Práctica 4. Programación multihilos con OpenMP  
Arquitectura de Computadoras, ITAM[[1]](#footnote-1)  
Mayo 2017

## Introducción

A la ejecución de cada secuencia de instrucciones se le conoce como un hilo de procesamiento o thread. Cada hilo posee sus propios registros de control y su propia pila de datos, mientas que comparte el uso de la memorias del *heap* y *data* con los demás hilos del programa. Fundamentalmente un programa o proceso se compone de uno o más hilos, y la programación multithreading se refiere a utilizar explícitamente varios hilos simultáneamente para acelerar la ejecución de un programa.

Para esto existen entornos como OpenMP, el cual es un API que sirve para paralelizar programas en C, C++ o Fortran y busca que escribir código en paralelo sea sencillo. Se basa en un esquema de paralelización para computadoras de memoria compartida (todos los procesadores acceden a la misma memoria) y su funcionamiento interno es con hilos, en el caso de sistemas POSIX (UNIX / Linux / OS X) utiliza internamente la librería de POSIX-Threads (libpthread).

Algunos compiladores que soportan OpenMP:

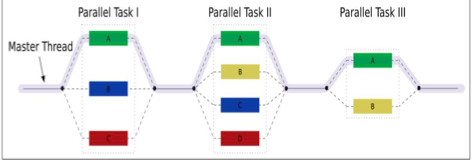
* GNU Compiler Collection GCC (versión >= 4.3.2)
* Intel Compiler Suite (version >= 10.1)
* Microsoft Visual Studio 2008 (Professional)
* Microsoft Visual Studio 2008 (Express) + Windows SDK for Windows Server 2008

La lista completa se puede encontrar en: <http://openmp.org/wp/openmp-compilers>

OpenMP se compone de variables de ambiente, librerías de funciones y directivas de compilación. Funciona indicando a los compiladores como generar códigos paralelos (multithreaded), mediante directivas incorporadas por los programadores que indican cómo un trabajo será dividido entre los hilos. De esta manera lo que busca es que las aplicaciones puedan aprovechar las capacidades de las máquinas con múltiples procesadores actuales.

Una directiva de compilador consiste en una línea de código con significado especial para ciertos compiladores (e ignorado por los demás compiladores). Por ejemplo en C/C++ las directivas de OpenMP son identificadas por **#pragma omp**. El código que se encuentra delimitado por las directivas de OpenMp se conoce como regiones paralelas.

La paralelización en OpenMP es realizada usando múltiples hilos dentro de un mismo proceso. Cada hilo pose su propia pila de ejecución y tiene la posibilidad de compartir memoria con otros hilos dentro del mismo proceso. La ejecución comienza con un sólo hilo que ejecuta las instrucciones hasta encontrar una región paralela. Al encontrar esa región se crea un grupo de hilos que ejecutan el código dentro de ésta, que al final de la región se sincronizan, es decir la ejecución de la región paralela termina sólo hasta que todos los hilos también terminan. Este modelo se llama *fork-join*, y se encuentra ejemplificado en la siguiente figura:



La sintaxis para los componentes de OpenMP son:

Variables de ambiente:

OMP\_NOMBREDELAVARIABLE

Directivas de compilación:

#pragma omp directiva

Funciones adicionales:

omp\_funcion()

Para mostrar cómo usar OpenMP veamos un ejemplo sencillo de código, en particular ejemplifica la paralelización for de OpenMP:

void Suma(double\* a, double\* b, double\* c, int size){

for (int i = 0; i < size; ++i){

c[i] = a[i] + b[i];

}

}

La característica más importante de este código es que cada iteración del for es independiente, es decir no depende del resultado de otras iteraciones, por lo tanto su paralelización es muy simple. Con OpenMP usamos la directiva para paralelizar for:

void Suma(double\* a, double\* b, double\* c, int size){

#pragma omp parallel for

for (int i = 0; i < size; ++i){

c[i] = a[i] + b[i];

}

}

Para compilar con GCC es necesario agregar:

gcc -o programa -fopenmp programa.c

En Visual C++ hay una opción en las preferencias del proyecto para activar OpenMP. Para comprender el funcionamiento del código anterior, supongamos que size = 30; si la computadora tiene tres procesadores, el código se ejecutaría como:

#pragma omp parallel for

for(inti=0;i<10;++i) for(inti=10;i<20;++i) for(inti=20;i<30;++i)

{ { {

c[i] = a[i] + b[i]; c[i] = a[i] + b[i]; c[i] = a[i] + b[i];

} } }

Por omisión el número de hilos es igual al número de núcleos en la computadora, sin embargo, se puede establecer por medio de variables de ambiente o también se puede definir en el código en ejecución.

