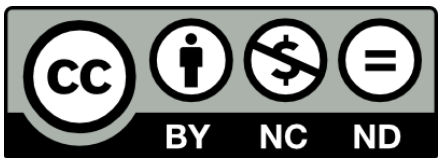
Arquitectura de Computadores

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Arquitectura de Computadores

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2023-2024

Índice general

1. Arquitecturas Paralelas

1.1. Clasificación del paralelismo implícito en una aplicación

1.1.1. Objetivos

Como ítems 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 una aplicación, lo que nos 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 con 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 transformar 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
: Las funciones llamadas en un programa se pueden ejecutar en paralelo, siempre que no haya dependencias (riesgos) inevitables entre ellas, 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 podemos practicar con 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 iteaciones de un bucle, siempre que eliminen los problemas derivados de las dependencias de datos verdaderas (RAW), en caso de haberlas.

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 secuencia al 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 posiblidad 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 bucles 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 instrución 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 ejecutándose al mismo tiempo; 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 del 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 código:

```
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 (en este ejemplo).

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 ver 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 para distinguir casuísticas, 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 proceso asociado a él,

y ya sólo dependerá del sistema operativo el llevar a cabo su paralelización con el resto de procesos[1] (creando así paralelismo entre programas). Para poder implementar este tipo de paralelismo, es necesario disponer de un multiprocesador, multicomputador, o cualquier sistema que nos permita ejecutar dos procesos de forma simultánea.

Nivel de funciones

El paralelismo a este nivel puede extraerse para realizarse a nivel de procesos (si la función realmente lo require) 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). En definitiva: crear varios entes del sistema operativo de forma que cada uno ejecute una o varias funciones.

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 permita 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, las cuales pueden conseguirse mediante replicado de componentes del procesador o segmentación.

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 lo detecta, cómo se implementa y en qué unidad se ejecuta. En este caso, la granularidad es inversamente proporcional a la facilidad de extracción del paralelismo.

[1] Mediante técnicas ya vistas en la asignatura de Sistemas Operativos.

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 principalmmente por arquitecturas ILP, que lo hacen usando instrucciones dedicadas a ello.

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 intra-instrucciones en el caso de aquellas paralelizaciones vectoriales ya comentadas; mientras que paralelizaciones del estilo 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 sóla 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 empotrados.

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 datos, 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 procesadsor 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); o NORMA (No Remote Memory Access), 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 proceador 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 uso de los multicomputadores de la red de interconexión, mientras que los multiprocesadores deben usarla 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 planteando 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 probabilida 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 esta 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 no es fácil 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 herramientras 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 hallá de un bus). Sin embargo, tratamos de buscar una solución que nos aporte más beneficios, que probablemente surja 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.

1.3.2. Definiciones

Tiempo de respuesta.
El tiempo de respuesta (elapsed) es el tiempo transcurrido entre que se lanza la ejecución de un programa y se tienen sus resultados.

Productividad
La productividad es el número de entradas procesadas por unidad de tiempo. A mayor sea el número de entradas que un computador pueda procesar a la vez, mayor será su productividad. Por tanto, calculamos la productividad mediante la siguiente fórmula:

$$P(n) = \frac{n}{t} \tag{1.1}$$

Donde n es el número de entradas y t el tiempo en el que las ha procesado. Notemos que en un computador que no implementa paralelismo, la productividad es la inversa del tiempo, al no poder procesar más de una entrada al mismo tiempo.

Tiempo de CPU.
Este tiempo está incluido dentro del tiempo de respuesta. Se trata del tiempo que el procesador dedica a ejecutar instrucciones máquina de su repertorio, tanto en modo de usuario como las que corresponden a la actividad que se debe llevar a cabo por el sistema operativo para permitir la ejecución del programa. Sólo se tiene en cuenta el tiempo asociado con la ejecución de las instrucciones relativas al programa. Es común diferenciar dentro del tiempo de CPU el *Tiempo de CPU de usuario* (user) y el *Tiempo de CPU de sistema* (sys), cuyos nombres son autoexplicativos.

Notemos que entre el tiempo de respuesta y tiempo de CPU hay una diferencia de tiempo presente. Este puede deberse a:

- Tiempo de espera debido a las E/S.

- Tiempo de ejecución de otros programas que comparten procesador con el nuestro.

Estos dos tiempos también se incluyen en el tiempo de ejecución, pero no en el de CPU.

1.3.3. Tiempo de CPU

A lo largo de esta sección, nos centraremos exclusivamente en el estudio de tiempo de CPU. Para simplificar el estudio de este tiempo, suponemos que tanto el tiempo de espera por E/S como el tiempo de ejecución de otros programas en el procesador son despreciables. Por tanto, para nosotros, el tiempo de CPU será igual al tiempo de respuesta (de forma práctica, no teórica). Hecha esta asunción, podemos calcular el tiempo de CPU como:

$$T_{CPU} = \text{Ciclos del programa} \cdot T_{ciclo} = \frac{\text{Ciclos del programa}}{F} \tag{1.2}$$

Donde "Ciclos del programa" es el número de ciclos de reloj del procesador que tarda en ejecutarse el programa, T_{ciclo} el tiempo de ciclo del procesador (habitualmente, el tiempo que tarda en ejecutarse su instrucción más costosa); y F la frecuencia de reloj:

$$F = \frac{1}{T_{ciclo}} \tag{1.3}$$

Dado que el número de ciclos del programa se puede expresar en términos del número de instrucciones máquina del repertorio del procesador que se han procesado, NI, y del número medio de ciclos por instrucción, CPI, la expresión (1.2) se puede reescribir usando la relación:

$$\text{Ciclos del programa} = NI \cdot CPI \tag{1.4}$$

de donde obtenemos:

$$T_{CPU} = NI \cdot CPI \cdot T_{ciclo} = \frac{NI \cdot CPI}{F} \tag{1.5}$$

Asumiendo que el número de ciclos por instrucción es constante (es decir, todas las instrucciones tardan el mismo tiempo en ejecutarse). Esto no es realista, por lo que usaremos en su lugar el número de ciclos por instrucción medio (CPIM), que para abreviar seguiremos notando por CPI:

$$CPI = \frac{\sum_{i=1}^{W} NI_i \cdot CPI_i}{NI} \tag{1.6}$$

Donde NI_i es el número de instrucciones del tipo i que tiene el programa, CPI_i el número de ciclos del procesador que necesita una instrucción de tipo i para procesarse; y W el número de instrucciones diferentes en el programa. Sustituyendo esta nueva ecuación en (1.5) obtenemos:

$$T_{CPU} = \sum_{i=1}^{W} (NI_i \cdot CPI_i) \cdot T_{ciclo} \tag{1.7}$$

A veces, se hará referencia al número de instrucciones por ciclo (IPC). No debemos asustarnos, pues según una sencilla cuenta obtenemos:

$$CPI = \frac{1}{IPC} \tag{1.8}$$

Por ejemplo, cuando nos mencionen que en un ciclo se pueden realizar dos instrucciones de coma flontante, nos estarán diciendo que $IPC_{flotante} = 2$.

1.3.4. MIPS y FLOPS

Los MIPS (millones de instrucciones por segundo) miden el número (en millones) de instrucciones máquina ejecutadas por unidad de tiempo (que se considera el tiempo de CPU), medido en segundos. Se pueden obtener a partir de:

$$MIPS = \frac{NI}{T_{CPU} \cdot 10^6} = \frac{\cancel{NI}}{\cancel{NI} \cdot CPI \cdot T_{ciclo} \cdot 10^6} = \frac{F}{CPI \cdot 10^6} \tag{1.9}$$

Debido al incremento de las prestaciones en los computadores actuales, el tamaño de los MIPS es cada vez más grande, por lo que podemos encontrar muchas veces que esta medida se realiza en GIPS (sustituir 10^6 por 10^9 en la ecuación (1.9)).

Si vamos a usar los MIPS para comparar las prestaciones de dos ordenadores, debemos tener en cuenta que ambos deben tener el mismo repertorio de instrucciones máquina; ya que por ejemplo, si consideramos dos computadores, uno con un repertorio complejo de instrucciones y otro con otro más sencillo y suponemos que el programa tarda el mismo tiempo en ejecutarse en las dos máquinas, el computador con el repertorio más sencillo tendrá un mayor valor de MIPS (al necesitar más instrucciones que el de repertorio complejo para ejecutar el programa). Sin embargo, dado que el tiempo ha sido el mismo, podemos decir que las prestaciones en ambos son iguales. La cuestión es que los MIPS miden la velocidad con que cada procesador ejecuta las instrucciones de su repertorio. Por tanto, sólo sirven para esto.

Otra medida disponible similar a los MIPS son los MFLOPS (millones de operaciones de coma flotante por segundo), que se obtienen mediante la expresión:

$$MFLOPS = \frac{\text{Operaciones en coma flotante}}{T_{CPU} \cdot 10^6} \tag{1.10}$$

Como en el caso anterior, también podemos considerar los GFLOPS o, incluso, TFLOPS (10^{12}), PFLOPS (10^{15}), EFLOPS (10^{18}), ...

No se trata de una medida adecuada para todos los programas, ya que sólo tiene en cuenta las operaciones de coma flotante. Además, ni las instrucciones de coma flotante son iguales en todas las máquinas ni su coste de ejecución. Se usa mayormente en evaluación de computadores dedicados a cálculo científico, donde las operaciones en coma flotante abundan.

1.3.5. Ganancia

Es común en Arquitectura de Computadores detectar cuellos de botella en la arquitectura del computador y proponer estrategias que nos ayuden a mejorar las

prestaciones. Para medir el resultado de una mejora, es habitual usar la ganancia de velocidad, que compara la velocidad de un computador antes y después de mejorar alguno de sus recursos. Gracias a la siguiente expresión definimos la ganancia de velocidad, S_p:

$$S_p = \frac{V_p}{V_b} = \frac{T_b}{T_p} \tag{1.11}$$

Donde V_b es la velocidad de ejecución del programa antes de aplicar la mejora (velocidad base), V_p es la velocidad tras aplicar la mejora; y T_b y T_p son los tiempos antes de aplicar la mejora y después, respectivamente. Notemos que, aplicando la fórmula (1.5) obtenemos:

$$S_p = \frac{T^b_{CPU}}{T^p_{CPU}} = \frac{NI^b \cdot CPI^b \cdot T^b_{ciclo}}{NI^p \cdot CPI^p \cdot T^p_{ciclo}} \tag{1.12}$$

La notación p se debe a que si tomamos $b = 1$ (tiempo de referencia, que era el base), entonces estamos obteniendo una mejora que hace p veces más rápido algunos recursos del ordenador.

Ejemplos de la mejora en prestaciones son pasar de un computador no segmentado a uno segmentado, introducir unidades que permitan funcionamento superescalar, unidades que permitan funcionalidades SIMD, ...

Uno de los mayores limitantes a la hora de introducir mejoras son los accesos a memoria (muy lentos en comparación a la velocidad del procesador) y los riesgos[2] (que recordamos que pueden ser de datos, de control y estructurales).

1.3.6. Ley de Amdahl

La Ley de Amdahl establece una cota superior a la ganancia de velocidad S_p que se puede conseguir al mejorar alguno de los recursos del computador en un factor p y según la frecuencia con la que se utiliza dicha mejora:

$$S_p \leqslant \frac{1}{f + \frac{1-f}{p}} = \frac{p}{1 + f \cdot (p-1)} \tag{1.13}$$

Donde f es el porcentaje del tiempo de ejecución del sistema base durante el que no se usa el componente mejorado.

Demostración. Tenemos que:

$$S_p = \frac{T_b}{T_p} \overset{(*)}{\leqslant} \frac{T_b}{f \cdot T_b + (1-f) \cdot \frac{T_b}{p}} = \frac{1}{f + \frac{1-f}{p}}$$

Notemos que, teóricamente, en $(*)$ deberíamos haber puesto un igual en lugar de un menor o igual. La razón de este símbolo es debido a que, gracias a la mejora introducida que nos da un tiempo de T_p, esperamos un factor de mejora de p veces el tiempo base (T_b). Sin embargo, debido a distintas variables que entran en juego (recordamos que estamos trabajando con un modelo infinitamente simple de la ejecución de un computador), probablemente el tiempo de mejora no sea tan bueno

[2] Vistos ya en la asignatura de EC

como nosotros queremos. Es por eso por lo que introducimos este menor o igual, ya que en la fracción de tiempo $1 - f$ obtenemos un tiempo de mejora de a lo más p veces el base. □

Por ejemplo, si $f = 1$ (el recurso mejorado no se usa en el programa), entonces $S_p \leqslant 1$, por lo que no se produce mejora alguna. Si en cambio, $f = 0$ (el recurso mejorado se usa todo el rato), entonces S_p podría alcanzar un valor de p. Consultamos un caso intermedio:

Ejemplo. Si un programa pasa el 25 % del tiempo en una máquina ejecutando instrucciones de coma flotante y se mejora la máquina haciendo que dichas instrucciones se ejecuten en mitad de tiempo, calcule la ganancia máxima de velocidad.

En este caso, tenemos que el tiempo de ejecución de una instrucción de coma flotante se reduce a la mitad, por lo que se ha obtenido una mejora que hace a la ejecución de dichas instrucciones el doble de eficientes, luego obtenemos un factor $p = 2$. Por otra parte, como el 25 % del tiempo de ejecución del programa se debe a operaciones en coma flotante, el otro 75 % no usará estas operaciones, luego tenemos $f = 0{,}75$. Calculamos ahora la ganancia máxima de velocidad gracia a la Ley de Amdalh:

$$S_p \leqslant \frac{p}{1 + f \cdot (p - 1)} = \frac{2}{1 + 0{,}75 \cdot (2 - 1)} = \frac{2}{1{,}75} \approx 1{,}14$$

Por tanto, la ganancia máxima de velocidad será de 1,14.

1.3.7. Benchmarks

Un *benchmark* es un conjunto de programas de prueba diseñados para medir de forma fiable (evalúan distintos componentes del computador y permite comparar distintos sistemas entre sí) las prestaciones de un computador. Se usan en la fabricación, investigación y distribución de hardware (probar distintos componentes) y software (probar la eficacia de distintos sistemas operativos, o programas). Podemos encontrarnos distintos tipos de benchmarks:

Benchmark de bajo nivel o microbenchmark. Evalúan de forma genérica las prestaciones de la arquitectura o software de un ordenador, evaluando tanto el procesador como la memoria y la E/S.

Núcleos (kernels). Son trozos de código muy utilizados en diferentes aplicaciones (como resolución de sistemas de ecuaciones, multiplicación de matrices, productos escalares, ...). Junto con los microbenchmarks permiten encontrar los puntos fuertes de cada computador.

Sintéticos. Trozos de código que no permiten obtener un resultado con significado. Es la peor elección.

Programas reales. Programas disponibles comercialmente que tratan de evaluar bases de datos, servidores, ...

Aplicaciones diseñadas. Se diseñan aplicaciones que tratan de imitar a aquellas para las que se usará el computador.

2. Programación Paralela

En el capítulo anterior, nos dedicamos a introducir los conceptos de paralelismo dentro de una aplicación, tanto a nivel implícito como explícito; así como formas de llevarlo a cabo y de evaluación de las mejoras relacionadas con implementar paralelismo. A continuación, nos toca conocer a un menor nivel de abstracción cómo se programan todas estas estrategias relacionadas con el paralelismo que ya hemos desarrollado y que el lector debería comprender.

2.1. Estructuras en programación paralela

En esta sección, planteamos los aspectos particulares de las herramientas de programación paralela (aquellas que nos ayudan a desarrollar código paralelo, para poder crear aplicaciones paralelas). Haremos incapié en el trabajo extra que supone hacer una aplicación paralela frente a una secuencial, así como aprender formas de comunicación (o sincronización) que se ofertan al programador.

2.1.1. Objetivos

En esta sección, tratamos de:

- Distinguir entre los diferentes tipos de herramientas de programación paralela, como compiladores paralelos, lenguajes paralelos, APIs de directivas y APIs de funciones.
- Distinguir entre los diferenties tipos de comunicaciones colectivas.
- Diferenciar el paradigma de programación de paso de mensajes con respecto al paradigma de variables compartidas.
- Diferenciar entre OpenMP y MPI, en cuanto a su estilo de programación y tipo de herramienta.
- Distinguir entre las esctructuras de tareas junto a procesos y hebras, master-slave, cliente-servidor, descomposición de dominio, flujo de datos (o segmentación) y divide y vencerás.

2.1.2. Problemas que plantea la programación paralela

La programación paralela requiere de algún ente (ya sea la herramienta que usemos o el propio programador) que realice el trabajo necesario para llevar a cabo

el paralelismo, a diferencia de un código secuencial. Los mayores trabajos (y los más comunes) que nos encontramos a la hora de pasar de una aplicación secuencial a una paralela los desarrollaremos a continuación.

Localicación o detección de paralelismo

Para poder implementar paralelismo dentro de una aplicación, esto es, dividir las aplicaciones en unidades de cómputo independientes que recibirán el nombre de *tareas*, es necesario primero localizar de dónde podemos extraer este paralelismo. Lo más cómodo y usual será, a partir del código secuencial que resuelve nuestra aplicación (o a partir de la definición de la aplicación), analizarlo para ver de dónde podemos extraer paralelismo (y en qué parte del código es donde debemos intentar introducir paralelismo). Los grafos pueden ser una gran herramienta en esta tarea, ya que nos permiten ver las tareas que pueden ejecutarse en paralelo (estos serán los nodos a misma altura, si la altura representa el tiempo de ejecución); así como las dependencias que hay entre ellas (los datos que una tarea requiere de otra para su ejecución). Finalmente, también nos permite ver cuál es el número máximo de tareas que se ejecutarán en paralelo. A este número máximo se le suele llamar *grado de paralelismo* de una aplicación.

Asignación de carga de trabajo

Tenemos que decidir qué tareas corresponderán a qué ente del sistema operativo (como procesos o threads), así como la asignación de flujos de instrucciones a los procesadores disponibles. Cabe destacar que no suele ser rentable usar más flujos de instrucciones que procesadores ni que los flujos cambien de procesador en tiempo de ejecución. Así, las asignaciones de flujos a procesadores puede hacerse estática o dinámicamente (en tiempo de ejecución); y explícita o implícita (lo hace la herramienta de forma automática). Notemos que la asignación dinámica requiere un costo extra, lo que introduce un retardo adicional. La asignación dinámica es la única posible cuando no puede conocerse el número de tareas a ejecutar.

Comunicación o sincronización

Muchas veces necesitaremos mecanismos de comunicación entre los distintos flujos de instrucciones, ya que todos estos están colaborando en la ejecución del programa (uno puede generar una variable que otro necesite).

Un ejemplo de la necesidad de todas estas tareas la vemos reflejada en el siguiente código:

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

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

    for(int i = 0; i < intervalos; i++){
        x = (i+0.5) * ancho;
```

```
            sum += 4.0/(1.0 + x * x);
        }

        sum *= ancho;
        // ...
    }
```

Tenemos un código secuencial que se encarga de realizar una tarea determinada. Ahora, queremos hacer uso del paralelismo para disminunir el tiempo de ejecución de la tarea. Vemos cómo hacerlo en la siguiente figura, donde hemos hecho uso de la herramienta OpenMP.

```
#include <omp.h>
#define NUM_THREADS 4

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

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

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

    sum *= ancho;
    // ...
}
```

Donde primero, hemos detectado qué parte podríamos mejorar del código. En este caso, repartir las iteraciones del bucle entre distintas hebras, ya que las iteraciones no están relacionadas entre sí. A continuación, hemos decidido que vamos a usar 4 hebras, y que vamos a repartir las `intervalos` iteraciones de forma equitativa entre las 4 hebras. Finalmente, comunicamos las hebras entre sí gracias a dos palabras:

- La cláusula `reduction(+:sum)` nos permite sumar en `sum` los distintos valores de la variable `sum` de cada hebra (cabe destacar que lo podríamos haber hecho con un proceso menos automático y más personalizado, como usando directivas `critical`, aunque en este caso la mejor elección es `atomic`).
- La directiva `for` tiene una barrera implícita al final que hace que todas las hebras se esperen entre sí (tarea de sincronización).

Modos de programación

A la hora de programar una aplicación paralela, podemos distinguir dos modos de programación:

SPMD (Single-Programa Multiple Data)
: También denominado a veces paralelismo de datos, todos los códigos que se ejecutan en paralelo se obtienen compilando el mismo programa. Cada copia trabaja con un conjunto de datos distintos y se ejecuta en un procesador diferente.

MPMD (Multiple-Program Multiple Data)
: También llamado a veces paralelismo de tareas o funciones, los códigos que se ejecutan en paralelo se obtienen compilando programas independientes. En este caso, la aplicación a ejecutar (o el código secuencial inicial) se divide en unidades independientes. Cada unidad trabaja con un conjunto de datos distintos y se ejecutan en un procesador diferente.

SPMD es recomendable en sistemas masivamente paralelos. Es más fácil resolver la aplicación escribiendo un único programa. Usado en sistemas con memoria distribuida, evita la necesidad de tener que distribuir el código entre los nodos, sólo habría que distribuir datos. En la práctica, se aplica en mayor medida SPMD antes que MPMD.

En los programas paralelos se pueden utilizar combinaciones de MPMD y SPMD. La programación dueño-esclavo se puede considerar una variante del modo MPMD (se verá a lo largo de esta sección). Si todos los esclavos tienen el mismo código, sería una mezcla de MPMD y SPMD. Los programas que conforman una solución dueño-esclavo con MPMD se pueden juntar en un único programa SPMD con el uso de estructuras condicionales.

2.1.3. Herramientas para obtener código paralelo

Las herramientas de programación paralela debería permitirnos de forma explícita o implícita:

1. Localizar paralelismo: descomponer la aplicación en tareas independientes.
2. Asignar las tareas: repartir la carga de trabajo entre procesos.
3. Crear (enrolar) y terminar (desenrolar) procesos.
4. Comunicar y sincronizar procesos.
5. Asignar procesos a procesadores.

Donde este último es el SO o el hardware quien realiza esta tarea (usualmente).

Cuanto mayor sea la abstracción que desarrolle la herramienta paralela, menor serán las labores que debe desarrollar el programador de aplicaciones paralelas. La labor más difícil para la herramienta es la primera, la detección del paralelismo. Podemos realizar una clasificación de las herramientas de programación paralela en función de la abstracción en la que sitúan al programador. Las enumeramos desde el mayor al menor nivel de abstracción:

Compiladores paralelos

Un compilador paralelo pretende ser aquella automatización capaz de extraer paralelismo a nivel de bucle (paralelismo de datos) y de función (paralelismo de tareas) a partir de un código secuencial. Para ello, realizan análisis de dependencias entre bloques de código. No generan código eficiente para cualquier programa y todavía se investiga en este campo.

Lenguajes paralelos y APIs de funciones y directivas

Generalmente, los lengaujes paralelos y directivas sitúan al programador en un nivel de abstracción superior a sólo bibliotecas de funciones. Encontramos lenguajes como Occam, Ada o Java, los cuales tienen construcciones particulares y bibliotecas de funciones que requieren un compilador exclusivo. Por otra parte, las APIs mencionadas (formadas tanto por directivas para el lenguaje como por bibliotecas de funciones) nos permiten trabajar en cualquier lenguaje para el que fueron diseñadas, como `C++` o `Fortran`, en el caso de OpenMP. En este nivel, el programador es el encargado de detectar el paralelísmo implícito en la aplicación. Sin embargo, el programador no hace el reparto (la asignación directa) de este paralelismo, así de eximir al programador las tareas de creación y terminación de flujos o de detalles para comunicación. Como ventaja, es más sencillo desarrollar aplicaciones paralelas, obteniendo códigos más cortos.

APIs de funciones

Como por ejemplo Pthreads o MPI, las cuales sólo consisten en una biblioteca de funciones que se añaden a un compilador de un lenguaje sencuencial. El cuerpo de procesos y hebras es escrito en lenguaje secuencial y es el programador quien se encarga de distribuir las tareas entre los procesos, crear o gestionar procesos, e implementar la comunicación y sincronización usando funciones de la biblioteca. Como ventajas a destacar:

- Los programadores están familiarizados con los lenguajes secuenciales.
- Las bibliotecas están disponibles para todos los sistemas paralelos.
- Las bibliotecas están más cercanas al hardware y permiten dar al programador un control a más bajo nivel.
- Se pueden utilizar a la vez bibliotecas para programar con hebras y con procesos.

Lenguajes paralelos para arquitecturas de propósito específico

Como por ejemplo CUDA (de NVIDIA). Consisten en construcciones del lenguaje y bibliotecas de funciones que requieren un compilador exclusivo. El programador debe participar en todas las labores salvo quizás en la asignación de instrucciones a unidades de procesamiento. Debe tener un gran conocimiento de las arquitecturas para poder escribr el código paralelo.

Comentamos ahora que, mientras OpenMP es el estándar industrial en programación paralela (gracias al alto nivel de abstracción que provee al programador), MPI es el estándar industrial para la programación de multicomputadores. OpenMP realiza de forma automática el reparto de trabajo, mientras que en MPI es el programador quien debe llevarlo a cabo (esto es lógico, debido al estar orientado a

multicomputadores, donde el reparto de la carga de trabajo es más difícil de realizar, como ya vimos en el capítulo anterior).

2.1.4. Comunicaciones y sincronizaciones

Las herramientas para la programación paralela también pueden ofrecer al programador, además de la comunicación entre dos procesos, comunicaciones en las que intervienen múltiples procesos. Estas comunicaciones se implementan para comunicar a todos los procesos que forman parte del grupo que colabora en la ejecución de un código. En muchos casos estas comunicaciones no tienen la finalidad de transmitir datos, sino de sincronizar procesos. Es común ver en varias aplicaciones las funcionalidades de:

- Reordenar datos entre procesos.
- Difusión de datos.
- Reducir un conjunto de datos a uno solo.
- Múltiples reducciones en paralelo con el mismo conjunto de datos.
- Sincronizar múltiples procesos en un punto.

Por tanto, se intenta que las herramientas de programación paralela nos permitan implementar dichas funcionalidades mediante comunicaciones entre procesos (flujos de instrucciones). A lo largo de esta sección, cada vez que aparezca "mensaje", estaremos haciendo referencia a un dato o estructura de datos. A continuación, enumeramos los distintos tipos de comunicaciones entre procesos que podemos encontrarnos:

Comunicación múltiple uno-a-uno. Hay componentes del grupo que envían un único mensaje y componentes que reciben un único mensaje. Si todos los componentes del grupo envían y reciben, diremos que se trata de una *permutación*. Nos podemos encontrar *rotaciones* (el proceso P_i envía al proceso P_{i+1} y el P_n al P_0), *intercambios*, *barajes*, *desplazamientos*, etc. (Ver Figura 2.1).

Comunicación uno-a-todos. Un proceso envía y todos los procesos del grupo reciben el mensaje (Ver Figura 2.2). Destacamos aquí:

Difusión (broadcast). Todos los procesos reciben el mismo mensaje.

Dispersión (scatter). Cada proceso recibe un mensaje diferente.

Comunicación todos-a-uno. Todos los procesos en el grupo envían un mensaje a un único proceso (Ver Figura 2.3). Destacamos:

Reducción. Los mensajes enviados por los procesos se combinan en un sólo mensaje mediante algún operador (como por ejemplo, el proceso recibe una suma de distintas variables de cada proceso).

Acumulación (gather). Los mensajes se reciben de forma concatenada en el receptor.

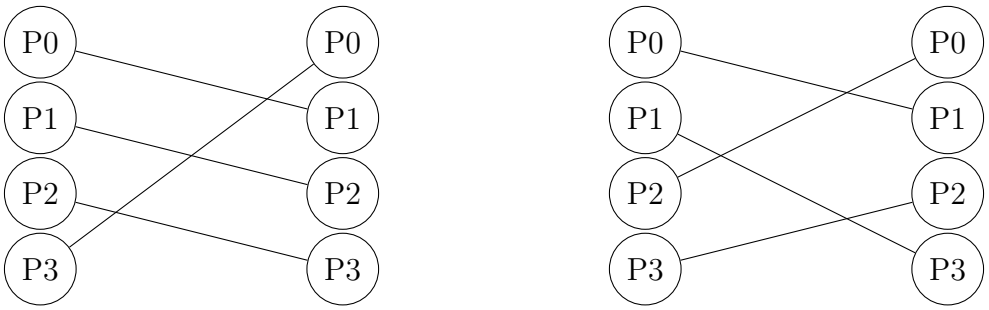


Figura 2.1: Dos permutaciones, la primera de ellas rotación.

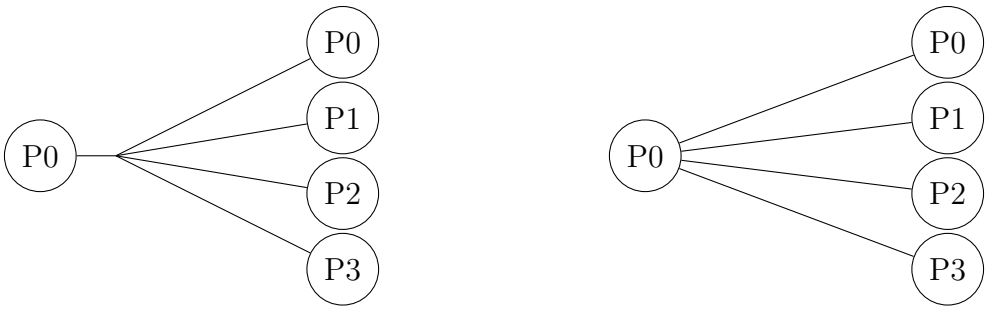


Figura 2.2: Una difusión a la izquierda y una dispersión a la derecha.

Comunicación todos-a-todos. Todos los procesos del grupo ejecutan una comunicación uno-a-todos (Ver Figura 2.4). Puede ser:

Todos difunden (all-broadcast). Todos los procesos realizan una difusión.

Todos dispersan (all-scatter). Todos los procesos realizan una dispersión.

Comunicaciones colectivas compuestas. Hay servicios que resultan de la combinación de algunos anteriores, como:

Todos combinan, o reducción y extensión. El resultado de aplicar una reducción se obtiene en todos los procesos (Ver Figura 2.5).

Barrera. Es un punto de sincronización que todos los procesos de un grupo deben alcanzar para que cualquier proceso del grupo pueda continuar con su ejecución.

Recorrido (scan). Todos los procesos envían un mensaje, recibiendo cada uno de ellos el resultado de reducir un conjunto de estos mensajes (Ver Figura 2.6).

- Recorrido prefijo: El proceso P_i recibe el resultado de reducir los mensajes de $P_0, P_1, \ldots, P_i$.
- Recorrido sufijo: El proceso P_i recibe el resultado de reducir los mensajes de $P_i, P_{i+1}, \ldots, P_n$.

Comunicaciones como "dispersión" o "todos dispersan" son usadas para reparto de datos. "Acumulación" es usada para fusionar datos intermedios. Los "desplazamientos" son tramos intermedios para realizar nuevos repartos de datos.

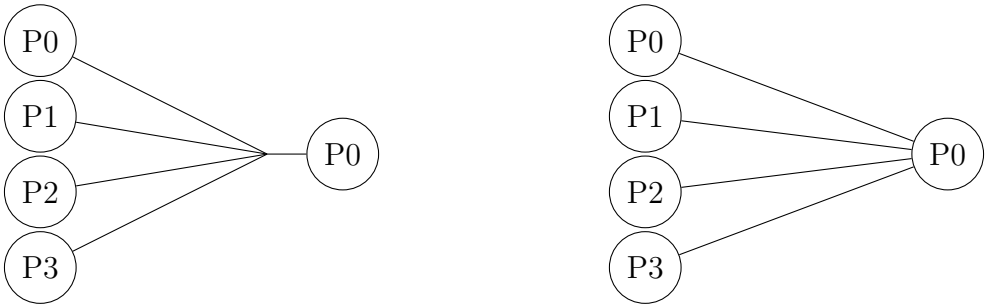


Figura 2.3: Una reducción a la izquierda y una acumulación a la derecha.

