
# UNIVERSIDAD DE GRANADA

## Grado en Ingeniería Informática

## Arquitectura de computadores — Teoría

**Atanasio José Rubio Gil**
https://github.com/Groctel/ugr-informatica

April 20, 2020



# Índice general

# Tema 1:
# Arquitecturas paralelas: Clasificación y prestaciones

## §1.1: Clasificación del paralelismo implícito en una aplicación

A la hora de trabajar en un sistema informático es muy común encontrarnos con el caso de que hay tareas que podemos ejecutar paralelamente. Por ejemplo, si tenemos los vectores $\vec{A}$, $\vec{B}$, $\vec{C}$, $\vec{D}$, $\vec{E}$ y $\vec{F}$, podemos realizar paralelamente las operaciones $\vec{A}+\vec{B}=\vec{C}$ y $\vec{D}\times\vec{E}=\vec{F}$, ya que el resultado de una no afecta al resultado de la otra. A más bajo nivel, definimos un vector de dimensión $n$ en un sistema informático como una lista ordenada de $n$ elementos tal que `vector[n] = [0,1,2,…,n-1]`, por lo que definimos la suma de dos vectores `v1[n]` y `v2[n]` de la siguiente forma:

```
1 for (int i=0; i<n; i++)
2    v3[i] = v1[i] + v2[i];
```

Para dos vectores de $n$ elementos y un sistema que permita $p$ cómputos paralelos, podemos dividir los vectores en $n/p$ secciones y calcularlas paralelamente. Por supuesto, no tiene sentido hablar de **paralelismo** si la máquina no tiene los recursos necesarios.

Podemos clasificar el paralelismo implícito en un programa mediante tres criterios:

- **Nivel:** El grado de abstracción sobre el cual existe paralelismo.
- **Paralelismo de tareas/datos:** Parelelismo en diferentes tareas (suma y multiplicación) o datos (suma de vectores).
- **Granularidad:** Conjunto de subtareas que conforman una tarea.

Decimos que una tarea tiene una granularidad más fina (o menos gruesa) cuantas menos operaciones sean necesarias para su ejecución. En orden descendiente de granularidad, podemos dividir los programas en rutinas formadas por bloques que ejecutan operaciones primitivas.

El paralelismo viene determinado por la posible existencia de **dependencias de datos**. Decimos que dos bloques de un programa son dependientes si referencian una misma variable y, más específicamente, decimos que un bloque $B_2$ es dependiente con respecto a $B_1$ si $B_1$ aparece secuencialmente antes que $B_2$. Distinguimos entre tres tipos de dependencias de datos:

- ***RAW* (*Read After Write*):** Dependencia verdadera. Se produce cuando un $B_2$ lee una variable después de que $B_1$ escriba sobre ella.
- ***WAW* (*Write After Write*):** Dependencia de salida. Se produce cuando $B_1$ y $B_2$ escriben secuencialmente sobre la misma variable, afectando a la lectura de la misma por otros bloques.
- ***WAR* (*Write After Read*):** Antidependencia. Se produce cuando $B_2$ escribe sobre una variable después de que $B_1$ la haya leído.

```
// RAW
a = b + c
d = a + c
// WAW
a = b + c
a = d + e
// WAR
b = a + c
a = d + e
```

No tiene sentido hablar de una dependencia *RAR*, ya que la lectura múltiple de una misma variable recoge siempre el mismo dato si ésta no se modifica en el proceso.

Podemos distinguir entre dos tipos de paralelismos **implícitos**:

- **Paralelismo de tareas o *TLP* (*Task Level Par.*):** Viene de extraer la estructura lógica de rutinas de un programa. Está relacionado con el paralelismo a nivel de función.
- **Paralelismo de datos o *DLP* (*Data Level Par.*):** Viene implícito en las operaciones con estructuras de datos y se extrae de la representación matemática del programa. Está relacionado con el paralelismo a nivel de bucle

Un programa que se ejecute con una segmentación de cauce[1], es decir, que esté dividido en varios estadios secuenciales, puede ofrecer paralelismo a nivel de función de forma que, mientras el tercer estadio esté trabajando con el primer lote de datos, el segundo está trabajando con el segundo y el primero, con el tercero.

A nivel de arquitecturas existe paralelismo debido a que un sistema puede ejecutar a la vez varios procesos que gestionan múltiples hebras para ejecutar instrucciones de forma paralela. Distinguimos aquí tres términos:

- **Instrucciones:** Son las operaciones que puede gestionar la unidad de control del computador.
- **Hebras[2]:** Representan la menor unidad de ejecución gestionable por el SO, la menor secuencia de instrucciones ejecutables paralela o concurrentemente.
- **Procesos:** Representan la mayor unidad de ejecución gestionable por el SO y constan de una o varias hebras.

A nivel de granularidad, las hebras son más finas que los procesos, ya que se tarda menos tiempo en crearlas, destruirlas y conmutar y establecer canales de comunicación entre ellas que en hacer las mismas tareas con procesos.

<table>
<tr><th>Detectable a nivel de</th><th>Programas</th><th>Funciones</th><th>Bloques</th><th>Operaciones</th></tr>
<tr><td rowspan="3">Utilizado (explícito) a nivel de</td><td colspan="3" rowspan="2"></td><td>Instrucciones</td></tr>
<tr><td rowspan="3">IntraInstrucciones</td></tr>
<tr><td colspan="2">Hebras</td></tr>
<tr><td colspan="3">Procesos</td><td rowspan="4"></td></tr>
<tr><td rowspan="4">Implementado por arquitecturas aprovechando</td><td colspan="3" rowspan="2"></td><td>ILP</td></tr>
<tr><td rowspan="3">SIMD</td></tr>
<tr><td colspan="2">Multihebra</td></tr>
<tr><td colspan="3">Multicomputador</td></tr>
<tr><td colspan="3">Multiprocesador</td><td rowspan="2"></td></tr>
<tr><td rowspan="3">Extraído por (implícito → explícito)</td><td colspan="3" rowspan="2"></td><td>Arquitectura</td></tr>
<tr><td rowspan="3">Herramienta de programación</td></tr>
<tr><td colspan="3">Usuario-Programador</td></tr>
</table>

Figura 1.1: Detección, utilización, implementación y extracción del paralelismo a diferentes niveles

[1] Estructura de computadores, tema 4.
[2] Sistemas operativos, tema 2.

