PRÁCTICA 3 – SISTEMAS EMPOTRADOS

Entrenamiento con ARM: OpenMP y OpenCV

Lucas Serrano Jiménez

César San Blas Leal

tarea 1.1

**Se deberá estudiar el API de OpenMP y su uso con GNU GCC (gcc, g++), comprobando el correcto funcionamiento de algunos de los ejemplos que hay disponibles en Internet.**

La API de OpenMP es una biblioteca de programación que permite a los programadores paralelizar sus aplicaciones (basados en C, C++ o Fortran) de forma explícita; es decir, el programador debe especificar qué partes del código deben ejecutarse en paralelo. El compilador y el sistema de ejecución se encargarán de distribuir el trabajo entre los distintos procesadores.

Esta API no realiza comprobaciones de ningún tipo en el programa, por lo tanto es responsabilidad del programador asegurarse de que la aplicación sea correcta y segura.

GNU GCC es un compilador de código abierto que soporta OpenMP. Para compilar un programa que utiliza OpenMP en GCC, se debe utilizar la opción -fopenmp.

Algunas de las directivas y cláusulas más comunes de OpenMP son las siguientes:

* **Directiva parallel:** Declara una región de código que se ejecutará en paralelo.
* **Directiva for:** Declara un bucle que se ejecutará en paralelo.
* **Cláusula private:** Declara que una variable solo será visible para el hilo que la creó.
* **Cláusula shared:** Declara que una variable será visible para todos los hilos.
* **Cláusula reduction:** Declara que una variable será compartida por todos los hilos y que su valor será la suma de los valores individuales.

A continuación, un sencillo programa de ejemplo que muestra el uso de algunas de estas directivas y cláusulas:

#include <stdio.h>

#include <omp.h>

int main() {

  int i, n = 1000;

  float sum = 0.0f;

  // Sección en paralelo

  #pragma omp parallel

  {

//`sum` será visible para todos los hilos

#pragma omp for

for (i = 0; i < n; i++) {

   sum += i;

}

  }

  printf("sum = %f\n", sum);

  return 0;

}

Cuyo resultado muestra el valor final de *sum* como suma de todos los bucles realizados en paralelo, aunque cada ejecución termina con un valor distinto. Esto se debe a que cada hilo tiene su propia variable *sum* y al final se juntan todas, produciendo variaciones porque no hay un orden establecido. Si se declarase *sum* como *shared*, escribiendo *#pragma omp for reduction(+:sum)* antes del *for* compartido, el resultado se estabiliza en: *sum = 499500.00.*

tarea 1.2

**1. Explique qué hace el código que se añade a continuación.**

El código expuesto en el guión de la práctica suma los valores de los arrays a y b, previamente inicializados secuencialmente de 0 a N-1. Lo característico del código es que utiliza OpenMP para ejecutar las sumas en paralelo, distribuyendo la carga de trabajo entre varios hilos.

**2. Delimite las distintas regiones OpenMP en las que se está expresando paralelismo.**

Las regiones donde se expresa paralelismo en el código implementado son:

* #pragma omp parallel shared(a, b, c, nthreads, chunk) private(i, tid)

Esta línea comienza la región paralela con OpenMP. En ella se establecen las variables que se mantendrán inalteradas entre hilos (*i* y *tid*) y las que se compartirán (*a, b, c, nthreads* y *chunk*).

* #pragma omp for schedule(dynamic,chunk)

Esta directiva divide el bucle for en tareas que se ejecutarán en paralelo entre los hilos.

**3. Explique con detalle qué hace el modificador de OpenMP *schedule*. ¿Cuáles pueden ser**

**sus argumentos? ¿Qué función tiene la variable *chunk* en el código? ¿A qué afecta?**

El operador *shedule* sirve para configurar la distribución de las iteraciones del bucle for entre los hilos paralelos.

El valor de chunk sirve para especificar cuántas iteraciones se asignan a cada hilo. En este caso, este valor viene definido por la constante CHUNKSIZE que vale 10, por lo que por cada bloque atenderá a 10 iteraciones.

**4. ¿Qué función tiene el modificador de OpenMP dynamic en el código?**

Establece que las iteraciones se asignan a medida que los hilos terminan su trabajo anterior, lo que ayuda a equilibrar la carga de trabajo de manera más eficiente si algunas iteraciones toman más tiempo que otras y a evitar que hayan hilos inactivos.

**5. Investigue qué pasa si no declara como privadas las variables *i* y *tid*.**

Si la variable *i* no fuese privada, se modificaría simultáneamente por todos los hilos que estén trabajando con ella por lo que el bucle for no funcionaría correctamente. En el caso de *tid,* si dentro del hilo el valor pudiera ser alterado mientras se ejecuta, al mostrar por pantalla el resultado de la suma el identificador del hilo sería erróneo.

