**Reporte Proyecto Apertura**

Emilia Hernández Medina

1. Descripción del código y estrategias de paralelización

Las bibliotecas que empleé para desarrollar el programa fueron las siguientes:

#include <iostream>

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fstream>

#include <sstream>

#include <string>

#include <vector>

#include <set>

#include <cctype>

#include <filesystem>

#include <iomanip>

Empleé a biblioteca omp.h para implementar la paralelización mediante OpenMP; las bibliotecas stdio.h, stdlib.h, fstream, sstream, *filesystem, iomanip* y set para la lectura y escritura de archivos CSV, así como para la creación de las matrices que almacenan los puntos. Usé las bibliotecas string y vector, pues facilitan la declaración de variables y la manipulación de los datos, ya que permiten estructuras dinámicas y de acceso directo que simplifican el manejo de información.

El primer paso para implementar el experimento que se presenta en este reporte fue generar ocho archivos CSV que contienen dos coordenadas con valores entre 0 y 1 con el código que nos proveyó el profesor. A partir de estos datos se realiza un análisis para determinar si cada punto corresponde a un outlier o a un core. La clasificación se define con base en dos criterios principales. En primer lugar, si un punto tiene al menos cinco (o los que indique el usuario) vecinos dentro de una distancia menor o igual a ε (epsilon), se clasifica como core. En caso contrario, se considera outlier. En segundo lugar, si un punto se clasifica inicialmente como outlier, pero existe un punto core a una distancia menor que ε, el punto se reclasifica como core.

Para cumplir con estos objetivos, primero desarrollé un método encargado de leer los archivos CSV y convertirlos en matrices. Posteriormente, procesé cada matriz mediante tres algoritmos distintos: una versión serial y dos versiones paralelas con metodologías diferentes. Por último, comparé estas versiones en términos de eficiencia y speedup, utilizando diferentes configuraciones de número de procesos (1, 5, 11 y 22).

En lugar de representar las matrices como arreglos de apuntadores a otros arreglos, opté por estructuras de vectores, donde cada matriz se define como un vector de filas y cada fila como un vector de valores. Esta representación permitió una implementación más clara y flexible, que facilitó la comprensión y el manejo de los datos durante el desarrollo del programa.

A continuación, se presentan los métodos utilizados en el código, junto con los comentarios necesarios para comprender su funcionamiento y las decisiones relevantes tomadas en su diseño.

* Convertir un csv a una matriz

En vez de trabajar con matrices que fueran un arreglo de apuntadores a otros arreglos, trabajé con matrices de vectores (la matriz es un vector de filas, en la que cada fila es a su vez un vector), pues me permitió entender de una manera mucho más clara lo que estaba haciendo.

vector<vector<double>> csvToMatix(string path){

//leer archivo para poder hacer una matriz con los puntos

vector<vector<double>> M;

//Abrir archivo

ifstream file(path);

// Verificación básica de que se abrió bien el archivo.

if (!file) {

cerr << "No pude abrir el archivo\n"; // Código de salida != 0 indica error.

return M;

}

// Matriz donde guardaremos el CSV:

// - Cada fila del CSV será un vector<double>

// - La "matriz" completa es vector< vector<double> >

// Variable temporal para almacenar cada línea de texto del archivo

string line;

// Lee el archivo línea por línea hasta EOF (end of file)

while (getline(file, line)) {

// Convierte la línea a un stream para poder extraer "celdas" separadas por coma

stringstream ss(line);

// 'cell' contendrá el texto de cada campo (entre comas)

string cell;

// 'row' será la fila numérica que construiremos a partir de esta línea

vector<double> row;

// Al terminar main() (o el bloque donde vive M), M y

//todas sus filas se destruyen y liberan memoria sin hacer nada, por lo que

// no hay que liberar memoria al final.

// Extrae cada celda separada por coma: getline lee hasta el delimitador ','

while (getline(ss, cell, ',')) {

// --- TRIM manual para quitar espacios en blanco alrededor del valor ---

size\_t i = 0, j = cell.size();

// Avanza 'i' mientras haya espacios al inicio

while (i < j && isspace((unsigned char)cell[i])) i++;

// Retrocede 'j' mientras haya espacios al final

while (j > i && isspace((unsigned char)cell[j - 1])) j--;

// Substring ya sin espacios iniciales/finales

string trimmed = cell.substr(i, j - i);

// Convierte el texto a double y lo agrega a la fila.

