

# Arquitectura de Computadores



Los Del DGIIM, [losdeldgiim.github.io](https://losdeldgiim.github.io)

Doble Grado en Ingeniería Informática y Matemáticas  
Universidad de Granada



Esta obra está bajo una Licencia Creative Commons Atribución-NoComercial-SinDerivadas 4.0 Internacional (CC BY-NC-ND 4.0).

Eres libre de compartir y redistribuir el contenido de esta obra en cualquier medio o formato, siempre y cuando des el crédito adecuado a los autores originales y no persigas fines comerciales.

# Arquitectura de Computadores

Los Del DGIIM, [losdeldgiim.github.io](https://losdeldgiim.github.io)

José Juan Urrutia Milán  
Arturo Olivares Martos

Granada, 2023-2024



# Índice general

<b>1. Arquitecturas Paralelas</b>	<b>5</b>
1.1. Clasificación del paralelismo implícito en una aplicación . . . . .	5
1.1.1. Objetivos . . . . .	5
1.1.2. Niveles y tipos de paralelismo . . . . .	5
1.1.3. Unidades de ejecución . . . . .	9
1.1.4. Implementación del paralelismo . . . . .	9
1.1.5. Detección y extracción del paralelismo . . . . .	10
1.2. Clasificación de arquitecturas paralelas . . . . .	11
1.2.1. Objetivos . . . . .	11
1.2.2. Computación distribuida . . . . .	11
1.2.3. Clasificación según el mercado . . . . .	12
1.2.4. Clasificación de Flynn o de flujos . . . . .	13
1.2.5. Clasificación según el paralelismo aprovechado . . . . .	18
1.3. Evaluación de prestaciones . . . . .	19
1.3.1. Objetivos . . . . .	19
<b>2. Relaciones de Problemas</b>	<b>21</b>
2.1. Arquitecturas Paralelas . . . . .	21
2.2. Programación paralela . . . . .	29



# 1. Arquitecturas Paralelas

## 1.1. Clasificación del paralelismo implícito en una aplicación

### 1.1.1. Objetivos

Como items a conocer en esta sección, destacamos:

- Conocer las clasificaciones usuales del paralelismo implícito en una aplicación. Distinguir entre paralelismo de tareas y paralelismo de datos.
- Distinguir entre las dependencias RAW, WAW y WAR.
- Distinguir entre *thread* y proceso.
- Relacionar el paralelismo 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. Niveles y tipos de paralelismo

En una aplicación, podemos encontrar distintos niveles de paralelismo. Para facilitar su comprensión, trataremos de clasificarlos en esta parte inicial de la asignatura. Comenzaremos por marcar varias capas de abstracción que se siguen a la hora de desarrollar la aplicación, lo que los facilitará marcar el paralelismo dentro de esta.

Podemos considerar que un programa está compuesto de funciones, las cuales a su vez están compuestas de bloques de código en la que abundan los bucles (para simplificar esto, diremos que las funciones están compuestas de bucles). Los cuales están basados en operaciones. Asimismo, puede que nuestra aplicación esté compuesta por distintos programas (como en el caso de LibreOffice de LibreOffice Writer, LibreOffice Calc, ...). Por todo esto, nos es natural tratar de clasificar el paralelismo de una aplicación en función de distintos niveles, los cuales serán:

- Nivel de programas.
- Nivel de funciones.
- Nivel de bucles (de bloques).
- Nivel de operaciones.

En general, el paralelismo lo podremos encontrar en distinta granularidad (en mayor o menor medida) en relación al nivel en el que nos encontremos. Para detectar mejor este grado de paralelismo, es cómodo tener una clara distinción del tipo de paralelismo (como estamos haciendo), lo que facilita la tarea del programador y del compilador. Destacamos la ventaja de poder manipular el código secuencial (que ya sabemos manejar) en código con funcionalidades paralelas, lo que nos libra de tener que conocer tecnologías nuevas para poder implementar paralelismo en nuestras aplicaciones.

A continuación, justificamos los niveles ya elegidos, junto con ejemplos de paralelismo en cada uno de ellos:

### Nivel de programas

Los diferentes programas que intervienen en una aplicación (o incluso en diferentes aplicaciones) se pueden ejecutar en paralelo, debido a que es poco probable que existan dependencias entre ellos.

### Nivel de funciones

Un nivel de abstracción más bajo; las funciones llamadas en un programa se pueden ejecutar en paralelo, siempre que no haya dependencias (riesgos) inevitables entre ellos, como dependencias de datos verdaderas (RAW). Como ejemplo, recomendamos la familiarización de la directiva `#pragma omp parallel sections` de OpenMP de la Sesión 1 de Prácticas, donde descubrimos el paralelismo a nivel de funciones de forma explícita.

### Nivel de bucles (de bloques)

Una función puede estar basada en la ejecución de uno o varios bucles. En muchas ocasiones, el código que se encuentra dentro de un bucle no está íntegramente asociado con la iteración en sí; sino que deseamos que una cierta tarea se ejecute un cierto número de veces. Se pueden ejecutar en paralelo las iteraciones de un bucle, siempre que eliminen los problemas derivados de las dependencias de datos verdaderas (RAW).

### Nivel de operaciones

En este nivel se extrae el paralelismo disponible entre operaciones. Las operaciones independientes se pueden ejecutar en paralelo. Por otra parte, podemos encontrar instrucciones compuestas de varias operaciones que se aplican en secuencial mismo tipo de datos de entrada. Por ejemplo, la instrucción `mac` nos permite realizar una suma tras una multiplicación. En este nivel se puede detectar la posibilidad de usar instrucciones compuestas como la ya mencionada.

A esta clasificación del paralelismo que se puede detectar en distintos niveles de un código secuencial se le denomina *paralelismo funcional*. Por otra parte, podemos hablar de *paralelismo de tareas* y de *paralelismo de datos*.

### Paralelismo de tareas

En inglés, Task Level Parallelism (TLP). Este paralelismo se encuentra extrayendo la estructura lógica de funciones de la aplicación. Esta estructura está formada



por las funciones, siendo las conexiones entre ellas el flujo de datos entre funciones. El paralelismo a nivel de funciones antes descrito en el paralelismo funcional equivale al paralelismo de tareas.

### Paralelismo de datos

En inglés, Data Level Parallelism (DLP). El paralelismo de datos se encuentra implícito en las operaciones con estructuras de datos (como vectores y matrices). Las operaciones vectoriales y matriciales engloban operaciones con diversos escalares, las cuales se pueden realizar en paralelo. Como estas operaciones se suelen implementar por bucles, decimos que el paralelismo de datos es equivalente al paralelismo a nivel de operaciones en el paradigma del paralelismo funcional. Por ejemplo, contamos con las instrucciones SIMD (se desarrollarán próximamente), que con una instrucción puede manipular múltiples datos. Un ejemplo de instrucción SIMD es la implementación de una instrucción que pueda sumar dos vectores de datos enteros.

Por ejemplo, si tenemos una aplicación que nos permite decodificar el formato de imagen JPEG a formato RGB para imprimir en pantalla, podemos encontrar paralelismo de tareas al tener distintos módulos que realizan cada uno de los pasos intermedios para realizar dicha transformación; mientras que disponemos de paralelismo a nivel de datos en las operaciones, al tener instrucciones que nos permitan sumar (con una sola instrucción) dos vectores.

### Granularidad

El paralelismo también puede clasificarse en función de la granularidad de la tarea a realizar. Esto es, de la magnitud del número de operaciones a realizar. Esta se suele hacer corresponder con los distintos niveles de paralelismo funcional anteriormente desarrollado. Ilustramos esta relación uno a uno en la siguiente enumeración:

- Grano grueso: Nivel de programas.
- Grano medio: Nivel de funciones.
- Grano fino-medio: Nivel de bucles.
- Grano fino: Nivel de operaciones.

### Dependencias de datos

Constantemente estamos haciendo alusión a las dependencias de datos, pero no nos hemos parado a plantear cuando una sección de código  $B_2$  presenta dependencias de datos con respecto a un bloque de código  $B_1$ . Para que se produzca una dependencia de datos entre ellos:

- Deben hacer referencia a una misma variable (una misma posición de memoria).
- Un bloque de código debe aparecer en la secuencia de código antes que el otro.

Una vez que conocemos que existe una dependencia de datos entre dos bloques de código nos surge la cuestión de si cualquier dependencia es igual de importante, de si hay dependencias evitables y de si hay otras que no lo son. Respondemos a todo tipando las dependencias de datos:

### RAW (Read After Write)

También llamada dependencia verdadera, sucede cuando tratamos de leer una variable (equivalentemente, posición de memoria) después de haberla modificado (de haberla escrito). Recordemos que nos encontramos en el paradigma de la paralelización: tratamos de hacer esto de forma paralela, luego puede que un ente encargado de leer la variable lo haga antes que el encargado de modificarla, haciendo invisible dicha modificación (no existente cuando se leyó) y causando condiciones de carrera junto con un posible mal funcionamiento del programa (así como de romper el esquema determinista de este). Podemos ver un ejemplo de RAW en el siguiente ejemplo:

```
int a = b * c;  
int d = a + c;
```

Tenemos en la segunda línea el uso (lectura) de la variable `a`, tras modificarla (escribir en ella) en la primera línea. Si empleamos paralelismo puede suceder que se ejecute la segunda línea antes que la primera, provocando condiciones catastróficas. De hecho, cuando esto se haga, la variable `a` no estará ni siquiera inicializada.

