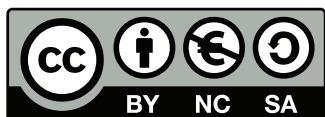


Arquitectura de Computadores

Paula Villanueva Núñez
Doble Grado de Informática y Matemáticas
Universidad de Granada



Este libro se distribuye bajo una licencia CC BY-NC-SA 4.0.

Eres libre de distribuir y adaptar el material siempre que reconozcas a los autores originales del documento, no lo utilices para fines comerciales y lo distribuyas bajo la misma licencia.

creativecommons.org/licenses/by-nc-sa/4.0/

Arquitectura de Computadores

Paula Villanueva Núñez
Doble Grado de Informática y Matemáticas
Universidad de Granada

Índice

1. Tema 1. Arquitecturas paralelas: clasificación y prestaciones	3
1.1. Lección 1. Clasificación del paralelismo implícito en una aplicación	3
1.1.1. 1.1 Objetivos	3
1.1.2. 1.2 Criterios de clasificaciones del paralelismo implícito en una aplicación.	3
1.1.3. 1.3 Dependencias de datos.	4
1.1.4. 1.4 Paralelismo implícito (nivel de detección), explícito y arquitecturas paralelas. . .	5
1.1.5. 1.5 Detección, utilización, implementación y extracción del paralelismo.	6
1.2. Lección 2. Clasificación de arquitecturas paralelas.	7
1.2.1. 2.1 Objetivos	7
1.2.2. 2.2 Computación paralela y computación distribuida.	8
1.2.3. 2.3 Clasificaciones de arquitecturas y sistemas paralelos.	8
1.2.4. 2.4 Nota histórica	22
1.3. Lección 3. Evaluación de prestaciones.	23
1.3.1. 3.1 Objetivos.	23
1.3.2. 3.2 Medidas usuales para evaluar prestaciones.	23
1.3.3. 3.3 Conjunto de programas de prueba (Benchmark).	27
1.3.4. 3.3.1 LINPACK.	28
1.3.5. 3.4 Ganancia en prestaciones.	29
2. Tema 2. Programación paralela	34
2.1. Lección 4. Herramientas, estilos y estructuras en programación paralela.	34
2.1.1. Objetivos.	34
2.1.2. 4.1 Problemas que plantea la programación paralela al programador. Punto de partida.	34
2.1.3. 4.2 Herramientas para obtener código paralelo.	36
2.1.4. 4.3 Estilos/paradigmas de programación paralela.	42
2.1.5. 4.4 Estructuras típicas de códigos paralelos.	43
2.2. Lección 5. Proceso de paralelización.	47
2.2.1. 5.1 Objetivos.	47

2.2.2. 5.2 Proceso de paralelización.	47
2.3. Lección 6. Evaluación de prestaciones en procesamiento paralelo.	56
2.3.1. Objetivos.	56
2.3.2. 6.1 Ganancia de prestaciones y escalabilidad.	56
2.3.3. 6.2 Ley de Amdahl.	59
2.3.4. 6.3 Ganancia escalable.	60
3. Bibliografía	62

1 Tema 1. Arquitecturas paralelas: clasificación y prestaciones

1.1 Lección 1. Clasificación del parallelismo implícito en una aplicación

1.1.1 1.1 Objetivos

- Clasificaciones del parallelismo implícito en una aplicación. Distinguir entre parallelismo de tareas y de datos.
- Distinguir entre dependencias RAW, WAW, WAR.
- Distinguir entre thread y proceso.
- Relacionar el parallelismo implícito en una aplicación con el nivel en el que se hace explícito para que se pueda utilizar (instrucción, thread, proceso) y con las arquitecturas paralelas que lo aprovechan.

1.1.2 1.2 Criterios de clasificaciones del parallelismo implícito en una aplicación.

- **Parallelismo funcional.**

- **Nivel de funciones.** Las funciones llamadas en un programa se pueden ejecutar en paralelo, siempre que no haya entre ellas dependencias inevitables, como dependencias de datos verdaderas (lectura después de escritura).
- **Nivel de bucle (bloques).** Se pueden ejecutar en paralelo las iteraciones de un bucle, siempre que se eliminen los problemas derivados de dependencias verdaderas. Para detectar dependencias habrá que analizar las entradas y las salidas de las iteraciones del bucle.
- **Nivel de operaciones.** Las operaciones independientes se pueden ejecutar en paralelo. En los procesadores de propósito específico y en los de propósito general podemos encontrar instrucciones compuestas de varias operaciones que se aplican en secuencia al mismo flujo de datos de entrada. Se pueden usar instrucciones compuestas, que van a evitar las penalizaciones por dependencias verdaderas.

- **Parallelismo de datos** (*data parallelism o DLP-Data Level Par.*). Se encuentra implícito en las operaciones con estructuras de datos (vectores y matrices). Se puede extraer de la representación matemática de la aplicación. Las operaciones vectoriales y matriciales engloban operaciones que se pueden realizar en paralelo. Por lo que el parallelismo de datos está relacionado con el parallelismo a nivel de bucle.
- **Parallelismo de tareas** (*task parallelism o TLP-Task Level Par.*). Se encuentra extrayendo la estructura lógica de funciones de una aplicación. Los bloques son funciones y se puede encontrar parallelismo entre las funciones.
- **Granularidad.** El grano más pequeño (*grano fino*) se asocia al parallelismo entre operaciones o instrucciones, el *grano medio* se asocia a los bloques funcionales lógicos y el *grano grueso* se asocia al parallelismo entre programas.

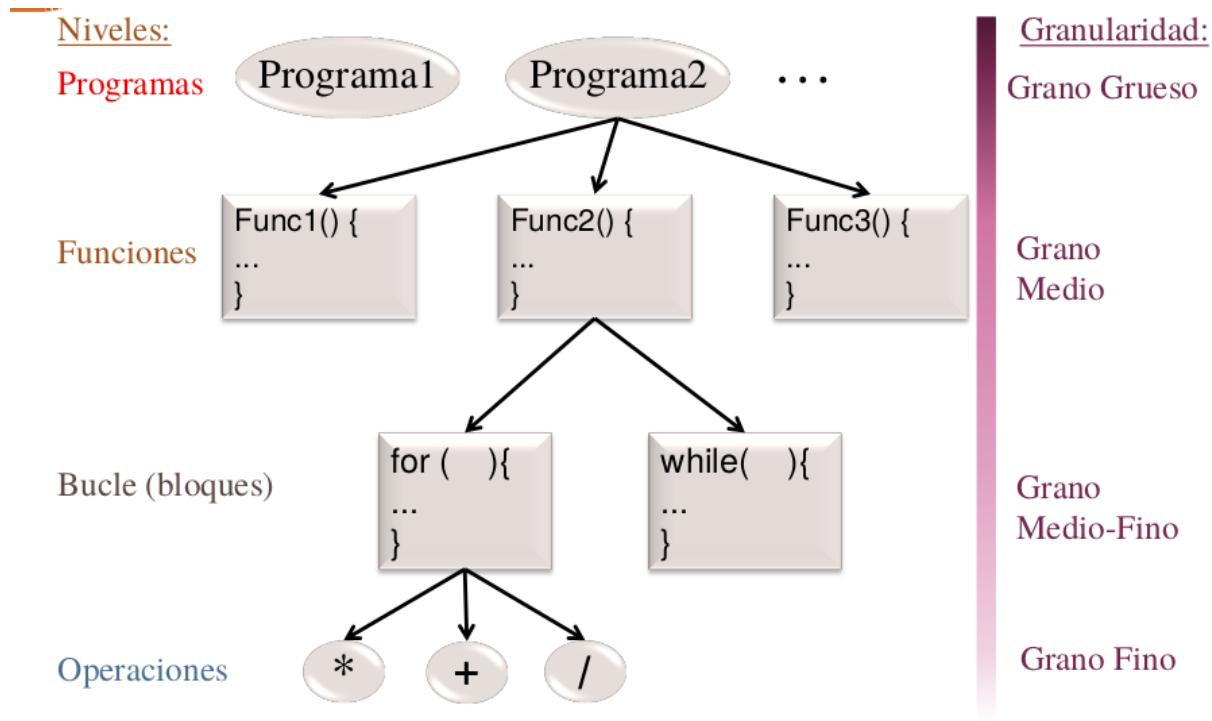


Figura 1:

1.1.3 1.3 Dependencias de datos.

Para que un bloque de código B_2 presente dependencia de datos con respecto a B_1 , deben hacer referencia a una misma posición de memoria M (variable) y B_1 aparece en la secuencia de código antes que B_2 .

Tipos de dependencias de datos (de B_2 respecto a B_1):

- **RAW** (*Read After Write*) o dependencia verdadera.
- **WAW** (*Write After Write*) o dependencia de salida.
- **WAR** (*Write After Read*) o antidependencia.

```

1 ...
2 a = b + c
3 ... //código que no usa a
4 d = a + c
5 ...

```

```

1 ...
2 a = b + c
3 ... //se lee a
4 a = d + e
5 ... //se lee a

```

```

1 ...
2 b = a + 1

```

```

3 ...
4 a = d + e
5 ... //se lee a

```

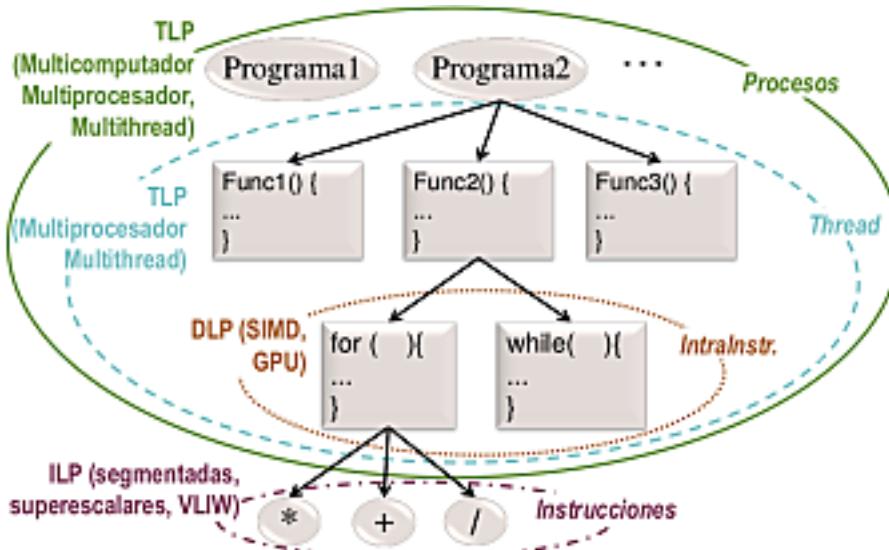
1.1.4 1.4 Parallelismo implícito (nivel de detección), explícito y arquitecturas paralelas.

El paralelismo entre **programas** se utiliza a nivel de procesos. Cuando se ejecuta un programa, se crea el proceso asociado al programa.

El paralelismo entre **funciones** se puede extraer para utilizarlo a nivel de procesos o de hebras.

El paralelismo dentro de un **bucle** se puede extraer a nivel de procesos o de hebras. Se puede aumentar la granularidad asociando un mayor número de iteraciones del ciclo a cada unidad a ejecutar en paralelo. Se puede hacer explícito dentro de una instrucción vectorial para que sea aprovechado por arquitecturas SIMD o vectoriales.

El paralelismo entre **operaciones** se puede aprovechar en arquitecturas con paralelismo a nivel de instrucción (ILP) ejecutando en paralelo las instrucciones asociadas a estas operaciones independientes.



1.4.1 Nivel de paralelismo explícito.

1.4.1.1 Unidades en ejecución en un computador.

- **Instrucciones.** La unidad de control de un core o procesador gestiona la ejecución de instrucciones por la unidad de procesamiento.
- **Thread o light process.** Es la menor unidad de ejecución que gestiona el SO. Menor secuencia de instrucciones que se pueden ejecutar en paralelo o concurrentemente.
- **Proceso o process.** Mayor unidad de ejecución que gestiona el SO. Un proceso consta de uno o varios thread.

1.4.1.2 Threads versus procesos.

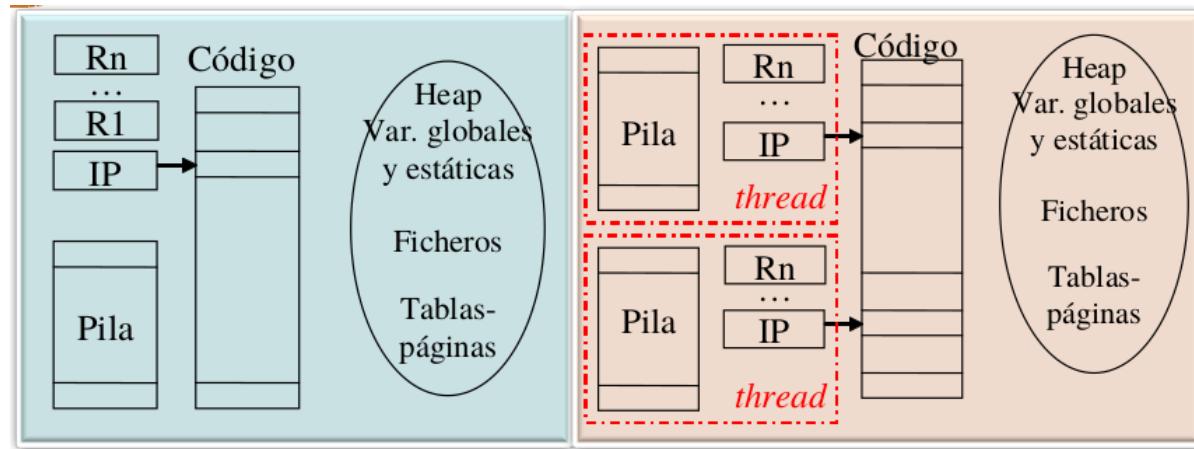
El hardware gestiona la ejecución de las instrucciones. A nivel superior, el SO se encarga de gestionar la ejecución de unidades de mayor granularidad, procesos y hebras. Cada proceso en ejecución tiene su propia asignación de memoria. Los SO **multihebra** permiten que un proceso se componga de una o varias hebras (hilos). Una **hebra** tiene su propia pila y contenido de registros, entre ellos el puntero de pila y el IP (Puntero de Instrucciones) que almacena la dirección de la siguiente instrucción a ejecutar de la hebra, pero comparte el código, las variables globales y otros recursos con las hebras del mismo proceso. Por lo que las hebras se pueden crear y destruir en menor tiempo que los procesos, y la comunicación (se usa la memoria que comparten), sincronización y conmutación entre hebras de un proceso es más rápida que entre procesos. Luego las hebras tienen menor granularidad que los procesos.

Un **proceso** comprende el código del programa y todo lo que hace falta para su ejecución:

- Datos en pila, segmentos (variables globales y estáticas) y en heap (BP1).
- Contenido de los registros.
- Tabla de páginas.
- Tabla de ficheros abiertos.

Para comunicar procesos hay que usar llamadas al SO.

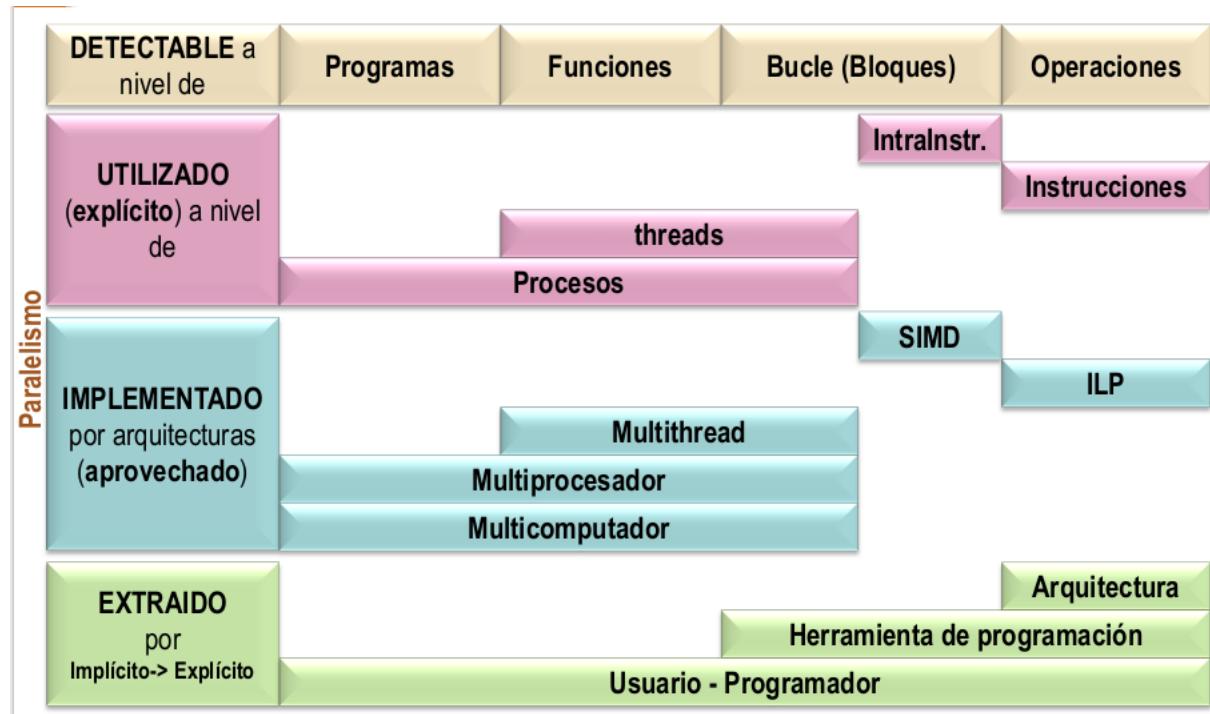
El paralelismo implícito en el código de una aplicación se puede hacer explícito a nivel de instrucciones, de hebras o de procesos.



1.1.5 1.5 Detección, utilización, implementación y extracción del paralelismo.

En los procesadores ILP superescalares o segmentados la arquitectura extrae paralelismo. Para ello, eliminan dependencias de datos falsas entre instrucciones y evitan problemas debidos a dependencias de datos, de control y de recursos. La arquitectura extrae paralelismo implícito en las entradas en tiempo de ejecución (dinámicamente). El grado de paralelismo de las instrucciones aprovechado se puede incrementar con ayuda del compilador y del programador. Podemos definir el grado de paralelismo de un conjunto de entradas a un sistema como el máximo número de entradas del conjunto que se puede ejecutar en paralelo. Para los procesadores las entradas son instrucciones. En las arquitecturas ILP VLIW el paralelismo que se va a aprovechar está ya explícito en las entradas. Las instrucciones que se van a

ejecutar en paralelo se captan juntas de memoria. El análisis de dependencias entre instrucciones en este caso es estático.



Hay compiladores que extraen el paralelismo de datos implícito a nivel de bucle. Algunos compiladores lo hacen explícito a nivel de hebra, y otros dentro de una instrucción para que se pueda aprovechar en arquitecturas SIMD o vectoriales. El usuario, como programador, puede extraer el paralelismo implícito en un bucle o entre funciones definiendo hebras y/o procesos. La distribución de las tareas independientes entre hebras o entre procesos dependerán de

- la granularidad de las unidades de código independientes,
- la posibilidad que ofrezca la herramienta para programación paralela disponible de definir hebras o procesos,
- la arquitectura disponible para aprovechar el paralelismo,
- el SO disponible.

Por último, los usuarios del sistema al ejecutar programas están creando procesos que se pueden ejecutar en el sistema concurrentemente o en paralelo.

1.2 Lección 2. Clasificación de arquitecturas paralelas.

1.2.1 2.1 Objetivos

- Distinguir entre procesamiento o computación paralela y distribuida.
- Clasificar los computadores según segmento del mercado.
- Distinguir entre las diferentes clases de arquitecturas de la clasificación de Flynn.
- Diferenciar un multiprocesador de un multicomputador.

- Distinguir entre NUMA y SMP.
- Distinguir entre arquitecturas DLP, ILP, TLP.
- Distinguir entre arquitecturas TLP con una instancia de SO y TLP con varias instancias de SO.

1.2.2 2.2 Computación paralela y computación distribuida.

- **Computación paralela.** Estudia los aspectos hardware y software relacionados con el desarrollo y ejecución de aplicaciones en un sistema de cómputo compuesto por múltiples cores/procesadores/computadores que es visto externamente como una unidad autónoma (multicores, multiprocesadores, multicomputadores, cluster).
- **Computación distribuida.** Estudia los aspectos hardware y software relacionados con el desarrollo y ejecución de aplicaciones en un sistema distribuido; es decir, en una colección de recursos autónomos (PC, servidores -de datos, software, . . . -, supercomputadores. . .) situados en distintas localizaciones físicas.
 - **Computación distribuida baja escala.** Estudia os aspectos relacionados con el desarrollo y ejecución de aplicaciones en una colección de recursos autónomos de un dominio administrativo situados en distintas localizaciones físicas conectados a través de infraestructura de red local.
 - **Computación distribuida gran escala.**
 - **Computación grid.** Estudia los aspectos relacionados con el desarrollo y ejecución de aplicaciones en una colección de recursos autónomos de múltiples dominios administrativos geográficamente distribuidos conectados con infraestructura de telecomunicaciones.
 - **Computación cloud.** Comprende los aspectos relacionados con el desarrollo y ejecución de aplicaciones en un sistema cloud. El sistema cloud ofrece servicios de infraestructura, plataforma y/o software, por los que se paga cuando se necesitan (pay-per-use) y a los que se accede típicamente a través de una interfaz (web) de auto-servicio. El sistema cloud consta de recursos virtuales que son una abstracción de los recursos físicos, parecen ilimitados en número y capacidad y son reclutados/liberados de forma inmediata sin interacción con el proveedor, soportan el acceso de múltiples clientes (multitenant) y están conectados con métodos estándar independientes de la plataforma de acceso.

1.2.3 2.3 Clasificaciones de arquitecturas y sistemas paralelos.

2.3.1 Criterios de clasificación de computadores

- Comercial. Segmento del mercado: embebidos, servidores gama baja. . .