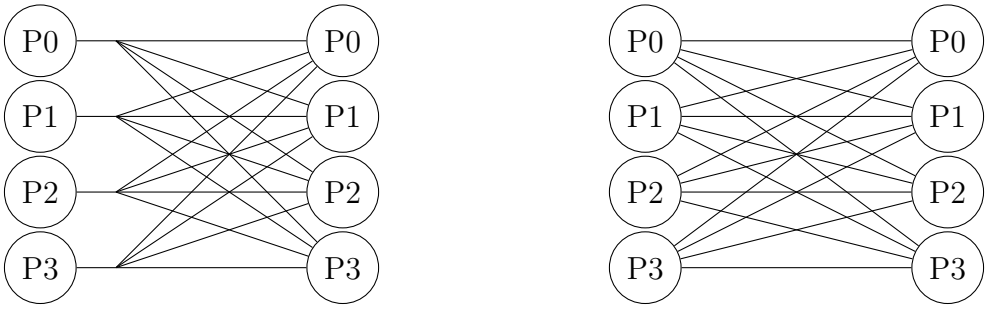


Figura 2.4: Todos difunden a la izquierda y todos dipersan a la derecha.

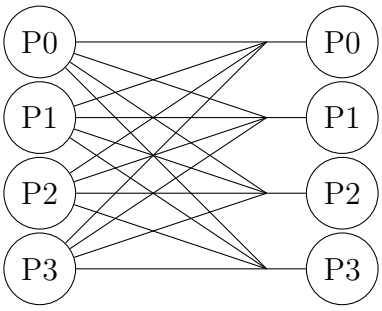


Figura 2.5: Todos combinan.

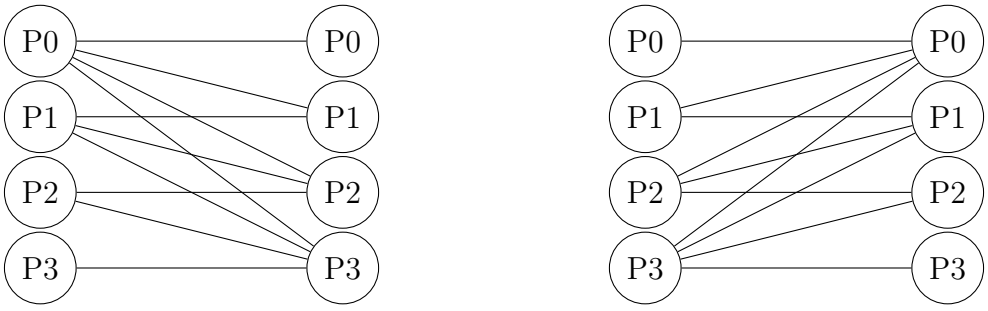


Figura 2.6: Recorrido en prefijo a la izquierda y recorrido en sufijo a la derecha.

Implementación en OpenMP

Varias comunicaciones de las anteriormente descritas podemos llevarlas a cabo usando la herramienta OpenMP:

Uno-a-todos Podemos llevar a cabo la difusión (lo haremos en la Sesión II de prácticas) con:

- La cláusula `firstprivate` desde el thread 0.
- La directiva `single` con la cláusula `copyprivate`.
- La directiva `threadprivate` y uso de la cláusula `copyin` en directiva `parallel` desde thread 0.

Todos-a-uno Podemos implementar reducción (lo haremos en la Sesión II de prácticas) con la cláusula `reduction`.

Servicios compuestos En la Sesión I de prácticas hemos usado la directiva `barrier`, que implementa una barrera.

Observemos el trabajo a alto nivel que nos facilitan las diversas directivas propias de una API de directivas y funciones.

Implementación en MPI

Por otra parte, podemos implementar las comunicaciones de una forma más cercana al hardware (a más bajo nivel) con una API de funciones tal y como lo es MPI:

Uno-a-uno De forma asíncrona, con las funciones `MPI_Send()` y `MPI_Receive()`.

Uno-a-todos Podemos implementar:

- Difusión, con la función `MPI_Bcast()`.
- Dispersión, con la función `MPI_Scatter()`.

Todos-a-uno Como:

- Reducción, con la función `MPI_Reduce()`.
- Acumulación, con la función `MPI_Gather()`.

Todos-a-todos Podemos implementar "todos acumulan" con la función `MPI_Allgather()`.

Servicios compuestos Como:

- Todos combinan, con la función `MPI_Allreduce()`.
- Barreras, con la función `MPI_Barrier()`.
- Scan, con la función `MPI_Scan()`.

2.1.5. Paradigmas de programación paralela

Cada tipo de arquitectura paralela (según la taxonomía de Flynn anteriormente estudiada) presenta distintas implementaciones en cuanto a su diseño se refiere. Es por esto que para cada tipo de arquitectura buscaremos un tipo de código que mejor se adapte a su diseño. Destacamos tres principales paradigmas en cuanto a programación paralela, cada uno asociado a un tipo de arquitectura:

- Paso de mensajes (para multicomputadores).
- Variables compartidas (para multiprocesadores).
- Paralelismo de datos (para computadores SIMD).

Con *paso de mensajes* se supone que cada procesador del sistema tiene su propio espacio de direcciones. Los mensajes llevan datos de un espacio de direcciones a otro y pueden aprovecharse para sincronizar procesos. Los datos transferidos estarán duplicados en el sistema de memoria. Con *variables compartidas*, se supone que los procesasdores comparten espacio de direcciones. Se realiza la transferencia de forma implícita usando instrucciones de lectura y escritura en memoria. Con *paralelismo de datos* la misma instrucción se ejecuta en paralelo en múltiples cores de forma que cada uno actúa sobre un conjunto de datos distinto. Este paradigma es apropiado para aquellas arquitecturas que sólo soportan paralelismo a nivel de bucle. La sincronización se encuentra implícita.

Asimismo, también podemos encontrar herramientas que permiten programar multiprocesadores mediante paso de mensajes, un software que se apoya en el hardware para variables compartidas. De igual forma, hay herramientas que permiten variables compartidas en multicomputadores. Hay lenguajes que soportan paralelismo de datos, tanto en multiprocesadores como en multicomputadores.

Paso de mensajes.

Se dispone de diversas herramientas software, como lenguajes de programación (Ada, Occam, ...) o bibliotecas de funciones (como MPI). Los fabricantes de supercomputadores suelen proporcionar este tipo de software, capaz de extraer un gran rendimiento de sus máquinas. Las funciones básicas de comunicación suelen ser `send()` y `receive()`. Generalmente, en la función `send` se especifica el proceso destino y el mensaje a enviar, mientras que en `receive` se especifica la fuente y la estructura de datos en la que se almacenará el mensaje. Podemos encontrar implementaciones *síncronas* (el proceso que ejecuta un `send` se bloquea hasta que el destinatario hace uso de `receive` y viceversa) o *asíncronas* (`send` no tiene por qué bloquear el proceso). Para esta última, es necesario usar un buffer en su implementación.

Variables compartidas.

Encontramos software como lenguajes de programación (Ada 95 o Java), bibliotecas de funciones y APIs de directivas y funciones (como OpenMP). Los propios fabricantes ofrecen compiladores secuenciales con estas extensiones para programar sus máquinas. Para la comunicación se suelen desarrollar instrucciones de *lectura* y *escritura* en memoria, las cuales serán usadas por distintos procesos (en el SO, las hebras comparten memoria entre sí, por lo que no es

necesario que usen estas instrucciones). El software ofrece mecanismos para implementar sincronización (para que un proceso no lea antes de que el otro escriba), como cerrojos, semáforos, variables condicionales, ... OpenMP dispone además de directivas para llevar a cabo paralelismo de datos (directiva `for`), de tareas (directiva `sections`), y muchas más funcionalidades.

Paralelismo de datos.
En este paradigma se aprovecha el paralelismo de datos inherente a aplicaciones en la que los datos se organizan en estructuras como vectores o matrices. El programador escribe un programa con construcciones que permiten aprovechar paralelismo de datos (como paralelización de bucles, instrucciones vectoriales, ...), así como construcciones para distribuir los datos (la carga de trabajo) entre los núcleos de procesamiento. El programador no lleva a cabo las sincronizaciones (se encuentran implícitas). Ejemplos software son Fortran 95, HPF (High Performance Fortran), NVIDIA CUDA, ...

2.1.6. Estructuras típicas de códigos paralelos

Analizando las estructuras (los grafos) de las tareas y de los procesos (junto con las conmunicaciones y sincronizaciones entre estos) que componen distintos programas paralelos, se puede encontrar que hay ciertos patrones que se repiten en distintos programas y dentro de un programa. Entre estras estructuras encontramos:

- Dueño-esclavo (*master-slave*), o "granja de tareas" (*task-farming*).
- Cliente-servidor (*client-server*).
- Paralelismo de datos, descomposición de datos, o descomposición de dominio.
- Estructura segmentada (*pipeline*), o flujo de datos.
- Divide y vencerás (*divide and conquer*), o descomposición recursiva.

Hay estructuras que quizás no se puedan clasificar en un sólo tipo de los anteriores. Por otra parte, nos podemos encontrar dentro de un mismo programa paralelo varias de estas estructuras, en distintos niveles.

Dueño-esclavo

En este caso, contamos con un proceso denominado dueño (o master) y con varios procesos denominados esclavos (o slaves). El dueño se encarga de distribuir las tareas de un conjunto entre el grupo de esclavos, y de ir recolectando los resultados parciales que van calculando los esvlavos, con los que el dueño obtiene el resultado final. Usualmente, no hay comunicación entre los esclavos. Puede implementarse con un único programa si el código de los esclavos no es muy complicado (SPMD), con dos programas, si el código de los esclavos es igual (modo mixto MPMD-SPMD), o con múltiples programas (MPMD). La repartición de carga puede hacerse de forma estática o dinámica. Un ejemplo de esto es un programa que tiene que calcular los primos hasta un número n, y que hace uso de r esclavos para que cada uno calcule

los primos que se encuentran en un vector de tamaño $^n/_r$. Combinando el resultado de cada esclavo, obtendremos todos los primos.

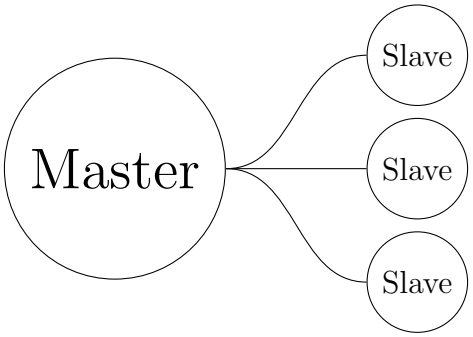


Figura 2.7: Representación gráfica dueño-esclavo.

Cliente-Servidor

También llamado *client-server*, se trata de una estructura similar a dueño-esclavo, pero con los roles invertidos: contamos con múltiples procesos denominados clientes y con un ente (puede ser un proceso, o tener a su vez una estructura paralela distinta en su interior) denominado servidor. En este caso, los clientes son procesos que en un momento determinado solicitan cierta información al servidor. Este, se encarga de procesar la petición y de dar al cliente correspondiente la respuesta esperada. Notemos que el servidor puede estar formado a su vez de otras estructuras, como de un sistema dueño-esclavo. Por tanto, podemos tener por un lado a los clientes, que necesitan del servidor información. Este es el encargado de gestionar los mensajes y de repartir las tareas entre sus esclavos, quienes realizan el cómputo, devolviendo el trabajo al maestro y este a los clientes.

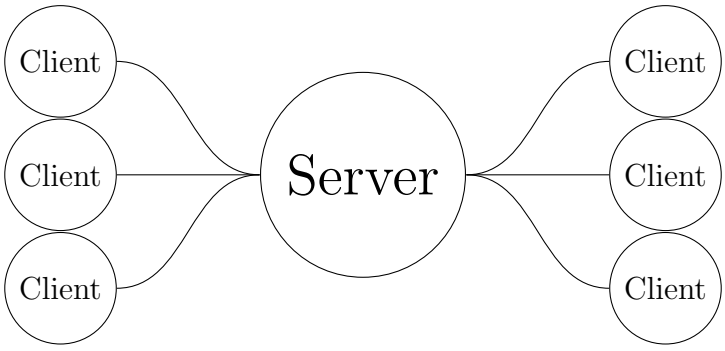


Figura 2.8: Representación gráfica cliente-servidor.

Descomposición de datos

Alternativa muy utilizada para obtener tareas paralelas en programas con grandes estructuras de datos. La estructura de datos de entrada (o la de salida, o ambas) es dividida en varias partes. A partir de esta división se derivan las tareas paralelas. Estas generalmente realizan operaciones similares. Los algoritmos con imágenes, por ejemplo, admiten una descomposición de datos. En este esquema, cada proceso puede englobar una o varias tareas. Los diferentes procesos ejecutan normalmente

el mismo código (SPMD), que se ejecuta sobre distintos conjuntos de datos. Puede haber comunicaciones entre los distintos procesos. Un ejemplo de descomposición de datos es el siguiente código, donde se invierten los colores de una imagen de tamaño $N \times N$:

```
for(int i = 0; i < N; i++)
    #pragma omp parallel for
    for(int j = 0; j < N; j++){
        imagen[i][j] = 255 - imagen[i][j];
    }
```

Estructura segmentada o flujo de datos

Esta estructura aparece en problemas en los que se aplican funciones a un flujo de datos en secuencia distintas (paralelismo de tareas). La estructura de los procesos y de las tareas es la de un cauce segmentado: cada proceso ejecuta por tanto distinto código. Es un caso típico de un programa MPMD puro. Para que resulte apropiada la estructura segmentada, se ebe aplicar el proceso a una secuencia de datos de entrada. Por ejemplo, podemos tener un decodificador de imágenes JPEG, que aplica a una secuencia de $N \times N$ píxeles (la imagen de entrada) las siguientes funciones: decodificación de entropía, cuantificación inversa, transformada del coseno inversa y conversión RGB. Este programa podría implementarse con cuatro procesos en una estructura segmentada. En este tipo de estructura, la comunicación entre procesos es necesaria, y suele ser unidireccional.

Figura 2.9: Representación gráfica de una estructura segmentada.

Divide y vencerás

Se utiliza cuando un problema puede dividirse en dos o más subproblemas de menor tamaño de forma que cada uno pueda resolverse independientemente y combinar al final las distintas soluciones. Si los subproblemas son a su vez instancias más pequeñas del problema original, entonces se podrán volver a subdividir de forma recursiva. Las tareas presentan, por tanto, una estructura de árbol, de forma que no habrá interacciones (comunicaciones o sincronizaciones) entre las tareas que cuelgan del mismo padre. Un ejemplo de uso de esta estructura podemos verlo cuando nos disponemos a sumar todas las componentes de un vector: podemos dividir el vector en 2, obteniendo dos subproblemas de tamaño $n/2$, siendo n el número de componentes del vector. Podemos seguir recursivamente dividiendo el vector hasta obtener vectores de tamaño relativamente pequeños, donde haríamos su suma iterativamente. Obtenemos así al final la suma de todas las componentes del vector.

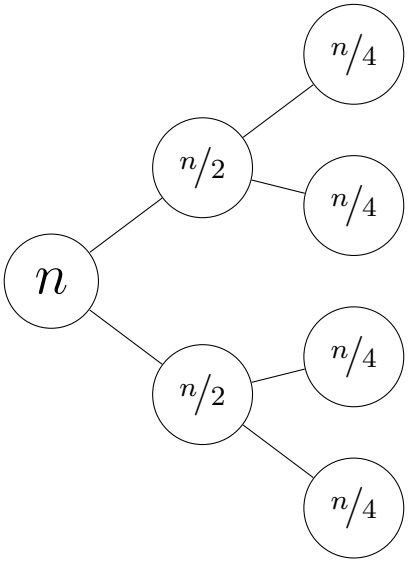


Figura 2.10: Representación gráfica de divide y vencerás.

2.2. Proceso de paralelización

Cuando queremos obtener una versión paralela de una aplicación con una biblioteca de funciones con directivas o con un lenguaje de programación, pueden seguirse los siguientes pasos:

- Descomposición de la aplicación en tareas independientes.
- Asignación de las tareas a procesos o hebras.
- Redactar el código paralelo.
- Evaluación de los tres pasos anteriores.

Estudiaremos a lo largo de esta sección todos estos pasos a fondo, junto con un ejemplo práctico.

2.2.1. Objetivos

Esta sección está dedicada a:

- Programar en paralelo una aplicación sencilla.
- Distinguir entre asignación estática y dinámica, destacando sus ventajas e inconvenientes.

2.2.2. Descomposición en tareas

En esta etapa, tenemos que buscar unidades de trabajo en la aplicación que puedan ejecutarse en paralelo (es decir, que sean independientes). Estas unidades, junto con los datos que usan, forman las *tareas*. Es conveniente en este paso representar las tareas y las relaciones entre ellas mediante un grafo, para observar las dependencias entre estas y el nivel de paralelización de la aplicación. Podemos situarnos en dos niveles de abstracción:

Nivel de función. Analizando las dependencias entre las funciones del código podemos encontrar aquellas que son independientes (o aquellas que pueden hacerse independientes), que serán las que se ejecutarán en paralelo. Estamos extrayendo paralelismo de tareas en nuestra aplicación.

Nivel de bucle. Analizando las iteraciones de los bucles dentro de una función, podemos encontrar si son (o se pueden hacer) independientes. Podemos así detectar paralelismo de datos. Además, si una función consta de varios bucles, puede verse la relación entre estos, para así ver si pueden ejecutarse en paralelo.

Ejemplo

Para paralelizar el cálculo de π, puede partirse de una versión secuencial disponible o de un planteamiento para el problema que se preste a paralelización. Procedemos pues a realizar este según el segundo método:

Podemos definir el cálculo de π como un problema de integración, susceptible de ser paralelizado. Podemos calcular la integral definida en el intervalo $[0, 1]$ de la derivada de la arcotangente de x:

$$\left.\begin{array}{rcl} \text{arctg}'(x) & = & \dfrac{1}{1+x^2} \\ \\ \text{arctg}(1) & = & \pi/4 \\ \text{arctg}(0) & = & 0 \end{array}\right\} \Longrightarrow \int_0^1 \frac{dx}{1+x^2} = [\text{arctg}(x)]_0^1 = \frac{\pi}{4} - 0$$

Y así multiplicar por 4 para obtener π. Esta integral puede obtenerse mediante métodos de integración numérica (rectángulo izquierdo, rectángulo derecho, punto medio, trapecio, fórmula de Simpson, ...). Podemos dividir el área de la derivada del arcotangente en $[0, 1]$ en subintervalos, calculando el área de la función en cada subintervalo y sumando. Cuantos más subintervalos tengamos, más exacta será la aproximción. Notemos que el cálculo del área de los diferentes intervalos es independiente, por lo que podemos repartir estos cálculos en un conjunto de procesadores. Si se divide el intervalo en 100 subintervalos y tenemos 10 procesadores, podemos asignar a cada procesador el cálculo de 10 subintervalos. Sobre código, tenemos lo siguiente:

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

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

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

    sum *= ancho;
}
```

Para su paralelización, tratamos de determinar si las iteraciones del bucle son independientes. Como podemos ver, entre las iteraciones existen dependencias, ya que se escribe y se lee en la misma variable `sum`. Sin embargo, estas dependencias pueden eliminarse fácilmente: basta suponer que cada iteración distinta escribe en una variable que depende de `i`. La suma de todas estas variables dará la aproximación de π.

2.2.3. Asignación de tareas

En esta etapa, tratamos de realizar la asignación de las tareas del grafo de dependencias a procesos y a hebras. Por lo general, no es conveniente asignar más de un proceso o hebra por procesador dentro de una misma aplicación (para que no compitan entre ellos). Por tanto, la asignación a procesos o hebras está ligada a

la asignación en procesadores. Podemos incluso realizar asignaciones de procesos a procesadores.

La granularidad de los procesos y las hebras depende del número de procesadores: cuanto mayor sea, menos tareas se asignarán a cada proceso (o hebra). Además, la granularidad depende del tiempo de comunicación y sincronización, debido a que el tiempo de ejecución en paralelo no sólo depende del código en sí mismo, sino de las comunicaciones y sincronizaciones entre procesos. Para disminuir este tiempo, pueden asignarse más tareas a un proceso (o hebra), reduciendo así el número de interacciones (comunicaciones o sincronizaciones) entre las tareas a través de la red de comunicación.

La posibilidad de usar procesos o hebras depende de varios factores:

- La arquitectura en la que se va a ejecutar: en un SMP y en procesadores multihebra es más eficiente usar hebras[1]. En arquitecturas mixtas (como cluster de SMP), sería conveniente usar tanto hebras como procesos, especialmente si el número de procesadores en un SMP es mayor que el número de nodos en un cluster.

- El sistema operativo debe ser multihebra para poder utilizar hebras (a no ser que las emulemos mediante una biblioteca de hebras, lo que introduce un costo adicional).

- Para usar hebras (también para procesos), las herramientas de programación que utilicemos deben permitir poder crearlas.

Como regla general, se tiende a asignar las iteraciones de un bucle (paralelismo de datos) a hebras y las funciones (paralelismo de tareas) a procesos.

Tener en cuenta la arquictura

La asignación debería repartir la carga de trabajo (el tiempo de cálculo), optimizando la comunicación y la sincronización entre las tareas (minimizándolas), de forma que todos los procesadores empiecen y terminen a la vez (idealmente). En el grafo de tareas previamente mencionado, veíamos tiempos de ejecución aproximados (sabiendo qué trabajo le estamos dando a qué tarea) y sincronizaciones entre estas (visualizando los arcos entre los nodos). Para realizar la asignación cumpliendo todo esto, conviene tener clara la arquitectura (prestaciones de los nodos de ejecución y de la red de comunicación).

Por un lado, la arquitectura puede ser **heterogénea** u **homogénea**. Si es heterogénea, consta de componentes con diferentes prestaciones. Por tanto, una buena repartición de carga de trabajo es asignar a los nodos de cómputo más rápido las tareas más pesadas.

Una arquitectura homogénea (aquella donde los nodos de cómputo son iguales, o casi) puede ser a su vez **uniforme** o **no uniforme** (decimos que las arquitecturas uniformes y no uniformes están dentro de las homogéneas porque las heterogéneas son normalmente no uniformes, luego no merece la pena concretar esos casos):

[1] Como bien ya sabemos gracias a Sistemas Operativos.

Uniforme

Si es uniforme, la comunicación de los procesadores con memoria (y probablemente con otros componentes, como E/S) supone el mismo tiempo sean cuales sean los dos componentes que intervienen. Las arquitecturas homogéneas uniformes son en las que menos tenemos que tener en cuenta la arquitectura para la asignación de tareas. Sin embargo, hay arquitecturas homogéneas uniformes en las que una buena asignación puede disminuir el número de colisiones en el acceso a la red.

No uniforme

Si es no uniforme, la velocidad de comunicación entre procesadores con memoria (y probablemente con otros componentes) depende del procesador que queramos que acceda a qué posición de memoria. En este tipo de arquitecturas es más difícil asignar las tareas (código y datos) de forma que se minimice el tiempo de comunicación y sincronización y, en general, el tiempo de ejecución. Pueden eliminarse comunicaciones asignando varias tareas al mismo procesador.

Asignar tareas a procesadores

La asignación de tareas (procesos o hebras) a procesadores puede realizarse de forma **estática** (en tiempo de compilación o al escribir el programa) o **dinámica** (en tiempo de ejecución). La asignación dinámica es útil para evitar repartos complejos de tareas en sistemas heterogéneos, especialmente cuando no se conoce el número total de tareas que se van a ejecutar antes de la ejecución. Además, permite que la aplicación se ejecute exitosamente si algún procesador falla. Sin embargo, esta asignación consume un tiempo adicional de sincronización y comunicación.

Un ejemplo de asignación estática es asignar iteraciones consecutivas de un mismo bucle a procesos consecutivos (*round-robin*), mientras que también podemos asignar iteraciones consecutivas a un mismo proceso (continua), como es el caso del bucle `for` en OpenMP. Un ejemplo de asignación dinámica es almacenar en una variable el índice de la siguiente iteración a ejecutar, de forma que el primer procesador desocupado lea el índice que le indique qué iteración ejecutar, aumentando su valor para que el siguiente procesador desocupado realice la siguiente. Notemos que este acceso a una variable que indique la siguiente iteración debe realizarse en exclusión mutua (por ejemplo, usando un cerrojo). Con paso de mensajes, la asignación dinámica podría hacerse desde el proceso dueño en un sistema dueño-esclavo. Hay herramientas como OpenMP que permiten que el procesador deje al compilador la asignación de tareas a procesos (o hebras).

Ejemplo. Si retomamos el ejemplo de la sección anterior en un sistema (por ejemplo) homogéneo uniforme, nos gustaría que la carga de trabajo se distribuya por igual en todos los procesadores. Por tanto, nos interesa más usar una asignación estática frente a una dinámica, debido a que esta última perjudicarían considerablemente las prestaciones debido a la comunicación adicional que añaden.

2.2.4. Código paralelo

El código que queremos desarrollar va a depender del estilo de programación que se utilice (variables compartidas, paso de mensajes, paralelismo de datos), del modo de programación (SPMD, MPMD, mixto), del punto de partida (versión secuencial, descripción del problema), de la herramienta software que usemos (lenguajes de programación, compiladores, bibliotecas de funciones, APIs de funciones y directivas, ...) y de la estructura (granja de tareas, segmentada, descomposición de datos, divide y vencerás, maestro-esclavo, ...) elegida. Todas estas a su vez dependen de la aplicación que queremos paralelizar. Habrá que añadir o utilizar en el programa las funciones, directivas o construcciones del lenguaje que hagan falta para:

1. Crear y terminar procesos (o hebras).
2. Localizar paralelismo.
3. Asignar la carga de trabajo conforme a las decisiones tomadas.
4. Comunicar y sincronizar las diferentes tareas (procesos o hebras).

Ejemplo implementado con OpenMP

Hemos decidido implementar el ejemplo que venimos realizando a lo largo de esta sección mediante el estilo de variables compartidas, gracias al modelo que ofrece OpenMP con directivas. Podemos programar con OpenMP a un mayor nivel de abstracción usando directivas (y dejando al compilador que implemente lo que le transmitimos), así como a un menor nivel usando las funciones que nos provee la librería `omp.h` de OpenMP.

Para obtener el programa paralelo a partir del secuencial ya realizado, hemos añadido a este directivas y funciones necesarias para crear y terminar hebras, asignar tareas a procesos y localizar paralelismo; y comunicar y sincronizar tareas.