// std::stod lanza excepción si el texto no es número válido

row.push\_back(std::stod(trimmed));

}

// Si la línea no estaba vacía, agrega la fila a la matriz

if (!row.empty()) {

M.push\_back(std::move(row)); // move evita copiar innecesariamente

}

}

return M;

}

* Clasificación serial

vector<vector<double>> serial(double epsilon, int minPts, const vector<vector<double>>& M)

{

int pts;

double dist;

bool reach;

int j;

vector<int> label(M.size(), -1); // Creamos un vector de etiquetas para cada punto. Inicializa todas las etiquetas como -1 (no-core)

vector<vector<double>> outliers; //creamos el vector dónde vamos a guardar los outliers (este es el output del método)

//Identificar los primeros puntos core

// 1 = core, -1 = no-core

// Con este for recorremos cada punto de nuestra matriz M para clasificarlo

for (int i = 0; i < (int)M.size(); i++)

{

pts = 0;

j = 0;

// opté por no usar un for para recorrer a todos los otros puntos para encontrar vecinos porque en mi clase de algoritmos y programación nuestra maestra nos decía que no se debían usar breaks en un for, para eso está el while

while(j<M.size() && pts<minPts) // el while recorre todos los posibles vecinos. Para cuando ya se revisaron todos los puntos del arreglo, o cuando ya se cumplió el requisito mínimo de vecinos.

{

if (j != i)

{

dist = sqrt(pow(M[i][0] - M[j][0], 2) + pow(M[i][1] - M[j][1], 2));

if (dist <= epsilon) //Identifica los vecinos con la distancia euclidiana

pts++;

}

j++;

}//Al salir del while se revisa si se salió del loop porque encontró suficientes vecinos, si esto es verdadero, entonces el punto es core, por lo que cambiamos el label a 1 (core)

if (pts >= minPts)

label[i]= 1;

}

//Identificar los epsilon alcanzables de los puntos core

for (int i = 0; i < (int)M.size(); i++)

{

if (label[i] == -1) // revisamos solo los no-core

{

reach = false; // esta variable nos dice si el punto ya fue alcanzado por un punto core

// buscar al menos un vecino que ya sea core

j = 0;

while(reach == false && j<M.size())// Para cuando ya revisó todos los puntos desde los cuales puede ser alcanzable o cuando ya se alcanzó por uno

{

if (label[j] == 1) // comparar solo contra cores

{

dist = sqrt(pow(M[i][0] - M[j][0], 2) + pow(M[i][1] - M[j][1], 2));

if (dist <= epsilon)

reach = true;// si ya encontró un punto core a partir del cual sí es espsilon alcanzable, entonces cambiamos el label a true para salir del while.

}

j++;

}

if (reach)

label[i] = 1; // alcanzable por core => lo tratamos como core

}

}

// recorro otra vez los puntos, y hago una matriz de vectores solo con los outliers

for(int i = 0; i < M.size(); i++)

{

if(label[i] == -1)

outliers.push\_back(M[i]);

}

return outliers;

}

* Primera clasificación paralela

Mismo código que el serial, pero paralelizo los fors (más detalle en la documentación del código).

vector<vector<double>> p1(double epsilon, int minPts, const vector<vector<double>>& M)

{

//primera versión paralela en la que se usa todo el espacio como indivisible

int pts;

double dist;

bool reach;

int j;

vector<int> label(M.size(), -1); // Inicializa todas las etiquetas como -1 (no-core)

vector<vector<double>> outliers; //creo un vector de etiquetas para cada punto

//Identificar los primeros puntos core

// 1 = core, -1 = no-core

//no puedo poner los pushes en el for que paralelizo

//porque se hacen condiciones de carrera

//todos los procesos comparten la matriz de datos, la de etiquetas, épsilon y minPts porque todos los hilos las necesitan para hacer sus procesos. Las privadas son j, pts y dist. Cada hilo va a trabajar de manera simultánea con estas variables, por lo que si todas tuvieran acceso se sobrescribirían dando pie a condiciones de carrera

#pragma omp parallel for shared(M,label,epsilon,minPts) private(j, pts,dist)

for (int i = 0; i < (int)M.size(); i++)

{

pts = 0;

j = 0;

while(j<M.size() && pts<minPts)

{

if (j != i)

{

dist = sqrt(pow(M[i][0] - M[j][0], 2) + pow(M[i][1] - M[j][1], 2));

if (dist <= epsilon)

pts++;

}

j++;

}

if (pts >= minPts)

label[i]= 1;

}

//Identificar los épsilon alcanzables de los puntos core. Misma lógica de paralelización que el ciclo anterior.