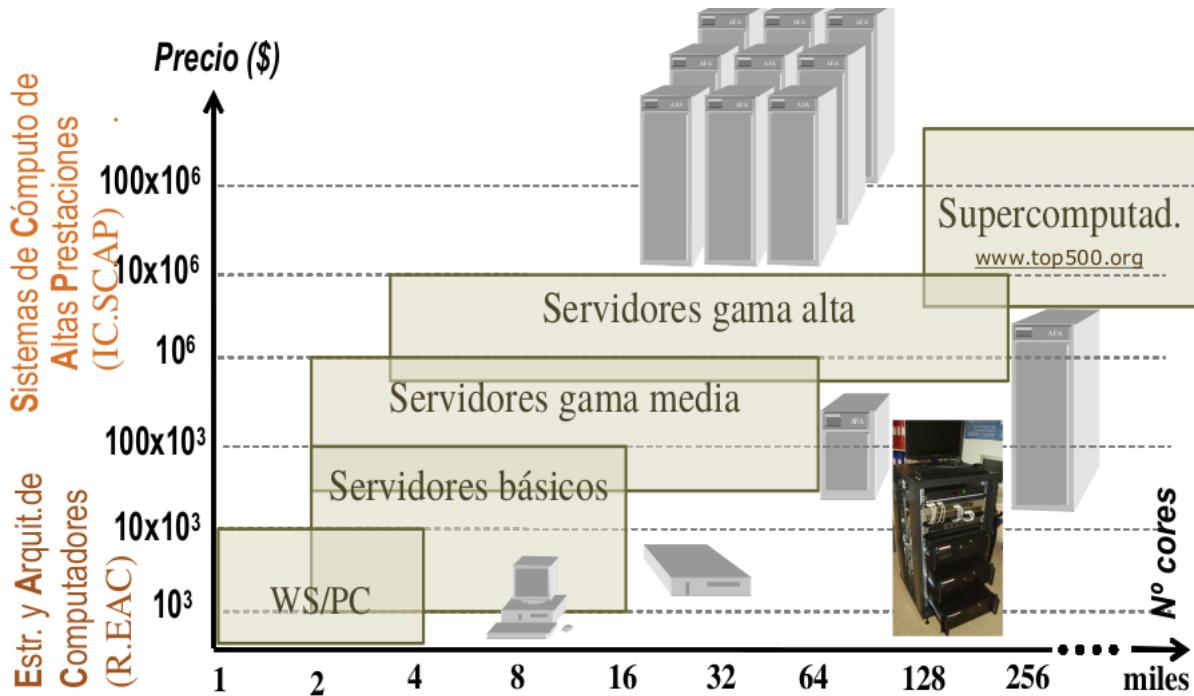


- Educación, investigación (también usados por fabricantes y vendedores):
 - Flujos de control y flujos de datos: clasificación de Flynn.
 - Sistemas de memoria.
 - Flujos de control (propuesta de clasificación de arquitecturas con múltiples flujos de control).
 - Nivel del paralelismo aprovechado (propuesta de clasificación).

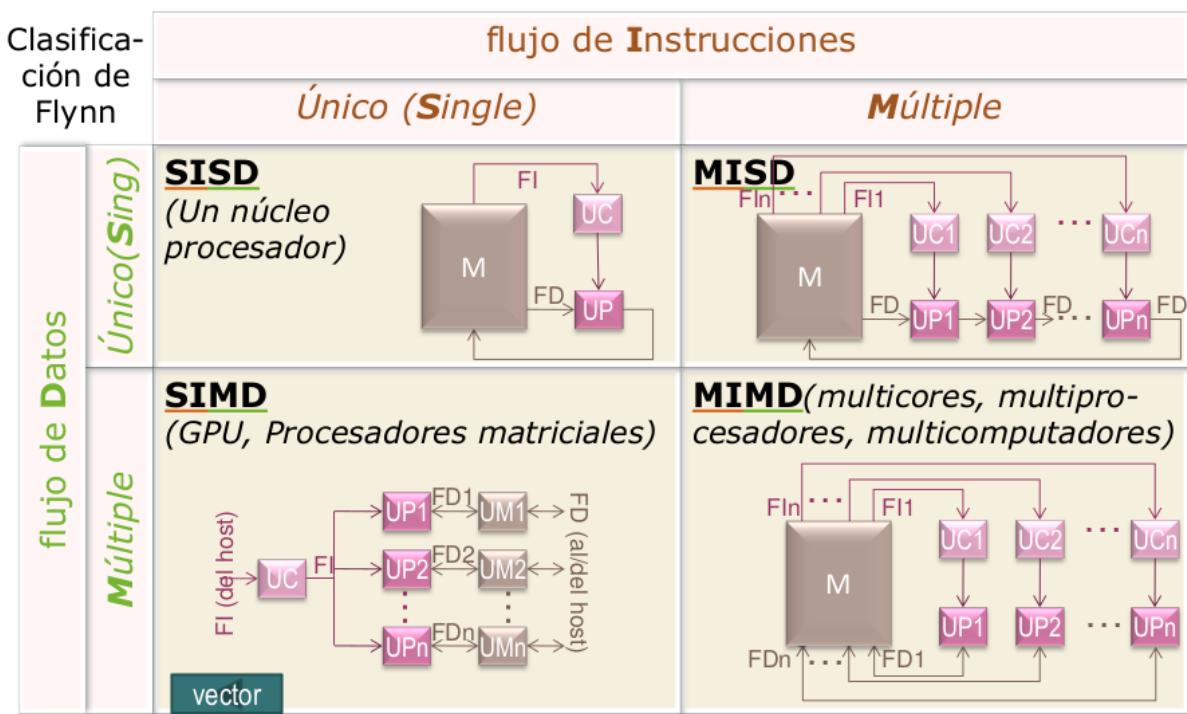
2.3.2 Clasificación de computadores según segmento

- **Externo** (*desktop, laptop, server, cluster...*) - R.EAC, IC.SCAP. Para todo tipo de aplicaciones:
 - Oficina, entretenimiento...
 - Procesamiento de transacciones o OLTP, sistemas de soporte de decisiones o DSS, e-commerce...
 - Científicas (medicina, biología, predicción del tiempo...) y animación (películas animadas, efectos especiales...).
- **Empotrado** (oculto) - IC.SCAE. Aplicaciones de propósito específico (videojuegos, teléfonos, coches, electrodomésticos...). Las restricciones típicas son: consumo de potencia, precio, tamaño reducido, tiempo real...

2.3.3 Clasificación de computadores externos según segmento del mercado



2.3.4 Clasificación de Flynn de arquitecturas (flujo instrucciones / flujo de datos)

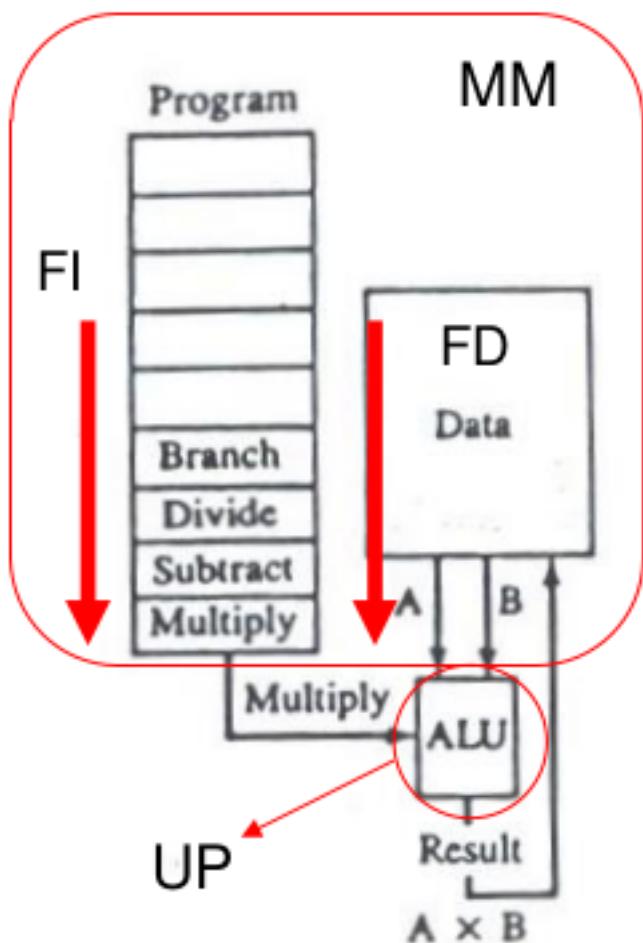


- **Computadores SISD.** Un único flujo de instrucciones (SI) procesa operandos y genera resultados, definiendo un único flujo de datos (SD).

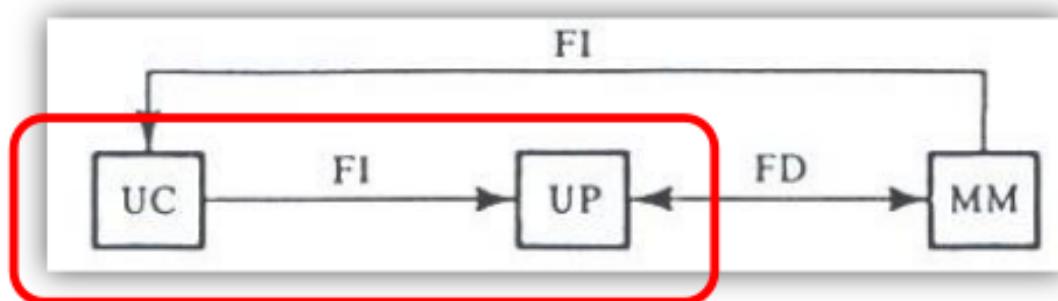
Corresponde a los computadores uni-procesador, ya que existe una única unidad de control (UC) que recibe las instrucciones de memoria, las decodifica y genera los códigos que definen la operación correspondiente a cada instrucción que debe realizar la unidad de procesamiento

(UP) de datos.

Descripción Funcional



Descripción Estructural

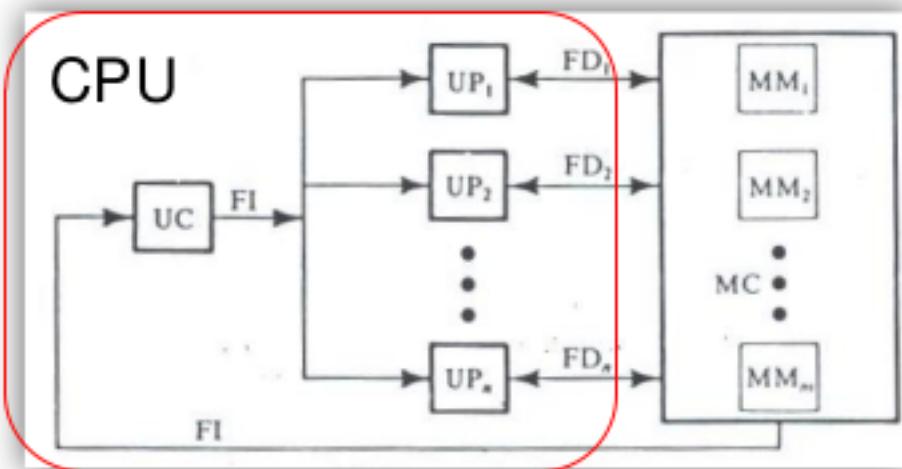


CPU, núcleo, procesador

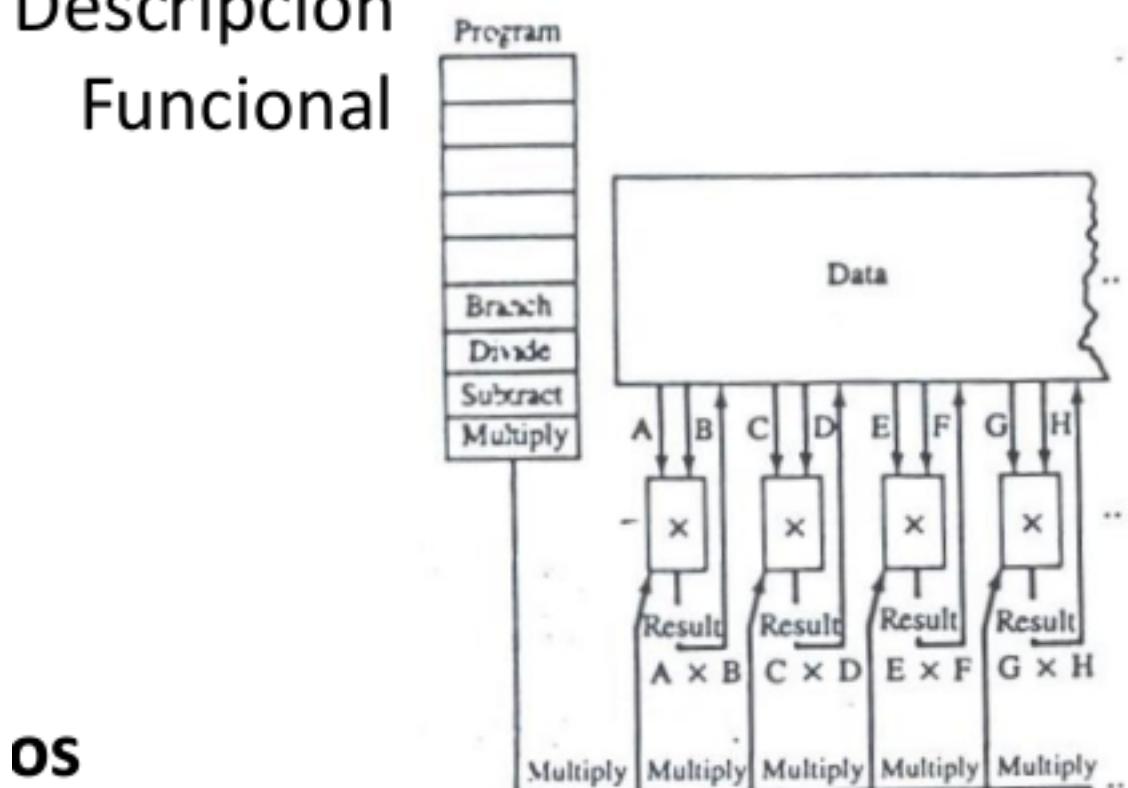
- **Computadores SIMD.** Un único flujo de instrucciones (SI) procesa operandos y genera resultados, definiendo varios flujos de datos (MD), dado que cada instrucción codifica realmente varias operaciones iguales, cada una actuando sobre operadores distintos.

Los códigos que generan la única unidad de control a partir de cada instrucción actúan sobre varias unidades de procesamiento distintas (UP_i). Por lo que se pueden realizar varias operaciones similares simultáneas con operandos distintos. Aprovechan paralelismo de datos.

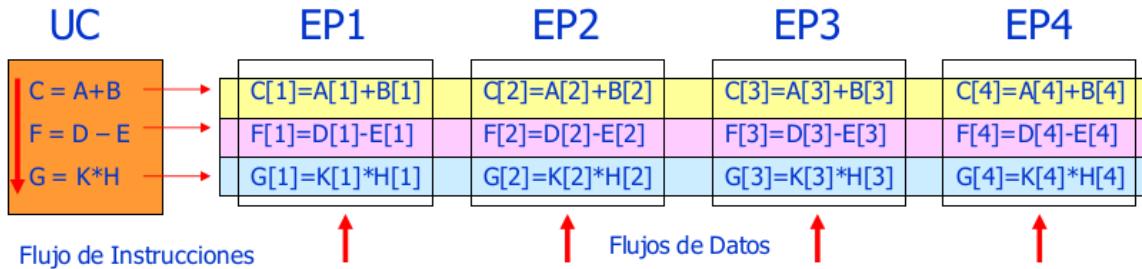
Descripción Estructural



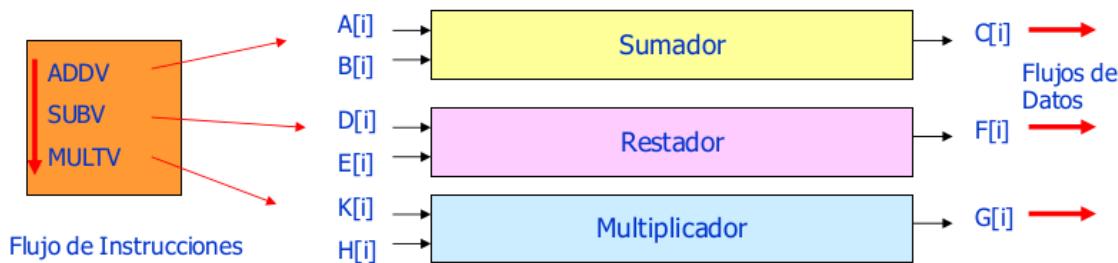
Descripción Funcional



Procesador Matricial

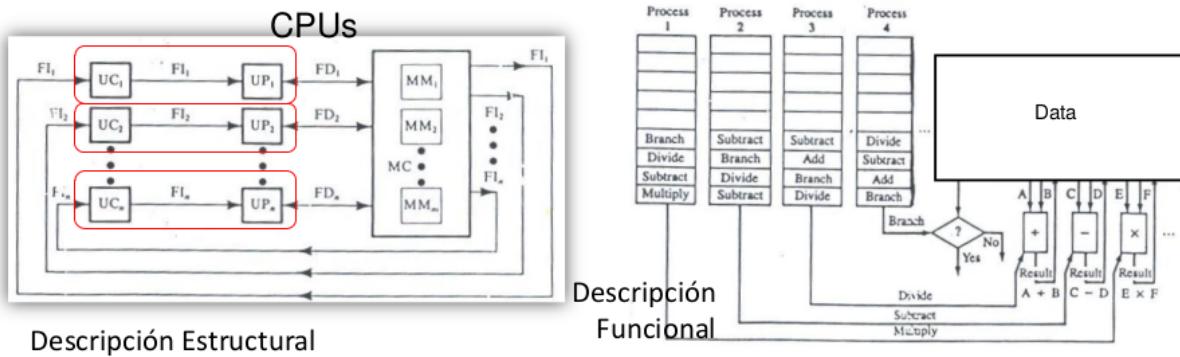


Procesador Vectorial



- **Computadores MIMD.** El computador ejecuta varias secuencias o flujos distintos de instrucciones (MI) y cada uno de ellos procesa operandos y genera resultados definiendo un único flujo de instrucciones, de forma que existen varios flujos de datos (MD) uno por cada flujo de instrucciones.

Corresponde con multinúcleos, multiprocesadores y multicamputadores. Puede aprovechar paralelismo funcional. Existen varias unidades de control que decodifican las instrucciones correspondientes a distintos programas. Cada uno de estos programas procesa conjuntos de datos diferentes, que definen distintos flujos de datos.



Corresponde con Multinúcleos, Multiprocesadores y Multicomputadores: Puede aprovechar, además, **paralelismo funcional**

```
for i:=1 to 4 do
begin
  C[i]:=A[i]+B[i];
end;
```

Proc 1

```
for i:=1 to 4 do
begin
  F[i]:=D[i]-E[i];
end;
```

Proc 2

```
for i:=1 to 4 do
begin
  G[i]:=K[i]*H[i];
end;
```

Proc 3

- **Computadores MISD.** Se ejecutan varios flujos distintos de instrucciones (MI) aunque todos actúan sobre el mismo flujo de datos (SD).

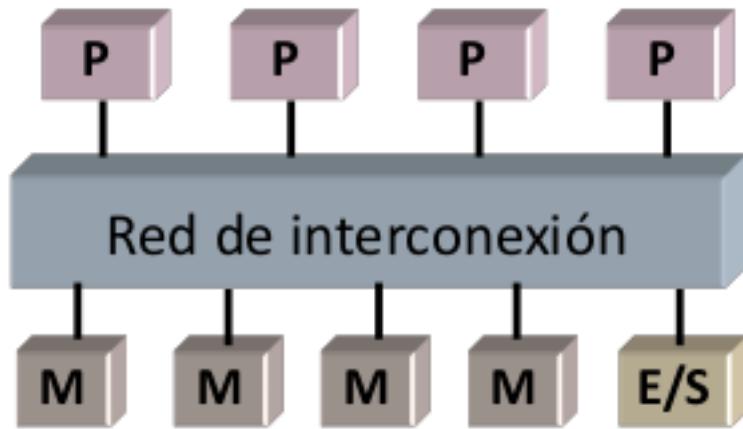
No existen computadores que funcionen según este modelo. Se puede simular en un código este modelo para aplicaciones que procesan una secuencia o flujo de datos.

2.3.5 Sistemas de memoria

2.3.5.1 Clasificación de computadores paralelos MIMD según el sistema de memoria

Los sistemas multiprocesadores se han clasificado atendiendo a la organización del sistema de memoria:

- **Sistemas con memoria compartida (SM) o multiprocesadores.** Son sistemas en los que todos los procesadores comparten el mismo espacio de direcciones. El programador no necesita conocer dónde están almacenados los datos.
- **Sistemas con memoria distribuida (DM) o multicomputadores.** Son sistemas en los que cada procesador tiene su propio espacio de direcciones particular. El programador necesita conocer dónde están almacenados los datos.



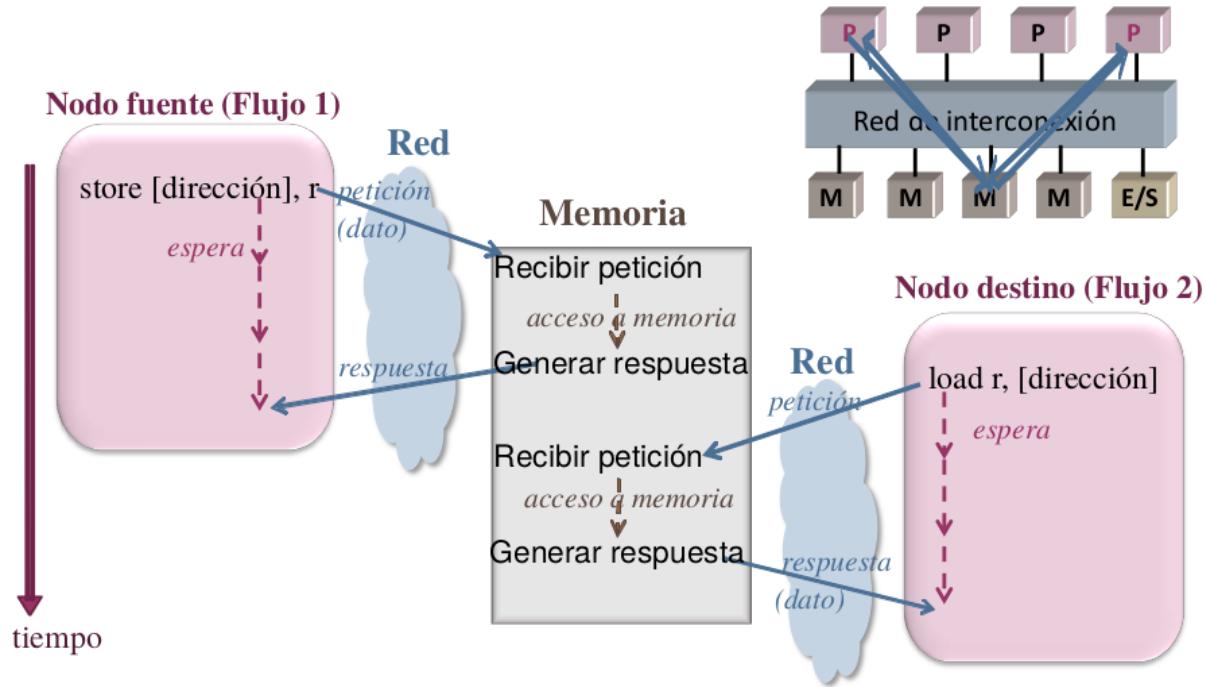
2.3.5.2 Comparativa SMP (Symmetric MultiProcessor) y multicomputadores.

