Práctica 4

INTRODUCCIÓN A OPENMP Y PREPARACIÓN DEL ENTORNO

Jordi Blasco Lozano

Computación de alto rendimiento

Grado en Inteligencia Artificial

## Indice:

[Indice: 2](#_Toc190980674)

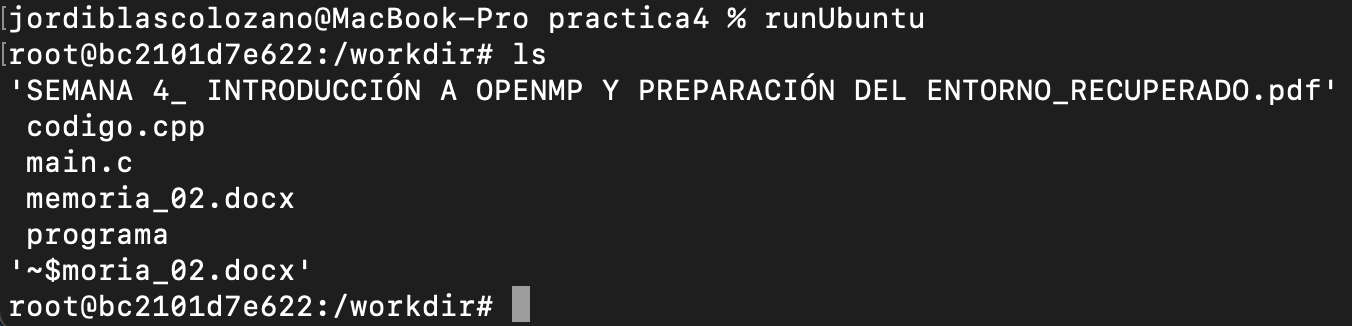
[1. INSTALACIÓN Y CONFIGURACIÓN 2](#_Toc190980675)

[2. PROGRAMACIÓN CON OPENMP 3](#_Toc190980676)

[CÓDIGO COMENTADO EJERCICIO FINAL 6](#_Toc190980677)

## 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.



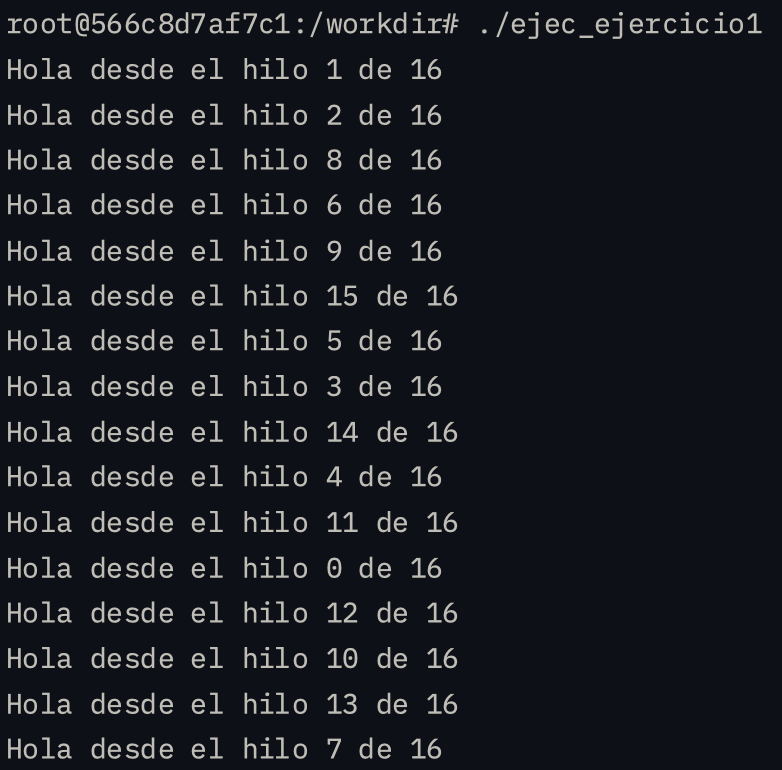
Solo he tenido que ejecutar el .sh para configurar el entorno de desarrollo que vamos a utilizar, no he tenido errores ya que la imagen docker actualiza el paquete apt usando el comando   
‘sudo apt update && sudo apt upgrade -y’ por defecto cuando se ejecuta.

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.

## 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

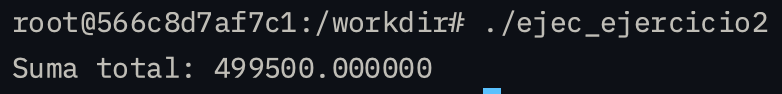
programación paralela.

**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 significativamente la eficiencia.



**Ejercicio 3 (final):**

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:

---------- 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

--------------------------------------------

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.

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

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.

## CÓDIGO COMENTADO EJERCICIO FINAL

## 

#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;

}