

# ARQUITECTURA DE COMPUTADORES

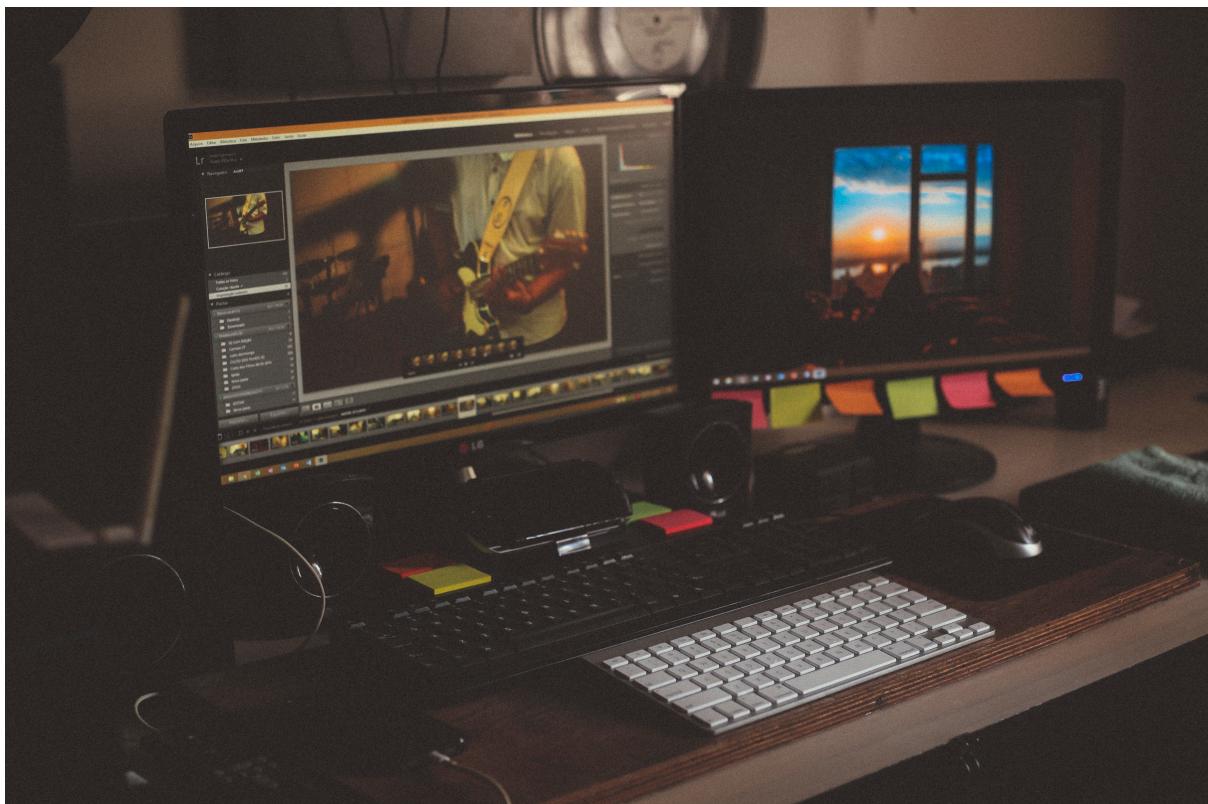


Photo by João Silas on unsplash.com

**Francisco Navarro Morales - GRG121**

Segundo curso del Grado de Ingeniería Informática  
Universidad de Granada  
curso 2016-2017

# Índice

<b>1. Tema 1: Arquitecturas paralelas: clasificación y prestaciones</b>	<b>2</b>
1.1. Clasificación del paralelismo implícito en una aplicación . . . . .	2
1.2. Evaluación de prestaciones . . . . .	3
1.2.1. Tiempo de CPU de un programa. . . . .	3
1.2.2. Medidas de la velocidad de procesamiento y benchmarks. . . . .	5
1.2.3. Estimar el tiempo de CPU . . . . .	6
1.2.4. Ganancia de velocidad y ley de Amdahl . . . . .	7
1.3. Cuestiones tema 1 . . . . .	8
1.4. Ejercicios tema 1 . . . . .	8

