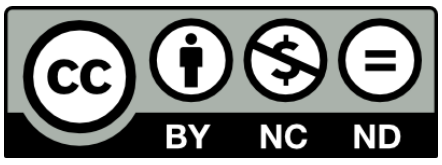
Arquitectura de Computadores

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



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

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

Arquitectura de Computadores

Los Del DGIIM, `losdeldgiim.github.io`

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 poseer.

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 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 disminuir 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 detalles:

- 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 impmlementar 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*, ...

Comunicación uno-a-todos. Un proceso envía y todos los procesos del grupo reciben el mensaje. 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. 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.

Comunicación todos-a-todos. Todos los procesos del grupo ejecutan una comunicación uno-a-todos. 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.

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.

- 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.

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*).
- Paralelismo de datos, descomposición de datos, o descomposición de dominio.
- Divide y vencerás (*divide and conquer*), o descomposición recursiva.
- Estructura segmentada (*pipeline*), o flujo de datos.

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.

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 esta 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, quien realizan el cómputo, devolviendo el trabajo al maestro y este a los clientes.

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 private(i)
    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 aplica a un flujo de datos en secuencia distintas funciones (paralelismo de tareas). La estructura de los procesos y de las tareas es la de un cauce segmentado: cada proceso ejecuta por tanto distinto código. 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.

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 independiéntemente 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.

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, trapcio, fórmula de Simpson, ...). 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: beg

```
main(int argc, char** argv){
    double ancho, x, 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;
}
```

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**:

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

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

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 últi 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). 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 `priate(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.

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 etapaas 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 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, esperamos un $T_O(4) > 0 \implies T_P(4) > 4$.

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 vió 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). 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):

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

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:

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

Sin embargo, la escalabilidad del modelo estará limitada por:

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

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:

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

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 denominzador 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. Es decir, 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 peude 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:

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

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 (además, tendremos en cuenta también la dependencia respecto a la variable n, anteriormente definida):

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

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

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

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 $1/p$. Una expresión que refleje en qué medida hay que incrementar el tamaño de los datos para conseguir una eficiencia constante, incluyendo el tiempo de sobrecarga, es una buena forma de evaluar una implementación paralela y/o arquitectura.

3. Relaciones de Problemas

3.1. Arquitecturas Paralelas

Ejercicio 3.1.1. 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 3.1.2. 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 3.1.3. 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 3.1.4. ¿Qué ocurriría en el problema del ejercicio anterior (Ejercicio 3.1.3) 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 3.1.5. 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 3.1.6. Se ha diseñado un compilador para la máquina `LOAD/STORE` del problema anterior (Ejercicio 3.1.5). 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 3.1.7. 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 3.1.8. 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 3.1.9. 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 3.1.10. 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.

3.1.1. Cuestiones

Cuestión 3.1.1. 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 3.1.2. ¿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 3.1.3. ¿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 3.1.4. 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 3.1.5. 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 3.1.6. ¿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 3.1.7. 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 3.1.8. ¿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 3.1.9. ¿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 3.1.10. ¿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 3.1.11. 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 3.1.12. 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.

3.1.2. Ejercicios adicionales

Ejercicio 3.1.11. 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 3.1.12. 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 3.1.13. 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?