### WAW (Write after Write)

También llamada anti-dependencia, sucede cuando tratamos de modificar una variable por segunda vez (después de haberla modificado ya). Esto puede plantear, al igual que explicábamos en RAW, condiciones de carrera. Mostramos un ejemplo a continuación:

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

Donde en la primera y tercera línea modificamos el valor de `a`. Sin embargo, esta dependencia es evitable, ya que si cambiamos el nombre de la variable (empleamos una dirección de memoria distinta), evitamos la dependencia. Por tanto, a esta dependencia también se le llama dependencia de nombre, al no ser una dependencia de datos real.

### WAR (Write after Read)

También llamada dependencia de salida, sucede cuando tratamos de escribir en una variable tras leer de ella. Esto también puede provocar condiciones de carrera, tal y como vemos en el siguiente ejemplo:

```
b = a + 1;  
a = d + e;
```

Donde en la primera línea leemos `a` y en la segunda modificamos su valor. Sin embargo, esta dependencia también recibe el nombre de dependencia de nombre, ya que puede solucionarse con un sencillo cambio de nombre, por lo que no se trata de una dependencia de datos real. Cabe destacar que esto lo suele realizar de forma automática el compilador.

### 1.1.3. Unidades de ejecución

El hardware es el encargado de la administración y ejecución de las instrucciones, mientras que a nivel superior nos encontramos con el SO, haciéndolo (no en el sentido que estás pensando) con las hebras y los procesos. Cada proceso en ejecución tiene su propia asignación de memoria. Los SO multihebra permiten que un proceso se conforme por una o varias hebras (o hilos). Cada hebra tiene su propia pila y banco de registros, mientras que comparte con sus hermanas la memoria que les oferta el proceso. Esto permite que las hebras puedan crearse, destruirse y comunicarse entre ellas de una forma más rápida que los procesos. Todo esto permite que las hebras dispongan de una menor granularidad que estos.

Esta sección nos ha servido para repasar entes que nos permiten hacer explícito el paralelismo, los cuales simplificarán el diseño de las aplicaciones, al ser las hebras y procesos automáticamente gestionadas por el sistema operativo; y las instrucciones por la arquitectura.

### 1.1.4. Implementación del paralelismo

A lo largo de este documento hemos hecho referencia en varias ocasiones al paralelismo implícito y explícito, sin nunca pararnos a desarrollar de qué estamos hablando. Es ahora la ocasión de hacerlo.

#### Paralelismo implícito.

Se trata de aquellas acciones que automáticamente se llevan a cabo (ya sea gracias al hardware, sistema operativo o compilador) de forma paralela.

#### Paralelismo explícito.

Se trata de aquellas acciones que deseamos que se hagan de forma paralela, y que obligamos a ello de forma explícita, como por ejemplo, con la ayuda de una API en el caso de las prácticas con OpenMP.

Esta diferencia la comentaremos en la siguiente subsección, que será fácil de comprender junto con el desarrollo de las prácticas.

Hecha esta distinción, comenzamos ahora sí con esta subsección, en la que podemos cómo se implementa el paralelismo implícito, así como el explícito, de una forma superficial. Además, será necesario indicar las especificaciones hardware requeridas para llevar esto a cabo (llamadas arquitecturas paralelas). Usaremos el paralelismo funcional, ya que para eso lo hemos desarrollado al inicio.

#### Nivel de programas

El paralelismo entre programas se implementa mediante diversos procesos: en el momento que se ejecuta un nuevo programa, se crea el programa asociado a él, y ya sólo dependerá del sistema operativo el llevar a cabo su paralelización

con el resto de procesos<sup>1</sup>. Para poder implementar este tipo de paralelismo, es necesario disponer de un multiprocesador o multicomputador.

### Nivel de funciones

El paralelismo a este nivel puede extraerse para realizarse a nivel de procesos (si la función realmente lo requiere) o de hebras, de forma que cada hebra (o proceso) ejecute una o varias funciones. Para ello, necesitaremos de un multiprocesador y, en caso de requerir hebras, será conveniente que este sea multihebra (o en su defecto, contar con una biblioteca de hebras, aunque esto es menos recomendable).

### Nivel de bucles

Este se puede realizar a nivel de procesos o hebras, tal y como se hacía en el nivel anterior. Sin embargo, el paralelismo a este nivel también puede implementarse con instrucciones en el caso de, por ejemplo, sumas de vectores. Para ello, debemos contar con un multiprocesador (a poder ser, multihebra) y para el último caso considerado, una arquitectura SIMD que permite realizar trabajos similares con vectores y, en general, estructuras de datos.

### Nivel de operaciones

El paralelismo entre operaciones se puede aprovechar en arquitecturas con paralelismo a nivel de instrucción (ILP), ejecutando en paralelo las instrucciones asociadas a operaciones independientes. Para ello, es claro que necesitamos arquitecturas ILP.

## 1.1.5. Detección y extracción del paralelismo

En los procesadores ILP superescalares o segmentados la arquitectura en sí misma extrae paralelismo (o como nosotros hemos llamado, implementa paralelismo implícito). Para ello, eliminan dependencias de datos falsas (no del tipo RAW) entre instrucciones y evitan problemas debidas a dependencias de datos, de control y de recursos.

Además, el grado de paralelismo de las instrucciones se puede incrementar con las ayudas del compilador y del programador. En general, se puede definir el grado de paralelismo de un conjunto de entradas a un sistema como el número máximo de entradas del conjunto que se pueden ejecutar en paralelo.

A continuación, para cada tipo de paralelismo, tratamos de explicar la extracción del paralelismo. Esto es, explicar qué ente lo detecta, cómo se implemente y en qué unidad se ejecuta. En este caso, la granularidad es inversamente proporcional a la facilidad de extracción del paralelismo.

### Nivel de operaciones

Puede ser detectado por la arquitectura del hardware, por herramientas de programación (como IDEs o compiladores) y por el programador. Se implementa o aprovecha principalmente por arquitecturas ILP, que lo hacen usando instrucciones dedicadas a ello.

---

<sup>1</sup>Mediante técnicas ya vistas en la asignatura de Sistemas Operativos

**Nivel de bucles**

La arquitectura ya escapa a este nivel de abstracción, por lo que sólo podemos detectarlo mediante herramientas de programación o por la destreza del programador. Se implementa a nivel de arquitecturas SIMD mediante intras-instrucciones en el caso de aquellas paralelizaciones vectoriales ya comentadas; mientras que paralelizaciones del estio TLP se implementan mediante multiprocesador multihebra o multicomputadores, usando threads o procesos.

**Nivel de funciones**

A este nivel ya sólo disponemos del programador para llevar la detección a cabo, quien puede hacer el paralelismo explícito mediante multiprocesadores multihebra, multiprocesadores y multicomputadores; mediante hebras y/o procesos.

**Nivel de programas**

El programador puede hacer explícito el paralelismo si dispone de un multiprocesador o multicomputador, mediante el uso de procesos.

## 1.2. Clasificación de arquitecturas paralelas

### 1.2.1. Objetivos

Una vez terminada la sección que acabamos de comenzar, tratamos de que el lector sea capaz de:

- 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 y TLP.
- Distinguir entre arquitecturas TLP con una instancia de SO y TLP con varias instancias de SO.

### 1.2.2. Computación distribuida

**Computación paralela**

Esta estudia los aspectos hardware y software relacionados con el desarrollo y ejecución de aplicaciones en un sistema de cómputo compuesto por varios cores, procesadores o computadores que es visto externamente como una sólo unidad autónoma, a la que le llamamos unidad multicore, multiprocesador o multicomputador.

### Computación distribuida

Esta se encarga de estudiar los aspectos hardware y software relacionadas con el desarrollo y ejecución de aplicaciones (hasta ahora, igual que en paralela) en un sistema distribuido. Es decir, en una colección de recursos autónomos (como servidores de datos, supercomputadores, bases de datos distribuidas) situados en distintas localizaciones físicas.

Durante toda esta asignatura nos centraremos en computación paralela, pero merece la pena contemplar algunos conocimientos de computación distribuida:

### Computación distribuida a baja escala

Estudia los 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*.

### Computacion 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

Estudia los aspectos relacionados con el desarrollo y ejecución de aplicaciones en un sistema cloud. Esto es, un sistema que ofrece servicios de infraestructura, plataforma y/o software pay-per-use (se paga cuando son requeridos). Son conformados por recursos virtuales que:

- Son una abstracción de los recursos físicos.
- Parece ilimitados en cuanto a número y capacidad gracias a la amplia cantidad de unidades autónomas disponibles. Los cuales son usados y liberados de forma inmediata sin interacción con el proveedor.
- Soportan el acceso de múltiples clientes.

En la sección anterior clasificamos ya el paralelismo que podíamos encontrar dentro de una aplicación. A continuación, nos dedicaremos a clasificar los tipos de arquitecturas y sistemas paralelos que podemos encontrarnos, según varios criterios.

### 1.2.3. Clasificación según el mercado

Según el segmento de mercado, observamos que el número de ventas es inversamente proporcional a la potencia de los computadores, junto con su número de cores y precio. Podemos agrupar todos los computadores en las siguientes categorías (en orden de precio descendente):

- Supercomputadores.
- Servidores de gama alta.
- Servidores de gama media.