#pragma omp parallel for shared(M,label,epsilon) private(j, dist,reach)

for (int i = 0; i < (int)M.size(); i++)

{

if (label[i] == -1) // revisamos solo los no-core

{

reach = false;

// buscar al menos un vecino que ya sea core

j = 0;

while(reach == false && j<M.size())

{

if (label[j] == 1) // comparar solo contra cores

{

dist = sqrt(pow(M[i][0] - M[j][0], 2) + pow(M[i][1] - M[j][1], 2));

if (dist <= epsilon)

reach = true;

}

j++;

}

if (reach)

label[i] = 1; // alcanzable por core => lo tratamos como core/borde

}

}

//este tiene que ser serial pq si no hay condiciones de carrera

for(int i = 0; i < M.size(); i++)

{

if(label[i] == -1)

outliers.push\_back(M[i]);

}

return outliers;

}

* Segunda clasificación paralela

Mismo código base que el serial, solo que ahora divido el espacio en n subespacios de la misma área. Paralelizo por cada subespacio.

vector<vector<double>> p2(double epsilon, int minPts, int n, const vector<vector<double>>& M)

{

//segunda versión paralela en la que divide el espacio en n partes

int pts;

double dist;

bool reach;

int j;

vector<int> label(M.size(), -1); // Inicializa todas las etiquetas como -1 (no-core)

vector<vector<double>> outliers; //creo un vector de etiquetas para cada punto

//Identificar los primeros puntos core

// 1 = core, -1 = no-core

// Elegir xcuts \* ycuts = n, con factores lo más parejos posible

int xcuts = (int)std::floor(std::sqrt(n));

while (n % xcuts != 0) --xcuts; // siempre termina (al menos en 1)

int ycuts = n / xcuts;

double splitx = 1.0 / xcuts;

double splity = 1.0 / ycuts;

// Ahora sí, hay exactamente N bloques

std::vector<std::vector<int>> blocks(xcuts \* ycuts);

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

// clamping para el borde derecho/superior

int bx = std::min((int)(M[i][0] / splitx), xcuts - 1);

int by = std::min((int)(M[i][1] / splity), ycuts - 1);

int block\_id = bx + by \* xcuts; // numeración fila mayor

blocks[block\_id].push\_back(i);

}

//cada proceso tiene su bloque. Todos comparten las de etiquetas, épsilon y minPts porque todos los hilos/bloques las necesitan para hacer sus procesos. Las privadas son j, pts y dist. Cada bloque va a calcular las distancias solo con los puntos de su propio bloque.

#pragma omp parallel for shared(M,label,epsilon,minPts) private(j, pts,dist)

for (int i = 0; i < n; i++) //este for recorre bloques y eso es lo que hacemos paralelo

{

for (int k = 0; k < blocks[i].size(); k++)//recorre registros de cada bloque

{

pts = 0;

j= 0;

while(j<blocks[i].size() && pts<minPts)

{

if(k!=j){

dist = sqrt(pow(M[blocks[i][k]][0] - M[blocks[i][j]][0], 2) + pow(M[blocks[i][k]][1] - M[blocks[i][j]][1], 2));

if (dist <= epsilon)

pts++;

}

j++;

}

if (pts >= minPts)

label[blocks[i][k]] = 1;

}

}

//Identificar los épsilon alcanzables de los puntos core.

//Lo haces con todos los puntos, no solo por bloque, pues el resultado puede no ser consistente ej. Un punto puede ser épsilon alcanzable por otro que no está en su bloque y no ser alcanzable por ninguno de su bloque (misma lógica que el anterior). Lo paralelicé igual que en p1.