- **Multiprocesador con memoria centralizada (SMP).** Es un multiprocesador en el que el tiempo de acceso de los procesadores a memoria es igual sea cual sea la posición de memoria a la que acceden, es una estructura simétrica.
 - Mayor latencia.
 - Poco escalable.
 - La comunicación es implícita mediante variables compartidas.
 - Los datos no están duplicados en memoria principal.
 - Necesita implementar primitivas de sincronización.
 - La distribución de código y datos entre procesadores no es necesaria.
 - La programación es más sencilla.
- **Multicomputador.**
 - Menor latencia.
 - Más escalable.
 - La comunicación es explícita mediante software para paso de mensajes (*send/receive*).
 - Los datos están duplicados en memoria principal y se copian datos entre módulos de memoria de diferentes procesadores.
 - La sincronización se hace mediante software de comunicación.
 - La distribución de código y datos entre procesadores es necesaria y se necesitan herramientas de programación más sofisticadas.
 - La programación es más difícil.

2.3.5.3 Comunicación uno-a-uno en un multiprocesador.

Los diferentes procesadores que ejecutan una aplicación pueden requerir sincronizarse en algún momento. Por ejemplo, si el procesador A utiliza un dato que produce el procesador B, A deberá esperar a que B lo genere. En la siguiente imagen vemos la transferencia de datos en un multiprocesador. El proceso que ejecuta la instrucción de carga espera hasta recibir el contenido de la dirección. El proceso que ejecuta la instrucción de almacenamiento puede esperar a que termine para garantizar que se mantiene un orden

en los accesos a memoria. Obsérvese que para que la transferencia de datos e realice de forma efectiva habría que sincronizar los procesos fuente y destino.



Secuencial	Paralelo	
<code>...</code> <code>A=valor;</code> <code>...</code> <code>copia=A;</code> <code>...</code>	<u>F1</u> <code>...</code> <code>A=valor;</code> <code>...</code>	<u>F2</u> <code>...</code> <code>copia=A;</code> <code>...</code>

F1 es el flujo de control productor del dato (*envía el dato*)

F2 es el flujo de control consumidor del dato (*recibe el dato*)

Paralelo multiproc. ($K=0$)

F1

...

A=valor;
K=1;

...

F2

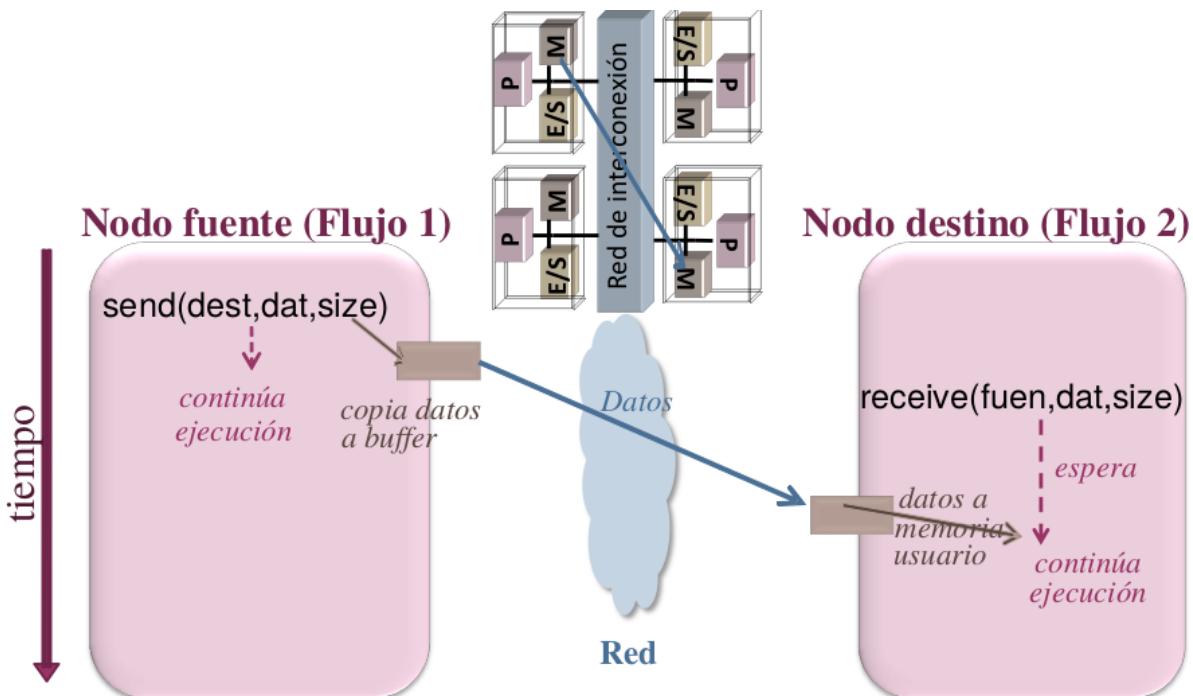
...

while (K==0) { };
copia=A;

...

Se debe garantizar que el flujo de control **consumidor** del dato lea la variable compartida (A) cuando el **productor** haya escrito en la variable el dato.

En multicomputadores se aprovechan los mecanismos de comunicación para implementar sincronización. Con una función de recepción bloqueante, es decir, que deja al proceso que la ejecuta detenido hasta que se reciba el dato, se puede implementar sincronización. En la siguiente figura podemos ver la transferencia asíncrona (con función *receive* bloqueante) de datos en un multicomputador. EN transferencia asíncrona se requiere almacenamiento intermedio para evitar esperas. El proceso fuente continúa la ejecución en cuanto los datos se copien en un *buffer*. El destino espera en el *receive* bloqueante a que lleguen los datos, en caso de que estos no hayan llegado aún.



Secuencial	Paralelo	
	<u>F1</u>	<u>F2</u>
... A=valor; ... copia=A; A=valor; copia=A; ...

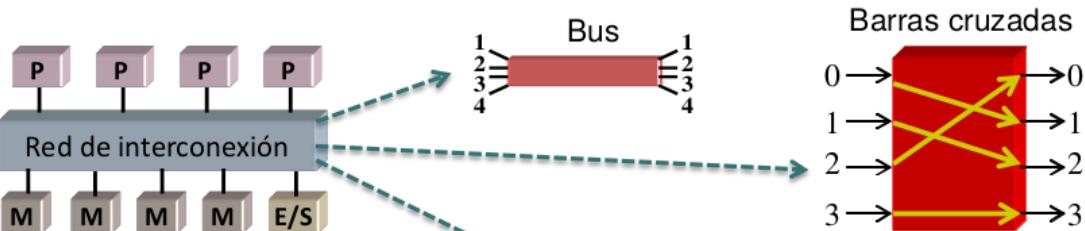
F1 es el flujo de control productor del dato (*envía el dato*)
F2 es el flujo de control consumidor del dato (*recibe el dato*)

Paralelo multicamputador (size = 4 byte)	
<u>F1</u>	<u>F2</u>
... send(F2, valor, 4); receive(F1,copia,4); ...

2.3.5.4 Incremento de escalabilidad en multiprocesadores y red de interconexión.

Como los SMP tienen escasa escalabilidad, se ha intentado incrementar la escalabilidad en multiprocesadores:

- Aumentar caché del procesador.
- Usar redes de menor latencia y mayor ancho de banda que un bus (jerarquía de buses, barras cruzadas, multietapa).
- Distribuir físicamente los módulos de memoria entre los procesadores (pero se sigue compartiendo espacio de direcciones).



- Incremento escalabilidad multiprocesadores:
- Aumentar cache del procesador
 - Usar redes de menor latencia y mayor ancho de banda que un bus (jerarquía de buses, barras cruzadas, multietapa)
 - Distribuir físicamente los módulos de memoria entre los procesadores (pero se sigue compartiendo espacio de direcciones)

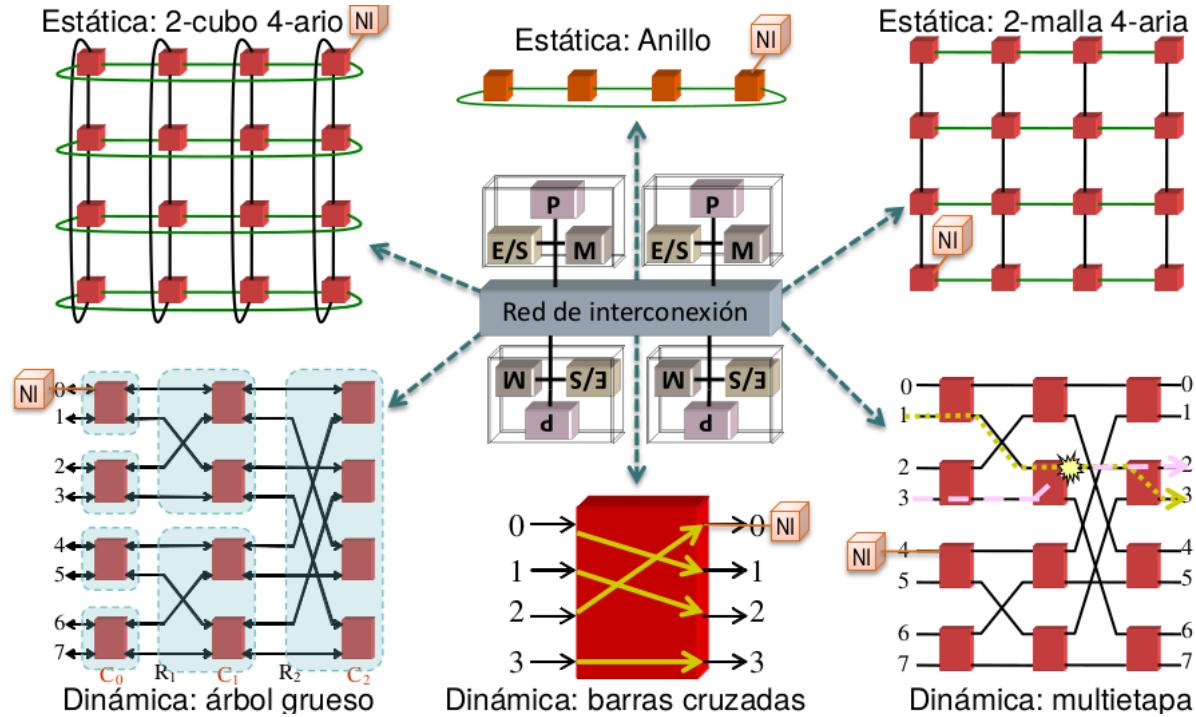
2.3.5.5 Clasificación completa de computadores según el sistema de memoria.

■ Multiprocesadores.

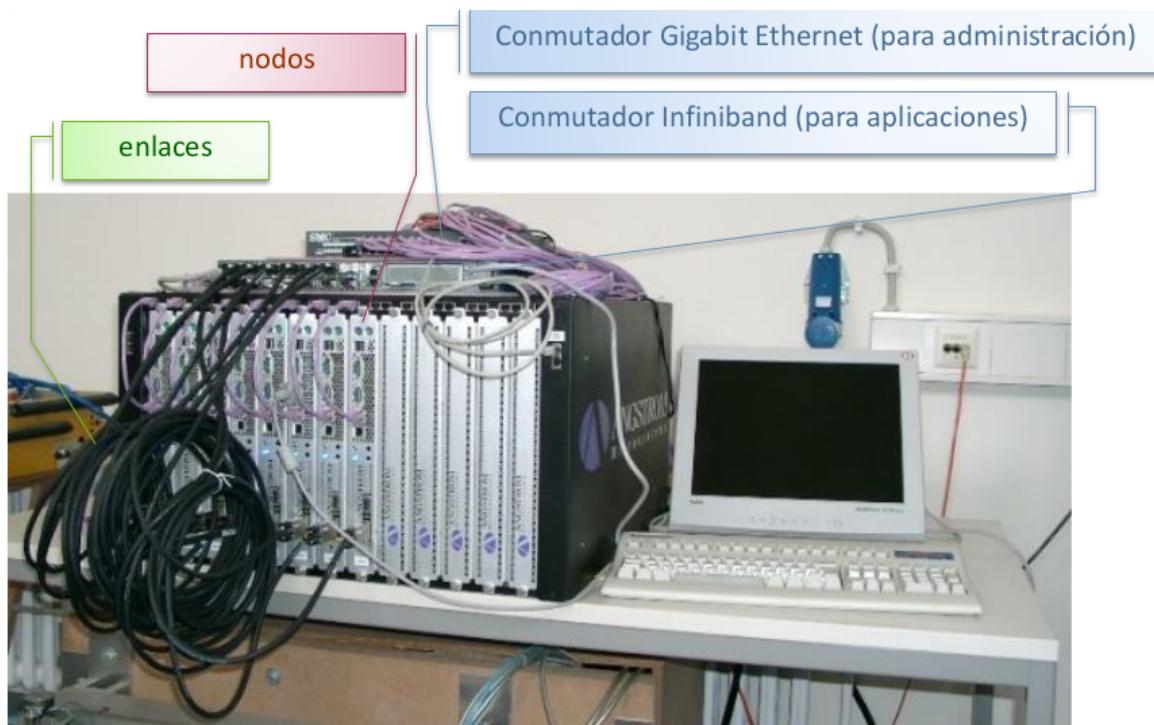
- UMA (*Uniform Memory Access*).
 - SMP.
- NUMA (*Non-Uniform Memory Access*).
 - NUMA. Arquitecturas con acceso a memoria no uniforme sin coherencia de caché entre nodos. No incorporan hardware para evitar problemas por incoherencias entre cachés de distintos nodos. Esto hace que los datos modificables compartidos no se puedan trasladar a caché de nodos remotos; hay que acceder a ellos individualmente a través de la red. Se puede hacer más tolerable la latencia utilizando precaptación (*prefetching*) de memoria y procesamiento multihebra.
 - CC-NUMA. Arquitecturas con acceso a memoria no uniforme y con caché coherente. Tienen hardware para mantener coherencia entre cachés de distintos nodos, que se encarga de las transferencias de datos compartidos entre nodos. El hardware para mantenimiento de coherencia supone un coste añadido e introduce un retardo que hace que estos sistemas escalen en menor grado que un NUMA.
 - COMA. Arquitecturas con acceso a memoria solo caché. La memoria local de los procesadores se gestiona como caché. El sistema de mantenimiento se encarga de llevar dinámicamente el código y los datos a los nodos donde se necesiten.

Multi-computadores Memoria no compartida	NORMA No Remote Memory Access	ej. cluster, red de computadores	Memoria físicamente distribuida	+ + Nivel de empaquetamiento y conexión
Multi-procesadores Memoria compartida Un único espacio de direcciones	NUMA Non-Uniform Memory Access	NUMA		
		CC-NUMA		
		COMA		
	UMA Uniform Memory Access	SMP Symmetric MultiProcessor	Memoria físicamente centralizada	- - Escalabilidad

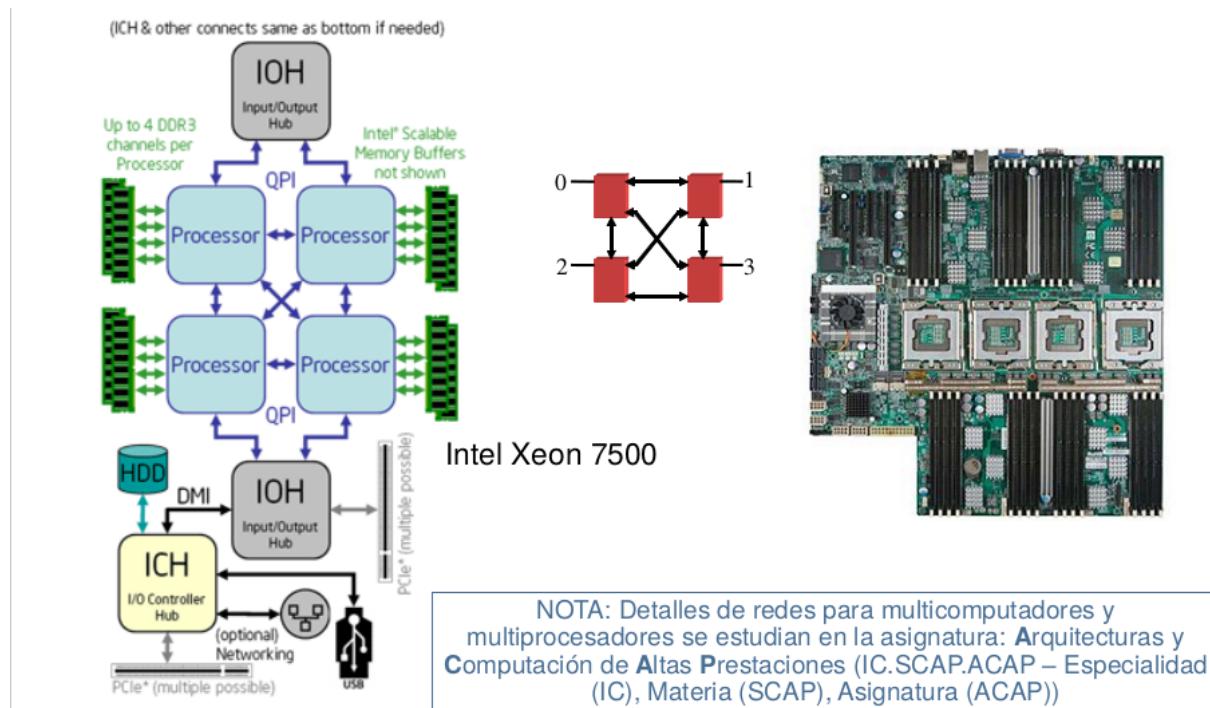
2.3.5.6 Red en sistemas con memoria físicamente distribuida (NI: Network Interface).



Ejemplo: Red (con conmutador o switch) de barras cruzadas.



Ejemplo: Placa CC-NUMA con red estática



2.3.6 Flujos de control (propuesta de clasificación de arquitecturas con múltiples flujos de control (o threads o flujos de instrucciones))

- **TLP (Thread Level Parallelism).** Ej. múltiples flujos de control concurrentemente o en paralelo.
 - **Implícito.** Flujos de control creados y gestionados por la arquitectura.
 - **Explícito.** Flujos de control creados y gestionados por el SO.

- **Con una instancia SO.** Multiprocesadores, multicores, cores multithread...
- **Con múltiples instancias SO.** Multicomputadores.

2.3.7 Nivel del paralelismo aprovechado (propuesta de clasificación)

- **Arquitecturas con DLP (*Data Level Parallelism*).** Ejecutan las operaciones de una instrucción concurrentemente o en paralelo: unidades funcionales vectoriales o SIMD.
- **Arquitecturas con ILP (*Instruction Level Parallelism*).** Ejecutan múltiples instrucciones concurrentemente o en paralelo: cores escalares segmentados, superescalares o VLIW/EPIC.
- **Arquitecturas con TLP (*Thread Level Parallelism*) explícito y una instancia de SO.** Ejecutan múltiples flujos de control concurrentemente o en paralelo.
 - **Cores** que modifican la arquitectura escalar segmentada, superescalar o VLIW/EPIC para ejecutar threads concurrentemente o en paralelo.
 - **Multiprocesadores:** ejecutan threads en paralelo en un computador con múltiples cores (incluye multicores).
- **Arquitecturas con TLP explícito y múltiples instancias SO.** Ejecutan múltiples flujos de control en paralelo.
 - **Multicomputadores:** ejecutan threads en paralelo en un sistema con múltiples computadores.

1.2.4 2.4 Nota histórica

- **DLP (*Data Level Parallelism*).** Unidades funcionales (o de ejecución) SIMD (o multimedia).
- **ILP (*Instruction Level Parallelism*).**
 - Procesadores/cores segmentados.
 - Procesadores con múltiples unidades funcionales.
 - Procesadores/cores superescalares
 - Procesadores/cores VLIW
- **TLP (*Thread Level Parallelism*).**
 - TLP explícito con una instancia de SO.
 - Multithread grano fino (FGMT).
 - Multithread grano grueso (CGMT).
 - Multithread simultánea (SMT).
 - Multiprocesadores en un chip (CMP) o multicores.
 - Multiprocesadores.
 - TPL explícito con múltiples instancias del SO (multicomputadores): IC.SCAP.

1.3 Lección 3. Evaluación de prestaciones.

1.3.1 3.1 Objetivos.

- Distinguir entre tiempo de CPU (sistema y usuario) de unix y tiempo de respuesta.
- Distinguir entre productividad y tiempo de respuesta.
- Obtener, de forma aproximada mediante cálculos, el tiempo de CPU, GFLOPS y los MIPS del código ejecutado en un núcleo de procesamiento.
- Explicar el concepto de ganancia en prestaciones.
- Aplicar la ley de Amdahl.

1.3.2 3.2 Medidas usuales para evaluar prestaciones.

3.2.1 Tiempo de respuesta.

- Real (*wall-clock time, elapsed time, real time*).
- $CPU\ time = user + sys$ (no incluye todo el tiempo).
- Con un flujo de control.
 - $elapsed \geq CPU\ time$.

```

1 | time ./program.exe
2 | elapsed 5.4
3 | user 3.2
4 | sys 1.0

```

- Con múltiples flujos de control
 - $elapsed < CPU\ time$, $elapsed \geq CPU\ time / n^{\circ}\ fluxos\ control$.

En el programa, **user 3.2** significa el tiempo de CPU de usuario (tiempo de ejecución en espacio de usuario). **sys 1.0** significa el tiempo de CPU de sistema (tiempo en el nivel del kernel del SO). Además, hay otro tiempo asociado a las esperas debidas a I/O o asociados a la ejecución de otros programas.

Comando time en Unix: 3.2u + 1.0s es el 78 % del tiempo transcurrido (5.4).

Alternativas para obtener tiempos:

Función	Fuente	Tipo	Resolución aprox. (microsegs)
time	SO (/usr/bin/time)	elapsed, user, system	10000
clock()	SO (time.h)	CPU	10000
gettimeofday()	SO (sys/time.h)	elapsed	1
clock_gettime() / clock_getres()	SO (time.h)	elapsed	0.001
omp_get_wtime() / omp_get_wtick()	OpenMP (omp.h)	elapsed	0.001
SYSTEM_CLOCK()	Fortran	elapsed	1

RISC: menos instrucciones que CISC

CISC tienen instrucciones de acceso a memoria de la ALU

Los RISC para acceder a memoria, solamente se pueden usar instrucciones dedicadas a memoria

for (i=0; i<N; i++) {
 v3[i]=v1[i]+v2[i];
}

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

$T_{CPU} = NI \times \frac{1}{IPC} \times T_{ciclo}$

$T_{CPU} = \frac{N \text{ ciclos}}{\text{código}} \times T_{ciclo}$

$T_{CPU} = \left[\sum_i NI_i \times CPI_i \right] \times T_{ciclo}$

$T_{CPU} = NI \times \left(\frac{\sum_i NI_i \times CPI_i}{NI} \right) \times T_{ciclo}$

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

$T_{ciclo} = 1/F$

Suma vectores de doubles

.L7:
... ; rax=0, rbx=8N
movsd v1(%rax), %xmm0
addsd v2(%rax), %xmm0
movsd %xmm0, v3(%rax)
addq \$8, %rax
cmpq %rbx, %rax
jne .L7
...

i	NI _i	CPI _i
movsd m,r	2N	4
movsd r,m	N	5
addsd m,r	N	1
addq i,r	N	1
cmp r,r	N	1
jne	N	1
		6N

Para $N = 10^3$ y $F = 100\text{MHz}$ ($\Rightarrow T_{ciclo} = 10^{-8}\text{seg./ciclo}$):

$$T_{CPU} \approx \left[6N \times \left(\frac{2N \times 4 + N \times 5 + 3N \times 1}{6N} \right) \right] \times T_{ciclo}$$

$$= 10^3 \times 16 \text{ ciclos/código} \times 10^{-8}\text{seg./ciclo}$$

$$= 16 \times 10^{-5}\text{seg./código}$$

$$TiempoDeCPU (T_{CPU}) = CiclosDelPrograma \cdot T_{CICLO} = \frac{CiclosDelPrograma}{FrecuenciaDeReloj}$$

$$CiclosporInstrucción (CPI) = \frac{CiclosDelPrograma}{NúmeroDeInstrucciones(NI)}$$

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

$$CiclosDelPrograma = \sum_{i=1}^n CPI_i \cdot I_i$$

$$CPI = \frac{\sum_{i=1}^n CPI_i \cdot I_i}{NI}$$

En el programa hay I_i instrucciones del tipo i ($i=1, \dots, n$).