Para determinar el número de hilos con variables de ambiente se modifica la variable OMP\_NUM\_THREADS:

En Linux/OSX/Unix (Bash): export OMP\_NUM\_THREADS=3

En Windows: set OMP\_NUM\_THREADS=3

Para modificar el número de hilos en el código es necesario incluir el header <omp.h>, el cual es uno de los componentes de la librería de OpenMP que incluye varias funciones para el control de hilos:

#include <omp.h>

int threads = 3;

void Suma(double\* a, double\* b, double\* c, int size){

omp\_set\_num\_threads(threads);

#pragma omp parallel for

for (int i = 0; i < size; ++i) {

c[i] = a[i] + b[i];

}

}

Otras funciones muy útiles en omp.h son omp\_get\_thread\_num, que regresa el número de hilo actual y permite tener mayor control sobre los cálculos realizados, y omp\_get\_num\_threads que regresa el número total de hilos. Por ejemplo, para utilizar OpenMP pero modificando la paralelización automática de for se puede hacer:

void Suma(double\* a, double\* b, double\* c, int size){

#pragma omp parallel

int id = omp\_get\_thread\_num();

int num\_its = size/omp\_get\_num\_threads();

int start\_it = id\*num\_its;

for (int i = start\_it; i < (start\_it+num\_its); ++i){

c[i] = a[i] + b[i];

}

}

Por omisión, en el ejemplo que vimos, OpenMP considera que a, b y c son variables compartidas (todos los hilos pueden acceder/modificar), y el iterador del for es una variable privada. También es posible decidir de forma explícita qué variables queremos que tengan un comportamiento privado o compartido:

int i;

int j[10];

float\* a;

float\* b;

#pragma omp parallel private(i,j) shared(a,b) {

int k;

// ... codigo ...

}

Una directiva muy importante es #pragma omp critical, la cual garantiza acceso exclusivo a un solo hilo, a una región del código.

Para controlar la asignación de hilos a núcleos de procesador, gcc provee la variable de ambiente GOMP\_CPU\_AFFINITY. Su funcionalidad es muy sencilla, por ejemplo si tenemos 4 hilos podemos controlar su ejecución de la siguiente manera:

GOMP\_CPU\_AFFINITY="0 3 1-2"

Esto asigna el primer hilo al núcleo 0, el segundo al 3, el tercero al 1 y el cuarto al 2.

Nuevos estándares de OpenMP tienen la variable OMP\_PROC\_BIND, un booleano que controla si los hilos los puede cambiar de núcleo el programa en su ejecución.

## Ejercicios

### Introducción: Hello Worlds

Compile el programa HelloWorlds.c y ejecútelo

Despliegue el resultado

Edite el código fuente y agregue una directiva omp parallel que incluya las cuatro primeras líneas, pero deje fuera la última línea: printf(“GoodBye…

#pragma omp parallel

{

...[Code to run in Parallel goes here]...

}

¿Necesita incluir la cabecera omp.h? ¿Por qué?

Compile el programa. Si el compilador reporta errores, corríjalos.

Modifique la variable de ambiente OMP\_NUM\_THREADS para definir seis hilos.

Compile el programa y ejecútelo varias veces. Observe si el resultado es el mismo en todos los casos

¿Qué puede deducir de los resultados observados?

### Cálculo de una función por integración numérica

Abra el archivo demoReducción.c y explique brevemente qué hace

Compile y ejecute el programa

¿Cuál fue el tiempo de ejecución? ¿Cuál fue el error?

Identifique la sección del código que puede ser paralelizable (el segundo for, donde se ejecuta la mayor parte del cálculo) y agregue la directiva correspondiente

#pragma omp parallel for

{

...[Código paralelo]...