```
1  #include <omp.h>
2  #define NUM_THREADS 4
3  main(int argc, char** argv){
4      double ancho, x, sum = 0;
5      int intervalos, i;
6      intervalos = atoi(argv[1]);
7      ancho = 1.0 / (double) intervalos;
8
9      omp_set_num_threads(NUM_THREADS)
10     #pragma omp parallel
11     {
12         #pragma omp for reduction(+:sum) private(x) schedule(dynamic)
13         for(int i = 0; i < intervalos; i++){
14             x = (i + 0.5) * ancho;
15             sum += 4.0 / (1.0 + x*x);
16         }
17     }
18     sum *= ancho;
19 }
```

Crear y terminar hebras. En el programa OpenMP se incluyen funciones que permiten la creación dinámica del grupo de hebras que van a intervenir en el cálculo y que permiten fijar el número de hebras que se crean:

- `omp_set_num_threads()` es una función OpenMP que modifica en tiempo de ejecución el número de hebras que se pueden utilizar en paralelo. Para ello, se sobrescribe el contenido de la variable `OMP_NUM_THREADS`, que contiene dicho número.
- `#pragma omp parallel` es una directiva OpenMP que crea un conjunto de hebras. El número será el que tenga la variable `OMP_NUM_THREADS` menos uno, ya que también interviene la hebra `master` en el cálculo. El código de las hebras creadas es el que sigue a esta directiva, que puede ser una o varias sentencias.

Asignar tareas a procesos y localizar paralelismo. El programador ha dejado el trabajo de repartir las tareas al compilador, simplemente le ha especificado una distribución dinámica, mediante la cláusula `schedule(dynamic)`.

- `#pragma omp for` es una directiva que identifica un conjunto de trabajo compartido iterativo paralelizable y especifica que se distribuyan las iteraciones del bucle `for` entre las hebras del grupo cuya creación se ha especificado con una directiva `parallel`. La política de distribución de las iteraciones del bucle puede especificarse a la directiva mediante la cláusula `schedule(kind[, chunk_size]);`, donde el tipo de distribución puede ser `static, dynamic, runtime` o `guided`.
- La cláusula `private(x)` hace que cada hebra tenga una copia privada de la variable `x`. Es decir, no será una variable compartida. La cláusula `reduction(+:sum)` hace que cada hebra utilice una variable local `sum` y que una vez terminada la ejecución de todas las hebras, se sumen todas las variables locales `sum`, almacenando el resultado en una variable compartida `sum`. La directiva `for` obliga a que todas las hebras se sincronicen al terminar el bucle (a no ser que se especifique `nowait`).

Usando esta directiva `for` se libera trabajo al programador, al no tener que implementar la distribución dinámica ni comunicaciones y sincronizaciones para obtener la suma total (así como las declaraciones de las variable locales `sum`). Además, incluye una barrera implícita que tampoco tiene que incluir el programador. La asignación de hebras a procesadores la hará el sistema operativo, aunque para incrementar las prestaciones, el programador puede orientarlo sobre dicha asignación.

Comunicación y sincronización. Como ya se ha mencionado, se distribuyen las iteraciones del bucle entre hebras de forma dinámica, de forma que cada una cuenta con su variable local `sum` que resulta en una variable compartida `sum` que al final contiene a la suma de todas las variables locales (todo esto gracias a la cláusula `reduction(+:sum)`). Se está utilizando una comunicación colectiva. Además, se usa una barrera implícita tras la finalización de la región afectada por `for` y tras la finalización de la región afectada por `parallel`.

2.2.5. Evaluación de prestaciones

Una vez redactado el programa paralelo, se pueden evaluar sus prestaciones. Si las prestaciones alcanzadas no son las que se requieren, se debe volver a etapas anteriores del proceso de planteamiento y programación. Probablemente se retrocederá más en la abstracción realizada cuando mayor queramos que sea la mejora a realizar. Podemos también elegir otra herramienta de programación que más se adecúe a conseguir la aplicación a desarrollar. Además, no todas estas herramientas ofrecen las mismas prestaciones: depende de su nivel de aprovechamiento de la arquitectura (a menor nivel de abstracción estas sean, mayor será su eficacia).

2.3. Evaluación de prestaciones

En la sección del capítulo anterior de evaluación de prestaciones vimos cómo evaluar mejoras realizadas en programas secuenciales. Nos toca ahora, evaluar programas paralelos, para comprobar si hemos conseguido alcanzar las prestaciones mínimas que requiere la aplicación programada gracias a la paralelización. En programacion paralela, se usa para evaluar las prestaciones de un código paralelo su tiempo de respuesta (`elapsed time` o tiempo de ejecución) y, adicionalmente, su escalabilidad y eficiencia.

2.3.1. Objetivos

Siguiendo con la evaluación en prestaciones de los computadores con aplicaciones paralelas, en esta sección aprenderemos a:

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

2.3.2. Ganancia en velocidad

La ganancia en velocidad (S) se utiliza para estudiar en qué medida se incrementan las prestaciones al ejecutar una aplicación en paralelo en un sistema con múltiples procesadores frente a un sistema uniprocesador. Este estudio puede interesarnos tanto por evaluar el computador paralelo como por evaluar la implementación paralela de la aplicación.

La ganancia en prestaciones (o ganancia de velocidad, `speedup`) con p procesadores alcanzada al aplicar paralelismo puede obtenerse como:

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

Es decir, dividiendo las prestaciones en un sistema con p procesadores frente a uno uniprocesador ($p = 1$). Puesto que las prestaciones son inversas al tiempo de respuesta, obtenemos la siguiente fórmula (que es la que nos va a interesar):

$$S(p) = \frac{T_S}{T_P(p)} \qquad \text{con} \qquad T_P(p) = T_C(p) + T_O(p) \tag{2.1}$$

Donde T_S sería el tiempo de ejecución del programa secuencial y $T_P(p)$ el tiempo de ejecución del programa paralelo con p procesadores. Para obtener T_S, se debería escoger el mejor programa secuencial que resuelva la aplicación (para sólo centrarnos en el cambio de la implementación del paralelismo).

El tiempo de ejecución en paralelo no sólo depende del tiempo de cómputo (de cálculo) en paralelo de las tareas detectadas en la aplicación ($T_C(p)$), sino también de un tiempo de penalización o sobrecarga (`overhead`) $T_O(p)$. Este tiempo de sobrecarga es debido a:

- El tiempo dedicado a comunicación y sincronización entre procesos.

- El tiempo para crear y terminar los procesos y hebras que permiten la paralelización.
- El tiempo de ejecución de operaciones añadidas en la versión paralela que no son necesarias en la secuencial.

Además, el tiempo de penalización resultante de un mal reparto de carga de trabajo en procesadores también puede incluirse en este $T_O(p)$. Tanto este tiempo como el tiempo de cómputo dependen del número de procesadores: cuanto mayor sea este, mayor será el grado de paralelismo (número máximo de tareas independientes que pueden ejecutarse en paralelo) de la aplicación aprovechado. A mayor número de procesos involucrados (debido probablemente a un mayor número de procesadores), mayor será el tiempo de sobrecarga[2].

Ejemplo. Por ejemplo, disponemos del siguiente código, un bucle de 16 iteraciones donde cada iteración del bucle tiene un tiempo medio de 1 segundo:

```
for(int i = 0; i < 16; i++){
    v[i] = f(i);
}
```

Las iteraciones no están relacionadas entre sí, luego decidimos paralelizar el bucle. Disponemos de 4 procesadores y decidimos dividir el bucle de forma que cada procesador ejecute 4 iteraciones. Tenemos por tanto, unos tiempos:

$$T_S = 16$$
$$T_C(4) = {}^{16}/_{4} = 4$$

Cabe destacar que tenemos $T_C(4) = 4$ y no $T_P(4) = 4$, ya que los costos de creación, comunicación, sincronización, borrado (y probablemente más acciones) de procesos (o hebras) se encuentran presentes. Por tanto, como esperamos un $T_O(4) > 0$, tenemos que $T_P(4) > 4$.

En el ejemplo anterior, cada iteración del bucle tardaba 1 segundo que, en comparación con los tiempos actuales de creación, comunicación, ... de procesos (o hebras) es ridículamente grande. Por tanto, podemos esperar un $T_P(4) \sim T_C(4)$. Sin embargo, si cada iteración del bucle tardase en vez de 1 segundo 1 milisegundo, ya sí tendríamos que probablemente el tiempo de ejecución tendría una diferencia relativa mayor con el tiempo de cálculo.

Escalabilidad

La representación de la ganancia en función del número de procesadores (p) nos permite evaluar la escalabilidad de una implementación o arquitectura paralela. Idealmente, esperaríamos conseguir un tiempo de ejecución paralelo de $T_P(p) = {}^{T_S}/_{p}$ (como se vio en el ejemplo anterior). Para lograrlo, se debería poder distribuir todo el programa secuencia en partes iguales entre los procesadores, y el tiempo de sobrecarga debería ser despreciable. En este caso, la ganancia sería:

$$S(p) = \frac{T_S}{T_P(p)} = \frac{T_S}{{}^{T_S}/_{p}} = p$$

[2]Si esto no queda claro, observar la enumeración superior.

de forma que la escalabilidad sería *lineal*. Para que siempre pueda ser p la ganancia independientemente del valor de p, el grado de paralelismo debe ser ilimitado, es decir, siempre podremos dividir el código entre los p procesadores posibles sea cual sea p. Debido a estas tres limitaciones, nos planteamos estudiar qué posibilidades se pueden plantear. Podemos encontrarnos con *escalabilidades superlineales*, casos donde $S(p) > p$ (no son de relevancia ahora mismo). Dos ejemplos de este suceso pueden ser:

- Estamos trabajando con una estructura de datos grande (como por ejemplo, un vector muy largo) y puede que al repartir el trabajo, reducimos los fallos de caché (un único procesador debería cargar en cada momento un subvector del vector con el que trabajamos. Sin embargo, puede darse que al repartir el vector entre los procesadores, se disminuya su tamaño y se den menos fallos en cada procesador).

- Por ejemplo, al realizar una búsqueda lineal sobre un vector, puede que el elemento a buscar se encuentre en la posición $\frac{n}{2} + 1$. Si paralelizamos el código de forma que una hebra busque en las primeras $n/2$ posiciones y el segundo en las $n/2$ restantes, la segunda hebra encontrará el elemento deseado en la primera iteración, reduciendo considerablemente el tiempo de ejecución en comparación con la implementación totalmente secuencial del mismo código.

Además, hemos considerado que todo el programa secuencial se puede paralelizar, pero en la práctica podemos encontrarnos (de forma bastante frecuente) tareas que no se puedan ejecutar en paralelo y que suponen un tiempo no despreciable. Más aún, no hemos considerado ninguna limitación en el grado de paralelismo, que sí estará limitado en la práctica (por ejemplo, un bucle de n iteraciones tiene un grado de paralelismo de, a lo sumo, n). Finalmente, tampoco hemos considerado tiempos de sobrecarga.

Generalizando lo ya visto, impondremos como mínimo que la carga paralelizable pueda repartirse por igual entre los procesadores disponibles (para simplificar), y además que el tiempo de ejecución T_S es constante (aún variando el número de procesadores). Con estas restricciones generales, procedemos a estudiar casos concretos donde ya damos cierta libertad al problema, con la finalidad de estudiar la escalabilidad de cada sistema:

1. Comenzamos suponiendo que la fracción no paralelizable del código secuencial es despreciable, que tenemos un grado de paralelismo ilimitado y un tiempo de sobrecarga despreciable. Estamos en el caso inicialmente estudiado como motivación de esta sección, el caso de **escalabilidad lineal** (o ganancia lineal):

$$S(p) = \frac{T_S}{T_P(p)} = \frac{T_S}{T_S/p} = p$$

2. Ahora, suponemos la presencia de una fracción no despreciable del código secuencial que no puede ser paralelizada. A esta fracción la llamaremos f. Además, consideramos al igual que en el punto anterior, un grado de paralelismo ilimitado y una sobrecarga despreciable. En este caso tendremos $T_P(p) = T_C(p) = f \cdot T_S + \frac{1-f}{p} \cdot T_S$ debido a que hay una fracción del código

secuencial f que no podemos paralelizar y paralelizamos la otra parte, $1 - f$, reduciendo el tiempo de forma lineal (al encontrarnos en el caso anterior), obteniendo así una ganancia de:

$$S(p) = \frac{1}{f + \dfrac{1-f}{p}}$$

Para un cierto número p de procesadores. A medida que aumentamos este número (ya que nos encontramos estudiando la escalabilidad), obtenemos que:

$$\lim_{p \to \infty} S(p) = \frac{1}{f}$$

Es decir, la escalabilidad de la aplicación paralela se encuentra limitada ($S(p)$ era creciente) por un factor $1/f$, que depende de la fracción de código que no podemos paralelizar (es decir, por mucho que mejoremos la fracción paralelizable, no obtendremos una mejor escalabilidad si no alteramos f).

3. A continuación, nos encontramos en las mismas condiciones que en el punto anterior, con la salvedad de que ya no consideramos un grado ilimitado de paralelismo, sino que será un cierto valor n. En este caso, para un p concreto obtendremos la misma ganancia que en el caso anterior:

$$S(p) = \frac{1}{f + \dfrac{1-f}{p}}$$

Sin embargo, la escalabilidad del modelo estará limitada por:

$$S(n) = \frac{1}{f + \dfrac{1-f}{n}}$$

Obteniendo un resultado similar al anterior pero en una situación un tanto más realista.

4. Finalmente, consideraremos el caso en el que tenemos una fracción de código secuencial no paralelizable (f), grado de paralelismo ilimitado (para estudiar más fácilmente la escalabilidad) y una sobrecarga no despreciable, que como ya hemos visto, sabemos que aumenta (de forma lineal) conforme lo hace p. En este caso, tendremos una ganancia:

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

Demostración. En primer lugar, calculamos $T_P(p)$:

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

A continuación, sustituyendo en la fórmula de la ganancia:

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

□

Que con un grado de paralelismo grande obtenemos:

$$\lim_{p \to \infty} S(p) = 0$$

Es decir, llega un punto (un grado de paralelismo) a partir del cual si aumentamos el número de procesadores la ganancia deja de crecer y comienza a decrecer. Este decremento se inicia cuando al incrementar p se incrementa el término de la sobrecarga $T_O(p)/T_S$ en mayor medida que se decrementa la parte del cálculo paralelo, de forma que el denominador de la expresión aumenta en lugar de disminuir.

2.3.3. Ley de Amdahl

Como puede deducirse de las ganancias anteriores, estas disminuyen conforme aumenta la fracción de código no paralelizable. Es decir, la mejora en prestaciones depende de la fracción de código no paralelizable. Por lo que la mejora en prestaciones depende de la fracción que no se puede mejorar. Este razonamiento fue formalizado por Gene Amdahl, dando origen a la fórmula:

$$S(p) \leqslant \frac{p}{1 + f(p-1)} \tag{2.2}$$

conocida como la **Ley de Amdahl**.

Demostración. Como ya hemos visto, la intuición de la fórmula proviene de suponer que estamos trabajando en un caso con una fracción de código secuencial no paralelizable f y con una sobrecarga despreciable. Tendremos por tanto que $T_P(p) = T_C(p)$, de forma que una parte de $T_P(p)$ está formada por $f \cdot T_S$ al ser código que no puede paralelizarse y tendremos por otra parte una fracción de $1 - f$ código paralelizable con sobrecarga nula, luego con una escalabilidad lineal. Tenemos por tanto que:

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

Recordemos que habíamos supuesto una sobrecarga nula. Considerando ahora que tenemos una sobrecarga, obtendremos una ganancia en velocidad menor (al ser mayor el denominador de su expresión), luego la ganancia total a la que podemos llegar es la fracción de la Ley de Amdahl, concluyendo así que:

$$S(p) \leqslant \frac{p}{1 + f(p-1)}$$

□

La Ley de Amdahl proporciona una visión pesimista de las ventajas de la paralelización, ya que limita la escalabilidad de la aplicación o arquitectura en un factor $\frac{1}{f}$, independientemente del grado de paralelismo y del código que sí puede paralelizarse.

Cabe destacar que para conseguir la Ley de Amdahl, hemos supuesto constante el tiempo de ejecución de la aplicación en un sistema uniprocesador, junto con la fracción de código no paralelizable. Sin embargo, en muchas aplicaciones se puede incrementar la fracción de código paralelizable aumentando el tamaño del problema que resuelve la aplicación.

2.3.4. Ganancia escalable. Ley de Gustafson

Los objetivos a la hora de paralelizar una aplicación pueden ser:

- Dismiuir el tiempo de ejecución hasta uno razonable.
- Aumentar el tamaño del problema a resolver, lo que puede llegar (a veces) a incrementar la precisión en el resultado.

Por tanto, cuando consigamos el primer objetivo, podemos plantearnos como meta el segundo para mejorar las prestaciones de la aplicación. Supongamos, por tanto, un modelo de código secuencial con una parte no paralelizable y un número n de tareas a paralelizar que puede incrementarse aumentando el tamaño del problema (no confundir n con el tamaño del problema). Considerando despreciable el tiempo de sobrecarga, podemos mantener constante el tiempo de ejecución paralelo T_P, variando el número de procesadores p y el número n de forma que $n = k \cdot p$, con k constante. En este caso, el tiempo de ejecución secuencial depende de n ($T_S(n)$). Teniendo en cuenta estas condiciones, la ganancia en prestaciones sería de:

$$S(p) = \frac{T_S(n)}{T_P} = \frac{f \cdot T_P + p(1-f) \cdot T_P}{T_P} = p(1-f) + f \tag{2.3}$$

Este valor será consistente si mantenemos constante el tiempo de ejecución en paralelo. En la expresión, el único término variable es p. Según la expresión, tenemos que la ganancia depende linealmente del número de procesadores p, con una pendiente de $1 - f$, que es la fracción de tiempo que supone la ejecución de la parte paralela. Por tanto, cuanto mayor sea $1 - f$, mayor será la escalabilidad (al igual que con la Ley de Amdahl, al ser menor f). Esta expresión (que mide cuanto escala la estabilidad) se conoce como la Ley de Gustafson.

Mientras que la Ley de Amdahl asume que el tiempo de ejecución secuencial (o tamaño del problema) se mantiene constante y muestra que la ganancia está limitada debido al código no paralelizado (tomando límite cuando $p \to \infty$), Gustafson mantiene constante el tiempo de ejecución paralelo y muestra que la ganancia en función de p puede crecer con pendiente constante. En ambos estudios se desprecia el tiempo de sobrecarga.

2.3.5. Eficiencia

La eficiencia permite evaluar qué tan cerca está la solución a una aplicación con código paralelo (carga de trabajo en general) respecto a la solución que idealmente

deberíamos obtener dados los recursos disponibles. Dicho de otra forma, permite evaluar el grado de aprovechamiento de los recursos del sistema. Las prestaciones que se pueden esperar idealmente con p recursos serían las prestaciones que se obtienen con un recurso multiplicado por p:

$$E(p) = \frac{Prestaciones(p)}{p \cdot Prestaciones(1)}$$

Teniendo en cuenta la definición de la ganancia, llegamos a que:

$$E(p) = \frac{S(p)}{p} \tag{2.4}$$

es decir, la eficiencia se obtiene dividiendo la ganancia entre el número de recursos (que es la ganancia ideal). La ganancia máxima es p, por lo que la eficiencia máxima es 1. Por otra parte, la ganancia mínima es 1, por lo que la eficiencia mínima será $1/p$.

3. Arquitecturas TLP

En este capítulo, nos centraremos en arquitecturas que permiten ejecutar de forma paralela o concurrente múltiples flujos de instrucciones (o *threads*) que comparten memoria. Se tratan de arquitecturas con paralelismo a nivel de *thread* (*Thread-Level Parallelism*) con una única instancia del Sistema Operativo. Por tanto, cada vez que mencionemos arquitecturas TLP, nos estamos refiriendo a arquitecturas TLP con una única instancia del Sistema Operativo. En este contexto, el SO es el encargado de gestionar los flujos de instrucciones.

Los paradigmas de programación paralela por variables compartidas son los más fáciles de implementar en este tipo de arquitecturas, mientras que las orientadas a paso de mensajes están más relacionadas con arquitecturas TLP con múltiples instancias del SO.

La compartición de memoria que se da trae conceptos como la coherencia del sistema de memoria, consistencia del sistema de memoria o la sincronización entre flujos; conceptos que estudiaremos a lo largo de este capítulo.

3.1. Tipos de Arquitecturas

3.1.1. Objetivos

Esta sección está orientada a:

- Distinguir entre cores multhread, multicores y multiprocesadores.
- Comparar entre cores multithread de grano fino, grueso y cores con multithread simultáneo.

3.1.2. Clasificaciones de arquitecturas TLP

Las arquitecturas TLP con una instancia del SO pueden clasificarse en:

Multiprocesadores. Son capaces de ejecutar en paralelo varios flujos de instrucciones (hilos) en un computador con varios núcleos o procesadores de forma que cada flujo se ejecuta en un núcleo o procesador distinto.

Podemos encontrarnos multiprocesadores en un chip (como los multinúcleos), en una placa o en uno o varios armarios.

Multinúcleos (*multicores*). Pueden ejecutar en paralelo varios flujos de instrucciones en un chip de procesamiento con múltiples núcleos de forma que cada flujo se ejecuta en un núcleo distinto. Un chip multinúcleo no es más que un multiprocesador en un chip.

La denominación de *multicores* proviene de un nombre comercial que dio Intel a sus multiprocesadores en un chip. Además, denominó *procesador* a los chips o encapsulados de procesamiento y *núcleos* (o *cores*) a los procesadores.

Núcleos (o cores) *multithread*. Se trata de un núcleo de procesamiento (un procesador) en el que se ha modificado su arquitectura ILP (*Instruction Level Parallelism*) para poder ejecutar flujos de instrucciones de forma concurrente o en paralelo.

3.1.3. Repaso de arquitecturas ILP

Recordamos lo que eran las arquitecturas ILP (*Instruction Level Parallelism*): son la capacidad por parte de un procesador de ejecutar múltiples instrucciones en paralelo para mejorar el rendimiento del sistema. Las arquitecturas se centran en identificar y aprovechar las dependencias de datos entre las instrucciones para ejecutarlas de forma eficiente.

Etapas de ejecución

Antes de continuar, recordamos las etapas básicas para la ejecución de una instrucción:

Etapa de captación de instrucciones (*Instruction Fetch*).
Etapa en la que se capta de la caché de instrucciones la siguiente instrucción a ejecutar, incrementando el valor del PC (*Program Counter*).

Etapa de decodificación de instrucciones (*Instruction Decode*).
Etapa en la que se decofica la instrucción captada para determinar su tipo y operaciones a realizar. Se identifican aquí las dependencias de datos y condiciones de control que pueden afectar la ejecución de la instrucicón.

Etapa de ejecución (*Execution*).
Se lleva a cabo la ejecución de la instrucción. Podemos encontrarnos 4 tipos de instrucciónes (en relación a ellos se ejecutará una cosa u otra).

- Operaciones de enteros.
- Operaciones de coma flotante.
- Saltos.
- Escrituras o lecturas de memoria.

Aunque esta última no se realizaría en esta etapa, es importante para el desarrollo que vamos a hacer de arquitecturas ILP.

Etapa de acceso a memoria (*Memmory*).
Se accede a memoria en caso de que sea necesario (depende de la instrucción).

Etapa de almacenamiento de resultados en registros (*Write-Back*).
Se almacenan en los registros del procesador los resultados de la instrucción, si es necesario.

Formas de arquitecturas ILP

Podemos encontrarnos con dos formas principales de paralelizar estas etapas a la hora de desarrollar una estructura ILP:

Escalar segmentada.
Segmentaremos las etapas de ejecución de forma que dispongamos de 5 módulos que sean capaces de procesar su parte de forma paralela. Incluiremos *buffers* auxiliares entre las distintas etapas.

VLIW y superescalar.
Consideraremos simplemente 4 etapas (fusionaremos las de ejecución y memoria en una), de forma que replicaremos los componentes de la unidad funcional para que esta sea capaz de ejecutar al mismo tiempo diversos tipos de instrucciones. También es necesario el uso de *buffers* auxiliares.

Por ejemplo, podemos replicar los componentes de la unidad funcional de forma que podamos ejecutar en paralelo (al mismo tiempo):

- Operaciones con enteros.
- Operaciones de coma flotante.
- Operaciones de lecturas y escrituras en memoria.
- Saltos.

Podemos además tener no sólo una unidad sino varias de las ya mencionadas (2 unidades para operaciones con enteros, ...).

Además, las 3 etapas restantes estarán segmentadas. Podemos tener también unidades superescalares en las que tengamos replicadas además las etapas de captación, decodificación y *write-back*.

3.1.4. Clasificaciones de cores multithread

Ahora vamos a clasificar los *cores multithread*, que no dejan de ser procesadores con arquitectura ILP que se aprovechan para ejecutar a la vez distintos hilos del sistema operativo de forma concurrente o paralela.

***Temporal Multithreading* (TMT)**
Ejecutan distintos hilos de forma concurrente en el mismo core. De esta forma, emite instrucciones de un único hilo en cada ciclo. Podemos pensar que el core se está multiplexando.

La conmutación entre hilos la decide el hardware.

***Simultaneous MultiThreading* (SMT)**
Ejecuta distintos hilos de forma paralela en el mismo core. Pueden llegar a

emitir en un sólo ciclo instrucciones de varios hilos. Para llevar esto a cabo, necesitamos un core superescalar.

No implementa conmutación entre hilos, al no ser necesaria.

Según qué tan seguido intercambiamos los hilos del core, nos encontramos con:

TMT de grano fino (*Fine-grain multithreading*, FGMT).
La conmutación de hilos en el core se realiza en cada ciclo. Presentan un coste de cambio de contexto bajo, no es necesario perder ningún ciclo para realizar los cambios de contexto.

La planificación del siguiente hilo a ejecutar puede ser *round-robin* u otra técnica de planificación (podemos guiarnos por el hilo menos recientemente ejecutado, por accesos a datos, por saltos no predecibles, por operaciones con gran latencia, ...).

TMT de grano grueso (*Coarse-grain multithreading*, CGMT).
La conmutación entre hilos no se realiza en cada ciclo. Presentan un mayor coste por cambios de contexto: pueden perderse entre ninguno y varios ciclos debido a los cambios de contexto.

La planificación puede depender cualquier técnica, como tras intervalos de tiempos prefijados (*timeslice multithreading*), por eventos de cierta latencia (*switch-on-event multithreading*), ...

Clasificación de cores con CGMT con conmutación por eventos

Podemos conmutar los hilos del core de grano grueso de forma:

Estática.
Realizando la conmutación de forma *explícita*, mediante nuevas instrucciones para conmutación añadidas al repertorio; o de forma *implícita*, al detectar instrucciones de carga, almacenamiento, salto, ...

- Como ventaja, destacamos el bajo coste de los cambios de contexto (de 0 o 1 ciclos).
- Como inconveniente, pueden producirse cambios de contexto innecesarios.

Dinámica
La conmutación se realiza típicamente por fallos de caché o por interrupciones (conmutación por señales).

- Como ventaja, reduce los cambios de contexto innecesarios de la estática.
- Como inconveniente, la sobrecarga que se añade por los cambios de contexto es mayor.

Hardware	CGMT	FGMT
Registros	Replicado (al menos el PC)	Replicado
Almacenamiento	Multiplexado	Cualquiera de las 4
Hardware de etapas del cauce	Multiplexado	Captación repartida o compartida, el resto multiplexadas
Necesidad de distinguir el hilo de una instruc.	Sí	Sí
Hardware para conmutar	Sí	Sí
Hardware	SMT	CMP
Registros	Replicado	Replicado
Almacenamiento	Todo menos multiplexado	Replicado
Hardware de etapas del cauce	Unidad funcional compartida, el resto repartidas o compartidas	Replicado
Necesidad de distinguir el hilo de una instruc.	Sí	No
Hardware para conmutar	No	No

Tabla 3.1: Comparación de tipos de multithreading.

Característica

En un núcleo multithread, podemos usar las siguiente modificaciones con el objetivo de ejecutar varios hilos en un mismo core:

- Multiplexado: Hacemos que los hilos se turnen en el uso de una unidad.
- Repartición: Repartimos una unidad entre (al menos) dos hilos, de forma que a uno le asociamos su zona y al otro la suya.
- Compartición: Hay (al menos) dos hilos que acceden a la misma unidad de forma simultánea.
- Replicación: Disponemos de varias unidades, de forma que cada hilo puede hacer uso de una.

Notemos que tanto el precio de implementación como la bondad de la técnica se encuentran en orden creciente (siendo más caro y mejor realizar la replicación de unidades).

Una vez desarrollada la clasificación de cores multithread, podemos mostrar la Tabla 3.1 a modo de resumen, que nos ayudará a entender mejor cada tipo de multithreading. "CMP" es un Chip multicore, aquel en el que tenemos replicado todo el cauce (esto es, todas las unidades de la etapa de ejecución).

3.1.5. Comparativa de cores multithread

ESTOS APUNTES ESTÁN AÚN EN DESARROLLO.

3.2. Coherencia del sistema de memoria

A la hora de usar multiprocesadores, surge un problema fundamental de manera natural: las incoherencias en memoria. A lo largo de esta sección definiremos este problema, dando ejemplo y protocolos que lo resuelven. Cabe mencionar que todos los multiprocesadores salvo los NUMA implementan la coherencia del sistema de memoria por hardware.

3.2.1. Objetivos

Tras esta sección, debería ser capaz de:

- Comparar los métodos de actualización de memoria principal implementados en caché.
- Comparar las alternativas para propagar una escritura en protocolos de coherencia de caché.
- Explicar qué debe garantizar el sistema de memoria para evitar problemas por incoherencias.
- Descibir las partes en las que se puede dividir el análisis o el diseño de protocolos de coherencia.
- Distinguir entre protocolos basados en directorios y protocolos de espionaje (*snoopy*).
- Explicar el protocolo de mantenimiento de coherencia de espionaje MSI.
- Explicar el protocolo de mantenimiento de coherencia de espionaje MESI.
- Explicar el protocolo de mantenimiento de coherencia MSI basado en directorios con difusión y sin difusión.

3.2.2. Definición del problema

La utilización de memoria caché trae consigo el esquema de jerarquía de memoria y la posibilidad de tener en distintas jerarquías una misma posición de memoria repetida. En sistemas uniprocesador, si tratamos de modificar una posición de memoria, esta se llevará a caché y será modificada en caché, indicando que ha sido modificada. En un cierto momento, la modificación en esta jerarquía de caché será comunicada a jerarquías de memoria mayores, hasta llegar a memoria principal y modificar dicho dato. De esta forma, la próxima vez que se necesite en caché dicha posición, estará actualizada conforme a la última modificación.

Recordamos ahora que cada procesador lleva asociada una memoria caché y, al ser mayor la memoria principal que la caché, necesitamos almacenar en algún sitio qué bloque de memoria principal contiene cada entrada de memoria caché. Esta información se almacena en una tabla asociada a la caché. En dicha tabla hay una entrada por cada dirección de la memoria caché y, en cada entrada, se almacena

0	Dirección de MP	Bits
1	Dirección de MP	Bits
	⋮	⋮
$N-1$	Dirección de MP	Bits

Tabla 3.2: Tabla para una caché de N direcciones de memoria.

qué bloque de memoria principal está cargado y unos bits de información sobre el bloque cargado. Representamos una aproximación de esta en la Tabla 3.2. En un sistema uniprocesador, nos basta con un bit sucio (que indique si el bloque ha sido modificado o no, para saber si hay que escribir en jerarquías superiores) y un bit de validez (que indique si el bloques es válido).

En un multiprocesador, cada procesador lleva consigo una memoria caché, de forma que todos los procesadores comparten el espacio de memoria. Puede suceder que dos procesadores distintos trabajen con la misma posición de memoria k (y por tanto, tengan al bloque que contiene a k en sus respectivas cachés). Si uno de los dos procesadores decide modificar k, se modificará la dirección de memoria caché correspondiente, pero el bloque en la caché del otro procesador y en memoria principal quedarán inalterados. Nos acabamos de encontrar con una incoherencia en el sistema de memoria.

Concretando las ideas, una **incoherencia en el sistema de memoria** se produce cuando en el sistema de memoria las copias de una misma dirección no tienen el mismo contenido. Como hemos ya comentado, en sistemas uniprocesdores teníamos este problema: podíamos tener un bloque en memoria caché modificado que no estuviese modificado en memoria principal. En este caso hablamos de incoherencias en distintas jerarquías de memoria. Sin embargo, ahora en multiprocesadores podemos tener incoherencias en la misma jerarquía de memoria, tal y como mencionábamos en el ejemplo anterior.

Métodos de actualización de memoria principal

Las situaciones de incoherencia se deben abordar no permitiendo que se produzcan nunca o bien evitando que causen problemas (que algún componente lea el valor no actualizado de la memoria) en caso de permitirse. Las cachés implementan dos métodos de actualización de memoria principal:

Escritura directa (*write-through*).
Con escritura directa, no se permiten situaciones de incoherencia entre caché y memoria principal al escribir en caché: cada vez que se escribe en un bloque de la caché, el correspondiente bloque de la memoria principal es modificado.

Por tanto, tenemos una escritura en memoria principal por cada escritura, lo que requiere usar la red de comunicación entre procesador y memoria principal. Este sobreuso de la red empeora más aún en multiprocesadores, al tener múltiples procesadores que son susceptibles de modificar datos de forma paralela.

Otro problema que se plantea es desaprovechar los principios de localidad espacial y temporal (al modificar una variable, es posible que se modifique otra próxima a ella, o que la ya modificada se vuelva a modificar próximamente), por lo que sería mejor esperar a que termine la modificación en curso (por ejemplo, si estamos iterando sobre un vector y modificando sus componentes), antes de escribir en memoria los cambios modificados.

Cabe destacar también que sí pueden producirse con escritura directa situaciones de incoherencia entre cachés (dos cachés tienen el mismo bloque y una lo modifica) y entre caché y memoria principal (cuando un componente modifica la dirección de un bloque de memoria principal que se encuentra en alguna caché del multiprocesador). En sistemas uniprocesadores, tenemos la ventaja de no necesitar un bit sucio.

Posescritura (*write-back*).
En posescritura, cuando una dato es modificado por un procesador, sólo se modifica en la caché correspondiente a dicho procesador. La actualización en memoria principal se produce cuando un bloque modificado (un bloque en el que se ha alterado una dirección de memoria mientras estaba en la caché) es retirado de la caché (por ejemplo, para hacer sitio a otro bloque más necesario). De esta forma, minimizamos el número de accesos a memoria y, por tanto, de uso de la red de conexión entre el procesador y la memoria; aprovechando los principios de localidad espacial y temporal.

De esta forma, se permiten situaciones de incoherencia entre caché y memoria principal (incluso en sistemas uniprocesador). También puede producirse cualquier tipo de incoherencia en este sistema, empeorando la situación que teníamos con escritura directa.

Es necesario además mantener la información sobre qué bloques han sido modificados en caché y cuáles no. Esta labor la realiza un bit en la tabla de la caché, llamdo "bit sucio".

Lo usual en cachés es que usen posescritura, debido a sus ventajas frente a escritura directa. A continuación, estaremos hablando siempre de sistemas multiprocesadores, ya que en sistemas uniprocesadores las situaciones de incoherencia ya están resueltas (tanto con escritura directa no permitiendo incoherencias tanto con posescritura, permitiendo incoherencias pero controlando que no provoquen fallos).

3.2.3. Protocolos de coherencia entre cachés

Como ya habrás podido deducir, nuestra tarea ahora es buscar cómo resolver las incoherenias ya mencionadas, y manejar las incoherencias que se permiten para que no provoquen fallos en el sistema. Para evitar situaciones de incoherencias entre cachés, se deben cumplir las siguientes dos condiciones:

1. **Propagación de escrituras.** Se debe garantizar que todo lo que se escribe en la copia de un bloque en caché se propague a las copias del bloque en otras cachés.

2. **Serialización de escrituras.** Se debe garantizar que las escrituras en una dirección se ven en el mismo orden por todos los procesadores: el sistema de memoria debe parecer que realiza en serie las operaciones de escritura en la misma dirección. Debe dar la impresión de que estas operaciones sean atómicas.

   Esto es de vital importancia en arquitecturas donde no todos los procesadores tienen el mismo (o similar) tiempo acceso a sus cachés, como los NUMA, debemos garantizar que el primer procesador que se dispuso a escribir en memoria sea el primero que lo haga.

Los **protocolos de coherencias de caché** buscan resolver el problema de las incoherencias entre cachés, de forma que cada escritura en caché sea visible para el resto de procesadores, propagando de forma fiable el valor escrito en una dirección. Los protocolos de coherencia usan dos alternativas principales para propagar escrituras a otras cachés:

Escritura con actualización (*write-update*).
Siempre que se modifique una dirección en la copia de un bloque en una caché, se modifica dicha dirección en las copias del mismo bloque de memoria que tengan el resto de procesadores en sus cachés (en caso de tenerlo). Si se cumplen los principios de localidad espacial y temporal, esto provoca una alta sobrecarga en la red de comunicación, al tener que avisar al resto de procesadores la modificación de una variable.

Escritura con invalidación (*write-invalidate*).
Antes de que un procesador modifique un bloque en su memoria caché, invalida el resto de copias del mismo bloque en las cachés de otros procesadores. Posteriormente, es libre de modificar su bloque tantas veces como desee, obteniendo un *acceso exclusivo* al bloque.

Cuando otro procesador quiera acceder a dicho bloque desde su caché, en caso de tenerlo, verá que estará invalidado y deberá solicitarlo a la memoria (en caso de que el bloque se encuentre actualizado en memoria) o al procesador que tenía el acceso exclusivo al bloque. Es necesario por tanto disponer de un bit en la tabla de la caché que indique si un bloque se encuentra invalidado o no.

Notemos que invalidar es más rápido que actualizar, ya que sólo necesitamos recibir la información del bloque invalidado y cambiar un bit, en lugar de reescribir el bloque completo.

Esta práctica sólo permite compartir bloques de memoria mientras sólo se lee del bloque.

Si escribimos varias veces sucesivas sin que otro procesador lea de su bloque correspondiente, con invalidación podemos reducir el número de accesos (transferencias) a la red. Por otra parte, si deseamos que un procesador escriba un dato para que el resto lo lean podría ser más eficiente usar escritura con actualización, que disminuye en este caso los accesos a la red (notemos que con invalidación necesitamos un acceso a la red por cada caché que falle, pero con actualización con un único acceso a la red podemos actualizar todas las cachés). Cabe destacar que esta ventaja de la actualización no se da en arquitecturas que no implementan difusión. Podemos decir

que en general, la política de actualización genera un tráfico innecesario cuando los datos compartidos se leen por pocos procesadores.

La propagación de las actualizaciones o invalidaciones entre los procesadores se puede realizar:

- Con una **difusión** de los paquetes de actualización o invalidación a todas las cachés.
- Con el envío de los paquetes de actualización o invalidación sólo a aquellas cachés con copias del bloque, que son sólo las que necesitan recibirlo. Es decir, realizar un **envío selectivo**.

  Para esta última alternativa, necesitamos mantener una tabla o directorio de memoria, que informe de las cachés que tienen copia de un determinado bloque para poder realizar la comunicación. Esta tabla tendría una entrada por cada dirección de memoria de cada caché y contendría el bloque que contiene, junto con bits de estado.

3.2.4. Protocolos de mantenimiento de coherencia

En una arquitectura UMA, podemos utilizar una red bus para las transferencias entre los procesadores y la memoria que comparten. Los buses implementan la difusión de forma natural. De esta forma, todos los paquetes enviados a la red son visibles por todos los componentes conectados al bus, de forma que todos ven las peticiones en el orden en el que se solicitan (dándose las dos condiciones para evitar situaciones de incoherencias). Todo esto hace que, en el caso de los multiprocesadores UMA con red bus, no sea necesario mantener información de la caché con copias de los bloques, pudiendo suprimir incluso el directorio de memoria del que antes se hablaba.

En este tipo de arquitecturas, es común el uso de **protocolos de espionaje** (*snoopy*), ya que todos los componentes pueden ver (espiar) dicho bus. Cada controlador de caché espía los paquetes del bus y actúa en consecuencia (si por ejemplo otro controlador solicita un paquete invalidado del que nuestra caché tiene acceso exclusivo y no está actualizado en memoria, nuestro controlador invalida la respuesta de la memoria y es él quien responde a la petición del paquete, devolviendo el paquete actualizado). Es por ello sencillo implementar protocolos de coherencia en una arquitectura que use buses. Sin embargo, los protocolos de espionaje no escalan bien debido a retardos que introducen otro tipo de redes[1].

En redes en las que la difusión es costosa de implementar o redes que requieren de una gran escalabilidad, se usan **esquemas basados en directorios**, en los que para reducir el tráfico, se envían los paquetes únicamente a las cachés implicadas (cachés con copia del bloque al que se accede). Es necesario por tanto el uso del directorio de memoria. Sin embargo, también obtenemos un cuello de botella al tener que acceder por cada procesador a dicho directorio. Se obtienen mejores prestaciones distribuyendo el directorio entre los módulos de memoria principal, de forma que el

[1] Al tener que esperar a que todos los procesadores reciban el paquete enviado.

subdirectorio de cada módulo mantenga la información sobre sólo los bloques que contiene cada módulo. De esta forma, los diferentes subdirectorios pueden procesar peticiones en paralelo.

Cabe destacar que no son los únicos tipos de protocolos de coherencia, sino que hay también esquemas organizados en **jerarquía**, compuestos de protocolos de espionaje y directorios, que dependen de la red utilizada en cada nivel.

Resumiento toda esta introducción a los protocolos de coherencia realizada, debemos tener claro que para diseñar un protocolo de mantenimiento de coherencia, debemos planificar:

- La política de actualización en memoria principal: escritura directa o posescritura.
- La política de propagación de escrituras entre cachés: escrituras con actualización o con invalidación.
- El comportamiento:
  - Los posibles estados de un bloque en caché y las acciones que el controlador de caché debe realizar ante eventos recibidos.
  - Los posibles estados de un bloque en memoria principal, junto con las acciones que el controlador de memoria principal debe hacer ante eventos.
  - Los paquetes que genera el controlador de memoria principal.
  - Saber relacionar las acciones con eventos y estados.

A continuación, vamos a estudiar cuatro protocolos de mantenimiento de coherencia, dos para multiprocesadores UMA con red bus (MSI y MESI) y dos para multiprocesadores NUMA en una placa (MSI con y sin difusión). Todos ellos usan posescritura y escrituras con invalidación, como cabría esperar.

Los protocolos de espionaje que estudiaremos se basan en la difusión de los paquetes asociados al mantenimiento de coherencia a todas las cachés. Se implementan de forma eficiente en sistemas basados en buses.

3.2.5. Protocolo MSI (*Modified-Shared-Invalid*) de espionaje

Describimos a continuación el protocolo MSI, el protocolo con menor número de estados que utiliza posescritura e invalidación. Debe su nombre a cada estado de un bloque en caché.

Estados de un bloque en caché

Modificado (M).
Un bloque modificado en caché es la única copia del bloque válida en todo el sistema de memoria.

- En caso de que otro módulo solicite este bloque por el bus, el controlador de su caché debe invalidar la respuesta de la memoria principal (la memoria principal siempre intentará responder)[2] y el controlador de caché debe responder con su propio bloque, que es el único válido.
- El controlador de su caché debe escribir dicho bloque en memoria en caso de ser extraído de su caché.
- El controlador de su caché debe invalidar el bloque si le llega por el bus una petición de invalidación del bloque correspondiente.

Compartido (S).
Un bloque compartido en caché indica que todas las copias de dicho bloque que pueda haber en la jerarquía de memoria están actualizadas.

- El controlador de su caché debe invalidar el bloque si le llega por el bus una petición de invalidación del bloque correspondiente.

Inválido (I).
Un bloque inválido en caché es un bloque que o no está físicamente en caché (esa posición de caché está vacía) o que está invalidado (por progragación de invalidaciones desde otra caché).

- Si el procesador de la caché accede a un bloque en estado inválido, enviará un paquete de petición del bloque por el bus de la red.

Podemos ordenar estos estados en relación al grado de disponibilidad del bloque por parte del procesador al que pertenece la caché en orden creciente por:

- Inválido.
- Compartido.
- Modificado.

El estado modificado supone que el procesador tiene el uso exclusivo del bloque, de forma que puede disponer de él tanto par aleer como para escribir, sin informar al resto del sistema. Como el módulo que contiene al bloque es el único que dispone de una copia válida del mismo, se trata del encargado de responder a todas las peticiones de este por el bus.

Todos estos estados se implementan añadiendo a la tabla de caché de cada módulo (Tabla 3.2) una serie de bits. Necesitamos representar 3 estados, luego nos será suficiente con disponer de mínimo 2 bits por cada entrada de la tabla. Las implementaciones suelen añadir normalmente los bits de invalidez (indica si el bloque en dicha posición de caché es válido) y sucio (indica si el bloque en dicha posición de caché ha sido modificado) en relación a nuestro modelo de estados MSI. De esta forma:

[2]En caso de estar así implementado; puede haber implementaciones que no realicen esta función, como se verá en breves.

Estados	Bit de invalidez	Bit sucio	Ambos bits
M	0	1	01
S	0	0	00
I	1	0	10

Estados de un bloque en memoria

Según lo anteriormente explicado, podemos evitar establecer estados para bloques en memoria (habilitando que los controladores de caché puedan inhibir respuestas del controlador de memoria principal). Sin embargo, si se desea almacenar puede hacerse simplemente con dos estados:

Válido. El bloque está actualizado en memoria principal y puede haber una copia en una o varias cachés (sin acceso exclusivo).

- El controlador de memoria principal responde con el bloque si ve en el bus una petición con lectura de un bloque que en memoria principal es válido.

Inválido. El bloque no está actualiza en la memoria principal y hay una copia válida en alguna caché.

- En caso de que este bloque sea pedido por el bus, será el controlador de caché de la caché correspondiente quien proporcione el bloque.

En caso de desear implementar la validez o no de los bloques en memoria, nos bastaría con disponer de un bit de invalidez por bloque de memoria.

Paquetes que genera un controlador de caché

Hemos comentado varias veces que los distintos controladores de caché de cada nodo tienen que tener un trabajo en la red de comunicación, pero no hemos especificado cómo se realizan estas comunicaciones.

Un controlador de caché puede generar los siguiente tipos de transferencias como consecuencia de acciones en caché del procesador, o como consecuencia de paquetes recibidos por otros nodos. Puede generar paquetes de petición (comienzan por `Pt`) o de respuesta (comienzan por `Rp`).

Petición de lectura de un bloque `PtLec(B)`.
Se genera como consecuencia de una lectura con fallo de caché del procesador del nodo (evento `PrLec`) en relación al bloque `B`. El paquete contendrá la dirección del bloque `B`.

Como respuesta a esta petición, recibirá un paquete de respuesta con el bloque (`RpBloque(B)`) de memoria principal o de la caché que lo tiene en estado modificado, en caso de haberlo.

Petición de acceso exclusivo a un bloque con lectura de bloque `PtLecEx(B)`.
Se genera como consecuencia de una escritura con fallo de caché del procesador

(evento `PrEsc`) en un bloque inválido `B` en la caché. El paquete contendrá la dirección del bloque `B`.

Como respuesta se invalidarán las copias del bloque en otras cachés y memoria principal; y se recibirá un paquete de respuesta con el bloque (`RpBloque(B)`) de memoria o de la caché que lo tiene en estado modificado, en caso de haberlo.

Petición de acceso exclusivo a un bloque `PtEx(B)`.
Se genera como consecuencia de una escritura del procesador (evento `PrEsc`) en un bloque `B` en estado compartido en la caché (si estuviera en estado modificado, no hace falta generar este paquete). El paquete contendrá la dirección del bloque `B`.

Como respuesta se invalidarán las copias del bloque en otras cachés y memoria principal.

En ciertas implementaciones no existe este paquete; en su lugar se usa el paquete `PtLecEx(B)` descartando el paquete de respuesta (`RpBloque(B)`) que es enviado a través del bus.

Petición de posescritura de un bloque `PtPEsc(B)`.
Se genera por el reeemplazo en caché de un bloque `B` en estado modificado, como consecuencia del acceso del procesador a un bloque que no se encuentra en caché. El paquete contendrá la dirección del bloque `B`, así como su contenido.

El procesador no espera ninguna respuesta.

Respuesta con bloque `RpBloque(B)`.
Se genera por el controlador de la caché que tiene el bloque `B` solicitado en estado modificado, cuando detecta por el bus una petición `PtLec(B)` o `PtLecEx(B)`.

Paquetes que genera el controlador de memoria principal

El controlador de memoria principal sólo genera paquetes de respuesta:

Respuesta con bloque `RpBloque(B)`.
Se genera como respuesta a paquetes de peticiones de un bloque `B` por `PtLec(B)` o `PtLecEx(B)`.

- Si la memoria guarda el estado de sus bloques de memoria (esto es, los controladores de caché no inhiben la respuesta de la memoria principal), espiará el bus y sólo generará el paquete de respuesta si el bloque `B` se encuentra en estado válido en memoria.
- Si la memoria no guarda el estado de sus bloques de memoria, siempre reponderá con estos paquetes, siendo su respuesta inhibida por el controlador correspondiente.

Estado	Evento	Acciones del controlador	Estado final
	`PrLec(B)`		M
	`PrEsc(B)`		M
M	`PtLec(B)`	Genera `RpBloque(B)`	S
	`PtLecEx(B)`	Invalida copia local y genera `RpBloque(B)`	I
	`Reemplazo(B)`	Genera `PtPEsc(B)`	I
	`PrLec(B)`		S
S	`PrEsc(B)`	Genera `PtEx(B)`	M
	`PtLec(B)`		S
	`PtLecEx(B)`	Invalida copia local de `B`	I
	`PrLec(B)`	Genera `PtLec(B)` y recibe `RpBloque(B)`	C
I	`PrEsc(B)`	Genera `PtLecEx(B)` invalida en otras cachés y recibe `RpBloque(B)`	M
	`PtLec(B)`		I
	`PtLecEx(B)`		I

Tabla 3.3: Tabla de acciones del controlador de caché.

Transiciones de estado de un bloque

En la Tabla 3.3 podemos ver las acciones del controlador de caché de un nodo (los paquetes que genera y los cambios de estado que realiza sobre sus bloques) que provocan los eventos relacionados con un bloque `B`, teniendo en cuenta el estado del bloque en la caché (eventos del procesador o del controlador de caché del nodo, recibidos del exterior a traés de la red en forma de paquete).

Fallo de lectura.

El procesaor lee (`PrLec(B)`) y el bloque no está en caché (o está en estado inválido). El controlador de caché difunde un paquete de petición de lectura del bloque de memoria `PtLec(B)`.

El estado del bloque en la caché será compartido. La copia del boque en otras cachés también será compartido y en memoria, válido.

`PtLec(B)` provoca los siguientes efectos:

1. Si el bloque se encuentra en otra caché en estado modificado, su controlador deposita en el bus el paquete `RpBloque(B)` y pasa a estado compartido. La memoria también recoge el bloque del bus, pasando a estado válido (si tiene estados).
2. Si el bloque está compartido en otras cachés, la memoria proporciona el bloque a la caché que la solicita. El bloque sigue en estado compartido en cada una de las cachés.

Fallo de escritura al no estar el bloque en la caché.
El procesador escribe (`PrEsc(B)`) y el bloque no está en caché (o está en estado inválido). El controlador de caché difunde un paquete de petición de acceso exclusivo al bloque de memoria `PtLecEx(B)`.

El estado del bloque en la caché después de la escritura será modificado.

`PtLecEx(B)` provoca los siguientes efectos:

1. Si la memoria tiene el bloque válido, lo deposita en el bus y pasa a estado inválido.
2. Si una caché tiene el bloque en estado modificado, deposita el bloque en el bus (invalidando la respuesta de la memoria) y pasa a estado inválido.
3. Si una caché tiene el bloque en estado compartido, pasa a estado inválido.

Acierto de escritura en bloque compartido.
El procesador escribe (`PrEsc(B)`) y el bloque está en caché en estado compartido. El controlador quiere acceso exclusivo al bloque, por lo que difunde un paquete de petición de acceso exclusivo al bloque `PtEx(B)`.

El estado del bloque en la caché después de la escritura será modificado[3].

`PtEx(B)` provoca los siguientes efectos:

1. Si la memoria tiene el bloque válido, lo deposita en el bus y pasa a estado inválido.
2. Si una caché tiene el bloque en estado modificado, deposita el bloque en el bus y pasa a estado inválido.
3. Si una caché tiene el bloque en estado compartido, pasa a estado inválido.

Acierto de escritura en bloque modificado.
El procesador escribe (`PrEsc(B)`) y el bloque está en caché en estado modificado. Como el nodo ya tiene acceso exclusivo al bloque, no se genera ningún paquete. El bloque sigue en estado modificado.

Acierto de lectura.
El procesador lee (`PrLec(B)`) y el bloque está en caché en estado compartido o modificado. No se genera ningún paquete y el bloque continúa en su mismo estado.

Reemplazo.
Fallo en el acceso a otro bloque y la política de reemplazo de la caché selecciona a `B` para hacer sitio al nuevo. Si el bloque reemplazado se encuentra en estado modificado, el controlador de caché difunde un paquete de posescritura en memoria `PtPEsc(B)`.

El estado del bloque en la caché pasa a ser inválido.

`PtPEsc(B)` provoca el siguiente efecto:

1. El bloque se transfiere a memoria principal, pasando el estado del bloque en memoria a válido.

[3]Notemos que el acierto de escritura en un bloque compartido se trata de forma similar al fallo de escritura.

3.2.6. Protocolo MESI (*Modified-Exclusive-Shared-Invalid*) de espionaje

El protocolo que estamos a punto de ver también utiliza posescritura como política de actualización de memoria principal y escritura con invalidación como política para mantener la coherencia entre cachés, como cabría esperar.

Con el protocolo MSI, siempre que se escribía en la copia de un bloque en una caché se genera un paquete de petición con acceso exclusivo al bloque, aunque no haya copias en otras cachés que se tengan que invalidar. Esto generaba un tráfico innecesario por la red que degrada las prestaciones de las aplicaciones, especialmente en aplicaciones secuenciales. Es por esto que el protocolo MESI divide el anterior estado compartido en dos (con la filosofía de que si un bloque está en estado compartido en un único nodo no está siendo compartido realmente).

Un bloque en estado *exclusivo* será válido sólo en dicha caché y en memoria principal. Si un bloque está en estado exclusivo en un nodo y el procesador del nodo escribe en el bloque, no se generará paquete para solicitar uso exclusivo del bloque (invalidando copias), ya que ninguna otra caché tiene copias del bloque. Si un bloque está en estado exclusivo, entonces son dos los propietarios del bloque: la memoria principal y una única caché.

Reordenamos ahora el esquema de disponibilidad de bloques según su estado, desarrollando a continuación cada estado en este nuevo protocolo:

- Inválido.
- Compartido.
- Exclusivo.
- Modificado.

Estados de un bloque en caché

Modificado (M).
Si el bloque en la caché se encuentra en estado modificado, significa que ese es el único nodo del sistema con una copia válida del bloque, mientras que el resto de cachés y memoria tienen una copia desactualizada (no válida).

El controlador de su caché debe proporcionar el bloque si observa al espiar el bus que algún componente lo solicita y debe invalidar su propia copia si algún otro nodo solicita una copia exclusiva del bloque para su modificación.

Exclusivo (E).
Si el bloque en caché se encuentra en este estado, significa que es la única caché en el sistema con una copia válida del bloque: el resto de cachés tienen una copia no válida. La memoria tiene también una copia actualizada el bloque.

La caché debe invalidar su copia si observa al espiar el bus que algún nodo solicita una copia exclusiva del bloque para su modificación.

Compartido (C).
Supone que el bloque es válido en esta caché, en memoria y en al menos alguna otra caché.

Estado	Evento	Acciones del controlador	Estado final
	`PrLec(B)`		E
E	`PrEsc(B)`		M
	`PrLec(B)`		S
	`PrLecEx(B)`		I
I	`PrLec(B)` y no hay cachés con copias de `B`	Genera `PtLec(B)` y recibe `RpBloque(B)`	S

Tabla 3.4: Tabla de nuevas acciones del controlador de caché.

La caché debe invalidar su copia si al espiar el bus algún otro nodo solicita una copia exclusiva del bloque para su modificación.

Inválido (I).
Supone que el bloque no está físicamente en lacaché, o si se encuentra, ha sido invalidado por el controlador como consecuencia de la escritura en la copia del bloque situado en otra caché.

En cuanto a los paquetes que usa MESI, son los mismos que MSI.

Transiciones de estado de un bloque

En la Tabla 3.4 podemos ver las nuevas acciones del controlador de caché de un nodo que provocan los eventos relacionados con un bloque `B`, teniendo en cuenta el estado del bloque en la caché que ahora origina el protocolo MESI frente a MSI (es decir, tener en cuenta la Tabla 3.3 y agregarle las acciones de la Tabla 3.4).

Detallamos las acciones que se cometen tras la disparación de cada evento con detalle.

Fallo de lectura.
El procesaor lee (`PrLec(B)`) y el bloque no está en caché (o está en estado inválido). El controlador de caché difunde un paquete de petición de lectura del bloque de memoria `PtLec(B)`.

El estado del bloque en la caché será pasará a ser compartido si hay copias del bloque en otras cachés y exclusivo en caso contrario. Se puede añadir una línea `OR` cableada que informe si hay cachés con copias del bloque solicitado. El estado en memoria será válido.

`PtLec(B)` provoca los siguientes efectos:

1. Si el bloque se encuentra en otra caché en estado modificado, su controlador deposita en el bus el paquete `RpBloque(B)` y pasa a estado compartido. La memoria también recoge el bloque del bus, pasando a estado válido (si tiene estados).
2. Si el bloque está compartido en otras cachés, la memoria proporciona el bloque a la caché que la solicita. El bloque sigue en estado compartido en cada una de las cachés.

3. Si el bloque se encuentra en estado exclusivo, pasa a estado compartido. En este caso, la respuesta con el bloque que llega a la caché del nodo solicitante procede de la memoria principal.

Hay implementaciones en las que si el bloque solicitado está disponible en estado válido en alguna caché, en lugar de proporcionar el bloque la memoria lo proporciona la caché. Es necesario aquí añadir hardware que realice esta funcionalidad (que decida qué caché seleccionar). Esta alternativa es interesante en sistemas con memoria físicamente distribuida (que probablemente use protocolos basados en directorios), ya que se puede proporcionar el bloque desde el nodo más cercano al que lo solicita, reduciendo tiempos.

Fallo de escritura al no estar el bloque en la caché.

El procesador escribe (`PrEsc(B)`) y el bloque no está en caché (o está en estado inválido). El controlador de caché difunde un paquete de petición de acceso exclusivo al bloque de memoria `PtLecEx(B)`.

El estado del bloque en la caché después de la escritura será modificado.

`PtLecEx(B)` provoca los siguientes efectos:

1. Si una caché tiene el bloque en estado modificado, deposita el bloque en el bus (invalidando la respuesta de la memoria) y pasa a estado inválido.
2. Si una caché tiene el bloque en estado exclusivo o compartido, pasa a estado inválido.

Acierto de escritura en bloque compartido.

El procesador escribe (`PrEsc(B)`) y el bloque está en caché en estado compartido (es decir, está en algún otra caché también). El controlador quiere acceso exclusivo al bloque, por lo que difunde un paquete de petición de acceso exclusivo al bloque `PtEx(B)`.

El estado del bloque en la caché después de la escritura será modificado[4].

`PtEx(B)` provoca los siguientes efectos:

1. Si la memoria tiene el bloque válido, lo deposita en el bus y pasa a estado inválido.
2. Si una caché tiene el bloque en estado modificado, deposita el bloque en el bus y pasa a estado inválido.
3. Si una caché tiene el bloque en estado compartido, pasa a estado inválido.

Acierto de escritura en bloque modificado o exclusivo.

El procesador escribe (`PrEsc(B)`) y el bloque está en caché en estado modificado o exclusivo. Como no hay ningún otro nodo con acceso al bloque, no se genera ningún paquete. El bloque pasa a estado modificado si estaba en estado exclusivo y si no, no se modifica su estado.

[4]Notemos que el acierto de escritura en un bloque compartido se trata de forma similar al fallo de escritura.

Acierto de lectura.

El procesador lee (`PrLec(B)`) y el bloque está en caché en estado compartido, exclusivo o modificado. No se genera ningún paquete y el bloque continúa en su mismo estado.

Reemplazo.

Fallo en el acceso a otro bloque y la política de reemplazo de la caché selecciona a `B` para hacer sitio al nuevo. Si el bloque reemplazado se encuentra en estado modificado, el controlador de caché difunde un paquete de posescritura en memoria `PtPEsc(B)`.

El estado del bloque en la caché pasa a ser inválido.

`PtPEsc(B)` provoca el siguiente efecto:

1. El bloque se transfiere a memoria principal, pasando el estado del bloque en memoria a válido.

3.2.7. Protocolos basados en directorios

Los protocolos de espionaje son apropiados para redes que implementan comunicaciones con comunicaciones uno-a-todos, tales como los buses. Sin embargo, para redes en las que las difusiones son costosas o cuando necesitamos una gran escalabilidad en el sistema, es mejor usar protocolos basados en directorios. Por tanto, nos serán útiles tanto en multiprocesadores con memoria físicamente distribuida (los NUMA) como en multiprocesadores con memoria centralizada (los UMA) con red escalable.

Estos protocolos de mantenimiento de coherencia reducen el tráfico en la red enviando selectivamente órdenes sólo a las cachés que disponen de una copia válida del bloque implicado en la operación de memoria. Para que esto sea posible, debe existir (como ya se ha comentado previamente) una tabla de memoria (o directorio de memoria) en la que haya una entrada de directorio asociada a cada bloque de memoria principal, con información de las cachés que tienen copia de dicho bloque; y de toda la información necesaria para tratar las situaciones de incoherencias.

Por ejemplo, en la Tabla 3.5, representamos una tabla con una entrada por cada bloque de memoria y una fila por cada caché en el sistema, junto con sus bits de presencia. La última fila corresponde con el bit de presencia en memoria principal.

Bloque	C_0	C_1	...	C_{N-1}	
0	1	0	...	1	0
1	0	0	...	1	1
⋮	⋮	⋮	⋮	⋮	⋮
$M-1$	1	1	...	1	0

Tabla 3.5: Directorio de memoria, mediante vector de bits.

Según cómo esté distribuido el almacenamiento del directorio en la jerarquía de memoria, podemos clasificar los tipos de directorios de memoria en:

- Directorio centralizado.
- Directorio distribuido entre módulos de memoria principal.
- Directorio distribuido entre módulos de memoria principal y caché.

El primer protocolo basado en directorio que se implementó consistía en un directorio centralizado monolítico, con una entrada para cada uno de los bloques de memoria principal con información de estado e información de las cachés con copias del mismo.

Este directorio tenía que atender todas las peticiones de acceso a memoria generadas por el sistema, lo que introducía un cuello de botella que degradaba las prestaciones del sistema. Buscamos por tanto repartir el directorio en el sistema. Esta repartición puede realizarse tanto por filas como por columnas (o ambas).

En una implementación distribuida entre los módulos de memoria principal, cada módulo tiene asociado un subdirectorio con información para los bloques de memoria de dicho módulo. En este caso, hemos repartido la tabla por filas, de forma que las peticiones podrán ser atendidas por los subdirectorios en paralelo.

En una implementación distribuida entre los módulos de memoria principal y cachés, además de distribuir las filas entre los módulos de memoria principal, se distribuye cada una de las filas entre las cachés con copia del bloque.

Protocolo MSI para multiprocesadores NUMA

A continuación, vamos a describir dos protocolos MSI con prosescritra e invalidación para NUMA basados en directorios: uno que difunde las peticiones a todos los nodos (con difusión) y uno que envía las peticiones sólo al nodo que tiene el bloque en el trozo de memoria principal más cercano al nodo (sin difusión). Al seguir esta filosofía, nos aparecen tres roles que desempeñan los nodos en la red NUMA:

Nodo solicitante de un bloque (S)**.** Es un nodo que genera una petición del bloque (`PtLec`, `PtEx`, `PtLecEx` o `PtPEsc`).

Nodo origen de un bloque (O)**.** Recordamos que en un multiprocesador NUMA la memoria está repartida físicamente entre los nodos, de forma que cada uno tiene en módulos de memoria más cercanos un trozo del espacio de direcciones total del multiprocesador. Decimos que esos módulos de memoria más cercanos se hospedan en el nodo. El nodo origen o *home* de un bloque es aquél que tiene el bloque en el trozo de memoria que hospeda.

Nodo propietario de un bloque (P)**.** Es un nodo que tiene copia del bloque en su caché.

De esta forma, en la situación más compleja (que estos tres nodos sean distintos), tenemos al controlador de caché de un nodo S que solicita un bloque `B` de memoria principal, por no tenerlo en su caché. Dicha petición la procesará el controlador de caccé del nodo O (origen del bloque), ya que la dirección de `B` corresponde a un módulo de memoria alojado en O. Dado que O no tendrá a `B` en estado válido (suponíamos que $O \neq P$), solicitará al nodo P el bloque, que lo tiene en estado válido para así poder devolver a S el bloque solicitado.

Bloque	Estado Memoria	C_0	C_1	...	C_{N-1}
⋮	⋮	⋮	⋮	⋮	⋮
$B-1$	V	0	0	...	0
B	V	1	1	...	1
$B+1$	I	0	0	...	1
⋮	⋮	⋮	⋮	⋮	⋮

Tabla 3.6: Subdirectorio de memoria principal para un nodo de un NUMA.

3.2.8. Protocolo MSI para procesadores NUMA sin difusión

Como en este protocolo no usamos difusión, necesitamos almacenar en el directorio de memoria prncipal, además del estado del bloque en memoria, información sobre las cachés con copia del bloque, para que las invalidaciones se puedan propagar sólo a los nodos con copia del bloque.

Al igual que en la Tabla 3.5, se puede usar un vector de bits de presencia para almacenar dicha información. Habría un bit para cada caché conectada a la red NUMA, de forma que un bit activo (un 1) significa que hay una copia válida del bloque en esa caché. El directorio a usar se encuentra distribuido entre módulos de memoria principal (entre los nodos del computador NUMA). En la Tabla 3.6 podemos observar cómo es uno de estos subdirectorios de un nodo, con una entrada por cada bloque de memoria alojado en el nodo. Este protocolo es el más básico a la hora de usar un directorio con posescritura, invalidación y sin difusión.

Todas las peticiones de un bloque se envían al nodo origen del bloque, que es quien tiene el subdirectorio con información sobre el bloque. El nodo origen propagará las invalidaciones a los nodos con copia de un bloque. Con esta propagación, cumplimos la condición 1 de coherencia. El orden de llegada de las peticiones de un bloque al origen será el orden en el que los nodos van a ver los accesos a memoria de ese bloque. Todos van a ver el mismo orden en las escrituras, cumplíendose la segunda condición de coherencia.

Estados de un bloque en caché

Son los mismos que teníamos ya en el protocolo MSI de la sección 3.2.5, Modificado (M), Compartido (C) e Inválido (I).

Estados de un bloque en memoria

Este protocolo cuenta con dos estados estables que ya tenía el protocolo MSI de espionaje: válido e inválido. Sin embargo, nos encontramos además implementaciones con estados "pendientes" de un estado estable, como pueden ser *pendiente de válido* o *pendiente de inválido*.

Un estado pendiente indica que se está procesando un acceso a memoria del bloque. Hasta que no se pueda procesar una nueva petición sobre dicho bloque, su estado continuará en pendiente. A las peticiones que se reciben de un bloque pendiente se responde con un paquete de reconocimiento negativo al solicitanto, para que intente de nuevo la petición más tarde.

Válido. Puede haber copias válidas en una o en más cachés.

Inválido. Hay una copia válida en una caché en estado Modificado.

Pendiete de válido. Hay una copia válida en una caché, cuyo contenido se trasladará a la memoria próximamente.

Pendiete de inválido. La copia del bloque en la memoria está esperando a invalidarse, en cuanto se invalide en el resto de cachés.

Paquetes generados por controladores e interacciones

A continuación, desarrollamos varias interacciones que pueden suceder entre los tres posibles roles principales de los nodos de un computador NUMA, desarrollando al mismo tiempo todos los paquetes que usan los contorladores de caché de cada nodo:

1. Paquetes de petición (`Pt`) desde S hasta O (cuando son distintos nodos).

   a) Cuando se produce una lectura del procesador del nodo S (`PrLec(B)`) con fallo de caché (no hay copia válida del bloque `B` en la caché privada del nodo), se genera el paquete de lectura de bloque `PtLec(B)`.

   Cuando reciba el paquete de respuesta `RpBloque(B)`, introducirá el bloque en la caché, poniendo su estado en Compartido.

   b) Cuando se produce una escritura del procesador del nodo S (`PrEsc(B)`) en bloque `B` en estado Compartido, se genera el paquete de petición de acceso exclusivo sin lectura `PtEx(B)`.

   Cuando reciba el paquete de respuesta `RpInv(B)` que confirma la invalidación en memoria y en otras cachés, modificará el bloque y cambiará el estado de `B` a Modificado.

   c) Cuando se produce una escritura del procesador del nodo S (`PrEsc(B)`) en el bloque `B` en estado Inválido, se genera el paquete de petición de lectura con acceso exclusivo `PtLecEx(B)`.

   Cuando reciba el paquete de respuesta `RpBloqueInv(B)` que confirma la invalidación en memoria y en otras cachés, junto con el propio bloque `B`, lo modificará y cambiará el estado de `B` a Modificado.

   d) Cuando el controlador de caché del nodo S reemplaza un bloque `B` que se encuentra en estado Modificado, se genera el paquete de petición de posescritura `PtPEsc(B)`.

   Posteriormente, el bloque dejará de estar físicamente en la caché.

2. Paquetes de reenvío (`Rv`), de petición desde O a nodos con copia P; y paquetes de respuesta (`Rp`), desde O al nodo S.

   a) Cuando O recibe el paquete `PtLec(B)` y el bloque `B` está en estado Inválido en memoria principal, entonces algún nodo dispone del bloque válido en su caché. Si es el nodo O, no se crea paquete de reenvío y se devuelve `RpBloque(B)`; por otra parte, si no es el nodo O, se envía el paquete de

reenvío de lectura del bloque `RvLec(B)` a algún nodo que tenga a `B` válido en su caché.

El nodo O pone el bloque en el directorio en estado Pendiente de válido hasta que se actualice el bloque en memoria una vez recibido de la única caché con copia válida. Entonces, pasará a estado Válido y podrá responder a S.

b) Cuando O recibe el paquete `PtEx(B)` de S y el bloque `B` está válido en cachés de otros nodos ó recibe el paquete `PtLecEx(B)` de S y el bloque `B` está válido tanto en cachés de otros nodos como en memoria, se envía el paquete de invalidación `RvInc(B)` a los nodos que tengan válido el bloque `B`.

El nodo O pondrá el estado de `B` en su directorio a Pendiente de inválido (para que el resto de llamadas no sean atendidas). Cuando O confirma la invalidación en el resto de cachés:

- Dejará activo sólo el bit de presencia de S en el directorio.
- Pondrá el estado de `B` en memoria a Inválido.
- Responderá a S confirmando su invalidación en memoria y en otras cachés con `RpInv(B)` (en caso de `PtLecEx(B)`, incluirá el bloque en el paquete y será de tipo `RpBloqueInv(B)`).

c) Cuando O recibe el paquete `PtLecEx(B)` de S y el bloque `B` está Inválido en memoria principal, entonces algún nodo dispone del bloque válido en su caché. Si es el nodo O, no se crea paquete de reenvío y se devuelve `RpBloqueInv(B)`; por otra parte, si no es el nodo O, se envía el paquete de reenvío de lectura con invalidación, al nodo que tenga válido a `B`.

El nodo O pondrá el estado de `B` a Pendiente de inválido. Cuando O reciba la confirmación de invalidación así como el bloque, pasará a estado Inválido y responderá a S con el bloque confirmando la invalidación, mediante el paquete `RpBloqueInv(B)`.

3. Paquetes de respuesta (`Rp`) desde P hasta O.

a) Cuando P recibe el paquete `RvLec(B)`, antes de responder cambia el estado del bloque `B` de Modificado a Compartido. Posteriormente, responde con el paquete de respuesta con bloque `RpBloque(B)`.

Cuando O recibe el paquete:

- Escribe el bloque en memoria principal.
- Pasa el estado del bloque de Pendiente de válido a Válido.
- Responde con el bloque al nodo S.

b) Cuando P recibe el paquete `RvInv(B)`, antes de responder cambia el estado del bloque `B` a Inválido. Posteriormente, responde con el paquete de respuesta confirmando invalidación `RpInv(B)`.

c) Cuando P recibe el paquete `RvLecEx(B)`, antes de responder combia el estado del bloque `B` a Inválido. Posteriormente, responde con el paquete de respuesta con bloque confirmando invalidación `RpBloqueInv(B)`.

3.2.9. Protocolo MSI para procesadores NUMA con difusión

Al usar difusión, enviamos los paquetes de petición a todas las cachés, tengan o no copia del bloque. Por tanto, no nos es necesario almacenar en el directorio de memoria principal información sobre las cachés con copias del bloque. Se trata del protocolo más sencillo que puede llevarse a cabo con posescritura, invalidación y con difusión.

Los estados de los bloques en caché y en memoria coinciden con los del protocolo MSI para NUMA sin difusión.

Paquetes generados por los controladores e interacciones

A continuación, desarrollamos igual que en el caso sin difusión varias interacciones que pueden suceer entre los tres roles principales de los nodos de un computador NUMA, comentando al mismo tiempo los paquetes empleados.

1. Paquetes de petición (`Pt`) desde S hasta cualquier nodo N.

   a) El nodo solicitante S envía cualquier paquete de petición (son los mismos que se usaban en el protocolo sin difusión) a todos los nodos del computador. Por tanto, lo reciben también el nodo O y los P. Son los mismos paquetes que generaba el nodo S al nodo O en el protocolo sin difusión, como consecuencia de los mismos eventos.

2. Paquetes de respuesta (`Rp`) desde nodos N a O.

   a) Si P recibe el paquete `PtLec(B)` desde S de un bloque `B` que tiene en estado Modificado (estará por tanto en estado Inválido en memoria), P cambia el estado del bloque `B` en su caché a Compartido y genera un paquete de respuesta con bloque `RpBloque(B)`.

   Cuando O reciba el paquete:

   - Escribirá el bloque en memoria principal.
   - Pasará el estado del bloque a válido.
   - Responderá con el bloque al nodo S.

   El resto de nodos no hacen nada.

   b) Si P recibe el paquete `PtEx(B)` o `PtLecEx(B)` desde S de un bloque `B` que tenga en estado Compartido o Inválido, invalidarán la copia de `B` en sus cachés (en caso de que estuviera en Compartido) y responderán a O confirmando la invalidación con el paquete `RpInv(B)`.

   c) Si P recibe el paquete `PtLecEx(B)` desde S de un bloque `B` que tenga en estado Modificado, primero invalidará la copia de su bloque y responderá a O con el bloque, confirmando la invalidación con el paquete `RpBloqueInv(B)`.

3. Paquetes de respuesta (`Rp`) desde O a S.

   a) Si O recibe del nodo S un paquete `PtLec(B)`:

- Si `B` está en estado válido, genera el paquete de respuesta `RpBloque(B)`.
- Si `B` está en estado inválido, pone el estado del bloque `B` a Pendiente de validación y esperará a que la caché con una copia válida del bloque envíe dicho bloque. En dicho caso, cambiará el estado a válido y generará el paquete de respuesta `RpBloque(B)`.

b) Si O recibe del nodo S un paquete `PtEx(B)`:

- Cambia el estado del bloque `B` a Pendiente de invalidación.
- Espera a recibir la confirmación de invalidación de todos los nodos N.
- Una vez recibidas todas las confirmaciones, cambia el estado de `B` a inválido y responde con el paquete `RpInv(B)`.

c) Si O recibe del nodo S un paquete `PtLecEx(B)`:

- Si tiene a `B` en estado Válido, lo cambiará a estado Pendiente de inválido y esperará a recibir la respuesta de invalidación en todas las cachés de los nodos N. Posteriormente, cambiará a estado Inválido y responderá con el paquete `RpBloqueInv(B)`, junto con el bloque `B`.
- Si tiene a `B` en estado Inválido, esperará a recibir la respuesta de invalidación en todas las cachés de los nodos N, junto con la respuesta del nodo P, que además contendrá el bloque `B`, el cual se actualizará en memoria y enviará a S en el paquete `RpBloqueInv(B)`.

3.3. Consistencia del sistema de memoria

El modelo de programación de memoria compartida ofrece varias ventajas al programador respecto al modelo basado en paso de mensajes: evita (o al menos relaja) la necesidad de distribuir la carga de trabajo (código y datos) entre los distintos procesadores. Esta es la principal razón por la que los multiprocesadores han tenido una amplia aceptación tanto en el mundo comercial como en computación científica (frente a los multicomputadores).

Sin embargo, para programar de forma correcta un multiprocesador, los programadores deben conocer bien cómo se comporta el sistema de memoria frente a las instrucciones que el programador coloca en el código paralelo a ejecutar.

3.3.1. Objetivos

Esta sección está orientada a adquirir los conocimientos necesarios para:

- Explicar el concepto de consistencia.
- Distinguir entre coherencia y consistencia.
- Distinguir entre el modelo de consistencia secuencial y los modelos relajados.
- Distinguir entre los diferentes modelos de consistencia relajados.

3.3.2. Concepto de consistencia

Para ejecutar instrucciones en el procesador es necesario (a veces) disponer de los datos requeridos de la memoria para completar dichas instrucciones. Por tanto, cuanto antes se consiga leer de memoria, antes se completarán estas instrucciones en la CPU. Es por esto que podemos desear que, por ejemplo, las lecturas adelanten a las escrituras que no son tan necesarias en dicho momento (esto es, que lecturas que suceden ciertas escrituras se ejecuten antes que dichas escrituras), con el fin de aumentar las prestaciones.

Notemos que mientras que las lecturas son un simple acceso a caché (de un ciclo) o una serie de envío de paquetes para traer el bloque a caché (en caso de fallo de caché) según lo visto en el apartado de coherencia de memoria, las escrituras siempre provocan el envío de múltiples paquetes, lo que implica un retardo mucho mayor al de las lecturas en promedio.

Podríamos también plantear un modelo en el que lecturas adelanten a lecturas (a aquellas que den fallo en caché y por tanto deben esperar a que la caché obtenga la información, introduciendo un retardo), escrituras que adelanten a lecturas o escrituras que adelanten a otras, obteniendo en cada tipo una leve optimización del código. De hecho, un código que no implemente ningún tipo de adelantamiento, probablemente se ejecute de forma anormalmente lenta. Las herramientas de programación, los compiladores y el propio hardware pueden realizar adelantamientos de instrucciones de acceso a memoria, con el fin de mejorar las prestaciones y reducir

tiempos de forma considerable.

Este tipo de adelantamientos entre instrucciones de acceso a memoria (tanto de lectura como de escritura) sólo se pueden realizar si los accesos a memoria son sobre distintas posiciones de memoria (ya que si no nos toparíamos con dependencias de datos, como de tipo RAW).

Antes de continuar con la sección, introducimos una notación que usaremos a lo largo de esta:

Notación. Cuando notemos $A \to B$, estaremos diciendo que la instrucción A precede a la instrucción B, pero que por un cierto motivo se ejecutó la instrucción B antes que la A. Podemos leer $A \to B$ como "B adelanta a A".

Usualmente usaremos esta notación para indicar que, por ejemplo, las lecturas adelantan a las escrituras: $W \to R$.

Motivada ya la razón por la que tenemos que desear que ciertas instrucciones adelanten a otras, podemos ya dar una definición informal de modelo de consistencia de memoria. Un modelo de consistencia de memoria especifica la forma en la que ciertas instrucciones relativas a memoria adelantarán a otras. Sin embargo, este modelo debe garantizar que se siga una secuencia lógica en la forma de los adelantamientos. En concreto, debe garantizarse que:

- Cada lectura de una dirección proporcione el último valor escrito en dicha dirección.
- Si se escribe varias veces en la misma dirección, se debe retornar el último valor escrito (el de la última escritura según el orden del programa).
- Si se escribe en una dirección a la que previamente se ha accedido para leer, no se debe obtener en la lectura previa el valor escrito posteriormente, sino el que había antes de la escritura.
- No se puede escribir en una dirección si la escritura depende de una condición que no se cumple. Esto se ilustra en el Código fuente 1, ya que si permitimos adelantamientos, puede que la evaluación de la variable booleana no sea el esperado en el momento de su evaluación, sucediendo la escritura a pesar del valor de la variable.