# 1 Tema 1: Arquitecturas paralelas: clasificación y prestaciones

## 1.1. Clasificación del paralelismo implícito en una aplicación

Para facilitar el estudio de los computadores hacemos uso de la abstracción. Así, a nivel hardware, diferenciamos nivel de componentes, nivel de circuitos electrónicos, nivel de lógica digital, niveles RT y niveles de sistema computador; y en cada uno de estos niveles se implementa algún tipo de paralelismo. Hay dos opciones para aprovechar el paralelismo en un sistema: **replicar** componentes del sistema, o **segmentar** el uso de componentes. Un ejemplo sería la réplica de unidades funcionales (paralelismo en el procesador) o la segmentación del propio procesador.

A nivel de software, podemos clasificar los distintos tipos de paralelismo posibles en una aplicación en función del nivel de abstracción dentro del código secuencial de un programa en el que podemos encontrar implícito el paralelismo:

- Paralelismo a nivel de funciones: pueden ejecutarse en paralelo si no hay dependencias de datos verdaderas (lectura después de escritura) entre ellas.
- Paralelismo a nivel de bucles: las iteraciones del bucle se pueden ejecutar en paralelo siempre y cuando se eliminen los problemas derivados de dependencias verdaderas.
- Paralelismo a nivel de operaciones: operaciones independientes se ejecutan en paralelo. Además, algunos procesadores de propósito específico (y algunos de propósito general) incluyen instrucciones completas de varias operaciones que evitan penalizaciones por dependencias verdaderas.
- Paralelismo a nivel de programas: se ejecutan en paralelo y pueden pertenecer o no a la misma aplicación y estar ejecutados o no por distintos usuarios.

Al paralelismo que se puede detectar en distintos niveles de código secuencial se le denomina **paralelismo funcional**. Además de esta clasificación, podemos llamar **paralelismo de tareas** a aquél que se extrae de la estructura de funciones de la aplicación o, por el contrario, **paralelismo de datos** al que se encuentra implícito en las operaciones con estructuras de datos (vectores y matrices). Así, el paralelismo de tareas está relacionado con el paralelismo a nivel de funciones y el de datos, con el paralelismo a nivel de bucle. Los procesadores que utilizan procesamiento SIMD (**arquitecturas multimedia**) aceleran el procesamiento vectorial aplicando la misma operación en paralelo a múltiples datos dentro de un registro. Otra clasificación posible es en función de la granularidad o magnitud de la tarea (número de operaciones) candidata a la paralelización. Esta clasificación tiene correspondencia con el paralelismo funcional. El grano fino se asocia generalmente a paralelismo entre operaciones o instrucciones; el grano medio a bloques funcionales lógicos, y el grano grueso al paralelismo entre programas.

El hardware se encarga de gestionar la ejecución de instrucciones. Por encima, el SO gestiona procesos y hebras. Cada proceso tiene su propia asignación de memoria y, en el caso de SO multihebras un proceso puede componerse de una o varias hebras (hilos). La principal diferencia entre hebras y procesos es que una hebra tiene su propia pila y contenido de registros (entre ellos el puntero de instrucciones), pero comparte el código, las variables globales y otros recursos (como archivos abiertos) con las hebras del mismo proceso. Así, las hebras se pueden crear y destruir en menor tiempo que los procesos y la comunicación, sincronización y conmutación es más rápida entre hebras que entre procesos.

