



**Universidad Nacional Autónoma de México**

**Materia: Sistemas Operativos**

**Profesor: GUNNAR EYAL WOLF ISZAEVICH**

**Tenorio Martinez Jesus Alejandro**

# Exposición: Como hacer programación paralela en C, con openMP

**Exposición: Como hacer programación paralela en C, con openMP**

**Introducción.**

Como estudiante de ingeniería en computación, es natural que cuando pienses en programar un sistema, lo enfoques de manera secuencial. Esto está bien, pero ¿alguna vez has considerado utilizar la programación paralela? ¿Sabes cuáles son las ventajas y desventajas de esta forma de programar?

Bienvenido a la programación paralela en C con OpenMP.

*Historia*

La programación paralela surgió por la necesidad de aumentar la capacidad de procesamiento de las computadoras para resolver problemas complejos y de gran escala. A medida que las aplicaciones informáticas crecieron en complejidad, la programación secuencial empezó a mostrar limitaciones, sobre todo en términos de velocidad y eficiencia.

En los años 60, Gordon Moore predijo que el número de transistores en los chips se duplicaría cada dos años. Si bien esta tendencia se mantuvo durante un tiempo, comenzaron a surgir limitaciones físicas, como el sobrecalentamiento y el consumo excesivo de energía. Estos problemas llevaron a la necesidad de buscar nuevas soluciones, como la creación de procesadores multinúcleo, lo que abrió las puertas a la programación paralela.

Los transistores en un chip se duplican cada dos años y el tamaño de los microchips se reduce, lo que sigue la tendencia establecida desde que fue publicada la Ley de Moore en 1965. Si observamos cómo ha crecido el número de transistores en una computadora desde 1960, vemos que se ha duplicado consistentemente. Sin embargo, este aumento ha traído consigo limitaciones físicas, como el sobrecalentamiento y el elevado consumo de energía, especialmente cuando los microchips se hacen más pequeños. Esto nos lleva al concepto del "muro de potencia", que representa el límite hasta donde es posible reducir el tamaño de los microchips sin provocar problemas de sobrecalentamiento. Si bien se podría optar por sistemas de enfriamiento, estos serían extremadamente costosos. Como agregar más transistores dejó de ser viable, se comenzó a desarrollar procesadores con múltiples núcleos para distribuir la carga de trabajo. Aunque surgieron estas limitaciones, ya se había investigado la idea de utilizar varios procesadores trabajando en paralelo para acelerar el procesamiento de tareas. Con el tiempo, esta investigación llevó al desarrollo de procesadores multinúcleo y supercomputadoras, lo que impulsó la necesidad de nuevos paradigmas de programación que aprovecharan varias unidades de procesamiento al mismo tiempo, como la programación paralela.

*¿Cuándo paralelizar?*

No todos los procesos o algoritmos se pueden o deben paralelizar. Cuando se trabaja con pocas instancias, paralelizar puede ser ineficiente, ya que el tiempo que tarda el compilador en dividir el trabajo entre varios hilos puede ser mayor que el que necesitaría un solo hilo. Depende del algoritmo y del tipo de problema si conviene paralelizar.

*¿Qué pasa cuando se paraleliza?*

En la programación secuencial, un procesador ejecuta cada instrucción de manera lineal. Cuando paralelizamos, el problema se segmenta en bloques, y cada procesador puede ejecutar un bloque diferente de forma simultánea. Esto puede acelerarse aún más si usamos clusters de computadoras, donde las tareas se distribuyen entre múltiples nodos.

**Desarrollo.**

*Latencia y Rendimiento*

Latencia: El tiempo necesario para completar una tarea.

Rendimiento: El número de tareas que se completan por unidad de tiempo.

La CPU suele optimizar la latencia, mientras que el GPU está diseñado para optimizar el rendimiento.

