Modelos de programación paralela y concurrente

J. Daniel García Sánchez (coordinador)
David Expósito Singh
Javier García Blas
Óscar Pérez Alonso
J. Manuel Pérez Lobato

Arquitectura de Computadores
Grupo ARCOS
Departamento de Informática
Universidad Carlos III de Madrid

1. Estructura del módulo

Este módulo está estructurado en cuatro lecciones:

- **Programación paralela con OpenMP**. Presenta los conceptos básicos del modelo de programación paralela de OpenMP mediante ejemplos.
- **Programación concurrente en C++11**. Introduce el modelo de concurrencia del lenguaje C++11 como mecanismo portable para la programación concurrente.
- **Consistencia de memoria en C++**. Refuerza los conceptos de consistencia de memoria a través del modelo de memoria estándar de C++. Presta especial atención a los modelos de memoria relajados.

2. Programación paralela con OpenMP

Esta lección tiene la siguiente estructura general:

1. Introducción.
2. Hilos en OpenMP.
3. Sincronización.
4. Bucles paralelos.
5. Sincronización con master.
6. Compartición de datos.
7. Secciones y planificación.

2.1. Introducción

OpemMP es un API que permite expresar aplicaciones paralelas para sistemas de memoria compartida, simplificando la forma de escribir los programas. Se compone de un conjunto de directivas de compilador, una biblioteca de funciones y un conjunto de variables de entorno.

Todas las directivas de OpenMP se indican mediante un **pragma** que usa el prefijo **omp**. Por ejemplo:

```
#pragma omp parallel
{
  f();
  g();
}
```

2.2. Hilos en OpenMP

El paralelismo en OpenMP sigue el modelo *fork-join* donde se tiene una aplicación secuencial con secciones paralelas. Al iniciar el programa existe un *hilo maestro* que ejecuta las secciones secuenciales. Cuando se entra en una región paralela (marcada con una directiva **omp parallel** se arranca un conjunto de hilos.

Para medir el tiempo que tarda en ejecutarse una sección de código se pueden usar las función suministrada por OpenMP (**omp_get_wtime()**).

2.3. Sincronización

OpenMP ofrece mecanismos de sincronización con el objetivo de que el programador pueda tratar de evitar las carreras de datos. La biblioteca ofrece mecanismos de alto nivel (*critical*, *atomic*, *barrier* y *ordered*) y de bajo nivel (*flush* y cerrojos).

La directiva **critical** garantiza que solamente un hilo puede entrar en la sección a la vez. Por otra parte, la directiva **atomic** garantiza la actualización atómica de una posición de memoria.

2.4. Bucles paralelos

La directiva **parallel for** realiza división de bucles. Es decir, reparte las iteraciones de un bucle entre los hilos disponibles.

Una de las operaciones más frecuentes en bucles paralelos es la reducción, que es una operación de acumulación realizada en un bucle. Las reducciones se puede realizar sobre distintas operaciones elementales que deben ser asociativas.

2.5. Sincronización con master

Una barrera permite sincronizar todos los hilos en un punto de modo que se espera a que todos los hilos alcancen dicho punto.

Otra forma de sincronización es el uso de una sección marcada con la directiva **omp master**. Dicha sección será solamente ejecutada por el hilo maestro. Si lo que se quiere garantizar es que la sección la ejecute un único hilo, sin importar cuál, se puede usar la directiva **omp single**. También se puede exigir la ejecución ordenada de una reducción con **omp ordered**.

OpenMP ofrece las primitivas de cerrojos **omp_set_lock()** y **omp_unset_lock()** que usan el tipo **omp_lock_t**.

2.6. Compartición de datos

En OpenMP una variable puede ser compartida o privada.

Una variable compartida puede ser una variable global, una variable **static** o bien un objeto alojado en memoria dinámica. Una variable privada es una variable local de una función invocada desde una sección paralela o una variable local definida en un bloque.

Una clausula **omp** puede usarse para controlar los atributos de almacenamiento de una variable. La directiva **private** crea una copia local de una variable en cada hilo. Las directivas **firstprivate** y **lastprivate** ayudan a controlar la gestión de los valores iniciales y finales de las variables.

2.7. Secciones y planificación

Se pueden definir un conjunto de secciones paralelas dentro de una región paralela. En este caso cada sección se pasa a un hilo distinto y todas las secciones se sincronizan con una barrera de finalización.

