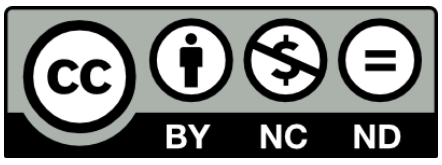
Arquitectura de Computadores

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Arquitectura de Computadores

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2023-2024

Índice general

1. Arquitecturas Paralelas

1.1. Clasificación del paralelismo implícito en una aplicación

1.1.1. Objetivos

Como items a conocer en esta sección, destacamos:

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

1.1.2. Niveles y tipos de paralelismo

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

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

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

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

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

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

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

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

Nivel de operaciones
: En este nivel se extra el paralelismo disponible entre operaciones. Las operaciones independientes se pueden ejecutar en paralelo. Por otra parte, podemos encontrar instrucciones compuestas de varias operaciones que se aplican en secuencial mismo tipo de datos de entrada. Por ejemplo, la instrucción `mac` nos permite realizar una suma tras una multiplicación. En este nivel se puede detectar la 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 operaciones en el paradigma del paralelismo funcional. Por ejemplo, contamos con las instrucciones SIMD (se desarrollarán próximamente), que con una instrucción puede manipular múltiples datos. Un ejemplo de instrucción SIMD es la implementación de una 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; mientras que disponemos de paralelismo a nivel de datos en las operaciones, al tener instrucciones que nos permitan sumar (con una sola instrucción) dos vectores.

Granularidad

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

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

Dependencias de datos

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

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

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

RAW (Read After Write)

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

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

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

WAW (Write after Write)

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

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

Donde en la primera y tercera linea 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 llamda 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 hebrs puedan crearse, destruirse y comunicarse entre ellas de una forma más rápida que los procesos. Todo esto permite que las hebras dispongan de una menor granularidad que estos.

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

1.1.4. Implementación del paralelismo

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

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

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

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

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

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

con el resto de procesos[1]. Para poder implementar este tipo de paralelismo, es necesario disponer de un multiprocesador o multicomputador.

Nivel de funciones

El paralelismo a este nivel puede extraerse para realizarse a nivel de procesos (si la función realmente lo 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).

Nivel de bucles

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

Nivel de operaciones

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

1.1.5. Detección y extracción del paralelismo

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

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

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

Nivel de operaciones

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

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

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

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

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

1.2. Clasificación de arquitecturas paralelas

1.2.1. Objetivos

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

- Distinguir entre procesamiento o computación paralela y distribuida.
- Clasificar los computadores según segmento del mercado.
- Distinguir entre las diferentes clases de arquitecturas de la clasificación de Flynn.
- Diferenciar un multiprocesador de un multicomputador.
- Distinguir entre NUMA y SMP.
- Distinguir entre arquitecturas DLP, ILP y TLP.
- Distinguir entre arquitecturas TLP con una instancia de SO y TLP con varias instancias de SO.

1.2.2. Computación distribuida

Computación paralela
Esta estudia los aspectos hardware y software relacionados con el desarrollo y ejecución de aplicaciones en un sistema de cómputo compuesto por varios cores, procesadores o computadores que es visto externamente como una sóla unindad autónoma, a la que le llamadamos 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 instrucciones, lo que nos permite tener múltiples unidades de procesamiento, cada una con comunicación independiente con memoria. De esta forma, un computador SIMD puede realizar varias operaciones similares simultáneas con distintos operandos. Cada una de las secuencias de datos y resultados constituyen flujos independientes. Un ejemplo de sistema SIMD puede ser un procesador vectorial.

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

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

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

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

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

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

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

En multicomputadores, cada proceadsor 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 ala 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 propósito de los multicomputadores, mientras que los multiprocesadores deben usarlo para comunicaciones y acceso a memoria, lo que dificulta la ejecución de los procesadores de forma paralela, al tener estos que acceder constantemente a memoria de forma simultánea. Por tanto, nos es más fácil añadir procesadores a un sistema multicomputador antes que a uno multiprocesador, ante el temor de saturar la red de intercomunicaciones. Posteriormente comentaremos una mejora de los multiprocesadores que trata de parchear este problema.

A continuación, seguimos planeando diferencias entre estos:

Latencia de acceso a memoria El tiempo de acceso a memoria (como se puede esperar) es mayor en multiprocesadores que en multicomputadores, al tener que atravesar toda la infraestructura de red de interconexión, junto con lo que esto conlleva, ya que puede darse la posibilidad de que varios procesadores ocupen la red de forma simultánea (lo cual ya plantea un problema), pero además deberemos arbitrar el acceso de distintos procesadores a una misma posición de memoria. Cuanto mayor sea el número de procesadores, la 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 este implementa paralelismo entre procesadores, que es el caso interesante), debemos ubicar en memoria (en la de cada procesador) el código del programa que estamos a punto de ejecutar, junto con los datos que este necesita. Es decir, es necesario realizar una distribución de carga de trabajo entre los distintos procesadores. Nótese que en multiprocesadores esta distribución no es necesaria, ya que todos los procesadores pueden acceder al mismo espacio de direcciones. Esto presenta un gran problema, ya que nos 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 venga de cambiar un poco el planteamiento del sistema.

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

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

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

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 cómun diferenciar dentro del tiempo de CPU el *Tiempo de CPU de usuario* y el *Tiempo de CPU de sistema*, 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.

A lo largo de esta sección, nos centraremos exclusivamente en el estudio de tiempo de CPU. Por tanto, supondremos que los dos tiempos anteriores son despreciables

y para nosotros, el tiempo de CPU será igual al tiempo de respuesta. Hecha esta asunción, podemos calcular el tiempo de CPU como:

$$T_{CPU} = \text{Ciclos del programa} \cdot T_{ciclo} = \frac{\text{Ciclos del programa}}{F} \tag{1.1}$$

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

$$F = \frac{1}{T_{ciclo}} \tag{1.2}$$

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 (1.1) se puede reescribir usando la relación:

$$\text{Ciclos del programa} = NI \cdot CPI \tag{1.3}$$

de donde obtenemos:

$$T_{CPU} = NI \cdot CPI \cdot T_{ciclo} = \frac{NI \cdot CPI}{F} \tag{1.4}$$

Asumiendo que el número de ciclos por instrucción es constante (es decir, todas las instrucciones tardan el mismo tiempo ene 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:

$$CPI = \frac{\sum_{i=1}^{W} NI_i \cdot CPI_i}{NI} \tag{1.5}$$

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 (1.1) obtenemos:

$$T_{CPI} = \sum_{i=1}^{W} (NI_i \cdot CPI_i) \times T_{ciclo} \tag{1.6}$$

1.3.3. MIPS y FLOPS

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

$$MIPS = \frac{NI}{T_{CPU} \cdot 10^6} = \frac{\cancel{NI}}{\cancel{NI} \cdot CPI \cdot T_{ciclo} \cdot 10^6} = \frac{F}{CPI \cdot 10^6} \tag{1.7}$$

2. Relaciones de Problemas