*Concurrencia y Paralelismo*

Concurrencia: Varias tareas pueden estar activas al mismo tiempo, pero no necesariamente ejecutándose simultáneamente. Si tienes un solo procesador, el S.O. alterna rápidamente entre varias tareas, dándoles pequeñas porciones de tiempo del procesador, haciendo que parezca que todas están ejecutándose al mismo tiempo. Esto es concurrencia, pero no implica que las tareas realmente se ejecuten de manera simultánea.

Paralelismo: Varias tareas se ejecutan realmente al mismo tiempo, si hay suficientes recursos de hardware (como múltiples procesadores). Si tienes un sistema con dos o más procesadores (o núcleos), varias tareas o hilos pueden ejecutarse simultáneamente en diferentes procesadores. En este caso, el sistema operativo no necesita alternar entre ellas porque hay suficiente capacidad de cómputo para que ambas corran al mismo tiempo.

*Prioridad de Hilos*

En un programa multihilo, los hilos compiten por el uso del procesador. El sistema operativo puede asignar prioridades a los hilos, lo que afecta su ejecución. Por ejemplo, en Java se pueden asignar niveles de prioridad:

- Prioridad normal: 5
- Máxima prioridad: 10

Esto no garantiza que el hilo con mayor prioridad termine primero, ya que la ejecución depende de otros factores, como el uso del procesador y la gestión de recursos.

*OpenMP*

OpenMP es un modelo de memoria compartida que permite escribir aplicaciones multihilos, la cual contiene un conjunto de directivas y bibliotecas con funciones y rutinas que permiten desarrollar aplicaciones paralelas. OpenMP permite transcribir de manera simple programas seriales desarrollados en Fortran, C y C++ en programas multihilo.

*Sintaxis básica de OpenMP*

Para que un archivo en lenguaje C realice programación paralela se debe incluir la biblioteca omp.h.

En OpenMP la mayoría de los constructores son directivas del compilador del estilo #pragma omp

La mayoría de los constructores de OpenMP aplican a un bloque estructurado. Un bloque estructurado contiene una o más sentencias con un punto de entrada al inicio y un punto de salida al final.

*Parallel*

La única manera de crear hilos en OpenMP es utilizando el constructor parallel. La directiva parallel genera el número de hilos establecidos para iniciar la ejecución paralela del programa. El código a ejecutar de manera paralela está delimitado por llaves. Las variables que se declaran fuera del bloque pragma omp parallel son compartidas por todos los hilos, las variables que se declaran dentro del bloque de la directiva se generan de manera individual para cada hilo.

Por tanto, el constructor parallel permite crear hilos en regiones paralelas, de la siguiente manera:

```
#pragma omp parallel
{
        // bloque paralelo
}
```

Cada hilo va a ejecutar una copia del código que se encuentra dentro de la estructura parallel.

Para obtener el tiempo de ejecución de un bloque de código se tiene la función:

```
double omp_get_wtime()
```

Para saber el ID del hilo que se está en ejecución se tiene la función:

```
int omp_get_thread_num()
```

*Niveles de paralelismo (Granularidad)*

La granularidad es un concepto que mide el grado o nivel de paralelismo que puede aprovechar un sistema, indicando cuántas operaciones pueden realizar los procesadores sin necesitar interacción constante entre ellos. El tamaño del grano refleja la cantidad de procesamiento que se requiere en un proceso, es decir, cuántas instrucciones contiene cada segmento del programa que puede ser ejecutado en paralelo.

Cuando un programa paralelo se divide en partes más pequeñas, estas se conocen como granos. Existen distintos niveles de granularidad, que se clasifican según el tamaño del grano:

Granularidad gruesa: El programa se divide en grandes bloques, que necesitan poca o ninguna comunicación entre ellos. Este enfoque es más eficiente en sistemas paralelos con procesadores potentes que no requieren estar estrechamente conectados.

Granularidad fina: Los procesadores necesitan comunicarse continuamente entre sí, ejecutando pocas instrucciones sin intercambiar información. Los sistemas con procesadores menos complejos pero muy bien interconectados son más adecuados para este tipo de granularidad.

