Práctica 4

INTRODUCCIÓN A OPENMP Y PREPARACIÓN DEL ENTORNO

Jordi Blasco Lozano
Computación de alto rendimiento
Grado en Inteligencia Artificial

Indice:

1. INSTALACIÓN Y CONFIGURACIÓN

Para instalar el entorno de la practica he utilizado una imagen Docker que ya tenia de Ubuntu con gcc y más paquetes que nos servirán para compilar y ejecutar programas de c y c++ haciendo uso de la paralelización que necesitemos. Tengo un .sh dentro de la carpeta usr/local/bin que usa esta imagen para generar un entorno de desarrollo que copie todo el directorio en el cual se ejecuta el comando para programar con el entorno correcto. Tengo mac y por eso he decidido usar Docker.

```
jordiblascolozano@MacBook-Pro practica4 % runUbuntu
root@bc2101d7e622:/workdir# ls
'SEMANA 4_ INTRODUCCIÓN A OPENMP Y PREPARACIÓN DEL ENTORNO_RECUPERADO.pdf'
 codigo.cpp
 main.c
 memoria_02.docx
 programa
'~$moria_02.docx'
root@bc2101d7e622:/workdir#
```

Solo he tenido que ejecutar el .sh para configurar el entorno de desarrollo que vamos a utilizar, no he tenido errores ya que la imagen docker actualiza el paquete apt usando el comando
‘sudo apt update && sudo apt upgrade -y’ por defecto cuando se ejecuta.
Para programar he decidido usar el editor de código de zed el cual uso normalmente para programar en c y c++ por su simpleza y baja latencia.

2. PROGRAMACIÓN CON OPENMP

Ejercicio 1:
En el primer ejercicio, hemos implementado un programa que imprime “hola mundos” desde múltiples hilos en paralelo, indicando el número de cada hilo. A partir de la salida, podemos concluir que se han utilizado 16 hilos, y que cada uno se ejecuta en paralelo sin seguir un orden específico. Esto podría parecer un error si estuviéramos trabajando con programación secuencial, donde se espera un orden predefinido. Sin embargo, en programación paralela, los hilos no siguen un orden fijo; la ejecución se distribuye entre los hilos y el más rápido en completar su tarea imprime su mensaje primero. Por lo tanto, el comportamiento observado es normal y esperado en un entorno de programación paralela.

```
compilation terminated.
root@0f82bd6bf4f0:/workdir# gcc -fopenmp main.c -o test_openmp && ./test_openmp
Hola desde el hilo 3
Hola desde el hilo 15
Hola desde el hilo 1
Hola desde el hilo 9
Hola desde el hilo 8
Hola desde el hilo 14
Hola desde el hilo 4
Hola desde el hilo 7
Hola desde el hilo 10
Hola desde el hilo 6
Hola desde el hilo 13
Hola desde el hilo 0
Hola desde el hilo 12
Hola desde el hilo 11
Hola desde el hilo 5
Hola desde el hilo 2
```

Ejercicio 2:

En el segundo ejercicio, desarrollamos un programa que inicializa un mapa de celdas. Cada celda contiene un número aleatorio de farolas y un consumo total calculado en función de cada farola. Posteriormente, se computa el total de farolas y el consumo total tanto de forma secuencial como paralela utilizando OpenMP. Aunque ambos métodos arrojan resultados numéricos idénticos, se observó que, en problemas de tamaño reducido, la versión paralela puede resultar más lenta debido al sobrecosto asociado a la creación y sincronización de hilos. Este comportamiento es esperado, ya que el beneficio del paralelismo solo se hace evidente cuando la carga de trabajo es lo suficientemente grande para amortizar dichos costos.

Adicionalmente, para evaluar el rendimiento en un escenario con mayor carga, se aumentó el tamaño del mapa de 200x200 celdas a 2000x2000 celdas. Con este incremento en el tamaño del problema, se obtuvieron los siguientes resultados de tiempo:

200 x 200	*secuencial 4,7 veces más rápida*

```
root@0f82bd6bf4f0:/workdir# ./programa
Secuencial - Total Farolas: 10956188, Consumo Total: 1954180391, Tiempo: 0.00049531 segundos.
Paralelo   - Total Farolas: 10956188, Consumo Total: 1954180391, Tiempo: 0.0023464 segundos.
```

2000 x 2000	*paralelo 6 veces más rápida*

```
root@0f82bd6bf4f0:/workdir# ./programa
Secuencial - Total Farolas: 1099585194, Consumo Total: 196094285409, Tiempo: 0.0288016 segundos.
Paralelo   - Total Farolas: 1099585194, Consumo Total: 196094285409, Tiempo: 0.00474135 segundos
```

Es importante destacar que, al aumentar el tamaño del problema, se puede apreciar de forma más clara la ventaja de la paralelización, ya que el tiempo invertido en la creación y coordinación de hilos se ve amortiguado por la mayor carga de trabajo. Sin embargo, este beneficio solo se vuelve significativo cuando las operaciones por iteración son lo suficientemente complejas o cuando el número de iteraciones es muy alto. En este caso, la versión paralela mostró una mejora considerable en el tiempo de ejecución comparada con la versión secuencial, lo que confirma que, para problemas de mayor magnitud, la paralelización se convierte en una herramienta poderosa para optimizar el rendimiento.