#pragma omp parallel for shared(M,label,epsilon) private(j, dist,reach)

for (int i = 0; i < (int)M.size(); i++)

{

if (label[i] == -1) // revisamos solo los no-core

{

reach = false;

// buscar al menos un vecino que ya sea core

j = 0;

while(reach == false && j<M.size())

{

if (label[j] == 1) // comparar solo contra cores

{

dist = sqrt(pow(M[i][0] - M[j][0], 2) + pow(M[i][1] - M[j][1], 2));

if (dist <= epsilon)

reach = true;

}

j++;

}

if (reach)

label[i] = 1; // alcanzable por core => lo tratamos como core/borde

}

}

//este tiene que ser serial pq si no hay condiciones de carrera

for(int i = 0; i < M.size(); i++)

{

if(label[i] == -1)

outliers.push\_back(M[i]);

}

return outliers;

}

* Convertir matriz a csv

void matrixToCsv(const vector<vector<double>>& M, const vector<vector<double>>& outliers, const string& path)

{

// Crear un set para búsqueda rápida de outliers

set<pair<double, double>> outlierSet;

for (const auto& pt : outliers) {

outlierSet.insert({pt[0], pt[1]});

}

ofstream file(path);

if (!file) {

cerr << "No pude abrir el archivo para escribir\n";

return;

}

// Escribir encabezado

file << "x,y,tag\n";

for (const auto& pt : M) {

string tag = (outlierSet.count({pt[0], pt[1]}) > 0) ? "outlier" : "core";

file << pt[0] << "," << pt[1] << "," << tag << "\n";

}

file.close();

}

* Para ir guardando los resultados en un csv para poder crear los gráficos utilicé estos métodos

// -------- header helper (sin filesystem) --------

static bool need\_header(const std::string& path) {

std::ifstream fin(path, std::ios::binary);

if (!fin.good()) return true; // no existe o no abre

fin.seekg(0, std::ios::end);

return fin.tellg() == 0; // vacío

}

// -------- método pedido --------

void append\_benchmark\_csv(const std::string& csv\_path,

int DB,

int threads,

const std::string& algoritmo,

double tiempo\_s,

double speedup)

{

static std::mutex m; // para evitar colisiones si se llama concurrente

std::lock\_guard<std::mutex> lk(m);

const bool write\_header = need\_header(csv\_path);

std::ofstream out(csv\_path, std::ios::app);

if (!out) {

std::cerr << "[ERROR] No pude abrir " << csv\_path << " para escribir.\n";

return;

}

if (write\_header) {

out << "DB,threads,algoritmo,tiempo\_s,speedup\n";

}

// Por si alguien pasa threads<=0: para serial = 1

int th = threads;

if (th <= 0 && algoritmo == "serial") th = 1;

out << DB << ','

<< th << ','

<< algoritmo << ','

<< std::fixed << std::setprecision(6) << tiempo\_s << ','

<< std::fixed << std::setprecision(6) << speedup

<< '\n';

out.flush();

}

vector<vector<double>> csvToMatrix(string path) {

vector<vector<double>> M;

std::ifstream file(path);

if (!file) {

std::cerr << "No pude abrir el archivo: " << path << "\n";

return M;

}

string line;

while (std::getline(file, line)) {

std::stringstream ss(line);

string cell;

vector<double> row;

while (std::getline(ss, cell, ',')) {

size\_t i = 0, j = cell.size();

while (i < j && isspace(static\_cast<unsigned char>(cell[i]))) ++i;

while (j > i && isspace(static\_cast<unsigned char>(cell[j-1]))) --j;

string trimmed = cell.substr(i, j - i);

if (!trimmed.empty()) { // evita stod("")

row.push\_back(std::stod(trimmed));

}

}

if (!row.empty()) {

M.emplace\_back(std::move(row));

}

}

return M;

}

static int db\_from\_path(const std::string& p) {

size\_t pos = p.find\_last\_of('/');

if (pos == std::string::npos) pos = 0; else pos++;

size\_t underscore = p.find('\_', pos);

return std::stoi(p.substr(pos, underscore - pos));

}

* Código principal

int main()

{

const std::string CSV\_PATH = "resultados\_benchmarks.csv";

double start, end; //para medir tiempos

double timeposerial;

std::vector<std::vector<double>> M;

std::vector<std::vector<double>> outliers;

int numThread[4] ={1,5,11,22};

std::string pathsIn[8]= {

"/Users/emiliahernandez/Desktop/comp para/proyecto1/20000\_data.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/40000\_data.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/80000\_data.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/120000\_data.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/140000\_data.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/160000\_data.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/180000\_data.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/200000\_data.csv"

};

std::string pathsOut[8]= {

"/Users/emiliahernandez/Desktop/comp para/proyecto1/20000\_CR.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/40000\_CR.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/80000\_CR.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/120000\_CR.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/140000\_CR.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/160000\_CR.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/180000\_CR.csv",