El paralelismo puede analizarse en diferentes niveles, según la estructura del programa:

Trabajo o programa: Se ejecutan dos o más programas distintos de forma paralela.

Tarea o subprograma: Dentro de un mismo programa, se ejecutan varias tareas independientes que pueden interactuar entre sí.

Proceso o subrutina: Un proceso es un bloque de instrucciones que cumple una función específica dentro de una tarea mayor.

Ciclo o iteración: Un ciclo repite un bloque de instrucciones hasta que se cumple una condición específica.

Instrucción o sentencia: Las instrucciones individuales que forman parte de un ciclo o proceso son las unidades más pequeñas del cómputo. Este enfoque en niveles de granularidad ayuda a determinar cómo organizar un programa paralelo para maximizar el rendimiento, dependiendo de las características del sistema de hardware disponible.

*SPMD*

Single Program Multiple Data (SPMD) es una estrategia para construir algoritmos paralelos, en la cual la información global se particiona, asignando una porción de datos a cada nodo. Este patrón de diseño es muy general y ha sido utilizado como soporte para la mayoría (no todos) de los patrones de estrategias de algoritmos paralelos.

*Rutina de entorno de ejecución*

OpenMP proporciona varias funciones para mostrar las cualidades del entorno paralelo en tiempo de ejecución.

Para establecer el número de hilos que se desean crear en el bloque paralelo se tienen dos opciones:

```
double A[1000];

omp_set_num_threads(4);
#pragma omp parallel
{
        // bloque paralelo
}
```

```
double A[1000];

#pragma omp parallel num_threads(4);
{
        // bloque paralelo
}
```

Para saber si se está en una región paralela activa se tiene la función:

omp_in_parallel().

Para saber cuántos procesadores posee el sistema se ocupa la función:

omp_get_num_procs().

*Algoritmos con memoria compartida*

Una computadora de memoria compartida es aquella que cuenta con múltiples unidades de procesamiento, las cuales acceden y comparten un mismo espacio de direcciones de memoria. Este tipo de arquitecturas se divide en dos clases:

- Multiprocesador Simétrico (SMP): Todos los procesadores tienen el mismo acceso a un espacio de memoria compartido, y el sistema operativo asigna el tiempo de acceso de manera equitativa para cada procesador.
- Multiprocesador de Acceso No Uniforme (NUMA): Las diferentes regiones de memoria tienen costos de acceso variables, lo que significa que algunos procesadores pueden acceder más rápidamente a ciertas partes de la memoria que a otras.

En un programa de memoria compartida, el proceso puede contener múltiples hilos que interactúan directamente con la misma memoria compartida (la del proceso). El sistema operativo se encarga de gestionar la ejecución de estos hilos.

En OpenMP, se pueden definir dos tipos de variables:

- Variables compartidas: Son accesibles por cualquier hilo que esté en ejecución. Sin embargo, el acceso no está controlado, lo que puede generar conflictos si varios hilos acceden a la misma variable simultáneamente.
- Variables privadas: Son creadas por cada hilo individualmente y solo existen durante la ejecución de ese hilo. Su valor se pierde una vez que el hilo termina.

Las variables que se declaran fuera de la región parallel se colocan en la memoria compartida, mientras que las que se declaran dentro de la región parallel son privadas para cada hilo.

El constructor parallel en OpenMP crea un programa bajo el modelo SPMD (Single Program, Multiple Data), donde cada hilo ejecuta de manera redundante el mismo código. Cuando se utiliza trabajo compartido (worksharing), es posible dividir el código para que cada hilo ejecute una parte específica del trabajo. Para ello OpenMP posee varios constructores:

- Constructor de ciclo
- Constructor de secciones
- Constructor sencillo
- Constructor de tareas

*Constructor de ciclo*

El constructor de ciclo (loop construct) es el más común de los constructores para realizar trabajo compartido, su sintaxis es la siguiente:

```
#pragma omp parallel
{
    int i, id;
    id = omp_get_thread_num();
    #pragma omp for
    for (i=0; i<ENE ; i++){
        do_it(id);
    }
}
```

Cuando la sentencia for es lo único dentro del bloque paralelo, es posible unir las instrucciones pragma, es decir:

```
#pragma omp parallel for
{
        for (i=0; i<MAX ; i++){
                res[i] = do_it(i);
        }
}
```

Este constructor se encarga de dividir el ciclo entre los hilos disponibles.