3. CÓDIGO COMENTADO EJERCICIO 2

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <omp.h>
using namespace std;

// Definición de constantes para configurar el tamaño del mapa y los rangos de valores
const int MAP_SIZE = 200;                  // Tamaño del mapa: 200 x 200 celdas
const int MIN_FAROLAS = 50, MAX_FAROLAS = 500; // Rango aleatorio para la cantidad de farolas en cada celda
const int BAJO_MIN = 70, BAJO_MAX = 100;       // Rango de consumo para farolas de tipo bajo
const int MEDIO_MIN = 150, MEDIO_MAX = 200;    // Rango de consumo para farolas de tipo medio
const int ALTO_MIN = 250, ALTO_MAX = 300;      // Rango de consumo para farolas de tipo alto

// Estructura que representa una celda en el mapa
struct Celda {
    int num_farolas;    // Número de farolas en la celda
    int consumo_total;  // Consumo total acumulado de todas las farolas en la celda
};

// Función que inicializa el mapa con valores aleatorios para cada celda
void inicializarMapa(vector<vector<Celda>> &mapa)
{
    srand(time(0));  // Inicializa la semilla para la generación de números aleatorios
    for (int i = 0; i < MAP_SIZE; i++)
    {
        for (int j = 0; j < MAP_SIZE; j++)
        {
            // Asigna un número aleatorio de farolas a la celda
            mapa[i][j].num_farolas = rand() % (MAX_FAROLAS - MIN_FAROLAS + 1) + MIN_FAROLAS;
            // Calcula y suma un consumo aleatorio para cada farola según su tipo
            for (int k = 0; k < mapa[i][j].num_farolas; k++)
            {
                int tipo = rand() % 3; // Selecciona aleatoriamente el tipo de farola
                if (tipo == 0)
                    mapa[i][j].consumo_total += rand() % (BAJO_MAX - BAJO_MIN + 1) + BAJO_MIN;
                else if (tipo == 1)
                    mapa[i][j].consumo_total += rand() % (MEDIO_MAX - MEDIO_MIN + 1) + MEDIO_MIN;
                else
                    mapa[i][j].consumo_total += rand() % (ALTO_MAX - ALTO_MIN + 1) + ALTO_MIN;
            }
        }
    }
}
```

```
// Función que calcula de forma secuencial el total de farolas y el consumo total del mapa
void calcularConsumoSecuencial(const vector<vector<Celda>> &mapa, long long &total_farolas, long long &consumo_total)
{
    total_farolas = 0;
    consumo_total = 0;
    for (int i = 0; i < MAP_SIZE; i++)
    {
        for (int j = 0; j < MAP_SIZE; j++)
        {
            total_farolas += mapa[i][j].num_farolas;
            consumo_total += mapa[i][j].consumo_total;
        }
    }
}

// Función que calcula en paralelo el total de farolas y el consumo total del mapa
void calcularConsumoParalelo(const vector<vector<Celda>> &mapa, long long &total_farolas, long long &consumo_total)
{
    total_farolas = 0;
    consumo_total = 0;
    // Paraleliza el bucle externo con reducción para acumular sumas correctamente
    #pragma omp parallel for reduction(+:total_farolas, consumo_total) schedule(dynamic)
    for (int i = 0; i < MAP_SIZE; i++)
    {
        for (int j = 0; j < MAP_SIZE; j++)
        {
            total_farolas += mapa[i][j].num_farolas;
            consumo_total += mapa[i][j].consumo_total;
        }
    }
}

int main()
{
    // Crea un mapa de celdas de tamaño MAP_SIZE x MAP_SIZE, inicializado a {0, 0}
    vector<vector<Celda>> mapa(MAP_SIZE, vector<Celda>(MAP_SIZE, {0, 0}));

    // Inicializa el mapa con valores aleatorios
    inicializarMapa(mapa);

    long long total_farolas_seq, consumo_total_seq;
    double start_seq = omp_get_wtime(); // Tiempo de inicio del cálculo secuencial
    calcularConsumoSecuencial(mapa, total_farolas_seq, consumo_total_seq);
    double end_seq = omp_get_wtime(); // Tiempo final del cálculo secuencial
    double tiempo_seq = end_seq - start_seq; // Tiempo transcurrido

    // Imprime los resultados del cálculo secuencial
    cout << "Secuencial - Total Farolas: " << total_farolas_seq
         << ", Consumo Total: " << consumo_total_seq
         << ", Tiempo: " << tiempo_seq << " segundos." << endl;

    long long total_farolas_par, consumo_total_par;
    double start_par = omp_get_wtime(); // Tiempo de inicio del cálculo paralelo
    calcularConsumoParalelo(mapa, total_farolas_par, consumo_total_par);
    double end_par = omp_get_wtime(); // Tiempo final del cálculo paralelo
    double tiempo_par = end_par - start_par; // Tiempo transcurrido

    // Imprime los resultados del cálculo paralelo
    cout << "Paralelo   - Total Farolas: " << total_farolas_par
         << ", Consumo Total: " << consumo_total_par
         << ", Tiempo: " << tiempo_par << " segundos." << endl;

    return 0;
}
```