Cada instrucción del tipo i consume CPI_i ciclos.

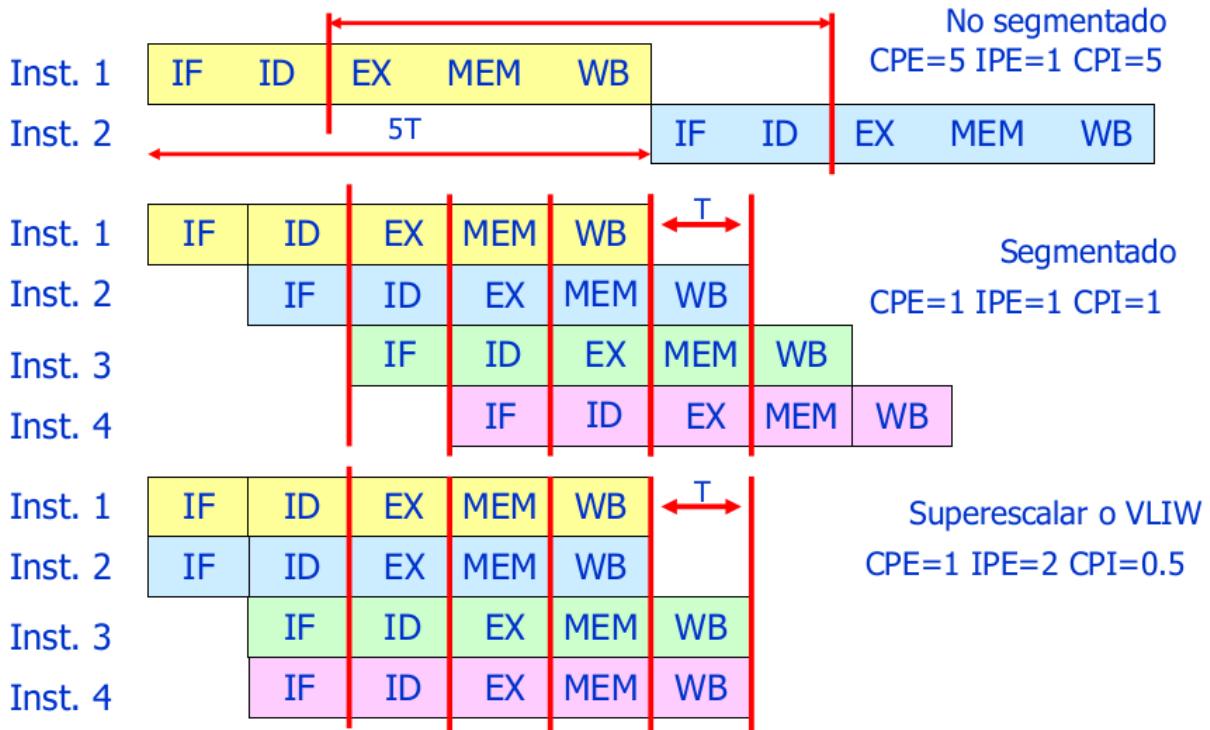
Hay n tipos de instrucciones distintos.

$$T_{CPU} = NI \cdot (CPE/IPE) \cdot T_{CICLO}$$

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

Hay procesadores que pueden lanzar para que empiecen a ejecutarse (emitir) varias instrucciones al mismo tiempo.

- **CPE:** Número mínimo de ciclos transcurridos entre los instantes en que el procesador puede emitir instrucciones
- **IPE:** Instrucciones que pueden emitirse (para empezar su ejecución) cada vez que se produce dicha emisión.

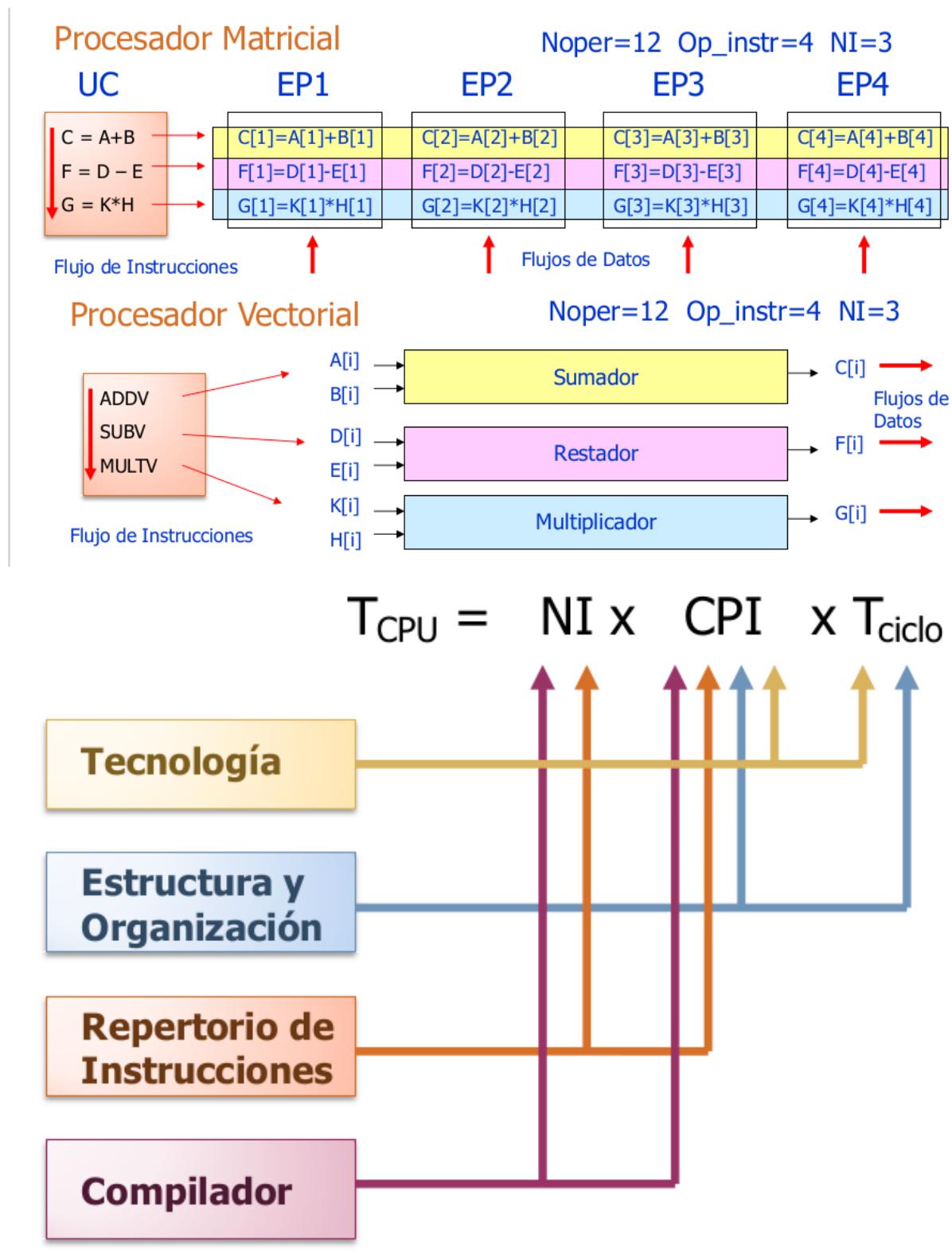


$$T_{CPU} = (Noper / OpInstr) \cdot CPI \cdot T_{CICLO}$$

$$NI = Noper / OpInstr$$

Hay procesadores que pueden codificar varias operaciones en una instrucción.

- **Noper:** Número de operaciones que realiza el programa
- **Op_instr:** Número de operaciones que puede codificar una instrucción.



3.2.2 Productividad: MIPS, MFLOPS.

MIPS: millones de instrucciones por segundo.

$$MIPS = \frac{NI}{T_{CPU} \cdot 10^6} = \frac{F(frecuencia)}{CPI \cdot 10^6}$$

- Depende del repertorio de instrucciones (difícil la comparación de máquinas con repertorios distintos)
- Puede variar con el programa (no sirve para caracterizar la máquina)
- Puede variar inversamente con las prestaciones (mayor valor de MIPS corresponde a peores prestaciones)

MFLOPS: millones de operaciones en coma flotante por segundo.

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

- No es una medida adecuada para todos los programas (sólo tiene en cuenta las operaciones en coma flotante del programa)
- El conjunto de operaciones en coma flotante no es constante en máquinas diferentes y la potencia de las operaciones en coma flotante no es igual para todas las operaciones (por ejemplo, con diferente precisión, no es igual una suma que una multiplicación..).
- Es necesaria una normalización de las instrucciones en coma flotante

MIPS y FLOPS

AC FTC

-O2

```
;r12=&x,r13=&y,rax=0,rbp=N,xmm1=a
.L6:
    movsd (%r12,%rax,8), %xmm0
    mulsd %xmm1, %xmm0
    addsd (%r13,%rax,8), %xmm0
    movsd %xmm0, (%r13,%rax,8)
    addq $1, %rax
    cmpl %eax, %ebp
    jg .L6
T(N=226)=0.182 seg.
```

$$\begin{aligned} GIPS &= \frac{NI}{T_{CPU} \times 10^9} = \frac{N \times 7}{0.182 \times 10^9} \\ &= \frac{2^{26} \times 7}{0.182 \times 10^9} \approx 2.58 \text{ GIPS} \end{aligned}$$

$$\begin{aligned} GFLOPS &= \frac{n^o FP}{T_{CPU} \times 10^9} = \frac{N \times 2}{0.182 \times 10^9} \\ &= \frac{2^{26} \times 2}{0.182 \times 10^9} \approx 0.737 \text{ GFLOPS} \end{aligned}$$

-O3

```
;r12=&x,r13=&y,rax=0,rbp=N/2,xmm1=a
.L7:
    movapd (%r12), %xmm0
    addq $1, %rax
    addq $16, %r12
    addq $16, %r13
    mulpd %xmm1, %xmm0
    addpd -16(%r13), %xmm0
    movaps %xmm0, -16(%r13)
    cmpl %ebp, %eax
    jb .L7
T(N=226)=0.178 seg.
```

$$\begin{aligned} GIPS &= \frac{NI}{T_{CPU} \times 10^9} = \frac{(N/2) \times 9}{0.178 \times 10^9} \\ &= \frac{2^{25} \times 9}{0.178 \times 10^9} \approx 1.7 \text{ GIPS} \end{aligned}$$

$$GFLOPS = \frac{2^{26} \times 2}{0.178 \times 10^9} \approx 0.754 \text{ GFLOPS}$$

1.3.3 3.3 Conjunto de programas de prueba (Benchmark).

- Propiedades exigidas a medidas de prestaciones:

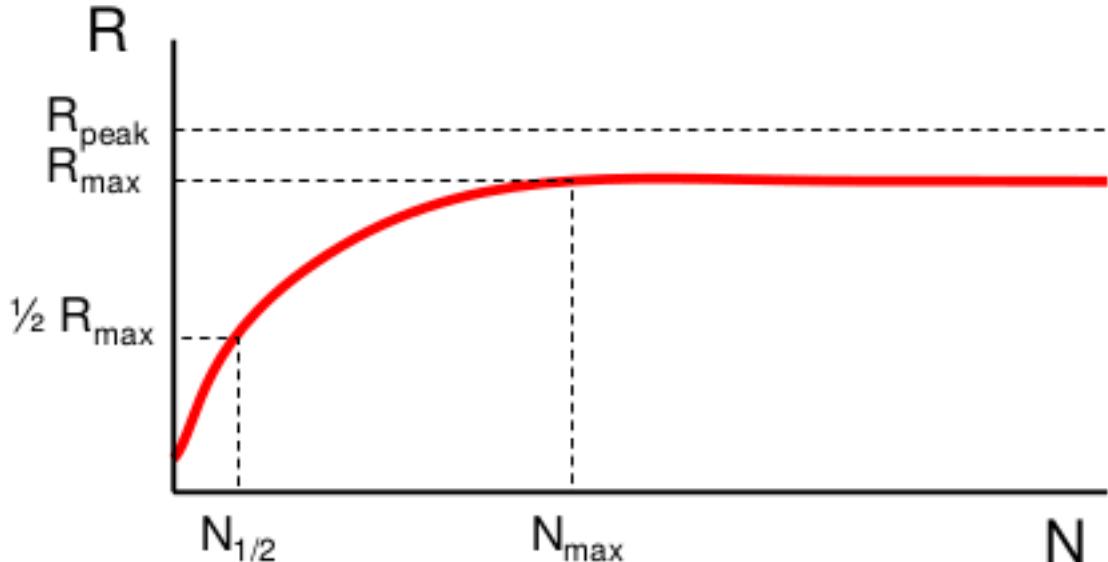
- Fiabilidad => Representativas, evaluar diferentes componentes del sistema y reproducibles.
 - Permitir comparar diferentes realizaciones de un sistema o diferentes sistemas => Aceptadas por todos los interesados (usuarios, fabricantes, vendedores).
- Interesados:
- Vendedores y fabricantes de hardware o software.
 - Investigadores de hardware o software.
 - Compradores de hardware o software.
- Tipos de Benchmarks:
- De bajo nivel o microbenchmark.
 - testping-pong, evaluación de las operaciones con enteros o con flotantes.
 - Kernels.
 - resolución de sistemas de ecuaciones, multiplicación de matrices, FFT, descomposición LU.
 - Sintéticos.
 - Dhrystone, Whetstone.
 - Programas reales.
 - SPEC CPU2006: enteros (gcc, gzip, perlbench).
 - Aplicaciones diseñadas.
 - Predicción de tiempo, simulación de terremotos.

1.3.4 3.3.1 LINPACK.

El núcleo de este programa es una rutina denominada DAXPY (Double precision- real Alpha X Plus Y) que multiplica un vector por una constante y los suma a otro vector. Las prestaciones obtenidas se escalan con el valor de N (tamaño del vector):

```
1 | for (i=0; i<N; i++)
2 |   y[i] = alpha*x[i] + y[i];
```

```
for (i=0 ; i<N ; i++)
y[i] = alpha*x[i] + y[i];
```



1.3.5 3.4 Ganancia en prestaciones.

3.4.1 Mejora o ganancia de prestaciones (speed-up o ganancia en velocidad).

Si en un computador se incrementan las prestaciones de un recurso haciendo que su velocidad sea p veces mayor (ejemplos: se utilizan p procesadores en lugar de uno, la ALU realiza las operaciones en un tiempo p veces menor...):

El incremento de velocidad que se consigue en la nueva situación con respecto a la previa (máquina base) se expresa mediante la ganancia de velocidad o speed-up, S_p

$$S_p = \frac{V_p}{V_1} = \frac{T_1}{T_p}$$

donde

V_1 : velocidad de la máquina base.

V_p : velocidad de la máquina mejorada (un factor p en uno de sus componentes).

T_1 : tiempo de ejecución en la máquina base.

T_p : tiempo de ejecución en la máquina mejorada.

Si se incrementan las prestaciones de un sistema, el incremento en prestaciones (velocidad) que se consigue en la nueva situación, p , con respecto a la previa (sistema base, b) se expresa mediante la ganancia en prestaciones o speed-up, S

$$S = \frac{V_p}{V_b} = \frac{T_b}{T_p}$$

$$S = \frac{T_{CPU}^b}{T_{CPU}^p} = \frac{NI^b \cdot CPI^b \cdot T_{CICLO}^b}{NI^p \cdot CPI^p \cdot T_{ciclo}^p}$$

donde

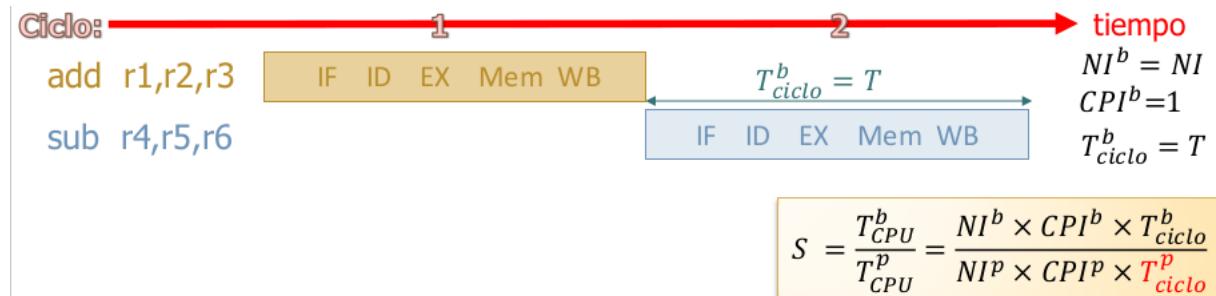
V_b : velocidad de la máquina base.

V_p : velocidad de la máquina mejorada (un factor p en uno de sus componentes).

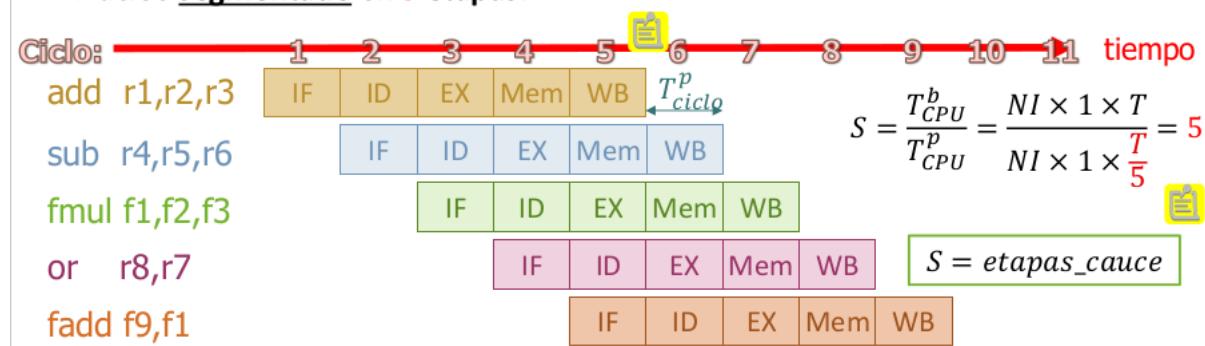
T_b : tiempo de ejecución en la máquina base.

T_p : tiempo de ejecución en la máquina mejorada.

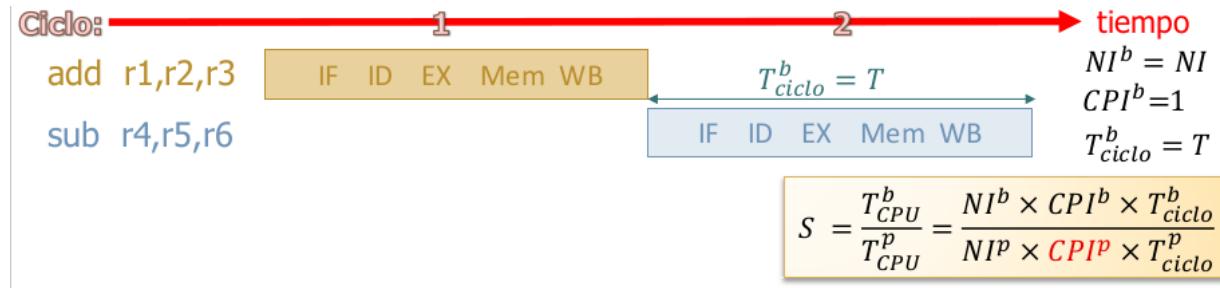
Mejora en un núcleo de procesamiento: segmentación.



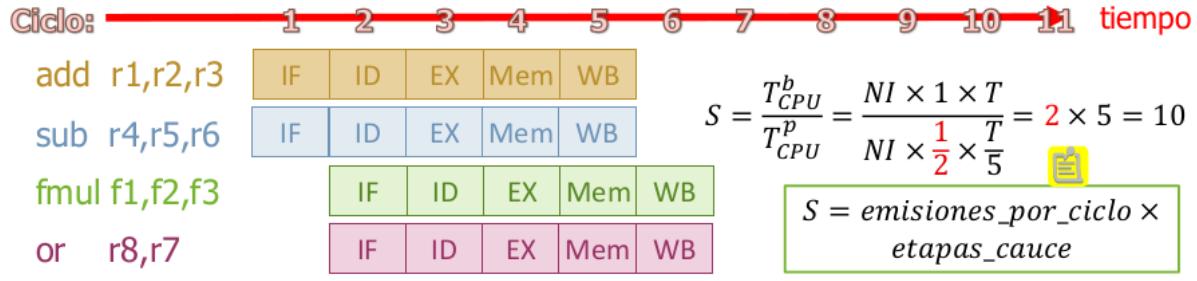
Núcleo segmentado en 5 etapas:



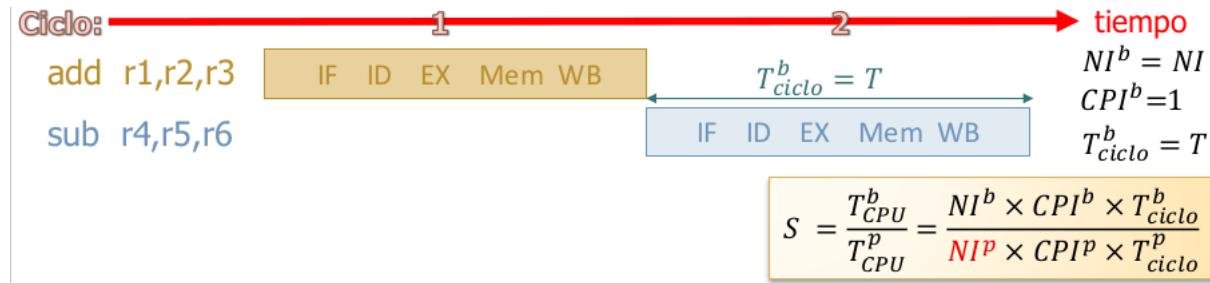
Mejora en un núcleo de procesamiento: operación superescalar.