Para determinar el uso correcto del constructor de ciclo se deben tener en cuenta los siguientes aspectos:

- Encontrar los ciclos que generen la mayor cantidad de cómputo del proceso.
- Hacer que las iteraciones del ciclo sean lo más independientes posibles, para que estos se puedan ejecutar sin depender del resultado de otro ciclo.
- Colocar la directiva de OpenMP en el lugar apropiado.
- Identificar las variables que deben ser compartidas y las que deben ser privadas.

Es posible anidar múltiples ciclos, mientras que estos se encuentren anidados, utilizando la cláusula collapse(num), especificando en el argumento el número de ciclos anidados a paralelizar.

*Carrera de Datos.*

Las condiciones de carrera pueden causar serios problemas de integridad, ya que el acceso a la información no está controlado de forma automática en los bloques paralelos. Para evitar

estos problemas, se utiliza la sincronización, que consiste en proteger los datos susceptibles de generar conflictos entre hilos. Sin embargo, la sincronización es un proceso costoso en términos de rendimiento, por lo que se recomienda minimizar su uso siempre que sea posible.

*Sincronización*

En OpenMP existen diferentes tipos de sincronización y, por ende, diversos constructores para sincronizar hilos. Así mismo, estos constructores se clasifican en dos niveles de sincronización: alto y bajo.

- El nivel alto de sincronización maneja cuatro constructores: critico, atómico, barrera y ordenado.
- El nivel bajo de sincronización maneja dos constructores flush y locks.

- *Barrier*

Barrier En la sincronización por barrera cada hilo se espera en el punto marcado (la barrera) hasta que todos los hilos lleguen a ese punto. Para definir una barrera se utiliza la siguiente sentencia:

#pragma omp barrier

- *Critical*

En la sincronización crítica solo un hilo a la vez puede ingresar a la región marcada. Para definir una región crítica se utiliza la siguiente sentencia:

#pragma omp critical

- *Atómic*

La sincronización atómica provee una exclusión mutua, pero solo aplica para la actualización de una localidad de memoria. Para definir una región atómica se utiliza la siguiente sentencia:

#pragma omp atomic

- Nowait

Cuando se establece una barrera, los hilos deben esperar la ejecución de todos para realizar el siguiente proceso, sin embargo, la clausula nowait permite generar una excepción a esa regla, es decir, le dice al compilador, para ejecutar el siguiente código, no es necesario esperar a los otros hilos.

#pragma omp for nowait

- Constructor Master

El constructor master delimita el bloque de código que solo es ejecutado por el hilo principal, es decir, ese código sólo puede ser ejecutado una vez y debe ser ejecutado por

el hilo 0. Los otros hilos simplemente saltan ese bloque, sin necesidad de establecer algún tipo de sincronización.

- Constructor single

El constructor single delimita un bloque de código que sólo es ejecutado por un único hilo (este hilo no es necesariamente el hilo principal). Al final del constructor single se establece una barrera de manera implícita, la cual se puede quitar (como ya se comentó) con la clausula nowait.

- Constructor de Secciones

El constructor sections proporciona una manera de segmentar un bloque estructurado (seccionado), de tal manera que cada bloque sea ejecutado por un hilo distinto. Por defecto, existe una barrera al final del bloque sections, la cual se puede quitar con la clausula nowait.

✓ Funciones de bloqueo

Un bloqueo o lock es similar a una sección crítica, ya que garantiza que una instrucción sea ejecutada por un solo proceso a la vez. La principal diferencia entre un bloqueo y una sección crítica es que lock protege la información, no el código. Con un bloqueo se asegura que la información solo es accedida por un solo proceso a la vez.