En los procesadores ILP superescalares o segmentados la arquitectura extrae el paralelismo eliminando dependencias de datos falsas entre instrucciones y evitando problemas debidos a dependencias de datos, control y recursos. En estos procesadores se extrae el paralelismo implícito en las entradas dinámicamente (en tiempo de ejecución). Se puede incrementar el grado de paralelismo de las instrucciones con ayuda de

un buen compilador o con esfuerzo del programador. Se define el grado de paralelismo como el máximo número de entradas del conjunto que se pueden ejecutar en paralelo (siendo para los procesadores las entradas instrucciones). En arquitecturas ILP VLIW el paralelismo aprovechable está ya explícito. El análisis de dependencias en este caso es estático, ya que es el compilador el principal encargado de extraer el paralelismo. No obstante, la ayuda del programador puede incrementar el grado de concurrencia entre instrucciones aprovechado finalmente por la arquitectura.

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 aprovechable por arquitecturas SIMD o vectoriales, pero aún es difícil para los compiladores extraer paralelismo a nivel de función sin ayuda del programador; este, puede extraer paralelismo implícito en bucles o funciones definiendo hebras o procesos. La elección de hebras, procesos o ambos depende de la granularidad de las unidades de código independientes, de la posibilidad que ofrezca la herramienta de programación para definir hebras y procesos, de la arquitectura disponible para aprovechar el paralelismo y del sistema operativo disponible.

## 1.2. Evaluación de prestaciones

### 1.2.1. Tiempo de CPU de un programa.

El tiempo transcurrido desde que se lanza la ejecución de un programa y se tienen sus resultados es el denominado **tiempo de respuesta**. Este tiempo viene condicionado por el **tiempo de CPU**, que es el tiempo que el procesador emplea en ejecutar instrucciones máquina de su repertorio, ya sea en modo usuario o en modo kernel (SO). Como tanto los datos como las instrucciones necesarias para ejecutar los programas están en memoria, dentro del tiempo de respuesta también va incluido el tiempo de espera debido a E/S. No obstante, si nos centramos en el tiempo de CPU y consideramos el tiempo de E/S despreciable, podemos definir el tiempo de CPU como:

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

donde:

- $NI$  es el número de instrucciones máquina del programa que se ejecuta.
- $CPI$  es el número medio de ciclos por instrucción.  $CPI = \frac{\text{Ciclos totales del programa}}{NI}$
- $T_{ciclo}$  es el periodo de reloj del procesador, inverso a la frecuencia del procesador ( $f$ ).
- $CPI \times NI = \text{Ciclos totales del programa}$

$T_{ciclo}$  se obtiene de la frecuencia del procesador, que es conocida; mientras que  $NI$  y  $CPI$  se pueden obtener gracias a algunos contadores de eventos que poseen la mayoría de los procesadores actuales. Además,  $T_{CPU}$  se puede medir empíricamente si es necesario.

Para conseguir los mejores tiempos de  $CPU$  posibles, tenemos que tener en cuenta estos factores:

- Repertorio de instrucciones: condiciona  $NI$  y  $CPI$
- Compilador: condiciona  $NI$ ,
- Organización del computador: condiciona  $CPI$  y  $f$

- Prestaciones tecnológicas: condicionan  $f$

Como la organización del computador afecta tanto a  $CPI$  como a  $f$ , lo que verdaderamente tenemos en cuenta es el cociente  $\frac{CPI}{f}$ .

El principal problema que encontramos es la dificultad para modificar los parámetros de forma aislada (mejorar unos valores sin empeorar otros). Por ejemplo, de la comparación entre arquitecturas *RISC* (Reduced Instruction Set Computing) y *CISC* (Complex Instruction Set Computing) podemos extraer que las arquitecturas *CISC* (tendencia dominante hasta los 80) pretendían reducir el valor de  $NI$  con repertorios de instrucciones complejas pero provocaban un valor de  $CPI$  mayor, que era contrarrestado con el aumento de la frecuencia de los procesadores debido a las mejoras tecnológicas. Por su parte, el planteamiento *RISC* busca reducir el valor de  $CPI$  con repertorios de instrucciones sencillas y aprovechar los beneficios del diseño segmentado de los procesadores y el mejor rendimiento de los compiladores. El incremento de  $NI$  se puede contrarrestar también con el aumento de la frecuencia de reloj. En cualquiera caso, dada una arquitectura con un  $NI$  constante para un programa, se consiguen mejoras en el tiempo de ejecución disminuyendo  $CPI$  (aprovechando el paralelismo entre instrucciones, *ILP*, a través de la segmentación de cauce y otras técnicas que permiten procesar varias instrucciones por etapa), o bien aumentando la frecuencia del procesador (que se consigue gracias a los avances tecnológicos).