- Servidores de gama baja.
- Computadores personales (PCs) o estaciones de trabajo (WSs).
- Sistemas empujados.

#### 1.2.4. Clasificación de Flynn o de flujos

La taxonomía de Flynn nos permite dividir el universo de los computadores en relación a la cantidad de flujos de instrucción y de datos que estos soportan. A continuación, definimos las 4 clases de Flynn, donde usaremos la notación  $xIxD$  donde I y D significan “Instruction” y “Data”, respectivamente. El carácter  $x$  lo sustituiremos por S en el caso de que queramos especificar “Single”; y por M en el caso de que queramos especificar “Multiple”. De esta forma, “SIMD” significa “Single Instruction Multiple Data” y no será necesario nunca más indicar el significado de estas siglas.

##### SISD

Estos son los que presentan un único flujo de instrucciones y un único flujo de datos. Por tanto, tendremos sólo una única unidad de control, así como una única unidad de procesamiento.

##### SIMD

Volvemos a disponer de un único flujo de instrucciones, luego volvemos a tener una única unidad de control, pero en este caso disponemos de múltiples flujos de instrucciones, lo que nos permite tener múltiples unidades de procesamiento, cada una con comunicación independiente con memoria. De esta forma, un computador SIMD puede realizar varias operaciones similares simultáneas con distintos operandos. Cada una de las secuencias de datos y resultados constituyen flujos independientes. Un ejemplo de sistema SIMD puede ser un procesador vectorial.

##### MIMD

Este es el primer caso de computador con varias unidades de control, cada una con su unidad de procesamiento correspondiente, la cual puede acceder de forma independiente a memoria. Por cada flujo de instrucciones existe un flujo de datos. Para ello, necesitaremos disponer de diversos programas, cada uno a ejecutar en un procesador.

##### MISD

En este caso, se ejecutan distintos flujos de datos (y por tanto, dispondremos de distintas unidades de control, cada una con su unidad de procesamiento) sobre el mismo flujo de datos. Notemos que este tipo de computadores puede implementarse mediante las prestaciones que ofrecen los computadores MIMD, donde se sincronizan los procesadores para que los datos vayan pasando de un procesador a otro. Por tanto, no existen computadores MISD específicos, sino que serán una adaptación de un MIMD a un problema particular en el que haya que procesar datos de forma sucesiva (un procesador tras otro).

Como ejemplo ilustrador de las taxonomías ya descritas (y de su capacidad de paralelismo), proponemos el siguiente código:

```
for(int i = 0; i < 4; i++){  
    C[i] = A[i] + B[i];  
    F[i] = D[i] - E[i];  
    G[i] = K[i] * H[i];  
}
```

Asumiendo que el código superior se basa en instrucciones máquina a bajo nivel (ya que es meramente ilustrativo para resaltar las diferencias en las taxonomías), mostramos a continuación las diversas programaciones y ejecuciones en distintos tipos de computadores:

### SISD

En un computador SISD, el procesador debe realizar 4 sumas, 4 restas y 4 multiplicaciones, un total de 12 operaciones que asumimos que se ejecutan en *12 unidades de tiempo*.

### SIMD

En un computador SIMD, podemos a lo mejor disponer de instrucciones vectoriales (las cuales nos permiten realizar operaciones con todos los escalares de un vector de forma atómica). De esta forma, el programa se podría ejecutar en *3 unidades de tiempo* (obviamente, estas unidades no son las mismas a las de un computador SISD; sino que son relativas al tipo de computador), al disponer de tres instrucciones (una suma, una resta y una multiplicación) que nos resuelven el programa sin necesidad del bucle.

### MIMD

Los computadores MIMD nos permiten aproximar el problema de diversas formas:

1. La primera es (suponiendo que disponemos al menos de 3 cores), crear 3 programas (uno que realice la suma, otro la resta y otro la multiplicación, mediante un bucle de 4 iteraciones) y repartirlos entre 3 cores. De esta forma, tardaríamos un tiempo de *4 unidades* (despreciando bastantes variables), debido a que cada core debería hacer 4 iteraciones y a que los cores ejecutan los bucles de forma paralela.
2. Una segunda aproximación al problema es (suponiendo que disponemos de al menos 4 cores), repartir las iteraciones en varios cores, de forma que el core número *i* (*i* entero entre 0 y 3) realice la iteración número *i* de la suma, resta y multiplicación. De esta forma, al tener que ejecutar cada core 3 instrucciones y haciéndolo estos de forma paralela, tenemos un tiempo de *3 unidades*.
3. Una última consideración es juntar las dos aproximaciones en una (suponiendo que disponemos de al menos 12 cores): de los tres programas creados en el primer punto, repartir las iteraciones de estos tal y como lo hacemos en el segundo punto. De esta forma, obtendríamos un tiempo de *1 unidad*.



*Observación.* Nótese que en la diferenciación anterior no hemos considerado los computadores MISD, ya que como se mencionó anteriormente, estos no son una clase de computadores en sí mismos, sino una instancia particular de resolución de una aplicación en computadores del estilo MIMD.

Obsérvese además que las unidades de tiempo en cada tipo de computador son distintas. Sin embargo, el tamaño de unidad temporal de SISD es similar a MIMD (ya que sus instrucciones más costosas no distan mucho entre sí), en contra de SIMD, donde las operaciones vectoriales son bastante costosas, elevando así su unidad de tiempo en comparación con las otras dos taxonomías.

Hemos podido comprobar cómo en SIMD podemos tener paralelismo a nivel de datos; mientras que en MIMD podemos tener tanto paralelismo a nivel de datos como a nivel de tareas, tanto de forma simultánea como de forma independiente.

Además, en computadores MIMD tenemos más libertad en cuanto a entes del sistema operativo (procesos o threads) podemos usar para llevar a cabo la paralelización.

## **Multiprocesadores y Multicomputadores**

Dentro de los computadores de tipo MIMD, encontramos a su vez dos tipos de computadores muy distintos, en función de cómo se encuentra distribuido su espacio de memoria. A continuación, trataremos de dar sus clasificaciones, así como destacar los beneficios y contras de cada uno:

### **Multiprocesadores**

También conocidos como sistemas de memoria compartida (SM, Shared Memory), son sistemas en los que disponemos de diversos procesadores, todos ellos compartiendo el mismo espacio de direcciones. En este caso, el programador no necesita conocer dónde se encuentran almacenados los datos (ya que cualquier procesador tiene físicamente acceso a cualquier dato en memoria).

### **Multicomputadores**

También conocidos como sistemas de memoria distribuida (DM, Distributed Memory), son sistemas con diversos procesadores en los que cada procesador tiene un propio espacio de direcciones particular. Por tanto, el programador necesita conocer dónde (en la memoria de qué procesador) se encuentran los datos, a la hora de realizar programas que aprovechen el paralelismo de tener diversos procesadores.

Las escuetas definiciones manifestadas arriba nos dan una primera idea de cuales son las diferencias entre los multiprocesadores y los multicomputadores. Sin embargo, trataremos de ahondar en este tema, expandiendo las contrapartidas y beneficios que posee cada tipo de sistema.

En un multicomputador, cada procesador tiene un propio espacio de direcciones, por lo que es lógico pensar que la memoria se encuentra de forma física cerca de cada procesador (y es así como normalmente se implementa). Es normal encontrar distribuido también el sistema de entrada y salida (aunque este no tendrá mucha relevancia en nuestro estudio). Por contraparte, en un multiprocesador, al compartir todos los procesadores el mismo espacio de memoria, es lógico plantear un diseño en el que todos los módulos de memoria se encuentren físicamente ubicados en la misma

zona del sistema, separándolos de los procesadores por una red de interconexión que arbitra el acceso a los módulos. Es natural también, centralizar los dispositivos de E/S. Dispuesto este modelo de memoria centralizada, el tiempo de acceso a memoria será igual para cualquier posición de memoria que se acceda desde cualquier procesador. Se trata de una estructura simétrica. Esta clase de multiprocesadores recibe el nombre *SMP* (*Symmetric MultiProcessor*), o multiprocesador simétrico. En estos, el acceso de los procesadores a memoria se realiza a través de la red de interconexión, por tanto, nos interesa disponer de una red buena que permita el acceso al mismo tiempo de distintos procesadores a distintas posiciones de memoria; y de un sistema que arbitre el acceso de los procesadores a una misma posición de memoria.

En multicomputadores, cada procesador tiene su propio módulo de memoria local, al que puede acceder directamente. Es por tanto, que el único fin de la red de interconexión es para comunicar los procesadores entre sí (transferencia de datos). Esto se hace mediante el uso de mensajes entre procesadores. En un multiprocesador, la comunicación entre procesadores puede hacerse de forma directa a través de memoria: un procesador escribe en una posición de memoria la información a comunicar y simplemente tiene que decirle al procesador deseado que lea de dicha posición de memoria (ya sólo queda ver cómo se pasa esta, a través de la red de interconexión).

