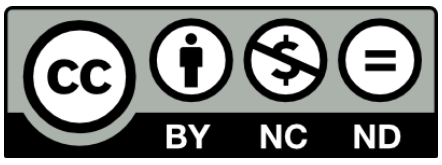
Arquitectura de Computadores

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. 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.

1.1. 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.

1.1.1. 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.

1.1.2. 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.

1.1.3. 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).

1.1.4. 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()`.

1.1.5. 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, ...

1.1.6. Estructuras típicas de códigos paralelos

1.2. Proceso de paralelización

1.2.1. Objetivos

Esta sección está dedicada a:

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

1.3. Evaluación de prestaciones

1.3.1. 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. Relaciones de Problemas