Para evaluar las prestaciones de un computador tenemos en cuenta:

- Tiempo de respuesta. Lo que tarda en procesar una entrada (trabajo, programa, instrucción...)
- Productividad. Entradas procesadas por unidad de tiempo.
- Funcionalidad. Tipos distintos de entradas que puede procesar.
- Expansibilidad. Posibilidad de ampliar la capacidad de procesamiento añadiendo bloques a la arquitectura existente.
- Escalabilidad. Posibilidad de ampliar el sistema sin que esto suponga una devaluación de las prestaciones.
- Eficiencia. Prestaciones/coste.

La importancia que se le da a cada una depende de lo que intentemos conseguir con ello, si priorizamos la organización de memoria entendemos por entradas los accesos a memoria; que están sujetos al ancho de banda (productividad), la latencia (tiempo de respuesta) y a la escalabilidad. Si priorizamos la computación, las entradas son los programas de aplicaciones, y nos interesa mejorar el tiempo de respuesta y la productividad.

Anteriormente hemos definido  $CPI$  como el número medio de ciclos por instrucción, algo así como

$$\frac{\sum_{i=1}^n NI_i \times CPI_i}{NI}$$

No obstante, si tenemos en cuenta el paralelismo entre instrucciones, podemos modificar el cálculo de  $CPI$  y obtener la expresión:

$$T_{CPU} = NI \times \frac{CPE}{IPE} \times T_{ciclo}$$

donde  $CPE$  es el número medio de ciclos por emisión (ciclos entre inicios de ejecución de instrucciones) e  $IPE$  el número medio de instrucciones por emisión. Así, podemos expresar el valor de  $CPI$  en función de las características de la microarquitectura en cuestión. Esto es, en un procesador no segmentado (en el que las instrucciones se ejecutan una a una)  $CPE$  sería igual al número medio de ciclos de instrucción

que se tarda en procesar la instrucción dado que hasta que no se termine de procesar una instrucción no se empieza a procesar la siguiente, e IPE sería 1 dado que las instrucciones se emiten una a una. Por el contrario, uno segmentado el valor máximo de CPE es igual a 1 dado que en cada ciclo podrían empezar a ejecutarse instrucciones, y por tanto,  $CPI = 1/IPE$ . Si el procesador segmentado sólo puede empezar a ejecutar una instrucción por ciclo, CPI = 1. Esta situación se tendría en un caso ideal, dado que en el procesador sementado no en todos los ciclos se puede empezar a ejecutar nuevas instrucciones (debido a dependencias de datos, de control, y las colisiones por acceso a recursos comunes), por ello CPE suele ser mayor que 1 y lo mismo ocurre con CPI. En el caso de procesadores superescalares o VLIW (Very Long Instruction Word) se pueden emitir varias instrucciones por ciclo, por lo que IPE es mayor que 1, es decir, que el valor de CPI de algunos procesadores podría llegar a ser menor que 1, y tanto menor cuanto más paralelismo a nivel de instrucción aproveche. Si tenemos en cuenta todo esto, nos quedaría la expresión:

$$T_{CPU} = NI \times CPI \times T_{ciclo} = \frac{N_{oper}}{OPI} \times \frac{CPE}{IPE} \times T_{ciclo}$$