Esta descripción básica de la red de intercomunicación ya nos plantea una primera desventaja de los multiprocesadores frente a los multicomputadores: *la falta de escalabilidad*. Mientras que en multicomputadores si queremos añadir un nuevo procesador, nos será tan simple como conectar a la red de interconexión un nuevo procesador (junto con sus módulos de memoria y de E/S). Por contraparte, en multiprocesadores, deberemos también conectar el procesador a la red, teniendo en cuenta de que ahora tendremos un nuevo nodo que use esta red de forma probablemente simultánea al resto: las comunicaciones entre procesadores (que son no muy frecuentes) son el único propósito de los multicomputadores, mientras que los multiprocesadores deben usarlo para comunicaciones y acceso a memoria, lo que dificulta la ejecución de los procesadores de forma paralela, al tener estos que acceder constantemente a memoria de forma simultánea. Por tanto, nos es más fácil añadir procesadores a un sistema multicomputador antes que a uno multiprocesador, ante el temor de saturar la red de intercomunicaciones. Posteriormente comentaremos una mejora de los multiprocesadores que trata de parchear este problema.

A continuación, seguimos planeando diferencias entre estos:

**Latencia de acceso a memoria** El tiempo de acceso a memoria (como se puede esperar) es mayor en multiprocesadores que en multicomputadores, al tener que atravesar toda la infraestructura de red de interconexión, junto con lo que esto conlleva, ya que puede darse la posibilidad de que varios procesadores ocupen la red de forma simultánea (lo cual ya plantea un problema), pero además deberemos arbitrar el acceso de distintos procesadores a una misma posición de memoria. Cuanto mayor sea el número de procesadores, la probabilidad de conflicto aumenta (lo que refleja el problema de escalabilidad previamente comentado).

**Comunicaciones** Como hemos comentado anteriormente, los multiprocesadores

pueden comunicarse entre sí mediante memoria, por lo que sólo será necesario implementar sencillas instrucciones de carga (load) y almacenamiento (save). Mientras que los multicomputadores necesitan desarrollar toda una estrategia de mensajes, junto con instrucciones de envío (send) y de recibo de datos (receive).

**Herramientas de programación** Antes de ejecutar una aplicación en un multicomputador (suponiendo que este implementa paralelismo entre procesadores, que es el caso interesante), debemos ubicar en memoria (en la de cada procesador) el código del programa que estamos a punto de ejecutar, junto con los datos que este necesita. Es decir, es necesario realizar una distribución de carga de trabajo entre los distintos procesadores. Nótese que en multiprocesadores esta distribución no es necesaria, ya que todos los procesadores pueden acceder al mismo espacio de direcciones. Esto presenta un gran problema, ya que nos es difícil prever el tiempo de ejecución de cada bloque de código, ni a cuánta carga de trabajo estará sometido cada procesador. Aún es esto parte de la responsabilidad del programador (aunque algunos compiladores ya intentan realizar esta distribución de trabajo). Por tanto, necesitamos herramientas de programación más sofisticadas a la hora de trabajar con multicomputadores.

## SMP frente a NUMA

Dentro de los multiprocesadores anteriormente comentados, tratamos de dar una solución que solvente el problema de escalabilidad anteriormente planteado. Algunas opciones temporales son el aumento del caché de cada procesador, así como el uso de redes de interconexión de menor latencia y mayor ancho de banda (así como de una forma de red que beneficie a nuestro sistema, más allá de un bus). Sin embargo, tratamos de buscar una solución que nos aporte más beneficios, que probablemente venga de cambiar un poco el planteamiento del sistema.

Para nosotros era lógico que un multiprocesador tuviera una arquitectura SMP, donde los módulos de memoria (y los de E/S) se encuentren centralizados y accedidos mediante una red de interconexión. Este era un ejemplo de arquitectura *UMA* (*Uniform Memory Access*), donde cada procesador tarda el mismo tiempo en acceder a cada módulo de memoria.

Sin embargo, planteamos ahora que, manteniendo la estructura de un multiprocesador (esto es, compartiendo el espacio de direcciones), repartir los módulos de memoria a lo largo del sistema, (estableciendo una asociación de un módulo por procesador), de forma que el tiempo de acceso a memoria sea menor para el procesador a su módulo correspondiente. De esta forma, un procesador podrá seguir accediendo al resto de módulos, aunque con una penalización en tiempo respecto a acceder a su módulo de memoria. A este tipo de arquitecturas de multiprocesadores se les conoce como *NUMA* (*Non-Uniform Memory Access*), provenientes de los 90. Al módulo de memoria próximo al procesador le llamaremos módulo de memoria local. Para que un NUMA sea realmente escalable (es la motivación de su creación), se deberá reducir la latencia media, reduciendo el número de accesos a la memoria local de otro procesador. Para ello, necesitamos distribuir (como hacíamos en multicomputadores) la carga de trabajo entre los módulos de memoria, de forma que en el módulo local se encuentren el código y los datos frecuentemente utilizados. Observemos que

acabamos de crear un paradigma similar a la caché de dentro de un procesador. Podemos por tanto, aproximarnos a este reparto de forma estática (repartiendo antes de ejecutar) o dinámica (realizando el reparto en tiempo de ejecución).

Como resumen a la comparativa de multicomputadores y multiprocesadores, podemos plantear el siguiente esquema:

### **Multicomputadores**

- Múltiples espacios de direcciones: memoria no compartida.
- Memoria físicamente distribuida.
- Gran escalabilidad.

### **Multiprocesadores**

- Un único espacio de direcciones: memoria compartida.

#### **NUMA**

- Memoria físicamente distribuida.
- Sistema escalable.

#### **UMA**

- SMP: memoria físicamente centralizada.
- Plantea problemas de escalabilidad.

## **1.2.5. Clasificación según el paralelismo aprovechado**

En función del tipo de paralelismo que aprovechen las máquinas, tenemos distintos tipos de clasificación:

### **Arquitectura con ILP**

Las arquitecturas con paralelismo a nivel de instrucción ejecuta las instrucciones de forma concurrente o en paralelo. Se trata de cores escalares segmentados, superescalares o VLIW (very long instruction word).

### **Arquitectura con DLP**

Las arquitecturas con paralelismo a nivel de datos ejecutan las operaciones de una instrucción de forma concurrente o en paralelo. Hacen referencia a unidades funcionales vectoriales o SIMD.

### **Arquitectura con TLP y una instancia de SO**

Este tipo de arquitecturas con paralelismo a nivel de tareas ejecutan múltiples flujos de instrucciones de forma concurrente o paralela usando para ello una única instancia de sistema operativo (esto es, un único proceso). Pueden hacer referencia a cores que modifican la arquitectura escalar segmentada, superescalar o VLIW para ejecutar threads de forma concurrente o en paralelo. Por otra parte, también puede hacer referencia a multiprocesadores, los cuales ejecutan threads en paralelo en un computador con múltiples cores (incluye multicore).

### **Arquitectura con TLP y múltiples instancias de SO**

Este tipo de arquitecturas con paralelismo a nivel de tareas ejecutan múltiples flujos de instrucción en paralelo. Hace referencia a los multicomputadores, los cuales ejecutan threads en paralelo en un sistema con muchos computadores.

## **1.3. Evaluación de prestaciones**

### **1.3.1. Objetivos**

En esta sección, aprenderemos a:

- Distinguir entre tiempo de CPU (sistema y usuario) de Unix y el 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.
- Calcular la ganancia en prestaciones/velocidad.
- Aplicar la ley de Amdahl.



## 2. Relaciones de Problemas

### 2.1. Arquitecturas Paralelas

**Ejercicio 2.1.1.** 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.

1. ¿Cuál es la ganancia que se puede obtener por reducción a 3 ciclos de las instrucciones con enteros?

Resumimos los datos del enunciado en la siguiente tabla:

$I_i$	$CPI_i^b$	$NI_i$
LOAD	4 ciclos	0,2 NI
STORE	3 ciclos	0,1 NI
FX. POINT	6 ciclos	0,25 NI
FLT. POINT	8 ciclos	0,15 NI
BRANCH	3 ciclos	0,3 NI

donde  $I_i$  es el tipo de instrucción,  $CPI_i^b$  es el número de ciclos por instrucción y  $NI_i$  es el número de instrucciones de ese tipo.

El tiempo base  $T_b$  que tardaría en ejecutarse el programa sin mejoras sería:

$$\begin{aligned}
 T_b &= NI \cdot CPI \cdot T_c = T_c \cdot \sum_i NI_i \cdot CPI_i = \\
 &= T_c \cdot NI \cdot \left( \underbrace{0,2 \cdot 4}_{LD} + \underbrace{0,1 \cdot 3}_{ST} + \underbrace{0,25 \cdot 6}_{FP} + \underbrace{0,15 \cdot 8}_{FLT \text{ POINT}} + \underbrace{0,3 \cdot 3}_{BRANCH} \right) = \\
 &= T_c \cdot NI \cdot 4,7
 \end{aligned}$$

donde  $T_c$  representa el tiempo de ciclo. Respecto al tiempo mejorado  $T_p$ , sabiendo ahora que en caso de los números enteros el número de ciclos se reduce

a 3, tendríamos:

$$\begin{aligned}
 T_p &= NI \cdot CPI \cdot T_c = T_c \cdot \sum_i NI_i \cdot CPI_i = \\
 &= T_c \cdot NI \cdot \left( \underbrace{0,2 \cdot 4}_{LD} + \underbrace{0,1 \cdot 3}_{ST} + \underbrace{0,25 \cdot 3}_{FP} + \underbrace{0,15 \cdot 8}_{FLT \text{ POINT}} + \underbrace{0,3 \cdot 3}_{BRANCH} \right) = \\
 &= T_c \cdot NI \cdot 3,95
 \end{aligned}$$

La expresión de la ganancia, por tanto, es:

$$S = \frac{T_b}{T_p} = \frac{\cancel{T_c} \cdot \cancel{NI} \cdot 4,7}{\cancel{T_c} \cdot \cancel{NI} \cdot 3,95} = \frac{4,7}{3,95} \approx 1,1898$$

2. ¿Cuál es la ganancia que se puede obtener por reducción a 3 ciclos de las instrucciones en coma flotante?