A la hora de ejecutar bucles paralelos se puede seleccionar entre varias políticas de ejecución. La política *static* planifica bloques de iteraciones del mismo tamaño para cada hilo. La política *dynamic* hace que cada hilo tome un número de iteraciones de una cola hasta que se hayan procesado todas. La política *guided* hace que cada hilo tome un grupo de iteraciones de una cola. En este último caso, a medida que pasa el tiempo se va reduciendo el número de iteraciones que se toma cada vez.

3. Programación concurrente en C++11

Esta lección tiene la siguiente estructura general:

- Introducción a la concurrencia en C++.
- Visión general de la biblioteca.
- La clase **thread**.
- Objetos *mutex* y variables condición.

3.1. Introducción a la concurrencia en C++

El estándar C++11 (ISO/IEC 14882:2011) ofrece un modelo de concurrencia como parte de la especificación del lenguaje. Esto supone un cambio importante con comparación con aproximaciones previas puesto que resuelve el problema de la portabilidad de código concurrente entre distintas plataformas. Además, también resuelve los problemas inherentes a soluciones de concurrencia basadas exclusivamente en una biblioteca, puesto que hay aspectos que se resuelven de forma más satisfactoria con soporte del lenguaje. Por tanto, la solución de C++11 para concurrencia abarca tanto aspectos del lenguaje como aspectos de la biblioteca estándar que lo acompaña. Por otra parte, esta norma ha tenido una gran influencia sobre el estándar C11 (ISO/IEC 9899:2011) que sigue las líneas establecidas por C++11.

Desde el punto de vista del lenguaje, C++11 ofrece un nuevo modelo de memoria (que se presenta en la siguiente lección), así como un nuevo tipo de variables con almacenamiento local al hilo (**thread_local**). Por su parte, la biblioteca estándar ofrece una familia de tipos atómicos, útiles en el contexto de la programación libre de cerrojos, así como un conjunto de abstracciones portables para la concurrencia (**thread**, **mutex**, **lock**, **packaged_task**, **future**).

3.2. Visión general de la biblioteca

3.2.1. Hilos

La abstracción de hilo de ejecución se ofrece a través de la clase **std::thread** y representa a un hilo ofrecido por la plataforma. Como en cualquier solución de concurrencia, dos hilos pueden acceder a un objeto compartido, lo que podría dar lugar a una carrera de datos.

En C++11, los hilos ofrecen un mecanismo de paso de argumentos simplificado sin necesidad de realizar ningún tipo de conversiones (*casts*).

En general, un hilo se puede construir a partir de cualquier objeto invocable. Esto incluye funciones y objetos función, así como expresiones lambda.

3.2.2. Acceso a datos compartidos

Un **std::mutex** es un tipo que permite controlar el acceso con exclusión mutua a un recurso. Ofrece dos operaciones básicas de adquisición (**lock**) y liberación (**unlock**). Para evitar posibles problemas, como el olvido de la liberación o los problemas derivados de las excepciones, se ofrece un envoltorio (**unique_lock**) que libera el cerrojo en la destrucción.

Otro problema clásico que resuelve la biblioteca es el de la adquisición de múltiples cerrojos. En este caso, la función **lock()** toma un número arbitrario de cerrojos permitiendo la adquisición de todos en una única operación.

3.2.3. Esperas

La biblioteca estándar ofrece mecanismos para acceder a diversos relojes. De ellos, merece especial atención el tipo **high_resolution_clock**, que es el de más alta resolución de los disponibles. La diferencias entre dos puntos temporales (diferencias de tiempo) se pueden expresar en distintas unidades que se pueden expresar de forma explícita.

La función **sleep_for()** permite especificar una espera durante una determinada cantidad de tiempo. Cabe destacar que dicha función se encuentra en el espacio de nombres anidado **std::this_thread**.

Una variable condición es un mecanismo que permite sincronizar hilos en el acceso a recursos compartido. Una variable condición permite especificar que un hilo quede esperando una notificación. La espera se asocia a un **mutex**. La variable condición también ofrece dos operaciones de notificación (**notify_one()** y **notify_all()**).

3.2.4. Tareas asíncronas

Una tarea asíncrona permite el lanzamiento simple de la ejecución de una tarea ya sea en otro hilo de ejecución o como una tarea diferida. Cuando se invoca una tarea asíncrona se obtiene un *futuro*, que es un objeto que permite devolver un valor transportándolo de un hilo a otro.