### 1.2.2. Medidas de la velocidad de procesamiento y benchmarks.

Además del tiempo de CPU, existen otras medidas para evaluar las prestaciones como son los MIPS y los MFLOPS; millones de instrucciones por segundo y millones de operaciones en coma flotante por segundo.

$$MIPS = \frac{NI}{T_{CPU} \times 10^6} = \frac{NI}{NI \times CPI \times T_{ciclo} \times 10^6} = \frac{1}{CPI \times T_{ciclo} \times 10^6} = \frac{f}{CPI \times 10^6}$$

Como estas medidas empiezan a resultar pequeñas a medida que construimos computadores más rápidos, también se empiezan a utilizar los GIPS, dónde el número de instrucciones ejecutado se expresa en términos de  $10^9$ . En el caso de los MFLOPS, aparecen no solo los GFLOPS, sino también los TFLOPS con  $10^{12}$ , los PFLOPS con  $10^{15}$ , y los EFLOPS (exaFLOPS), con  $10^{18}$ .

Hay que tener en cuenta que estas medidas dependen del repertorio de instrucciones del procesador. Un programa codificado con instrucciones RISC generará un número mayor de instrucciones máquina que uno codificado con instrucciones CISC; y sin embargo, es muy probable que el tiempo de ejecución del RISC sea igual o incluso mejor que el del CISC, por lo que esta medida que tiene en cuenta las instrucciones ejecutadas no nos aporta información sobre cual tiene mejores prestaciones. Por tanto, tenemos que entender que MIPS y MFLOPS miden la velocidad con la que cada procesador ejecuta las instrucciones **de su repertorio**. Si comparamos computadores con el mismo repertorio, tener información sobre cual ejecuta más instrucciones por segundo si nos permite comparar prestaciones.

Estos parámetros también sirven para calcular la velocidad pico de los procesadores. como  $CPI = \frac{1}{IPC}$  (Ciclos por instrucción es inverso a instrucciones por ciclo), el valor más pequeño de CPI correspondería al máximo número de instrucciones que el procesador puede ejecutar por ciclo, IPC. Conociendo entonces la frecuencia del procesador, el valor pico para IPC nos permite obtener los MIPS pico del procesador. El valor pico de IPC se puede deducir de las características de la propia microarquitectura. Por ejemplo, si un procesador superescalar que funciona a una frecuencia de 2GHz tiene recursos para completas dos instrucciones por ciclo como máximo, tendremos que  $IPC = 2$  o lo que es lo mismo,  $CPI = 0,5$ ; y por lo tanto proporcionará 4GIPS.

Para calcular el número de instrucciones por segundo de un computador se han establecido unos programas por convenio llamados Benchmarks. Pese a que son muchos y distintos (en función del tipo de instrucciones que se quiera tener en cuenta, ya que no tienen los mismos requisitos un computador para

cálculos relacionados con BigData que uno para cálculo científico, etc..), suponen un estándar a la hora de comparar computadores.

### 1.2.3. Estimar el tiempo de CPU

Si despreciamos el tiempo dedicado a entrada/salida y la existencia de fallos de caché, es posible hacer una estimación del tiempo de CPU considerando que el procesador funciona a máximo rendimiento. Se utiliza el valor pico de IPC, que corresponde con el valor más pequeño posible de CPI, y a partir del número de instrucciones máquina del programa, NI, y de la frecuencia a la que funciona el procesador, se obtendría el tiempo de CPU **mínimo** que necesita el programa.