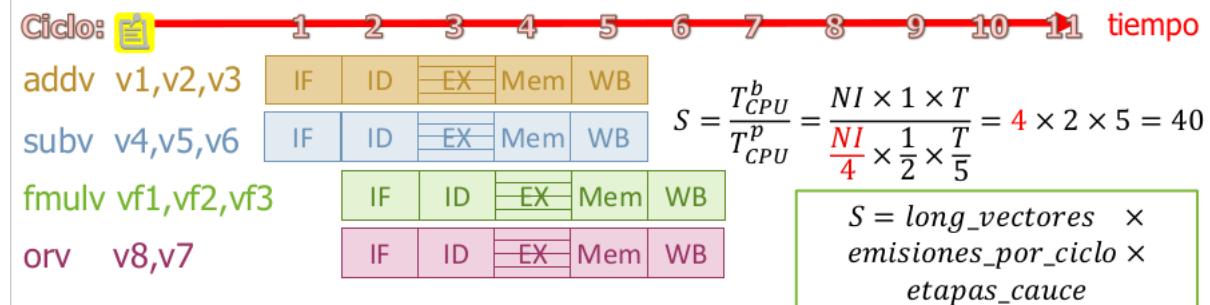
Núcleo superescalar con 2 emisiones por ciclo y 5 etapas:



Mejora en un núcleo de procesamiento: unidades funcionales SIMD.



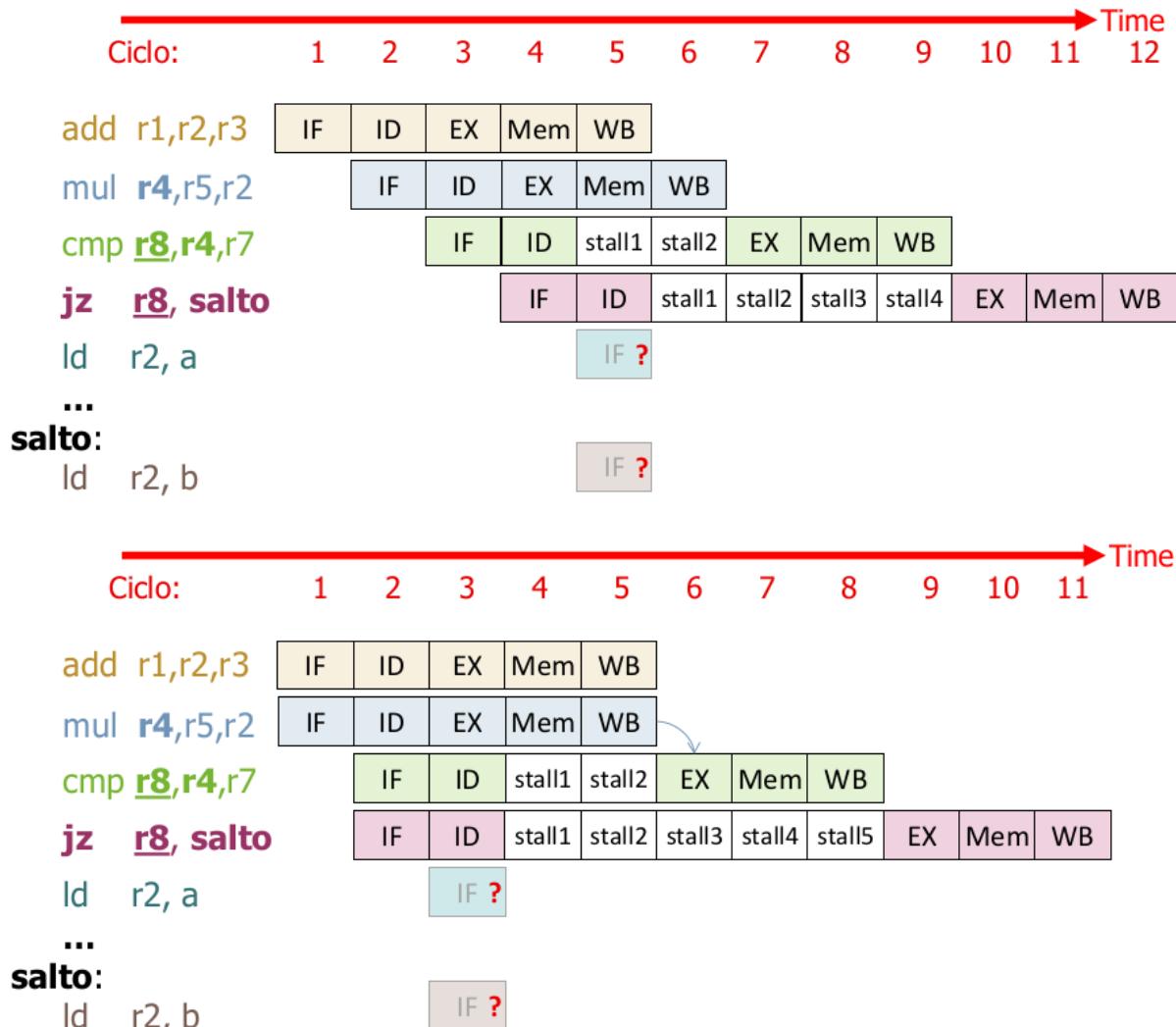
Núcleo superescalar con 2 emisiones por ciclo y 5 etapas, y unidades funcionales SIMD (vectoriales) que procesan **vectores de 4 componentes**:



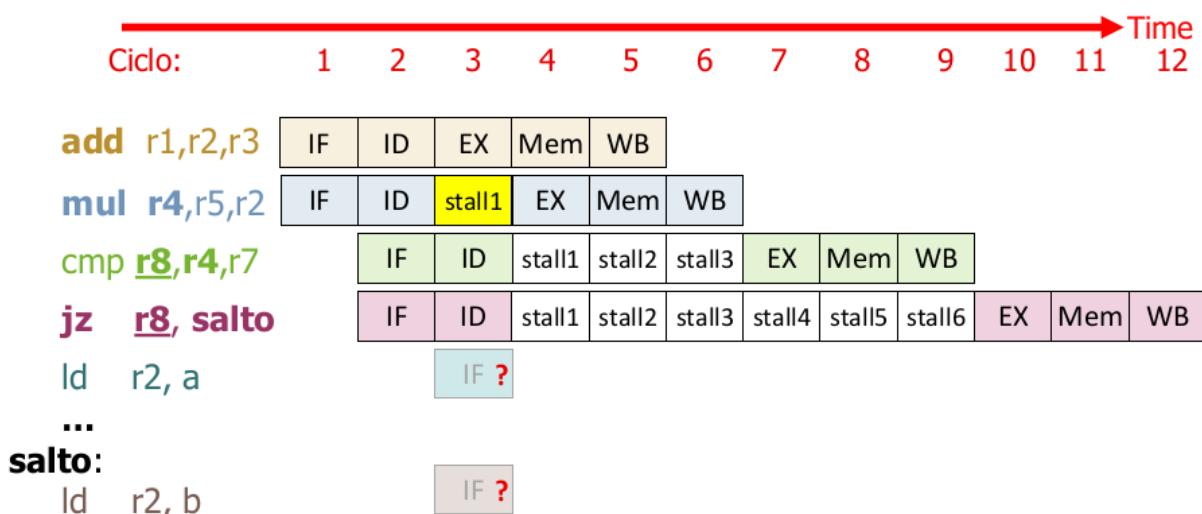
¿Qué impide que se pueda obtener la ganancia en velocidad pico?

- Riesgos:
 - Datos.
 - Control.
 - Estructurales.
- Accesos a memoria (debido a la jerarquía).

Riesgos de datos y control:



Riesgos de datos, control y estructural:



add y mul usan la misma unidad funcional

3.4.2 Ley de Amdahl.

La mejora de velocidad, S , que se puede obtener cuando se mejora un recurso de una máquina en un factor p está limitada por:

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

Si $p \rightarrow \infty$, entonces $\frac{p}{1+f(p-1)} \rightarrow 1/f$.

Si $f \rightarrow 0$, entonces $\frac{p}{1+f(p-1)} \rightarrow p$.

donde f es la fracción del tiempo de ejecución en la máquina sin la mejora durante el que no se puede aplicar esa mejora.

Ejemplo: Si un programa pasa un 25 % de su tiempo de ejecución en una máquina realizando instrucciones de coma flotante, y se mejora la máquina haciendo que estas instrucciones se ejecuten en la mitad de tiempo, entonces $p=2$; $f=0.75$.

$$S \leq 2/(1+0.75)=1.14$$

$$S = \frac{T_b}{T_p} = \frac{1}{0.75+\frac{0.25}{2}} = 1.14$$

Hay que mejorar el caso más frecuente (lo que más se usa)

Ley enunciada por Amdahl en relación con la eficacia de los computadores paralelos: dado que en un programa hay código secuencial que no puede parallelizarse, los procesadores no se podrían utilizar eficazmente.

2 Tema 2. Programación paralela

2.1 Lección 4. Herramientas, estilos y estructuras en programación paralela.

2.1.1 Objetivos.

- Distinguir entre los diferentes tipos de herramientas de programación paralela: compiladores paralelos, lenguajes paralelos, API Directivas y API de funciones.
- Distinguir entre los diferentes tipos de comunicaciones colectivas.
- Diferenciar el estilo/paradigma de programación de paso de mensajes del de variables compartidas.
- Diferenciar entre OpenMP y MPI en cuanto a su estilo de programación y tipo de herramienta.
- Distinguir entre las estructuras de tareas/procesos/threads master-slave, cliente-servidor, descomposición de dominio, flujo de datos o segmentación, y divide y vencerás.

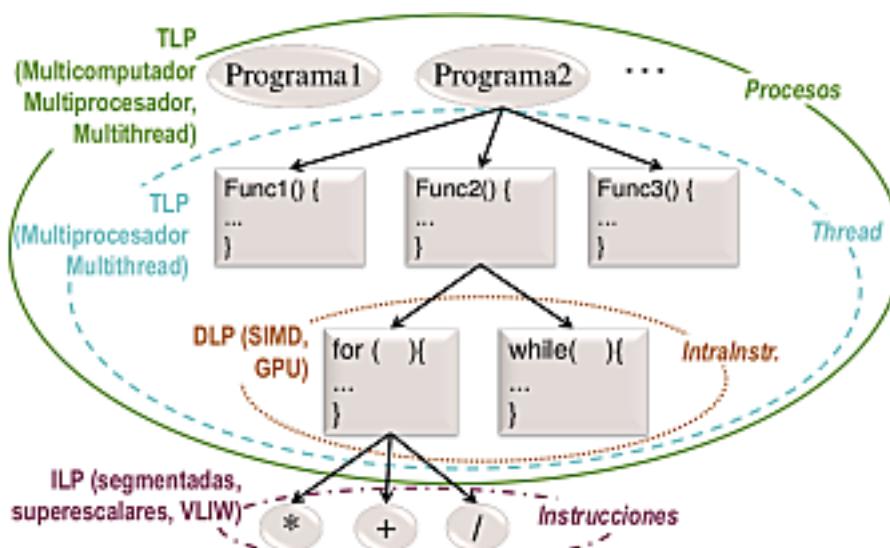
2.1.2 4.1 Problemas que plantea la programación paralela al programador. Punto de partida.

4.1.1 Problemas que plantea la programación paralela al programador.

Nuevos problemas, respecto a programación secuencial:

- **División** en unidades de cómputo independientes (tareas).
- **Agrupación/asignación** de tareas o carga de trabajo (códigos, datos) en procesos/threads.
- **Asignación** a procesadores/núcleos.
- **Sincronización y comunicación.**

Los debe abordar la herramienta de programación o el programador o SO.



4.1.2 Punto de partida.

Para obtener una versión paralela de una aplicación

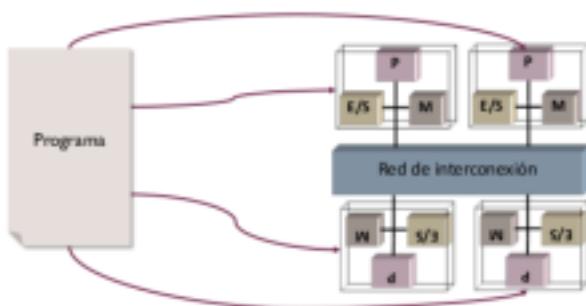
- se puede partir de una **versión secuencial** que resuelva el problema y buscar la parallelización sobre este. La versión paralela depende de la descripción del problema que se ha utilizado en la versión secuencial de partida. Ventaja: se puede saber el tiempo de ejecución real de las diferentes funciones o tareas, lo que facilita la distribución equilibrada de la carga de trabajo entre procesadores.
- se puede partir de la **definición de la aplicación**.

Se apoya en:

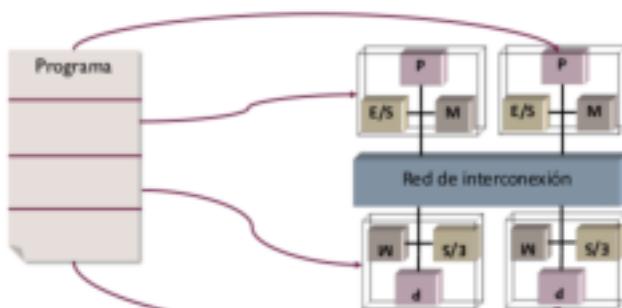
- **Programa** paralelo que resuelva un problema parecido.
- **Versiones** paralelas u optimizadas de bibliotecas de funciones: BLAS (*Basic Linear Algebra Subroutine*), LAPACK (*Linear Algebra PACKAGE*)...

4.1.3 Modos de programación MIMD.

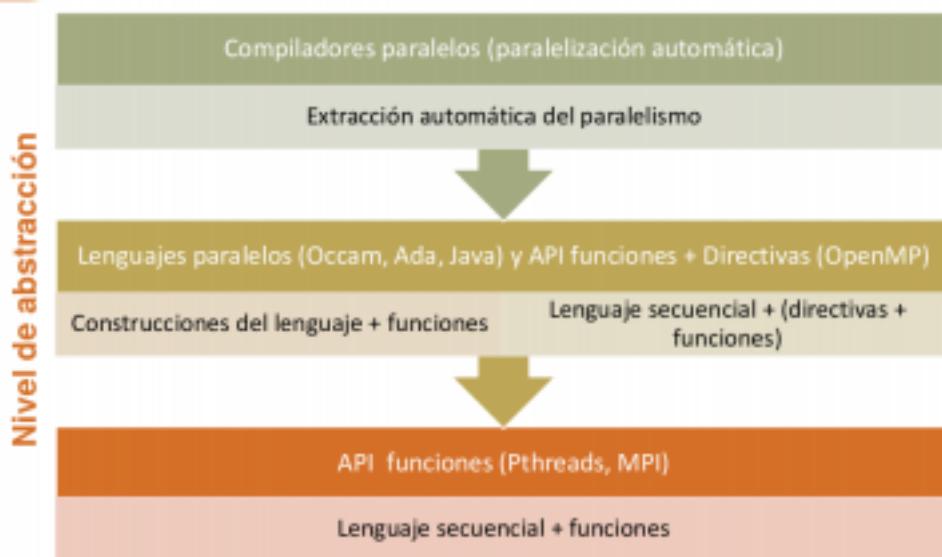
- **SPMD (Single-Program Multiple Data)**: parallelismo de datos. Todos los códigos que se ejecutan en paralelo se obtienen compilando el mismo programa. Cada copia trabaja con un conjunto de datos distintos y se ejecuta en un procesador diferente. Es recomendable en sistemas masivamente paralelos. Se usa en sistemas con memoria distribuida, en multiprocesadores y multicomputadores.



- **MPMD (Multiple-Program Multiple Data)**: parallelismo de tareas o funciones. Los códigos que se ejecutan en paralelo se obtienen compilando programas independientes. La aplicación a ejecutar (o el código secuencial inicial) se divide en unidades independientes. Cada unidad trabaja con un conjunto de datos y se asigna a un procesador distinto.



2.1.3 4.2 Herramientas para obtener código paralelo.



- Las **herramientas** permiten de forma implícita (lo hace la propia herramienta) o explícita (lo hace el programador):
 - Localizar **paralelismo** o descomponer en **tareas independientes** (*decomposition*).
 - Asignar las tareas, es decir, la **carga de trabajo** (código + datos), a procesos/threads (*scheduling*).
 - **Crear y terminar** procesos/threads (o enrolar y desenrolar en un grupo).
 - **Comunicar y sincronizar** procesos/threads.
- El programador, la herramienta o el SO se encarga de **asignar procesos/threads** a unidades de procesamiento (*mapping*).

Ejemplo: cálculo de PI con OpenMP/C.

```
#include <omp.h>
#define NUM_THREADS 4
main(int argc, char **argv) {
    double ancho,x, sum=0; int intervalos, i;
    intervalos = atoi(argv[1]);
    ancho = 1.0/(double) intervalos;
    omp_set_num_threads(NUM_THREADS); →Crear y Terminar
#pragma omp parallel →Comunicar y sincronizar
{
    #pragma omp for reduction(+:sum) private(x) \
                schedule(dynamic) →Asignar
    for (i=0;i< intervalos; i++) {
        x = (i+0.5)*ancho; sum = sum + 4.0/(1.0+x*x);
    }
    sum* = ancho;
}
```

Annotations in the code:

- A purple arrow labeled "Localizar y Asignar" points to the first line of the parallel block: `#pragma omp parallel`.
- A green arrow labeled "Comunicar y sincronizar" points to the `schedule(dynamic)` clause.
- A blue arrow labeled "Asignar" points to the `reduction(+:sum)` clause.
- A red arrow labeled "Crear y Terminar" points to the `omp_set_num_threads` call.

reduction: tiene que sumar todas las variables `sum` y las guarda en `sum`.

Ejemplo: cálculo de PI en MPI/C. (Modificación del bucle `for` para repartir el trabajo entre los procesos)

```
#include <mpi.h>
main(int argc, char **argv) {
    double ancho,x,sum,lsum; int intervalos,i,nproc,iproc;
    if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1); → Enrolar
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
    intervalos=atoi(argv[1]);
    ancho=1.0/(double) intervalos; lsum=0;
    for (i=iproc; i<intervalos; i+=nproc) {
        x = (i+0.5)*ancho; lsum+= 4.0/(1.0+x*x);
    }
    lsum*= ancho; → Comunicar/sincronizar
    MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE,
               MPI_SUM,0,MPI_COMM_WORLD);
    MPI_Finalize(); → Desenrolar
}
```

4.2.1 Comunicaciones colectivas.

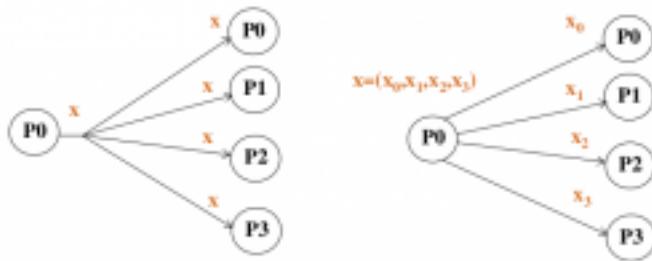


4.2.2 Comunicación uno-a-todos.

Un proceso envía y todos los procesos reciben. Hay variantes en las que el proceso que envía no forma parte del grupo y otras en las que reciben todos los del grupo excepto el que envía. Dentro de este grupo están:

- **Difusión**: todos los procesos reciben el mismo mensaje.
- **Dispersión (scatter)**: cada proceso receptor recibe un mensaje diferente.

Difusión (*broadcast*) Dispersión (*scatter*)

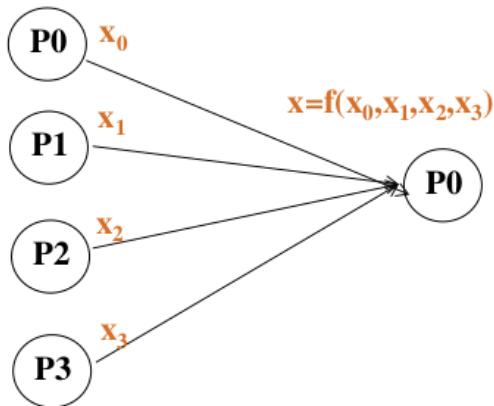


4.2.3 Comunicación todos-a-un.

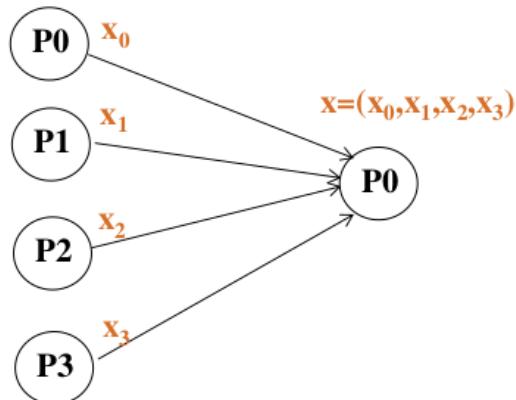
Todos los procesos en el grupo envían un mensaje a un único proceso:

- **Reducción:** los mensajes enviados por los procesos se combinan en un solo mensaje mediante un operador. La operación de combinación es usualmente commutativa y asociativa.
- **Acumulación (*gather*):** los mensajes se reciben de forma concatenada en el receptor (en una estructura vectorial). El orden en la que se concatenan depende del identificador del proceso.

Reducción



Acumulación (*gather*)

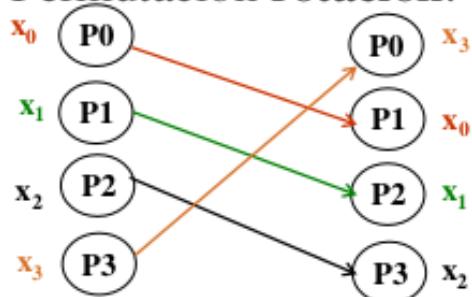


En la reducción, lo que envían todos los procesos se reduce a un único valor, aplicando conmutativa y asociativa. En acumulación se aplican los valores tal cual.

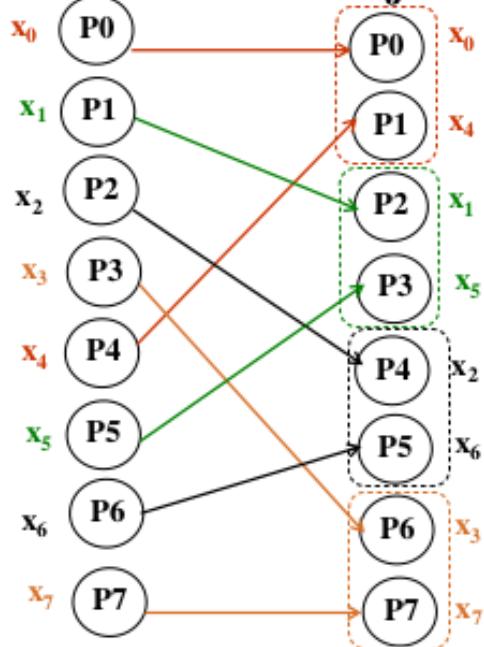
4.2.4 Comunicación múltiple uno-a-uno.

Hay componentes del grupo que envían (escriben) un único mensaje y componentes que reciben (lean) un único mensaje. Si todos los componentes envían y reciben, se implementa una **permutación**.