# §1.2: Clasificación de arquitecturas paralelas

## 1.2.1: Computación paralela y computación distribuida

La **computación paralela** estudia los aspectos hardware y sofrware relacionados con el desarrollo y ejecución de aplicaciones en un sistema de cómputo compuesto por múltiples núcleos, procesadores o computadores que es visto externamente como una unidad autónoma. Por otro lado, la **computación distribuida** estudia los aspectos hardware y software relacionados con el desarrollo y ejecución de aplicaciones en un sistema distribuido, que es una colección de recursos autónomos situados en distintas localizaciones físicas. En general, nos referimos a computación paralela cuando el trabajo es realizado por un único computador y a computación distribuida cuando es realizado por varios.

### Computación grid y *cloud*

La **computación distribuida a baja escala** estudia los aspectos relacionados con el desarrollo y ejecución de aplicaciones en un conjunto de computadores de un único dominio situados en distintas localizaciones físicas conectadas a través de una infraestructura de red local Por su parte, la **computación grid** estudia los aspectos relacionados con el desarrolo y ejecución de aplicaciones en un conjunto de computadores de varios dominios geográficamente distribuidos conectados con una infraestructura de telecomunicaciones.

La computación ***cloud*** estudia los aspectos relacionados con el desarrollo y ejecución de aplicaciones en un sistema *cloud*, que es un sistema que ofrece servicios de infraestructura, plataforma y/o software a los que se accede normalmente mediante una interfaz de auto-servicio. Estos servicios son de pago (*pay-per-use*) y ofrece recursos virtuales que, al ser una abstracción de los físicos, parecen ilimitados en número y capacidad y son gestionados de forma inmediata sin interacción con el proveedor. Estos recursos soportan el acceso de múltiples clientes y están conectdos con métodos estándar independientes de la plataforma de acceso.

## 1.2.2: Clasificaciones de arquitecturas y sistemas paralelos

### Clasificación comercial: Segmento de mercado

En función del consumidor objetivo de los computadores podemos clasificarlos de mayor a menor precio y prestaciones:

| **Computador** | **Núcleos** | **Precio (€)** |
|---|---:|---:|
| Supercomputadores | $128 < x$ | $5000000 < x$ |
| Servidores de gama alta | $4 < x < 256$ | $750000 < x < 10000000$ |
| Servidores de gama media | $2 < x < 64$ | $50000 < x < 1000000$ |
| Servidores de gama baja | $2 < x < 16$ | $1000 < x < 10000$ |
| PCs y WorkStations | $x < 4$ | $x < 10000$ |
| Computadores empotrados | $- - -$ | $- - -$ |

Los computadores externos (de escritorio, portátiles, servidores, clústers…) se utilizan para todo tipo de aplicaciones, ya sean de oficina, entretenimiento, procesamiento de trasacciones (OLTP), sistemas de soporte de decisiones (DSS), científicas, animación…Por su parte, los computadores empotrados se utilizan para aplicaciones de propósito específico, como videojuegos, coches, teléfonos, electrodomésticos…Estos últimos tienen restricciones como un consumo de potencia, precio y tamaño reducidos y que deben realizar cómputos en tiempo real.

### Clasificación de Flynn de arquitecturas

En 1972, Michael J. Flynn propone el siguiente esquema de clasificación de arquitecturas en función del flujo de instrucciones y datos:

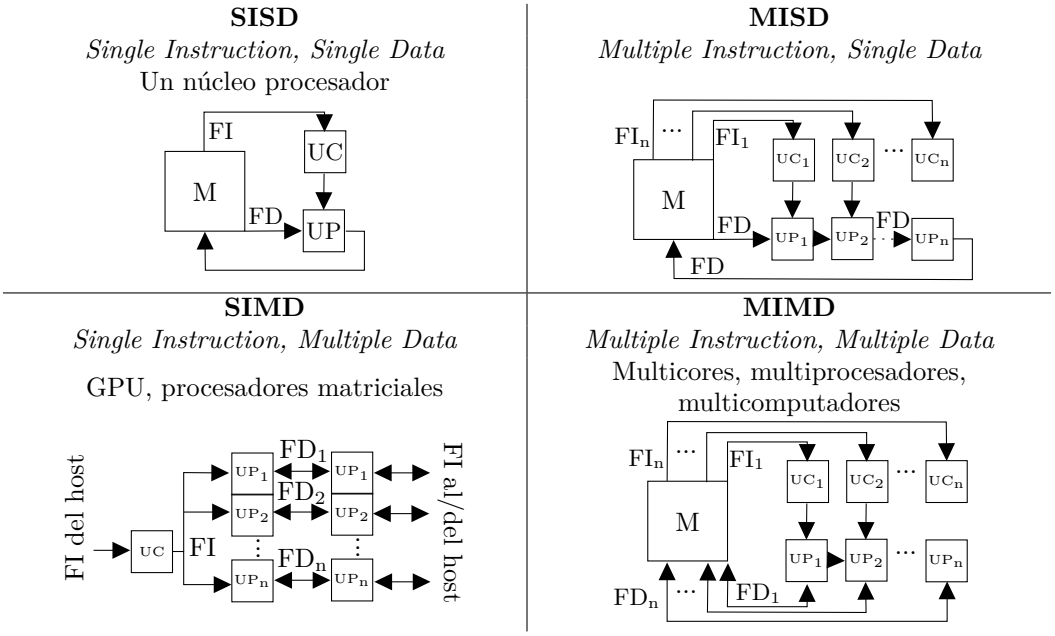


Figura 1.2: Clasificación de Flynn de arquitecturas

### Arquitecturas SISD

Corresponden a computadores uni-procesador, en los que se evalúa una instrucción y un dato cada vez. Siguen una estructura puramente secuencial y no permiten paralelismo.

```
for i in 1 to 4 do
   C[i] = A[i] + B[i]
   F[i] = D[i] - E[i]
   K[i] = H[i] * G[i]
done
```

### Arquitecturas SIMD

Aprovechan el paralelismo de datos para poder procesar un número de datos mayor en cada operación. Es el caso de los operadores vectoriales, que permiten trabajar con vectores mediantes instrucciones como `ADDV`, `SUBV` o `MULV`. Estas instrucciones permiten trabajar directamente con elementos de vectores con mayor rendimiento. Por su parte, los procesadores matriciales permiten trabajar con vectorialmente con múltiples vectores (por ejemplo, con matrices $EP_i$).

```
for all EPi(i in 1 to 4) do
   C[i] = A[i] + B[i]
   F[i] = D[i] - E[i]
   K[i] = H[i] * G[i]
done
```

En este caso, el procesador vectorial ejecutará el bucle `for all` cuatro veces, aprovechando en cada una el paralelismo de las instrucciones `ADDV`, `SUBV` y `MULV`.

