Trabajo de fin de grado

PROPUESTA DE PARALELIZACIÓN PARA ALGORITMO DE CLUSTERING DBSCAN

Vojislav Momcilovic Ivanovic

Dirigido por Carles Aliagas Castell

UNIVERSITAT ROVIRA I VIRGILI

Grau d'Enginyeria Informàtica

Tarragona

2022

AGRADECIMIENTOS

Primeramente, quiero dedicar mis más sinceros agradecimientos a todo el equipo docente del grado de ingeniería informática de la URV por ser unos profesores maravillosos a los que se les nota la ilusión y pasión por enseñar y transmitir conocimientos de manera profesional y personal. Con los que aprendí más de lo que pude llegar a imaginar durante estos años.

A Sergi Pavlenko, con quien la preinscripción a este grado empezó como una broma y se acabó convirtiendo en una inscripción por error. Gracias a él, he cometido posiblemente el error más afortunado de mi vida.

A todos mis compañeros y en especial a Marc Infante, mi compañero de laboratorios, con quien he podido compartir conocimientos, debatir ideas, discutir sobre ellas y ha sido un apoyo muy importante para poder ampliar los puntos de vista hacia más horizontes.

A mi querida pareja, la futura doctora Marta Balanyà, quien se enfadará si no la incluyo en los agradecimientos.

Al equipo docente de la universidad de Maribor, en Eslovenia por ser tan apasionados como los de la URV. Donde realice una estancia de erasmus de un año escolar. En especial al doctor Božidar Potočnik, quien ha hecho un sobreesfuerzo ayudándome en temas académicos y personales referentes a su universidad.

A Ivana Mičkena Perić, quien fue una compañera indispensable durante la travesía académica en Eslovenia. La cual dio todo lo que podía de sí y más para ayudarme en las clases y evaluaciones de la universidad de Maribor.

Dedicarle, posiblemente la mayor de mis gratitudes, al profesor Carles Aliagas quien, a parte de un excelente docente, ha tenido la amabilidad y paciencia de ayudarme y adaptarse a mis necesidades personales no solo en este proyecto, si no en las lecciones, durante un año muy convulso en mi ámbito personal. No puedo imaginar un tutor de TFG mejor a él.

Por último, un agradecimiento a mis padres y amigos que me han apoyado en esta aventura que es la universidad durante todos estos años. En especial a mi padre, Vladimir Momčilović, que después de tantos años de ánimos e ilusión se quedó a escasos meses de poder apreciar este momento, que representa la tan esperada llegada de mi título como Graduado en Ingeniería Informática.

A todos vosotros, mis más sinceros agradecimientos por toda vuestra ayuda y apoyo,

Vojislav Momčilović Ivanović

Resumen.

En este proyecto se pretende analizar el algoritmo de clustering basado en densidades llamado *DBSCAN* y realizar una propuesta para su paralelización que permitirá escalar el algoritmo a máquinas más potentes para tratar volúmenes de datos masivos.

Las implementaciones se realizarán usando el lenguaje de programación *C* mediante el uso de las tres interfaces de programación de aplicaciones:

- *openMP* para máquinas multihilo.
- *openMPI* para las máquinas multicomputadoras.
- *CUDA* para la ejecución de operaciones escalares con el uso de la tarjeta gráfica.

El objetivo de este proyecto es profundizar en los conocimientos relacionados con el hardware y las arquitecturas de las que se disponen. Gracias a estos análisis se podrán proponer modelos de paralelización escalables que puedan llegar a presentar eficiencias mayores que las ejecutadas en serie siendo conscientes de las limitaciones que se disponen.

Resum.

En aquest projecte es pretén analitzar l'algorisme de clustering basat en densitats anomenat *DBSCAN* i fer una proposta per a la seva paral·lelització que permetrà escalar l'algorisme a màquines més potents per tractar volums de dades massives.

Les implementacions es faran usant el llenguatge de programació *C* mitjançant l'ús de les tres interfícies de programació d'aplicacions:

- *OpenMP* per a màquines multi-fil.
- *OpenMPI* per a les màquines multi-computadores.
- *CUDA* per a l'execució d'operacions escalars amb l'ús de la targeta gràfica.

L'objectiu d'aquest projecte és aprofundir els coneixements relacionats amb el hardware i les arquitectures de què es disposen. Gràcies a aquestes anàlisis es podran proposar models de paral·lelització escalables que puguin arribar a presentar eficiències més grans que les executades en sèrie sent conscients de les limitacions que es disposen.

Abstract.

This project aims to analyze the density-based clustering algorithm called *DBSCAN* and make a proposal for its parallelization that will allow the algorithm to be scaled to more powerful machines to process massive data volumes.

The implementations will be done using the *C* programming language by using the three application programming interfaces:

- *OpenMP* for multithreaded machines.
- *OpenMPI* for multicomputer machines.
- *CUDA* for the execution of scalar operations with the use of the graphics card.

The objective of this project is to deepen the knowledge related to the hardware and its current architectures. Thanks to these analyses, it will be possible to propose scalable parallelization models that can present greater efficiencies than those executed in serial, being aware of its limitations.

Índice

1 INTRODUCCIÓN ... 7
1.1 DESCRIPCIÓN ... 7
1.2 MOTIVACIONES ... 8
1.3 OBJETIVOS ... 8
1.3.1 *Objetivos generales* ... 8
1.3.2 *Objetivos especificos* ... 9
1.3.2.1 Objetivos academicos ... 9
1.3.2.2 Objetivos personales ... 9
***2 CONOCIMIENTO PREVIO AL DESARROLLO* ... 10**
2.1 APRENDIZAJE NO SUPERVISADO ... 10
2.2 CLUSTERING ... 10
2.2.1 *Caracteristicas* ... 10
2.2.2 *Diseño* ... 11
2.2.3 *Tipos de clustering* ... 11
2.3 DBSCAN ... 12
2.3.1 *Parametros* ... 12
2.3.1.1 Dataset ... 12
2.3.1.2 Epsilon ... 13
2.3.1.3 Mínimo de puntos ... 14
2.3.1.4 Función distancia ... 14
2.3.2 *Estructuras* ... 14
2.3.2.1 Punto ... 14
2.3.2.2 Vecindad Epsilon ... 14
2.3.3 *Algoritmo* ... 15
2.3.4 *Evaluación* ... 17
2.3.4.1 Complejidad espacial ... 17
2.3.4.2 Complejidad computacional ... 18
2.3.5 *Usos prácticos* ... 18
2.4 MULTIPROCESADOR ... 19
2.5 OPENMP ... 21
2.5.1 *Modelos de ejecución* ... 21
2.5.1.1 Fork-Join ... 21
2.5.1.2 Tareas ... 21
2.6 MULTICOMPUTADORAS ... 22
2.7 OPENMPI ... 22
2.8 CUDA ... 23
2.8.1 *Arquitectura* ... 23
2.8.2 *Modelos de Programación* ... 25
3 ANALISIS DEL PROBLEMA ... 26
3.1 PROFILING ... 26
3.1.1 *Parametros para profiling* ... 26
3.1.2 *Evaluacion de resultados* ... 27
3.2 ANÁLISIS DE DEPENDENCIAS DE DATOS ... 29
3.2.1 *Dependencias de función DBSCAN* ... 31
3.2.2 *Dependencias de función de expansión* ... 32
3.2.3 *Dependencias de función de difusión* ... 35
3.2.4 *Dependencias de función de consulta de region de vecinos* ... 37
4 PROPUESTA DE PARALELIZACIÓN ... 41
4.1 TÉCNICA DE DESCOMPOSICIÓN ... 41
4.1.1 *Descomposición de datos* ... 41
4.1.2 *Asignación de procesos* ... 42

4.2 REDUCCION DEL EFECTO DE LA INTERACCION 42
4.2.1 *Localidad de los datos* 42
4.2.2 *Frecuencia de interacción* 43
4.3 PSEUDOCÓDIGO 43

5 IMPLEMENTACIÓN 45
5.1 OPENMP 45
5.2 OPENMPI 47
5.3 CUDA 52

6 EVALUACIÓN 56
6.1 MULTIPROCESADOR 57
6.1.1 *Máquina* 57
6.1.2 *Resultados* 58
6.2 MULTICOMPUTADORA 60
6.2.1 *Máquina* 60
6.2.2 *Resultados* 61
6.2.3 *Propuesta alternativa* 63
6.3 UNIDAD DE PROCESADO GRÁFICO 64
6.3.1 *Máquina* 64
6.3.2 *Resultados* 64
6.3.3 *Propuesta alternativa* 65

7 CONCLUSIONES 67
7.1 USOS 67
7.2 LIMITACIONES 67
7.2 CONCLUSIONES 68

8 REFERÈNCIES 69
8.1 BIBLIOGRAFÍA 69
8.2 RECURSOS WEB 69
8.3 SOFTWARE 70

ANEXOS 71
ANEXO A. ARCHIVOS NECESARIOS 71
ANEXO B. INSTALACIONES NECESARIAS 72
ANEXO C. COMPILACIÓN Y EJECUCIÓN 73
ANEXO D. ARCHIVOS DE SALIDA 74

Índice de tablas

Tabla 1. Profiling de DBSCAN con 10000 puntos procesados 27
Tabla 2. Representación de ejecución vertical en asignación estática de datos 42
Tabla 3. Localidad de los datos por cada proceso para un conjunto de 10000 puntos 43
Tabla 4. Muestra del valor constante en la eficiencia para un algoritmo escalable 57
Tabla 5. Niveles de memoria cache para AMD Ryzen Threadripper 57
Tabla 6. Eficiencia obtenida sobre conjuntos de datos de distintos tamaños según el número procesos 59
Tabla 7. Isoeficiencia obtenida sobre conjuntos de datos de distintos tamaños según el número de procesos 60
Tabla 8. Niveles de memoria cache para AMD Ryzen 7 6800HS 61
Tabla 9. Eficiencia obtenida sobre conjuntos de datos de distintos tamaños según el número de nodos en el sistema 62
Tabla 10. Isoeficiencia obtenida sobre conjuntos de datos de distintos tamaños según el número nodos en el sistema 63

Índice de figuras

Figura 1. Comparación de clusters creados entre distintos algoritmos ... 7
Figura 2. Puntos ordenados por distancia a los tres vecinos más cercanos ... 13
Figura 3. Tipos de puntos en la Vecindad Epsilon con parámetros Epsilon marcado por circunferencias y mínimo de puntos igual a cuatro ... 15
Figura 4. Flujo de datos e instrucciones para arquitecturas MIMD ... 20
Figura 5. Acceso a memoria compartida UMA ... 20
Figura 6. Acceso a memoria compartida NUMA ... 21
Figura 7. Sistema de paso de mensajes para multicomputadoras ... 22
Figura 8. Comparación en el uso de transistores entre CPU y GPU ... 24
Figura 9. Jerarquía de memoria de unidad de procesado gráfico ... 25
Figura 10. Comparativa de tiempo total de procesado de las funciones en función del número de puntos ... 28
Figura 11. Errores derivados de no solucionar la dependencia de datos entre distintos procesos ... 30
Figura 12. Grafo de dependencia de datos de función DBSCAN ... 32
Figura 13. Grafo de dependencia de datos de función para expandir cluster ... 34
Figura 14. Grafo de dependencia de datos de función para expandir cluster con transformaciones aplicadas ... 35
Figura 15. Grafo de dependencia de datos de función de difusión ... 37
Figura 16. Grafo de dependencias de datos en consulta región de vecinos ... 39
Figura 17. Grafo de dependencias de datos en consulta región de vecinos en ejecución paralela ... 40
Figura 18. Representación de partición de datos de entrada unidimensional para varios procesos ... 41
Figura 19. Tiempos de ejecución de conjuntos de datos de distintos tamaños según el número de procesos ... 58
Figura 20. Speed-Up obtenido sobre conjuntos de datos de distintos tamaños según el número de procesos ... 59
Figura 21. Tiempos de ejecución de conjuntos de datos de distintos tamaños según el número de nodos en el sistema ... 61
Figura 22. Speed-Up obtenido sobre conjuntos de datos de distintos tamaños según el número de nodos en el sistema ... 62
Figura 23. Tiempos de ejecución de conjuntos de datos de distintos tamaños para ejecución en GPU ... 65

Índice de Códigos

Código 1. DBSCAN 31
Código 2. Función de expansión de cluster 32
Código 3. Función de difusión 36
Código 4. Consultar región de vecinos de un punto 38
Código 5. Declaración de variables para recurso compartido 45
Código 6. Declaración de región paralela y la localidad de las variables 46
Código 7. Declaración paralela del modelo de ejecución fork-join 46
Código 8. Actualización de recursos de memoria compartida 46
Código 9. Concatenación de resultados en el proceso maestro 47
Código 10. Consulta de identificador del nodo 48
Código 11. Diferenciación de ejecución entre nodo raíz y otros nodos 48
Código 12. Sincronización de procesos mediante broadcast 49
Código 13. Asignación de datos mediante el rango del proceso y computación paralela 50
Código 14. Envío al proceso raíz del número de resultados encontrados por cada nodo 51
Código 15. Recolección de datos final y concatenación de resultados 52
Código 16. Cálculo de distribución de bloques, reserva de memoria y transferencia de datos a la GPU 53
Código 17. Adaptación de consulta de región de vecinos para GPU 53
Código 18. Implementación local de función distancia en GPU 54
Código 19. Lanzamiento de función en GPU, recepción de región de vecinos y adaptación al algoritmo en serie 54
Código 20. Liberación de memoria reservada en GPU 55

Índice de Pseudocódigos

PSEUDOCÓDIGO 1. DBSCAN 15
PSEUDOCÓDIGO 2. FUNCIÓN DE EXPANSIÓN DE CLUSTER 16
PSEUDOCÓDIGO 3. FUNCIÓN DE VECINDAD EPSILON PARALELA 44

1 Introducción

1.1 Descripción

Hoy en día existen muchos algoritmos de machine learning e inteligencia artificial para la minería y el procesado de datos. Entre ellos destacan los algoritmos denominados de *clustering* cuya función es la de agrupar conjuntos de datos con características similares en base a atributos proporcionados por el usuario.

El algoritmo *DBSCAN* fue propuesto en 1996 y sigue siendo uno de los más usados. Se basa en la agrupación de datos en función de la densidad espacial de los mismos.

Algunos de los algoritmos de *clustering* más populares similares al estudiado en este proyecto como el *KMeans* y el *Fuzzy K-means* requieren que, a través de los parámetros de entrada, se establezcan un número concreto de agrupaciones que queremos encontrar. Igual de útil para el análisis de datos, pero con aplicaciones diferentes.

