Usar redes de menor latencia y mayor ancho de banda
- Distribuir físicamente los módulos de memoria entre los procesadores pese a que se siga compartir el espacio de direcciones

Clasificación completa según el sistema de memoria



Nivel de paralelismo aprovechado.



4. Evaluación de prestaciones de una arquitectura

Medidas usuales para evaluar las prestaciones

Hay dos medidas:

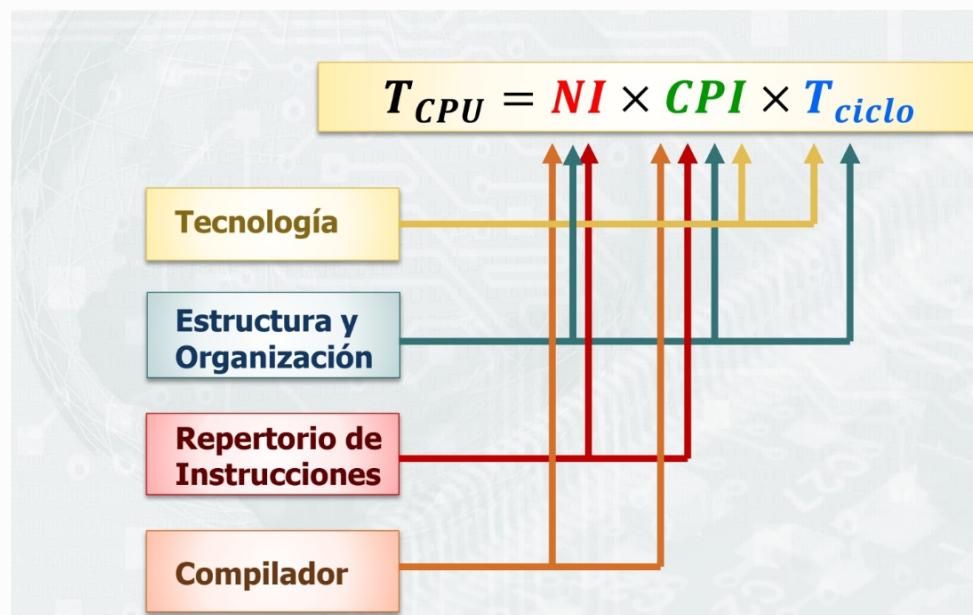
•) Tiempo de respuesta: Tiempo que transcurre desde que se inicia el trabajo hasta que produce una salida. En ocasiones, el tiempo de respuesta lleva consigo un tiempo de espera.

•) Productividad: Se conoce como el nº de tareas resueltas en un tiempo dado. Para ello se usa el tiempo de CPU:

$$T_{CPU} = NI \cdot CPI \cdot T_c$$

$$T_c = \frac{1}{F} \quad \text{Nº ciclos} = NI \cdot CPI = \sum_i NI_i \cdot CPI_i$$

Puede en la siguiente gráfica se ven las dependencias con los distintos componentes y estructuras.



Puede realmente se ve la productividad es en las unidades de instr. por segundo (MIPS). Este dato depende del repertorio de instrucciones (comparación difícil con máquinas de distintos repertorios) y de las prestaciones (variando inversamente, es decir, a mayor MIPS peores prestaciones)

$$\text{MIPS} = \frac{NI}{T_{CPU} \cdot 10^6} = \frac{NI}{NI \cdot CPI \cdot T_c \cdot 10^6} = \frac{F}{CPI \cdot 10^6}$$

Un dato más fiable son las MFLOPS que es lo mismo que los MIPS pero con clítoris punto flotante.