### Arquitecturas MISD

Aunque podemos simular este modelo en un código para programas que procesan secuencias o flujos de datos, no sexisten computadores que funcionen con esta arquitectura.

### Arquitecturas MIMD

Es la arquitectura de las máquinas multinúcleo, multiprocesador y multicomputador. Esta multitud de componentes hace que puedan aprovechar el paralelismo a nivel de procesos.

```
{Proceso 1}
for i in 1 to 4 do
   C[i] = A[i] + B[i]
done
{Proceso 2}
for i in 1 to 4 do
   F[i] = D[i] - E[i]
done
{Proceso 3}
for i in 1 to 4 do
   K[i] = H[i] * G[i]
done
```

## Clasificación según el sistema de memoria

### Multiprocesadores

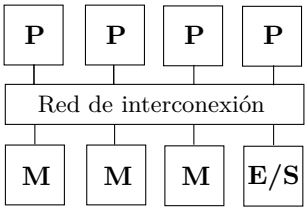


Figura 1.3: Arquitectura multiprocesador con memoria centralizada (SMP)

Son aquellos en los que todos los procesadores comparten el mismo espacio de direcciones, permitiendo al programador trabajar sin necesitar conocer dónde están almacenados los datos. La comunicación entre procesos se hace explícita mediante variables compartidas, de forma que no existe varias instancias del mismo dato en memoria principal. Sin enbargo, la latencia de las operaciones es alta y el sistema es poco escalable, ya que requiere aumentar la caché dle procesador, usar redes de menor latencia y ancho de banda que un bus y distribuir físicamente los módulos de memoria entre los procesadores sin dejar de compartir el espacio de direcciones.

Debido a que la distribución de código y datos entre procesadores no es necesaria en estas arquitecturas y que la sincronización se implementa mediante primitivas, programar en arquitecturas SMP es, generalmente, más sencillo que en arquitecturas multicomputador.

### Multicomputadores

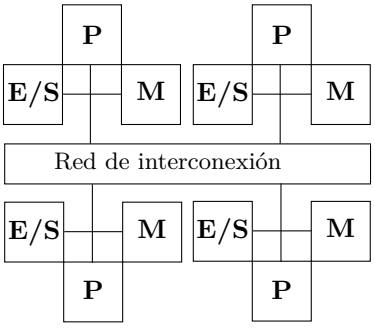


Figura 1.4: Arquitectura multicomputador

Son aquellos en los que cada procesador tiene un espacio de direcciones propio, por lo que el programador debe conocer dónde están almacenados los datos con los que opera. La comunicación entre procesos se hace explícita mediante programas de paso de mensajes, de forma que eisten múltiples instancias de los datos en memoria principal, ya que tienen que leerse por la memoria de cada computador individual. Como contrapartida a esta

complejidad, la latencia de las operaciones es baja y el sistema es más escalable, ya que únicamente requiere conectar más computadores al sistema de paso de mensajes.

Debido a que la distribución de código y datos entre los procesadores es necesaria, lo que conlleva a usar herramientas de programación más sofisticadas, y que la sincronización entre procesos se hace mediante programas de comunicación, la programación en arquitecturas multicomputador es, generalmente, más difícil que en arquitecturas SMP.

#### Comunicación *uno a uno* en SMP y multicomputador

En las arquitecturas SMP, la comunicación *uno a uno*, que se abordará en **??**, se da mediante accesos concurrentes a la memoria principal, de forma que un nodo fuente esnvía una dirección a un nodo destino y esperan a una respuesta del otro para poder seguir operando. Esta sincronización se conoce como el **problema del productor-consumidor**[3].

En las arquitecturas multicomputador, la red de comunicación es un búfer de datos gestionado por un programa de paso de mensajes que recibe peticiones de envío y recepción de datos y las coordina adecuadamente para garantizar que se cumplen las propiedades de seguridad y vivacidad del sistema. Debido a la estructura de esta red de comunicación, las funciones de recepción de mensajes son bloqueantes para el proceso receptor.

#### Incremento de escalabilidad en multiprocesadores

Las arquitecturas multiprocesador presentan un gran inconveniente en su escalabilidad con respecto a las arquitecturas multicomputador. Mientras las últimas requieren poco más que instalar un nuevo sistema en la red de comunicación, las primeras requieren que se realicen modificaciones estructurales sobre el sistema, como aumentar la cache de los procesadores, usar redes de menor latencia y ancho de banda que un bus y distribuir físicamente los módulos de memoria entre los procesadores asegurando que se siga compartiendo el espacio de direcciones.

## Clasificación según el sistema de memoria

Debido a que las arquitecturas multicomputador están compuestas por diferentes máquinas con sus módulos de procesamiento, E/S y memoria independientes, éstas siguen el sistema de memoria **NORMA** (*No Remote Memory Access*).

Por otro lado, las arquitecturas multiprocesador sí comparten memoria en un único espacio de direcciones. Esta memoria puede ser uniforme (**NUMA**) o no (**UMA**).

#### NUMA (*Non-Uniform Memory Access*)

Este tipo de memoria, que también puede ser utilizada en redes multicomputador, se caracteriza porque el tiempo de acceso depende de la ubicación de la memoria relativa al procesador que realiza la petición de acceso, siendo el acceso a la memoria local al procesador más rápido que a las memorias externas. Una variante de esta memoria es **CC-NUMA** (*Cache Coherent NUMA*). Aunque mantener coherencia en la caché en memorias NUMA lleva consigo una gran sobrecarga (**??**), la escalabilidad de las memorias NUMA sin coherencia en caché son prohibitivamente complejas de gestionar, por lo que se prefiere utilizar memorias CC-NUMA por mucho que ofrezcan un muy bajo rendimiento cuando varios procesadores intentan acceder en sucesiones rápidas a la misma dirección de memoria.

Como caso particular de las memorias NUMA, las memorias **COMA** (*Cache-Only Memory Architecture*) sólo trabajan con cachés, de forma que un acceso a un dato en memoria puede hacer que éste migre a otra. Esto reduce el número de copias redundantes de los datos a lo largo del sistema, pero plantea problemas de ubicación de los datos y las acciones a realizar al llenarse el sistema mememoria, que suelen subsanarse mediante mecanismos de hardware de coherencia de memoria.