La diferencia principal con el *DBSCAN* es que este algoritmo encuentra por sí solo un numero arbitrario de agrupaciones. Tanto este número de agrupaciones como su forma puede variar en función de la densidad espacial de los puntos dependiendo de los parámetros de distancia entre puntos *épsilon* y el *mínimo de puntos* que debe haber para que se considere una agrupación.

Esto hace que, como se puede apreciar en la **Figura 1**, el algoritmo *DBSCAN* es el único cuyos resultados, como dice el autor H. Antonio Vazquez, se aproximan a los de un ser humano si estuviese clasificando los puntos aglomerados en un mapa a simple vista [9].

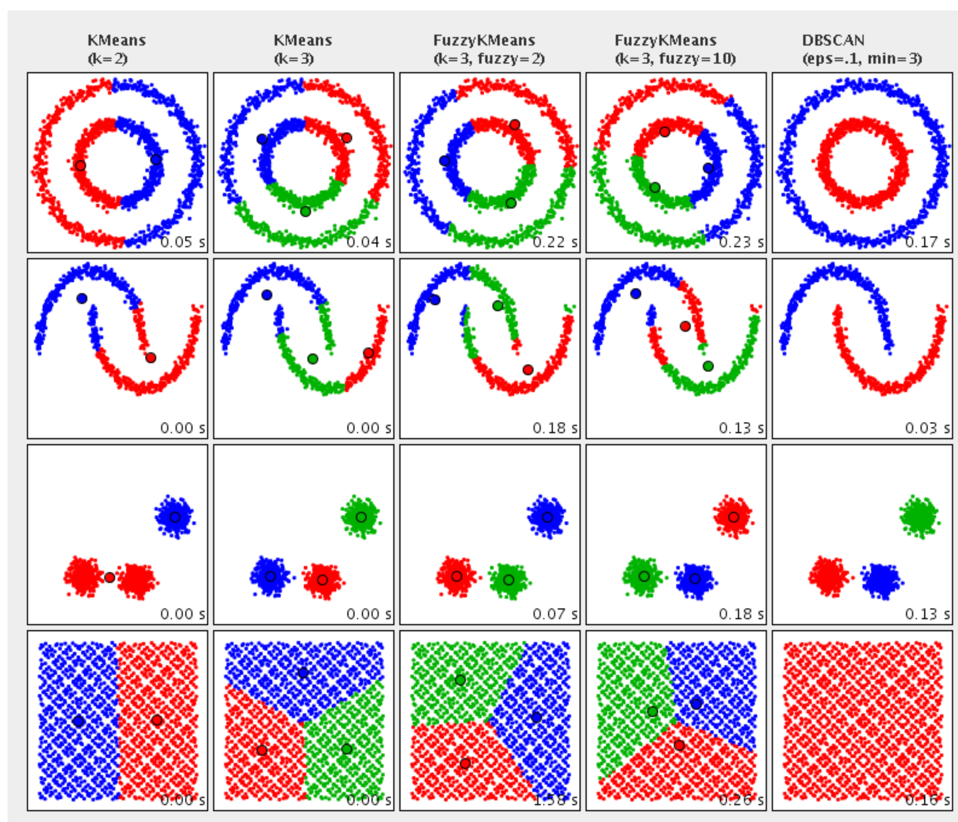


Figura 1. Comparación de *clusters* creados entre distintos algoritmos.

1.2 Motivaciones

La motivación para realizar este estudio surge del gran interés personal desarrollado a lo largo de los años de estudio por entender en profundidad la manera en que funciona una computadora en sus niveles más esenciales y básicos.

Para ello hay que conocer el modo en que el procesador realiza las operaciones y lo que en este caso es más importante y es donde se encuentran las mayores barreras actuales, los mecanismos del control y la sincronización con los distintos niveles de memoria.

Hay una infinidad de maneras de implementar cualquier algoritmo, pero esto no implica que será eficiente. La paralelización es uno de los métodos de optimización que puede ser un medio ideal para cumplir la motivación de este proyecto.

Conseguir encontrar un método para distribuir la carga computacional y de memoria entre distintos procesos que sean efectivas, hace aumentar la eficiencia de los procesadores. Que es en esencia el objetivo principal de este proyecto.

Paralelizar implica replicar instrucciones en distintos procesos. Referente a la memoria, mediante esta técnica nunca se puede conseguir un uso menor, sino solo aumentarlo. Es por ello por lo que el diseño de un algoritmo paralelo tiene que mantener el control de los datos que se distribuyen y que se comparten entre procesos para no sobrecargar los distintos niveles de memoria.

En cuanto a la distribución de los procesos, muchos algoritmos presentan grandes retos cuando hay que distribuir la carga computacional. Se debe encontrar una manera que aproveche el tiempo de cálculo de los distintos procesos para dar como resultado una ejecución global menor del algoritmo a la vez que se minimiza la sincronización entre ellos.

La paralelización del algoritmo *DBSCAN* presenta ser la ocasión ideal para alcanzar este objetivo. La manera en que este algoritmo crea las agrupaciones es visitando en serie punto tras punto en el espacio decidiendo a que agrupación pertenecen hasta terminar su ejecución.

Una implementación donde las iteraciones de estas visitas se fragmenten y realicen concurrentemente incurriría en que muchos puntos puedan a llegar a formar parte de varias agrupaciones a la vez y esto de resultados totalmente erróneos. O creando mecanismos de control costosos para asegurar que se agrupen de manera correcta en los que la versión paralelizada tendría una eficiencia igual o incluso menor que el algoritmo original.

Es por ello por lo que proponer una implementación paralela que obtenga como resultados los mismos valores de salida con un aumento notable del *Speed-Up* [1]supone un gran reto en cuanto al análisis e investigación para poder lograr este objetivo.

1.3 Objetivos

1.3.1 *Objetivos generales*

El objetivo general de este proyecto es, como se mencionó anteriormente, adaptar el algoritmo para que, partiendo de una implementación básica del *DBSCAN*, esta realice una ejecución en paralelo siguiendo distintos estándares de programación para poder ser usado en distintos tipos de computadoras consiguiendo un mayor Speed-Up; realizando un

[1] Número de veces que es más rápido el algoritmo paralelo en comparación a su versión secuencial.

posterior análisis de los resultados obtenidos para determinar si su implementación es escalable para resultar eficiente tratando volúmenes de datos masivos.

1.3.2 *Objetivos específicos*

Del mismo modo, existen varios motivos, tanto académicos como personales, por los cuales el desarrollo de este proyecto resulta de gran utilidad para un futuro tanto en ambos ámbitos como en el laboral.

1.3.2.1 Objetivos académicos

- Profundizar en conceptos teóricos relacionados con la arquitectura de computadores y la computación paralela y masiva.
- Adquirir conocimientos sobre algoritmos usados con frecuencia sobre machine learning y ciencia de datos.
- Realización correcta de análisis profundo de código y propuestas para nuevas versiones siguiendo diversos estándares.
- Evaluación y análisis críticos sobre implementaciones de algoritmos propios.

1.3.2.2 Objetivos personales

- Realizar un proyecto de búsqueda profunda por iniciativa propia.
- Ampliar horizontes para posibles futuros académicos.
- Desarrollar habilidades de análisis más objetivos orientados al ámbito profesional.
- Investigar sobre el trabajo previo realizado por otros investigadores para partir de un punto más avanzado y poder expandir más el conocimiento relacionado con el proyecto.

2 *Conocimiento previo al desarrollo*

Antes del inicio del análisis y la propuesta de paralelización, se deben conocer algunos conceptos previos referentes al algoritmo y las herramientas usadas para poder desarrollar el proyecto con precisión. Estos se presentan en este apartado.

2.1 Aprendizaje no supervisado

Los algoritmos no supervisados pertenecen a la rama del machine learning y son aquellos en los que la máquina recibe un aprendizaje automático del dominio del problema. Al contrario que el aprendizaje supervisado, la máquina no tiene ni un conocimiento previo sobre las características de los datos [1].

Los algoritmos de aprendizaje supervisado son aquellos que asocian las entradas y las salidas de datos. Requieren de etiquetas para identificarlos y de un entrenamiento por parte del usuario para poder realizar las ejecuciones. De esta manera es como predice el atributo deseado sobre el conjunto de datos.

En cambio, los no supervisados trabajan sobre conjuntos de datos sobre los cuales no se tiene conocimiento previo sobre el contenido, sus propiedad e interrelaciones. Para los cuales se espera encontrar algún patrón o algunos atributos desconocidos a simple vista. Por lo que estos atributos o etiquetas para los datos de entrada no se pueden contemplar.

Sus usos principales suelen ser en combinación con los supervisados para optimizarlos o formando agrupaciones de datos dado un conjunto.

Durante muchos años se les dio más importancia a los algoritmos de aprendizaje supervisado porque podían generar resultados concretos. Los no supervisados suponían un gran coste computacional por los algoritmos y por tener que ejecutar muchas pruebas variando parámetros al obtener resultados no concluyentes.

Pero en los últimos años con la aparición de algoritmos de clustering y el Deep Learning, se le está dando más importancia a la optimización de los algoritmos de aprendizaje no supervisade por su potencial en la minería de datos y sus aplicaciones futuras.

2.2 Clustering

El clustering pertenece a la familia de algoritmos de aprendizaje no supervisado. Esta es una herramienta de análisis estadístico que pretende crear grupos cerrados y homogéneos con similitudes entre si a partir de un conjunto de datos con diferentes características y propiedades. A estos grupos se les denomina *cluster*.

2.2.1 *Características*

Para ello, estos algoritmos tienen que cumplir una serie de características. Aunque hoy estas características no están estandarizadas, se dispone de las más comunes que los diseñadores más recurrentemente usan [2]:

- Las instancias dentro del cluster deben ser lo más similares posible entre ellas.
- Las instancias en diferentes cluster deben ser diferentes entre ellas lo máximo posible.
- Las medidas de similitud y diferenciación deben ser claras y tener un significado práctico.

2.2.2 *Diseño*

Cuando se debe diseñar un algoritmo de clustering, también es importante seguir varios pasos para asegurar la máxima eficiencia y efectividad posible [3]:

- Poder extraer los datos más relevantes del conjunto de datos.
- Diseñar o escoger el algoritmo de clustering acorde al problema que hay que solventar.
- Evaluar el resultado del clustering y juzgar la validez de los resultados.
- Ser capaz de dar una explicación a los resultados del clustering.

2.2.3 *Tipos de clustering*

En los tiempos actuales se dispone de muchas maneras distintas de clasificar los algoritmos de clustering. Aquí mencionaremos los tipos de clasificaciones más comunes [10]:

- **Basados en particiones**: Estos algoritmos se basan en relocalizar los puntos en el espacio de datos de manera iterativa entre los distintos clusters intentando minimizar así la diferencia entre ellos logrando así particiones locales optimas. Algunos de las más usados son el *K-Means* que se basa en centroides o *K-Mediods*. Su gran ventaja es la baja complejidad computacional, aunque cuenta con algunas desventajas como no ser apto para formas no convexas o estar sujeto a tener que definir el número de clusters deseado.
- **Jerárquico**: La implementación más típica de los algoritmos de clustering jerárquicos se basa en partir de que cada punto constituye un cluster por sí solo. Estos puntos se juntan con instancias similares formando clusters nuevos y, siguiendo así de manera recursiva hasta crear un cluster que los englobará a todos. Este método también se puede hacer de manera inversa. Su uso tiene grandes ventajas para el análisis de datos con formas arbitrarias y dispone de un alto grado de escalabilidad. Aunque como desventajas disponemos de una gran complejidad computacional y el número de clusters como en el apartado anterior, debe ser establecido.
- **Distributivo**: La idea de esta implementación erradica en que los datos se tienen que distribuir con alguna norma o función, y estos son los que decidirán a que cluster pertenece cada conjunto de datos que satisfaga los requisitos. La mayor ventaja es que presenta resultados más realistas a un conjunto de datos que cumplan la función de distribución y tiene la opción de tener un gran grado de escalabilidad. Sus desventajas son el ser poco intuitivo al tener que estudiar un elevado número de parámetros de entrada para asegurar una correcta ejecución del algoritmo y la alta complejidad computacional.
- **Basado en densidades**: Estos algoritmos se centran en separar los datos en regiones espaciales donde su densidad de instancias sea mayor, formando así los clusters en función de las regiones más densas. El algoritmo sobre el que se trabaja en este proyecto, el DBSCAN, es el más usado para este tipo de clustering. Sus ventajas son su baja complejidad computacional y la posibilidad de encontrar clusters con formas arbitrarias. También tiene las desventajas de no tener resultados fiables cuando los puntos están muy esparcidos en el espacio o los parámetros no se han ajustado de manera adecuada y, el gran uso de memoria a medida que la cantidad de datos aumenta.

2.3 DBSCAN

El *DBSCAN* (Density Based Spatial Clustering of Applications with Noise) es un algoritmo de agrupación de datos basado en densidades que encuentra un número no determinado de clusters hasta el final de su ejecución. Este número viene definido por tres parámetros [4]:

- El número mínimo de puntos que debe tener para considerarse cluster.
- Una variable que marca el rango de distancia máxima a la que pueden estar los puntos entre sí para pertenecer al mismo cluster.
- La función que se usa para calcular esa distancia.

Hay que conocer la manera en que el algoritmo funciona en su implementación y ejecución secuenciales tal y como fue propuesta por los autores en *la conferencia líder de la minería de datos* en el año 1996 ya que esa será la base en que aquí se partirá.

Se introdujo para solventar tres principales problemas de la época:

- Accesible para todos teniendo conocimiento mínimo del dominio del problema para proporcionar los parámetros de entrada.
- Descubrir clusters con formas arbitrarias en el espacio. Este puede encontrar clusters circulares, cónicos, alargados, etc.…
- Eficiencia adecuada para bases de datos de gran volumen.

Cumpliendo esa definición, dispone de puntos que tienen la posibilidad de no pertenecer a ningún cluster y a esta funcionalidad se le llamará *Detección de ruido*. Especialmente útil para la identificación de resultados anómalos o poco frecuentes.

2.3.1 *Parámetros*

Para solucionar el problema de los complejos algoritmos que requieren de mucho conocimiento y análisis previos del problema introducidos en el *apartado 2.2*, este algoritmo cuenta con una serie de parámetros sencillos y relativamente fáciles de determinar expuestos en este apartado.

2.3.1.1 Dataset

Se refiere al conjunto de datos denotados por *D* que contienen todos los puntos en el espacio y que puede contemplar *K*-dimensiones mientras se puedan cumplir los requisitos de la *función distancia* entre puntos.

Este parámetro es tanto de entrada como el de salida con la diferencia de que, una vez terminada la ejecución, los puntos en *D* tendrán un nuevo valor marcando a que cluster pertenecen, el *clusterID*.