"/Users/emiliahernandez/Desktop/comp para/proyecto1/200000\_CR.csv"

};

for (int i = 0; i < 8; i++) {

M = csvToMatrix(pathsIn[i]);

std::cout << "Filas: " << M.size() << "\n";

const int DB = db\_from\_path(pathsIn[i]);

// repito 10 veces cada experimento para tener un promedio

for (size\_t t = 0; t < 10; t++) {

// ---------- SERIAL ----------

std::cout << "Puntos atípicos serial: ";

start = omp\_get\_wtime();

outliers = serial(.05, 5, M);

end = omp\_get\_wtime();

std::cout << outliers.size() << "\n";

std::cout << "Tiempo serial: " << (end - start) << " segundos\n";

timeposerial = end - start;

// log CSV (threads=1 para serial)

append\_benchmark\_csv(CSV\_PATH, DB, 1, "serial", timeposerial, 1.0);

// ---------- PARALELOS (p1 y p2) ----------

for (int j = 0; j < 4; j++) {

omp\_set\_num\_threads(numThread[j]);

std::cout << "Número de threads: " << numThread[j] << "\n";

// ---- p1 ----

std::cout << "Puntos atípicos p1: ";

start = omp\_get\_wtime();

outliers = p1(.05, 5, M);

end = omp\_get\_wtime();

double tp1 = (end - start);

std::cout << outliers.size() << "\n";

std::cout << "Tiempo p1: " << tp1 << " segundos\n";

std::cout << "Speedup p1: " << (timeposerial / tp1) << "\n";

// log CSV p1

append\_benchmark\_csv(CSV\_PATH, DB, numThread[j], "p1", tp1, timeposerial / tp1);

// ---- p2 ----

std::cout << "Puntos atípicos p2: ";

start = omp\_get\_wtime();

outliers = p2(.05, 5, 4, M); // épsilon .05, min puntos 5, 4 subespacios

end = omp\_get\_wtime();

double tp2 = (end - start);

std::cout << outliers.size() << "\n";

std::cout << "Tiempo p2: " << tp2 << " segundos\n";

std::cout << "Speedup p2: " << (timeposerial / tp2) << "\n";

// log CSV p2

append\_benchmark\_csv(CSV\_PATH, DB, numThread[j], "p2", tp2, timeposerial / tp2);

}

}

matrixToCsv(M, outliers, pathsOut[i]);

}

return 0;

}

1. Descripción Experimental del desempeño
   1. Explicación detallada de la definición del experimento.

El experimento tuvo como propósito evaluar el desempeño y la eficiencia de distintas estrategias de paralelización aplicadas al algoritmo DBSCAN, cuyo objetivo principal es identificar puntos de ruido u outliers dentro de un conjunto de datos bidimensionales. La implementación se realizó en C++ con la biblioteca OpenMP, con el fin de analizar cómo el uso de múltiples hilos de ejecución puede reducir el tiempo de procesamiento en comparación con una versión serial del mismo algoritmo.

El algoritmo DBSCAN se basa en dos parámetros fundamentales: **epsilon (ε),** que representa la distancia máxima para considerar que dos puntos son vecinos, y **minPts**, que define el número mínimo de puntos dentro del radio ε necesario para que una región sea considerada densa. Con base en estos parámetros, cada punto del conjunto se clasifica como punto core si cumple la condición de densidad o es alcanzable por otro *core*, o bien como punto de ruido si no alcanza dicho umbral. El proceso implica calcular distancias entre pares de puntos y determinar la densidad local, lo que convierte a DBSCAN en un algoritmo computacionalmente intensivo cuando se trabaja con grandes volúmenes de datos.

Para evaluar el efecto del paralelismo, se implementaron tres versiones del algoritmo. La primera fue una versión **serial**, que sirvió como referencia base. La segunda, denominada **P1**, aplicó paralelización sobre los bucles principales sin dividir el conjunto de datos, de modo que todo el espacio de puntos se trató como una única unidad de trabajo. La tercera versión, **P2**, dividió el conjunto total en submatrices independientes que fueron procesadas por distintos procesos y posteriormente unificadas para mantener la consistencia en la clasificación de los puntos. El objetivo de esta comparación fue analizar de qué manera la granularidad del paralelismo y la forma de particionar los datos influyen en el rendimiento general del algoritmo.