Ahora bien, si tenemos información de la jerarquía de memoria del computador, podemos considerar que existe solapamiento entre el tiempo de ejecución de instrucciones en el procesador y el tiempo de carga o escritura de datos cuando se producen fallos de caché. En el **mejor de los casos** se consideraría que existe un solapamiento total entre el tiempo consumido por las instrucciones del procesador (tiempo mínimo considerando que no hay fallos de caché), y el tiempo de acceso a memoria principal según la jerarquía de memoria y el patrón de acceso a los datos del programa (que genere mayor o menor cantidad de fallos). Así, el tiempo de ejecución mínimo del programa será el máximo de estos dos tiempos.

$$T_{ejecucion\ minimo} = \max(T_{procesador}, T_{memoria})$$

Poco solapamiento:



Máximo solapamiento:



Para estimar el tiempo medio de acceso a memoria hay que tener en cuenta las tasas de fallos que se producen y los tiempos de acceso de los distintos niveles de caché y de memoria principal. Por ejemplo, para una jerarquía de dos niveles en que el primero corresponde a la memoria caché (L1):

$$t_{memoria} = a_1 \times t_1 + (1 - a_1) \times (t_1 + t_M)$$

Dónde  $a_1$  es la tasa de aciertos de la caché,  $t_1$  el tiempo de acceso a la caché y  $t_M$  el tiempo de acceso a memoria principal. Ahora bien, esta fórmula no contempla la necesidad de actualizar la memoria principal con los datos que han sido modificados en caché. Si tuviéramos esto en cuenta nos quedaría:

$$t_{memoria} = a_1 \times t_1 + (1 - a_1) \times (t_1 + t_M + P_{reemplazo} \times t_{linea})$$

donde  $P_{reemplazo}$  es la probabilidad de que, cuando se produzca un fallo de caché, haga falta reemplazar alguna línea para traer a caché otra necesaria; y  $t_{linea}$  es el tiempo necesario para actualizar en memoria principal la línea que ha sido modificada.

Hay que tener en cuenta que en esta fórmula sólo se tiene en cuenta la actualización de datos de memoria principal cuando es necesaria porque no hay espacio en caché suficiente para traer nuevas líneas y hay

que reemplazar alguna línea. Si la actualización se produce en otro momento para mantener la coherencia con otros procesadores no se tiene en cuenta excepto en que a la hora de reemplazar podría darse el caso de que una línea de caché estuviera ya actualizada en memoria principal. Los tiempos de acceso a memoria y de actualización de línea vienen determinados por las características de los buses de memoria y la tecnología de los circuitos de memoria DRAM. Los ciclos necesarios para realizar una transferencia entre el procesador y memoria se denominan ciclos de bus.

#### 1.2.4. Ganancia de velocidad y ley de Amdahl

Para localizar y evitar los cuellos de botella del computador debemos medir de alguna manera las ‘mejoras’ de velocidad que podemos conseguir; para ello se emplea el parámetro de ganancia de velocidad,  $S_p$ , la velocidad de un computador antes y después de mejorar sus recursos.

$$S_p = \frac{V_p}{V_1} = \frac{(W/T_p)}{(W/T_1)} = \frac{T_1}{T_p}$$

dónde  $V_1$  es la velocidad antes de la mejora,  $V_p$ , después de la mejora,  $W$ , la carga de trabajo y  $T_i$  los tiempos antes y después de la mejora. La ley de Amdahl establece una cota superior a la ganancia de velocidad que se puede conseguir al mejorar alguno de los recursos del computador en un factor  $p$ , según la frecuencia en que se utiliza ese recurso en la máquina de partida:

$$S_p \leq \frac{1}{1 + f \times (p - 1)}$$

dónde  $f$  es la fracción de tiempo de ejecución antes de aplicar la mejora en un recurso en la que no se utiliza dicho recurso. Así, si  $f = 1$  (el recurso mejorado no se utiliza),  $S_p$  es menor o igual a 1 (no hay mejora). Sólo si  $f = 0$  (el recurso se utiliza todo el tiempo) se podría alcanzar un valor igual a  $p$ .

