Practica 4
Introducción a OpenMP y preparación del entorno

Jordi Blasco Lozano
Universidad de Alicante

10 de septiembre de 2025

Resumen

Este informe detalla una exploración práctica de la computación paralela utilizando la API de OpenMP. El trabajo abarca la configuración inicial de un entorno de desarrollo mediante una imagen de Docker con Ubuntu y GCC. El núcleo de la práctica consiste en la implementación y el análisis de tres ejercicios distintos diseñados para ilustrar los principios y ventajas de la programación paralela.

Índice

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

2.1. 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.

Bloque 1: Salida de la ejecución del Ejercicio 1.

```
root@566c8d7af7c1:/workdir# ./ejec_ejercicio1
Hola desde el hilo 1 de 16
Hola desde el hilo 2 de 16
Hola desde el hilo 8 de 16
Hola desde el hilo 6 de 16
Hola desde el hilo 9 de 16
Hola desde el hilo 15 de 16
Hola desde el hilo 5 de 16
Hola desde el hilo 3 de 16
Hola desde el hilo 14 de 16
Hola desde el hilo 4 de 16
Hola desde el hilo 11 de 16
Hola desde el hilo 0 de 16
Hola desde el hilo 12 de 16
Hola desde el hilo 10 de 16
Hola desde el hilo 13 de 16
Hola desde el hilo 7 de 16
```

2.2. Ejercicio 2

En este ejercicio, se usa OpenMP para hacer más rápida la suma de los elementos de una lista. Primero, definimos un número N=1000 y un arreglo A (lista) de tipo double (números decimales con 2 veces más decimales que los float normales), inicializado con valores de 0.00 a N.

Luego, usamos OpenMP con la cláusula `reduction(+:suma)`, que permite paralelizar el cálculo, consiguiendo que varios hilos sumen partes del arreglo al mismo tiempo y luego combinen los resultados sin errores. Así, el programa aprovecha mejor los recursos del procesador.

Al final, el programa muestra en pantalla la suma total. Aunque el beneficio en rendimiento es limitado para un tamaño pequeño de datos, el uso de OpenMP es crucial en tareas con mayor carga computacional, donde la paralelización mejora sigificativamente la eficiencia.

Bloque 2: Salida de la ejecución del Ejercicio 2.

```
root@566c8d7af7c1:/workdir# ./ejec_ejercicio2
Suma total: 499500.000000
```

2.3. Ejercicio 3

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. Cambie la configuracion de la parelizacion al igual que la cantidad de hilos implicados para sacar diferentes estadisticas:

Bloque 3: Salida de la ejecución del Ejercicio 3 completo

```
---------- Mapa tamaño: 200 ----------
Secuencial -> | Tiempo: 0.00033140 s
Paralelo -> Schedule: static | Hilos: 4 -> | Tiempo: 0.00031300 s
Paralelo -> Schedule: static | Hilos: 8 -> | Tiempo: 0.00031629 s
Paralelo -> Schedule: static | Hilos: 16 -> | Tiempo: 0.00606599 s
Paralelo -> Schedule: dynamic | Hilos: 4 -> | Tiempo: 0.00014344 s
Paralelo -> Schedule: dynamic | Hilos: 8 -> | Tiempo: 0.00046062 s
Paralelo -> Schedule: dynamic | Hilos: 16 -> | Tiempo: 0.00410690 s
Paralelo -> Schedule: guided | Hilos: 4 -> | Tiempo: 0.00011550 s
Paralelo -> Schedule: guided | Hilos: 8 -> | Tiempo: 0.00039829 s
Paralelo -> Schedule: guided | Hilos: 16 -> | Tiempo: 0.00623665 s
Total Farolas: 10961284 | Consumo Total: 1954689529
-----------------------------------------
---------- Mapa tamaño: 1000 ----------
Secuencial -> | Tiempo: 0.00821534 s
Paralelo -> Schedule: static | Hilos: 4 -> | Tiempo: 0.00301534 s
Paralelo -> Schedule: static | Hilos: 8 -> | Tiempo: 0.00207143 s
Paralelo -> Schedule: static | Hilos: 16 -> | Tiempo: 0.00200318 s
Paralelo -> Schedule: dynamic | Hilos: 4 -> | Tiempo: 0.00279922 s
Paralelo -> Schedule: dynamic | Hilos: 8 -> | Tiempo: 0.00160860 s
Paralelo -> Schedule: dynamic | Hilos: 16 -> | Tiempo: 0.00537862 s
Paralelo -> Schedule: guided | Hilos: 4 -> | Tiempo: 0.00280626 s
Paralelo -> Schedule: guided | Hilos: 8 -> | Tiempo: 0.00201314 s
Paralelo -> Schedule: guided | Hilos: 16 -> | Tiempo: 0.00545558 s
Total Farolas: 274657662 | Consumo Total: 48979727324
-----------------------------------------
---------- Mapa tamaño: 2000 ----------
```

```
Secuencial -> | Tiempo: 0.03278805 s
Paralelo -> Schedule: static | Hilos: 4 -> | Tiempo: 0.00903295 s
Paralelo -> Schedule: static | Hilos: 8 -> | Tiempo: 0.00762345 s
Paralelo -> Schedule: static | Hilos: 16 -> | Tiempo: 0.00745852 s
Paralelo -> Schedule: dynamic | Hilos: 4 -> | Tiempo: 0.01074274 s
Paralelo -> Schedule: dynamic | Hilos: 8 -> | Tiempo: 0.00540706 s
Paralelo -> Schedule: dynamic | Hilos: 16 -> | Tiempo: 0.00453005 s
Paralelo -> Schedule: guided | Hilos: 4 -> | Tiempo: 0.01412565 s
Paralelo -> Schedule: guided | Hilos: 8 -> | Tiempo: 0.00632569 s
Paralelo -> Schedule: guided | Hilos: 16 -> | Tiempo: 0.00432434 s
Total Farolas: 1100449601 | Consumo Total: 196249983382
-----------------------------------------
```

Bloque 4: Salida de la ejecución del Ejercicio 3 resumen

```
Secuencial 200x200: 0.00033140 s
Promedio Paralelo con 4 hilos 200x200: 0.00019065 s
Promedio Paralelo con 8 hilos 200x200: 0.00039173 s
Promedio Paralelo con 16 hilos 200x200: 0.00546984 s
Secuencial 1000x1000: 0.00821534 s
Promedio Paralelo con 4 hilos 1000x1000: 0.00287361 s
Promedio Paralelo con 8 hilos 1000x1000: 0.00189773 s
Promedio Paralelo con 16 hilos 1000x1000: 0.00427913 s
Secuencial 2000x2000: 0.03278805 s
Promedio Paralelo con 4 hilos 2000x2000: 0.01130045 s
Promedio Paralelo con 8 hilos 2000x2000: 0.00645207 s
Promedio Paralelo con 16 hilos 2000x2000: 0.00543764 s
Promedio de ejecución Secuencial: 0.01399788 s
Promedio con schedule static: 0.00421113 s
Promedio con schedule dynamic: 0.00390858 s
Promedio con schedule guided: 0.00464457 s
```

En las estadisticas podemos observar como a mayor tamaño del problema mayor efectividad de la paralelización. Se ve como claramente la tiempo de ejecucion secuencial aumenta exponencialmente en problemas más grandes, mientras la paralelización consigue frenar este incremento. Con 16 nucleos no se incrementa el tiempo de ejecución en lo mas minimo lo que nos hace pensar que el mapa de 2000x2000 se nos queda corto el problema al menos para los 16 hilos.

Las estrategias de paralelización que hemos utilizado, a estas escalas tan pequeñas, no podemos detectar que ninguna resalte especialmente sobre otra, aunque si podemos decir que minimamente el schedule dynamic resulta ser la estrategia más eficiente. En conclusión hemos visto como aumentando el tamaño de un problema pequeño podemos sacar diferentes estadisticas que nos evidencian que la paralelización es extremadamente util en problemas complejos.