Permutación rotación:



Permutación baraje-2:

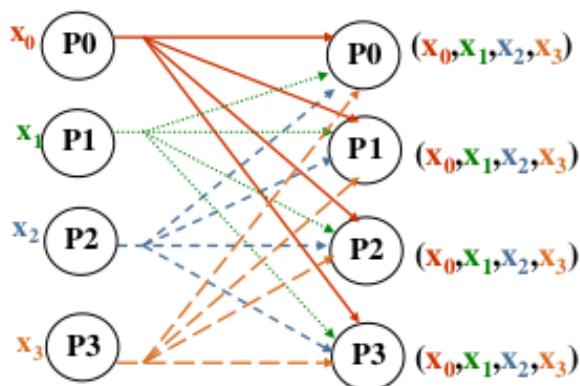


4.2.5 Comunicación todos-a-todos.

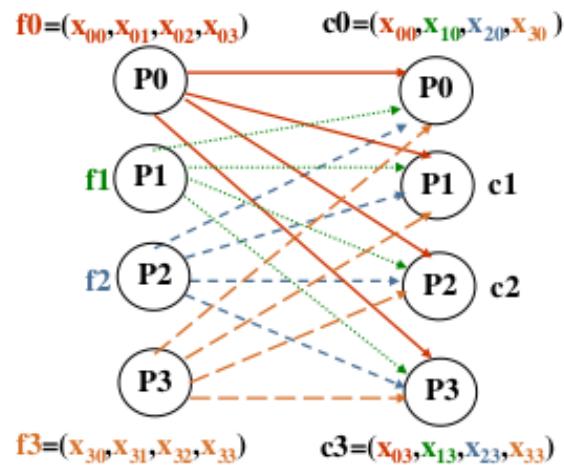
Todos los procesos del grupo ejecutan una comunicación uno-a-todos. Cada proceso recibe n mensajes, cada uno de un proceso diferente del grupo.

- **Todos difunden:** todos los procesos realizan una difusión. Usualmente las n transferencias recibidas por un proceso se concatenan en función del identificador del proceso que envía, de forma que todos los procesos reciben lo mismo.
- **Todos dispersan:** los procesadores concatenan diferentes transferencias. En la figura se ilustra la trasposición de una matriz 4×4 : el procesador P_i dispersa la fila i ($x_{i0}, x_{i1}, x_{i2}, x_{i3}$), tras la ejecución, el procesador P_i tendrá la columna i ($x_{0i}, x_{1i}, x_{2i}, x_{3i}$).

Todos Difunden (*all-broadcast*)
o chismorreo (*gossiping*)



Todos Dispersan (*all-scatter*)

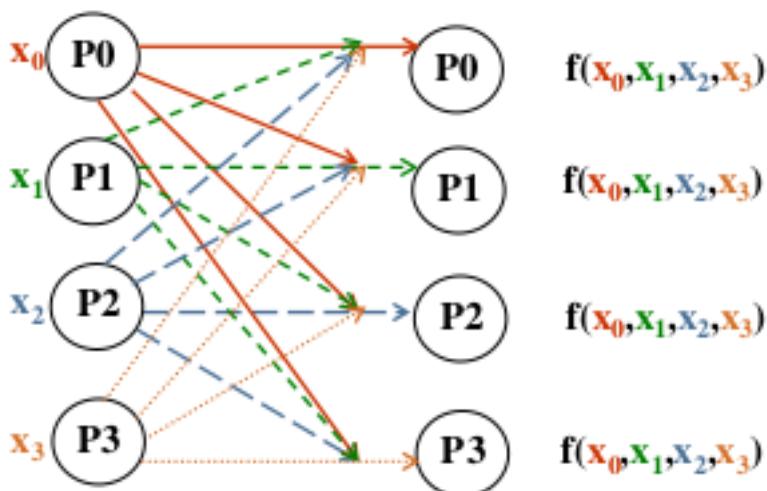


c = columna matriz
 f = fila matriz

4.2.6 Servicios compuestos.

- Todos combinan o reducción y extensión: el resultado de aplicar una reducción se obtiene en todos los procesos porque la reducción se difunde una vez obtenida (reducción y extensión) o porque se realizan tantas reducciones como procesos (todos combinan).

Todos combinan



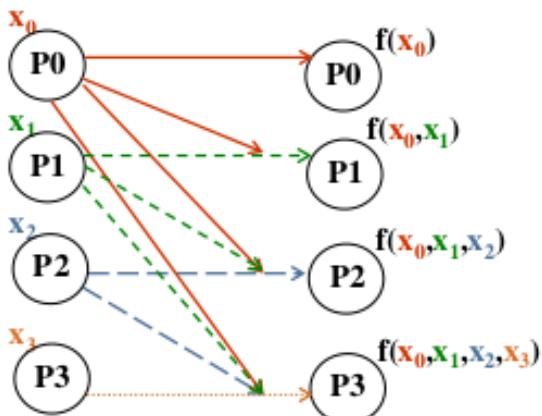
En la desviación típica se haría un todo reduce:

$$s = \sqrt{\frac{\sum_{i=1}^N (x_i - \text{media})^2}{N - 1}}$$

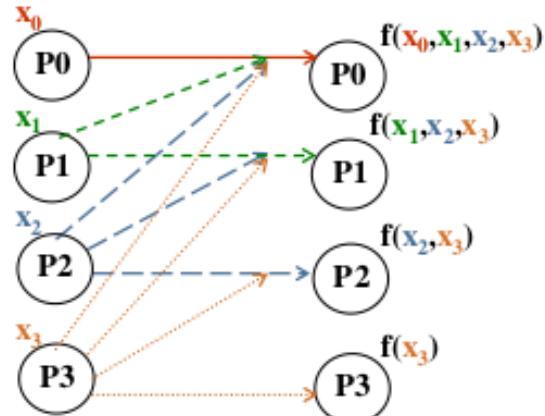
- **Recorrido (scan):** todos los procesos envían un mensaje, recibiendo cada uno de ellos el resultado de reducir un conjunto de estos mensajes.

- **Recorrido prefijo paralelo:** el proceso P_i recibe la reducción de los mensajes P_0, \dots, P_i .
- **Recorrido sufijo paralelo:** recibe la reducción de P_i, \dots, P_{n-1} .

Recorrido (scan) prefijo paralelo



Recorrido sufijo paralelo



Ejemplo: comunicación colectiva en OpenMP.

Uno-a-todos	Difusión (Seminario pract. 2)	<ul style="list-style-type: none"> ✓ Cláusula <code>firstprivate</code> (desde thread 0) ✓ Directiva <code>single</code> con cláusula <code>copyprivate</code> ✓ Directiva <code>threadprivate</code> y uso de cláusula <code>copyin</code> en directiva <code>parallel</code> (desde thread 0)
Todos-a-uno	Reducción (Seminario pract. 2)	<ul style="list-style-type: none"> ✓ Cláusula <code>reduction</code>
Servicios compuestos	Barreras (Seminario pract. 1)	<ul style="list-style-type: none"> ✓ Directiva <code>barrier</code>

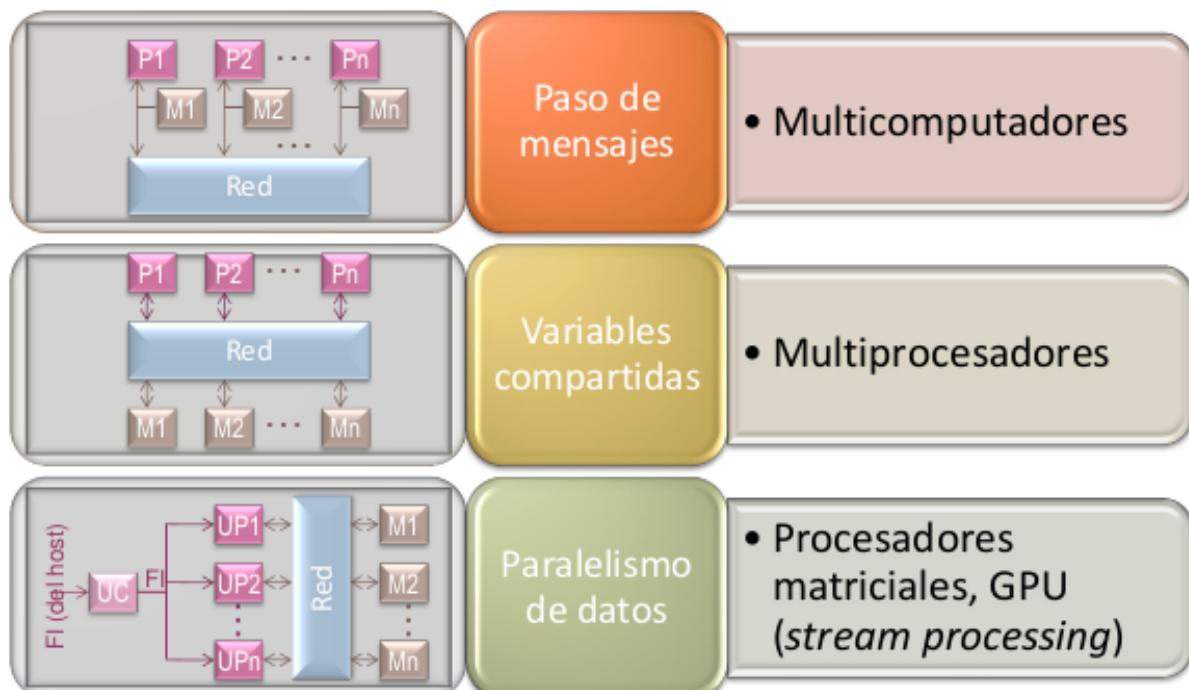
Ejemplo: comunicación en MPI.

Uno-a-uno	Asíncrona	<code>MPI_Send() / MPI_Receive()</code>
Uno-a-todos	Difusión	<code>MPI_Bcast()</code>
	Dispersión	<code>MPI_Scatter()</code>
Todos-a-uno	Reducción	<code>MPI_Reduce()</code>
	Acumulación	<code>MPI_Gather()</code>
Todos-a-todos	Todos difunden	<code>MPI_Allgather()</code>
Servicios compuestos	Todos combinan	<code>MPI_Allreduce()</code>
	Barreras	<code>MPI_Barrier()</code>
	Scan	<code>MPI_Scan</code>

2.1.4 4.3 Estilos/paradigmas de programación paralela.

4.3.1 Estilos de programación y arquitecturas paralelas.

- **Paso de mensajes:** también se puede usar en multiprocesadores.
- **Paralelismo de datos:** se corresponde con la arquitectura SIMD.



4.3.2 Estilos de programación y herramientas de programación.

- **Paso de mensajes** (*message passing*): cada procesador en el sistema tiene su espacio de direcciones propio. Los mensajes llevan datos de uno a otro espacio de direcciones y además se pueden aprovechar para sincronizar procesos. Los datos transferidos estarán duplicados en el sistema de memoria.
 - Lenguajes de programación: Ada, Occam.
 - API (Bibliotecas de funciones): MPI, PVM.
- **Variables compartidas** (*shared memory, shared variables*): se supone que los procesadores en el sistema comparten el mismo espacio de direcciones. Luego no necesitan transferir datos explícitamente, implícitamente se realiza la transferencia utilizando instrucciones del procesador de lectura y escritura en memoria. Para sincronizar, el programador utiliza primitivas que ofrece el software que se amparan en primitivas hardware para incrementar prestaciones.
 - Lenguajes de programación: Ada, Java.
 - API (directivas del compilador + funciones): OpenMP.
 - API (Bibliotecas de funciones): POSIX Threads, shmem, Intel TBB.
- **Paralelismo de datos** (*data parallelism*): las mismas operaciones se ejecutan en paralelo en múltiples unidades de procesamiento de forma que cada unidad aplica la operación a un conjunto de datos distinto. Solo soporta paralelismo a nivel de bucle. La sincronización está implícita. Dispone de construcciones para la distribución de datos entre los elementos de procesamiento.
 - Lenguajes de programación + funciones: HPF (High Performance Fortran), Fortran 95 (forall, operaciones con matrices/vectores), Nvidia CUDA.
 - API (directivas del compilador + funciones - stream processing): OpenACC.

2.1.5 4.4 Estructuras típicas de códigos paralelos.

4.4.1 Estructuras típicas de procesos/threads/tareas.

Estructuras típicas de procesos/threads en código paralelo:

- Descomposición de dominio o descomposición de datos cliente/servidor.
- Divide y vencerás o descomposición recursiva.
- Segmentación o flujo de datos.
- Master-Slave, o granja de tareas.

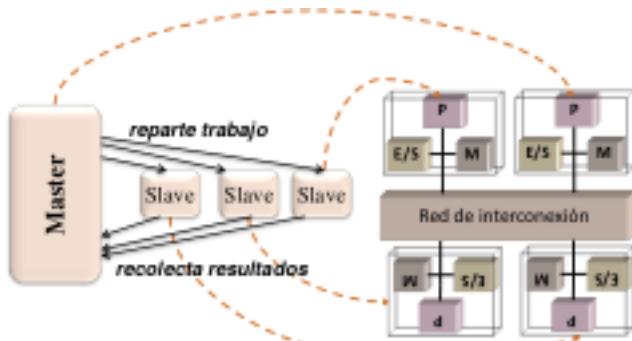
4.4.2 Master-Slave o granja de tareas.

Las tareas se representan con un círculo y los arcos representan flujo de datos.

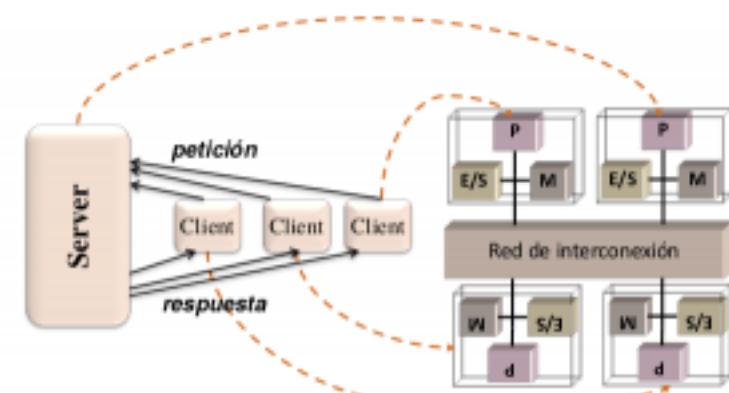
Tenemos un flujo de instrucciones que se encarga de repartir el trabajo entre esclavos y recolecta resultados. Los esclavos están ejecutando el mismo código. El máster puede hacer un trabajo distinto. Luego combinamos un MPMD con SPMD. La distribución de tareas entre los esclavos se puede realizar de forma dinámica o estática.

La escalabilidad del programa paralelo va a depender del número de esclavos y del camino de comunicación entre los esclavos y el dueño. Para incrementar la escalabilidad se puede dividir el dueño en múltiples dueños, cada uno con un conjunto diferente de esclavos.

Se puede llegar a esta estructura al paralelizar las iteraciones de un bucle



4.4.3 Cliente-Servidor.

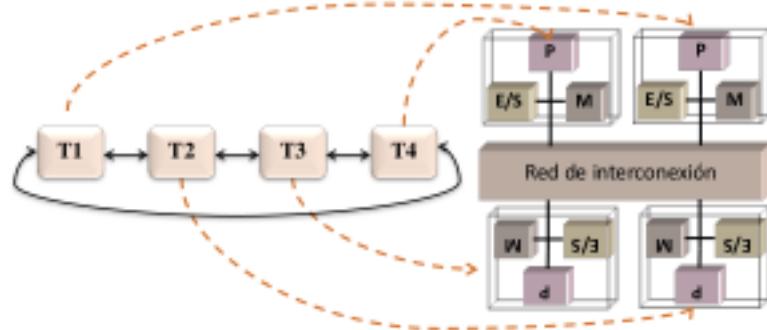


4.4.4 Descomposición de dominio o descomposición de datos.

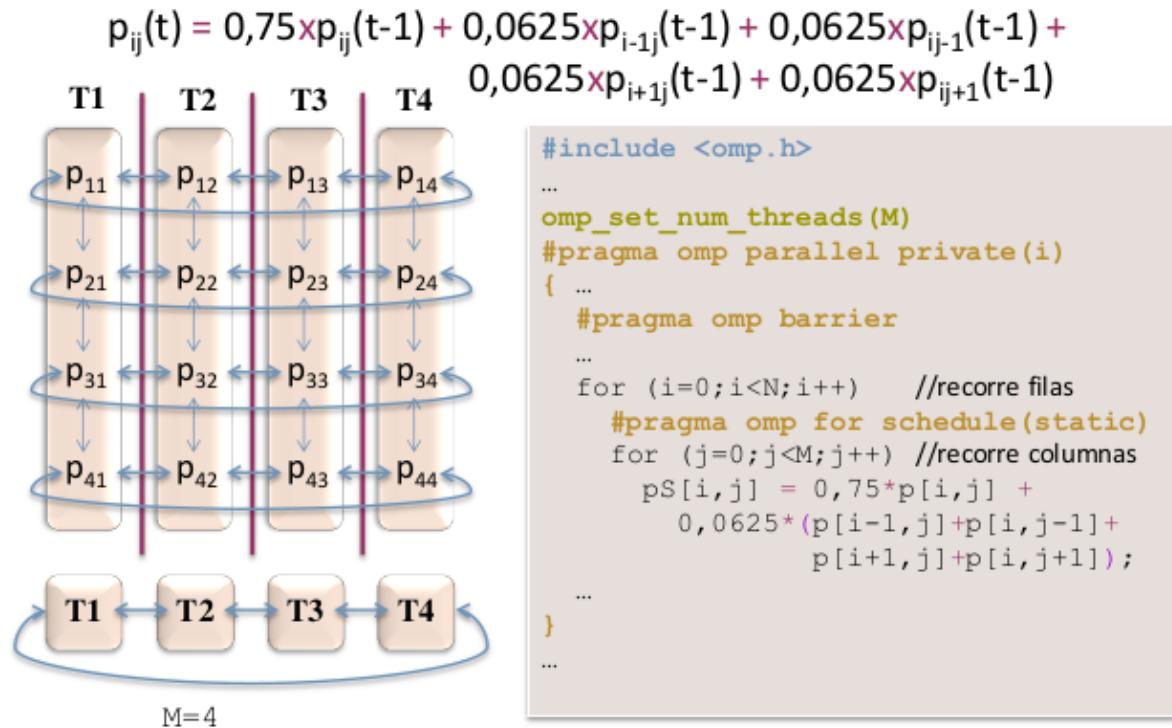
Es muy utilizada en problemas en los que se opera con grandes estructuras de datos. La estructura de datos de entrada o de salida o ambas se dividen en partes y se derivan las tareas paralelas, que realizan operaciones similares.

Los procesos pueden englobar varias tareas. Los diferentes procesos ejecutan típicamente el mismo código (SPMD), aunque cada uno trabaja sobre un conjunto de datos distintos. Puede haber comunicaciones entre los procesos.

- Se pueden representar con matrices
- Aplicación: inundaciones, software metereológico...



Ejemplo: filtrado imagen.



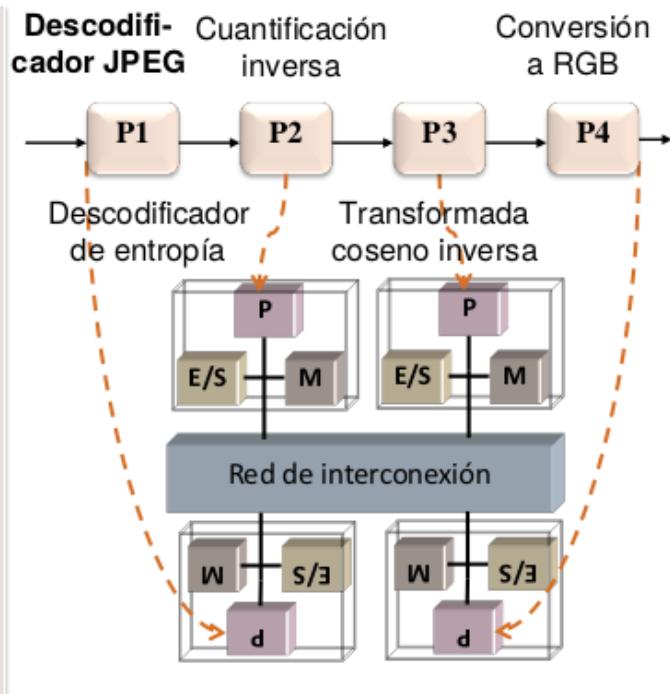
4.4.5 Estructura segmentada o de flujo de datos.

Aparece en problemas en los que se aplica a un flujo de datos en secuencia distintas funciones (paralelismo de tareas). La estructura de los procesos y de las tareas es la de un cauce segmentado. Cada proceso ejecuta por tanto distinto código (MPMD). Necesitamos que en la aplicación se aplique a una un flujo de entrada en secuencia una serie de operaciones, una detrás de otra. Ejemplo: MP3, MP4, multimedia...

En el caso de JPEG, los bloques se dividen en 8x8 bloques y se decodifica en el orden que indiquen las flechas. No puedo aplicar descomposición de dominio porque hay dependencia de bloques.

Podemos parallelizar una sola etapa, cuando se encuentren distintas estructuras en etapas.

```
#include <omp.h>
...
omp_set_num_threads(4);
...
#pragma omp parallel
{
    ...
    for (i=0;i<N;i++) {
        ...
        #pragma omp sections
        {
            #pragma omp section
            P1();
            #pragma omp section
            P2();
            #pragma omp section
            P3();
            #pragma omp section
            P4();
        }
        ...
    }
    ...
}
```



4.4.6 Divide y vencerás o descomposición recursiva.

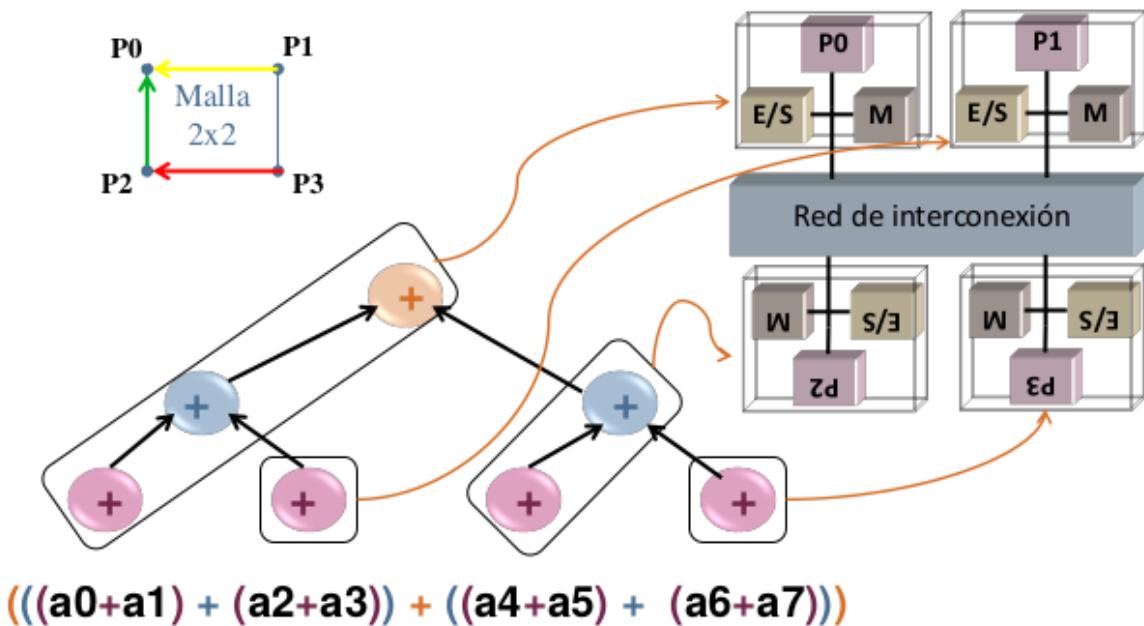
Se utiliza cuando un problema se puede dividir en dos o más subproblemas, de forma que cada uno se puede resolver independientemente, combinándose los resultados para obtener un resultado final.