Distinguimos dos espacios temporales, uno en el que se utiliza el recurso ( $f$ ) y otro en el que no,  $(1 - f)$ , luego tenemos que:

$$T_p = f \times T_1 + \frac{(1 - f) \times T_1}{p}$$

; porque el recurso mejora el tiempo de respuesta en un factor  $p$ . Luego la ganancia queda:

$$\begin{aligned} S_p &= \frac{T_1 \times p}{f \times T_1 \times p + (1 - f) \times T} \\ S_p &= \frac{p}{fp + 1 - f} = \frac{p}{f(p - 1) + 1} \end{aligned}$$

Luego se ha obtenido la fórmula de la ley de Amdahl. Ahora sería interesante advertir que:

$$\lim_{p \rightarrow \infty} \frac{p}{f(p - 1) + 1} = \frac{1}{f}$$

Es decir que la ganancia posible mejorando un componente del sistema está limitada por el tiempo en que ese componente no se está usando. Por lo que da igual cuánto mejoremos un elemento; si este se usa poco, la mejora final será muy pequeña.

Del mismo modo:

$$\lim_{f \rightarrow 0} \frac{p}{f(p-1) + 1} = p$$

Lo que nos indica que es imposible conseguir mejoras superiores a  $p$ , por mucho que se utilice el elemento mejorado (aunque se utilice todo el tiempo).

### 1.3. Cuestiones tema 1

**Cuestión 1:** Indique cómo se podría aprovechar un computador MISD para acelerar la determinación de si  $n$  números son primos o no. Considere que se conocen los  $M$  números primos entre 1 y el máximo valor que puede tener un número de entrada.

**Cuestión 2:** Indique cuál es la diferencia fundamental entre una arquitectura CC-NUMA y una arquitectura SMP.

**Cuestión 3:** ¿Cuándo diría que un computador es un multiprocesador y cuándo que es un multicomputador? Decimos que un computadores es un multiprocesador si todos los procesadores comparten el mismo espacio de direcciones; diremos que se trata de un multicomputador si cada procesador tiene su espacio de direcciones propio.

**Cuestión 4:** ¿Un CC-NUMA escala más que un SMP? ¿Por qué? La principal diferencia entre un SMP y un CC-NUMA es que, pese a que en ambos toda la memoria es compartida por todos los procesadores, en el CC-NUMA la memoria se divide en fragmentos que serán cada uno más próximo a un procesador; es decir, aunque todos los procesadores pueden acceder a todos los fragmentos de memoria, cada procesador accederá más rápido a los fragmentos que le corresponden.

**Cuestión 12:** En la Lección 2 de AC se han presentado diferentes criterios de clasificación de computadores y en el Seminario 0 de prácticas se ha presentado atcgrid. Clasifique atcgrid, sus nodos, sus encapsulados y sus núcleos dentro de la clasificación de Flynn y dentro de la clasificación que usa como criterio el sistema de memoria. Razone su respuesta.

**Cuestión 13:** En la Lección 1 de AC se han presentado diferentes criterios de clasificación del paralelismo implícito en una aplicación y en el Seminario 0 de prácticas se ha presentado atcgrid. ¿Qué tipos de paralelismo aprovecha atcgrid? Razone su respuesta.

### 1.4. Ejercicios tema 1

**Ejercicio 1:** En el código de prueba (benchmark) que ejecuta un procesador no segmentado que funciona a 300 MHz, hay un 20 % de instrucciones LOAD que necesitan 4 ciclos, un 10 % de instrucciones STORE que necesitan 3 ciclos, un 25 % de instrucciones con operaciones de enteros que necesitan 6 ciclos, un 15 % de instrucciones con operandos en coma flotante que necesitan 8 ciclos por instrucción, y un 30 % de instrucciones de salto que necesitan 3 ciclos.  
 (a) ¿Cuál es la ganancia que se puede obtener por reducción a 3 ciclos de las instrucciones con enteros?

(b) ¿cuál es la ganancia que se puede obtener por reducción a 3 ciclos de las instrucciones en coma flotante?