3.3. La clase std::thread

La clase **std::thread**, introducida anteriormente, representa la abstracción de hilo ofrecida por la plataforma (ya sea el hardware o el sistema operativo). Todos los hilos que se crean dentro de un programa comparten un mismo espacio de direcciones la que simplifica la compartición de memoria.

Es importante resaltar que cada hilo que se crea tiene su propia pila. Esto plantea posibles peligros para los programas. Por una parte, si se pasa un puntero o una referencia no constante a otro hilo se está dando acceso a la pila del hilo original. También, si se pasa una referencia a través de una captura de una expresión lambda se está dando acceso a la pila del hilo original. ¿Por qué supone esto

un problema? Si el hilo original termina, su pila se libera y esta memoria podría ser asignada a otro objeto, pero otros hilos todavía tendrían referencias a dicha memoria.

Los objetos de la clase **std::thread** no se pueden copiar. No obstante si que soportan la semántica de movimiento, lo que permite que un hilo pueda ser transferido desde un contexto a otro.

Existen múltiples formas de construir los hilos. En todos los casos una de las características es que al construir el hilo se pueden pasar los argumentos de la función a ejecutar sin necesidad de realizar conversiones de tipo ni realizar uso innecesario de punteros.

Una técnica que se suele usar a veces se conoce como construcción en dos etapas. Para ello, se define por cada hilo una clase que tiene definidos el constructor y el operador de invocación a función (**operator()**). En la primera etapa se construye el objeto. En la segunda etapa se crea un hilo pasando el objeto creado. Esta técnica es especialmente útil es estructuras complejas de dependencias entre hilos.

Cada hilo tiene un identificador único (obtenido mediante **mihilo.get_id()**). Si bien el tipo de los identificadores es un tipo definido por cada implementación, el conjunto de requisitos que debe satisfacer permite que los identificadores se puedan usar como clave para almacenar un conjunto de hilos en una estructura de datos.

Cuando se desea esperar la terminación de un hilo se debe invocar la operación **join()**. Si un objeto de tipo hilo se destruye sin que se haya invocado para él la operación **join()** se produce un error y la biblioteca invoca a la función **terminate()**. Existen distintas alternativas a esta solución. Una posibilidad es definir un tipo de hilo especializado cuyo destructor invoque a **join()** siempre que sea necesario. Otra solución es el uso de hilos no asociados. No obstante esta solución suele estar indicada solamente en casos de hilos que actúan como demonios.

3.4. Objetos mutex y variables condición

La biblioteca estándar ofrece varios tipos de **mutex**. El más simple de todos es **std::mutex**. Si el objeto necesita ser adquirido más de una vez por un mismo hilo (p. ej. funciones recursivas) se puede usar un **std::recursive_mutex**. Si es necesario hacer uso de operaciones con tiempo límite se puede usar un **std::timed_mutex**. Las propiedades de estos dos últimos tipos se combinan en **std::timed_recursive_mutex**.

Las operaciones **lock()** y **unlock()** permiten adquirir de forma bloqueante u liberar un **mutex**. En casos en que la adquisición bloqueante no es admisible se puede usar **try_lock()** que intenta adquirir el objeto y devuelve una indicación de éxito/fracaso.

En el caso de los **std::timed_mutex** se añaden operaciones para adquirir un **mutex** indicando un plazo de tiempo en forma de duración (**try_lock_for(dur)**) o en forma de punto temporal (**try_lock_until(t)**).

Como complemento a estos tipos, la biblioteca ofrece variables condición. El tipo **std::condition_variable** está optimizado para usarse con **std::mutex**. En otro caso puede usarse el tipo **std::condition_variable_any**. Se debe recordar que antes de destruir una variable condición se debe despertar a todos los hilos que estén esperando en ella o se corre el riesgo de generar un interbloqueo.

Existen dos operaciones de notificación sobre una variable condición. La operación **notify()** despierta a uno de los hilos que está esperando en la misma. La operación **notify_all()** despierta a todos los hilos que están esperando.

Las operaciones de espera toman siempre como argumento un cerrojo. De esta forma una operación **wait()** bloquea hasta que se consigue adquirir el cerrojo pasado como argumento. Se puede especificar un tiempo límite como segundo argumento (**wait_for()** o **wait_until()**).