Tenemos que:

$$\begin{aligned}
 T_p &= NI \cdot CPI \cdot T_c = T_c \cdot \sum_i NI_i \cdot CPI_i = \\
 &= T_c \cdot NI \cdot \left( \underbrace{0,2 \cdot 4}_{LD} + \underbrace{0,1 \cdot 3}_{ST} + \underbrace{0,25 \cdot 6}_{FP} + \underbrace{0,15 \cdot 3}_{FLT \text{ POINT}} + \underbrace{0,3 \cdot 3}_{BRANCH} \right) = \\
 &= T_c \cdot NI \cdot 3,95
 \end{aligned}$$

La expresión de la ganancia, por tanto, es:

$$S = \frac{T_b}{T_p} = \frac{\cancel{T_c} \cdot \cancel{NI} \cdot 4,7}{\cancel{T_c} \cdot \cancel{NI} \cdot 4,25} = \frac{4,7}{3,95} \approx 1,1898$$

Como podemos ver, la ganancia es la misma que en el caso anterior. Esto se debe a que, aun recudiendo más ciclos de reloj (5 en este caso, frente a 3 en el anterior), el número de instrucciones de coma flotante es menor que el número de instrucciones de enteros, por lo que la ganancia se compensa. Se tiene que  $0,25 \cdot 3 = 0,15 \cdot 5$ .

**Ejercicio 2.1.2.** 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 acoplo que introduce un retardo de 25 ns.

1. ¿Cuál es la máxima ganancia de velocidad posible? ¿Cuál es la productividad máxima del cauce?

Tenemos que el ciclo de reloj es de  $T_c = 125ns + 25ns = 150ns$ , ya que depende de la etepa de ejecución más lenta. La ganancia de velocidad, siendo  $N$  el número de operaciones, es:

$$S(N) = \frac{T^b(N)}{T^s(N)}$$



donde  $T^b(N)$  es el tiempo base y  $T^s(N)$  es el tiempo usando segmentación. El tiempo base es directo ver que es  $T^b(N) = N \cdot T_{op}$ , mientras que el tiempo segmentado es algo más complejo, ya que hay que tener en cuenta el tiempo que tarda el cauce en llenarse. Una vez está lleno, ejecuta una operación cada ciclo de reloj, pero la primera operación tarda  $N_{etapas} \cdot T_c$ . Por tanto, el tiempo segmentado es:

$$\begin{aligned} T^s(N) &= [1 \cdot N_{etapas} \cdot T_c] + [(N - 1) \cdot T_c] = N_{etapas} \cdot T_c + (N - 1)T_c = \\ &= 4 \cdot T_c + (N - 1)T_c \end{aligned}$$

donde el 4 se debe a que es el número de etapas del cauce. Por tanto, la ganancia de velocidad es:

$$S(N) = \frac{T^b(N)}{T^s(N)} = \frac{N \cdot T_{op}}{4 \cdot T_c + (N - 1)T_c} = \frac{N \cdot T_{op}}{T_c \cdot (4 + N - 1)} = \frac{N \cdot 450}{150 \cdot (3 + N)} = \frac{3N}{3 + N}$$

La ganancia máxima se presupone que se alcanza cuando  $N \rightarrow \infty$ , por lo que:

$$S_{\text{máx}} = S(N \gg) = \lim_{N \rightarrow \infty} S(N) = \lim_{N \rightarrow \infty} \frac{N \cdot T_{op}}{T_c \cdot (3 + N)} = \frac{T_{op}}{T_c} = \frac{450}{150} = 3$$

La productividad máxima del cauce es:

$$P_{\text{máx}} = \frac{S_{\text{máx}}}{N_{etapas}} = \frac{3}{4}$$

2. ¿A partir de qué número de operaciones ejecutadas se consigue una productividad igual al 90 % de la productividad máxima?

**Ejercicio 2.1.3.** En un procesador sin segmentación de cauce, determine cuál de estas dos alternativas para realizar un salto condicional es mejor:

- ALT1: Una instrucción **COMPARE** actualiza un código de condición y es seguida por una instrucción **BRANCH** que comprueba esa condición. Se usan dos instrucciones.
- ALT2: Una sola instrucción incluye la funcionalidad de las instrucciones **COMPARE** y **BRANCH**. Se usa una única instrucción.

Hay que tener en cuenta que hay un 20 % de instrucciones **BRANCH** para ALT1 en el conjunto de programas de prueba; que las instrucciones **BRANCH** en ALT1 y **COMPARE+BRANCH** en ALT2 necesitan 4 ciclos mientras que todas las demás necesitan sólo 3; y que el ciclo de reloj de la ALT1 es un 25 % menor que el de la ALT2, dado que en este caso la mayor funcionalidad de la instrucción **COMPARE+BRANCH** ocasiona una mayor complejidad en el procesador.

En todo el ejercicio, el superíndice 1 denotará la ALT1, mientras que el superíndice 2 denotará la ALT2. Como el tiempo de ciclo de reloj depende de la ejecución más lenta, es normal que este cambie (como se especifica en el enunciado). La relación entre los tiempos de ciclo, dada por el enunciado, es la siguiente:

$$T_c^1 = T_c^2 - 0,25T_c^2 = 0,75T_c^2$$

Resumimos los datos del enunciado en la siguiente tabla:

$I_i^1$	$CPI_i^1$	$NI_i^1$	$I_i^2$	$CPI_i^2$	$NI_i^2$
BRANCH	4 ciclos	$0,2 \cdot NI^1$	COMPARE+BRANCH	4 ciclos	$0,2 \cdot NI^1$
COMPARE	3 ciclos	$0,2 \cdot NI^1$			
Demás	3 ciclos	$0,6 \cdot NI^1$	Demás	3 ciclos	$0,6 \cdot NI^1$
		$NI^1$			$0,8 \cdot NI^1 = NI^2$

Hay que tener en cuenta que sabemos que cada salto conlleva un **BRANCH** y un **COMPARE** en la primera alternativa, por lo que hay el mismo número de instrucciones de dichos tipos. Además, como el programa es el mismo, tenemos que hay el mismo número de instrucciones de salto en ambos, por eso deducimos el número de instrucciones de salto en la segunda alternativa. Tiene sentido que  $NI^2 < NI^1$ , ya que cada instrucción de salto conlleva 2 órdenes en la primera alternativa, mientras que en la segunda conlleva una sola.

Tenemos que ver qué alternativa nos da un tiempo de ejecución menor (tengamos en cuenta que el tiempo de ciclo de cada uno no es el mismo, por lo que tenemos que pasarlo todo al mismo tiempo de ciclo):

Por ser  $T_{CPU}^1 < T_{CPU}^2$ , concluimos que la opción ALT1 es la mejor, en cuanto a tiempos de ejecución.

**Ejercicio 2.1.4.** ¿Qué ocurriría en el problema del ejercicio anterior (Ejercicio 2.1.3) si el ciclo de reloj fuese únicamente un 10 % mayor para la ALT2?

**Ejercicio 2.1.5.** Considere un procesador no segmentado con una arquitectura de tipo **LOAD/STORE** en la que las operaciones sólo utilizan como operandos registros de la CPU. Para un conjunto de programas representativos de su actividad se tiene que el 43 % de las instrucciones son operaciones con la ALU (3 CPI), el 21 % **LOADs** (4 CPI), el 12 % **STOREs** (4 CPI) y el 24 % **BRANCHs** (4 CPI). Se ha podido comprobar que un 25 % de las operaciones con la ALU utilizan operandos en registros que no se vuelven a utilizar. Compruebe si mejorarían las prestaciones si, para sustituir ese 25 % de operaciones, se añaden instrucciones con un dato en un registro y otro en memoria. Tengan en cuenta en la comprobación que para estas nuevas instrucciones el valor de CPI es 4 y que añadirlas ocasiona un incremento de un ciclo en el CPI de los **BRANCH**, pero no afectan al ciclo de reloj.

Resumimos los datos del enunciado en la siguiente tabla:

$I_i^1$	$CPI_i^1$	$NI_i^1$	$I_i^2$	$CPI_i^2$	$NI_i^2$
Instrucción ALU	3 ciclos	$0,43 \cdot NI$	ALU r,r	3 ciclos	$0,2325 \cdot NI^1$
			ALU r,m	4 ciclos	$0,1075 \cdot NI^1$
LOADs	4 ciclos	$0,21 \cdot NI^1$	LOADs	4 ciclos	$0,1225 \cdot NI^1$
STOREs	4 ciclos	$0,12 \cdot NI^1$	STOREs	4 ciclos	$0,12 \cdot NI^1$
BRANCHs	4 ciclos	$0,24 \cdot NI^1$	BRANCHs	5 ciclos	$0,24 \cdot NI^1$
					$0,8925 \cdot NI^1 = NI^2$

Donde para completar la columna  $NI_i^2$  hemos usado que:

$$\begin{aligned} NI_{\text{ALU } r,r}^1 &= 0,75(0,43 \cdot NI^1) = 0,2325 \cdot NI^1 \\ NI_{\text{ALU } r,m}^1 &= 0,25(0,43 \cdot NI^1) = 0,1075 \cdot NI^1 \\ NI_{\text{LOADs}}^1 &= 0,21 \cdot NI^1 - 0,1075 \cdot NI^1 = 0,1225 \cdot NI^1 \end{aligned}$$

Calculamos los tiempos en CPU:

$$\begin{aligned} T_{CPU}^1 &= NI^1 (\overbrace{0,43 \cdot 3c}^{\text{ALU } r,r} + (0,21 + 0,12 + 0,24) \cdot 4c) \cdot T_c \\ &= NI^1 \cdot 3,57 \cdot T_c \end{aligned}$$

$$\begin{aligned} T_{CPU}^2 &= NI^1 (0,3235 \cdot 3c + (0,1075 + 0,1225 + 0,12) \cdot 4c + 0,24 \cdot 4c) \cdot T_c \\ &= NI^1 \cdot 3,4875 \cdot T_c \end{aligned}$$

Y tenemos que  $T_{CPU}^2 < T_{CPU}^1$ , luego sí que mejorarían las prestaciones.