Sin ningún tipo de reducción tendríamos, en total:

$$\frac{4 * 20 + 3 * 10 + 6 * 25 + 8 * 15 + 3 * 30}{100} = 4,70 \text{ ciclos.}$$

Si realizamos una reducción a 3 ciclos de las instrucciones con enteros, tendríamos:

$$\frac{4 * 20 + 3 * 10 + 3 * 25 + 8 * 15 + 3 * 30}{100} = 3,95 \text{ ciclos.}$$

Si realizamos una reducción a 3 ciclos de las instrucciones de coma flotante, tendríamos:

$$\frac{4 * 20 + 3 * 10 + 6 * 25 + 3 * 15 + 3 * 30}{100} = 3,95 \text{ ciclos.}$$

Entonces, en ambos casos estamos reduciendo 0.75 ciclos, es decir, estamos reduciendo un 15,957 % de los ciclos ejecutados en la prueba.

**Ejercicio 2:** Un circuito que implementaba una operación en un tiempo de  $T_{op}=450$  ns se ha segmentado mediante un cauce lineal con cuatro etapas de duración  $T_1=100$  ns,  $T_2=125$  ns,  $T_3=125$  ns y  $T_4=100$  ns respectivamente, separadas por un registro de acople que introduce un retardo de 25 ns.

(a) ¿Cuál es la máxima ganancia de velocidad posible? ¿Cuál es la productividad máxima del cauce?

(b) ¿A partir de qué número de operaciones ejecutadas se consigue una productividad igual al 90 % de la productividad máxima?

Dividimos las operaciones en un cauce de 4 etapas,  $T_1, T_2, T_3$  y  $T_4$ , tenemos, para 4 instrucciones:

		$I_4$	$T_1$	$T_2$	$T_3$	$T_4$	
	$I_3$	$T_1$	$T_2$	$T_3$	$T_4$		
	$I_2$	$T_1$	$T_2$	$T_3$	$T_4$		
$I_1$		$T_1$	$T_2$	$T_3$	$T_4$		
		$C_1$	$C_2$	$C_3$	$C_4$	$C_5$	$C_6$
							$C_7$

Como las etapas más lentas duran 125ns ( $C_i$ ), y el retardo de acoplamiento es de 25ns, tendremos un “ciclo de reloj” de 150ns en cada una de las etapas  $T_i$ . Esto significa que, para ejecutar cuatro instrucciones como en la figura, harían falta  $7 * 150\text{ns} = 1113\text{ns}$ ; mientras que, sin segmentación, harían falta  $450\text{ns} * 4 = 1800\text{ns}$ , luego se estarían ahorrando 687ns, que son un 28 % del tiempo sin segmentación. Entonces ¿cuál es la productividad máxima del cauce? es decir, ¿para cual sería el máximo número de operaciones a partir del cual no se mejoran los resultados? Sea  $k$  el número de instrucciones que se ejecutan y  $n$  el número de etapas:

$$AceleracionEjemplo = \frac{4T}{7T/4} = \frac{16T}{7T} = 2,28$$

$$AceleracionIdeal = \frac{kT}{(n - 1 + k)T/n} = \frac{nkT}{T(n - 1 + k)}$$

Cuando  $k \rightarrow \infty$ ,  $AceleracionIdeal = n$

Es decir, la ganancia máxima de velocidad posible es el número de etapas de fragmentación, 4 en este caso; y la productividad máxima del cauce sería

$$\frac{k \text{ instrucciones}}{(4 - 1 + k)450\text{ns}/4} = \frac{k}{112,5(3k)} = \frac{1}{337,5} \text{instrucciones/ns} = 0,002962 \text{ instrucciones/ns}$$

El 90 % de la productividad máxima (4), sería 3,6, entonces:

$$\frac{4k}{3 + k} = 3,6 \leftrightarrow 10,8 + 3,6k = 4k \leftrightarrow 0,4k = 10,8 \leftrightarrow k = \frac{10,8}{0,4} = 27$$

luego, a partir de 27 instrucciones se consigue un 90 % de la productividad máxima.