2.3.1.2 Epsilon

Este valor especifica al algoritmo la máxima distancia a la que pueden estar los puntos entre sí para que se consideren vecinos[2] unos de otro y puedan así pertenecer al mismo cluster.

Este es el único que puede ser algo difícil de determinar y existen varias heurísticas para ello. La manera más habitual de hacerlo es de manera automatizada sin necesidad de un análisis previo con un algoritmo propuesto por los mismos autores del DBSCAN.

Esto se hace calculando un numero de *K-Vecinos* más próximos a un punto para cada punto del conjunto de datos con una función de distancia. Una vez se obtienen las distancias, estas se ordenan con los valores obtenidos en función del número de puntos analizados como podemos observar en la **Figura 2**.

La decisión del valor de *K* viene determinada por el usuario. Cuanto mayor sea, más preciso será el resultado, pero exigirá de un mayor poder computacional. Se recomienda como norma general que este sea igual o superior al número de dimensiones en el conjunto de datos [5].

Una vez obtenemos los resultados ordenados, el valor de *Epsilon* se determinará donde se pueda apreciar un cambio brusco en los resultados para las distancias. La explicación a este hecho es que esta grafica es una representación de las densidades en función del número de puntos. Si cogemos un valor por debajo de este cambio, tendremos muchos clusters de un tamaño pequeño y, si cogemos un valor de distancia mayor, este englobara más puntos formando pocos clusters de mayor tamaño.

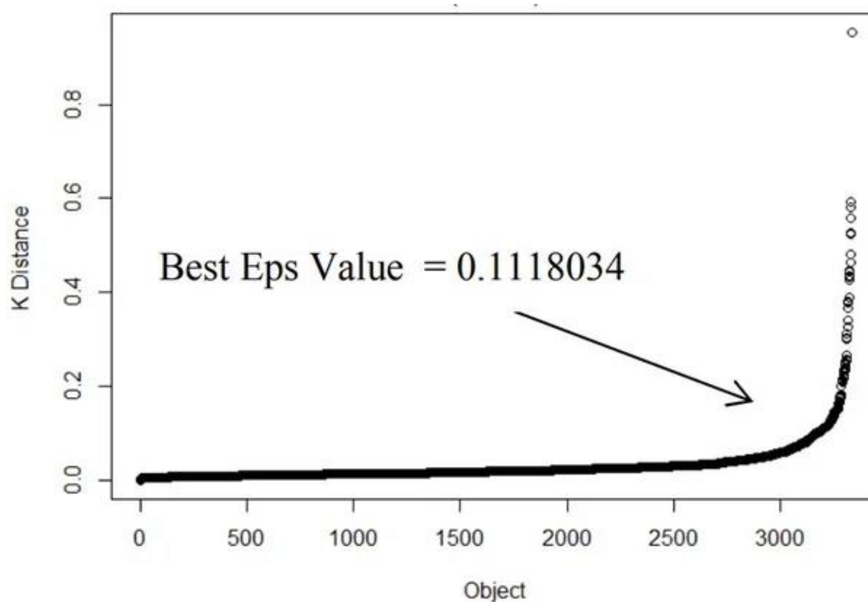


Figura 2. Puntos ordenados por distancia a los tres vecinos más cercanos.

[2] Conjunto de puntos para los cuales se cumple que la distancia es igual o menor a Epsilon para un punto determinado.

2.3.1.3 Mínimo de puntos

El mínimo de puntos viene ligado de manera directa a la densidad de un cluster ya que es el que determina el número mínimo de puntos que debe tener un cluster para ser considerado como tal.

La decisión de este valor viene dada de manera ideal por el conocimiento del dominio del problema. No obstante, existe una regla general para determinarlo, la cual está relacionada con la cantidad de dimensiones en el conjunto de datos. Esta dice que se debe establecer un valor mayor o igual al número de dimensiones más uno.

Para conjuntos de datos masivos con muchos puntos clasificados como ruido se recomienda que este valor sea el doble del número de dimensiones para solventar el posible aumento de distancia espacial entre puntos.

2.3.1.4 Función distancia

Es la función con la que se calculará la distancia métrica entre puntos. Del mismo modo que el parámetro anterior, este se decide en función del dominio del problema. Pero la más usada y recomendada es la función de distancia euclidiana representada por la siguiente función:

$$euclidean_dist = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2}$$

2.3.2 *Estructura de datos*

Para la ejecución del algoritmo es necesaria establecer las estructuras de datos necesarias para poder llegar a identificar los diferentes clusters y a cuál pertenece cada conjunto de datos.

2.3.2.1 Punto

El punto es una única instancia dentro del conjunto de datos. Esta se compone de:

- Posición espacial.
- Identificador de cluster.

Inicialmente, al cargar la estructura de datos en la memoria para la ejecución, todos estos identificadores se marcan con un valor que identifiquen que no pertenecen a ningún cluster. Ya que se determina que el algoritmo debe finalizar una vez que todos los puntos tengan un cluster asignado o sean ruido. Por lo que no debe quedar ninguno con este atributo inicial.

La manera en que estos se almacenarán en memoria durante la ejecución viene a elección del programador. En este caso se tratará el almacenamiento dinámico en memoria según la cantidad en el conjunto de datos.

2.3.2.2 Vecindad Epsilon

Esta estructura se usa para marcar las regiones densas dentro de la estructura de datos. En ella se almacenan todas las instancias que cumplen los requisitos de distancia entre puntos para un punto concreto y comprobar si podrán pertenecer a un cluster.

Dentro de esta vecindad existen dos tipos de puntos con requisitos distintos para poder entrar a formar parte de ella:

- **Núcleo:** Es el punto desde el que se parte la búsqueda y se considera con suficiente densidad para ello, se puede ver en color rojo en la **Figura 3**. Para que sea núcleo, este debe de contener dentro del rango de distancias Epsilon una cantidad de puntos igual o mayor al parámetro mínimo de puntos.
- **Frontera:** Son la serie de puntos denotados por el color amarillo en la **Figura 3** que se encuentran dentro del rango Epsilon de un punto núcleo, pero, dentro de su propio rango no cumplen el criterio de densidad para ser núcleos. Estos entrarán a formar parte dentro del mismo cluster al que pertenece el núcleo, pero, nos dará a entender que en esta dirección el cluster no se expandirá más.

Los puntos que no cumplen ninguno de estos requisitos en ningún momento, no pertenecerán a ninguna vecindad por lo que serán clasificados en el conjunto de datos como ruido, marcado en color azul en la **Figura 3**.

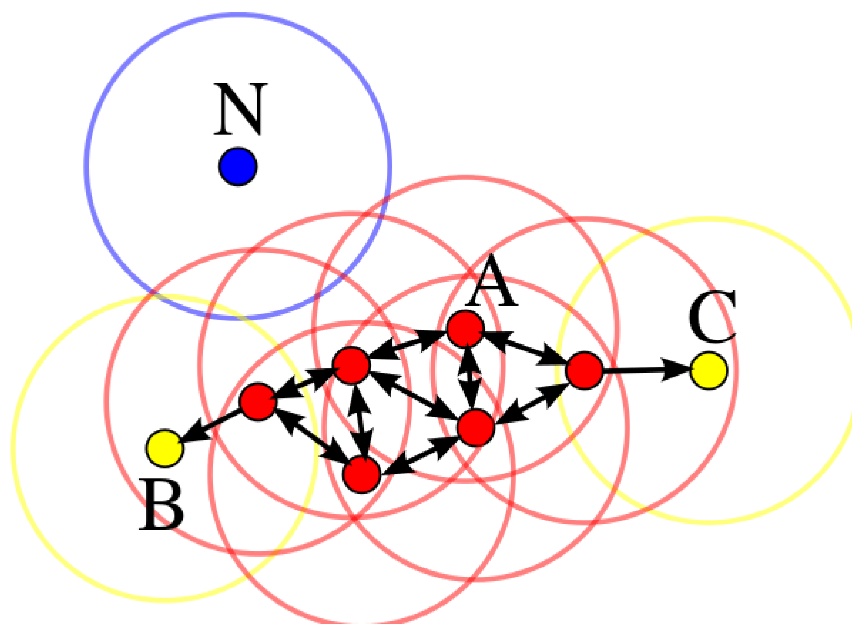


Figura 3. **Tipos de puntos en la *Vecindad Epsilon* con parámetros *Epsilon* marcado por circunferencias y *mínimo de puntos* igual a cuatro.**

2.3.3 *Algoritmo*

Como se mencionó al inicio del *apartado 2.2*, este proyecto partirá de la implementación del algoritmo propuesta por los mismos autores en el **Pseudocódigo 1**.

Una vez explicados los conocimientos necesarios referente al DBSCAN, se procederá a explicar la mecánica del algoritmo.

```
DBSCAN (DataSet, Epsilon, MinPts)
      //DataSet inicializado a no_clasificado
      ClusterId := nextId(NOISE);
      FOR i FROM 1 TO DataSet.size DO
            Punto := DataSet.get(i);
            IF Punto.CLusterId = no_clasificado THEN
                  IF ExpandCluster (DataSet, Punto,
                                    ClsuterId, Epsilon, MinPts) THEN
                        ClusterId := nextId(ClusterId);
```

```
                END IF
            END IF
        END FOR
END; //DBSCAN
```

Pseudocódigo 1. DBSCAN.

El algoritmo comienza buscando sobre todos los puntos en el conjunto de datos en la búsqueda de un punto que tenga suficiente densidad para poder considerarlo como comienzo de un nuevo cluster. Se considerará a un punto suficientemente denso si tiene una cantidad igual o mayor al número de puntos dentro de su rango de distancia Epsilon, es decir, que en su Vecindad Epsilon haya al menos la cantidad establecida por el parámetro *MinPts*.

Esto se realiza iterando uno a uno sobre ellos y consultando si el identificador del cluster es no definido, lo que implica que no pertenece aun a ningún cluster y tampoco es un punto ruido.

Una vez encontrado, se comprueba que el punto disponga de suficiente densidad para poder comenzar un nuevo cluster a partir de él.

Si tiene suficiente densidad, se crea un cluster y se expande hasta añadir a todos los puntos que pertenezcan a él.

Si no tiene suficiente densidad, el punto será marcado como ruido para ser tratado posteriormente como posible vecino de otro punto y pertenecer a otro cluster.

Una vez creado un cluster, se realiza el mismo procedimiento al siguiente punto creando un cluster nuevo. Repitiendo las iteraciones hasta haber visitado todos los puntos y no quede ninguno con un identificador de cluster no definido.

Dentro de este algoritmo, la función más importante es *ExpandCluster* que se encarga de añadir los puntos que cumplan con los requisitos de distancia partiendo de un punto núcleo que se puede ver en el **Pseudocódigo 2**.

```
ExpandCluster (DataSet, Punto, ClsuterId, Epsilon, MinPts): Boolean;
      seeds := DataSet.regionQuery(Point, Eps);
      IF seeds.size < MinPts THEN //No Core Point
            DataSet.changeClsuterId(Punto, NOISE);
            RETURN false;
      ELSE  //Todos los puntós en “seeds” entran dentro de la densidad
            //alcanzable por el Punto
            DataSet.changeClusterID(seeds, ClusterId);
            Seeds.delte(Punto);
            WHILE seeds no este vacio DO
                  pntActual := seeds.first();
                  resultado := DataSet.regionQuery(PntActual, Epsilon);
                  IF resultado.size >= MinPts THEN
                        FOR i FROM 1 TO resultado.size DO
                              resPnt := resultado.get(i);
                              IF resPnt.ClusterId
                                    IN {no_clasificado, NOISE} THEN
                                    IF resPnt.ClusterId =
                                          no_clasificado THEN
                                          seeds.append(resPnt);
                                    END IF;
                                    DataSet.changeClusterId(resPnt,
                                                            clusterId);
                              END IF; // no_clasificado o NOISE
                        END FOR;
                  END IF; //resultado.size >= MinPts
```

```
                seds.delete(pntActual);
            END WHILE; //seeds vacio, todos los puntós se han tratado
            RETURN true;
        END IF
END; //ExpandCluster;
```

Pseudocódigo 2. Función de expansión de cluster.

Lo primero que se hace es realizar una consulta a la estructura de datos para determinar cuáles son los que cumplen con el requisito de distancia Epsilon hacia el punto especificado y almacenarlos en la estructura llamada *seeds*.

Si la cantidad de puntos es menor a la que indica el parámetro *MinPts*, esto nos indicara que el punto no tiene suficiente densidad y por lo tanto se clasificará como punto ruido. La ejecución de la función expansión se detendrá y se procederá a procesar la del siguiente punto en el conjunto de datos.

En este caso la función devolverá el valor *falso* indicando que no se debe variar el identificador del cluster porque no se creó uno.

Si la cantidad es igual o mayor al parámetro *MinPts*, el punto será clasificado como núcleo y será añadido al cluster actual.

Para todos los puntos que pertenecen a la estructura de vecinos, se procesara de nuevo la consulta de región de vecinos para ellos.

Si al procesarlos se encuentra que su cantidad de vecinos es mayor o igual al parámetro *MinPts*, se considerará como otro punto núcleo y la estructura encontrada se concatenará con la estructura *seeds* del punto original. Este punto será añadido al cluster actual como punto núcleo.

Si es menor, se considerará como un punto frontera y se añadirá al cluster actual.

Una vez terminadas las iteraciones, la función devolverá el valor *verdadero* indicando que se ha creado un nuevo cluster y se puede proceder a la creación del siguiente.

Con esta implementación podría llegar a pasar que un punto se pudiese clasificar como frontera en más de un cluster y por la definición del conjunto de datos, este no puede pertenecer a más de uno. No obstante, los estudios demuestran que no es un problema que tenga un impacto significativo en los resultados, por lo que este punto pertenecerá al primer cluster que lo añadió a su conjunto.

Para la propuesta de paralelización de este proyecto se usará la implementación del licenciado del MIT Gagarine Yaikhom [11] que realiza este proceso sobre un espacio vectorial de tres dimensiones en lenguaje de programación *C*.

2.3.4 *Evaluación*

Se presentarán aquí las complejidades espaciales y de computación que presenta este algoritmo en la implementación secuencial expuesta en el apartado anterior.

2.3.4.1 Complejidad espacial

El mayor uso de memoria que se produce en el algoritmo es al procesar el conjunto de datos de entrada que forzosamente será de $O(n)$ al tener que almacenar de manera constante todos los puntos y este será un coste fijo.

En cuanto a la ejecución, el algoritmo trata de manera muy efectiva la memoria ya que solo se reservan espacios en el momento de encontrar la Vecindad Epsilon en la iteración

actual y, una vez procesados los nodos y teniendo asignado un numero de cluster, estos espacios de memoria se liberan.