Las tareas presentan una estructura en forma de árbol. No habrá interacciones entre las tareas que cuelgan del mismo padre. Puede haber paralelismo de tareas y de datos.

Los arcos representan flujo de datos. (flechas negras).

Agrupación/Asignación de tareas a flujos de instrucciones. (flechas negras).

En la imagen, usaríamos 4 flujos de datos como máximo porque el grado de paralelismo es 4. Las flechas naranjas representan la asignación de flujos de instrucciones.



2.2 Lección 5. Proceso de paralelización.

2.2.1 5.1 Objetivos.

- Programar en paralelo una aplicación sencilla.
- Distinguir entre asignación estática y dinámica (ventajas e inconvenientes).

2.2.2 5.2 Proceso de paralelización.

Los arcos en el grafo representan flujo de datos y de control, y los vértices, tareas.

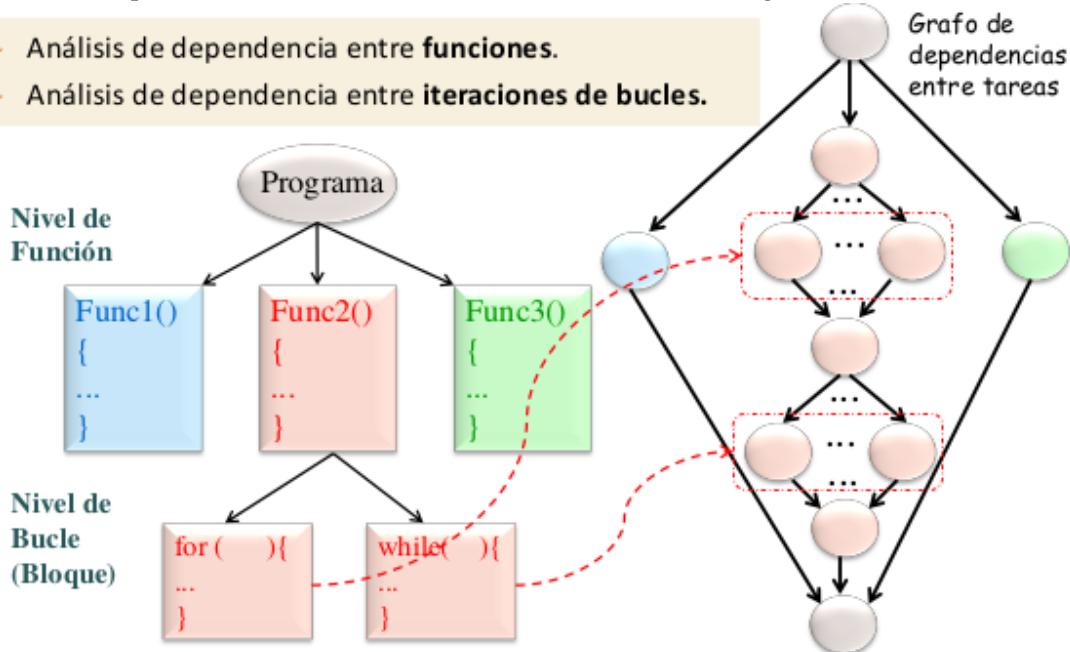
■ Descomponer en tareas independientes.

- Análisis de dependencia entre **funciones**. Para extraer el paralelismo de tareas, tenemos que analizar las dependencias entre las funciones que pueden ser independientes o pueden hacerse independientes.
- Análisis de dependencia entre **iteraciones de bucles**. Analizando las iteraciones de los bucles dentro de una función, podemos encontrar si son o se pueden hacer independientes. Podemos detectar paralelismo de datos. Si hay varios bucles, se puede analizar la dependencia entre ellos para ver si se pueden ejecutar en paralelo las iteraciones de múltiples bucles.

En la siguiente imagen podemos ver la descomposición en tareas. Una de las funciones consta de dos bucles. Dado el grafo de dependencias entre tareas de la figura, se ha encontrado que son independientes las iteraciones del `for`, cada iteración es una tarea en el grafo. También son

independientes las iteraciones de `while`. Además, el grafo muestra que la salida del bucle `for` se usa en `while`, por eso las tareas de ambos ciclos se encuentran en el grafo a distintos nivel.

- Análisis de dependencia entre **funciones**.
- Análisis de dependencia entre **iteraciones de bucles**.

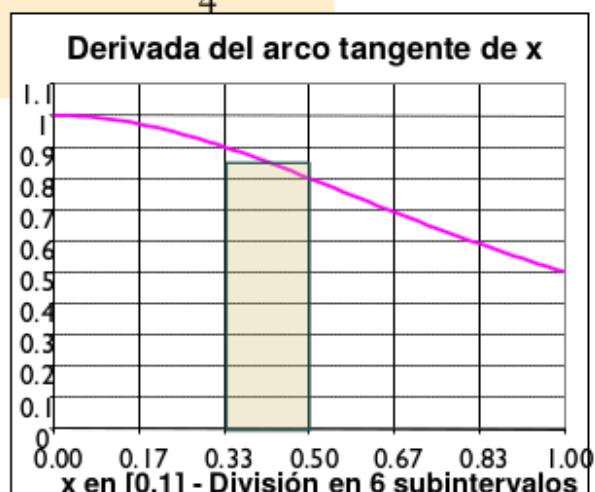


- Ejemplo de cálculo PI: descomposición en tareas independientes.

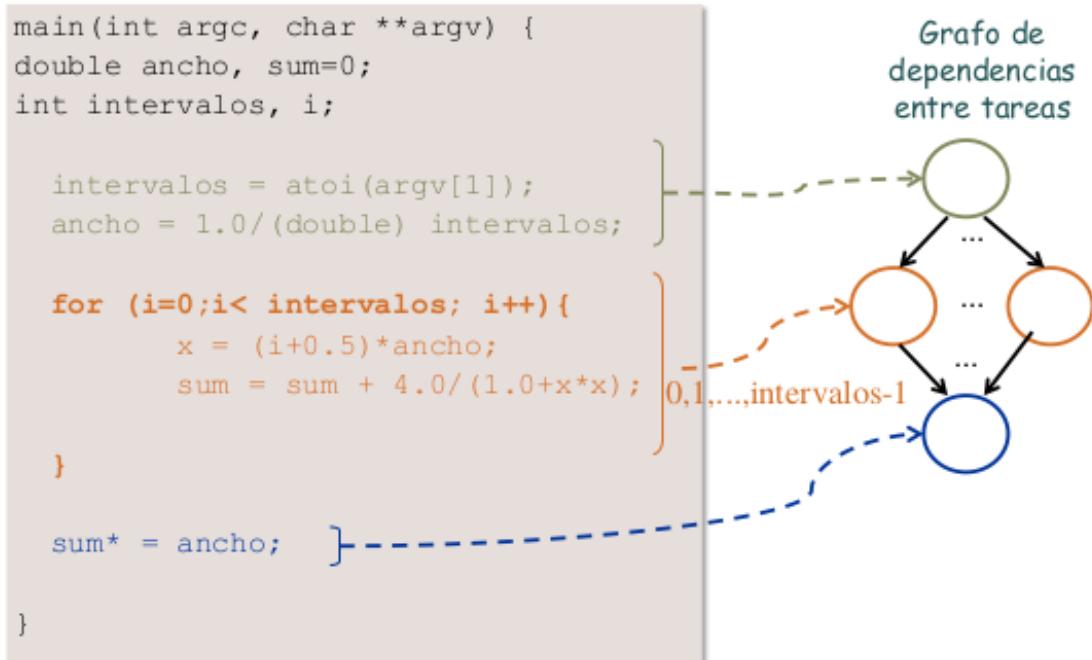
Se puede parallelizar pi fácilmente. Como la integral en el intervalo [0,1] de la derivada del arco tangente de x es $\pi/4$:

$$\left. \begin{array}{l} \arctg'(x) = \frac{1}{1+x^2} \\ \arctg(1) = \frac{\pi}{4} \\ \arctg(0) = 0 \end{array} \right\} \Rightarrow \int_0^1 \frac{1}{1+x^2} dx = \arctg(x)|_0^1 = \frac{\pi}{4} - 0$$

- PI se puede calcular por integración numérica.

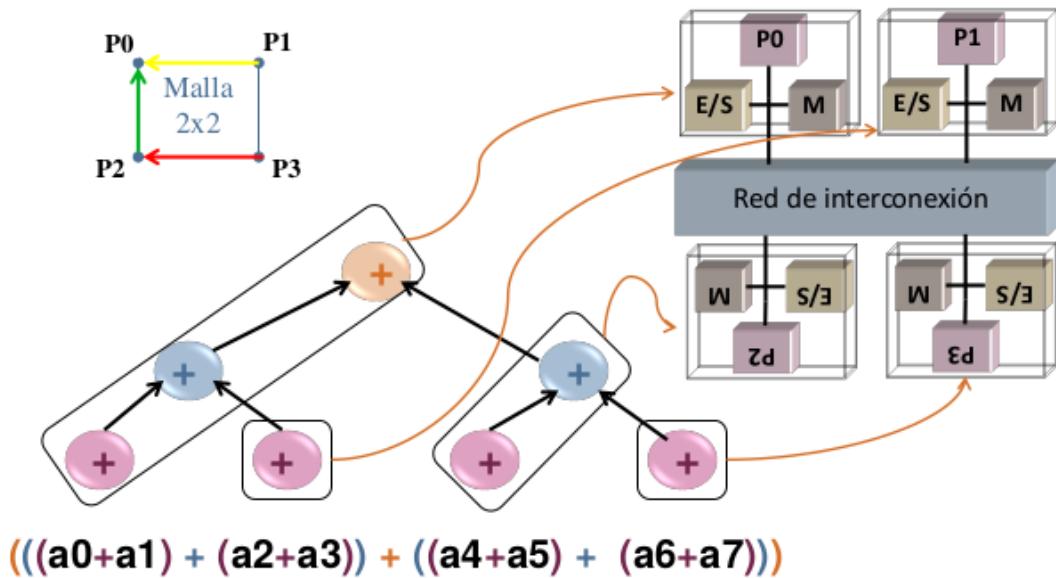


En la siguiente imagen se calcula π con una versión secuencial approximando el área en cada subintervalo utilizando rectángulos en los que la altura es el valor de la derivada del arco tangente en el punto medio.



- **Asignar (planificar+mapear) tareas a procesos y/o threads.** La asignación a procesos o hebras está ligada con la asignación a procesadores, incluso se puede realizar la asignación asociando los procesos (hebras) a procesadores concretos.

- Ejemplo: filtrado de imagen.



- Incluimos: agrupación de tareas en procesos/threads (scheduling) y mapeo a procesadores/cores (mapping).
- La **granularidad** de la carga de trabajo (tareas) asignada a los procesos/threads depende de:
 - número de cores o procesadores o elementos de procesamiento, cuanto mayor sea su número, menos tareas se asignarán a cada proceso (hebra).
 - tiempo de comunicación/sincronización frente a tiempo de cálculo.

- Para disminuir este tiempo, se pueden asignar más tareas a un proceso (hebra), así se reduce el número de interacciones entre tareas a través de la red.
- ¿Utilizar hebras o procesos? Depende de
 - **Arquitectura:**
 - ◊ Es más eficiente usar hebras en SMP y procesadores multihebra.
 - ◊ En arquitecturas mixtas se usan hebras y procesos, especialmente si el número de procesadores de un SMP es mayor que el número de nodos de un cluster.
 - **Sistema operativo:** debe ser multihebra.
 - **Herramientas de programación** para crear hebras y procesos.
- Se asignan hebras a las iteraciones de un **bucle** (paralelismo de datos).
- Se asignan procesos a **funciones** (paralelismo de tareas).
- **Equilibrado de la carga** (tareas = código + datos) o load balancing:
 - Objetivo: unos procesos/threads no deben esperar a otros.
- ¿De qué depende el equilibrado?
 - **La arquitectura:**
 - ◊ homogénea frente a la heterogénea.
 - ◊ uniforme frente a no uniforme.
 - La aplicación/descomposición.
- Tipos de asignación:
 - **Estática.**
 - ◊ Está determinado qué tarea va a realizar cada procesador o core.
 - ◊ Explícita: programador.
 - ◊ Implícita: herramienta de programación al generar el código ejecutable.
 - **Dinámica** (en tiempo de ejecución).
 - ◊ Permite que acabe una aplicación aunque falle algún procesador.
 - ◊ Consumo un tiempo adicional de comunicación y sincronización.
 - ◊ Distintas ejecuciones pueden asignar distintas tareas a un procesador o core.
 - ◊ Explícita: el programador.
 - ◊ Implícita: herramienta de programación al generar el código ejecutable.
- Mapeo de procesos/threads a unidades de procesamiento.
 - Normalmente se deja al SO el mapeo de threads (light process).
 - Lo puede hacer el entorno o sistema en tiempo de ejecución (runtime system de la herramienta de programación).
 - El programador puede influir.

- Códigos filtrado por imagen.

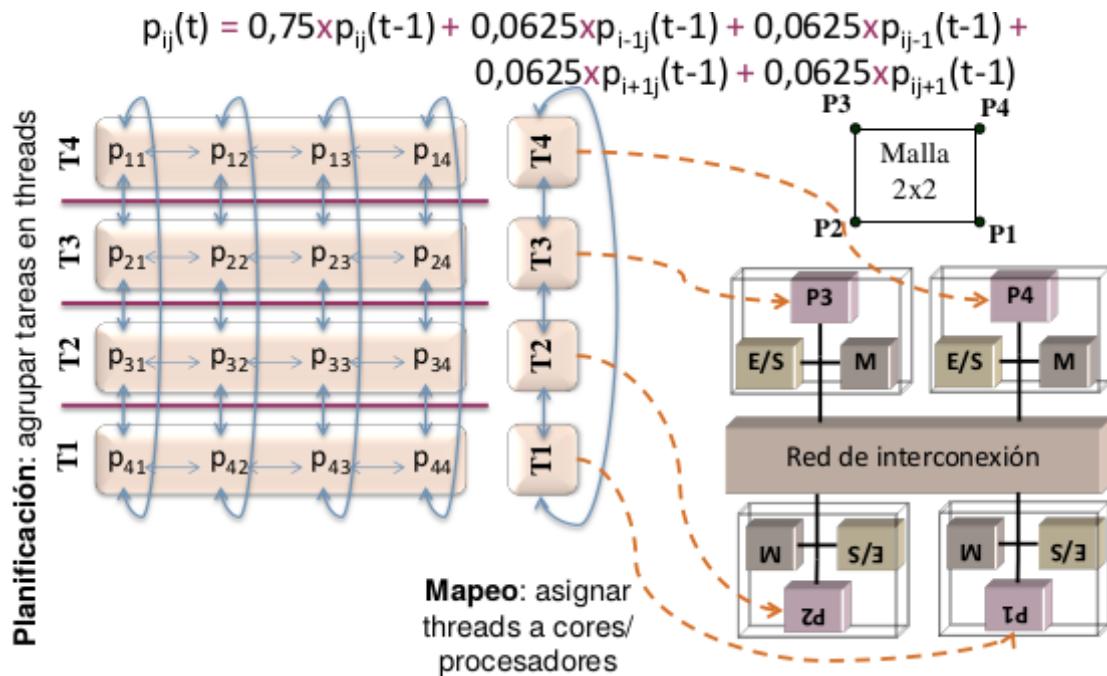
Descomposición por **columnas**.

```
1 #include <omp.h>
2 ...
3 omp_set_num_threads(M)
4 #pragma omp parallel private(i)
5 {
6     for (i=0;i<N;i++) {
7         #pragma omp for
8         for (j=0;j<M;j++) {
9             pS[i,j] = 0,75*p[i,j] + 0,0625*(p[i-1,j]+p[i,j-1]
10                + p[i+1,j]+ p[i,j+1]);
11         }
12     }
13 }
14 ...
```

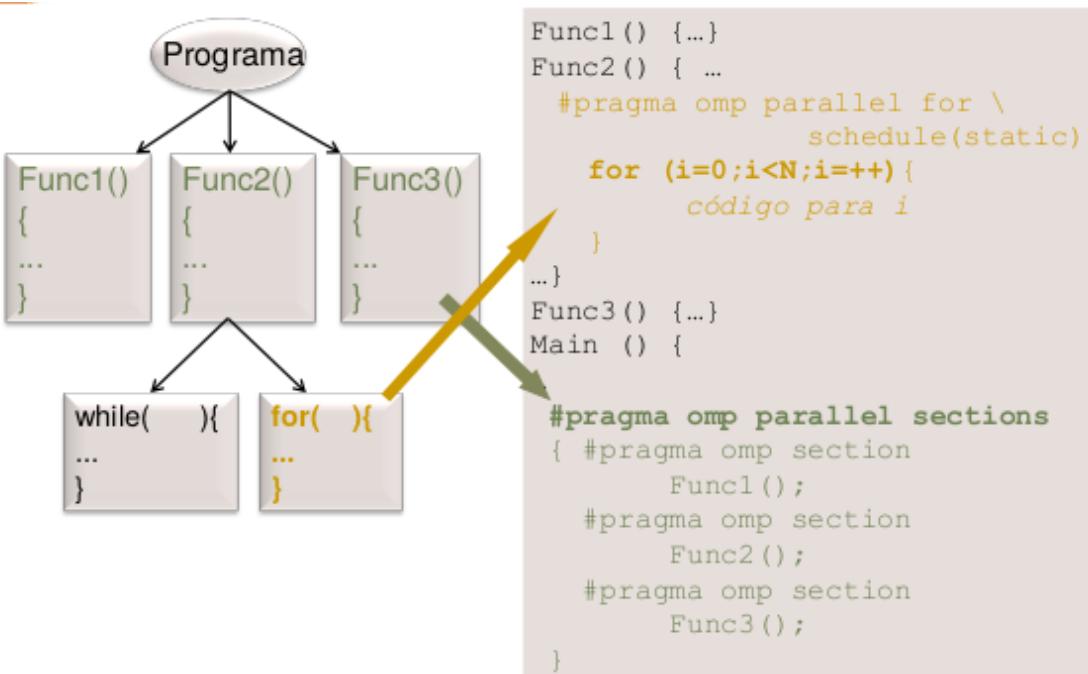
Descomposición por **filas**.

```
1 #include <omp.h>
2 ...
3 omp_set_num_threads(N)
4 #pragma omp parallel private(j)
5 {
6     #pragma omp for
7     for (i=0;i<N;i++) {
8         for (j=0;j<M;j++) {
9             pS[i,j] = 0,75*p[i,j] + 0,0625*(p[i-1,j]+p[i,j-1]+ p[i
10                +1,j]+ p[i,j+1]);
11         }
12     }
13 ...
```

- Ejemplo: filtrado de imagen.



- Ejemplo de asignación estática del parallelismo de tareas y datos con OpenMP.

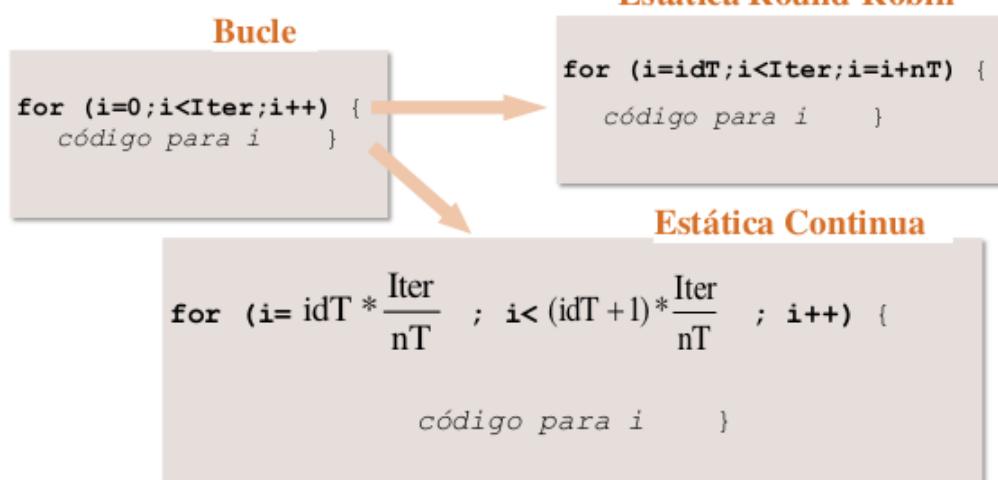


- Asignación estática.

- Asignación estática y explícita de las iteraciones de un bucle.

En la siguiente imagen se pueden ver dos alternativas que el programador puede utilizar explícitamente para asignar las iteraciones de un bucle a procesos (hebras) de forma estática. En la asignación turno rotatorio (*round-robin*), iteraciones consecutivas del bucle se asignan a procesos consecutivos (con identificador consecutivo). En el otro ejemplo se asignan iteraciones consecutivas al mismo proceso.

➤ Asignación **estática** y explícita de las iteraciones de un bucle:

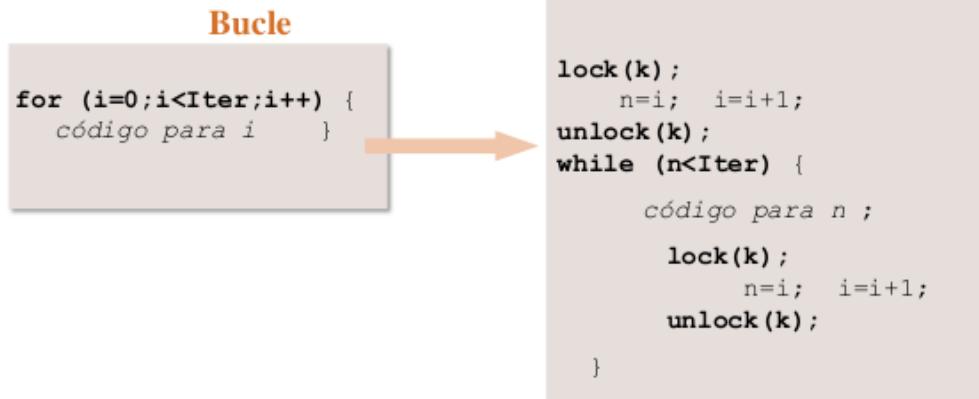


- Asignación dinámica.

- Asignación dinámica y explícita de las iteraciones de un bucle.