```
if(var_booleana){
    A = 1;
}
```

Código fuente 1: Ejemplo de escritura condicional.

Antes de continuar, cabe mencionar que las arquitecturas de memoria para multiprocesadores aprovechan el hecho de que los distintos procesadores pueden emitir instrucciones de acceso a memoria en paralelo, para realizarlas en paralelo. Sin embargo, los procesadores observan estos accesos de forma secuencial. Esto es, en caso de haberse realizado dos accesos en paralelo, los procesadores habrán observado que uno se ha realizado antes que otro.

3.3.3. Modelo de consistencia secuencial

Un multiprocesador presenta un modelo de consistencia secuencial si el resultado de cualquier ejecución coincide con el que se esperaría el programador. Definido formalmente, el modelo de consistencia secuencial garantiza:

Orden del programa. En cada procesador, se respeta el orden entre las instrucciones de memoria que se indican en el código que cada procesador ejecuta. De esta forma, no se realizan adelantamientos.

Orden global o Atomicidad. Todos los procesadores deben ver el mismo orden en el acceso a memoria. De esta forma, ningún procesador puede ver una escritura en memoria antes que el resto de procesadores, por ejemplo.

Una ejecución atómica en los accesos a memoria garantiza el orden global, ya que si una instrucción es atómica, globalmente parece que se ejecuta y completa antes de que comience la siguiente, al no poder introducir ningún otra instrucción durante la ejecución de esta. Por ello que al orden global a veces se le llame "atomicidad".

El modelo de consistencia secuencial es el modelo de consistencia de memoria más simple que podemos considerar.

3.3.4. Modelos de consistencia relajados

Antes de definir el modelo de consistencia secuencial, hablábamos de lo provechoso que es que ciertas instrucciones de memoria adelanten a otras, lo que nos permite eliminar latencias y aumentar tiempos de ejecución. Sin embargo, vemos cómo el modelo de consistencia secuencial no permite realizar adelantamiento de instrucciones. Es por esto por lo que se desarrollan los modelos de consistencia relajados, los cuales incumplen de alguna forma el orden del programa, o el orden global, o ambos.

Por ello, cuando queramos definir un modelo de consistencia que seguirá un multicomputador, bastará con especificar qué parte del orden del programa no se cumple (como permitir que lecturas adelanten a escrituras) o qué parte del orden global no se cumple.

Todavía no hemos dado un ejemplo ventajoso que sea resultado de no cumplir el orden global. Procedemos a desarrollarlo: puede ser deseable que, en un mismo procesador, una lectura de una posición de memoria k adelante a una escritura de la misma posición de memoria k. Si permitiéramos esto cumpliendo con el orden global, se produciría una clarísima falta de consistencia en el programa (obtenemos una lectura que no coincide con una escritura anterior). Es por esto por lo que puede relajarse el orden global (no cumplirse), de forma que la lectura en vez de acceder a memoria, lea el contenido de la petición de escritura (y por tanto, pudiendo acceder al contenido de la posición k antes que el resto de procesadores).

Notemos que este ejemplo también relaja el orden del programa (al permitir que lecturas adelanten a escrituras).

Actualmente, todos los modelos de consistencia de memoria permiten, al menos, el adelantamiento $W \rightarrow R$ (lecturas adelantan escrituras), debido al retardo que eliminan en los programas. El modelo de consistencia secuencial sólo tiene utilidad teórica, al producir programas con grandes latencias en la práctica.

3.3.5. Consistencia software y hardware

Todo este tiempo hemos estado hablando de consistencia de memoria a nivel hardware, que es la consistencia que implementa el propio hardware. Aquel programador que programe a bajo nivel (como en ensamblador o que se encuentre desarrollando un sistema operativo) debe conocer bien el sistema de consistencia de su hardware, con el fin de desarrollar códigos que hagan lo que se espera que hagan y no se obtengan errores indeseados difíciles de localizar.

Sin embargo, nos encontramos con que herramientas de programación y compiladores pueden realzar adelantamientos de instrucciones de memoria con la finalidad de optimizar el código e incrementar las prestaciones de nuestro programa. Estas utilidades software siguen unas filosofías de reordenación (cada una la que desee) a las que nos referiremos como modelos de consistencias software.

De esta forma, los modelos de consistencia software proporicionan al programador una nueva capa de abstracción a la hora de desarrollar código a alto nivel. El programador no tiene por qué conocer el modelo de consistencia hardware de su máquina. Serán las propias herramientas de programación las que determinen cómo reordenar el código para que todo parezca haberse ejecutado siguiendo un modelo de consistencia secuencial (aunque este nunca se aplique en realidad).

3.3.6. Tipos de modelos de consistencia relajados

Como hemos comentado ya, en realidad los modelos de consistencia relajados son en realidad los verdaderos modelos de consistencia útiles (ya que si no, trabajaríamos con el modelo de consistencia secuencial, que deja ejecutar programas con latencias enormes). Para describir dichos modelos nos es suficiente con describir qué relaja:

- Indicar por qué no se garantiza el orden del programa: qué órdenes entre instrucciones no se garantizan debido a adelantamientos.
- Indicar por qué no se garantiza el orden global: en qué casos un procesador puede acceder de forma prematura a una posición de memoria.
- La necesidad de incluir nuevas instrucciones máquina en el repertorio del procesador que nos permitan mantener el orden secuencial.

La tercera cuestión no la hemos mencionado hasta el momento y es de crucial importancia: si disponemos de un modelo de consistencia que no nos permite especificar que un cierto código debe ejecutarse de forma secuencial sin permitir ningún adelantamiento, no tenemos forma de desarrollar códigos que implementen sincronizaciones entre procesadores (ya que estos códigos necesitan actualizar direcciones de memoria

en orden, protocolos que fallarán si se reordenan las instrucciones de memoria).

Cabe destacar también que la necesidad de disponer de códigos de sincronización es vital. Si no disponemos de códigos de sincronización y nos disponemos, por ejemplo, a ejecutar el código de el Código fuente 2 de forma que la línea 1 sea ejecutada por un procesador y la línea 2 por otro, no tenemos forma de garantizar que la variable `k` del segundo procesador sea igual a 1.

```
1  int A = 1;
2  int k = A;
```

Código fuente 2: Código secuencial a paralelizar.

Es por esto por lo que necesitamos tener códigos de sincronización, los cuales para implementarlos necesitan garantizar de alguna forma la secuenciación de las instrucciones de acceso a memoria. De donde surge la necesidad de que los modelos de consistencia de memoria relajados implementen dichas instrucciones.

A continuación, desarrollamos dos modelos de consistencia de memoria relajados distintos, que nos permitirán terminar de entender la importancia de la consistencia.

Modelo de consistencia de los procesadores x86

Es el modelo de consistencia relajado más sencillo. Para describirlo, desarrollamos las tres cuestiones que comentamos anteriormente:

- Orden del programa: Permite que las lecturas adelanten a las escrituras: $W \rightarrow R$.
- Orden global: Sólo permite acceso adelantado a sus propias modificaciones de la memoria.
- Instrucciones para garantizar un orden: `xchange` y la posibilidad de incluir `lock` en las instruccines aritméticas y lógicas.

Ambas instrucciones garantizan un orden (es decir, todas las instrucciones de lectura y escritura que precedan a `xchange` o a una con `lock` se ejecutarán antes que estas y todas las que sucedan se ejecutarán después de estas instrucciones). La instrucción `xchange` implementa una especie de *swap* (intercambio) entre un registro y memoria. Por otra parte, si incluimos `lock` delante de una instruccion aritmética o lógica, nos permite implementar un cerrojo en la ejecución de la instrucción (en realidad, se produce un acceso al bus de forma exclusiva por parte del procesador que lanzó la instrucción con `lock`).

Recordemos que los procesadores x86 tienen una arquitectura de instrucciones CISC (instrucciones complejas), pero que en realidad tienen un procesador RISC de forma interior con un decodificador que cambia instrucciones CISC por RISC. De esta forma, en realidad la instrucción `xchange` realiza una lectura y una escritura, pero de forma atómica.

Modelo de consistencia de los procesadores ARMv7

A modo de resumen, podemos decir que su modelo de consistencia lo relaja todo:

- Orden del programa: Permite cualquier tipo de adelantamiento.
- Orden global: Lo relaja totalmente.
- Instrucciones para garantizar un orden: `dmb`.

La instrucción `dmb` sólamente tiene la utilidad de garantizar la ejecución de las instrucciones a memoria que se incluyen antes de ella antes de ejecutar la instrucción `dmb`, además de ejecutar las instrucciones que se encuentran tras ella sólamente tras su ejecución.

```
1  int A = 1;
2  int B = A;
3  int C = A + B;
4  dmb;
5  int A = C + B;
6  int C = C - A;
7  int B = B + 1;
```

Código fuente 3: Ejemplo de uso de dmb.

Por ejemplo, en el pseudocódigo de el Código fuente 3 (la instrucción `dmb` es a nivel de ensamblador y el resto de código es código de C), ninguna de las instrucciones de la línea 5 hacia abajo puede adelantar a ninguna de las instrucciones hasta la lína 3; y ninguna de las instrucciones de hasta la línea 3 puede ser ejecutada tras una instrucción que comienza a partir de la línea 5.

Podemos pensar en la instrucción `dmb` como en una especia de barrera (no confundir con una barrera en el contexto de sincronización) relativa a la memoria: cuando llegamos a una instrucción `dmb` se asegura que ninguna de las instrucciones anteriores se haya ejecutado y que todas las siguientes no han comenzado a ejecutarse todavía.

3.4. Sincronización

La comunicación entre distintos procesadores cuando estos comparten memoria (como en multiprocesadores), se lleva a cabo a través de variables compartidas. Debemos saber cómo funcionan los protocolos que permiten realizar sincronizaciones, además de implementarlos, todo esto teniendo en cuenta el modelo de consistencia hardware de nuestro computador, que es de vital importancia a la hora de desarrollar códigos que permitan realizar sincronizaciones, los cuales son muy propensos a alterar su funcionalidad por los adelantamientos de instrucciones de memoria.

3.4.1. Objetivos

Tras esta sección, debería ser capaz de:

- Explicar por qué es necesaria la sincronización en multiprocesadores.
- Describir las primitivas para sincronización que ofrece el hardware.
- Implementar cerrojos simples, cerrojos con etiqueta y barreras a partir de instrucciones máquina de sincronización y ordenación de accesos a memoria.

3.4.2. Necesidad de sincronización

Motivamos la necesidad de la sincronización entre procesadores con el siguiente ejemplo, que probablemente sea el más sencillo que podamos considerar.

Ejemplo. Estamos desarrollando un programa paralelo que será ejecutado por dos hilos (que intentaremos que se ejecuten en dos procesadores de forma paralela). En un determinado instant, queremos copiar el contenido de la variable compartida `A` actualizada por el procesador 1 al procesador 2 (la numeración no es relevante). Plasmamos en código secuencial lo que queremos hacer en el Código fuente 4:

```
1  A = valor;  // código a ejecutar en P1
2  copia = A;  // código a ejecutar en P2
```

Código fuente 4: Código a paralelizar.

De esta forma, el programa que tengamos en el procesador 1 tendrá en alguna parte de su código la instrucción de la línea 1 y el procesador 2 tendrá la de la línea 2. El problema es asegurarnos de que el procesador 2 ejecute dicha instrucción tras haber ejecutado el procesador 1 su instrucción, porque si no estaremos copiando en el procesador 2 el antiguo valor de la variable `A`, que probablemente no coincida con el valor de `valor`.

Deseamos buscar un código del siguiente estilo (tras esta sección, sabrá cómo implementarlo y cómo usarlo), que plasmamos en el Código fuente 5 y 6.

```
A = valor;
// Avisar a procesador 2 de la actualización
```

Código fuente 5: Código para el procesador 1.

```
// Esperar a que el procesador 1 avise de la actualización
copia = A;
```

Código fuente 6: Código para el procesador 2.

No es difícil imaginar una forma en la que usar código basado en variables compartidas para sincronizar ambos procesadores de forma que se ejecute de forma correcta el código anterior (garantizando que la instrucción `A = valor;` del procesador 1 se ejecuta antes que `copia = A;` del procesador 2). Esto lo vemos en el siguiente ejemplo, resaltando la necesidad de tener en cuenta el modelo de consistencia hardware del computador en el que estemos trabajando:

Ejemplo. Proponemos el código del Código fuente 7 y 8 como solución del ejemplo anterior:

```
// A y K son variables compartidas, inicialmente K = 0
A = valor;
K = 1;      // Indica que ya se ha copiado el valor
```

Código fuente 7: Código para el procesador 1.

```
// A y K son variables compartidas, inicialmente K = 0
while(K == 0){;}    // Esperar a que se actualice K
copia = A;
```

Código fuente 8: Código para el procesador 2.

Si ahora pensamos en qué ocurriría en los dos casos de ejecución más sencillos:

El código del procesador 1 se ejecuta antes. Entonces, se copia en `A` el valor de `valor` y se pone `K = 1`. Al ejecutar el código del procesador 2, tendríamos `K = 1`, por lo que no se realizaría el bucle y se copiaría el contenido de `A`.

El código del procesador 2 se ejecuta antes. Entonces, entraríamos en un aparante bucle infinito, el cual se detendría cuando el procesador 1 ejecute su código, copiando ya el contenido de `valor` en `A` y haciendo que el procesador 2 salga del bucle, tras lo cual copiaría el contenido de `A`.

Si pensamos esto, el código funcionaría perfectamente y no obtendríamos ningún error. Sin embargo, no estamos teniendo en cuenta el modelo de consistencia de memoria de nuestro computador (el razonamiento no está bien del todo). Procedemos a distinguir casos:

Modelo de consistencia secuencial. En dicho caso, se produce lo que es lógico esperar según el código desarrollado. En este caso el razonamiento anterior es válido.

Modelo de consistencia de un procesador x86. En este caso, las lecturas pueden adelantar a las escrituras. Sin embargo, en el código del procesador 1 sólo tenemos dos escrituras, luego se respeta el orden del programa. En el procesador 2, tenemos muchas lecturas (tantas como iteraciones del bucle) y una escritura. Por el orden en el que están también se respeta el orden del programa, el código funcionaría como se espera.

Modelo de consistencia de un procesador ARMv7. En dicho caso, los códigos de ambos procesadores no funcionan como esperamos:

- Procesador 1: La escritura en la variable `K` puede adelantar a la de `A`, y por tanto avisando al procesador 2 (según la lógica del código) de que la variable `A` ya ha sido modificada, cuando esto no es cierto.
- Procesador 2: Al no garantizarse orden global, puede suceder que la modificación de la variable `K` se haga visible antes que la modificación de la variable `A` (incluso si el código del procesador 1 se ejecutó en correspondencia con el orden del programa), modificando la variable `A` antes de lo previsto (todavía no se ha actualizado su contenido).

  Cabe destacar que esta es una casuística de error, pero que con este modelo de consistencia podría pasar cualquier cosa: que una lectura adelante a otra, que la escritura en `copia` adelante a una lectura de `K`, ... hay múltiples situaciones en las que sucedería algo no esperado.

Para garantizar el orden secuencial del programa (crucial a la hora de desarrollar este código de sincronización entre dos procesadores) necesitamos, como ya se vió en la sección de consistencia, usar las instrucciones que nos proporcionan los modelos de consistencia para asegurar ejecuciones secuenciales del mismo.

En este caso, sólo debemos introducir dichas instrucciones en el modelo de consistencia de los ARMv7 (en los ejemplos considerados), que sólo dispone de la instrucción `dmb` (la cual nos es suficiente). El código correcto para este tipo de procesadores lo mostramos en el siguiente pseudocódigo (hablamos de pseudocódigo por estar mezclando instrucciones ensamblador como `dmb` con lenguaje C) del Código fuente 9 y 10:

```
// A y K son variables compartidas, inicialmente K = 0
A = valor;
dmb;
K = 1;
```

Código fuente 9: Código para el procesador 1.

```
// A y K son variables compartidas, inicialmente K = 0
while(K == 0){;}
dmb;
copia = A;
```

Código fuente 10: Código para el procesador 2.

De esta forma, obtendríamos el resultado deseado para sincronizar los dos procesadores.

Mostramos ahora un último ejemplo como motivación de los dos protocolos a desarrollar que son tan necesarios para realizar la sincronización entre procesadores.

Ejemplo. Queremos paralelizar el siguiente código secuencial del Código fuente 11:

```
1 for(int i = 0; i < n; i++){
2     sum = sum + a[i];
3 }
4 printf(sum);
```

Código fuente 11: Código secuencial a paralelizar.

Es decir, queremos paralelizar la suma de un vector y su impresión en consola. Para ello, podríamos pensar en un primer código del Código fuente 12.

```
1 // La variable sump es privada y a, sum son compartidas
2 // inicilmente sum, sump = 0
3 for(i = id_thread; i < n; i = i+num_threads){
4     sump = sump + a[i];
5 }
6
7 sum = sum + sump;
8 if(id_thread == 0) printf(sum);
```

Código fuente 12: Código paralelizado con condiciones de carreara.

Este presenta un problema que podemos ya intuir: una condición de carrera en la línea 7, al acceder todas hebras a una variable compartida de forma no exclusiva.

Para solucionar la condición de carrera y garantizar el acceso exclusivo a la variable compartida `sum`, podemos hacer uso de un cerrojo (que suponemos ya implementado), como hacemos en el Código fuente 11.

```
1  // La variable sump es privada y a, sum son compartidas
2  // inicilmente sum, sump = 0
3  for(i = id_thread; i < n; i = i+num_threads){
4      sump = sump + a[i];
5  }
6
7  lock();
8  sum = sum + sump;
9  unlock();
10
11 if(id_thread == 0) printf(sum);
```

Código fuente 13: Código paralelizado mejorado.

Hemos ya eliminado la anterior condición de carrera, pero tenemos una en la que nos habíamos fijado, en la línea 11: puede que la hebra 0 llegue al `if` antes de que el resto de hebras hayan terminado de sumar su aportación a la variable `sum`, imprimiendo en pantalla un resultado prematuro (y posiblemente erróneo) de la variable `sum`. Es por ello que en este código tenemos que usar además una barrera (que también suponemos ya implementada), como hacemos en el Código fuente 14.