Los experimentos se realizaron con conjuntos de datos generados de forma aleatoria que incluyeron tamaños de veinte mil hasta doscientos mil puntos. Para cada tamaño se probó el rendimiento con diferentes configuraciones de hilos: uno, la mitad del número de núcleos virtuales del procesador (5), el número total de núcleos virtuales (11) y el doble de esa cantidad (22). Cada combinación de tamaño y número de hilos se ejecutó diez veces y se calculó el promedio de los tiempos obtenidos para reducir el efecto de variaciones aleatorias.

La métrica de evaluación empleada fue el **speedup**, definido como la razón entre el tiempo de ejecución de la versión secuencial y el tiempo de ejecución de la versión paralela. Esta medida permitió cuantificar la ganancia en eficiencia lograda por el uso de múltiples hilos. Los datos de entrada se generaron mediante la libreta DBSCAN\_noise.ipynb disponible en el repositorio del curso, y los resultados, exportados en formato CSV, se analizaron posteriormente con Python utilizando las bibliotecas pandas y matplotlib.

En conjunto, el experimento permitió examinar el comportamiento del algoritmo DBSCAN ante diferentes volúmenes de datos y niveles de paralelización. El análisis comparativo entre las versiones serial y paralelas ofreció evidencia sobre las ventajas de emplear estrategias de paralelización completas frente a esquemas de división espacial, así como sobre las limitaciones que aparecen cuando el número de hilos genera sobrecarga de sincronización.

* 1. Descripción del equipo donde se ejecutaron los experimentos en términos de hardware y software.

Los experimentos se realizaron en una computadora **MacBook Pro de 14 pulgadas (noviembre de 2023)** equipada con un **chip Apple M3 Pro** y **18 GB de memoria unificada**. El sistema operativo utilizado fue **macOS Sequoia 15.6.1**. El chip M3 Pro integra una arquitectura de **CPU y GPU unificadas basadas en ARM**, optimizada para tareas de cómputo intensivo y paralelización. Tiene un total de 11 cores virtuales.

El software empleado incluyó herramientas de compilación y análisis compatibles con la arquitectura ARM, en particular **Apple Clang/LLVM** con soporte para **OpenMP**. Los experimentos se programaron en **C++17**, y las visualizaciones de los resultados se realizaron con **Python 3.13** utilizando las librerías pandas y matplotlib.

* 1. Interpretación y análisis de resultados.

Los resultados obtenidos muestran comportamiento coherente entre el incremento del número de hilos y la mejora en el tiempo de ejecución de las versiones paralelas del algoritmo DBSCAN. En términos generales, la versión P1, que aplica paralelización sobre todo el conjunto de datos sin dividir la matriz, logró los mejores valores de aceleración en todas las pruebas realizadas. En contraste, la versión P2, que divide el espacio en submatrices independientes, presentó una mejora más limitada (al número de bloques) y estable, sin incrementos significativos al aumentar el número de hilos.

En la prueba con una base de datos de 20,000 puntos los *speedups* fueron buenos, pero no los mejores que obtuve en todo el proyecto debido al bajo costo computacional total (por ser pocos datos). En esta escala, la versión paralela P1 disminuyó el tiempo promedio de ejecución (*speedup*) más de 6 veces al utilizar 11 hilos, mientras que la P2 redujo el tiempo 3 veces con 11 hilos. Creo que P1 fue casi dos veces mejor que P2 debido a que metemos mucho *overhead* al separar los datos en bloques, y a que no usamos al máximo las capacidades de cómputo de los cores e hilos (se usan solo 4 hilos a pesar de que declaramos 11, porque solo hay 4 bloques). Es mucho más eficiente cuando dejas que la máquina distribuya la carga de manera óptima que hacerlo manualmente mediante bloques.