La complejidad de esta rutina varía en función de la heurística usada para encontrar la vecindad. En una implementación simple como la contemplada en este proyecto puede ser arbitraria en función de la distancia entre los parámetros de entrada. En el peor de los casos podría llegar a ser $O(n)$ si se llegase a formar un solo cluster con todos los datos del conjunto. Pero es un caso raramente contemplable, por lo que sería más razonable asignarle una complejidad de $O(k)$, donde k es un numero arbitrario en función del conjunto de datos, pero se espera que sea pequeño según los mismos creadores del algoritmo.

Por lo que en nuestro caso podemos determinar que la complejidad espacial total será de $O(n+k)$.

2.3.4.2 Complejidad computacional

La complejidad computacional esta estrictamente ligada a la manera en que se implementan las funciones que consultan las regiones que encuentran la *Vecindad Epsilon*, la manera que esta implementada la estructura de datos del conjunto de entrada ya que es desde el cual se consultan los puntos, y a la heurística que se usa para calcular la distancia entre los puntos.

El orden en que se estructuran los puntos en el conjunto de datos en este proyecto es arbitrario ya que se almacenarán en un vector en el orden en que se leen originalmente del archivo de entrada. Por lo que el mayor coste computacional y el que marcara la eficiencia del algoritmo en su totalidad se encuentra calculando las distancias entre puntos. Al tener que realizar esta consulta para cada punto su complejidad computacional será de $O(n^2)$.

Algunas implementaciones en serie más optimas usan estructuras de datos como el R*-*tree*[3]. Gracias a estas estructuras de datos, las funciones de consulta de regiones para vecinos pueden llegar a tener una complejidad tan baja como $O(log\ n)$ por llamada. Lo que nos daría una complejidad computacional total de $O(n * log\ n)$.

2.3.5 *Usos prácticos*

Aun habiendo sido presentado al mundo en una fecha tan lejana como 1996, el algoritmo DBSCAN sigue estando en uso para un gran número de aplicaciones actuales y puede abarcar un espectro de campos de trabajo muy amplio.

Tanto es así que, en 2014 este algoritmo fue galardonado con un premio a la prueba temporal en la SIGKDD[4], que es la organización encargada del estandarizado internacional para los conceptos de minería de datos. Este se otorga a algoritmos que tuvieron un impacto muy relevante entre la comunidad de investigadores en este campo [12].

Entre algunos de sus usos potenciales se tienen los siguientes ejemplos que destacan por efectividad o incluso por su curiosidad:

---

[3] Estructura de datos en forma de árbol que organiza los elementos según la distancia espacial.

[4] Grupo de interés especial sobre descubrimiento de conocimiento y minería de datos de la asociación de maquinaria de computación. [14]

- Prevenir la rotura de tuberías en el sistema urbano mediante un clustering de tuberías rotas en el pasado. Para poder así identificar patrones de comportamiento o situaciones similares entre ellas [13].
- Haciendo alusión a la situación actual con el COVID-19, se puede usar para identificar clusters de densidad de población infectada y poder determinar el tratamiento a seguir por zonas según su densidad.
- En el procesado de imágenes, como separar los histogramas de colores de una imagen para poder categorizar diversas partes dentro de ella. Teniendo entre otros usos, el del análisis de radiografías [6].
- Diversas aplicaciones médicas. Como por ejemplo realizando clusters con síntomas de enfermedades para saber cuáles son los que más se repiten para alguna afección en concreto. De este modo se podrá establecer el mejor tratamiento o el que pueda ayudar a una mayor parte de los afectados [7].
- Detectar focos de actividad urbanística creando clusters de diversos patrones de datos como tiendas visitadas, lugares donde más tiempo se ha estado, etc… Útil para actividades publicitarias, urbanismo y control demográfico.

Estos son solo unos pocos ejemplos para representar el amplio abanico de posibilidades que ofrece este algoritmo y así poder tener una pequeña visión de los potenciales usos reales que este ofrece.

2.4 Multiprocesador

Los multiprocesadores son sistemas que cuentan con dos o más microprocesadores y tienen un espacio de memoria principal compartida.

Estos surgen para solventar el problema del aumento del coste energético al crear procesadores más potentes a base de aumentar la frecuencia. Para combatirlo se opta por añadir núcleos con frecuencias más reducidas que realicen tareas[5] concurrentemente.

Para diseñarlo, se replican todos los componentes de un procesador común en una misma oblea y estos tendrán todos los elementos privados en su arquitectura, hasta la memoria principal que será compartida. O el ultimo nivel de la memoria cache en los últimos años.

Así a estas máquinas se las clasifica con una arquitectura *MIMD*[6](Multiple Instruction, Multiple Data) en la cual se pueden procesar distintas operaciones sobre un flujo de distintos datos a la vez. Representado en la **Figura 4**.

[5] Conjunto de instrucciones para la máquina dentro del lenguaje de programación.

[6] Arquitectura donde el computador procesa diversos flujos de instrucciones con múltiples flujos de datos concurrentemente.

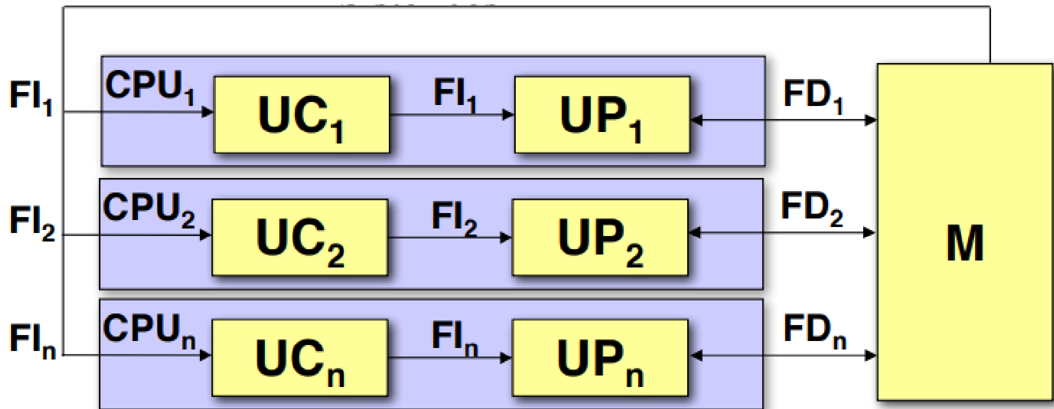


Figura 4. Flujo de datos e instrucciones para arquitecturas MIMD [33].

En el caso de tener que modificar un dato o enviárselo a otro proceso, este procedimiento se realizaría mediante cargas y escrituras en memoria principal.

Los multiprocesadores se clasifican por como acceden a los recursos compartidos y hay de dos tipos:

- **Uniform Memory Access(UMA):** Este les da un acceso uniforme a todos los núcleos al espacio de memoria compartido y para ello se tiene que realizar la transferencia mediante un medio de interconexión como se ve en la **Figura 5**. El bus de datos es el más común para este tipo de máquinas. Resulta más eficiente, pero asimismo es poco escalable debido a la congestión del bus por el tráfico.
- **Non Uniform Memory Access(NUMA):** En este modelo, la memoria principal compartida se fragmenta y se destina una sección de memoria privada para cada uno de los núcleos del multiprocesador como se ve en la **Figura 6**. Es en el momento en el que es necesario comunicar datos cuando se hace uso de la red de interconexión. Gracias a reducir la congestión en el tráfico de datos, esto hace a estos sistemas altamente escalables.

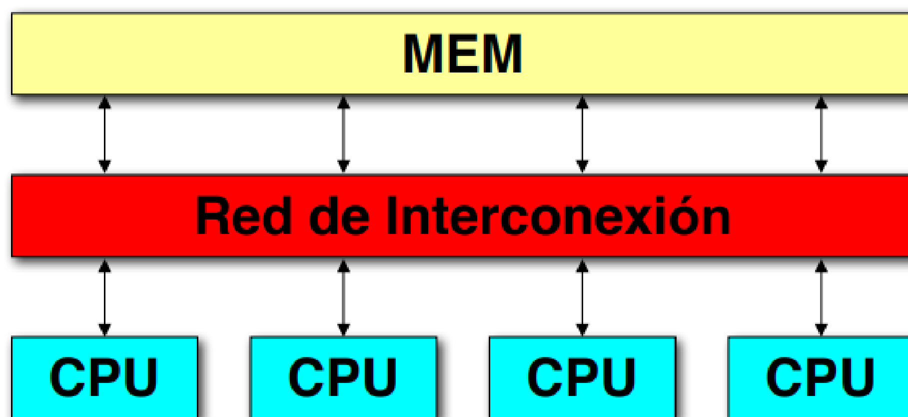


Figura 5. Acceso a memoria compartida UMA [33].

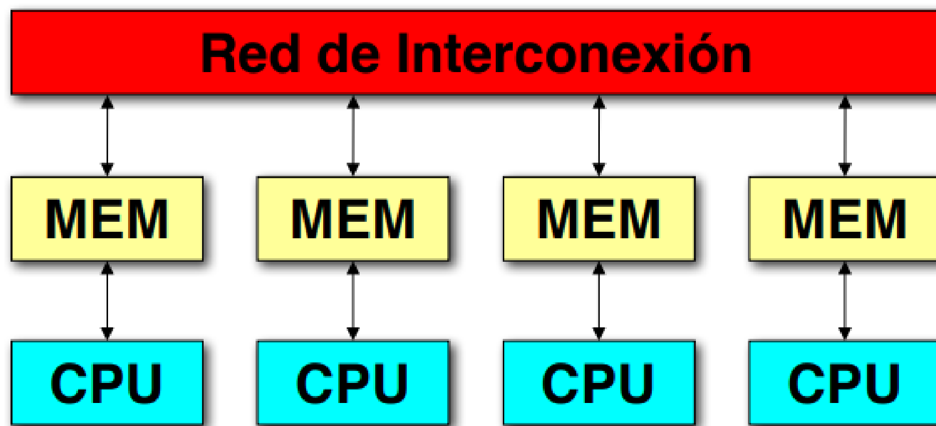


Figura 6. Acceso a memoria compartida NUMA [33].

2.5 OpenMP

Es una API[7] para la paralelización de procesos en multiprocesadores. Esta permite la paralelización de programas escritos en lenguajes *C, C++* y *Fortran*.

El desarrollo de aplicaciones mediante este estándar se realiza a alto nivel con una serie de directivas propias las cuales el compilador se encarga de transformar en ejecución paralela al lenguaje máquina [15].

Dispone de una sincronización entre procesos muy rápida ya que los procesos se comunican mediante el bus de datos y esta resulta muy eficiente. Pero, por el contrario, la limitación de espacio que se tiene en un mismo chip hace que su escalabilidad no sea muy alta.

2.5.1 *Modelos de ejecución*

2.5.1.1 Fork-Join

El modelo de ejecución en el que se basa este estándar es el de fork-join que consiste en fraccionar el problema en una serie de tareas paralelas que se ejecutarán en distintos hilos (fork) para luego ser recolectados por el hilo maestro (join).

Esto implica que al final de cada sección paralela debe haber una barrera para asegurar la sincronización de los distintos hilos y que todos hayan terminado el procesado antes de proseguir con la ejecución en serie del algoritmo.

2.5.1.2 Tareas

OpenMP también admite un modelo de tareas ejecutando códigos totalmente distintos, haciendo que cada bloque especificado con sus directivas se procese por un hilo. Estas ejecuciones se hacen de manera asíncrona sin necesidad de sincronización. Tampoco es necesario especificar identificador del hilo ni la cantidad ya que es el sistema operativo quien reordena los procesos creados en función de sus necesidades.

[7] Interfaz de programación de aplicaciones.

2.6 Multicomputadoras

Las multicomputadoras son sistemas de cálculo en el que cada computadora tiene un espacio de memoria privado y distribuido. Esta distribución puede ser en un mismo terminal, en un mismo emplazamiento o incluso repartidos por el mundo.

La manera en que los procesos se comunican entre ellos es mediante el paso de mensajes a través de un controlador de interfaz de red usando protocolos de red como se ve en la **Figura 7**.

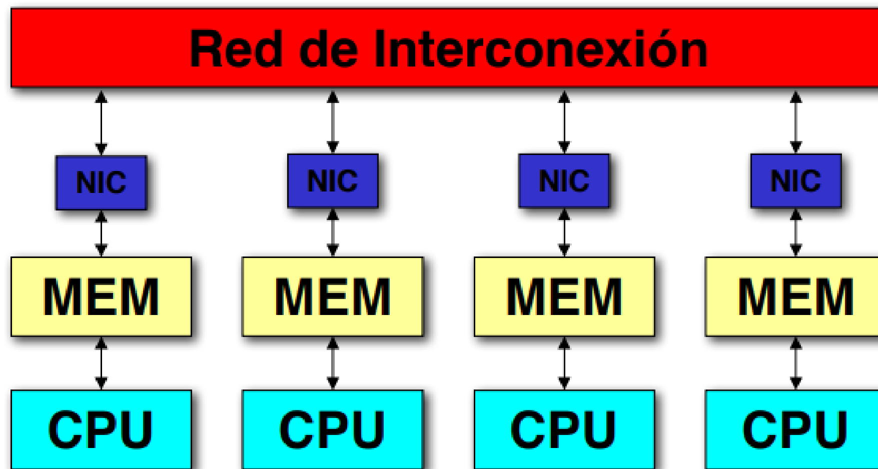


Figura 7. Sistema de paso de mensajes para multicomputadoras [33].

Estos, como los multiprocesadores, hacen uso de una arquitectura MIMD en los que cada máquina tiene su propio flujo de datos e instrucciones independientes.

Tienen la ventaja de tener una gran escalabilidad al aprovechar todos los recursos de una computadora, incluida la opción de ejecutar la paralelización mediante sus multiprocesadores si los tuviese. El gran reto en estos sistemas es el de la mejora de las redes de interconexión ya que en raras ocasiones esta será un bus de datos.

2.7 OpenMPI

OpenMPI es un estándar de programación implementado en código abierto cuyo objetivo es el de desarrollar aplicaciones para multicomputadoras [16]. Un claro ejemplo de uso practico son las supercomputadoras, donde se implementa la arquitectura vista en el apartado anterior.

Este estándar basa su modelo de ejecución en la paralelización de procesos que ejecutan algoritmos o instrucciones independientes. Para que estos procesos se coordinen y sincronicen en los momentos necesarios se usa el paso de mensaje entre ellos mediante un controlador de interfaz de red.

Gracias a esto, resulta en una arquitectura con unas opciones de escalabilidad muy altas y mayores a los de los multiprocesadores.