En la siguiente imagen se ve cómo el programador puede implementar explícitamente en el código una asignación dinámica para memoria compartida. Las iteraciones se reparten en orden. La variable *i* es compartida. Los procesos (hebras) consultan la variable *i* para “coger” la siguiente iteración que van a realizar, e incrementan su valor en uno para que el siguiente proceso no coja una iteración ya asignada. Para que una iteración no se ejecute dos veces, se utiliza un cerrojo *k* para excluir la lectura y modificación. Esquema: dueño-esclavo.

➤ Asignación **dinámica** y explícita de las iteraciones de un bucle:



➤ Dinámica e implícita

Ej. con OpenMP

NOTA: La variable *i* se supone inicializada a 0

- Dinámica e implícita.

- Asignación. Ejemplo: multiplicación matriz por vector.

$$c = A \bullet b; \quad c_i = \sum_{k=0}^{M-1} a_{ik} \bullet b_k = a_i^T \bullet b, \quad c(i) = \sum_{k=0}^{M-1} A(i,k) \bullet b(k), \quad i = 0, \dots, N-1$$

\bullet

■ Redactar código paralelo.

- El código depende de:
 - Estilo de programación (variables compartidas, paralelismo...).
 - Modo de programación (SPMD, MPMD...).
 - Punto de partida.
 - Herramienta software para el paralelismo.
 - Estructura.
- Se añaden o utilizan en el programa las funciones, directivas o construcciones del lenguaje que hagan falta para:
 - Crear y terminar procesos (hebras). Si se crean de forma estática, enrolar o desenrolar procesos en el grupo que va a cooperar en el cálculo.
 - Localizar paralelismo.
 - Asignar la carga de trabajo.
 - Comunicar y sincronizar.
- Ejemplo: cálculo de PI con OpenMP/C.

Con la cláusula de planificación, estamos haciendo la carga dinámica.

```

#include <omp.h>
#define NUM_THREADS 4
main(int argc, char **argv) {
    long double ancho,x, sum=0; int intervalos, i;
    intervalos = atoi(argv[1]);
    ancho = 1.0/(double) intervalos;
    omp_set_num_threads(NUM_THREADS); →Crear/Terminar
    #pragma omp parallel →Comunicar/sincronizar
    {
        #pragma omp for reduction(+:sum) private(x) \
                    schedule(dynamic) →Agrupar/Asignar
        for (i=0;i< intervalos; i++) {
            x = (i+0.5)*ancho; sum = sum + 4.0/(1.0+x*x);
        }
    }
    sum* = ancho;
}

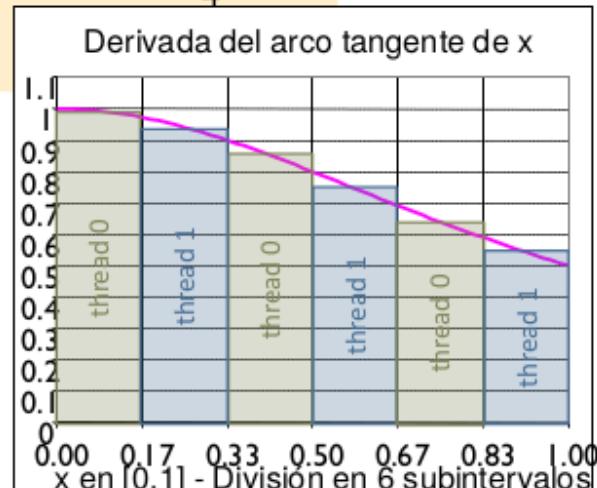
```

Localizar

- Asignación de tareas a 2 threads estática por turno rotatorio.

$$\left. \begin{array}{l} \arctg'(x) = \frac{1}{1+x^2} \\ \arctg(1) = \frac{\pi}{4} \\ \arctg(0) = 0 \end{array} \right\} \Rightarrow \int_0^1 \frac{1}{1+x^2} = \arctg(x)|_0^1 = \frac{\pi}{4} - 0$$

- PI se puede calcular por integración numérica.



- Ejemplo: cálculo de PI en MPI/C.
 - `MPI_Init()`: enrôle el proceso que lo ejecuta dentro del mundo MPI, es decir, dentro del grupo de procesos denominado `MPI_COMM_WORLD`.
 - `MPI_Finalize()`: se debe llamar antes de que un proceso enrolado en MPI acabe su ejecución.
 - `MPI_Comm_size(MPI_COMM_WORLD, &nproc)`: pregunta a MPI el número de procesos enrolados en el grupo, se devuelve en `nproc`.
 - `MPI_Comm_rank(MPI_COMM_WORLD, &iproc)`: se devuelve al proceso su identificador, `iproc`, dentro del grupo.

```

#include <mpi.h>
main(int argc, char **argv) {
    double ancho,x,lsum,sum; int intervalos,i,nproc,iproc;
    if(MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc); →Enrolar
    MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
    intervalos=atoi(argv[1]); →Localizar-Agrupar
    ancho=1.0/(double) intervalos; lsum=0;
    for (i=iproc; i<intervalos; i+=nproc) {
        x = (i+0.5)*ancho; lsum+= 4.0/(1.0+x*x);
    }
    lsum*= ancho; →Comunicar/sincronizar
    MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE,
               MPI_SUM,0,MPI_COMM_WORLD);
    MPI_Finalize(); →Desenrolar
}

```

- Evaluar prestaciones. Para ver la bondad, es decir, para ver si hemos hecho un buen código.

2.3 Lección 6. Evaluación de prestaciones en procesamiento paralelo.

2.3.1 Objetivos.

- Obtener ganancia y escalabilidad en el contexto de procesamiento paralelo
- Aplicar la ley de Amdahl en el contexto de procesamiento paralelo
- Comparar la ley de Amdahl y ganancia escalable.

2.3.2 6.1 Ganancia de prestaciones y escalabilidad.

6.1.1 Evaluación de prestaciones.

- **Medidas usuales.**
 - Tiempo de respuesta.
 - Real (wall-clock time, elapsed time) (/usr/bin/time).
 - Usuario, sistema, CPU time = user + sys.
 - Productividad.
- **Escalabilidad.** Evolución del incremento (ganancia) en prestaciones (tiempo de respuesta o productividad) que se consigue en el sistema conforme se añaden recursos.
- **Eficiencia.**
 - Relación prestaciones/prestaciones máximas.
 - Rendimiento = prestaciones/no_recursos.
 - Otras: Prestaciones/consumo_potencia, prestaciones/área_ocupada.

6.1.2 Ganancia en prestaciones. Escalabilidad.

Ganancia de prestaciones:

$$S(p) = \frac{\text{Prestaciones}(p)}{\text{Prestaciones}(1)} = \frac{T_S}{T_P(p)}$$

$$T_p(p) = T_C(p) + T_O(p)$$

$\text{Prestaciones}(p)$: prestaciones para la aplicación en el sistema multiprocesador con p procesadores.

$\text{Prestaciones}(1)$: prestaciones obtenidas ejecutando la versión secuencial en un sistema uniprocesador.

T_S : tiempo de ejecución (respuesta) del programa secuencial. Para obtener T_S se debería escoger el mejor programa secuencial para la aplicación.

$T_P(p)$: tiempo de ejecución del programa paralelo con p procesadores.

$T_O(p)$: tiempo de penalización.

Ganancia en velocidad (Speedup)

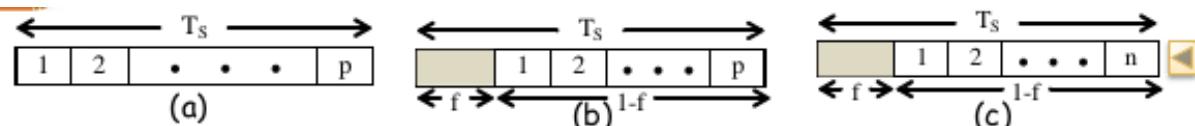
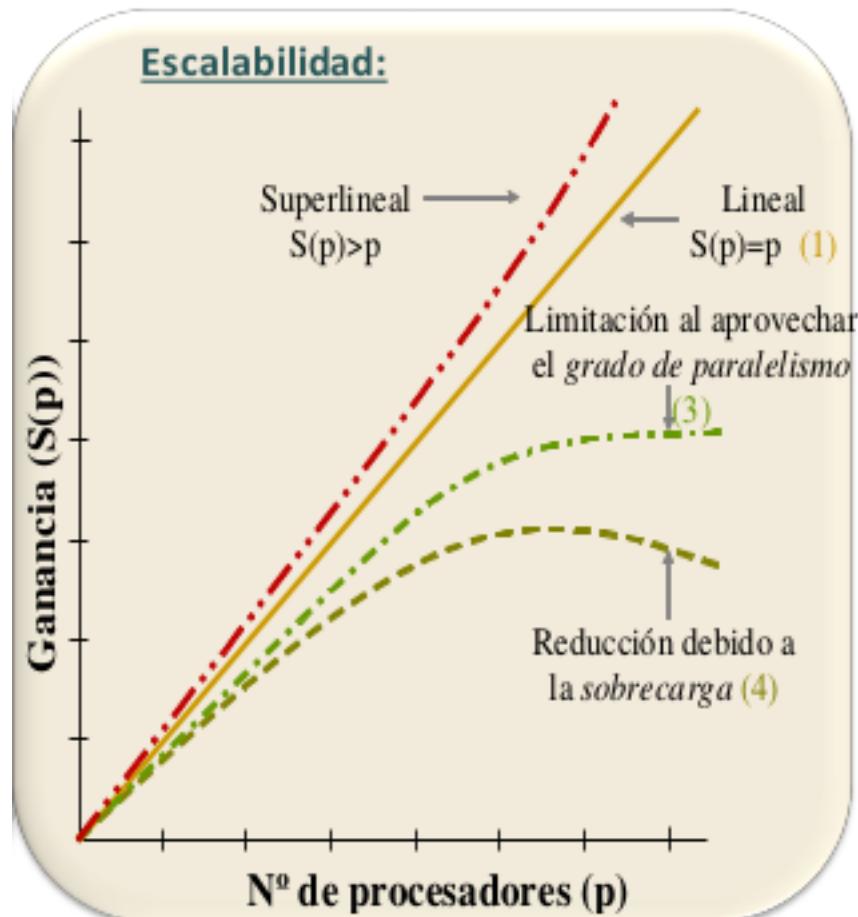
- Ganancia máxima de la eficiencia = 1
- Ganancia mínima de la eficiencia = $1/p$

$$T_p(p) = \frac{T_s}{p} \rightarrow S(p) = T_s / T_p(p) = \frac{T_s}{T_s/p} = p$$

- Sobrecarga (*overhead*):
 - Comunicación/sincronización.
 - Crear/terminar procesos/threads.
 - Cálculos o funciones no presentes en versión secuencial.
 - Falta de equilibrado.

$$E(p) = \frac{\text{Prest}(p)}{\text{PrestMax}(p)} = \frac{\text{Prest}(p)}{p \cdot \text{Prest}(1)} = \frac{S(p)}{p}$$

En la siguiente imagen, se mantiene constante el tamaño del problema (T_S).



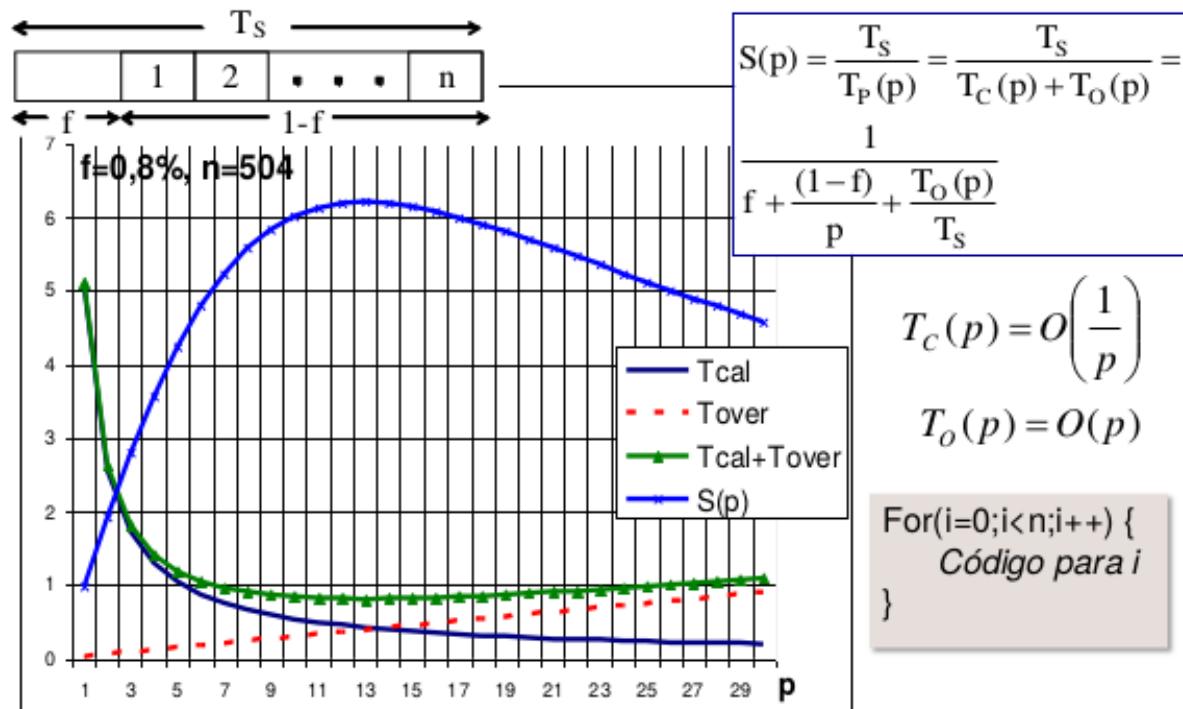
Modelo código	Fracción no paral. en T_S	Grado paralelismo	Overhead	Ganancia en función del número de procesadores p con T_S constante
(a)	0	ilimitado	0	$S(p) = \frac{T_S}{T_p(p)} = p$ Ganancia lineal (1)
(b)	f	ilimitado	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p \rightarrow \infty} \frac{1}{f}$ (2)
(c)	f	n	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p=n} \frac{1}{f + \frac{(1-f)}{n}}$ (3)
(b)	f	ilimitado	Incrementa linealmente con p	$S(p) = \frac{1}{f + \frac{(1-f)}{p} + \frac{T_O(p)}{T_S}} \xrightarrow{p \rightarrow \infty} 0$ (4)

- Número de procesadores óptimo:

$$S(p) = \frac{T_S}{T_P(p)} = \frac{T_S}{T_C(p) + T_O(p)} = \frac{1}{f + \frac{1-f}{p} + \frac{T_O(p)}{T_S}}$$

$$T_C(p) = O\left(\frac{1}{p}\right)$$

$$T_O(p) = O(p)$$

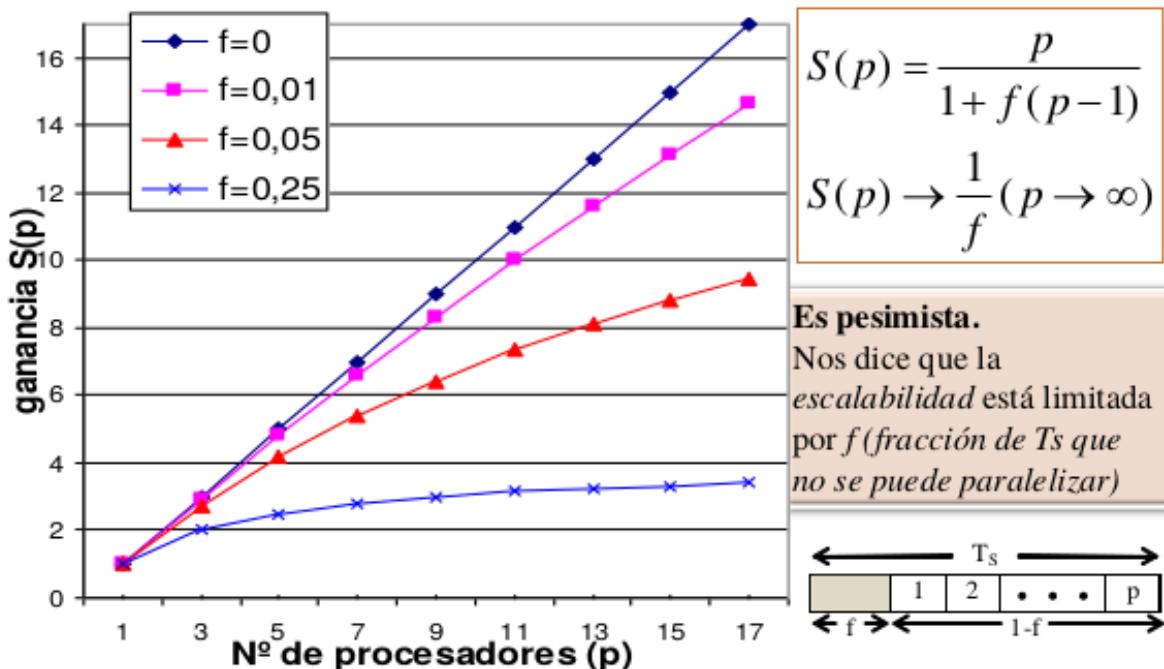


2.3.3 6.2 Ley de Amdahl.

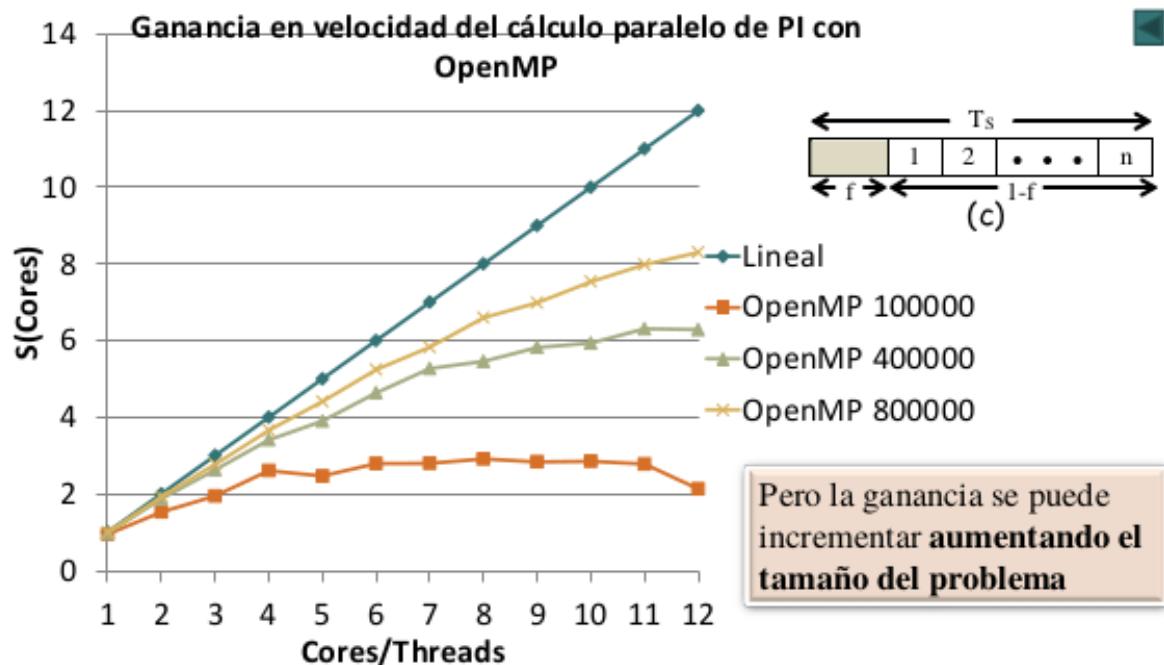
- Ley de Amdahl: la ganancia en prestaciones utilizando p procesadores está limitada por la fracción de código que no se puede parallelizar.

$$S(p) = \frac{T_S}{T_P(p)} \leq \frac{T_S}{f \cdot T_S + \frac{(1-f)T_S}{p}} = \frac{p}{1+f(p-1)} \rightarrow \frac{1}{f} (p \rightarrow \infty)$$

- S : incremento en velocidad que se consigue al aplicar una mejora. (parallelismo)
- p : Incremento en velocidad máximo que se puede conseguir si se aplica la mejora todo el tiempo. (número de procesadores)
- f : fracción de tiempo en el que no se puede aplicar la mejora. (fracción de t. no parallelizable)

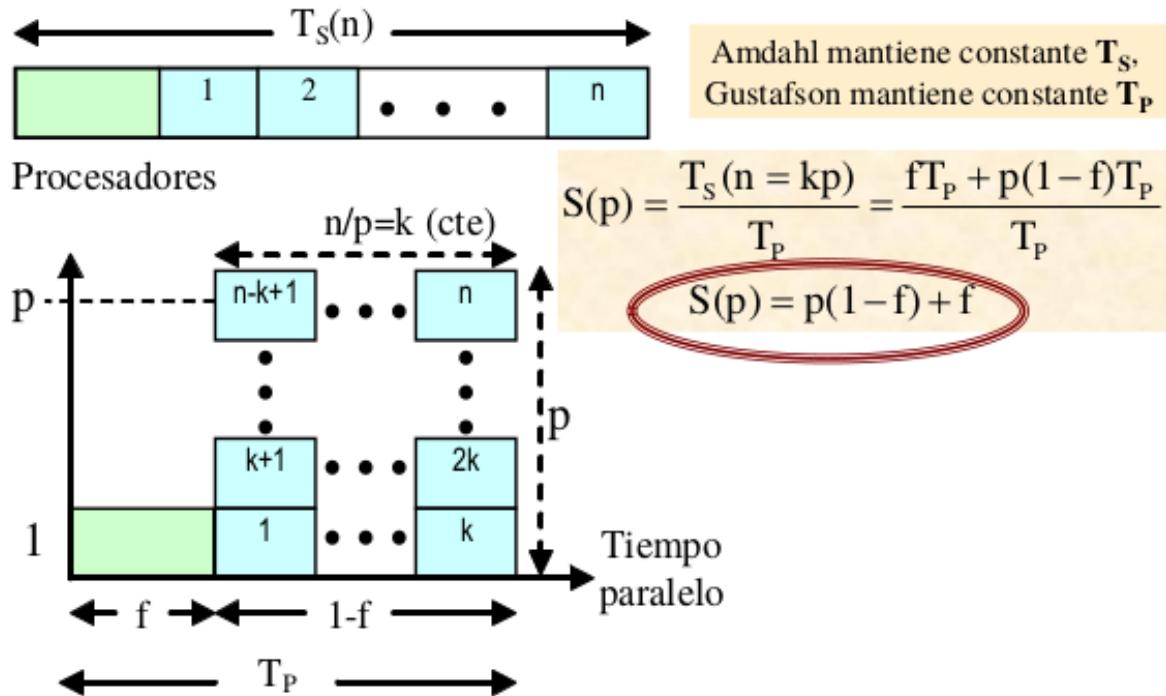


2.3.4 6.3 Ganancia escalable.



En la siguiente imagen, al incrementar el número de procesadores, se añade más trabajo y se incrementa el tamaño del problema (T_S), pero el tiempo paralelo se mantiene constante (T_p).

f : fracción de ejecución paralelo frente a la no paralelizable.



3 Bibliografía

Ortega, M. Anguita, A. Prieto. Arquitectura de Computadores, Thomson, 2005.