**Ejercicio 2.1.6.** Se ha diseñado un compilador para la máquina LOAD/STORE del problema anterior (Ejercicio 2.1.5). Ese compilador puede reducir en un 50 % el número de operaciones con la ALU, pero no reduce el número de LOADs, STOREs, y BRANCHs. Suponiendo que la frecuencia de reloj es de 50 Mhz, ¿Cuál es el número de MIPS y el tiempo de ejecución que se consigue con el código optimizado? Compárelos con los correspondientes del código no optimizado.

**Ejercicio 2.1.7.** En un programa que se ejecutan en un procesador no segmentado que funciona a 100 MHz, hay un 20 % de instrucciones LOAD que necesitan 4 ciclos, un 15 % de instrucciones STORE que necesitan 3 ciclos, un 40 % de instrucciones con operaciones en la ALU que necesitan 6 ciclos, y un 25 % de instrucciones de salto que necesitan 3 ciclos.

1. Si en las instrucciones que usan la ALU el tiempo en la ALU supone 4 ciclos, determine cuál es la máxima ganancia que se puede obtener si se mejora el diseño de la ALU de forma que se reduce su tiempo de ejecución a la mitad de ciclos.
2. Con qué porcentaje de instrucciones con operaciones en la ALU se podría haber obtenido en los cálculos del apartado 1 una ganancia mayor que 2? Razone su respuesta.

**Ejercicio 2.1.8.** Suponga que en los programas que constituyen la carga de trabajo habitual de un procesador las instrucciones de coma flotante consumen un promedio del 13 % del tiempo del procesador.

1. Ha aparecido en el mercado una nueva versión del procesador en la que la única mejora con respecto a la versión anterior es una nueva unidad de coma flotante que permite reducir el tiempo de las instrucciones de coma flotante a tres cuartas partes del tiempo que consumían antes. ¿Cuál es la máxima ganancia de velocidad que puede esperarse en los programas que constituyen la carga de trabajo si se utiliza la nueva versión del procesador?

La  $p$  de la ley sería  $3/4$  y  $f = 0,87$ . Para calcular la ganancia adicional:

$$S = \frac{T_b}{T_p} = \frac{\mathcal{T}_b}{0,87\mathcal{T}_b + 0,13\mathcal{T}_b \cdot 3/4} = 1,033$$

- ¿Cuál es la máxima ganancia de velocidad con respecto a la versión inicial del procesador que, en promedio, puede esperarse en los programas debido a mejoras en la velocidad de las operaciones en coma flotante?

Lo que esperamos en el mejor caso es llevar a 0 el tiempo que tarda en ejecutarse la parte de punto flotante.

$$S = \frac{T_b}{T_p} = \frac{\mathcal{T}_b}{0,87\mathcal{T}_b + 0} = 1,149$$

- ¿Cuál debería ser el porcentaje de tiempo de cálculo con datos en coma flotante en los programas para esperar una ganancia máxima de 4 en lugar de la obtenida en el apartado 2?

Buscamos la  $f$  para obtener 4:

$$\frac{\mathcal{T}_b}{f\mathcal{T}_b + \underbrace{(1-f)\mathcal{T}_b}_p} = 4 \Rightarrow f = \frac{1}{4}$$

- ¿Cuánto debería reducirse el tiempo de las operaciones en coma flotante con respecto a la situación inicial para que la ganancia máxima sea 2 suponiendo que en la versión inicial el porcentaje de tiempo de cálculo con coma flotante es el obtenido en el apartado 3?

Se trata de buscar  $p$ , suponiendo que tenemos un 75 % de operaciones de coma flotante. Simplemente hay que despejar  $p$ .

**Ejercicio 2.1.9.** Suponga que, en el código siguiente,  $a[]$  es un array de números de 32 bits en coma flotante y  $b$  un número de 32 bits en coma flotante y que debería ejecutarse en menos de 0,5 segundos para  $N = 10^9$ :

```
for (i=0; i<N; i++)
    a[i+2]=(a[i+2]+a[i+1]+a[i])*b;
```

- ¿Cuántos GFLOPS se necesitan para poder ejecutar el código en menos de 0,5 segundos?

$$T(10) < 0,5\text{seg}$$

$$GFLOPS = \frac{NumFFP}{T_{CPU} \cdot 10^9} = \frac{3 \cdot N}{T_{CPU} \cdot 10^9} > \frac{3 \cdot N}{0,5 \cdot 10^9}$$

2. Suponiendo que este código en ensamblador tiene 7N instrucciones y que se ha ejecutado en un procesador de 32 bits a 2 GHz. ¿Cual es el número medio de instrucciones que el procesador tiene que poder completar por ciclo para poder ejecutar el código en menos de 0,5 segundos?

Sabemos que el tiempo en CPU lo podemos calcular de la forma:

$$T_{CPU} = NI \cdot CPI \cdot T_c = \frac{NI}{IPC \cdot F}$$

Con:

$$IPC = \frac{1}{CPS} =$$

Sabemos que  $NI = 7 \cdot 10^9$ , buscamos el valor de  $IPC$  y  $F = 2 \cdot 10^9$  c/s:

$$\frac{7 \cdot 10^9}{IPC \cdot 2 \cdot 10^9} < 0,5$$

$$IPC > 7$$

3. Estimando que el programa pasa el 75 % de su tiempo de ejecución realizando operaciones en coma flotante, ¿cuánto disminuiría como mucho el tiempo de ejecución si se redujesen un 75 % los tiempos de las unidades de coma flotante?

Sea  $T_p$  el tiempo de mejora y  $T_b$  el tiempo base:

$$Porcreduccion = \frac{T_b - T_p}{T_b} \cdot 100$$

$$T_p = 0,25T_p + 0,75T_b \cdot 0,25 = 0,4375T_b$$

$$\left(1 - \frac{T_p}{T_b}\right) \cdot 100 = \left(1 - \frac{0,4375T_b}{T_b}\right) \cdot 100 = 56,25 \%$$

$$p = 4 \quad f = 0,25$$

**Ejercicio 2.1.10.** Un compilador ha generado un código máquina optimizado para el siguiente programa

```

par=0; impar=0;
for (i=0;i<N;i++)
    if ((i%2) == 0)
        par=par+c*x[i];
    else
        impar=impar-c*x[i];

```

sin utilizar instrucciones de salto dentro de las iteraciones del bucle (porque se ha usado la técnica de desenrollado el bucle que veremos en el Seminario 4): el código tiene un número de iteraciones de  $N/2$ , 7 instrucciones fuera del bucle (2 de almacenamiento en memoria, 5 instrucciones para inicializar registros), 9 instrucciones dentro del bucle (4 instrucciones para implementar el bucle for: incremento de la variable de control  $i$ , comparación, salto condicional y un salto incondicional; 4 instrucciones coma flotante y 2 instrucciones de carga desde memoria a registro -se leen dos componentes de  $x$ ). El computador donde se ejecuta dispone de:

- Un procesador superescalar de 32 bits a 2 GHz capaz de terminar dos instrucciones de coma flotante por ciclo y dos instrucciones de cualquier otro tipo por ciclo, excepto instrucciones de carga, cuyo tiempo depende de si hay o no fallo de cache (si no hay fallo de cache suponen 1 ciclo), y las instrucciones de almacenamiento que suponen 1 ciclo.
- Dos caches integradas en el chip de procesamiento (una para datos y otra para instrucciones) de 512 KBytes cada una, mapeo directo, política de actualización de postescritura, líneas de 32 bytes, y latencia de un ciclo de reloj.
- Una memoria principal con latencia de 30 ns. y ciclos burst 6-1-1-1 a través de un bus de memoria de 200 MHz con 64 bits.

Conteste a las siguientes cuestiones:

1. ¿Cuál es la velocidad pico del procesador (en GFLOPS)?
2. ¿Cuál es el tiempo mínimo que tarda en ejecutarse el programa para  $N = 211$ ?
3. ¿Cuántos MFLOPS alcanza el programa?

*Observación.* Considere que el vector  $x$  se almacena en memoria en una dirección múltiplo del tamaño de una línea de cache y que ningún componente está en cache cuando se referencia;  $N$ ,  $i$  estarán en registros de enteros, `par`, `impar`, `c`, y `x[]` son números de 32 bits en coma flotante; dentro del bucle `c`, `par` e `impar` estarán en registros.