#### UMA (*Uniform Memory Access*)

En este sistema, todos los procesadores comparten uniformemente la misma memoria física, de forma que los accesos a la misma son de igual para todos los procesadores y no existe redundancia de datos en la memoria principal (aunque podría existirla en la caché de los procesadores). Es el tipo de memoria utilizada por las arquitecturas SMP.

[3] Sistemas concurrentes y distribuidos, tema 1

# §1.3: Evaluación de prestaciones de una arquitectura

## 1.3.1: Medidas usuales para evaluar prestaciones

### Tiempo de respuesta

El tiempo de respuesta de un programa está compuesto por la suma de tres tiempos:

- **Tiempo de usuario:** Tiempo en ejecución en el espacio del usuario.
- **Tiempo de sistema:** Tiempo de ejecución en el epsacio del kernel.
- **Tiempo en espera:** Tiempo que el programa está esperando a operaciones de E/S o a que finalice otro proceso.

$$Tiempo\ de\ respuesta = tCPU_{user} + tCPU_{sys} + t_{espera}$$

Tenemos varias formas de obtener el tiempo de ejecución de un programa:

| Función | Fuente | Tipo | Precisión (ms) |
|---|---|---|---|
| `time` | `/usr/bin/time` | elapsed, user, system | 10000 |
| `clock()` | `time.h` | CPU | 10000 |
| `gettimeofday()` | `sys/time.h` | elapsed | 1 |
| `clock_gettime()` | `time.h` | elapsed | 0.001 |
| `omg_get_wtime()` | `omp.h` | elapsed | 0.001 |
| `SYSTEM_CLOCK()` | Fortran | elapsed | 1 |

### Tiempo de CPU

El **tiempo de CPU** ($T_{CPU}$) de un programa se calcula como el producto del número de ciclos del mismo y el tiempo que tarda el procesador en ejecutar cada ciclo. Podemos expresarlo tambien como el cociente entre el número de ciclos y la frecuencia de reloj del procesador, que es inversamente proporcinal al tiempo de ciclo.

$$T_{CPU} = Ciclos \cdot T_{ciclo} = \frac{Ciclos\ del\ programa}{Frecuencia\ del\ reloj}$$

También podemos calcular el tiempo de CPU de un programa como el producto entre el número de instrucciones del mismo, el número de **ciclos por instrucción** ($CPI$) y el tiempo por ciclo. Trivialmente, se tiene que el $CPI$ de un programa es el cociente entre los ciclos del mismo y el número de instrucciones por las que está compuesto.

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

$$\text{Ciclos por instrucción } (CPI) = \frac{Ciclos\ del\ programa}{Número\ de\ instrucciones\ (NI)}$$

Dado un programa cualquiera, éste está compuesto por un número $I_i$ de instrucciones del tipo $i \forall i \in \mathbb{N}$. Si cada instrucción del tipo $i$ consume $CPI_i$ ciclos y hay $k$ tipos de instrucciones distintos, podemos expresar el número de ciclos del programa y, por consiguiente, el CPI de la siguiente manera:

$$Ciclos\ del\ programa = \sum_{i=0}^{k} CPU_i \cdot I_k$$

$$CPI = \frac{\sum_{i=0}^{k} CPI_i \cdot I_i}{Número\ de\ instrucciones}$$

Otra forma de obtener el $CPI$ es mediante el cociente entre los **ciclos por emisión** ($CPE$) e **instrucciones por emisión**, que son el número mínimo de ciclos transcurridos entre los instantes en los que el procesador puede emitir instrucciones y el número de instrucciones emitibles cada vez que se produce una emisión, respectivamente.

$$CPI = \frac{CPE}{IPE}$$

Distinguimos cinco segmentos en el cauce de un procesador[4]. Un procesador no segmentado ejecutará dos instrucciones de forma secuencial ocupando cada una los cinco segmentos cada vez, de forma que cada instrucción deberá emitirse una a una y tardará cinco ciclos en completarse. Un procesaador segmentado ejecutará las instrucciones superpuestas entre sí de forma que mientras una instrucción $I_i$ esté ejecutando en el segmento $n_i$, la instrucción $I_{i-1}$ estará ejecutando en el segmento $n_{i-1}$. Por último, un procesador superescarlar podrá ejecutar varias instrucciones en un solo segmento ($I_i$ e $I_{i-1}$ ejecutándose paralelamente en $n_i$).

Para un modelo no segmentado, uno segmentado sin riesgos y uno superescalar sin riesgos y capaz de ejecutar dos instrucciones paralelamente obtendríamos los siguientes valores:

| | **CPE** | **IPE** | **CPI** |
|---|---|---|---|
| **Procesador no segmentado** | 5 | 1 | 5 |
| **Procesador segmentado** | 1 | 1 | 1 |
| **Procesador superescalar** | 1 | 2 | 0.5 |

El número de instrucciones es calculable como el cociente entre el **número de operaciones** ($N_{op}$) que realiza el programa y el **número de operaciones codificables en una instrucción** ($OP_{instr}$).

$$NI = \frac{N_{op}}{OP_{instr}}$$

A la hora de mejorar el tiempo de CPU, las mejoras en la tecnología y en la estructura y organización del computador afectan al $CPI$ y al tiempo por ciclo, mientras que las mejoras en el repertorio de instrucciones y del compilador afectan al $CPI$ y al número de instrucciones.