4. Consistencia de memoria en C++

Esta lección tiene la siguiente estructura general:

1. Modelo de memoria.
2. Tipos atómicos.
3. Relaciones de ordenación.
4. Modelos de consistencia.
5. Barreras.

4.1. Modelo de memoria

Como ya se ha indicado anteriormente, C++11 define un modelo de concurrencia propio como parte del lenguaje. El objetivo último es evitar que sea necesario escribir código en lenguajes de más bajo nivel para obtener mayores prestaciones. Para ello, el lenguaje ofrece una biblioteca de tipos atómicos que deben tener soporte por parte del lenguaje. Además la biblioteca también incorpora un conjunto de operaciones de sincronización de bajo nivel.

Una definición básica usada en la definición del modelo de memoria es la de *objeto*. Desde este punto de vista, un *objeto* es una región de almacenamiento. Es decir, se trata de una secuencia de uno o más bytes. Una *posición de memoria* es un objeto de un tipo escalar o una secuencia de campos de bits adyacentes. Teniendo presente esto, se puede afirmar que un objeto de cualquier tipo se almacena siempre en una o más posiciones de memoria.

La definición de *posición de memoria* es importante porque las condiciones que eventualmente pueden dar lugar a una carrera de datos se definen en términos del propio concepto de posición de memoria.

Así, si dos hilos intentan acceder de forma simultánea a la misma posición de memoria y alguno de los accesos es de escritura existe una condición de carrera potencial. Para evitar que dicha condición de carrera se materialice es necesario forzar un orden entre las operaciones de acceso a memoria.

Para forzar un orden en las operaciones de acceso a memoria una solución es el uso de mecanismos de sincronización de alto nivel. Una alternativa que puede ser en algunos casos más eficiente, aunque más compleja es el uso de operaciones atómicas.

4.2. Tipos atómicos

Las operaciones atómicas son operaciones indivisibles. Por tanto, si dos hilos realizan operaciones de lectura o escritura sobre una misma variable dichas operaciones se realizan en algún orden y no dan lugar a carreras de datos. Sin embargo, si alguna de las operaciones no es atómica, el comportamiento no esta definido.

La biblioteca C++ ofrece acceso a operaciones atómicas a través de un conjunto de tipos (**atomic<T>**). Se ofrecen operaciones atómicas para tipos integrales, punteros, y booleanos. Sin embargo, no se ofrecen para tipos en coma flotante.

Las operaciones sobre tipos atómicos admiten que se especifique el modelo de consistencia de memoria. Por defecto, se usa el modelo de consistencia secuencial, pero se pueden especificar modelos de consistencia más relajados.

4.3. Relaciones de ordenación

Existen dos relaciones que se usan para razonar sobre el comportamiento de programas concurrentes: *sincroniza-con* y *ocurre-antes-que*.

La relación *sincroniza-con* es una relación que se da entre operaciones efectuadas sobre tipos atómicos desde distintos hilos. Por otra parte, la relación *ocurre-antes-que* es una relación que se da entre operaciones que ocurren en el mismo hilo.

4.4. Modelos de consistencia

El modelo de consistencias sobre el que es más simple razonar es la consistencia secuencial, ya que en este caso, el programa es consistente con una vista secuencial. Es decir, el comportamiento es como si todas las operaciones se realizasen en algún orden particualr en un único hilo. Aunque es un modelo que simplifca el razonamiento, tiene un mayor coste en rendimiento.

En los modelos secuencialmente no consistentes deja de haber un orden global de los eventos y cada hilo puede tener una vista distinta del orden en que ocurren. Aún así, todos los hilos deben estar de acuerdo en el orden de modificación de cada variable. Entre los modelos no consistentes se encuentran la consistencia relajada y la consistencia de liberación-adquisición.

Las operaciones que usan una consistencia relajada no participan en relaciones *sincroniza-con*, aunque si participan en relaciones *ocurre-antes-que* dentro del mismo hilo. Como consecuencia, no se pueden reordenar operaciones dentre del mismo hilo, pero no se puede asegurar nada sobre el ordenamiento de operaciones en distintos hilos.

Las operaciones que usan consistencia de liberación-adquisición suponen un nivel intermedio entre la consistencia secuencial y la consistencia relajada puesto que aunque no establecen exactamente el orden si que establecen algunas restricciones.

Una propoiedad interesante es que, en muchos casos, se puede obtener el efecto de la consistencia secuencial combinando la consistencia de liberación-adquisición y la consistencia relajada, con lo que se puede obtener mejor rendimiento.

4.5. Barreras

Las barreras son operaciones de sincronización que establecen ordenamiento entre accesos a memoria sin modificar datos. Típicamente se utilizan para establecer restricciones sobre otros accesos a variables atómicas.