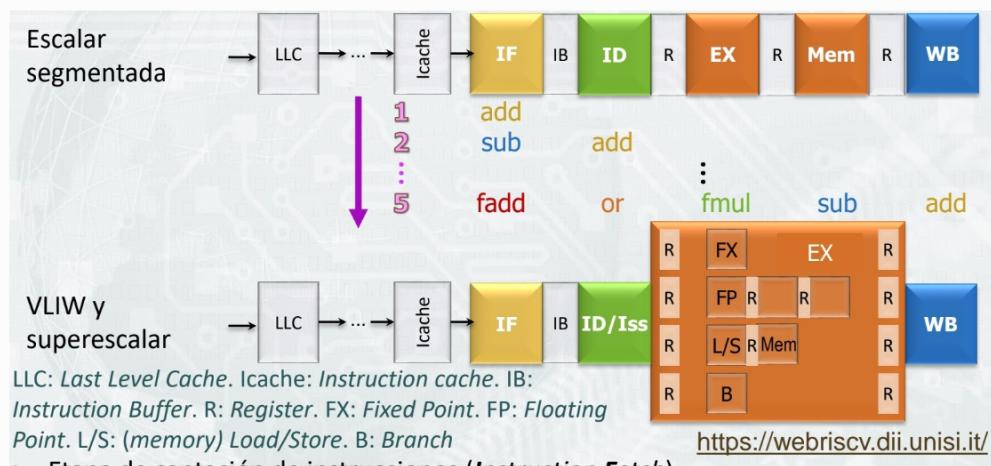
$$MFLOPS = \frac{NFP}{T_{CPU} \cdot 10^6}$$

Generación de prestaciones al hacer una mejoría

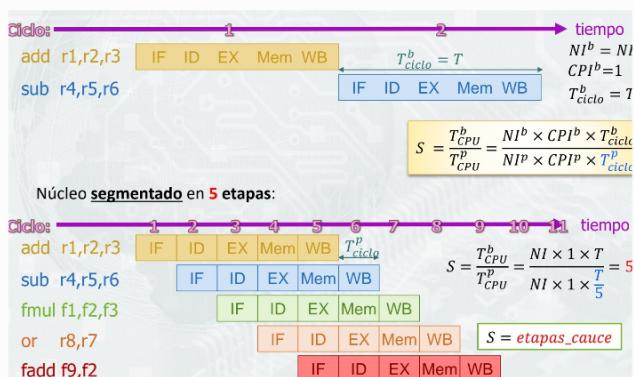
Ser T_b el tiempo base dentro sistema y para mejorar, diremos que la generación de mejoría sistema es:

$$\frac{V_b}{V_p} = S = \frac{T_b}{T_p} = \frac{T_{CPU}^b}{T_{CPU}^p}$$

En arquitecturas con paralelismo a nivel de instrucción tenemos como ilustración la siguiente gráfica:



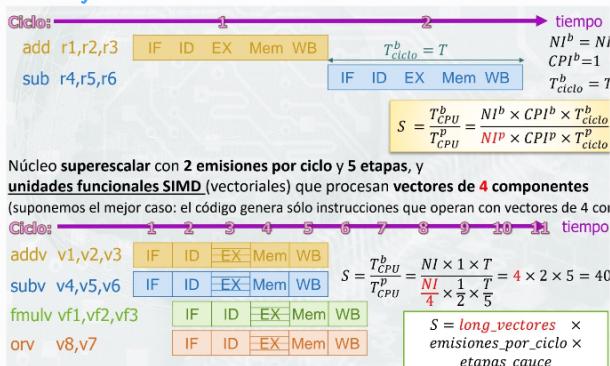
En un núcleo de procesamiento se mejoraría usando segmentación de cauce:



En el mismo caso, también podríamos mejorarlo usando operaciones superescalares:



O con funcionalidades SIMD: SIND:



Sin embargo, hay una serie de factores que provocan que no se puede llegar a la ganancia en velocidad pico: los riesgos ya conocidos de datos, control y estructurales vistos en EC (anterior apartado).

Ley de Amdahl

Claramente por S este mejoramiento ocasionado por un factor p que está limitado por

$$S = \frac{T_b}{T_p} \leq \frac{1}{f + \frac{1-f}{p}} = \frac{p}{1+f(p-1)}$$

lim $S = \frac{1}{f}$
lim $S = p$
 $f \rightarrow 0$

donde f es la fracción del tiempo de ejecución del sistema base durante el que se vea dicha mejora.

Esta ley se puede deducir de un ejemplo abstracto básico.

Programas Benchmark

Hay una serie de propiedades aplicadas a medidas de prestaciones:

- Fidabilidad; que sean representativas, evalúen diferentes componentes del sistema y reproducibles.
- Permitir comparar diferentes realizaciones de un sistema o diferentes sistemas.

Hay varios tipos de Benchmark:

- De bajo nivel: test plug-pang, evaluación de los operadores con enteros y flotantes.
- Kernels: resolución de sistemas de ecuaciones, multiplicación de matrices, descomposiciones ...
- Sistémicos: Blas, Linpack, Whetstone
- Programas reales
- Aplicaciones diseñadas