## Productividad: MIPS y MFLOPS

### MIPS

El número de instrucciones que un procesador puede ejecutar en un segundo se mide en ***MIPS***[5] (*Millions of Instructions Per Second*). El valor de los $MIPS$ depende del repertorio de instrucciones, por lo que es difícil comparar máquinas con repertorios distintos, y puede variar en función del programa ejecutado, por lo que no sirve para caracterizar la máquina que se evalúa. Un mayor valor de $MIPS$ corresponde a peores prestaciones de la máquina.

$$MIPS = \frac{NI}{T_{CPU} \cdot 10^6} = \frac{Frecuencia\ del\ reloj}{CPI \cdot 10^6}$$

### MFLOPS

Por otro lado tenemos los ***MFLOPS*** (*Millions of FLoating Operations Per Second*). Ésta no es una medida adecuada para todos los programas, ya que algunos pueden no realizar operaciones en coma flotante, ni es directamente desplazable a todas las máquinas, ya que el conjunto de operaciones en coma flotante no es

[4] Estructura de computadores, tema 4.
[5] *Meaningless Indication of Processor Speed.*

constante en máquinas diferentes y la potencia de las operaciones no es igual en todas ellas. Se hace necesaria una normalización de las instrucciones en coma flotante para poder calcular correctamente los $MFLOPS$.

$$MFLOPS = \frac{Operaciones\ en\ coma\ flotante}{T_{CPU} \cdot 10^6}$$

## 1.3.2: Conjunto de programas de prueba (*Benchmark*)

Los ***benchmarks*** son programas que analizan las prestaciones de los procesadores para comparar diferentes sistemas o realizaciones de un mismo sistema con un método fiable y reproducible. Distinguimos diferentes tipos de *benchmarks*:

- **De bajo nivel (*microbenchmark*):** Evaluación de operaciones con números enteros o de coma flotante.
- **Kernels:** Resolución de sistemas de ecuaciones, factorización y multiplicación de matrices…
- **Sintéticos:** Programas como Dhrystone o Whetstone utilizados para poner a prueba las capacidades de cálculo de las arquitecturas.
- **Programas reales:** Puede usarse como *benchmark* el rendimiento de `gcc`, `zip` u otros programas que realicen trabajos pesados con una gran cantidad de datos.
- **Aplicaciones diseñadas:** Programas de predicción de tiempo o simulación de terremotos, entre otros, pueden utilizarse para evaluar las prestaciones del computador debido a la inmensa cantidad de datos que manejan.

## 1.3.3: Ganancia en prestaciones

Al incrementar las prestaciones de un recurso de un procesador haciendo que su velocidad sea $n$ veces mayor (usando $n$ procesadores en lugar de uno, realizando la ALU las operaciones en un tiempo $n$ veces menor…), se tiene que el incremento de velocidad alcanzable en la nueva situación con respecto a la previa (la máquina base), se expresa como el cociente entre la velocidad de la máquina mejorada $V_n$ y la velocidad de la máquina base $V_0$ o como el cociente entre el tiempo de ejecución en la máquina mejorada $T_n$ y el tiempo de ejecución en la máquina base $T_0$. Llamamos a este incremento la **ganancia de velocidad** o *speed-up* $S_n$.

$$S_n = \frac{V_n}{V_0} = \frac{T_0}{T_n}$$

### Ley de Amdahl

La ley de Amdahl afirma que la mejora de velocidad $S$ que se puede obtener cuando se mejora un recurso de una máquina en un factor $n$ está limitada por la fracción del tiempo de ejecución en la máquina sin la mejora durante el tiempo que dicha mejora es inaplicable:

$$S \leq \frac{n}{1 + f \cdot (n-1)}$$

Por ejemplo, si un programa pasa el 25 % de su tiempo de ejecución realizando instrucciones de coma flotante y se mejora la máquina de forma que estas instrucciones se realicen en la mitad de tiempo, tenemos que $n = 2$, $f = 0{,}75$.

$$S \leq \frac{2}{1 + 0{,}75 \cdot (1-1)} = 1{,}14$$

# Tema 2:
# Programación paralela

## §2.1: Herramientas, estilos y estructuras en programación paralela

### 2.1.1: Problemas que plantea la programación paralela al programador

La programación paralela plantea problemas inherentes a la misma que no se dan en la programación secuencial, como la división del cómputo total en tareas independientes, la agrupación de dichas tareas en procesos o hebras, la asignación de estos procesos y hebras a los procesadores y la sincronización y comunicación entre estos procesos. Estos problemas deben ser abordados tanto por la herramienta de programación como por el programador.

Para escribir un programa de forma paralela partimos de su versión secuencial y lo dividimos en bloques sin dependencias de datos. También podemos utilizar versiones optimizadas para la programación paralela de las bibliotecas que importemos.

Al trabajar con programación paralela en arquitecturas MIMD podemos hacerlo de forma **SPMD**, paralelizando un solo programa, (*Single Program Multiple Data*) y **MPMD** (*Multiple Program Multiple Data*), paralelizando la ejecución de varios programas que a su vez están programados de forma paralela.

### 2.1.2: Herramientas para obtener código paralelo

Para crear programas de ejecución paralela podemos utilizar las tres técnicas ordenadas de menor a mayor abstracción:

- **Compiladores paralelos:** Extraen automáticamente el paralelismo de los programas que compilan, de forma que el programador no tiene que explicitarla.
- **Lenguajes paralelos y API de directivas:** La sintáxis de los lenguajes paralelos como Occam, Ada o Haskell o las directivas de OpenMP permiten indicar cómo paralelizar el programa en el código a gusto del programador.
- **API de funciones:** Las APIs de alto nivel como OpenMPI permiten paralelizar la programación mediante paso de mensajes y otras técnicas de abstracción alta.

Las herramientas de paralelización permiten, implícita o explícitamente, localizar el paralelismo de los programas dividiéndolos en tareas independientes, asignar tareas a los procesos y hebras, crear y terminar estos procesos y hebras y comunicarlos y sincronizarlos. El ***mapping***, la asignación de las diferentes tareas a procesos y threads, puede hacerlo el programador, la herramienta de programación paralela o el propio sistema operativo.

Por ejemplo, así se haría un cálculo de el número $\pi$ con OpenMP:

```c
#include <opm.h>
#include <stdlib.h>
#define NUM_THREADS 4

int main (int argc, char ** argv) {
   double ancho,
          sum = 0,
          x;
   int intervalos;

   intervalos = atoi(argv[1]);
   ancho      = 1.0 / (double) intervalos;

   omp_set_num_threads(NUM_THREADS);

   #pragma omp parallel
   {
      #pragma omp for reduction(+:sum) private(x) schedule(dynamic)
      for (int i=0; i<intervalos; i++) {
         x    = (i + 0.5) * ancho;
         sum += 4.0 / (1.0 + x * x);
      }
   }

   sum *= ancho;
}
```

Y así se haría el mismo cómputo con MPI:

```c
#include <mpi.h>
#include <stdlib.h>

int main (int argc, char ** argv) {
   double ancho,
          lsum,
          sum = 0,
          x;
   int intervalos
       iproc,
       nproc;

   if (MPI_Init(&argc, &argv) != MPI_SUCCESS)
      exit (1);

   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
   MPI_Comm_rank(MPI_COMM_WORLD, &iproc);

   intervalos = atoi(argv[1]);
   ancho = 1.0 / (double) intervalos;

   for (int i=iproc; i<intervalos; i=nproc) {
      x = (i + 0.5) * ancho;
      sum += 4.0 / (1.0 + x * x);
   }

   lsum *= ancho;

   MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   MPI_Finalize();
}
```

## Comunicaciones colectivas

Distinguimos entre diferentes tipos de comunicaciones entre procesos y hebras:

### Comunicación *uno a todos*

Se caracterizan porque un único proceso envía un mensaje a varios procesos al mismo tiempo. Esto puede conseguirse mediante una **difusión** (*broadcast*) del mensaje, que envía un mensaje $x$ a todos los procesos a la vez, o mediante una **dispersión** (*scatter*), que envía envía un mensaje $x_i$ a cada proceso $i$.