```
1  // La variable sump es privada y a, sum son compartidas
2  // inicilmente sum, sump = 0
3  for(i = id_thread; i < n; i = i+num_threads){
4      sump = sump + a[i];
5  }
6
7  lock();
8  sum = sum + sump;
9  unlock();
10
11 barrier();
12 if(id_thread == 0) printf(sum);
```

Código fuente 14: Código paralelizado definitivo.

De esta forma, todas las hebras esperarán en la línea 12 antes de proseguir con su código (incluida la hebra 0, que era la que provocaba el fallo a la hora de imprimir).

Hemos visto ya en este ejemplo la necesidad de disponer de cerrojos y barreras. De hecho, con ambas nos es suficiente para desarrollar cualquier código paralelo. Esta sección está dedicada a comentar los tipos de cerrojos y barreras, así como de su implementación.

3.4.3. Soporte hardware

A la hora de usar cerrojos o barreras, disponemos de tres opciones:

- Usar las implementadas por las herramientas de programación que estemos usando (en caso de que estas disopngan de las mismas).
- Usar las ya implementadas en el sistema operativo en el que esté trabajando.
- Implementar usted mismo los propios cerrojos y barreras.

Independientemente de la opción que elija, es bueno conocer cómo se implementan estas dos. Antes de proceder a la implementación como tal, necesitamos conocer qué soporte nos ofrece el hardware para su desarrollo.

Como ya se ha comentado, las funciones de sincronización (cerrojos y barreras) utilizan instrucciones que provee el procesador para fijar explícitamente un orden en los accesos a memoria, con las instrucciones que nos provee cada modelo de consistencia de memoria.

Los primeros algoritmos usaban sólo instrucciones de acceso a memoria atómicas (eran una solución, pero de una alta complejidad y grandes tiempos de ejecución al necesitar muchas variables compartidas y operaciones con la finalidad de coordinar a ccesos a secciones críticas). Surgió la necesidad de desarrollar instrucciones que realizasen operaciones en secuencia sobre una misma variable, operaciones de lectura, modificación y escritura (llamadas *lectura-modificación-escritura*[5]) de forma atómica.

Estas instrucciones permiten realizar una coordinación entre procesos de forma rápida y sencilla. Además, el acceso atómico garantiza que ningún otro procesador pueda acceder a la posición durante el acceso.

A continuación, destacamos unas cuantas primitivas que pueden utilizarse para la lectura-modificación-escritura atómica de algunos procesadores (y multiprocesadores).

Test&Set.

Realiza de forma atómica sobre una variable `x` las siguientes operaciones:

- Lee en una variable local (es decir, en un registro del procesador) el contenido de la variable `x` y lo devuelve como resultado.
- Escribe un 1 en `x`.

Intercambio.

Realiza un intercambio de contenidos entre una variable compartida y un registro. Supone leer y escribir en memoria.

En una primitiva de intercambio para sincronización, se realizan (la lectura y escritura) de forma atómica. Es más general que `Test\&Set`, ya que permite almacenar cualquier valor. Un ejemplo es la instrucción `xchange` de los procesadores x86.

Fetch&Operation.

Es una familia de primitivas, formada por `Fetch&Add`, `Fetch&Or` y `Fetch&Increment`. Una primitiva de este grupo realiza sobre una variabe `x` las siguientes opreaciones:

[5]Dudo que este paréntesis sea necesario, pero lo incluyo por si acaso.

- Lee en una variable local (en un registro del procesador) el valor de `x` y lo devuelve como resultado.
- Escribe en `x` el resultado de realizar una operación (que siempre es de naturaleza asociativa y conmutativa) con el contenido de `x` y una variable local `a`.

  Con `Fetch&Add` la operación es una suma, con `Fetch&Or` un OR lógico y con `Fetch&Increment` incrementa en uno la variable `x`. Un ejemplo de `Fetch&Add` es la instrucción `lock xadd reg, mem` ($reg \leftarrow mem$, $mem \leftarrow reg + mem$), de los procesadores x86.

Compare&Swap.
Esta opera con dos variable locales `a` y `b` y con una variable compartida `x`. Realiza sobre la variable `x` de forma inninterrumpida las siguientes operaciones:

- Lee el contenido de `x`.
- Si el contenido de `x` coincide con el de `a` se intercambian los contenidos de `x` y de `b`.

LL y SC.
Llamado carga enlazada con almacenamiento condicional (*load linked/locked with store conditional*). Disponemos de dos instrucciones (`ll` y `sc`), las cuales se escriben de forma secuencial (primero `ll` y luego `sc`).

Si al ejecutar el almacenamiento condicional el procesador descubre (gracias a una línea que funciona a modo de chivato) que entre la instrucción de *linked load* y *store conditional* se ha modificado la posición de memoria con la que se trabajaba, no se realiza la escritura en memoria. Realmente no se realiza una operación de lectura-escritura atómica, sino que se realizan ambas impidiendo la escritura si la atomicidad fue violada[6]. Podemos ver un ejemplo de la funcionalidad de estas dos instrucciones en el Código fuente 15, donde la instrucción `sc(k, j)` no realiza su acción, al ser la posición de memoria k modificada por la variable local i.

El resto de primitivas de lectura-modificación-escritura previamente comentadas pueden implementarse mediante estas dos instrucciones.

```
1  ll(k, &i);
2  ll(k, &j);
3  // Modificar i
4  // Modificar j
5  sc(k, i);
6  sc(k, j);
```

Código fuente 15: Ejemplo de instrucciones ll y sc.

En definitiva, a la hora de realizar funciones para sincronización (tal y como vamos a hacer a continuación), es necesario conocer tanto las instrucciones del procesador

[6]Esto es, si una instrucción que puede modificar dicha posición se interpuso entre ambas.

que nos provee nuestro modelo de consistencia de memoria para ejecutar código de forma secuencial, como instrucciones que nos permitan implementar los códigos de forma eficiente, a elegir entre:

- Instrucciones atómicas de lectura-modificación-escritura.
- Instrucciones de lectura y escritura especiales (`ll` y `sc`).

Las arquitecturas de los procesadores x86 (y ARMv6) usan las instrucciones atómicas de lectura-modificación-escritura, mientras que ARMv7 y sus versiones sucesivas usan LL/SC.

Por último, cabe destacar los distintos tipos de primitivas que disponemos para sincronización, tanto a alto nivel como a bajo nivel, que podemos recoger en la Tabla 3.7.

Soporte software	Monitores Regiones críticas condicionales Cerrojos, barreras y semáforos
Suporte hardware	Instrucciones lectura-modificación-escritura, LL/SC, escritura y lectura atómicas, instrucciones del modelo de consistencia

Tabla 3.7: Primitivas para sincronización

3.4.4. Cerrojos

Los cerrojos (también llamados a veces *mutex*, de *mutual exclusion*) proporcionan una forma de asegurar la exclusión mutua entre hilos a la hora de acceder a una sección crítica del código que queremos que sea ejecutada "de forma secuencial"[7].

Tendremos dos estados en el cerrojo:

- Cerrojo abierto: No hay ningún hilo ejecutando la seccion crítica.
- Cerrojo cerrado: Hay un hilo ejecutando la seccion crítica.

Los cerrojos utilizan dos funciones para sincronizar los distintos hilos:

Cierre del cerrojo (lock).

Con esta función un proceso (o hilo) intenta adquirir el derecho de acceso a una región crítica (que probablemente contenga el uso de variables compartidas). Si se le deniega temporalmente el acceso, debe pasar a una etapa de **espera**. Su implementación debe cumplir que:

1. Si varios flujos de instrucciones intentan la adquisición (la ejecución de `lock()`) a la vez de un cerrojo abierto, sólo uno de ellos debe conseguir la adquisición y cerrar el cerrojo, el resto pasará a la espera.

[7] Que no haya dos hilos ejecutando la sección de forma simultánea, sino que se organicen para ir de uno en uno.

2. Si varios flujos de instrucciones intentan la adquisición a la vez de un cerrojo cerrado, a todos se les debe poner en una etapa de espera.

Apertura del cerrojo (unlock).
Lo ejecuta un proceso (o hilo) cuando ha terminado la ejecución de la región crítica y da paso a otro flujo de instrucciones. Esta:

- Abrirá el cerrojo en caso de no haber ningún flujo a la espera.
- Dejará pasar sólo a un flujo, en caso de haber uno o varios flujos a la espera.

Recordamos ahora el uso de un cerrojo que hicimos en un ejemplo anterior, en el Código fuente 13.

Hemos hablado antes de que tenemos que enviar a menudo procesos a etapas de espera. Esta etapa puede desarrollarse mediante dos mecanismos principales:

Espera ocupada.
Llamada *busy-waiting* o *spin-lock*, el proceso se queda en esperae ejecutando un ciclo en el que constantemente[8] consulta si ha cambiado el valor de alguna variable de control de la espera. Es fácil de implementar pero estamos desaprovechando temporalmente el procesador.

Bloqueo.
El proceso que debe esperar se bloquea por el Sistema Operativo, queda suspendido dejando el procesador a otro proceso. Utilizamos el procesador para algo útil pero si el tiempo de ejecución de la región crítica es ínfimo en comparación con el tiempo necesario en suspender y reanimar un proceso, estamos introduciendo sobrecarga innecesaria.

Por tanto, usaremos espera ocupada cuando la sección crítica tenga un tiempo de ejecución pequeño y el bloqueo cuando el tiempo de ejecución de la región crítica sea un tiempo considerable (hablando siempre en relación al Sistema Operativo[9]).

Los cerrojos de espera ocupada son la base de construcción de funciones software de alto nivel, tales como los semáforos o monitores, que se usan ampliamente en la construcción de los sistemas operativos paralelos y aplicaciones paralelas.

Como puede deducirse de todo esto, hay tres componentes esenciales en el desarrollo de un código para sincronización:

- Método de adquisición. En cerrojos, la función `lock()`.
- Método de espera. En cerrojos, a elegir entre espera ocupada o bloqueo.
- Método de liberación. En cerrojos, la función `unlock()`.

A continuación, seguiremos con el desarrollo de dos tipos de implementaciones de cerrojos con espera ocupada.

[8] De aquí viene el nombre *spin*.

[9] Un tiempo que puede parecer despreciable para nosotros para el Sistema Operativo puede parecer muy largo.

Cerrojos simples

Es la implementación más sencilla de un cerrojo, la cual se realiza con una variable compartida `k` que adopte dos valores: cerrojo abierto (0) o cerrojo cerrado (1).

La función `unlock()` abre el cerrojo simplemente escribiendo un 0 en `k`:

```
void unlock(k){
    k = 0;
}
```

Código fuente 16: Código de unlock para cerrojos simples.

La función `lock()` lee el contenido del cerrojo:

- Si está cerrado, el proceso queda esperando en un ciclo mientras que el cerrojo permanezca cerrado.
- Si está abierto, para que el proceso obtenga el cerrojo debe escribir un 1 de forma que él sea el único proceso que ha visto el 0 y escrito el 1. Para cumplir esto, puede usarse una instrucción de lectura-modificación-escritura con acceso atómico o LL/SC, para garantizar que sólo un proceso realiza esta funcionalidad.

Podríamos implementar el `lock()` de la siguiente forma:

```
void lock(k){
    while(leer-asignar_1-escribir(k) == 1){;}   // Espera
    k = 1;                                      // Pasa y cierra cerrojo
}
```

Código fuente 17: Código de lock para cerrojos simples.

Sin embargo, esta implementación escribe en la variable `k` siempre (incluso cuando esta ya está puesta a 1), lo que introduce una gran carga al saber ya qué sucede en los protocolos de coherencia con las escrituras. Es por esto que proponemos el siguiente código como mejora, el cual minimiza las escrituras realizadas.

```
void lock(k){
    do{
        while(k == 1){;}
    }while(leer-asignar_1-escribir(k) == 1);

    k = 1;
}
```

Código fuente 18: Código de lock para cerrojos simples mejorado.

Finalmente, debemos tener en cuenta el modelo de consistencia de memoria sobre el que estamos trabajando y añadir si es necesario las instrucciones convenientes para garantizar que los accesos a memoria tras la instrucción `lock` no adelanten a los accesos en la sección crítica (ya que si no se garantiza, no estamos garantizando el acceso exclusivo a la región crítica); asímismo, la instrucción `unlock` debe realizarse una vez garantizados los accesos a memoria de la región crítica.

Esto podría implementarse en un procesador ARMv7 con la instrucción `dmb`.

Cerrojos con etiqueta

Una crítica que podemos hacer a los cerrojos simples es la injusticia cometida a la hora de seleccionar el siguiente proceso a ejecutar la sección crítica cuando hay varios procesos esperando, ya que el criterio de selección es el que primero vea que el cerrojo está abierto.

Sin embargo, es lógico pensar que puede sernos de mayor utilidad un cerrojo que garantice un orden FIFO en cuanto al momento de la petición de la adquisición[10] (de la llamada a `lock()`).

Para implementarlo, hacemos uso de dos contadores: el contador de adquisición y el de liberación. Cuando un proceso ejecuta `lock()`, obtiene el número de procesos que han solicitado ya el cerrojo mediante el valor del contador de adquisición[11] e incrementa su valor en 1 para el siguiente. Posteriormente, pasa a espera mientras que su indicador no coincida con el contador de liberación. La función `unlock()` simplemente aumenta en uno el contador de liberación.

```
1  struct Cerrojo{
2      int lib;
3      int adq;
4  };
5
6  int inc(int contador){
7      return (contador + 1) % num_threads;
8  }
9
10 void unclock(Cerrojo cerrojo){
11     cerrojo.lib = inc(cerrojo.lib);    // Aumenta liberacion
12 }
13
14 void lock(Cerrojo cerrojo){
15     int etiqueta = cerrojo.adq;         // Obtiene valor adquisicion
16     cerrojo.adq = inc(cerrojo.adq);    // Aumenta adquisicion
17     while(etiqueta != cerrojo.lib){;}  // Espera
18 }
```

Código fuente 19: Ejemplo de implementación de cerrojo con etiqueta.

[10] Como si de una pescadería se tratara. De hecho, sigue su misma lógica.
[11] El ticket de la pescadería.

Los accesos a memoria han de ser mediante instrucciones de lectura-modificación-escritura atómicas o LL/SC y puede que sea necesario añadir instrucciones para garantizar el correcto orden de acceso a memoria, depende del tipo de modelo de consistencia que estemos considerando.

3.4.5. Barreras

Una barrera es un punto en el código en el que los flujos de instrucciones que colaboran en la ejecución del código se esperan entre sí. Una vez que todos han llegado a la barrera, salen de la misma y continúan ejecutando el código que se sitúa tras la barrera. Se usan para asegurar que todos los flujos han llegado hasta cierto punto antes de continuar con la ejecución de las siguientes instrucciónes. Recordamos el ejemplo de uso del Código fuente 14.

Implementación

Para implementar la versión más simple de una barrera necesitamos 3 variables de control por barrera. Para almacenar de alguna forma todas las barreras de un código, usaremos un array donde almacenaremos en cada posición estas 3 variables de control. Procedemos ahora a concretar las tres variables de control que tenemos que usar a la hora de implementar una barrera:

Contador. Necesitamos un contador para llevar la cuenta del número de hebras que han llegado ya a la barrera (las cuales pondremos en estado de espera). Cuando el contador sea igual al número total de hebras, abriremos la barrera dejando pasar a todas las hebras.

Bandera. Indicativo de cuando la barrera está cerrada (no deja pasar a ninguna hebra que llegue a la barrera), que es usualmente el valor 0 y 1 cuando está abierta. Por tanto, el método de espera ocupada será un ciclo que se ejecute mientras la bandera no sea 1.

Cerrojo. Es necesario un cerrojo para realizar de forma correcta los accesos a las variables compartidas (al contador y a la bandera).

De esta forma, tenemos una primera implementación en el Código fuente 20. Cuando una hebra llegue al punto en el que queremos implementar la barrera, simplemente ha de llamar a la función `barrera`, especificándole qué barrera quiere usar (mediante un identificador) y el número total de hebras que se encuentra ejecutando la región paralela: `barrera(num_barrera, num_threads());`. Es de importancia que la variable `cuenta_local` sea privada, para el buen funcionamiento de la barrera[12].

Este primer código presenta un error difícil de detectar que se produce cuando se reusa una misma barrera en al menos dos partes distintas del código (cuando se usó una vez hace tiempo y se desea volver a usar otra barrera que, por ahorrar memoria, se decide usar la misma). Para sacar a la luz dicho error, el cual **puede dejar programas enteros bloqueados**, nos preguntamos qué sucedería si algún flujo que ya se encuentra esperando en la barrera se suspende (ya sea por el Sistema Operativo o por otro ente), llegan todas las hebras a la barrera por lo que se

[12]Se comentó en clase que debía ser privada pero no sé por qué, contactar conmigo si se sabe.

```
1  struct Barrera{
2      int contador;
3      int bandera;
4      Cerrojo cerrojo;
5  }
6
7  void barrera(int id, int num_threads){
8
9      // Cerrojo para la sección crítica
10
11     lock(barreras[id].cerrojo);
12
13     // Si es la primera hebra en llegar cierra la barrera
14     if(barreras[id].contador == 0) barreras[id].bandera = 0;
15     // Aumenta el contador y guarda su valor en variable PRIVADA
16     int cuenta_local = ++barreras[id].contador;
17
18     unlock(barreras[id].cerrojo);
19
20     // Si todas las hebras han llegado
21     if(cuenta_local == num_threads){
22         barreras[id].contador = 0;  // Para un próximo uso
23         barreras[id].bandera = 1;   // Libera de espera
24     }else{
25         // Pone a la hebra en espera
26         while(barreras[id].bandera == 0) {;}
27     }
28 }
```

Código fuente 20: Ejemplo de implementación de barrera.

liberaría la barrera (pero nuestra hebra sigue suspendida, luego no ha avanzado de la función de la barrera), y puede que alguna hebra llegue al siguiente uso de la misma barrera antes de que nuestra hebra suspendida se reanude, con lo que volvería a establecer la bandera a 0, haciendo que nuestra hebra nunca pase del uso de la barrera en el que se suspendió y que el resto de las hebras se queden esperando por siempre (ya que nunca igualarán el número total de hebras, al haber al menos una que no superó la barrera anterior) en el siguiente uso de la misma barrera.

Para solventar este gran problema, proponemos la siguiente implementación de la función `barrera`, que puede observarse en el Código fuente 21. En este caso, en vez de que el valor 0 es barrera cerrada y 1 es barrera abierta, contamos con una variable privada `bandera_local` en cada hebra, de forma que en el primer uso de la barrera 0 será cerrado y 1 abierto y en el siguiente uso de la barrera al revés: 1 cerrado y 0 abierto. De esta forma, solventamos de forma elegante el problema que surgió con el código anterior.

```
1  void barrera(int id, int num_threads){
2      // Variable privada de cada hebra,
3      // complementa la bandera anterior
4      int bandera_local = !bandera_local;
5
6      // Acceso a sección crítica
7
8      lock(barreras[id].cerrojo);
9      // Aumenta el contador y guarda su valor en variable privada
10     int cuenta_local = ++barreras[id].contador;
11     unlock(barreras[id].cerrojo);
12
13     // Si todas las hebras han llegado
14     if(cuenta_local == num_threads){
15         barreras[id].contador = 0;      // Para un próximo uso
16
17         // Si todas han llegado, todas tienen la misma bandera local
18         barreras[id].bandera = bandera_local;
19     }else{
20         // Pone a la hebra en espera
21         while(barreras[id].bandera != bandera_local) {;}
22     }
23 }
```

Código fuente 21: Ejemplo de implementación de barrera SIN FALLOS.

4. Relaciones de Problemas

4.1. Arquitecturas Paralelas

Ejercicio 4.1.1. 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:

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

   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:

$$T_p = NI \cdot CPI \cdot T_c = T_c \cdot \sum_i NI_i \cdot CPI_i =$$

$$= T_c \cdot NI \cdot \left(\underbrace{0{,}2 \cdot 4}_{LD} + \underbrace{0{,}1 \cdot 3}_{ST} + \underbrace{0{,}25 \cdot 3}_{FP} + \underbrace{0{,}15 \cdot 8}_{FLT\ POINT} + \underbrace{0{,}3 \cdot 3}_{BRANCH} \right) =$$

$$= T_c \cdot NI \cdot 3{,}95$$

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:

$$T_p = NI \cdot CPI \cdot T_c = T_c \cdot \sum_i NI_i \cdot CPI_i =$$

$$= T_c \cdot NI \cdot \left(\underbrace{0{,}2 \cdot 4}_{LD} + \underbrace{0{,}1 \cdot 3}_{ST} + \underbrace{0{,}25 \cdot 6}_{FP} + \underbrace{0{,}15 \cdot 3}_{FLT\ POINT} + \underbrace{0{,}3 \cdot 3}_{BRANCH} \right) =$$

$$= T_c \cdot NI \cdot 3{,}95$$

   La expresión de la ganancia, por tanto, es:

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

   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 4.1.2. 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 $N_{\text{etapas}} \cdot T_c$. Por tanto, el tiempo segmentado es:

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

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 $N \to \infty$, por lo que:

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

Respecto a la productividad del cauce, se tiene que:

$$P(N) = \frac{N}{T^s(N)} = \frac{N}{4 \cdot T_c + (N-1)T_c} = \frac{N}{150 \cdot (3+N)} \,{}^{\text{op}}/_{\text{ns}} = \frac{N}{150 \cdot (3+N)} \cdot 10^3 \,{}^{\text{Mop}}/_{\text{s}}$$

La productividad máxima del cauce es:

$$P_{\text{máx}} = \lim_{N\to\infty} P(N) = \lim_{N\to\infty} \frac{N}{150 \cdot (3+N)} \cdot 10^3 = \frac{10^3}{150} \approx 6{,}667 \,{}^{\text{Mop}}/_{\text{s}}$$

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

   Tenemos que:

$$\begin{aligned} \frac{N}{150 \cdot (3+N)} \cdot 10^3 = 0{,}9 \cdot P_{\text{máx}} = 0{,}9 \cdot \frac{10^3}{150} \Longrightarrow \frac{N}{3+N} = 0{,}9 \Longrightarrow \\ \Longrightarrow N = 2{,}7 + 0{,}9N \Longrightarrow 0{,}1N = 2{,}7 \Longrightarrow N = 27 \end{aligned}$$

   Por tanto, a partir de 27 operaciones ejecutadas se consigue una productividad igual al 90 % de la productividad máxima.

Ejercicio 4.1.3. 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 intrucciones.

- 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 instucción 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):

$$T_{CPU}^1 = NI^1 \cdot (\underbrace{0{,}2 \cdot 4}_{BR} + \underbrace{0{,}8 \cdot 3}_{CMP+\text{Resto}}) \cdot T_c^1 = NI^1 \cdot 3{,}2 \cdot 0{,}75 \cdot T_c^2 = NI^1 \cdot 2{,}4 \cdot T_c^2$$

$$T_{CPU}^2 = NI^1 \cdot (\underbrace{0{,}2 \cdot 4}_{CMPBR} + \underbrace{0{,}6 \cdot 3}_{\text{Resto}}) \cdot T_c^2 = NI^1 \cdot 2{,}6 \cdot T_c^2$$

Por tanto,

$$\frac{T_{CPU}^1}{T_{CPU}^2} = \frac{2{,}4}{2{,}6} = \frac{12}{13} \approx 0{,}923 \Longrightarrow T_{CPU}^1 < T_{CPU}^2$$

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

Ejercicio 4.1.4. ¿Qué ocurriría en el problema del ejercicio anterior (Ejercicio 4.1.3) si el ciclo de reloj fuese únicamente un 10 % mayor para la ALT2?

En este caso, el tiempo de ciclo de la ALT1 sería:

$$T_c^1 = T_c^2 - 0{,}1T_c^2 = 0{,}9T_c^2$$

Por tanto, los tiempos de ejecución serían:

$$T_{CPU}^1 = NI^1 \cdot (\underbrace{0{,}2 \cdot 4}_{BR} + \underbrace{0{,}8 \cdot 3}_{CMP+\text{Resto}}) \cdot T_c^1 = NI^1 \cdot 3{,}2 \cdot 0{,}9 \cdot T_c^2 = NI^1 \cdot 2{,}88 \cdot T_c^2$$

$$T_{CPU}^2 = NI^1 \cdot (\underbrace{0{,}2 \cdot 4}_{CMPBR} + \underbrace{0{,}6 \cdot 3}_{\text{Resto}}) \cdot T_c^2 = NI^1 \cdot 2{,}6 \cdot T_c^2$$

Por tanto,

$$\frac{T_{CPU}^1}{T_{CPU}^2} = \frac{2{,}88}{2{,}6} = \frac{72}{65} \approx 1{,}1077 \Longrightarrow T_{CPU}^1 > T_{CPU}^2$$

Por ser $T_{CPU}^1 > T_{CPU}^2$, concluimos que la opción ALT2 es la mejor, en cuanto a tiempos de ejecución.

Ejercicio 4.1.5. 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 % `LOAD`s (4 CPI), el 12 % `STORE`s (4 CPI) y el 24 % `BRANCH`s (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, donde el superíndice 1 denotará la primera alternativa y el superíndice 2 denotará la segunda:

I_i^1	CPI_i^1	NI_i^1	I_i^2	CPI_i^2	NI_i^2
Instrucción ALU	3 ciclos	$0{,}43 \cdot NI^1$	`ALU r,r`	3 ciclos	$0{,}3225 \cdot NI^1$
			`ALU r,m`	4 ciclos	$0{,}1075 \cdot NI^1$
`LOAD`s	4 ciclos	$0{,}21 \cdot NI^1$	`LOAD`s	4 ciclos	$0{,}1025 \cdot NI^1$
`STORE`s	4 ciclos	$0{,}12 \cdot NI^1$	`STORE`s	4 ciclos	$0{,}12 \cdot NI^1$
`BRANCH`s	4 ciclos	$0{,}24 \cdot NI^1$	`BRANCH`s	5 ciclos	$0{,}24 \cdot NI^1$
					$0{,}8925 \cdot NI^1 = NI^2$

donde, cada NI_i^2 se ha calculado de la siguiente forma:

- Para `ALU r,r`, se ha usado que son el 75 % de las operaciones con la ALU, es decir, ${NI^2}_{\texttt{ALU r,r}} = 0{,}75 \cdot 0{,}43 \cdot NI^1 = 0{,}3225 \cdot NI^1$.

- Para `ALU r,m`, se ha usado que son el 25 % de las operaciones con la ALU, es decir, $NI^2_{\texttt{ALU r,m}} = 0{,}25 \cdot 0{,}43 \cdot NI^1 = 0{,}1075 \cdot NI^1$.
- Para `LOAD`s, se ha usado que en la alternativa 2 se hacen $0{,}1075 \cdot NI^1$ `LOAD`s menos que en la alternativa 1, ya que las instrucciones que usan un operando de memoria no es necesario que se traigan de memoria. Por tanto, tenemos que son $NI^2_{\texttt{LOADs}} = 0{,}21 \cdot NI^1 - 0{,}1075 \cdot NI^1 = 0{,}1025 \cdot NI^1$.

Calculamos los tiempos en CPU de ambas alternativas:

$$\begin{aligned} T^1_{CPU} &= NI^1[0{,}43 \cdot 3 + (0{,}21 + 0{,}12 + 0{,}24) \cdot 4] \cdot T_c \\ &= NI^1 \cdot 3{,}57 \cdot T_c \end{aligned}$$

$$\begin{aligned} T^2_{CPU} &= NI^1[0{,}3225 \cdot 3 + (0{,}1075 + 0{,}1025 + 0{,}12) \cdot 4 + 0{,}24 \cdot 5] \cdot T_c \\ &= NI^1 \cdot 3{,}4875 \cdot T_c \end{aligned}$$

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

Ejercicio 4.1.6. Se ha diseñado un compilador para la máquina `LOAD/STORE` del problema anterior (Ejercicio 4.1.5). Ese compilador puede reducir en un 50 % el número de operaciones con la ALU, pero no reduce el número de `LOAD`s, `STORE`s, y `BRANCH`s. 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.

El código no optimizado se representa con el superíndice 1, mientras que el código optimizado se representa con el superíndice 2.

I^1_i	CPI^1_i	NI^1_i	I^2_i	CPI^2_i	NI^2_i
Instrucción ALU	3 ciclos	$0{,}43 \cdot NI^1$	Instrucción ALU	3 ciclos	$0{,}215 \cdot NI^1$
`LOAD`s	4 ciclos	$0{,}21 \cdot NI^1$	`LOAD`s	4 ciclos	$0{,}21 \cdot NI^1$
`STORE`s	4 ciclos	$0{,}12 \cdot NI^1$	`STORE`s	4 ciclos	$0{,}12 \cdot NI^1$
`BRANCH`s	4 ciclos	$0{,}24 \cdot NI^1$	`BRANCH`s	4 ciclos	$0{,}24 \cdot NI^1$
					$0{,}785 \cdot NI^1 = NI^2$

Calculamos los tiempos en CPU de ambas alternativas:

$$\begin{aligned} T^1_{CPU} &= NI^1[0{,}43 \cdot 3 + (0{,}21 + 0{,}12 + 0{,}24) \cdot 4] \cdot T_c \\ &= NI^1 \cdot 3{,}57 \cdot T_c = NI^1 \cdot 3{,}57 \cdot \frac{1}{50 \cdot 10^6} \text{ seg} = NI^1 \cdot 7{,}14 \cdot 10^{-8} \text{ seg} \\ T^2_{CPU} &= NI^1[0{,}215 \cdot 3 + (0{,}21 + 0{,}12 + 0{,}24) \cdot 4] \cdot T_c \\ &= NI^1 \cdot 2{,}925 \cdot T_c = NI^1 \cdot 2{,}925 \cdot \frac{1}{50 \cdot 10^6} \text{ seg} = NI^1 \cdot 5{,}85 \cdot 10^{-8} \text{ seg} \end{aligned}$$

Calculamos ahora el número de MIPS de ambas alternativas:

$$MIPS^1 = \frac{NI^1}{T^1_{CPU} \cdot 10^6} = \frac{NI^1}{NI^1 \cdot 7{,}14 \cdot 10^{-8} \cdot 10^6} = \frac{1}{7{,}14 \cdot 10^{-2}} = 14 \text{ MIPS}$$

$$MIPS^2 = \frac{NI^2}{T^2_{CPU} \cdot 10^6} = \frac{0{,}785 \cdot NI^1}{NI^1 \cdot 5{,}85 \cdot 10^{-8} \cdot 10^6} = \frac{1}{7{,}4522 \cdot 10^{-2}} = 13{,}4188 \text{ MIPS}$$

Como podemos ver, de forma objetiva el código optimizado es mejor que el no optimizado, ya que $T^2_{CPU} < T^1_{CPU}$. No obstante, si solo nos fijásemos en el número de MIPS, podríamos pensar que el código no optimizado es mejor que el optimizado, ya que $MIPS^1 > MIPS^2$; pero sabemos que esta medida no es una medida objetiva, ya que tan solo depende de la frecuencia y de los ciclos por instrucción, y no tiene en cuenta el número de instrucciones.

Ejercicio 4.1.7. 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.

   Resumimos los datos del enunciado en la siguiente tabla:

I_i	CPI^b_i	NI_i	CPI^p_i
`LOAD`	4 ciclos	0,2 NI	4 ciclos
`STORE`	3 ciclos	0,15 NI	3 ciclos
Instrucción ALU	6 ciclos	0,4 NI	4 ciclos
`BRANCH`	3 ciclos	0,25 NI	3 ciclos

   Hemos de notar que el número de instrucciones de la ALU no cambia, ya que no se ha especificado que se reduzca el número de instrucciones de la ALU. Además, no se reduce de 6 a 3, ya que tan solo se reducen a la mitad los 4 ciclos que tarda en ejecutarse la instrucción de la ALU. Por tanto, se tiene que $CPI^2_{\text{ALU}} = 2 + 4 \cdot 1/2 = 4$. Por tanto, la ganancia para un número de instrucciones NI es:

   $$S(NI) = \frac{T^b(NI)}{T^p(NI)} = \frac{NI \cdot (0{,}2 \cdot 4 + 0{,}15 \cdot 3 + 0{,}4 \cdot 6 + 0{,}25 \cdot 3) \cdot T_c}{NI \cdot (0{,}2 \cdot 4 + 0{,}15 \cdot 3 + 0{,}4 \cdot 4 + 0{,}25 \cdot 3) \cdot T_c} = \\ = \frac{4{,}4}{3{,}6} = \frac{11}{9} \approx 1{,}222$$

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.

   Para ver que no es posible, usaremos la Ley de Amdahl. Supongamos que el porcentaje de instrucciones con operaciones que no son de la ALU es f. El factor de mejora de la ALU es $p = \frac{6}{4} = 1{,}5$. Por tanto, por la Ley de Amdahl, tenemos que:

   $$S \leqslant \frac{p}{1 + f(p-1)} = \frac{1{,}5}{1 + f \cdot 0{,}5} \leqslant 1{,}5 \qquad \forall f \in [0, 1]$$

Por tanto, tenemos que la ganancia está acotada por 1,5, por lo que no es posible obtener una ganancia mayor que 2.

Ejercicio 4.1.8. 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?

   El porcentaje de ejecución de las instrucciones que no son de coma flotante es $f = 1 - 0{,}13 = 0{,}87$; mientras que el factor de mejora es $p = 4/3$. Por tanto, la ganancia máxima es:

   $$S = \frac{T_b}{T_p} \leqslant \frac{p}{1 + f(p-1)} = \frac{4/3}{1 + 0{,}87 \cdot 1/3} \approx 1{,}0336$$

2. ¿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?

   Sea p el factor de mejora de la unidad de coma flotante. Tenemos que:

   $$S \leqslant \frac{p}{1 + f(p-1)} = \frac{p}{1 + 0{,}87(p-1)}$$

   Suponiendo que el factor de mejora es muy grande, es decir, $p \to \infty$, tenemos que:

   $$S_{\text{máx}} = \lim_{p \to \infty} \frac{p}{1 + 0{,}87(p-1)} = \frac{1}{0{,}87} \approx 1{,}149$$

3. ¿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?

   Calcularemos en primer lugar f, que representa el porcentaje de tiempo de cálculo con datos que no son en coma flotante:

   $$4 = S_{\text{máx}} = \lim_{p \to \infty} \frac{p}{1 + f(p-1)} = \frac{1}{f} \Longrightarrow f = \frac{1}{4} = 0{,}25$$

   Por tanto, el porcentaje de tiempo de cálculo con datos en coma flotante es $1 - f = 0{,}75$; es decir, el 75 %.

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 como flotante es el obtenido en el apartado 3?

   Se trata de buscar p, suponiendo que tenemos un 75 % de operaciones de coma flotante; es decir, $f = 0{,}25$. Tenemos que:

   $$2 = S_{\text{máx}} = \frac{p}{1 + f(p-1)} = \frac{p}{1 + 0{,}25(p-1)} = \frac{p}{0{,}75 + 0{,}25p} \Longrightarrow$$
   $$\Longrightarrow 2(0{,}75 + 0{,}25p) = p \Longrightarrow 1{,}5 + 0{,}5p = p \Longrightarrow 1{,}5 = 0{,}5p \Longrightarrow p = 3$$

   Por tanto, el tiempo de las operaciones en coma flotante debería reducirse a un tercio del tiempo que consumían antes.

Ejercicio 4.1.9. 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 $\texttt{N} = 10^9$:

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

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

   En este caso, necesitamos $T_{CPU} \leqslant 0{,}5$ segundos. Sabemos que el número de operaciones en coma flotante es $3 \cdot N$ (dos sumas y una multiplicación por cada iteración del bucle). Por tanto, necesitamos:

   $$GFLOPS = \frac{\text{nº FP}}{T_{CPU} \cdot 10^9} = \frac{3 \cdot N}{T_{CPU} \cdot 10^9} \geqslant \frac{3 \cdot 10^9}{0{,}5 \cdot 10^9} = 6$$

   Por tanto, necesitamos al menos 6 GFLOPS para poder ejecutar el código en menos de 0,5 segundos.

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?

   Tenemos que:

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

   Como buscamos que $T_{CPU} \leqslant 0{,}5$ segundos, tenemos que:

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

   Por tanto, 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 es al menos 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?

   En este caso, sea f el porcentaje de tiempo de cálculo con datos que no son en coma flotante, $f = 0{,}25$; y sea p el factor de mejora de la unidad de coma flotante, $p = 4$, ya que el tiempo se reduce a una cuarta parte. Tenemos que la mejora máxima sería:

$$S_{\text{máx}} = \frac{T^b_{CPU}}{T^p_{CPU}} \leqslant \frac{p}{1 + f(p-1)} = \frac{4}{1 + 0{,}25 \cdot 3} = \frac{4}{1{,}75} = \frac{16}{7} \approx 2{,}2857$$

   Por tanto, tenemos que $T^b_{CPU} \leqslant 2{,}2857 \cdot T^p_{CPU}$; o equivalentemente tenemos $T^p_{CPU} \geqslant 0{,}4375 \cdot T^b_{CPU}$; es decir, el tiempo de ejecución disminuiría como mucho un $(100 - 43{,}75)\,\% = 56{,}25\,\%$.

Ejercicio 4.1.10. 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 del 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.

4.1.1. Cuestiones

Cuestión 4.1.1. Indique cuál es la diferencia fundamental entre una arquitectura CC-NUMA y una arquitectura SMP.

Una arquitectura cc-NUMA es un tipo concreto de NUMA, y una SMP es un tipo concreto de UMA. La diferencia fundamental entre ambas arquitecturas es que en las NUMA la memoria no está unificada completamente, ya que cada procesador tiene su propio módulo local de memoria. Al igual que en el caso del UMA, en las NUMA se comparte el espacio de direcciones, pero hay un retardo si en vez de acceder a la memoria local se accede a la memoria de otro procesador.

Cuestión 4.1.2. ¿Cuándo diría que un computador es un multiprocesador y cuándo que es un multicomputador?

La diferencia entre ambos consiste en el mapa de memoria. En el caso de un multiprocesador, todos los procesadores comparten el mismo espacio de direcciones, es decir, la memoria es compartida. En cambio, en el caso de un multicomputador, cada procesador tiene su propio espacio de direcciones independiente y el cual no puede ser accedido por otro procesador; es decir, la memoria no es compartida.

Cuestión 4.1.3. ¿Un CC-NUMA escala más que un SMP? ¿Por qué?

Un CC-NUMA escala mejor que un SMP, ya que añadir un procesador nuevo en un SMP implica que un nuevo procesador accederá a la memoria compartida, provocando ahí más comflictos. En el caso del NUMA, como el nuevo procesador añadido tendrá su memoria local, sí es verdad que provocará conflictos cuando acceda a memorias de otros procesadores, pero no tantos como en el caso de un SMP, ya que estos accesos serán minoritarios.

Cuestión 4.1.4. 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.

Un PC con un procesador de 4 cores puede aprovechar los siguientes niveles de paralelismo implícito:

- Paralelismo de instrucción, ILP: cada core tiene una microarquitectura segmentada y superscalar, por lo que puede ejecutar varias instrucciones en paralelo.

- Paralelismo de datos: cada core tiene unidades funcionales SIMD, por lo que puede ejecutar varias operaciones en paralelo, como la suma de vectores, la multiplicación de matrices, ...

- Paralelismo de tareas: cada core es independiente, por lo que puede ejecutar tareas distintas en paralelo.

Cuestión 4.1.5. Si le dicen que un ordenador es de 20 GIPS ¿puede estar seguro que ejecutará cualquier programa de 20000 instrucciones en un microsegundo?

No. Si un programa que ejecuta 20000 instrucciones tarda 1 microsegundo, tendríamos efectivamente que el programa se ha ejecutado a una velocidad de 20 GIPS. Sin embargo, no podemos estar seguros de que cualquier programa de 20000 instrucciones tarde en ejecutarse un microsegundo, ya que depende de las características de las instrucciones que componen al programa:

- El número de instrucciones que constituyen un programa puede ser dinstinto del número de instrucciones que finalmente ejecuta el procesador (se trata por tanto de un número dinámico de instrucciones), ya que puede haber instruccioens de salto, bucles, ... que hacen que ciertas instrucciones del código se ejecuten más de una vez; y puede haber otras que no se ejecuten nunca.

- Por otro lado, el tipo de instrucciones que constituyen el programa y las dependencias entre ellas pueden variar los tiempos de ejecución que tardan en ejecutarse las instrucciones.

Cuestión 4.1.6. ¿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).

No, por diversos motivos:

- En primer lugar, estas se pueden implementar mediante una arquitectura MIMD, que es más flexible, por lo que es un caso concreto de algo que ya se puede hacer.

- Además, se tendría un gran cuello de botella, ya que aunque puedas ejecutar varias instrucciones en paralelo, tan solo puedes acceder a un dato a la vez por solo tener un cauce para estos, algo que provocaría que las instrucciones estuviesen esperando a que se accediese a los datos, realentizando entonces el proceso.

Cuestión 4.1.7. 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.

Razonado en la sección 1.3.6.

Cuestión 4.1.8. ¿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).

Sí. Sea el factor de mejora p y la probabilidad de no utilizar el recurso f. Entonces, la ley de Amdahl nos dice que:

$$S \leqslant \frac{p}{1 + f(p-1)}$$

Si $p \to \infty$, que representa que el recurso se mejora infinitamente, entonces:

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

Esto se debe a que aunque esa parte del sistema se mejore, si no se utiliza en la ejecución del programa, no se verá reflejado en la mejora de la velocidad. Por tanto, llegará un momento en el que la mejora de la velocidad se sature.

Cuestión 4.1.9. ¿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).

Tenemos que la cota, notada por $S_{\text{máx}}$, para la ganancia que establece la ley de Amdahl es:

$$S_{\text{máx}} = \frac{p}{1 + f(p-1)}$$

Para ver si crece a medida que aumenta el valor del factor de mejora (p aumenta), derivamos la expresión respecto de p:

$$S'_{\text{máx}} = \frac{1 + f(p-1) - pf}{(1 + f(p-1))^2} = \frac{1-f}{(1 + f(p-1))^2} \geqslant 0 \quad \forall f \in [0,1]$$

donde hemos afirmado que es positiva puesto que $f \in [0,1]$. Por tanto, tenemos que efectivamente dicha función es creciente a medida que aumenta p, por lo 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.

Cuestión 4.1.10. ¿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).

El tiempo que tardaría en el ordenador superescalar, fijado el número de operaciones N, sería:

$$T_{\text{superescalar}} = \frac{N \cdot T_{\text{ciclo superescalar}}}{IPC} = \frac{N \cdot T_{\text{ciclo superescalar}}}{4}$$

En el caso de un procesador vectorial, y suponiendo que cualquier instrucción puede ser empaquetada (algo que es complicado), el tiempo que tardaría en el ordenador vectorial sería:

$$T_{\text{vectorial}} = \frac{N \cdot T_{\text{ciclo vectorial}}}{IPC} = \frac{N \cdot T_{\text{ciclo vectorial}}}{8}$$

Por tanto, tenemos que:

$$\frac{T_{\text{superescalar}}}{T_{\text{vectorial}}} = \frac{N \cdot T_{\text{ciclo superescalar}}}{4} \cdot \frac{8}{N \cdot T_{\text{ciclo vectorial}}} = 2 \cdot \frac{T_{\text{ciclo superescalar}}}{T_{\text{ciclo vectorial}}} = 2 \cdot \frac{f_{\text{vectorial}}}{f_{\text{superescalar}}}$$

Por tanto, tan solo podemos establecer esta relación entre las frecuencias de reloj de ambos procesadores, ya que no se nos ha dado información sobre ellas. Suponiendo que fuesen iguales (algo que no tiene por qué ser así), entonces el procesador vectorial sería mejor. Para poder sacar más conclusiones, necesitaríamos más información.

Cuestión 4.1.11. 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 4.1.12. 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.

4.1.2. Ejercicios adicionales

Ejercicio 4.1.11. En el bucle siguiente, los arrays `a[]`, `b[]`, `c[]` y `d[]`, son números en coma flotante de 64 bits y $n = 2 \cdot 10^{10}$:

```
for (i = 0; i < n; i++)
    d[i] = a[i] + b[i] + c[i];
```

Si el programa se ejecuta en un procesador a 2 GHz que puede terminar dos operaciones en coma flotante por ciclo, ¿cuál es el tiempo mínimo que tardaría en ejecutarse? ¿Cuántos GFLOPS de velocidad pico tiene el procesador?

Para calcular el tiempo mínimo que tardaría en ejecutarse, calculamos de forma teórica el tiempo que se necesita para ejecutar las instrucciones descritas en el código. Este es el tiempo mínimo ya que estamos despreciando muchos factores por simplificar el estudio que hacen que el tiempo de ejecución aumente. Hacemos uso de la fórmula del tiempo de CPU:

$$T_{CPU} = NI_{float} \cdot CPI_{float} \cdot T_{ciclo} = \frac{NI_{float}}{IPC_{float} \cdot F} = \frac{2 \cdot 10^{10}}{2 \cdot (2 \cdot 10^{9})} = 5 \text{ segundos}$$

A continuación, calculamos el número de GFLOPS pico del procesador, haciendo uso de la fórmula:

$$GFLOPS = \frac{1}{CPI \cdot T_{ciclo} \cdot 10^9} = \frac{IPC \cdot F}{10^9} = \frac{2 \cdot (2 \cdot 10^9)}{10^9} = 4$$

Por lo que el procesador tiene 4 GFLOPS de velocidad pico.

Ejercicio 4.1.12. La empresa DataNimbus estima que debe adquirir un nuevo computador con una velocidad pico de 100 TFLOPS para alcanzar los niveles de tiempos de respuesta requeridos en su nueva generación de algoritmos para aplicaciones Big Data. Se ha decidido configurar la máquina a base de nodos HP ProLiant SL230s Gen8. Concretamente, cada uno de estos servidores tiene dos procesadores Sandy Bridge Intel© Xeon© E5-2670 a 2,60 GHz con 8 núcleos/procesador:

1. ¿Cuántos nodos (servidores HP ProLiant SL230s) se necesitan para configurar la máquina de 100 TFLOPS?

2. Clasifique el nuevo servidor que se pretende adquirir, 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.

3. ¿Cuál es el número máximo de operaciones de coma flotante por ciclo de cada core del Intel Xeon E5-2670?

Nota: El Yellowstone National enter for Atmospheric Research, que utiliza el mismo procesador que quermos montar, tiene una velocidad pico de 1503590 GFLOPS y contiene 72288 núcleos.

1. Como sabemos, por la información que nos proporcina la nota, la velocidad pico de un núcleo para operaciones en coma flotante es:

$$\frac{1503590}{72288} = 20{,}8 \text{ GFLOPS/núcleo}$$

   Dado que hay que alcanzar 100 TFLOPS $= 100 \cdot 10^3$ GFLOPS, el número de núcleos que necesitamos es:

$$\frac{100 \cdot 10^3}{20{,}8} = 4807{,}7 \text{ núcleos}$$

   Es decir, 4808 núcleos.

   Como el servidor HP ProLiant SL230s que se utiliza en cada nodo tiene 16 núcleos (2 microprocesadores con 8 núcleos cada uno), el número de nodos necesarios sería:

$$\frac{4808}{16} = 300{,}5 \longrightarrow 301 \text{ nodos}$$

2. Desde el punto de vista de la taxonomía Flynn, es un computador MIMD. Cada uno de los microprocesadores que hay en el nodo tiene 8 núcleos que comparten la memoria local del microprocesdor y por tanto son multiprocesadores UMA. Estos dos microprocesadores se interconectan en el nodo constituyendo un procesador NUMA dado que cada uno de ellos tiene su memoria principal local. Los nodos están interconectados a través de una red y configuran un computador NORMA o cluster.

3. Para este último punto, simplemente tenemos que observar la fórmula de los GFLOPS:

$$GFLOPS = \frac{\text{Operaciones coma flotante}}{T_{CPU} \cdot 10^9} = \frac{\text{Operaciones coma flotante}}{\text{Ciclos de programa} \cdot T_{ciclo} \cdot 10^9}$$

De donde despejamos las operaciones de coma flotante (Ops) entre los ciclos de programa (cdp):

$$\frac{Ops}{cdp} = GFLOPS \cdot 10^9 \cdot T_{ciclo} = \frac{GFLOPS \cdot 10^9}{F} = \frac{20{,}8 \cdot \cancel{10^9}}{2{,}6 \cdot \cancel{10^9}} = 8$$

Por tanto, el número máximo de operaciones en coma flotante que puede terminar el núcleo por ciclo es 8.

Ejercicio 4.1.13. Un procesador superescalar de 64 bits a 1 GHz capaz de finalizar tres instrucciones por ciclo ejecuta el programa que se indica a continuación:

```
1  start:  ld      f0, a           // f0 = a
2          add     r8, r0, r2      // r8 = r2 (r0 = 0)
3          addi    r6, r8, #2048   // r6 = r8 + 2048
4          add     r12, r0, r4     // r12 = r4 (r0 = 0)
5  loop:   ld      f2, 0(r8)       // f2 = m(r8)
6          multd   f2, f0, f2      // f2 = f0 * f2
7          ld      f4, 0(r12)      // f4 = m(r12)
8          addd    f4, f2, f4      // f4 = f2 + f4
9          sd      0(r12), f4      // m(r12) = f4
10         addi    r8, r8, #8      // r8 = r8 + 8
11         addi    r12, r12, #8    // r12 = r12 + 8
12         sub     r16, r6, r8     // r16 = r6 - r8
13         bnez    r16, loop       // Si r16 != 0, salta
```

En el programa, `a` es un número real, `r0` es un registro que siempre está a cero, `r2` contiene la dirección a partir de la cual empieza un array, `X`, de números reales de 64 bits, y `r4` contiene la dirección a partir de la que empieza otro array también de números reales de 64 bits. ¿Qué hace el programa? ¿Cuál es el tiempo mínimo que tardaría en ejecutarse?

4.2. Programación paralela

Ejercicio 4.2.1. 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?

Por el enunciado, tenemos que $T_P = 40\,\text{s}$ y $T_O(p) = 0\,\text{s}$. Por tanto:

$$T_P(p) = T_C(p) + \cancelto{0}{T_O(p)} = T_C(p)$$

1. Gracias al enunciado, deducimos que durante $0{,}2T_P$ se usan 4 procesadores, durante $0{,}6T_P$ se usan 3 y durante $0{,}2T_P$ se usa un único procesador. Por tanto, si ejecutásemos el programa en un único procesador, tendríamos que ejecutar de forma secuencial el código correspondiente a los 4 procesadores, luego a los tres procesadores y por último la parte que no es paralelizable. Por tanto, tenemos que:

$$T_S = 4 \cdot 0{,}2T_P + 3 \cdot 0{,}6T_P + 0{,}2T_P = 2{,}8T_P = 2{,}8 \cdot 40\,\text{s} = 112\,\text{s}$$

2. Calculamos la ganancia:

$$S(4) = \frac{T_S}{T_P(4)} = \frac{2{,}8\cancel{T_P}}{\cancel{T_P}} = 2{,}8$$

Observación. Buscamos ahora comprobar si la ganancia obtenida es correcta. Suponiendo que podemos paralelizar todo el programa, el tiempo paralelo será $T_P(p) = {}^{T_S}/_{p}$. Por otro lado, si no podemos paralelizar nada, el tiempo paralelo será $T_P(p) = T_S$. Por tanto, tenemos que ${}^{T_S}/_{p} \leqslant T_P(p) \leqslant T_S$. Por tanto:

$$1 = \frac{T_S}{T_S} \leqslant \frac{T_S}{T_P(p)} = S(p) \leqslant \frac{T_S}{{}^{T_S}/_{p}} = p$$

Al ser $1 < 2{,}8 < 4$, podemos intuir que hemos calculado bien la solución.

3. Calculamos la eficiencia:

$$E(p) = \frac{\text{Prestaciones}(p)}{p \cdot \text{Prestaciones}(1)} = \frac{S(p)}{p} = \frac{2{,}8}{4} = 0{,}7$$

Observación. Buscamos ahora comprobar si la eficiencia obtenida es correcta. Como $1 \leqslant S(p) \leqslant p$, tenemos que:

$$\frac{1}{p} \leqslant \frac{S(p)}{p} = E(p) \leqslant 1$$

Al ser $0{,}25 = 1/4 \leqslant 0{,}7 \leqslant 1$, podemos intuir que hemos calculado bien la solución.

Ejercicio 4.2.2. Un programa tarda 40 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?

Del enunciado, tenemos que $T_S^{P_1} = 40\,\text{s}$, $T_S^{P_2} = 30\,\text{s}$ y $T_O(p) = 0$. Por tanto:

$$T_P(p) = T_C(p) + \cancelto{0}{T_O(p)} = T_C(p)$$

1. Si a los dos le asignamos la mitad del trabajo, tendremos que:

$$T_S^{P_1}\left(\frac{1}{2}\right) = \frac{1}{2} \cdot 20\,\text{s} = 10\,\text{s} \qquad T_S^{P_2}\left(\frac{1}{2}\right) = \frac{1}{2} \cdot 30\,\text{s} = 15\,\text{s}$$

   Por tanto:

$$T_P^{P_1,P_2}\left(\frac{1}{2}, \frac{1}{2}\right) = \text{máx}\left\{T_S^{P_1}\left(\frac{1}{2}\right), T_S^{P_2}\left(\frac{1}{2}\right)\right\} = \text{máx}\{10\,\text{s}, 15\,\text{s}\} = 15\,\text{s}$$

   Como inconveniente, tenemos que el procesador P_1 está ocioso durante 5 segundos, por lo que no es la mejor solución.

2. Repartimos el trabajo de forma que a P_1 le asignamos una fracción de trabajo de x, por lo que a P_2 le tendremos que asignar una carga de $1-x$. Para obtener los mejores tiempos, imponemos que:

$$T_S^{P_1}(x) = T_S^{P_2}(1-x) \iff x \cdot 20\,\text{s} = (1-x) \cdot 30\,\text{s} \iff 2x = 3 - 3x \iff x = \frac{3}{5}$$

   Por tanto, tenemos que $T_S^{P_1}(3/5) = T_S^{P_2}(2/5) = 3/5 \cdot 20\,\text{s} = 12\,\text{s}$. Por tanto, el tiempo de ejecución en paralelo será de:

$$T_P^{P_1,P_2}(3/5, 2/5) = \text{máx}\left\{T_S^{P_1}(3/5), T_S^{P_2}(2/5)\right\} = \text{máx}\{12\,\text{s}, 12\,\text{s}\} = 12\,\text{s}$$

   Como vemos, al realizar una distribución de carga equilibrada obtenemos un tiempo de ejecución menor.

Ejercicio 4.2.3. ¿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)?

En primer lugar, cabe destacar que vamos a suponer que el tiempo de sobrecarga es despreciable, ya que si no no podríamos dar ninguna solución. Sabiendo esto, tenemos que el tiempo secuencial sería:

$$T_S = 1 \cdot 50\,\text{ns} + 8 \cdot 50\,\text{ns} = 9 \cdot 50\,\text{ns}$$

El tiempo secuencial que tardaría en ejecutarse la parte paralelizada (código paralelo) sería $8 \cdot 50$ ns. por tanto, tenemos que la fracción del código que es paralelizable es:

$$\frac{8 \cdot 50\,\text{ns}}{9 \cdot 50\,\text{ns}} = \frac{8}{9}$$

Por tanto, la fracción de código paralelo es de $^8/_9$.

Ejercicio 4.2.4. 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?

El máximo valor de ganancia que podemos obtener resulta cuando el tiempo de sobrecarga es despreciable:

$$T_P(p) = T_C(p) + \cancelto{0}{T_O(p)}$$

Una vez sabiendo esto, podemos aplicar la Ley de Amdahl. Dado que un 25 % del programa no puede paralelizarse, tenemos que $f = {}^1/_4$. Por tanto, usando dicha Ley tenemos:

$$S(p) \leqslant \frac{1}{f + \dfrac{1-f}{p}} = \frac{1}{\dfrac{1}{4} + \dfrac{3}{4p}} = \frac{4p}{p+3}$$

Esta la mayor ganancia que podemos obtener al paralelizarlo en p procesadores. Tomando límite con $p \to \infty$, tenemos:

$$S_{\text{máx}} = \lim_{p\to\infty} S(p) = \lim_{p\to\infty} \frac{4p}{p+3} = 4$$

Finalmente, nos preguntamos por el número p de procesadores necesarios para conseguir ganancias mayores o iguales que 2:

$$S(p) = \frac{4p}{p+3} \geqslant 2 \iff 4p \geqslant 2p + 6 \iff 2p \geqslant 6 \iff p \geqslant 3$$

Por tanto, necesitamos como mínimo 3 procesadores para obtener una ganancia mayor o igual a 2.

Ejercicio 4.2.5. En la Figura 4.1, 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.

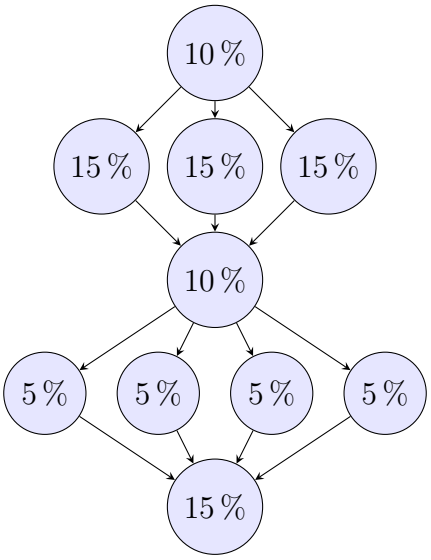


Figura 4.1: Grafo de tareas del Ejercicio 4.2.5.

Del enunciado, tenemos que $T_S = 60\,\text{s}$ y $T_O(p) = 0$. Por tanto:

$$T_P(p) = T_C(p) + \cancelto{0}{T_O(p)} = T_C(p)$$

Numeramos las tareas (los nodos) de arriba a abajo y de izquierda a derecha. Observando el gráfico, vemos que el número máximo de nodos en el mismo nivel es 4, luego el número máximo de procesadores (cores) que demos usar será de 4.

1. En este caso, usaremos 4 procesadores, por lo que estamos en el caso ideal como hemos mencionado. Tenemos que:

   - La tarea 1 tendrá un tiempo de ejecución de $0{,}1T_S$.
   - Las tareas 2, 3 y 4 tendrán tiempos de ejecución $0{,}15T_S$.
   - La tarea 5 tendrá un tiempo de ejecución de $0{,}1T_S$.
   - Las tareas 6, 7, 8 y 9 tendrán tiempos de ejecución $0{,}05T_S$.
   - La tarea 10 tendrá un tiempo de ejecución de $0{,}15T_S$.

Además, tenemos que no se pueden ejecutar las tareas de nivel $i+1$ si no se han terminado las del nivel i, ya que las del siguiente nivel necesitan de los datos generador por el nivel anterior. Por tanto, tenemos que:

$$T_P(4) = 0{,}1T_S + 0{,}15T_S + 0{,}1T_S + 0{,}05T_S + 0{,}15T_S = 0{,}55T_S = 0{,}55 \cdot 60\,\text{s} = 33\,\text{s}$$

Calculamos la ganancia:

$$S(4) = \frac{T_S}{T_P(4)} = \frac{\cancel{T_S}}{0{,}55\cancel{T_S}} = \frac{1}{0{,}55} = \frac{20}{11} \approx 1{,}82$$

Este resultado tiene sentido, ya que $1 \leqslant S(4) \leqslant p = 4$, por lo que la ganacia calculada está dentro del valor mínimo y máximo de esta.

2. En este caso, usaremos 2 procesadores:

   - La tarea 1 tendrá un tiempo de ejecución de $0{,}1T_S$, al igual que en el caso de paradores.
   - Las tareas 2, 3 y 4 no pueden ejecutarse en paralelo todas a la vez. Tendremos que ejecutar en primer lugar la 2 y la 3 ($0{,}15T_S$) y después la 4 ($0{,}15T_S$).
   - La tarea 5 tendrá un tiempo de ejecución de $0{,}10T_S$.
   - Las tareas 6, 7, 8 y 9 no pueden ejecutarse en paralelo todas a la vez. Tendremos que ejecutar en primer lugar la 6 y la 7 ($0{,}05T_S$) y después la 8 y 9 ($0{,}05T_S$).
   - La tarea 10 tendrá un tiempo de ejecución de $0{,}15T_S$.

   Por tanto, tenemos que:

$$T_P(2) = 0{,}1T_S + 2 \cdot 0{,}15T_S + 0{,}1T_S + 2 \cdot 0{,}05T_S + 0{,}15T_S = 0{,}75 \cdot T_S = 45\,\text{s}$$

   Como vemos, el tiempo es mayor debido a que los niveles 2 y 4 no los podemos paralelizar de forma completa. La ganancia queda:

$$S(2) = \frac{T_S}{T_P(2)} = \frac{\cancel{T_S}}{0{,}75 \cdot \cancel{T_S}} = \frac{1}{0{,}75} = \frac{4}{3} \approx 1{,}333 < 2$$

Ejercicio 4.2.6. 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 4.2. La ejecución de estas tareas en un procesador supone un tiempo de 2 s. 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?

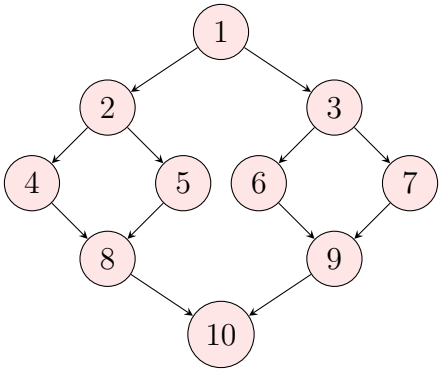


Figura 4.2: Grafo de tareas del Ejercicio 4.2.6

Según el grafo 4.2, como máximo tenemos un total de 4 tareas independientes a ejecutar. Por tanto, a pesar de disponer de 8 procesadores, sólo usaremos 4.

1. Dados los porcentajes del enunciado, calculamos el tiempo de ejecución paralelo a partir del secuencial.

$$\begin{aligned} T_P(4) &= 0{,}1T_S + 0{,}15T_S + 0{,}09T_S + \max\{0{,}08, 0{,}1\}T_S + 0{,}06T_S \\ &= 0{,}5T_S = 0{,}5 \cdot 2\,\mathrm{s} = 1\,\mathrm{s} \end{aligned}$$

   Además, como 4 es el número máximo de procesadores que emplearemos, tenemos que $T_P(n) = T_P(4)$ para todo $n \geqslant 4$, y en particular para $n = 8$.

2. La ganancia respecto a la secuencial es:

$$S(4) = \frac{T_S}{T_P(4)} = \frac{2\,\mathrm{s}}{1\,\mathrm{s}} = 2$$

   Al igual que ocurría con el tiempo de ejecución en paralelo, para la ganancia tenemos que $S(n) = S(4)$ para todo $n \geqslant 4$, y en particular para $n = 8$.

Ejercicio 4.2.7. 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}
```

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 $k_1 p$ (k_1 constante), los procesos se comunican y se sincronizan, lo que supone un tiempo $k_2 p$ (k_2 constante); $k_1 p + k_2 p$ 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?