**Cuestión 2.1.1.** Indique cuál es la diferencia fundamental entre una arquitectura CC-NUMA y una arquitectura SMP.

**Cuestión 2.1.2.** ¿Cuándo diría que un computador es un multiprocesador y cuándo que es un multicomputador?

**Cuestión 2.1.3.** ¿Un CC-NUMA escala más que un SMP? ¿Por qué?

**Cuestión 2.1.4.** Indique qué niveles de paralelismo implícito en una aplicación puede aprovechar un PC con un procesador de 4 cores, teniendo en cuenta que cada core tiene unidades funcionales SIMD (también llamadas unidades multimedia) y una microarquitectura segmentada y superscalar. Razone su respuesta.

**Cuestión 2.1.5.** Si le dicen que un ordenador es de 20 GIPS ¿puede estar seguro que ejecutará cualquier programa de 20000 instrucciones en un microsegundo?

**Cuestión 2.1.6.** ¿Aceptaría financiar/embarcarse en un proyecto en el que se plantease el diseño e implementación de un computador de propósito general con arquitectura MISD? (Justifique su respuesta).

**Cuestión 2.1.7.** Deduzca la expresión que se usa para representar la ley de Amdahl suponiendo que se mejora un recurso del procesador, que hay una probabilidad  $f$  de no utilizar dicho recurso y que la mejora supone un incremento en un factor de  $p$  de la velocidad de procesamiento del recurso.

**Cuestión 2.1.8.** ¿Es cierto que si se mejora una parte de un sistema (por ejemplo, un recurso de un procesador) se observa experimentalmente que, al aumentar el factor de mejora, llega un momento en que se satura el incremento de velocidad que se consigue? (Justifique la respuesta)

**Cuestión 2.1.9.** ¿Es cierto que la cota para el incremento de velocidad que establece la ley de Amdahl crece a medida que aumenta el valor del factor de mejora aplicado al recurso o parte del sistema que se mejora? (Justifique la respuesta).

**Cuestión 2.1.10.** ¿Qué podría ser mejor suponiendo velocidades pico, un procesador superescalar capaz de emitir cuatro instrucciones por ciclo, o un procesador vectorial cuyo repertorio permite codificar 8 operaciones por instrucción y emite una instrucción por ciclo? (Justifique su respuesta).

**Cuestión 2.1.11.** 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 2.1.12.** 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.

## 2.2. Programación paralela

**Ejercicio 2.2.1.** Un programa tarda 40 s en ejecutarse en un multiprocesador. Durante un 20 % de ese tiempo se ha ejecutado en cuatro procesadores; durante un 60 %, en tres; y durante el 20 % restante, en un procesador (consideramos que se ha distribuido la carga de trabajo por igual entre los procesadores que colaboran en la ejecución en cada momento, despreciamos sobrecarga).

1. ¿Cuánto tiempo tardaría en ejecutarse el programa en un único procesador?
2. ¿Cuál es la ganancia en velocidad obtenida con respecto al tiempo de ejecución secuencial?
3. ¿Cuál es la ganancia en eficiencia obtenida con respecto al tiempo de ejecución secuencial?

**Ejercicio 2.2.2.** Un programa tarda 20 s en ejecutarse en un procesador  $P_1$ , y requiere 30 s en otro procesador  $P_2$ . Si se dispone de los dos procesadores para la ejecución del programa (despreciamos sobrecarga):

1. ¿Qué tiempo tarda en ejecutarse el programa si la carga de trabajo se distribuye por igual entre los procesadores  $P_1$  y  $P_2$ ?
2. ¿Qué distribución de carga entre los dos procesadores  $P_1$  y  $P_2$  permite el menor tiempo de ejecución utilizando los dos procesadores en paralelo? ¿Cuál es este tiempo?

**Ejercicio 2.2.3.** ¿Cuál es fracción de código paralelo de un programa secuencial que, ejecutado en paralelo en 8 procesadores, tarda un tiempo de 100 ns, durante 50 ns utiliza un único procesador y durante otros 50 ns utiliza 8 procesadores (distribuyendo la carga de trabajo por igual entre los procesadores)?

**Ejercicio 2.2.4.** Un 25 % de un programa no se puede paralelizar, el resto se puede distribuir por igual entre cualquier número de procesadores. ¿Cuál es el máximo valor de ganancia de velocidad que se podría conseguir al paralelizarlo en  $p$  procesadores, y con infinitos? ¿A partir de cuál número de procesadores se podrían conseguir ganancias mayores o iguales que 2?

**Ejercicio 2.2.5.** En la Figura 2.1, se presenta el grafo de dependencia entre tareas para una aplicación. La figura muestra la fracción del tiempo de ejecución secuencial que la aplicación tarda en ejecutar grupos de tareas del grafo. Suponiendo un tiempo de ejecución secuencial de 60 s, que las tareas no se pueden dividir en tareas de menor granularidad y que el tiempo de comunicación es despreciable, obtener el tiempo de ejecución en paralelo y la ganancia en velocidad en un computador con:

1. 4 procesadores.
2. 2 procesadores.

**Ejercicio 2.2.6.** Un programa se ha conseguido dividir en 10 tareas. El orden de precedencia entre las tareas se muestra con el grafo dirigido de la Figura 2.2. La ejecución de estas tareas en un procesador supone un tiempo de 2 sg. El 10 % de ese tiempo es debido a la ejecución de la tarea 1; el 15 % a la ejecución de la tarea 2; otro 15 % a la ejecución de 3; cada tarea 4, 5, 6 o 7 supone el 9 %; un 8 % supone la tarea 8; la tarea 9 un 10 %; por último, la tarea 10 supone un 6 %. Se dispone de una arquitectura con 8 procesadores para ejecutar la aplicación. Consideramos que el tiempo de comunicación se puede despreciar.

1. ¿Qué tiempo tarda en ejecutarse el programa en paralelo?
2. ¿Qué ganancia en velocidad se obtiene con respecto a su ejecución secuencial?

**Ejercicio 2.2.7.** Se quiere paralelizar el siguiente trozo de código:

```
// {Cálculos antes del bucle}  
for( i=0; i<w; i++) {  
    // Código para i  
}  
// {cálculos después del bucle}
```



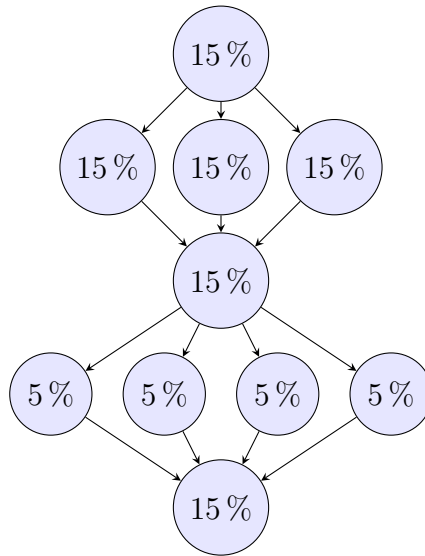


Figura 2.1: Grafo de tareas del Ejercicio 2.2.5

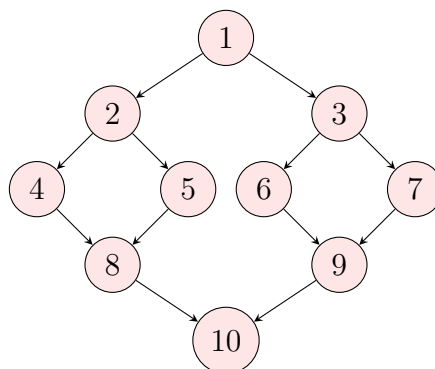


Figura 2.2: Grafo de tareas del Ejercicio 2.2.6

Los cálculos antes y después del bucle suponen un tiempo de  $t_1$  y  $t_2$ , respectivamente. Una iteración del ciclo supone un tiempo  $t_i$ . En la ejecución paralela, la inicialización de  $p$  procesos supone un tiempo  $k_1p$  ( $k_1$  constante), los procesos se comunican y se sincronizan, lo que supone un tiempo  $k_2p$  ( $k_2$  constante);  $k_1p + k_2p$  constituyen la sobrecarga.

1. Obtener una expresión para el tiempo de ejecución paralela del trozo de código en  $p$  procesadores ( $T_p$ ).
2. Obtener una expresión para la ganancia en velocidad de la ejecución paralela con respecto a una ejecución secuencial ( $S_p$ ).
3. ¿Tiene el tiempo  $T_p$  con respecto a  $p$  una característica lineal o puede presentar algún mínimo? ¿Por qué? En caso de presentar un mínimo, ¿para qué número de procesadores  $p$  se alcanza?

**Ejercicio 2.2.8.** Supongamos que se va a ejecutar en paralelo la suma de  $n$  números en una arquitectura con  $p$  procesadores o cores ( $p$  y  $n$  potencias de dos) utilizando un grafo de dependencias en forma de árbol (divide y vencerás) para las tareas.