RETO JEDI 1

A screen shot of a computer program

Description automatically generatedRepita la tarea 1.2 empleando un planificador estático y no dinámico. Puede hacerlo “a mano” o usando algún modificador de OpenMP.

El programa difiere respecto al código del guón de la práctica en que la asignación de iteraciones del bucle for a los hilos se hace de manera estática en lugar de dinámica. Esto se consigue sustituyendo el argumento *dynamic* de *schedule* por *static,* lo que implica que cada hilo se encarga exclusivamente del número de iteraciones establecido en la variable chunk. A diferencia del modo dinámico, donde las iteraciones se asignaban a medida que los hilos iban terminando su trabajo, para este modo cada hilo realizará únicamente el número de iteraciones que dicte chunk. En este caso la variable vale 10, por lo que cada hilo realizará 10 iteraciones, como se muestra en la figura.

tarea 1.3

**Diseñe un programa en C/C++ para multiplicar dos matrices cuadradas con elementos de tipo double(punto flotante de 64 bits) entre 0 y 1. Después multiplique la matriz resultante por otra matriz de enteros entre 0 y 255 elemento a elemento.**

El código implementado se encuentra en el archivo *tarea1\_3.cpp*.

**Muestre en una serie de gráficas cómo varía el tiempo paralelo de ejecución en función de algún parámetro que modifique la carga computacional del problema. Luego, paralelice el código con OpenMP y tome los siguientes tiempos de ejecución: código secuencial original (sin paralelizar), código paralelo ejecutado en mono mono-hebra y código con dos o más hebras. Apunte las ganancias en velocidad que se observan.**

En este caso se ha empleado N, que define las dimensiones de las matrices cuadradas (NxN) usadas en la operación, como parámetro para observar la variación del tiempo de ejecución y el comportamiento según el tipo de paralelismo. Puesto que el código desarrollado contiene directivas de OpenMP para el paralelismo, la compilación requiere del argumento -fopenmap, como ya se advirtió en la tarea 1.1. En este ejercicio se ha ido incrementando el valor de N de 10 en 10 hasta llegar a 600, obteniendo un número significativo de muestras, y midiendo el tiempo, la ganancia y la eficiencia de funciones con paralelismo monohebra, bihebra y multihebra (cuatro en este caso). A continuación se muestran los resultados obtenidos.

A group of text on a screen

Description automatically generated with medium confidence

Los resultados obtenidos, para graficarlos posteriormente, se han guardado en el fichero *tarea1\_3.time* empleado una tubería*.* Después, con el código *tarea1\_3.gpi* implementado y mediante la herramienta Gnuplot se han graficado estos valores.



La primera gráfica recoge los tiempos medidos para cada caso de ejecución. Como es de esperar, la ejecución con 4 hebras presenta un tiempo de ejecución menor que en el resto de casos. Además, la ejecución secuencia y la monohebra no muestra diferencia alguna, indicando que el procedimiento es el mismo.

A graph of different colored lines

Description automatically generated

La siguiente gráfica muestra la ganancia obtenida por parte de los diferentes modos de paralelización respecto a la función secuencial. Como se observó en la anterior gráfica, el modo monohebra no encuentra mejora respecto al secuencial, por lo que su ganancia se encuentra en valores próximos al 1. Por otro lado, la función con más hebras muestra una ganancia superior al resto, proporcional a la mejora en el tiempo de ejecución.

A graph of different colored lines

Description automatically generated

La última gráfica presenta los resultados de la eficiencia en función del número de hebras. En este caso, a diferencia de las dos gráficas anteriores, la paralelización que domina es la de una hebra (equivalente a secuencial). Destaca la poca eficiencia que presenta el modelo de paralelización de cuatro hebras. Esto indica que la ganancia en velocidad no es proporcional al número de hebras.

A graph of different colored lines

Description automatically generated

RETO JEDI 2

Si la matriz B es transpuesta, el código que produce el compilador para multiplicar las dos matrices A y B producirá menos fallos de caché, dado que maximizamos la localidad espacial de los datos en la cachés. Intente demostrar y medir si este hecho afecta a su problema. Nota: puede que tenga que escalar el problema para que el efecto sea apreciable.

Se ha añadido una nueva función *operacionMatrizTraspuesta,* que realiza la misma operación requerida en la tarea anterior pero trasponiendo la matriz B antes de realizar la multiplicación. Los resultados temporales se muestran en la siguiente gráfica donde la línea lila representa el código base secuencial y el verde el que añade la matriz traspuesta. A pesar de haber escalado considerablemente el tamaño del problema (nótese que se ha ampliado hasta matrices de 800x800), no se detecta ninguna diferencia con respecto al programa que no traspone la matriz.

A graph of a number of points

Description automatically generated with medium confidence