Por lo contrario, nos encontramos de que el paso de mensaje requiere de protocolos muy costosos temporalmente. También puede pasar que algunos nodos[8] no hagan uso de su

[8] Unidad de computación en una máquina multicomputadora con una memoria privada distribuida.

potencia de cálculo estando a la espera de la recepción de algún mensaje en el caso de los protocolos bloqueantes con código mal optimizado. Por ello este estándar es más difícil de programar y puede no resultar adecuado para muchos tipos de programas.

2.8 CUDA

Al contrario de las arquitecturas *MIMD* necesarias para las interfaces de programación vistas en los *apartados 2.4 y 2.6*, usando el entorno CUDA, lo que se trata es adaptar el código a su ejecución en una unidad de procesado grafico que trabaja con operaciones escalares con arquitectura *SIMD*[9]. Aunque estos dispositivos disponen de arquitectura MIMD combinada, de lo cual se hablará en el siguiente apartado.

Por lo que CUDA se define como la API para lenguajes *C* y *C++* que nos permite realizar programas para que estos sean transmitidos y ejecutados por la unidad de procesado gráfico [17].

2.8.1 *Arquitectura*

El uso de la unidad de procesado grafico es especialmente útil por los cambios en la arquitectura que tiene con respecto a los de un procesador común.

Los procesadores invierten la mayoría de los transistores en el chip dedicados a memoria cache, unidades de predicción, controles de memoria y lógicos, etc… Todo ello es de gran utilidad para acelerar las máquinas con un único flujo de instrucciones.

La arquitectura de las unidades de procesado grafico cambian completamente esta heurística. La manera más fácil de visualizarla es, como dicen Dominik Göddekke y Robert Strzodka, la de eliminar todos los elementos necesarios para la optimización de un único flujo de instrucciones [18].

Cuando realizamos esta modificación, obtenemos un gran número de transistores libres que pueden ser dedicados a la asignación de unidades aritméticas[10] para el cálculo de operaciones. Se puede observar una comparativa grafica de este cambio en la **Figura 8**. Estas están especializadas en una computación intensiva y altamente paralelizable de operaciones aritméticas.

[9] Arquitectura donde el computador procesa mediante un único flujo de instrucciones, un conjunto múltiple de datos. Usado también en los procesadores vectoriales.

[10] Porción de la unidad funcional del procesador que se encarga de ejecutar las operaciones lógicas.

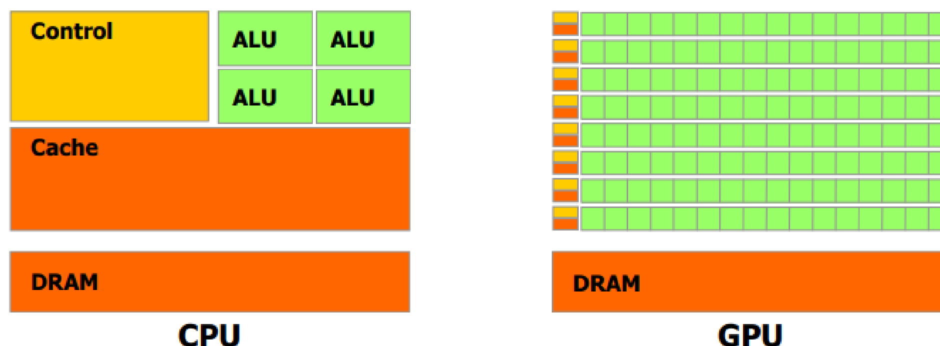


Figura 8. Comparación en el uso de transistores entre CPU y GPU [17].

Obtenemos así la arquitectura SIMD donde una unidad carga una instrucción a realizar en una porción de la unidad de control y la ejecuta sobre un vector de unidades aritméticas con distintos datos en cada una. Estas unidades aritméticas se pueden considerar como hilos independientes de ejecución en una máquina multiprocesador.

Esta unión entre la unidad de control y un vector de unidades aritméticas se le asignara el término de *bloque*.

Al realizarlo de este modo, no es necesario disponer de sofisticados controles de flujo que ralenticen la ejecución de una instrucción. La latencia que se produce en un cambio en el contexto de la ejecución se alivia mediante la ejecución de bloques alternados. Mientras un bloque está ejecutando una operación, otro está preparando el siguiente contexto de ejecución para ser realizado inmediatamente cuando acaba el previo.

Para poder implementar este nuevo paradigma, los dispositivos de procesado grafico más actuales introducen un mecanismo de control de flujo llamado *SMX*(Next Generation Streaming Multiprocessor). Este se encarga de encapsular los bloques en grupos de 32 llamados *Warp*, los cuales pueden enviar instrucciones a distintos bloques y ordenan su ejecución. Es aquí donde el diseño de este dispositivo se combina con la arquitectura MIMD en el caso de que el dispositivo disponga de más de un SMX.

Aunque este cambio no implica que la memoria de acceso rápido sea eliminada del todo. En las jerarquías de memoria de las unidades de procesado grafico se sigue teniendo transistores reservados para la memoria, aunque estos sean más pequeños como se puede ver en la **Figura 9.**

Cada hilo tiene un espacio privado, cada bloque tiene un espacio de memoria compartida entre hilos y, por último, existe una memoria global que hace la misma función que la memoria principal de un computador. Además, esta memoria global también es la que se usa para la comunicación de datos con el procesador.

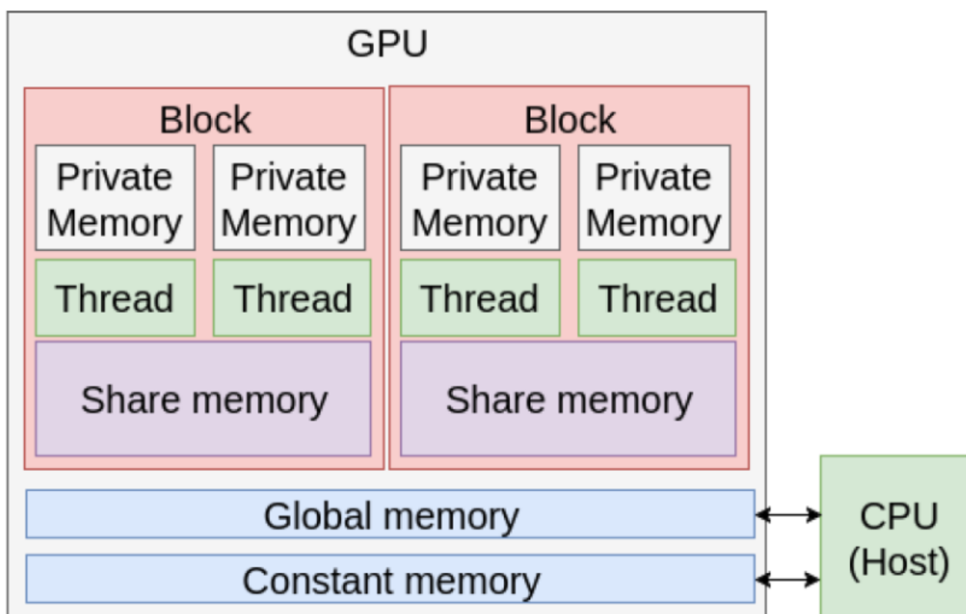


Figura 9. Jerarquía de memoria de unidad de procesado gráfico [17].

2.8.2 *Modelo de programación*

El modelo de programación de CUDA está adaptado por el fabricante para que sea escalable a cualquier unidad de procesado gráfico en combinación con cualquier procesador, ya que el mercado exige que estas se renueven y sean más potentes año tras año.

El modelo de ejecución resulta muy parecido al del explicado en la *sección 2.3.1* referentes a OpenMP. Este admite un paralelismo de hilos similar al modelo *fork-join* o un paralelismo por ejecución de tareas lanzadas dentro de su contexto.

Según el propio fabricante, la programación en CUDA se diseñó para ser especialmente sencilla. Asegurando que para poder realizar un programa solo es necesario estar familiarizado con los conceptos de bloque de hilos, memoria compartida y sincronización por barrera.

Comprobando su certeza ya que, para poder ejecutar cualquier algoritmo o función dentro de la unidad de procesado gráfico, solo se necesita saber cuántos hilos se necesitarán lanzar para ser computados. En términos más sencillos, este número de hilos se podría asignar según el número de iteraciones de un bucle que queremos paralelizar. Gracias a la interfaz que esta implementa, la propia unidad de procesado grafico organizará las ejecuciones de la manera más óptima posible.

3 Análisis del problema

Para proceder con la propuesta de paralelización del DBSCAN, primero hay que realizar un análisis previo del algoritmo con los puntos más críticos al realizar cualquier tipo de optimización.

Inicialmente se aplicará un estudio analítico del número de llamadas a las funciones y al tiempo de ejecución de estas para poder establecer si existen algunas partes del algoritmo en las que tengamos que poner especial atención o centrar el esfuerzo para solucionar la carga computacional.

Seguidamente se realizará un estudio de las dependencias de los datos que tienen las tareas entre sí en una ejecución secuencial. Esta pretende mostrar cómo se escriben y leen los datos para poder determinar que partes se pueden paralelizar y que técnicas se pueden aplicar para que esta se ejecute sin errores.

3.1 Profiling

Se procede a hacer uso de la técnica de *profiling* la cual se utiliza para el análisis de la ejecución de un algoritmo. Esta herramienta es especialmente adecuada a la hora de encontrar cuellos de botella, sobrecarga de memoria o de computación e incluso errores en el código algoritmo.

En este caso, analizaremos un conteo de las llamadas a las funciones y también del tiempo que se dedica en procesar cada una de ellas. Para ello se usará la herramienta *Gprof* para entornos Linux y código escrito en lenguaje de programación *C*. De este modo se podrá saber con certeza los puntos en los que centrarse a la hora de paralelizar el algoritmo.

3.1.1 *Parámetros profiling*

Como fue explicado en los apartados anteriores, la eficiencia de este algoritmo varía mucho en función del conjunto de datos de entrada ya que para cada punto en el espacio hay que calcular la distancia hacia todos los demás.

Por ello para este tipo de análisis se crearán un conjunto de puntos aleatorios dentro de un espacio delimitado en tres dimensiones. Los parámetros *Epsilon* y *MinPts* se mantendrán fijos con valores seguidos por las recomendaciones expuestas en el *apartado 2.2.1*. Lo que se variara para las pruebas es la cantidad de puntos dentro del conjunto de datos y para cada uno de ellos se realizará una ejecución del algoritmo *DBSCAN*.

A continuación, se muestra una lista con los parámetros fijos y la variación de cantidad de puntos que se harán para cada una de las pruebas.

- **Espacio de puntos**: Puntos en coma flotante pertenecientes a la región:

$$\{ 0 \le x, y, z \le 100 \}$$

- **Epsilon**: 2,5. Valor extraído del cálculo de los K-vecinos para un conjunto de 100000 puntos.
- **Mínimo de puntos**: 6. Al centrarnos en un volumen de datos grande, esta vendrá regida por la ecuación:

$$MinPts = n_{dims} * 2$$

- **Función distancia**: Euclidiana.

- **Cantidad de puntos**:
  - 1000
  - 10000
  - 50000
  - 100000
  - 150000
  - 200000
  - 250000

3.1.2 *Evaluación de resultados*

Una vez realizadas las ejecuciones y obtenidos los resultados del profiling, resulta bastante evidente cuales son las funciones que ocupan el mayor número de tiempo de ejecución del algoritmo. Tal y como se ve en la **Tabla 1** usada de ejemplo para presentar los resultados de todas las ejecuciones.

Nombre	**Porcentaje de tiempo usado**[11]	**Segundos totales de función**[12]	**Número de Llamadas**	**ms / llamada**[13]	**ms / llamada total**[14]
dbscan()	100.0	0.00	1	0.00	33.13
expand_cluster()[15]	100.0	0.00	10000	0.00	33.13
get_epsilon_neighbours()[16]	100.0	0.13	10000	13.05	33.13
euclidean_distance()[17]	60.6	0.2	99990000	0.00	0.00

Tabla 1. Profiling de DBSCAN con 10000 puntos procesados.

Todas las tablas, independientemente de la variación al número de puntos del conjunto de datos, presentan unos resultados similares en cuanto al porcentaje de uso de las funciones. Gracias a este porcentaje y a los segundos totales que el algoritmo invierte en cada función se puede determinar de qué manera se imbrican las funciones.

[11] Porcentaje de tiempo usado por la función en la totalidad del algoritmo.

[12] Los segundos totales en que el algoritmo ha estado en la función combinando todas las llamadas sin funciones herederas.

[13] Media de milisegundos dedicados a la llamada de una función individualmente sin funciones herederas.

[14] Media en milisegundos dedicados a la llamada de una función incluyendo a las funciones descendientes.

[15] Función que expande el cluster cuando se encuentra un punto suficientemente denso

[16] Función que crea la estructura de vecindad Epsilon en función de un punto dado

[17] Función que determina la distancia entre dos puntos en el espacio.

Se llaman una a otra de manera descendiente según la **Tabla 1**. Se aprecia que las funciones *dbscan()* y *expand_cluster()* ocupan un 100% del uso total del algoritmo, pero el tiempo en que se invierte en ellas es cero milisegundos descontando a las siguientes funciones imbricadas.

Es en las siguientes dos funciones donde se invierte todo el tiempo de cálculo. Si la función de cálculo de distancias entre puntos ocupa un 60.6% del algoritmo, viendo las tablas temporales se determina que el porcentaje de uso de la función que calcula la región de vecinos será por si sola 39.4%.

Por lo que se centrará el análisis en una propuesta que reparta la carga de las funciones *euclidean_distance()* y *get_epsilon_neighbours().*

Viendo los resultados, por el momento se abandonarán las propuestas de paralelización para las funciones de expansión y difusión, aunque tengan un relativo gran número de llamadas, ya que el porcentaje de tiempo que ocupan es cero dentro del rango de valores presentados por el programa. Los costes de sincronización o envíos de mensajes pueden suponer una mayor carga computacional que incurriría en un aumento del tiempo de ejecución.

La función euclidean_distance() solo se invoca desde la función get_epsilon_neighbours() en este algoritmo y es una única operación de coste computacional $O(1)$.

Pero si vemos la comparativa en la **Tabla 1** para la función get_epsilon_neighbours() entre los dos últimos valores, que representan la media de milisegundos de uso por función, para ella misma (13.05 ms) y, ella y sus descendientes (33.13 ms). Vemos que el uso total de la función heredera euclidean_distance() ocupa una mayor parte del tiempo de ejecución por el alto número de veces que se llama.