Los códigos de bloqueo pueden ser muy peligrosos, ya que, si no se sincroniza correctamente, se puede generar lo que se conoce como deadlock (bloqueo de la muerte o abrazo mortal).

Un abrazo mortal o deadlock sucede cuando un hilo A se bloquea esperando la ejecución de otro hilo B, el cuál, a su vez, se bloquea esperando la ejecución del primero, debido a que ninguno termina su proceso, el bloqueo se mantienen impidiendo el flujo del programa.

*Coherencia y Consistencia*

El modelo de memoria de OpenMP se basa en la memoria compartida entre los hilos en ejecución. Cuando un procesador accede a un valor de la memoria compartida, puede crear una copia de ese valor en su caché local. Sin embargo, cuando otro procesador necesita trabajar con el mismo valor, no siempre es seguro si accederá al valor actualizado en la memoria compartida o a una copia obsoleta en la caché. Para asegurar que un programa funcione correctamente y procese la información de manera coherente, es necesario utilizar mecanismos que controlen y aseguren el acceso correcto a los valores. Un modelo de memoria se define por dos aspectos clave:

Coherencia de la memoria: Se refiere a cómo el sistema gestiona el acceso a una misma localidad de memoria por varios hilos.

Consistencia de la memoria: Se refiere al orden en que se realizan las operaciones de lectura, escritura y sincronización (RWS) entre varias localidades de memoria e hilos.

Aunque el programador define un orden específico en el código, el compilador puede reordenar algunas instrucciones para mejorar el rendimiento. Sin embargo, en la ejecución paralela, este orden optimizado puede variar, afectando la ejecución del programa. Un multiprocesador puede tener consistencia secuencial, donde el orden de las operaciones RWS se respeta en cada procesador, lo que, aunque correcto, puede impactar negativamente el rendimiento. Alternativamente, existe la consistencia relajada, que permite cierta flexibilidad en el orden de las operaciones para mejorar la eficiencia.

- *Directiva flush*

La directiva flush en OpenMP obliga a que los valores de las variables de un hilo se escriban en la memoria principal, y permite que los demás hilos lean los valores actualizados desde esa memoria, en lugar de desde la caché local de cada procesador. Esto es crucial en entornos paralelos, ya que, sin esta operación, las actualizaciones hechas por un hilo podrían no ser visibles para otros hilos.

El uso de flush en OpenMP es esencial para mantener la coherencia de la caché y la consistencia de la memoria, asegurando que todos los hilos trabajen con la versión más reciente de las variables compartidas.

**Conclusiones**

La programación paralela en C con OpenMP es una poderosa herramienta que permite optimizar el rendimiento de aplicaciones al distribuir el trabajo entre varios procesadores o núcleos. A medida que la complejidad de los problemas crece, la programación secuencial empieza a mostrar limitaciones, lo que ha llevado al desarrollo de procesadores multinúcleo y la necesidad de aprovechar paradigmas como el paralelismo. OpenMP facilita este proceso mediante directivas sencillas que permiten paralelizar código escrito en C, gestionando el uso de múltiples hilos que trabajan en conjunto para resolver tareas de manera más eficiente.

Al emplear OpenMP, el programador puede dividir tareas complejas en bloques más pequeños y ejecutarlos de forma simultánea en diferentes hilos, optimizando el tiempo de ejecución y el uso de recursos del sistema. Sin embargo, no todos los algoritmos son adecuados para la paralelización, y es importante identificar aquellos en los que el paralelismo será beneficioso. Con OpenMP, las tareas compartidas entre hilos pueden gestionarse fácilmente, pero también es crucial manejar correctamente la sincronización para evitar problemas como condiciones de carrera o bloqueos.

Referencias


- http://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf
- http://www.openmp.org/
- Introduction to Algorithms.Thomas H. Cormen, Charles E.Leiserson, Ronald L. Rivest, Clifford Stein, McGraw-Hill.