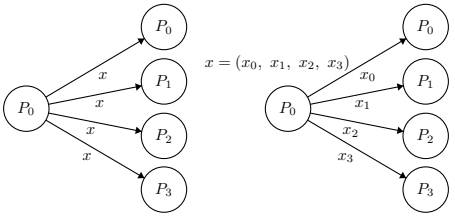


Figura 2.1: Difusión y dispersión de mensajes

### Comunicación *todos a uno*

Se caracterizan porque un único proceso recibe un mensaje a partir de los mensajes enviados por varios procesos. Esta recepción puede hacerse mediante **reducción** cuando los mensajes recibidos son argumentos de una función o mediante **acumulación** (*gather*), cuando se recogen indistintamente todos los mensajes.

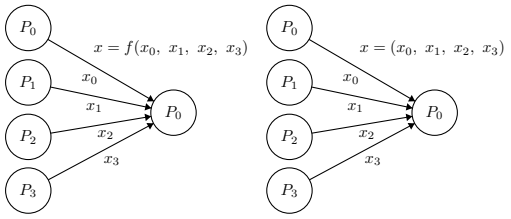


Figura 2.2: Reducción y acumulación de mensajes

### Comunicación *todos a todos*

En este tipo de comunicación todos los procesos se comunican con todos. Esto se puede hacer mediante un sistema en el que **todos difunden** (*all-broadcast*), también conocido como chismorreo (*gossiping*), o mediante un sistema en el que **todos dispersan** (*all-scatter*).

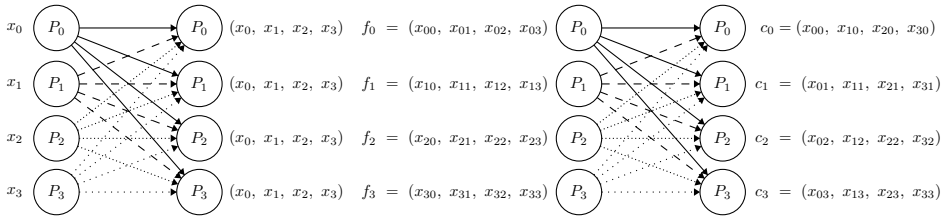


Figura 2.3: Todos difunden y todos dispersan (matriz de filas $f_i$ y columnas $c_i$)

### Comunicación *múltiple uno a uno*

Se produce cuando varios procesos ejecutándose paralelamente se comunican *uno a uno* entre sí. Esta comunicación puede hacerse por permutaciones de **rotación** o por permutaciones de **baraje-$x$**, en las que cada proceso $i$ manda su mensaje al proceso $i \cdot k \ mód\ x$ para un total de $k$ procesos.

### Comunicaciones *compuestas*

Las comunicaciones compuestas se definen por diferentes modos de reducción de los mensajes enviados por los procesos en los que **todos combinan** o re realizan por un **recorrido** (*scan*) paralelo que puede ser **prefijo** o **sufijo** en función de si los mensajes se recogen en orden de índice ascendente o descendente respectivamente.

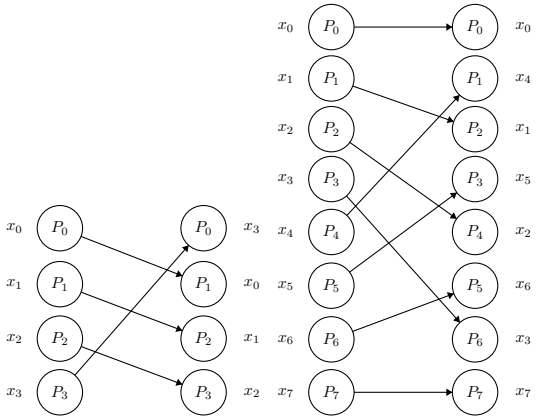

Figura 2.4: Permutación por rotación y por baraje-2

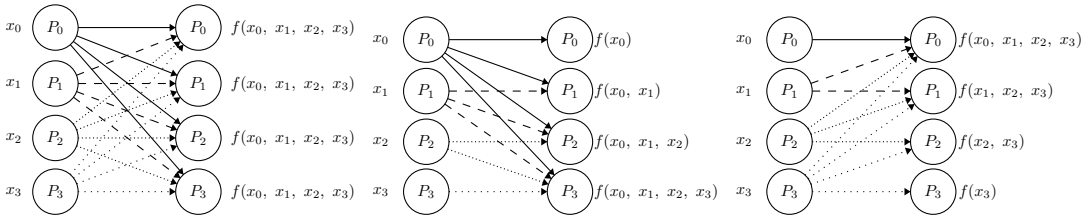

Figura 2.5: Todos combinan, recorrido prefijo paralelo y recorrido sufijo paralelo

En OpenMP podemos utilizar las siguientes directivas y cláusulas para crear sistemas de comunicación colectiva:

| **Servicio** | **Tipo** | **Directivas** |
|---|---|---|
| *uno a todos* | Difusión | Cláusula `firstprivate` desde la hebra 0<br>Directiva `single` con cláusula `copyprivate`<br>Directiva `threadprivate` y cláusula `copyin` en directiva `parallel` |
| *todos a uno* | Reducción | Cláusula `reduction` |
| *servicios compuestos* | Barreras | Directiva `barrier` |

En MPI podemos usar las siguientes funciones:

| **Servicio** | **Tipo** | **Función** |
|---|---|---|
| *uno a uno* | Asíncrona | `MPI_Send()` o `MPI_Receive()` |
| *uno a todos* | Difusión<br>Dispersión | `MPI_Bcast()`<br>`MPI_Scatter()` |
| *todos a uno* | Reducción<br>Acumulación | `MPI_Reduce()`<br>`MPI_Gather()` |
| *todos a todos* | Todos difunden | `MPI_Allgather()` |
| *todos a uno* | Todos combinan<br>Barreras<br>Recorrido | `MPI_Allreduce()`<br>`MPI_Barrier()`<br>`MPI_Scan()` |

## 2.1.3: Paradigmas de programación paralela

Dependiendo de la arquitectura con la que estemos trabajando, podemos utilizar diferentes formas de programación paralela:

### Paso de mensajes

Es el paradigma utilizado en las arquitecturas multicomputador, que necesitan que el computador emisor envíe al receptor los datos con los que necesita trabajar. Estos sistemas se pueden programar con lenguajes como Ada u Occam y APIs como MPI o PVM.