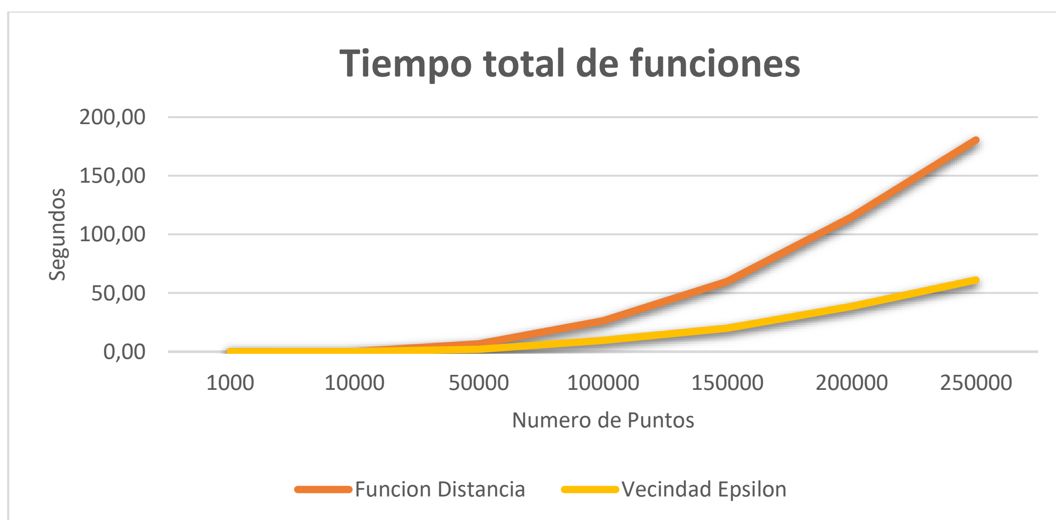


Figura 10. Comparativa de tiempo total de procesado de las funciones en función del número de puntos.

En la **Figura 10** se muestran los valores más significativos a la hora del análisis del uso de las funciones, el tiempo total que ha sido usada la función en la ejecución de todo el algoritmo a medida que aumenta el número de puntos en el conjunto de datos sin las funciones herederas.

Esta, junto a los resultados del número de llamadas de la **Tabla 1**, nos confirman el análisis del coste computacional del *apartado 2.2.4* que nos presenta una complejidad cuadrática de este algoritmo ya que se cumple:

$$Num_Llamadas(expand_cluster()) \times Num_Llamadas(get_epsilon_neighbours()) \approx Num_Llamadas(euclidean_distance())$$

Aunque debería ser una equivalencia exacta, no lo será, debido a que el cálculo del punto actual hacia él mismo se omite y debido a mecánicas estadísticas del propio Gprof que ya nos avisan que es una aproximación.

Así que la implementación final en los distintos estándares se centrara en el balanceo de carga de la función get_epsilon_neighbours() para aliviar el coste computacional.

3.2 Análisis de dependencias de datos

Una vez establecidas cuales son las funciones donde se tiene que centrar el esfuerzo para el alivio de la carga, hay que realizar un estudio de dependencias. Este estudio es el que realmente determina si se pueden aplicar técnicas de paralelización y como se harán.

Las dependencias de datos son aquellas que modifican o leen una variable. Analizarlas resulta de gran importancia ya que es el medio por el cual se determina el orden en que las tareas hacen uso de ellas.

Gracias a este análisis, se obtendrán grafos de dependencias mediante los cuales se podrá visualizar estas dependencias y facilitarán el determinar que técnicas de paralelización se podrán aplicar para asegurar que se puedan acceder concurrentemente a ellas sin incurrir en errores de datos entre procesos.

Para ello, primero se expondrán los algoritmos de las funciones con comentarios del formato */*Sx*/*. Estos comentarios representaran las tareas más relevantes dentro de las funciones en las cuales nos podemos encontrar la existencia de dependencias. Donde el valor *x* representa el identificador de la tarea y el orden en que se ejecutan estas en el código en serie.

Estos identificadores se usarán a posteriori para representar el grafo de dependencias, cada uno representará un nodo dentro de este. Las conexiones entre los nodos representarán el tipo de dependencia que tiene y en qué orden se producen según su direccionamiento.

Asimismo, se especificará el nivel de la dependencia. Su nivel marca, dentro de un bucle, a que distancia se encuentra entre sus iteraciones. Si es en la misma iteración o en la parte serial del código se denotará por el valor cero; si es en otra iteración se denotará con el valor uno; y si es en iteraciones de bucles imbricados[18], con valores mayores a uno.

Se usará la nomenclatura *d_x_n* en las conexiones donde:

- **d:** Representa que existe una dependencia.
- **x:** El tipo de dependencia.
- **n:** Distancia de la dependencia.

Existen tres tipos de dependencias que pueden afectar a la ejecución paralela del código, estas son:

- **Dependencia verdadera:** Estas se dan cuando una tarea intenta leer un dato de una posición de memoria después de haber sido modificado. En la nomenclatura se denota sin letra, x vacía.

[18] Uso de bucles dentro de otros bucles.

- **Anti-dependencia:** Se da cuando una tarea intenta escribir en una posición de memoria donde hay un dato que previamente ha sido leído. En la nomenclatura denotado por la letra *a*.
- **Dependencia de salida:** Se da cuando una tarea intenta escribir en un dato en el que se ha escrito previamente. En la nomenclatura denotado por la letra *o*.

Estas dependencias en un algoritmo en serie también suponen un problema a la hora de ejecutar las tareas. Pero estos son resueltos por mecanismos de optimización que tienen el compilador y mecanismos de control del propio procesador. Por lo que a priori este tipo de dependencias no nos preocuparán.

Un ejemplo de fallo en las técnicas de paralelización que no resuelven los conflictos de datos son los que se pueden ver en la **Figura 11**. Esta representa un bucle que suma valores sobre una variable hasta alcanzar el resultado.

Las flechas entre la posición de memoria y los procesos simbolizan el dato cargado de memoria y el escrito. Los números en ellas son el orden en que se ejecutan cuando este bucle se paraleliza en varios procesos. Y los datos al lado de la posición de memoria representan el valor que coge la variable en cada una de las escrituras.

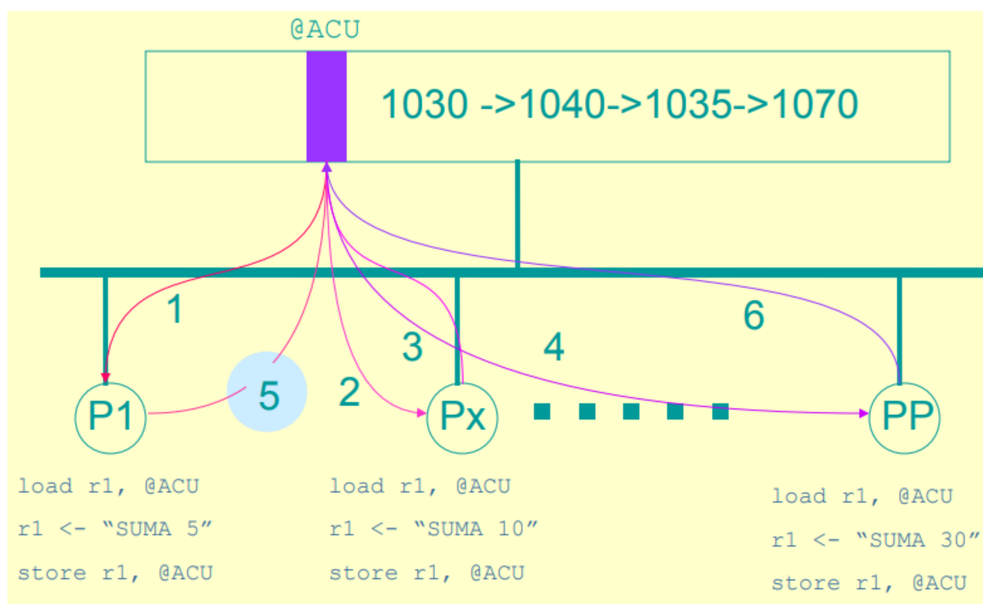


Figura 11. Errores derivados de no solucionar la dependencia de datos entre distintos procesos [31].

En este ejemplo existe una dependencia real cuando se lee el dato escrito por una iteración previa del bucle y una dependencia de salida cuando una iteración futura escribe lo que ya ha sido escrito. En esta ejecución paralela, al no disponer de mecanismos que controlen y sincronicen los accesos y escrituras en memoria, se produce un desorden en estas dando un resultado erróneo.

A la hora de realizar un algoritmo con funciones, la visualización de las dependencias puede resultar más difícil al no tener presente lo que realiza la función que se usa. Una de las dependencias más importantes cuando se debe paralelizar es aquella que existe entre los distintos niveles de los bucles imbricados y estas tienen presencia en este algoritmo.

Es por ello por lo que el análisis se realizará en distintas etapas desde las funciones más altas en la jerarquía hacia sus descendientes y poder llegar a tener una visualización más

clara de los distintos niveles de dependencias. Siguiendo la nomenclatura de los identificadores de las funciones, la declaración de la función que conectara las dependencias con sus ascendientes se identificara con el formato */*S0*/.*

3.2.1 *Dependencias de función DBSCAN*

La función DBSCAN no resulta muy compleja de analizar ya que su mecánica es expandir el cluster a través de la función de expansión clasificada como */*S2*/* en el **Código 1**. Una vez se completa la creación, esta solo modifica el identificador del cluster para crear uno nuevo. Este proceso se repite hasta que todos los puntos se han visitado y se decide si pertenecen a algún cluster o están clasificados como ruido.

```
void dbscan(point_t *points, unsigned int num_points, double epsilon,
unsigned int minpts, double (*dist)(point_t *a, point_t *b))
{
      unsigned int cluster_id = 0;

      for (int i = 0; i < num_points; ++i)
      {
/*S1*/      if (points[i].cluster_id == UNCLASSIFIED)
            {
/*S2*/            if (expand(i, cluster_id, points, num_points,
                              epsilon, minpts, dist) == CORE_POINT)
                  {
/*S3*/                  ++cluster_id;
                  }
            }
      }
}
```

Código 1. DBSCAN.

Aun así, como se comentó en el *apartado 1.2,* y se puede apreciar en el grafo de la **Figura 12**, la paralelización de este fragmento de código resulta extremadamente compleja y probablemente resulte ineficiente al tener que implementar los mecanismos de sincronización debido a las dependencias que existen entre distintos niveles de los bucles imbricados del algoritmo.

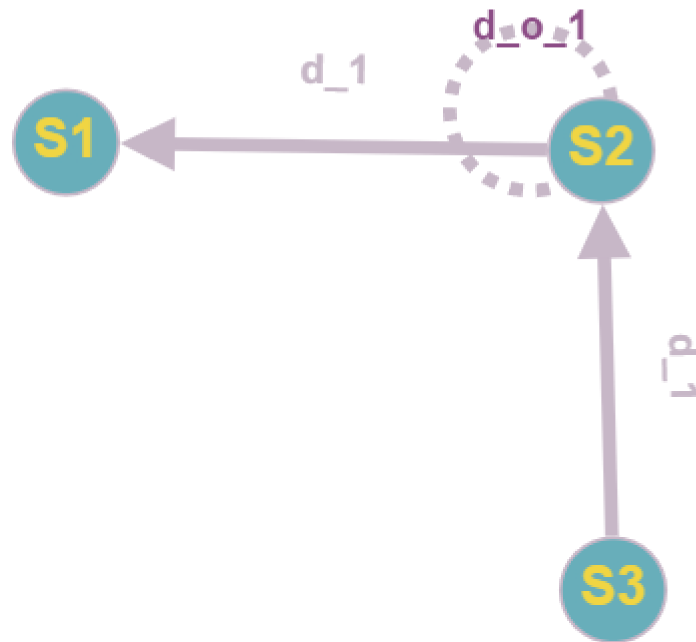


Figura 12. Grafo de dependencias de datos de función DBSCAN.

Debido a estas dependencias entre distintos niveles y en dirección invertida a la ejecución, es imperativo que estas tres tareas se ejecuten en el orden estipulado ya que los puntos pueden ser modificados en posiciones arbitrarias del vector que representa el conjunto de datos.

Esto nos obliga a tener que esperar a la finalización de los bucles imbricados de niveles interiores para poder proseguir con su ejecución y leer los datos de la siguiente iteración del bucle principal en el orden correcto.

3.2.2 *Dependencias de función de expansión*

Este es, como ya se mencionó, el fragmento más importante del algoritmo. Este busca los puntos núcleo dentro del conjunto de datos y, si cumplen los requisitos de densidad determinados por las variables *Epsilon* y *MinPts,* les asigna a un cluster. Acto seguido se encarga de expandir el cluster.

```
/*S0*/int expand(unsigned int index, unsigned int cluster_id, point_t
*points, unsigned int num_points, double epsilon, unsigned int minpts,
double (*dist)(point_t *a, point_t *b))
{
      int return_value = NOT_CORE_POINT;
/*S1*/epsilon_neighbours_t *seeds = get_epsilon_neighbours(index,
                                    points, num_points, epsilon, dist);
      if (seeds == NULL)
      {
            return FAILURE;
      }
/*S2*/if (seeds->num_members < minpts)
      {
/*S3*/  points[index].cluster_id = NOISE;
      }
      else
```

```
      {
/*S4*/      points[index].cluster_id = cluster_id;
/*S5*/      node_t *h = seeds->head;
            while (h)
            {
/*S6*/            points[h->index].cluster_id = cluster_id;
/*S7*/            h = h->next;
            }
/*S8*/      h = seeds->head;
            while (h)
            {
/*S9*/            spread(h->index, seeds, cluster_id, points,
                                  num_points, epsilon, minpts, dist);
/*S10*/           h = h->next;
            }
            return_value = CORE_POINT;
      }
      destroy_epsilon_neighbours(seeds);
      return return_value;
}
```

Código 2. Función de expansión de cluster.

Esta función tiene tres bucles principales vistos en el **Código 2**. Uno es para encontrar los vecinos del punto especificado por la función que lo llama, representado por /*S1*/. Otro es para añadir al cluster los vecinos del punto encontrado si este es suficientemente denso, representados por */*S6*/* y */*S7*/*. Y la última para difundir el cluster a lo largo de los mismos vecinos representado por */*S9*/* y */*S10*/*.