También hemos visto que se pueden usar diferentes modos de paralelización, e incluso limitar los hilos para sacar estadisticas más especificas y observar más claramente como al aumentar el número de hilos tambien aumentamos la productividad.

Bloque 5: Ejemplo de código en C++.

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

// Constantes de tamaños de mapa, número de hilos y schedules
const int MAP_SIZES[] = {200, 1000, 2000};
const int NUM_MAPS = 3;
const int THREAD_COUNTS[] = {4, 8, 16};
const int NUM_THREADS = 3;
```

```
const string SCHEDULES[] = {"static", "dynamic", "guided"};
const int NUM_SCHEDULES = 3;

const int MIN_FAROLAS = 50, MAX_FAROLAS = 500;
const int BAJO_MIN = 70, BAJO_MAX = 100;
const int MEDIO_MIN = 150, MEDIO_MAX = 200;
const int ALTO_MIN = 250, ALTO_MAX = 300;

struct Celda {
 int num_farolas;
 int consumo_total;
};

void inicializarMapa(vector<vector<Celda>> &mapa, int map_size) {
 // Usamos srand solo una vez; en caso de llamar repetidamente dentro del bucle
 // puede reinicializar la semilla, pero esto es solo un ejemplo.
 srand(time(0));
 for (int i = 0; i < map_size; i++) {
  for (int j = 0; j < map_size; j++) {
   mapa[i][j].num_farolas = rand() % (MAX_FAROLAS - MIN_FAROLAS + 1) + MIN_FAROLAS;
   mapa[i][j].consumo_total = 0;  // reinicializamos para cada celda
   for (int k = 0; k < mapa[i][j].num_farolas; k++) {
    int tipo = rand() % 3;
    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 secuencial para calcular totales
void calcularConsumoSecuencial(const vector<vector<Celda>> &mapa, long long &total_farolas,
   long long &consumo_total, int map_size) {
 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 paralela con schedule(runtime).
// Se utiliza omp_set_schedule() para configurar la política según la cadena recibida.
void calcularConsumoParalelo(const vector<vector<Celda>> &mapa, long long &total_farolas,
   long long &consumo_total,
int map_size, int num_threads, string schedule, double &tiempo) {
 total_farolas = 0;
 consumo_total = 0;
 omp_set_num_threads(num_threads);

 // Configuramos la política de schedule según el parámetro recibido.
 if(schedule == "static")
 omp_set_schedule(omp_sched_static, 0);
 else if(schedule == "dynamic")
 omp_set_schedule(omp_sched_dynamic, 0);
 else if(schedule == "guided")
 omp_set_schedule(omp_sched_guided, 0);

 double start = omp_get_wtime();
 #pragma omp parallel for reduction(+:total_farolas, consumo_total) schedule(runtime)
 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;
  }
 }
 tiempo = omp_get_wtime() - start;
}

int main() {
 // Para mostrar resultados con más decimales
 cout << fixed << setprecision(8);

 double totalSecuencialGlobal = 0;
```

```
int countSecuencial = 0;

// Para acumular promedios por schedule globalmente
double sumaSchedule[NUM_SCHEDULES] = {0};
int countSchedule[NUM_SCHEDULES] = {0};

// Matriz para almacenar el tiempo promedio (sobre schedules) de cada mapa y configuración
   de hilos.
double promedioParaleloPorMapa[NUM_MAPS][NUM_THREADS] = {0};

// Además, para el mensaje final se requiere imprimir "Promedio Paralelo con X hilos "MxM
// para cada mapa.