### Variables compartidas

En los sistemas multiprocesador las variables compartidas pueden alojarse en la memoria compartida y ser accesible por todos los procesadores al mismo tiempo (con sus consecuentes problemas de concurrencia). Lenguajes como Ada o Java trabajan en este paradigma, así como APIs como OpenMP, Intel TBB (*Threading Building Blocks*) y las hebras POSIX.

### Paralelismo de datos

Es la forma de trabajar de los procesadores matriciales, que requieren una gran potencia de procesamiento. Se implementa con lenguajes como HPF (*High Performance Fortran*) o Fortran 95, en el que los bloques `forall` permiten paralelizar operaciones con matrices y vectores, y con APIs como Nvidia CUDA.

## 2.1.4: Estructuras típicas de códigos paralelos

### *Maestro-Esclavo* o granja de tareas

El proceso *maestro* reparte el trabajo a varios procesos *esclavos*[1], que realizan su cómputo individualmente y envían el resultado al proceso *maestro* para que éste haga un cómputo final con todos los resultados recolectados.

### Cliente/servidor

Varios procesos *cliente* mandan peticiones a un proceso *servidor*, que las gestiona y envia respuestas a los procesos *cliente* con el resultado de los cómputos.

### Descomposición de dominio o datos

Cada proceso adquiere una parte de la computacióna a realizar y entre todos resuelven el problema dividiéndolo en partes equitativas. Un ejemplo de esto sería el uso de la directiva `omp for`:

```
1 omp_set_num_threads(hebras)
2
3 #pragma omp for
4 for (int i=0; i<hebras; i++)
5    // Acción a realizar individualmente por cada hebra
```

### Segmentación o flujo de datos

Los procesos se organizan de la misma forma que un procesador segmentado de forma que, para cada segmento $i$, éste pueda ejecutarse mientras el resto de segmentos están procesando otra información que llegará al procesador $i$ o de éste.

[1] Es curioso que teniendo el término *granja de tareas* se siga utilizando la notación *Maestro-Esclavo* en lugar de algo más humanitario como *Granjero-Plantación*. El proceso granjero *planta* una serie de subprocesos hijos y espera para cosechar los resultados.

### Divide y vencerás, descomposición recursiva

El problema se divide en subproblemas recursivos más pequeños y cada uno de los procesos ejecuta los cómputo de cada uno de los subproblemas. Cuando dos subproblemas llegan a su caso ancestro común, uno de los procesos se elimina y se continúan los cómputos con el proceso restante.

# §2.2: Proceso de paralelización

A la hora de paralelizar un programa seguimos cuatro pasos:

- Descomponer el programa en tareas independientes.
- Asignar tareas a procesos y hebras.
- Redactar el código paralelo como vimos en §??.
- Evaluar las prestaciones que ofrece la paralelización.

Estos cuatro pasos deben seguirse secuencialmente, y de la evaluación podemos volver a cualquiera de los tres anteriores en función de los errores que hayamos detectado o dar por finalizada la paralelización del programa.

## 2.2.1: Descomposición en tareas independientes

Para descomponer un programa secuencial en tareas independientes debemos llevar a cabo un análisis de dependencias de datos entre las diferentes funciones en las que se divide y, dentro de éstas, entre los diferentes bloques que las componen. Al hacer esto podemos crear un grafo de dependencias que ilustre qué funciones y bloques pueden ejecutarse paralelamente en cada momento.

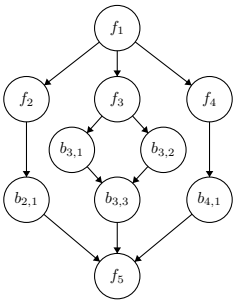


Figura 2.6: Grafo de dependencias entre las tareas de un programa

Por ejemplo, tengamos el siguiente código para aproximar el número pi:

```
int main (int argc, char ** argv) {
   double ancho,
          sum = 0,
          x;
   int intervalos;

   intervalos = atoi(argv[1]);
   ancho      = 1.0 / (double) intervalos;

   for (int i=0; i<intervalos; i++) {
      x    = (i + 0.5) * ancho;
      sum += 4.0 / (1.0 + x * x);
   }

   sum *= ancho;
}
```

En él, identificamos tres bloques principales:

- Declaración e inicialización de variables.
- Cálculo de la aproximación en bucle.
- Ajuste final de la aproximación.

Mientras que el primer y último bloque son indivisibles, el segundo podemos dividirlo en $n$ tareas que se ejecuten paralelamente para $n$ intervalos usados en la aproximación, de forma que cada uno de los valores de `i` en el bucle se ejecute de forma paralela a los otros.

## 2.2.2: Asignación de tareas a procesos y hebras

En esta fase distinguimos dos tipos de asignación de las tareas:

- **Planificación:** Agrupación de las tareas en hebras.
- ***Mapping*:** Asignación de las hebras a núcleos o procesadores.

La granularidad de la carga asignada a los procesos y hebras depende del número de núcleos o procesadores y del tiempo de comunicación y sincronización frente al tiempo de cálculo de cada hebra y procesador. Para que la ejecución de unas tareas no dependa de esperar el resultado de otras, se busca un **equilibrado de la carga** (*load balancing*) teniendo en cuenta que todas trabajen con un número lo más equitativo posible de datos y ejecuten un código lo más similar posible. Este equilibrado depende de la homogeneidad y uniformidad de la arquitectura sobre la que se trabaja y sobre la descomposición del programa realizada anteriormente.