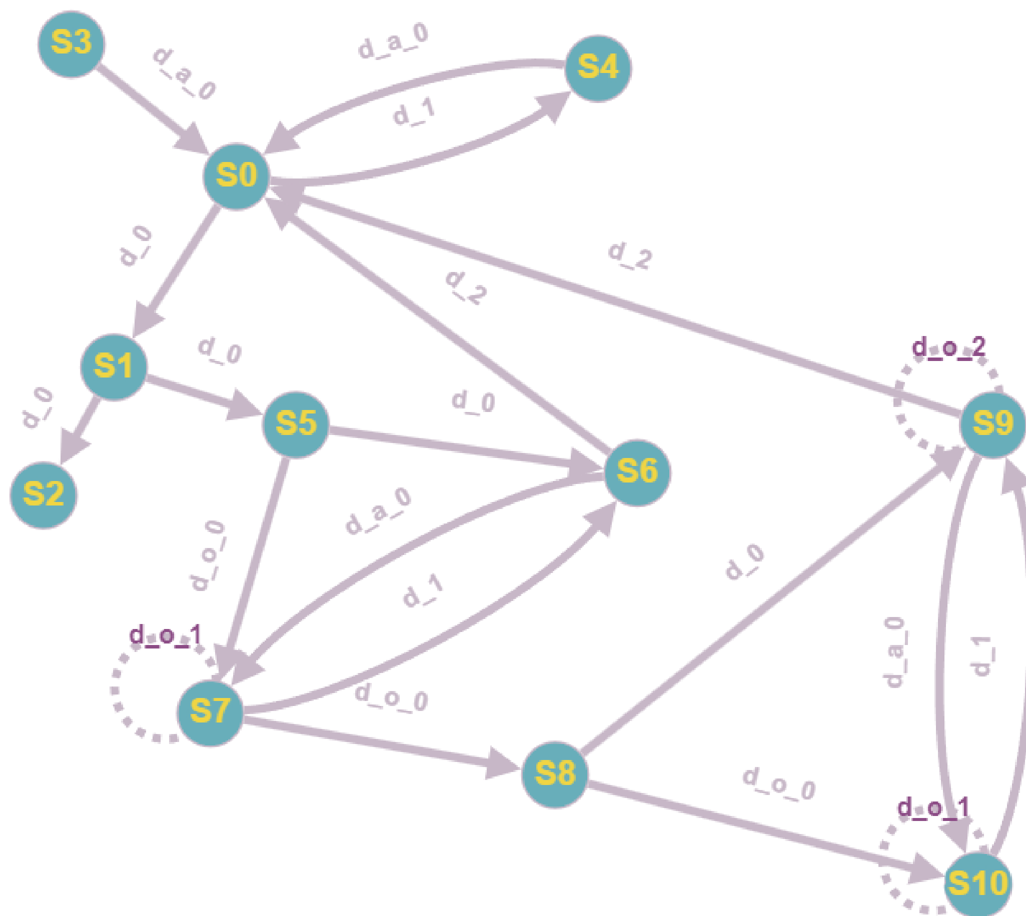


Figura 13. Grafo de dependencias de datos de función para expandir cluster.

Con este grafo ya podemos determinar que el bucle que está en la función DBSCAN visto en el apartado anterior en el **Figura 12**, no se puede paralelizar. Debido principalmente a la dependencia real de nivel dos que existen con /*S6*/ y /*S9*/. Estas tienen que modificar variables en la estructura de puntos dentro del conjunto global de datos antes de que el bucle principal pueda proseguir con la siguiente iteración.

En la propuesta de paralelización de esta función, no podemos aplicar una técnica que aprovechase el modelo de ejecución *fork-join* ya que la lista sobre la que se opera se actualiza constantemente. Este hecho se ve en la tarea */*S9*/* por la cual el tamaño de la lista varia hasta que se sale del bucle debido a la función de difusión. Pero si se podrá aplicar en la función de la tarea */*S1*/* de manera interna que se analizará más adelante.

No obstante, se puede aplicar la técnica de declaración de nuevas variables, por la que se duplicarían los vecinos encontrados representados por las tareas */*S5*/* y */*S8*/*, y con esa base podríamos implementar una paralelización que tuviese un modelo de ejecución en tareas.

Un grafo con estas reestructuraciones aplicadas en el algoritmo nos daría unas dependencias vistas en el **Figura 14**. Donde después de la ejecución de */*S1*/*, se pueden lanzar tareas que ejecuten el resto del código concurrentemente.

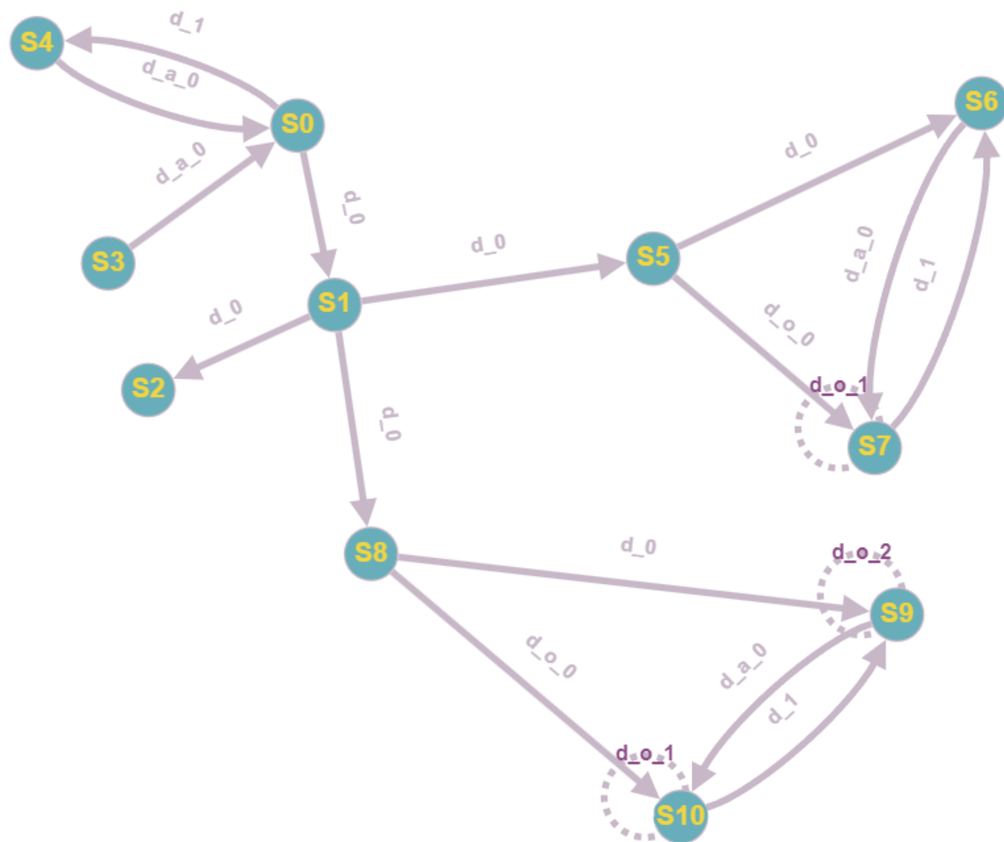


Figura 14. Grafo de **d**ependencias de datos de función de expansión con transformaciones aplicadas.

Las dependencias reales que existían entre /*S6*/ y /*S9*/ con /*S0*/ no afectaría a la ejecución en una implementación paralela ya que estas requerirán una sincronización por barrera al final de la ejecución de las tareas para poder terminar de escribir los datos antes de ser leídos. Además de que son dependencias de nivel dos y estas se producen en el nivel anterior del bucle.

Aunque esta propuesta será solo un posible diseño de optimización futuro ya que como se determinó en el *apartado 3.1.2*, la función donde se centraré el esfuerzo para la posible paralelización es la que consulta a la región de vecinos del punto indicado.

3.2.3 *Dependencias de función de difusión*

La función difusión es complementaria a la función de expansión. Mientras que la función expansión se centra en buscar los puntos de tipo núcleo, la de difusión se encarga de procesar a los puntos fronterizos y determinar si en ellos acaba la expansión o se clasificará como otro punto núcleo.

Esta se implementa para facilitar la comprensión y organización del código y opera de un modo similar a la función de expansión de cluster vista en el apartado anterior.

```
/*S0*/int spread(unsigned int index, epsilon_neighbours_t *seeds,
unsigned int cluster_id, point_t *points, unsigned int num_points,
double epsilon, unsigned int minpts, double (*dist)(point_t *a, point_t
*b))
{
/*S1*/epsilon_neighbours_t *spread = get_epsilon_neighbours(index,
                                    points, num_points, epsilon, dist);
      if (spread == NULL)
      {
            return FAILURE;
      }
      // <Procesar vecinos de vecinos>
      // Para cada punto añadido al cluster, se procesan sus epsilon
      //vecinos si es un CORE_POINT
/*S2*/if (spread->num_members >= minpts)
      {
/*S3*/      node_t *n = spread->head;
            point_t *d;
            while (n)
            {
/*S4*/            d = &points[n->index];
            //Si un vecino del nuevo core es ruido o UNCLASIFIED, se
            //añaden al cluster
/*S5*/            if (d->cluster_id == NOISE || d->cluster_id ==
                                                UNCLASSIFIED)
                  {
                  // Si es UNCLASIFIED, añadimos a la estructura del
                  // CORE_POINT actual
/*S6*/                  if (d->cluster_id == UNCLASSIFIED)
                        {
                        //Se añade el punto a la estructura actual
/*S7*/                        if (append_at_end(n->index, seeds) ==
                                                      FAILURE)
                              {
                                    destroy_epsilon_neighbours(spread);
                                    return FAILURE;
                              }
                        }
/*S8*/                  d->cluster_id = cluster_id;
                  }
/*S9*/            n = n->next;
            }
      }
    // <FIN Procesar vecinos de vecinos>
      destroy_epsilon_neighbours(spread);
      return SUCCESS;
}
```

Código 3. Función de difusión.

De este modo tenemos dos bucles principales que se pueden ver en el **Código 3**. Uno es la función que consulta la región de vecinos representado por la tarea */*S1*/*. El otro es el *bucle while* que determina si un punto es núcleo o frontera.

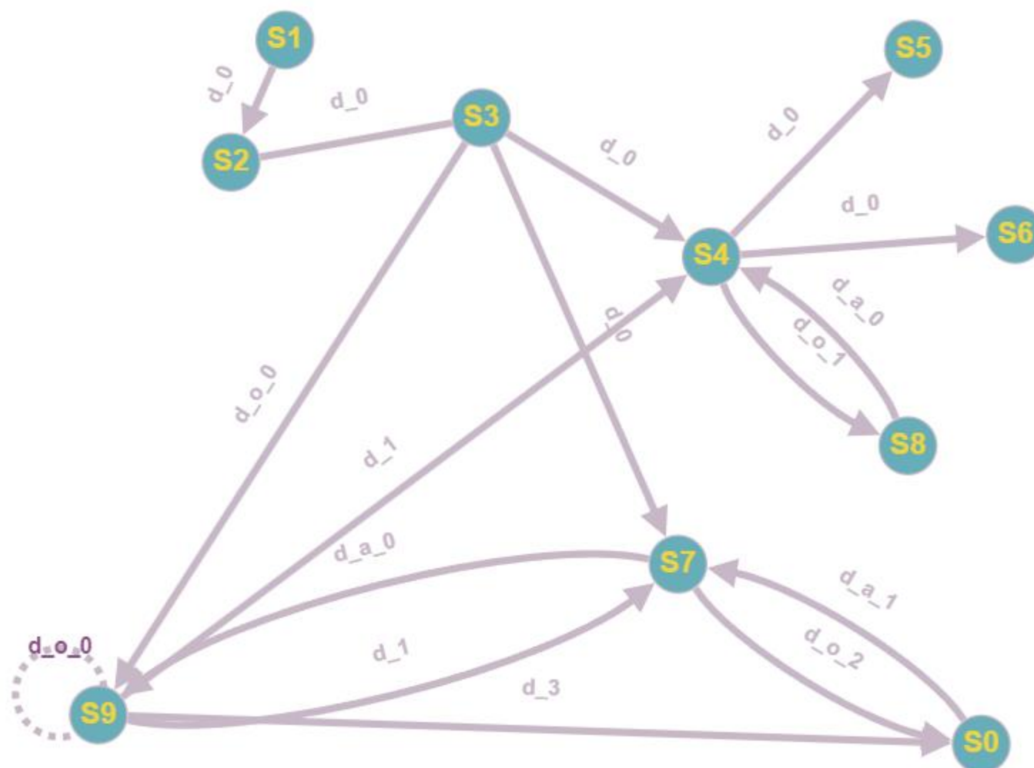


Figura 15. Grafo de **d**ependencias de datos de función de difusión.

En la **Figura 15** observamos varios problemas. El primero es la existencia de bucles en la conexión entre nodos con distintos tipos y distintos niveles de dependencias. Después tenemos una dependencia en el orden de ejecución entre */*S9*/* y */*S4*/.* Estos son potenciales problemas y pueden llegar a evitar la paralelización en estos puntos para esta función.

Pero de la misma manera que el apartado anterior, transformando algunas variables de escalares a vectoriales y reestructurando el código, podemos hacer que este sí que sea paralelizable mediante un modelo de ejecución *fork-join* en el *bucle while*.

Esta eliminaría la dependencia real que existe entre /*S4*/ con /*S9*/ y, */*S3*/* con */*S7*/* pudiendo así realizar una paralelización del bucle después de /*S3*/.

No obstante, esta es una posibilidad que se realizará a posteriori después del análisis de la función de consulta de región de vecinos de un punto por la misma razón que en el apartado anterior.

3.2.4 *Dependencias de función de consulta de región de vecinos*

Esta es la función que, dado un punto, realiza un cálculo de las distancias con todos los demás puntos. Crea una lista enlazada encapsulando todos los puntos que cumplan con el requisito de distancia Epsilon y lo devuelve a la función que lo ha llamado.

Esta se llama desde dos regiones del código y es la que se centrará el esfuerzo para la paralelización ya que es la que ocupa el mayor tiempo de computación.

```
/*S0*/epsilon_neighbours_t *get_epsilon_neighbours(unsigned int index,
point_t *points, unsigned int num_points, double epsilon, double
(*dist)(point_t *a, point_t *b))
{
/*S1*/epsilon_neighbours_t *en = (epsilon_neighbours_t *) calloc(1,
                                          sizeof(epsilon_neighbours_t));
      if (en == NULL)
      {
            perror("Failed to allocate epsilon neighbours.");
            return en;
    }
      for (int i = 0; i < num_points; ++i)
      {
            if (i == index)
                  continue;
            if (dist(&points[index], &points[i]) > epsilon)
                  continue;
            else
            {
/*S2*/            if (append_at_end(i, en) == FAILURE)
                  {
                        destroy_epsilon_neighbours(en);
                        en = NULL;
                        break;
                  }
            }
      }
    return en;
}
```

Código 4. Consultar región de vecinos de un punto.

Se ve en el **Código 4** que solo existe un *bucle for* en esta función. Este opera sobre datos de entrada constantes que están definidos desde antes de la primera llamada a la función DBSCAN y en base a ellos crea una nueva estructura de datos. Por lo que no existirán dependencias de datos que creen conflictos graves con las tareas de las funciones ascendentes.