Cuando el tamaño de la base de datos aumentó a 80,000 y 120,000 puntos, las diferencias se hicieron más notorias. En estos casos, la versión P1 alcanzó aceleraciones cercanas a 7.3 veces con 22 hilos, lo que demuestra una utilización eficiente de los recursos del chip M3 Pro. A partir de ese punto, el rendimiento parece que se va a estabilizar, lo que sugiere que el procesador alcanzó su límite óptimo de paralelización para esa carga de trabajo. Lo más sorprendente fue que el *speedup* de 11 a 22 hilos (con 80,000- 200,000) mejoró, lo que indica que el *overhead* de duplicar el número de procesos (al número de *cores* virtuales) no fue significativo, y fue tan pequeño que mejoró el rendimiento. En cambio, la versión P2 mantuvo un aumento más moderado con una aceleración máxima cercana a tres, sin mejoras adicionales al incrementar el número de hilos (llega a su máximo en 5, que igualmente tiene sentido, pues trabajé con 4 bloques, por lo que no se paraleliza más a pesar de tener más procesos).

En el experimento con la base de datos más grande, de 200,000 puntos, los resultados confirmaron la tendencia observada. La versión P1 mantuvo una aceleración promedio entre 7.2 veces, lo que la posicionó como la más eficiente, mientras que la versión P2 permaneció prácticamente constante alrededor de 2.9 veces. Este patrón muestra que la estrategia de paralelización total se adapta mejor al procesamiento masivo de datos en un entorno con memoria unificada, mientras que la división espacial no ofrece beneficios proporcionales al tamaño del problema.

En conjunto, los resultados demuestran que la versión P1 supera ampliamente tanto a la versión serial como a la versión P2, especialmente en volúmenes de datos medianos y grandes. La aceleración obtenida refleja una buena escalabilidad hasta cierto punto, seguida de una meseta derivada de la saturación de los núcleos virtuales y de los costos de sincronización. Por el contrario, la estrategia P2 mantiene una eficiencia constante pero limitada al número de bloques que se asignan.

En conclusión, el paralelismo en DBSCAN resulta altamente ventajoso cuando se aplica sobre el conjunto total de datos y se evita la fragmentación espacial. La versión P1 logró un mejor equilibrio entre carga de trabajo, coherencia de resultados y aprovechamiento del hardware disponible, lo que confirma la efectividad de la paralelización global frente a las estrategias basadas en particiones.

* 1. Sección de “Uso responsable y ético de IA Generativa”

Durante el desarrollo del proyecto utilicé inteligencia artificial generativa como herramienta de apoyo en tareas específicas, siguiendo los lineamientos éticos establecidos por el IEEE y Elsevier. En particular, empleé ChatGPT para redactar partes de este reporte, que después revisé. También, utilicé la IA para pulir la redacción del reporte.

Consulté la herramienta para aclarar dudas de notación y para crear funciones auxiliares en el código, como los métodos csvToMatrix y matrixToCsv, que facilitan la lectura y escritura de los archivos de entrada y salida del programa, ya que no conocía la sintaxis en C+ para hacer esto; así como para los métodos que hacen el log de los resultados en una csv. Igualmente, la utilicé para elaborar la parte del código que dividía el espacio en n submatrices de la misma área y que asignaba cada registro a su bloque correspondiente. El uso de inteligencia artificial se limitó a estas tareas de apoyo, mientras que el diseño, implementación, experimentación y análisis de resultados fueron realizados de manera independiente y comprendidos en su totalidad por la autora.

Este uso se enmarca en una práctica responsable de la inteligencia artificial, entendida como un medio complementario que no sustituye la comprensión técnica ni el trabajo original.

Referencias:  
IEEE Guidelines for Generative AI Usage. <https://www.ieee-ras.org/publications/guidelines-for-generative-ai-usage>  
Elsevier Generative AI Policies for Journals.  
<https://www.elsevier.com/about/policies-and-standards/generative-ai-policies-for-journals>

* 1. Incluir gráfica

A graph with a line and a line

AI-generated content may be incorrect.

A graph with a line and a line

AI-generated content may be incorrect.A graph with a line and a line

AI-generated content may be incorrect.A graph with a line and a line

AI-generated content may be incorrect.A graph with a line and a line

AI-generated content may be incorrect.A graph with a line and a line

AI-generated content may be incorrect.A graph with a line and a blue line

AI-generated content may be incorrect.A graph with a line and a line

AI-generated content may be incorrect.

* 1. Archivo con los datos de los experimentos

<https://drive.google.com/drive/folders/1l7a1v9YkT-QAY7d7LxEYWgK9qjGYKcxV?usp=sharing>

Los archivos csv que tiene terminación *\_CR* (core o ruido) contienen los datos con las etiquetas. Los que tienen terminación *\_data* son los archivos raw, que son el input del programa. El archivo proyecto1.cpp contiene el código.