// Recorremos cada tamaño de mapa
for (int m = 0; m < NUM_MAPS; m++) {
 int map_size = MAP_SIZES[m];
 vector<vector<Celda>> mapa(map_size, vector<Celda>(map_size, {0,0}));
 inicializarMapa(mapa, map_size);

 // Cálculo secuencial (se ejecuta una vez por mapa)
 long long total_farolas_seq, consumo_total_seq;
 double start = omp_get_wtime();
 calcularConsumoSecuencial(mapa, total_farolas_seq, consumo_total_seq, map_size);
 double tiempo_seq = omp_get_wtime() - start;
 totalSecuencialGlobal += tiempo_seq;
 countSecuencial++;

 cout << "\n---------- Mapa tamaño: " << map_size << " ----------\n";
 cout << "Secuencial ->  | Tiempo: " << tiempo_seq << " s" << endl;

 // Variable para acumular, por cada cantidad de hilos, el tiempo (sobre los 3 schedules)
 double sumaTiempoPorThreads[NUM_THREADS] = {0};

 // Recorremos cada schedule y cada configuración de hilos
 for (int s = 0; s < NUM_SCHEDULES; s++) {
  for (int t = 0; t < NUM_THREADS; t++) {
   long long total_farolas_par, consumo_total_par;
   double tiempo_par;
   start = omp_get_wtime();
   calcularConsumoParalelo(mapa, total_farolas_par, consumo_total_par, map_size,
  THREAD_COUNTS[t], SCHEDULES[s], tiempo_par);
   tiempo_par = omp_get_wtime() - start;
   // Imprimimos cada ejecución paralela (no se vuelcan los totales en cada línea)
   cout << "Paralelo -> Schedule: " << SCHEDULES[s]
   << " | Hilos: " << THREAD_COUNTS[t]
   << " ->  | Tiempo: " << tiempo_par << " s" << endl;

   sumaTiempoPorThreads[t] += tiempo_par;

   // Acumulamos para promedios globales por schedule (cada prueba cuenta)
   sumaSchedule[s] += tiempo_par;
   countSchedule[s]++;
  }
 }

 // Imprimimos los resultados numéricos (una sola vez por mapa, según la versión secuencial)
 cout << "Total Farolas: " << total_farolas_seq
 << " | Consumo Total: " << consumo_total_seq << endl;

 cout << "-------------------------------------------\n";

 // Calculamos y guardamos el promedio (sobre los 3 schedules) para cada cantidad de hilos
   para éste mapa.
 for (int t = 0; t < NUM_THREADS; t++) {
  promedioParaleloPorMapa[m][t] = sumaTiempoPorThreads[t] / NUM_SCHEDULES;
 }
}

// Promedio global de la ejecución secuencial
double promedioSecuencialGlobal = totalSecuencialGlobal / countSecuencial;
cout << "\nPromedio de ejecución Secuencial: " << promedioSecuencialGlobal << " s\n" << endl
   ;

// Imprimir promedios paralelos por configuración y por mapa
for (int m = 0; m < NUM_MAPS; m++) {
 int map_size = MAP_SIZES[m];
 for (int t = 0; t < NUM_THREADS; t++) {
  cout << "Promedio Paralelo con " << THREAD_COUNTS[t] << " hilos "
  << map_size << "x" << map_size << ": "
  << promedioParaleloPorMapa[m][t] << " s" << endl;
```

```
   }
   cout << endl;
  }

  // Imprimir promedios globales por schedule (a lo largo de todos los mapas y configuraciones
     )
  for (int s = 0; s < NUM_SCHEDULES; s++) {
   double promSch = sumaSchedule[s] / countSchedule[s];
   cout << "Promedio con schedule " << SCHEDULES[s] << ": " << promSch << " s" << endl;
  }

  return 0;
}
```

Bloque 6: Ejemplo de código en Python.

```
import numpy as np

# Esto es un comentario de una sola línea
@my_decorator
def process_data(data):
"""
Esta es una docstring multilínea.
Procesa los datos y devuelve la media.
"""
if data is not None and len(data) > 0:
mean_value = np.mean(data)
print(f"El resultado es: {mean_value}")
return mean_value
return None

my_list = [1, 2, 3, 4, 5]
process_data(my_list)
```