La asignación de las tareas a cada una de las hebras se puede hacer de forma estática, asignando las hebras en tiempo de compilación, o dinámica, en cuyo caso la asignación se hace en tiempo de ejecución. En esta última diferentes ejecuciones del programa pueden dar lugar a asignaciones de las tareas sobre diferentes hebras o procesadores. Ambas asignaciones se pueden hacer de forma explícitas por el programador o de forma implícita por la herramienta de programación utilizada.

El *mapping* de las hebras se suele dejar al SO, que lo implementa mediante un sistema llamado *light-weight process*. También puede hacerse por el entorno o el sistema en tiempo de ejecución (*runtime system*) o explicitarse por el programador.

Debemos tener en cuenta que dividir el programa en más tareas paralelas que procesadores requiere una carga de trabajo de cambio de contexto para los procesadores que ralentiza la ejecución global del programa y que la creación de hebras conlleva un tiempo de ejecución que puede ser mayor que el tiempo de ejecución secuencial al sumarlo a la ejecución paralela de las tareas.

```
1  void F1 () {
2     #pragma omp parallel for schedule(static)
3     for (int i=0; i<N; i++)
4        // Código para i
5  }
6
7  void F2 () { /* ... */ }
8  void F3 () { /* ... */ }
9
10 int main () {
11    #pragma omp parallel section
12    {
13       #pragma omp section
14          F1();
15       #pragma omp section
16          F2();
17       #pragma omp section
18          F3();
19    }
20 }
```

En este ejemplo de asignación estática con OpenMP[2], cada función `F[1-3]` se ejecuta paralelamente con las otras y, dentro de ellas, los bucles `for` se ejecutan paralelamente en sus $N$ iteraciones.

# §2.3: Evaluación de prestaciones en procesamiento paralelo

## 2.3.1: Ganancia en prestaciones y escalabilidad

Podemos evaluar las prestaciones de un sistema de procesamiento paralelo en función de su tiempo de respuesta y productividad como vimos en §?? y según su escalabilidad y eficiencia. Para esta última podemos tener en cuenta las siguientes razones:

$$Eficiencia = \frac{Prestaciones}{Prestaciones\ máximas} \quad Rendimiento = \frac{Prestaciones}{N^{\circ}\ recursos} \quad \frac{Prestaciones}{Consumo\ potencia} \quad \frac{Prestaciones}{Área\ ocupada}$$

### Escalabilidad

Definimos la **gananancia en prestaciones** $S(p)$ (*speed-up*) para $p$ procesadores como la razón entre el tiempo de ejecución secuencial $T_s$ del programa y el tiempo de ejecución paralela $T_p(p)$ para $p$ procesadores. Este tiempo de ejecución paralela podemos calcularlo como la suma del tiempo de cómputo paralelo $T_c(p)$ y el **tiempo de sobrecarga** (*overhead*) $T_o(p)$ introducido por el coste de paralelizar el programa para $p$ procesadores. Esta sobrecarga se debe al tiempo sumado por la sincronización y comunicación de las hebras, así como su creación y finalización, los cómputos añadidos necesariamente en la versión paralela y el déficit de equilibrado de la versión paralela.

$$S(p) = \frac{T_s}{T_p(p)} \qquad T_p(p) = T_c(p) + T_o(p)$$

Distinguimos cuatro tipos de escalabilidad de un programa paralelo:

#### Lineal

Se da cuando todos los bloques son paralelizables ($T_s = 0$) y la sobrecarga es nula, de forma que $S(p) = p$. Si tenemos un bloque secuencial que ocupa una fracción $s \leq 1$ no paralelizable en el código, debemos tener en cuenta que la escalabilidad se ve reducida por la presencia del mismo:

$$S(p) = \lim_{p\to\infty} \frac{1}{s + \frac{1-s}{p}} = \frac{1}{s}$$

Existe el caso de que la ganancia sea **superlineal**, de forma que $S(p) > p$.

#### Limitada en el aprovechamiento del grado de paralelismo

Se produce cuando el sistema únicamente puede aprovechar hasta un número $n$ de prestaciones, de forma que la ganancia queda maximada:

$$S(p) = \lim_{p\to n} \frac{1}{s + \frac{1-s}{p}} = \frac{1}{s + \frac{1-s}{n}}$$

#### Reducida debido a sobrecarga

[2]En MPI la asignación estática se explicita mediante los índices de los procesos a los que se envían los mensajes, mientras que en OpenMP es el compilador quien implicita la asignación.

Se produce cuando la sobrecarga incrementa linealmente con $p$, de forma que en algún punto comienza a afectar negativamente al tiempo de ejecución paralelo:

$$S(p) = \lim_{p \to \infty} \frac{1}{s + \frac{1-s}{p} + \frac{T_o(p)}{T_s}} = 0$$

Podemos calcular el número máximo de prestaciones igualando tiempo de cálculo $T_c(p) = O\left(\frac{1}{p}\right)$ al tiempo de sobrecarga $T_o(p) = O(p)$ añadido para una función $O$ de variacón del tiempo de ejecución.

## 2.3.2: Ley de Amdahl

Como ya vimos en **??**, la ganancia está limitada por una fracción del código que no podemos paralelizar. Como consecuencia lógica de esto, cuanto más pequeña es esta fracción, mayor es la ganancia en prestaciones al paralelizar el programa. Sin embargo, esta ley no tiene en cuenta la sobrecarga introducida en la paralelización del programa. Dado que esta sobrecarga es inversamente proporcional al tamaño del problema ejecutado por un determinado número de núcleos, podemos incrementar la ganancia aumentando el número de cómputos.

## 2.3.3: Ganancia escalable (ley de Gustafson)

De forma contraria a la ley de Amdahl, la ley de Gustafson plantea que cualquier programa suficientemente grande puede ser eficientemente paralelizado. Mientras que Amdahl no escala la disponibilidad del poder de cómputo junto con el aumento del número de máquinas, Gustafson propone que, con la mayor potencia de cómputo disponible, se podrán resolver problemas mayores en el mismo tiempo debido a la reducción de la parte secuencial no paralelizable. Para un programa con $p$ prestaciones y una porción secuencial $s$, Gustafson define la escalabilidad de la siguiente manera:

$$S(p) = p - s \cdot (p - 1)$$

# Tema 3:
# Arquitecturas con paralelismo a nivel de hebra (*TLP*)

# Tema 4:
# Arquitecturas con paralelismo a nivel de instrucción (*ILP*)