1. Dibujar el grafo de dependencias entre tareas para  $n = 16$  y  $p = 8$ . Hacer una asignación de tareas a procesos.
2. Obtener el tiempo de cálculo paralelo para cualquier  $n$  y  $p$  con  $n > p$  suponiendo que se tarda una unidad de tiempo en realizar una suma.
3. Obtener el tiempo comunicación del algoritmo suponiendo:
  - a) Que las comunicaciones en un nivel del árbol se pueden realizar en paralelo en un número de unidades de tiempo igual al número de datos que recibe o envía un proceso en cada nivel del grafo de tareas (tenga en cuenta la asignación de tareas a procesos que ha considerado en el apartado 1)
  - b) Que los procesadores que realizan las tareas de las hojas del árbol tienen acceso sin coste de comunicación a los datos que utilizan dichas tareas.
4. Suponiendo que el tiempo de sobrecarga coincide con el tiempo de comunicación calculado en el apartado 3, obtener la ganancia en prestaciones.
5. Obtener el número de procesadores para el que se obtiene la máxima ganancia con  $n$  números.

**Ejercicio 2.2.9.** Se va a paralelizar un decodificador JPEG en un multiprocesador. Se ha extraído para la aplicación el siguiente grafo de tareas que presenta una estructura segmentada (o de flujo de datos):

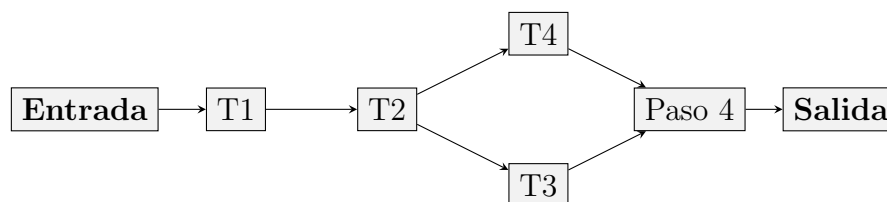


Figura 2.3: Segmentación del Ejercicio 2.2.9

La entrada tenemos que es el bloque de la imagen a decodificar (supone 8x8 pixels de la imagen). La salida será el bloque decodificado de 8x8 pixel. Las tareas 1, 2 y 5 se ejecutan en un tiempo igual a  $t$ , mientras que las tareas 3 y 4 suponen  $1,5t$ . El decodificador JPEG aplica el grafo de tareas de la figura a bloques de la imagen, cada uno de 8x8 píxeles. Si se procesa una imagen que se puede dividir en  $n$  bloques de 8x8 píxeles, a cada uno de esos  $n$  bloques se aplica el grafo de tareas de la figura. Obtenga la mayor ganancia en prestaciones que se puede conseguir paralelizando el decodificador JPEG en (suponga despreciable el tiempo de comunicación/sincronización):

1. 5 procesadores.
2. 4 procesadores.

En cualquier de los dos casos, la ganancia se tiene que calcular suponiendo que se procesa una imagen con un total de  $n$  bloques de 8x8 píxeles.

**Ejercicio 2.2.10.** Se quiere implementar un programa paralelo para un multi-computador que calcule la siguiente expresión para cualquier  $x$  (es el polinomio de interpolación de Lagrange):  $P(x) = \sum_{i=0}^n (b_i \cdot L_i(x))$ , donde:

$$L_i(x) = \frac{(x - a_0) \dots (x - a_{i-1})(x - a_{i+1}) \dots (x - a_n)}{k_i} = \frac{\prod_{\substack{j=0 \\ j \neq i}}^n (x - a_j)}{k_i} \quad i = 0, 1, \dots, n$$

$$k_i = (a_i - a_0) \dots (a_i - a_{i-1})(a_i - a_{i+1}) \dots (a_i - a_n) = \prod_{\substack{j=0 \\ j \neq i}}^n (a_i - a_j) \quad i = 0, 1, \dots, n$$

Inicialmente  $k_i$ ,  $a_i$  y  $b_i$  se encuentran en el nodo  $i$  y  $x$  en todos los nodos. Sólo se van a usar funciones de comunicación colectivas. Indique cuál es el número mínimo de funciones colectivas que se pueden usar, cuáles serían, en qué orden se utilizarían y para qué se usan en cada caso.

**Ejercicio 2.2.11.**

1. Escriba un programa secuencial con notación algorítmica (podría escribirlo en C) que determine si un número de entrada,  $x$ , es primo o no. El programa imprimirá si es o no primo. Tendrá almacenados en un vector, NP, los  $M$  números primos entre 1 y el máximo valor que puede tener un número de entrada al programa.
2. Escriba una versión paralela del programa anterior para un multicomputador usando un estilo de programación paralela de paso de mensajes. El proceso 0 tiene inicialmente el número  $x$  y el vector NP en su memoria e imprimirá en pantalla el resultado. Considere que la herramienta de programación ofrece funciones `send()/receive()` para implementar una comunicación uno-a-uno asíncrona, es decir, con función `send(buffer, count, datatype, idproc, group)` no bloqueante y `receive(buffer, count, datatype, idproc, group)` bloqueante. En las funciones `send()/receive()` se especifica:

- **group**: identificador del grupo de procesos que intervienen en la comunicación.
- **idproc**: identificador del proceso al que se envía o del que se recibe.
- **buffer**: dirección a partir de la cual se almacenan los datos que se envían o los datos que se reciben.
- **datatype**: tipo de los datos a enviar o recibir (entero de 32 bits, entero de 64 bits, flotante de 32 bits, flotante de 64 bits, ...).
- **count**: número de datos a transferir de tipo **datatype**.

**Ejercicio 2.2.12.** Escribir una versión paralela del programa secuencial del ejercicio 2.2.11 para un multicomputador usando un estilo de programación paralela de paso de mensajes y suponiendo que la herramienta de programación ofrece las funciones colectivas de difusión y reducción (escribir primero la versión secuencial). Sólo el proceso 0 imprimirá en pantalla. En la función de difusión, `broadcast(buffer, count, datatype, idproc)` se especifica:

- **group**: identificador del grupo de procesos que intervienen en la comunicación, todos los procesos del grupo reciben.
- **idproc**: identificador del proceso que envía.
- **buffer**: dirección de comienzo en memoria de los datos que difunde **idproc** y que almacenará, en todos los procesos del grupo, los datos difundidos.
- **datatype**: tipo de los datos a enviar/recibir (entero de 32 bits, entero de 64 bits, flotante de 32 bits, flotante de 64 bits, ...).
- **count**: número de datos a transferir de tipo **datatype**.

En la función de reducción, `reduction(sendbuf, recvbuf, count, datatype, oper, idproc, group)`, se especifica:

- **group**: identificador del grupo de procesos que intervienen en la comunicación, todos los procesos del grupo envían.
- **idproc**: identificador del proceso que recibe.
- **recvbuf**: dirección en memoria a partir de la cual se almacena el escalar resultado de la reducción de todos los componentes de todos los vectores **sendbuf**.
- **sendbuf**: dirección en memoria a partir de la cual almacenan todos los procesos del grupo los datos de tipo **datatype** a reducir (uno o varios).
- **datatype**: tipo de los datos a enviar y recibir (entero de 32 bits, entero de 64 bits, flotante de 32 bits, flotante de 64 bits, ...).
- **oper**: tipo de operación de reducción. Puede tomar los valores OR, AND, ADD, MUL, MIN, MAX
- **count**: número de datos de tipo **datatype**, del buffer **sendbuffer** de cada proceso, que se van a reducir.

**Ejercicio 2.2.13.**

1. Escribir una versión paralela del programa paralelo del ejercicio 2.2.12 suponiendo que, además de las dos funciones colectivas anteriores, se dispone de dispersión y que  $M$  es divisible entre el número de procesos (escribir primero la versión secuencial). Sólo el proceso 0 imprimirá en pantalla. La función `scatter(sendbuf, sendcnt, recvbuf, recvcnt, datatype, idproc, group)` especifica:
  - **group**: identificador del grupo de procesos que intervienen en la comunicación, todos los procesos del grupo envían.
  - **idproc**: identificador del proceso que envía.
  - **recvbuf**: dirección en memoria a partir de la cual se almacenan los datos recibidos.
  - **sendbuf**: dirección en memoria a partir de la cual almacena el proceso `idproc` los datos a enviar.
  - **datatype**: tipo de los datos a enviar y recibir.
  - **recvcnt**: número de datos de tipo `datatype` a recibir en `recvbuf`.
  - **sendcnt**: número de datos de tipo `datatype` a enviar.
2. ¿Qué estructura de procesos/tareas implementa el código paralelo del apartado 1? Justifique su respuesta.

**Ejercicio 2.2.14.** Escribir una versión paralela del programa secuencial del ejercicio 2.2.11 para un multiprocesador usando el estilo de programación paralela de variables compartidas; en particular, use OpenMP (escribir primero la versión secuencial).

**Cuestión 2.2.1.** Indique las diferencias entre OpenMP y MPI.

**Cuestión 2.2.2.** Ventajas e inconvenientes de una asignación estática de tareas a procesos/threads frente a una asignación dinámica.

**Cuestión 2.2.3.** ¿Qué se entiende por escalabilidad lineal y por escalabilidad superlineal? Indique las causas por las que se puede obtener una escalabilidad superlineal.

**Cuestión 2.2.4.** Enuncie la ley de Amdahl en el contexto de procesamiento paralelo.

**Cuestión 2.2.5.** Deduzca la expresión matemática que se suele utilizar para caracterizar la ley de Gustafson. Defina claramente y sin ambigüedad el punto de partida que va a utilizar para deducir esta expresión y cada una de las etiquetas que utilice. ¿Qué nos quiere decir Gustafson con esta ley?

**Cuestión 2.2.6.** Deduzca la expresión que caracteriza a la ley de Amdahl. Defina claramente el punto de partida y todas las etiquetas que utilice.