1. Para obtener una expresión del tiempo de ejecución en paralelo del trozo de código en p procesadores, necesitamos:

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

Sabemos ya que $T_O(p) = p(k_1 + k_2)$, pero no tenemos de forma directa $T_C(p)$. Tenemos una sección de código no paralelizable (los cálculos que se realizan antes del bucle y los que se realizan después), que tiene un costo $t_1 + t_2$. Además, contamos con w iteraciones de un código paralelizable, cada una de ellas con un costo t_i. Por tanto, este código tardaría $w \cdot t_i$ en un solo procesador. Sin embargo, al disponer de p procesadores, el número de iteraciones en cada procesador se vería reducido a $\left\lceil \frac{w}{p} \right\rceil$ (siendo $\lceil x \rceil$ la función techo de x). El redondeo hacia arriba se debe a que no podemos tener un número decimal de iteraciones, por lo que en algunos procesadores se realizará una iteración más que en otros. Por tanto, el tiempo de cómputo en paralelo será:

$$T_C(p, w) = t_1 + t_2 + \left\lceil \frac{w}{p} \right\rceil \cdot t_i$$

En consecuencia:

$$T_P(p, w) = t_1 + t_2 + \left\lceil \frac{w}{p} \right\rceil \cdot t_i + p(k_1 + k_2)$$

2. Para el cálculo de la ganancia en velocidad, debemos primero calcular el tiempo de ejecución secuencial del programa. Por un razonamiento análogo, tenemos que ejecutar la parte del código no paralelizable (con un tiempo de $t_1 + t_2$) y las w iteraciones del bucle, cada una de tiempo t_i:

$$T_S(w) = t_1 + t_2 + w \cdot t_i$$

De donde:

$$S(p, w) = \frac{T_S(w)}{T_P(p, w)} = \frac{t_1 + t_2 + w \cdot t_i}{t_1 + t_2 + \left\lceil \frac{w}{p} \right\rceil \cdot t_i + p(k_1 + k_2)}$$

3. Para terminar con el ejercicio, vamos ahora a estudiar si $T_P(p)$ tiene mínimo o no. En primer lugar, tenemos que $T_P(p)$ es una función no lineal, ya que en uno de los sumanos depende inversamente de p de forma no lineal. Lo demostraremos además argumentando que tiene un mínimo. Para ello, es decerario descartar el redondeo, ya que la buscamos obtener una función derivable cuya monotonía sea fácil de estudiar. La influencia de la función *techo* se mencionará más adelante. Sea $\widetilde{T}_P(p)$ la función que no tiene en cuenta el redondeo:

$$\widetilde{T}_P(p) = t_1 + t_2 + \frac{w}{p} t_i + p(k_1 + k_2)$$

Tenemos que es diferenciable, con:

$$\widetilde{T}'_P(p) = -\frac{w}{p^2}t_i + k_1 + k_2$$

Calculamos sus puntos críticos:

$$\widetilde{T}'_P(p) = 0 \iff \frac{-w}{p^2}t_i + (k_1 + k_2) = 0 \iff \frac{w}{p^2}t_i = k_1 + k_2 \iff p = \pm\sqrt{\frac{w \cdot t_i}{k_1 + k_2}}$$

Descartamos la solución negativa por no tener sentido en este caso. Comprobamos ahora si este punto crítico es un mínimo o un máximo, calculando la segunda derivada de $\widetilde{T}_P(p)$:

$$\widetilde{T}''_P(p) = \frac{2pwt_i}{p^4} = \frac{2wt_i}{p^3} > 0$$

Luego se trataba de un mínimo relativo. Aunque es cierto que no tenemos de forma directa con cuántos procesadores se alcanza el mínimo (ya que estos son enteros), evaluando en los dos enteros más próximos a $\sqrt{\frac{w \cdot t_i}{k_1+k_2}}$ podremos determinar cuál de ellos es el mínimo. Además, estos valores se podrán incluso evaluar en $T_P(p)$ en vez de en $\widetilde{T}_P(p)$ para obtener resultados más precisos. En cualquier caso, tenemos que T_p tiene un mínimo, por lo que no depende linealmente de p.

Ejercicio 4.2.8. 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.

1. Para $n = 16$, tendríamos el siguiente grafo de tareas:

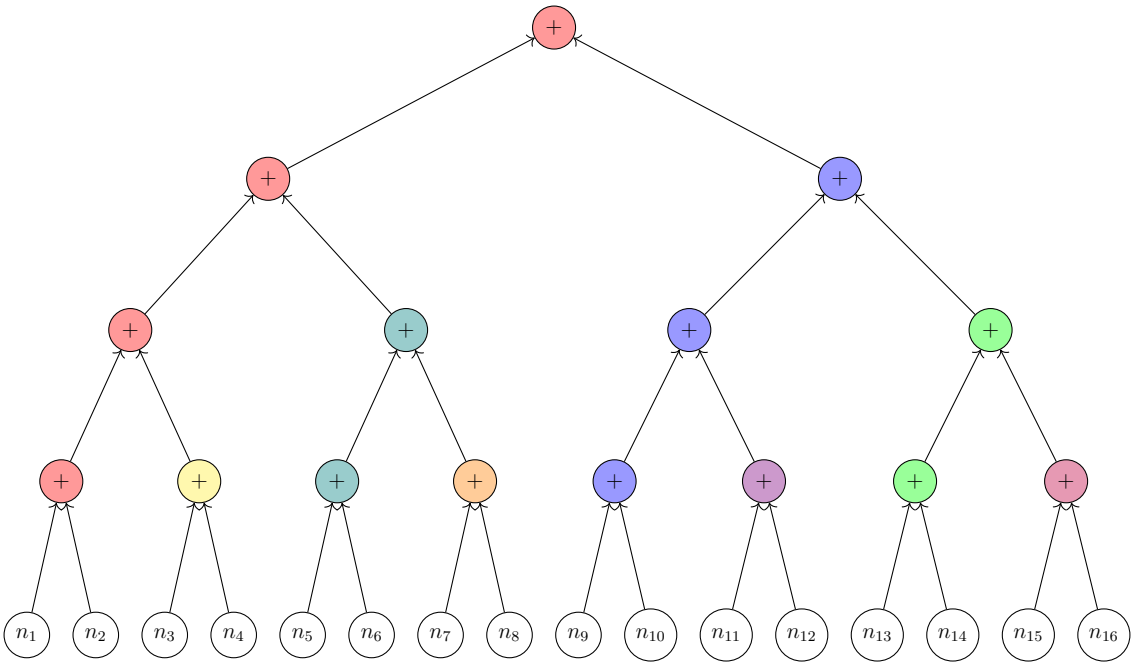


Figura 4.3: Grafo de tareas del Ejercicio 4.2.8.

Como podemos ver, se usan 8 procesos distintos (cada uno de un color). Notaremos las tareas de forma creciente de izquierda a derecha y de arriba a abajo. Tenemos entonces que la asignación de tareas a procesos sería:

- Proceso 1: Tareas 1, 2, 4, 8.
- Proceso 2: Tareas 3, 6, 12.
- Proceso 3: Tareas 5, 10.
- Proceso 4: Tareas 7, 14.
- Proceso 5: Tarea 9.
- Proceso 6: Tarea 11.
- Proceso 7: Tarea 13.
- Proceso 8: Tarea 15.

Ejercicio 4.2.9. 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):

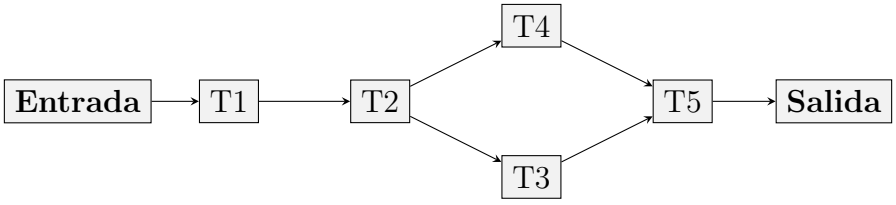


Figura 4.4: Segmentación del Ejercicio 4.2.9.

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.

1. Contamos con 4 etapas en el cauce, ya que el bloque 3 y 4 se ejecutan en paralelo al poder asignar cada tarea a un procesador. Veamos en qué momento entra y sale de cada etapa, para ver cómo evoluciona el cauce:

Bloque	Entra 1	Sale 1	Entra 2	Sale 2	Entra 3	Sale 3	Entra 4	Sale 4
B1	0	t	t	$2t$	$2t$	$3{,}5t$	$3.5t$	$4.5t$
B2	t	$2t$	$2t$	$3t$	$3{,}5t$	$5t$	$5t$	$6t$
B3	$2t$	$3t$	$3t$	$4t$	$5t$	$6{,}5t$	$6{,}5t$	$7{,}5t$
B4	$3t$	$4t$	$4t$	$5t$	$6{,}5t$	$8t$	$8t$	$9t$

Desde que sale el bloque 1 hasta el bloque 2, transcurren $1{,}5t$, al igual que desde 2 hasta 3 y desde 3 hasta 4. En un cauce, el tiempo de ejecución depende del TLI (Tiempo de Latencia Inicial) y del TEML (Tiempo de la Etapa Más Lenta). En este caso, el TLI es $4{,}5t$, ya que es el tiempo que tarda en llenarse el cauce. El TEML es $1{,}5t$, que es la tercera etapa. Por tanto, el tiempo de ejecución en paralelo es:

$$\begin{aligned} T_P(5) &= \underbrace{4{,}5t}_{B1} + \underbrace{1{,}5t}_{B2} + \underbrace{1{,}5t}_{B3} + \cdots + \underbrace{1{,}5t}_{Bn} \\ &= \text{TLI} + (n-1)\text{TEML} = 4{,}5t + 1{,}5t(n-1) = 3t + 1{,}5tn \end{aligned}$$

El tiempo secuencial tenemos que es:

$$T_S(n) = n \cdot (t + t + 2 \cdot 1{,}5t + t) = 6nt$$

Por tanto, la ganancia tenemos que es:

$$S(5,n) = \frac{T_S(n)}{T_P(5,n)} = \frac{6nt}{3t + 1{,}5tn} = \frac{6n}{3 + 1{,}5n}$$

La ganancia máxima se obtiene cuando $n \to \infty$, que es cuando el cauce estará lleno:

$$\lim_{n\to\infty} S(5,n) = \lim_{n\to\infty} \frac{6n}{3 + 1{,}5n} = \frac{6}{1{,}5} = 4$$

2. En este caso, la asignación de procesadores no es trivial, pues no podemos asignar cada tarea a un procesador. Las tareas 3 y 4 tendrán un único procesador por ser las más lentas. Así, tenemos que:

   - La tarea 1 al procesador 1.
   - La tarea 2 al procesador 1.
   - La tarea 3 al procesador 2.
   - La tarea 4 al procesador 3.
   - La tarea 5 al procesador 4.

   Se trata de una distribución en 3 etapas para segmentación, donde la primera etapa se corresponde con las dos primeras tareas, la segunda con la tercera y la cuarta, y la tercera con la quinta. En este caso, tenemos que el TEML es de $2t$, y que el TLI es de $(2 + 1{,}5 + 1)t = 4{,}5t$. Por tanto, el tiempo de ejecución en paralelo es:

   $$T_P(4) = 4{,}5t + 2t(n-1) = 4{,}5t + 2tn - 2t = 2tn + 2{,}5t$$

   Por tanto, la ganancia en prestaciones es:

   $$S(4,n) = \frac{6nt}{2tn + 2{,}5t} = \frac{6n}{2n + 2{,}5}$$

   La ganancia máxima se obtiene cuando $n \to \infty$, que es cuando el cauce estará lleno:

   $$\lim_{n\to\infty} S(4,n) = \lim_{n\to\infty} \frac{6n}{2n + 2{,}5} = \frac{6}{2} = 3$$

Ejercicio 4.2.10. Se quiere implementar un programa paralelo para un multicomputador 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:

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

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

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 4.2.11.

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 programanción 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 4.2.12. Escribir una versión paralela del programa secuencial del ejercicio 4.2.11 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,group)`, 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 4.2.13.

1. Escribir una versión paralela del programa paralelo del ejercicio 4.2.12 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.
   - `sendtype`: 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 4.2.14. Escribir una versión paralela del programa secuencial del ejercicio 4.2.11 para un multiprocesador usando el estilo de programación paralela de variables compartidas; en particular, use OpenMP (escribir primero la versión secuencial).

4.2.1. Cuestiones

Cuestión 4.2.1. Indique las diferencias entre OpenMP y MPI.

OpenMP es una API de directivas y funciones, mientras que MPI es una API de funciones. Ambas son herramientas que permiten desarrollar paralelismo. Sin embargo, como hemos ya mencionado en la parte teórica, la abstracción en la que nos sitúa OpenMP es mayor a al que nos provee MPI: mientras que en MPI hemos de preocuparnos de muchos detalles a la hora de implementar el paralelismo, en OpenMP le decimos a la herramienta que realice cierta tarea y ya es el compilador quien se encarga de resolver estos detalles.

Cuestión 4.2.2. Ventajas e inconvenientes de una asignación estática de tareas a procesos/threads frente a una asignación dinámica.

La principal desventaja de la asignación dinámica frente a la estática es la necesidad de calcular en cada momento qué asignación de trabajo es la mejor a desarrollar en cada momento: tener en cuenta qué nodos de trabajo se encuentran ocupados y cuales no y dividir el trabajo conforme a ello. Esto puede llevar a una sobrecarga de trabajo en el cálculo de la asignación de tareas, lo que puede llevar a una pérdida de tiempo en la ejecución del programa. Como ventajas, podemos destacar dos:

- En caso de una asignación compleja (podemos realizar una asignación estática, pero obtenerla es muy dificultosa), podemos simplemente realizar una asignación dinámica, ahorrando al programador la tarea del cálculo de los trabajos de cada nodo.
- Cuando no se conoce el número de tareas que se ejecutarán (por ejemplo, porque depende de un parámetro que se conoce en tiempo de ejecución), no es posible llevar a cabo una asociación estática.

Cuestión 4.2.3. ¿Qué se entiende por escalabilidad lineal y por escalabilidad superlineal? Indique las causas por las que se puede obtener una escalabilidad superlineal.

Decimos que un programa que resuelve una aplicación de forma paralela es escalable linealmente cuando su ganancia de velocidad en comparación con el tiempo secuencial resulta en el número de procesadores. Es decir, decimos que una aplicación paralela escala linealmente si:

$$S(p) = p$$

Por otra parte, decimos que escala de forma superlineal si:

$$S(p) > p$$

Un ejemplo de escalabilidad superlineal es el resultado de aplicar una búsqueda lineal paralela a cierta instancia, de forma que en pocas iteraciones encuentre el elemento buscado gracias a partir el vector para su paralelización, tal y como comentamos en la Sección 2.3.2.

Cuestión 4.2.4. Enuncie la ley de Amdahl en el contexto de procesamiento paralelo.

Dado un código secuencial cuyo tiempo de ejecución suponemos constante en T_S segundos, suponemos que tenemos una fracción f de código no paralelizable y que usamos p procesadores para la fracción de código que sí es paralelizable. En dicho caso, obtendremos una ganancia con p procesadores de a lo sumo:

$$S(p) \leqslant \frac{p}{1 + f(p-1)}$$

Cuestión 4.2.5. 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?

Supuesto que el tiempo de ejecución de un código paralelo es constante fijado un número p de procesadores en $T_P(p)$, procedemos a calcular el tiempo de ejecución secuencial de dicho programa en función del tiempo de ejecución en paralelo, sabiendo que existe una fracción f de código no paralelizable:

$$T_S = f \cdot T_P(p) + p(1-f) \cdot T_P(p)$$

Ya que la fracción de código no paralelizable será la misma. Además, si antes tardábamos $(1-f)T_P(p)$ en ejecutar la parte paralelizable entre p procesadores, si asumimos un reparto equitativo, en un procesador tendremos que hacer p veces este tiempo. De esta forma, calculamos la ganancia:

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

Con esta ley, Gustafson nos indica que la ganancia depende de forma lineal del número p de procesadores, con una pendiente de $1-f$, lo que nos indica que, a mayor sea $1-f$ (luego, menor sea f), es decir, la parte paralelizable; mayor será la ganancia de velocidad al paralelizar.

Cuestión 4.2.6. Deduzca la expresión que caracteriza a la ley de Amdahl. Defina claramente el punto de partida y todas las etiquetas que utilice.

Supuesto que el tiempo de ejecución de un programa en un procesador es constante en T_S segundos y supuesto que pasamos a una versión del programa con código paralelo con p procesadores y una fracción de código no paralelizable f. Bajo estas hipótesis, calculamos el tiempo de ejecución en paralelo a partir del secuencial:

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

Debido a que tenemos una fracción f no paralelizable y que la parte que tardaba $(1-f) \cdot T_S$ se reparte de forma equitativa entre p procesadores. De esta forma, calculamos la ganancia:

$$S(p) = \frac{T_S}{f \cdot T_S + \left(\frac{1-f}{p} \cdot T_S\right) + T_O(p)} = \frac{1}{f + \frac{1-f}{p} + \frac{T_O(p)}{T_S}}$$

Con $\frac{T_O(p)}{T_S} > 0$, luego obtendremos la mayor ganancia cuando $T_O(p)$ sea despreciable, por lo que cualquier ganancia estará limitada:

$$S(p) \leqslant \frac{1}{f + \frac{1-f}{p}} = \frac{p}{1 + f(p-1)}$$

4.3. Arquitecturas TLP

Ejercicio 4.3.1. En un multiprocesador SMP con 4 procesadores o nodos (N0-N3) basado en un bus, que implementa el protocolo MESI para mantener la coherencia, supongamos una dirección de memoria incluida en un bloque que no se encuentra en ninguna caché. Indique los estados de este bloque en las cachés y las acciones que se producen en el sistema ante la siguiente secuencia de eventos para dicha dirección:

1. Lectura generada por el procesador 1.
2. Lectura generada por el procesador 2.
3. Escritura generada por el procesador 1.
4. Escritura generada por el procesador 2.
5. Escritura generada por el procesador 3.

Ejercicio 4.3.2. Para un multiprocesador de memoria distribuida con 8 nodos se quiere implementar un protocolo para mantenimiento de coherencia basado en directorios. Suponiendo que se necesitan un bit de estado para un bloque en el directorio de memoria principal y que el tamaño de una línea de caché es de 64 bytes, calcular el porcentaje del tamaño de memoria principal que supone el tamaño del directorio de vector de bits completo.

Ejercicio 4.3.3. Suponga que en un CC-NUMA de red estática de 4 nodos (N0-N3) se implementa un protocolo MSI basado en directorios sin difusión con dos estados en el directorio (válido e inválido). Cada nodo tiene 8 GBytes de memoria y una línea de caché supone 64 Bytes. Considere que el directorio utiliza vector de bits completo.

1. Calcule el tamaño del directorio de uno nodo en bytes.
2. Indique cual sería el contenido del directorio, las transiciones de estados (en caché y en el directorio) y la secuencia de paquetes generados por el protocolo de coherencia en los siguientes accesos sobre una dirección D que se encuentra en la memoria del nodo 3 (inicialmente D no está en ninguna caché):

   a) Lectura generada por el procesador del nodo 1.

   b) Escritura generada por el procesador del nodo 1.

   c) Lectura generada por el procesador del nodo 2.

   d) Lectura generada por el procesador del nodo 3.

   e) Escritura generada por el procesador del nodo 0.

Ejercicio 4.3.4. Supongamos que se va a ejecutar en paralelo el siguiente código, donde `x` e `y` son variables compartidas (inicialmente `x=y=0`):

```
1 x=1;
2 x=2;
3 print y;
```

(a) P1.

```
1 y=1;
2 y=2;
3 print x;
```

(b) P2.

¿Qué resultados se pueden imprimir si (considere que el compilador no altera el código):

1. Se ejecutan P1 y P2 en un multiprocesador con consistencia secuencial.

2. Se ejecutan en un multiprocesador basado en un bus que garantiza todos los órdenes excepto el orden `W→R`. Esto es debido a que los procesadores tienen buffer de escritura, permitiendo el procesador que las lecturas en el código que ejecuta adelanten a las escrituras que tiene su buffer. Obsérvese que hay varios resultados posibles.

Ejercicio 4.3.5. Supongamos que se va a ejecutar en paralelo el siguiente código (inicialmente `x=y=0`):

```
1 x=30;
2 y=40;
3 flag=1;
```

(a) P1.

```
1 while (flag==0) {};
2 r1=x;
3 r2=y;
```

(b) P2.

¿Qué datos puede obtener P2 en los registros `r1` y `r2` si (considere que el compilador no altera el código):

1. Se ejecutan P1 y P2 en un multiprocesador con consistencia secuencial.

2. Se ejecutan en un multiprocesador con un modelo de consistencia que relaja todos los órdenes en los accesos a memoria. Razone su respuesta.

Ejercicio 4.3.6. Se quiere implementar un cerrojo simple en un multiprocesador SMP basado en procesadores de la línea x86 de Intel, en particular, procesadores Intel Core.

1. Teniendo en cuenta el modelo de consistencia de memoria que ofrece el hardware de este multiprocesador ¿podríamos implementar la función de liberación del cerrojo simple usando `mov k, 0`, siendo `k` la variable cerrojo? Razone su respuesta.

2. ¿Cómo se debería implementar la función de liberación de un cerrojo simple si se usan procesadores con la arquitectura ARMv7? Razone su respuesta.

Ejercicio 4.3.7. Se ha ejecutado el siguiente código en un multiprocesador con un modelo de consistencia que no garantiza ni W→R ni W→W (garantiza el resto de órdenes):

```
1 sump = 0;
2 for (i=ithread ; i<8 ; i=i+nthread) {
3     sump = sump + a[i];
4 }
5 while (Fetch_&_Or(k,1)==1) {};
6 sum = sum + sump;
7 k=0;
```

Conteste a las siguientes preguntas (considere que el compilador no altera el código):

1. Indique qué se puede obtener en `sum` si se suma la lista `a={1,2,3,4,5,6,7,8}`. `k` y `sum` son variables compartidas que están inicialmente a 0 (el resto de variables son privadas), `nthread = 3`, `ithread` es el identificador del thread en el grupo (0,1,2). Si hay varios posibles resultados, se tienen que dar todos ellos. Justifique su respuesta.

2. ¿Qué resultados se pueden obtener si lo único que no garantiza el modelo de consistencia es el orden W→R? Justifique su respuesta.

Ejercicio 4.3.8. ¿Qué ocurre si en el segundo código para implementar barreras visto en clase eliminamos la variable local, `cont_local`, sustituyéndola en los puntos del código donde aparece por el contador compartido asociado a la barrera `bar[id].cont`?

Ejercicio 4.3.9. Suponiendo que la arquitectura dispone de instrucciones `Fetch&Add`, simplifique el segundo código para barreras visto en clase.

Ejercicio 4.3.10. Se quiere paralelizar el siguiente ciclo de forma que la asignación de iteraciones a los procesadores disponibles se realice en tiempo de ejecución (dinámicamente):

```
1 for (i=0; i<100; i++) {
2     // Código que usa i
3 }
```

Observación. Considerar que las iteraciones del ciclo son independientes, que el único orden no garantizado por el sistema de memoria es W→R, que las primitivas atómicas garantizan que sus accesos a memoria se realizan antes que los accesos posteriores y que el compilador no altera el código.

1. Paralelizar el ciclo para su ejecución en un multiprocesador que implementa la primitiva `Fetch&Or` para garantizar exclusión mutua.

2. Paralelizar el anterior ciclo en un multiprocesador que además tiene la primitiva `Fetch&Add`.

Ejercicio 4.3.11. Un programador está usando el siguiente código para barreras (`bar` es un vector compartido, `k` es una variable compartida, el resto son variables locales, `Fetch_&_Or(k,1)` realiza sus accesos a memoria antes de que puedan realizarse los accesos posteriores):

```
1  Barrera(id, num_procesos)
2  {
3      band_local= !(band_local)
4      while (Fetch_&_Or(k,1)==1) {};
5      cont_local = ++bar[id].cont;
6      k=0;
7      if (cont_local == num_procesos) {
8          bar[id].cont=0;
9          bar[id].band=band_local;
10     }
11     else while (bar[id].band != band_local) {};
12 }
```

Conteste a las siguientes cuestiones (considere que el compilador no altera el código):

1. ¿Funciona bien este código como barrera en un multiprocesador en el que lo único que no garantiza su modelo de consistencia es el orden `W`→`R`? Razone por qué.

2. Funciona bien este código como barrera en un multiprocesador con modelo de consistencia que no garantice ningún orden en los accesos a memoria? Razone por qué.

Ejercicio 4.3.12. Se quiere implementar un programa que calcule en paralelo la siguiente expresión en un multiprocesador en el que sólo se relaja el orden `W`→`R` y en el que sólo se dispone de primitiva de sincronización `test_&_set`:

$$d = \frac{1}{N}\sum_{i=1}^{N} x_i^2 - \overline{x}^2, \quad \overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Un programador ha implementado el código de abajo. Tenga en cuenta lo siguiente:

- El código lo ejecutan `nthread` threads en paralelo.

- `ithread` es una variable local que nota el identificador del thread.
- `i`, `medl` y `varil` son variables locales.
- `i`, `med`, `vari`, el vector `x` y `N` son variables compartidas.
- Inicialmente `med`, `vari`, `medl` y `varil` son 0.

```
1 for (i=ithread;i<N;i=i+nthread) {
2     medl=medl+x[i];
3     varil=varil+x[i]*x[i];
4 }
5 med = med + medl/N; vari = vari + varil/N;
6 vari= vari - med*med;
7 if (ithread==0) printf("varianza = %f", vari); //imprime en pantalla
```

Conteste a las siguientes cuestiones (considere que el compilador no altera el código):

1. Se ha ejecutado este código usando varios threads y se ha visto que, aunque `N` y el vector `x` no varían, no siempre se imprime lo mismo. ¿Por qué ocurre esto?
2. Añada lo mínimo necesario para solucionar el problema teniendo en cuenta que sólo se dispone para implementar sincronización de `test_&_set` (tampoco se dispone de primitivas software de sincronización). Indique qué variables son ahora compartidas y cuáles locales.
3. Escriba el programa suponiendo que el multiprocesador además tiene primitivas de sincronización `fetch_&_add` (se tendrá en cuenta las prestaciones). Indique qué variables son compartidas y cuáles locales.
4. Escriba el programa ahora suponiendo que el multiprocesador sólo tiene primitivas de sincronización `compare_&_swap` (se tendrá en cuenta las prestaciones). Indique qué variables son compartidas y cuáles locales.

Observación. En todos los apartados puede añadir o quitar variables si lo estima conveniente.

Ejercicio 4.3.13. Se ha extraído la siguiente implementación de cerrojo (*spin-lock*) para x86 del kernel de Linux, el cual se muestra en el Código Fuente 4.1.

```
1 typedef struct{
2     unsigned int slock;
3 } raw_spinlock_t;
4
5 // Para un numero de procesadores mayor que 256=2^8
6 #if (NR_CPUS < 256)
7
8 static __always_inline void __ticket_spin_lock(raw_spinlock_t *lock){
9     short inc = 0x0100;
10
```

```
11      asm volatile (
12          "lock xaddw %w0,%1\n"    /*w: se queda con los dos ultimos LSB.*/
13          "1:                \t"   /*b: se queda con el LSB*/
14          "cmpb %h0, %b0     \n\t"  /*h: se queda con el segundo LSB*/
15          "je 2f             \n\t"  /*f: forward */
16          "rep ; nop         \n\t"  /*retardo, es equivalente a pause*/
17          "movb %1, %b0      \n\t"
18          /* don't need lfence here, because loads are in-order */
19          "jmp 1b            \n"    /*b: backward */
20          "2:"
21          : "+Q" (inc), "+m" (lock->slock) /*%0 es inc, %1 es lock->slock */
22          /*Q asigna cualquier registro al que se pueda acceder con rh: a, b, c y
        d; ej. ah, bh ... */
23          :
24          : "memory", "cc");
25  }
26
27  static __always_inline void __ticket_spin_unlock(raw_spinlock_t *lock){
28      asm volatile( "incb %0"          /*%0 es lock->slock */
29          : "+m" (lock->slock)
30          :
31          : "memory", "cc");
32  }
33
34  // ...
35  #endif
```

Código Fuente 4.1: Código de un cerrojo en el kernel de Linux.

Conteste a las siguientes preguntas:

1. Utiliza una implementación de cerrojo con etiquetas ¿Cuál es el contador de adquisición y cuál es el contador de liberación?

2. Describa qué hace `xaddw %w0, %1` ¿opera con el contador de adquisición, con el de liberación o con los dos? ¿qué operaciones hace con ellos?

3. Describa qué hace `cmpb %h0, %b0` ¿opera con el contador de adquisición, con el de liberación o con los dos? ¿qué operaciones hace con ellos?

4. ¿Por qué cree que se usa el prefijo `lock` delante de la instrucción `xaddw`?

Observación. Es recomendable consultar recursos siguientes:

- Puede consultar las instrucciones en el manual de Intel con el repertorio de instrucciones (Volumen 2 o volúmenes 2A, 2B y 2C) que puede encontrar aquí.

- Si no recuerda la interfaz entre C/C++ y ensamblador en `gcc` (se ha presentado en Estructura de Computadores), consulte el manual de `gcc` aquí.

4.3.1. Cuestiones

Cuestión 4.3.1. Diferencias entre núcleos con multithread temporal y núcleos con multithread simultánea.

Cuestión 4.3.2. Diferencias entre núcleos con multithread temporal de grano fino y núcleos con multithread temporal de grano grueso.

Cuestión 4.3.3. Suponga un multiprocesador con protocolo MESI de espionaje. Si un controlador de cache observa en el bus un paquete de petición de lectura exclusiva de un bloque que tiene en estado C, debe (indique cuál sería la respuesta correcta y razone por qué es la respuesta correcta):

1. Generar un paquete de respuesta con el bloque y pasar el bloque a estado I.
2. Pasar el bloque a estado I.
3. Generar un paquete de respuesta con el bloque y pasar el bloque a estado E.
4. No tiene que hacer nada.

Cuestión 4.3.4. Suponga un multiprocesador con protocolo MESI de espionaje. Si un nodo observa en el bus un paquete de petición de lectura exclusiva de un bloque que tiene en estado M, ¿qué debe hacer? Razone su respuesta.

Cuestión 4.3.5. Suponga un multiprocesador con el protocolo MESI de espionaje. Si el procesador de un nodo escribe en un bloque que tiene en su cache en estado I, debe (indique cuál sería la respuesta correcta y razone por qué es la respuesta correcta):

1. Generar paquete de petición de acceso E al bloque y pasar el bloque a M.
2. Generar paquete de petición de acceso E y pasar el bloque a estado E.
3. Generar paquete de petición de acceso E con lectura y pasar el bloque a estado E.
4. Generar paquete de petición de acceso E al bloque con lectura y pasar el bloque a M.

Cuestión 4.3.6. ¿Cuál de los siguientes modelos de consistencia permite mejores tiempos de ejecución? Justifique su respuesta.

1. Modelo de consistencia que no garantiza los órdenes `W`→`W` y `W`→`R`.
2. Modelo implementado en los procesadores de la línea x86.
3. Modelo de consistencia secuencial.
4. Modelo de consistencia que no garantiza ningún orden.

Cuestión 4.3.7. Indique qué expresión no se corresponde con la serie (justifique su respuesta):

1. `Lock`.
2. `Fetch_and_Or`.
3. `Compare_and_Swap`.
4. `Test_and_Set`.