}

Compile y ejecute el programa

¿Cuál fue el tiempo de ejecución? ¿Cuál fue el error?

Dependiendo del procesador, quizás habrá identificado una fuerte reducción en el tiempo de procesamiento. Lo que no pasa desapercibido, es que el error se incrementó sustancialmente pues estamos accediendo concurrentemente a la suma de las áreas en la variable llamada *integral*.

Cree dentro del for una variable local *d\_local* y asígnele el cálculo de 1/sqrt(x)\*dx. Posteriormente, declare una sección crítica para sumar esta variable local a la variable compartida *integral*. La esencia del código es ésta:

const double d\_local =1.0/sqrt(x) \* dx;

#pragma omp critical

{

integral += d\_local;

}

DIVIDA AL MENOS EN 100 EL NÚMERO DE ITERACIONES (la variable *nSteps).* Compile y ejecute el programa

**Aproximadamente,** ¿Cuál fue el tiempo de ejecución? ¿Cuál fue el error?

Considere que a los resultados obtenidos los debe ajustar por la reducción en el número de pasos

Dado que la operación de agregación es muy sencilla, cambie la declaración de la sección crítica por una declaración de operación atómica (#pragma omp atomic)

**Aproximadamente,** ¿Cuál fue el tiempo de ejecución? ¿Cuál fue el error?

Considere que a los resultados obtenidos los debe ajustar por la reducción en el número de pasos

La acumulación en la variable integral es simplemente una operación de reducción, así que el código se puede optimizar aún más. Simplemente regrese al código original (incluyendo el número de iteraciones del for) y agregue a la declaración para paralelizar el for, que realizará una reducción de suma sobre la variable *integral.*

Muestre el código resultante

¿Cuál fue el tiempo de ejecución? ¿Cuál fue el error?

Compile y ejecute el programa con diferentes valores para la variable de ambiente OMP\_NUM\_THREADS

Muestre una tabla con el número de hilos y el tiempo de ejecución obtenido.

¿Qué puede deducir de los resultados esperados?

### Cálculo de números primos

### Ejecución secuencial

En esta sección se buscarán números primos dentro de un rango de enteros con un algoritmo de “fuerza bruta” que se encuentra en el archivo “primos\_serial.c”. Se divide el candidato potencial entre sus posibles factores. Si existe el factor, entonces el número no es primo. Los primos encontrados se almacenan en un arreglo.

El programa despliega un indicador de avance que se actualiza con cada incremento de 10% de los números procesados. Finalmente, el programa también despliega el tiempo de ejecución.

Compile y ejecute el programa.

¿Cuántos primos encontró en el rango [1..10,000,000]?

¿Cuánto tiempo tardó el programa?

### Ejecución paralela

Modifique el programa para paralelizar el ciclo dentro de la rutina FindPrimes.

Compile y ejecute el programa varias veces.

En promedio, ¿Cuántos primos encontró en el rango [1..10,000,000]?

En promedio, ¿Cuánto tiempo tardó el programa?

En general, deberá haber observado diferencias en el número de primos encontrados. Esto se debe a que hay variables compartidas por los hilos en el código, lo que ocasiona condiciones de carrera.

Identifique las regiones de código en las que estas condiciones de carrera se dan, y protéjalas insertando directivas #pragma omp critical (Ayuda: típicamente hay dos secciones, pero solo una de ellas afecta el resultado)

Compile y ejecute el programa varias veces para comprobar que el resultado no cambia y es igual a la versión serie del código

Para una mayor precisión al computar el tiempo de ejecución en OpenMP, modifique el código para cambiar los llamados a la función clock() por omp\_get\_wtime(). Observe que esta última devuelve un double y que ya solo es necesario calcular la diferencia de valores sin dividir por CLOCKS\_PER\_SEC.

Compile y ejecute el programa para un rango de valores de [1..10,000,000], variando el número de hilos de 1 a 8.

Muestre una tabla con los resultados obtenidos y coméntelos

## 

1. Práctica realizada con apoyo de Dante Gama y de los tutoriales de Intel [↑](#footnote-ref-1)