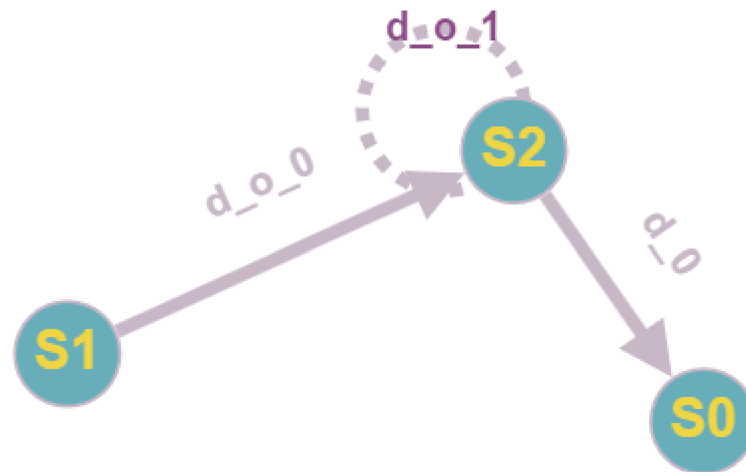


Figura 16. Grafo de dependencias de datos en consulta región de vecinos.

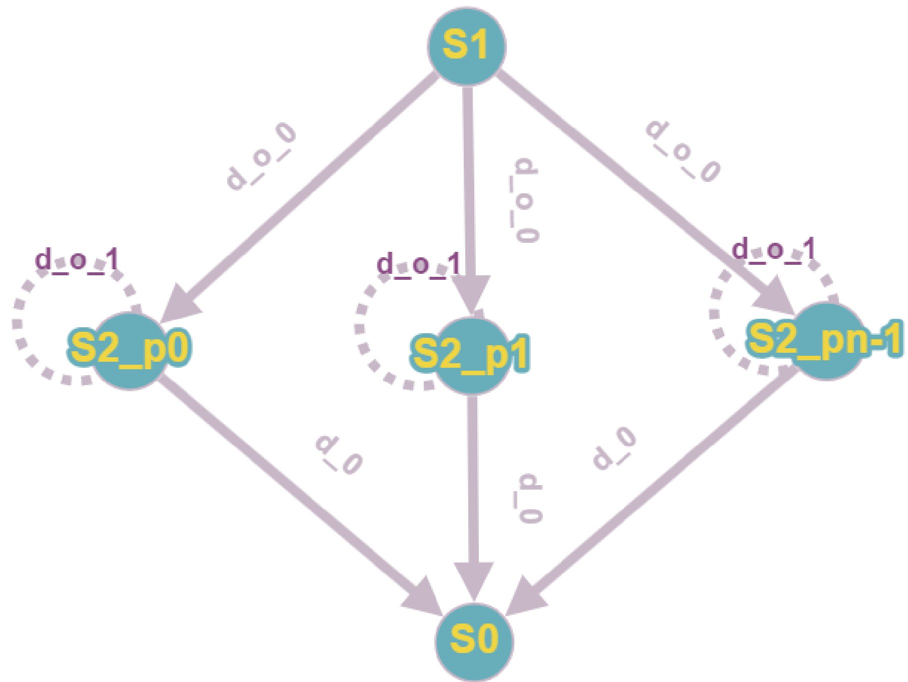
Debido a que los nodos en la lista enlazada no se tienen que añadir en orden, su paralelización puede resultar sencilla viendo las dependencias de datos en la **Figura 16**. Reestructurando el código para implementar una paralelización con el modelo fork-join en el *bucle for*.

Esta se puede realizar mediante una sincronización entre procesos en la escritura de la variable en este punto o, una expansión escalar[19]. Esto nos daría el mismo grafo sin la dependencia de salida que existe en */*S2*/* hacia ella misma.

La dependencia real que existe entre /*S2*/ y /*S0*/ se respetaría ya que el modelo de ejecución aplicado tiene implícitamente una sincronización por barrera al final de la ejecución de la parte paralela.

Con los cambios aplicados obtendríamos el **Grafo 17** donde se mantienen las dependencias, pero repartidas entre varios procesos.

[19] Eliminar variables escalares transformándolas a vectoriales para así eliminar dependencias en un bucle.



Grafo 17. Grafo de dependencias de datos en consulta región de vecinos en ejecución paralela.

4 Propuesta de paralelización

Con los resultados que se han obtenido mediante la técnica de profiling, podemos determinar cuál es la parte del algoritmo donde tenemos que centrar el esfuerzo para conseguir la optimización deseada.

Como todas las partes del código sin las funciones imbricadas representan un gasto temporal de casi cero, se pondrá el foco de atención en la función que determina la región de vecinos de un punto para saber la mejor manera de como balancear su carga entre los distintos procesos.

Por otra parte, gracias al análisis de las dependencias de datos, podemos crear las propuestas más adecuadas para llegar al objetivo que nos ha marcado el profiling evitando los errores en la ejecución de la paralelización implementada.

En función a estos hechos, se adaptará el algoritmo para que funcione de la manera más eficiente con los distintos estándares de programación de los que aquí tratamos y se propondrán las siguientes técnicas de diseño.

4.1 Técnica de descomposición

Los primeros pasos serán proponer de qué manera se dividirán los datos para que sean procesados por cada hilo, de qué manera se dividirán estas tareas para que el balanceo de carga sea equitativo y que técnicas de optimización se aplicarán.

4.1.1 *Descomposición de datos*

Debido a que no existen modificaciones en la estructura de datos entre las distintas iteraciones del *bucle for* en la función *get_epsilon_neighbours(),* no existen restricciones en el orden de ejecución de este, por lo que como técnica de descomposición se propondrá la de datos.

Esta técnica descompone los datos de entrada en distintas partes para ser computadas por los procesos como se ve en la **Figura 18**, por lo que permite una gran concurrencia al operar sobre grandes estructuras de datos como los que tratamos en este proyecto.

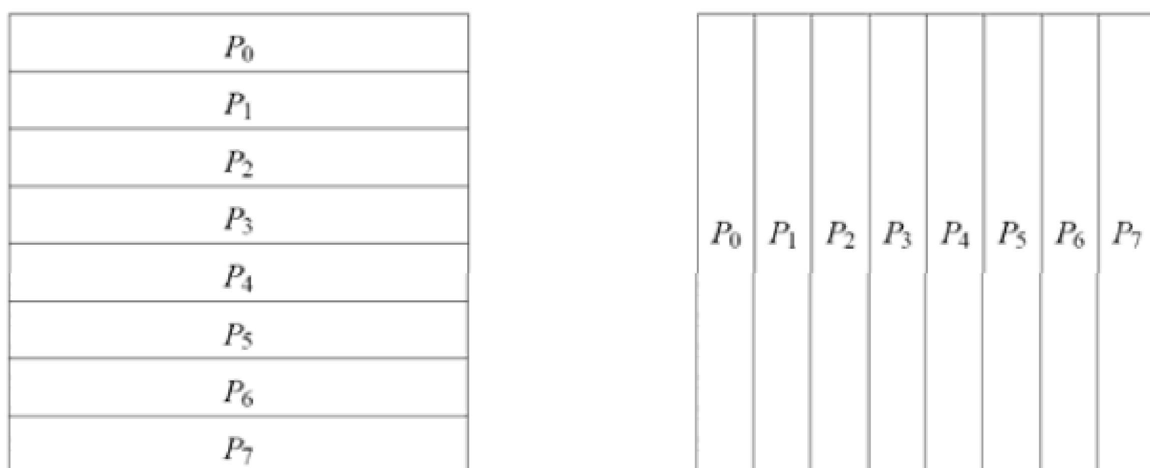


Figura 18. Representación de partición de datos de entrada unidimensional para varios procesos [34].

Como la base de la función es calcular la distancia de un punto seleccionado hacia todos los demás, es sencillo repartir los datos en porciones equitativas. Esta partición se realizará dividiendo la estructura de datos que almacena los puntos en el espacio.

4.1.2 *Asignación de procesos*

Esta división se realiza cuando se deben ejecutar las iteraciones del bucle para repartir al máximo la carga computacional y que los distintos hilos de ejecución tengan una cantidad de trabajo similar. Asimismo, con el objetivo de minimizar el tiempo de interacción entre los procesos.

Dado que las tareas se han generado usando la técnica de descomposición de datos sobre un conjunto fijo, se establece que la asignación de procesos se hará de manera estática por bloques distribuyendo cantidades de datos equitativas entre ellos, ya que el tamaño de estas es conocido.

Con esta técnica se determina de antemano cuales son las iteraciones del bucle que hará cada proceso y requerirá de una sincronización por barrera al final de la ejecución para asegurar que todos terminen a tiempo y pueda seguir la ejecución del algoritmo de manera segura.

En este caso se propone una distribución vertical, donde cada proceso ejecutara bloques contiguos de iteraciones representados en la **tabla 2**.

Iteración	1	2	3	4	5	6	7	8	9
Proceso	**P1**	**P1**	**P1**	**P2**	**P2**	**P2**	**P3**	**P3**	**P3**

Tabla 2. Representación de ejecución vertical en asignación estática de datos.

4.2 Reducción del efecto de la interacción

Este apartado hace referencia a la manera en que los distintos procesos acceden a los datos compartidos y como interactúan entre ellos ya que esta es la parte más costosa de las técnicas de paralelización. El objetivo, por una parte, es maximizar la localidad de los datos para que así los procesos acudan lo mínimo posible a los recursos compartidos. Por otro, reestructurar el algoritmo para que la frecuencia de interacción entre los procesos sea lo menor posible y evitar así usar canales de interconexión.

4.2.1 *Localidad de los datos*

Se intenta replicar el máximo número de datos de una manera temporal para cada proceso. Así la frecuencia de interacción entre ellos se reduce drásticamente ya que no se tiene que acceder a memoria principal para consultar su valor o modificarlo.

Al replicar los datos para distintos procesos, su volumen aumenta y esto puede provocar un gran problema en la memoria privada. Es por ello por lo que la interpretación será ligeramente distinta en máquinas de memoria principal compartida y máquinas con memoria principal distribuida.

Para las máquinas de memoria principal compartida podemos privatizar todos los datos de entrada. Estos son los parámetros que requiere la función y el bloque de puntos contiguos de la estructura de datos principal que tratará cada proceso en sus iteraciones. Como se ve en la **Tabla 3** ejemplificando la ejecución de la función con dos procesos disponibles para una cantidad de diez mil puntos.

	Proceso 1	**Proceso 2**
Índice	Privada	Privada
Puntos	Privada[0…4999]	Privada[5000…9999]
Num_Puntos	Privada	Privada
Epsilon	Privada	Privada

Tabla 3. Localidad de los datos por cada proceso para un conjunto de 10000 puntos.

Aunque en la implementación, para el conjunto de puntos no será esta interpretación del todo exacta por limitaciones de la propia interfaz. Esta se definirá como un conjunto compartido con acceso restringido a una franja de datos por cada proceso.

Para las máquinas de memoria principal distribuida, al disponer de una memoria mayor, si es posible, se replicarán todos los datos incluyendo todos los puntos en el conjunto de datos de entrada original.

4.2.2 *Frecuencia de interacción*

Gracias a la técnica de localidad de los datos aplicada en el apartado anterior, se consigue reducir al mínimo la interacción de los procesos durante la ejecución.

Debido a la estructura del algoritmo, una vez realizada la ejecución paralela de la fracción de código, nos queda un conjunto de datos sueltos que de por si nos ofrecen solo soluciones parciales al problema actual.

Para arreglar este hecho es necesario que al final de la ejecución el proceso maestro recolecte los datos que se han generado entre todos los procesos y los concatene para crear la solución final.

En el caso de las máquinas de memoria principal compartida, esto se consigue con la expansión escalar, reservando un espacio de memoria compartida en forma de vector donde los procesos almacenarán los resultados para posteriormente ser juntados.

Para las máquinas de memoria principal distribuida, se aplica la misma solución, pero al no poder usar un espacio de memoria compartido, este se hará realizando pasos de mensaje entre los distintos procesos y el maestro.

Con las unidades de procesado grafico se realizará de una manera combinada en que todos los resultados parciales se transmitirán con una duplicación de memoria entre el dispositivo y el procesador. Luego el procesador se encargará de concatenarlos.

Así logramos que la frecuencia de interacción sea igual al número de puntos totales del que se dispone en el conjunto de datos, ya que para cada uno es necesario computar las distancias.

4.3 Pseudocódigo

Una vez realizado el análisis del problema y expuesta la propuesta de paralelización se procede a mostrar el pseudocódigo del algoritmo en el **Pseudocódigo 3** donde se muestran los pasos para la implementación paralela de la función *get_epsilon_neighbours()* paralela.

```
get_epsilon_neighbours (Indice, Puntos, num_puntos, Epsilon,
                              Funcion_distancia): Vecinos_Epsilon;
      vecinos_global := allocate_memory();
      vecinos_mem_compartida [num_procesos];
      PARALLEL REGION
            vecinos_local := allocate_memory();
            PAR-FOR EACH i FROM (num_puntos/num_procesos) * num_proceso
                      TO (num_puntos/num_procesos) * num_proceso +1 DO
                  IF i = Indice THEN
                        Continue;
                  END IF
                  IF Funcion_distancia(Puntos[Indice], Puntos[i])
                                                       > Epsilon THEN
                        Continue;
                  ELSE
                        vecinos_local.Add(i);
                  END IF
            END PAR-FOR
            Vecinos_mem_compartida[num_proceso] := vecinos_local;
      END PARALLEL REGION
      FOR EACH vecinos_Epsilon IN vecinos_mem_compartida DO
            Vecinos_global.concatenate(vecinos_Epsilon);
      END FOR
      RETURN vecinos_global;
END;
```

Pseudocódigo 3. Función de vecindad Epsilon paralela.

En ella se asumen la localidad y la compartición de los datos que tiene cada proceso definidos en el apartado anterior, la sincronización que se requiere al final de la ejecución del *bucle for* paralelo y como posteriormente el proceso maestro junta todos los resultados parciales para obtener el resultado final después de la declaración *"END PARALLEL REGION"*.

La implementación final dependerá del estándar de programación que se use para los distintos tipos de máquinas.

5 Implementación

Cada uno de los tres estándares sobre los cuales se trabaja en este proyecto, trabajan sobre distintos tipos de máquinas y dispositivos con sus respectivas librerías propias. Por lo que la implementación de la propuesta de paralelización variará en función de la metodología a usar.

En este apartado se explican las decisiones tomadas en la implementación del algoritmo paralelo con cada estándar.

5.1 OpenMP

La interfaz y la sintaxis que usa OpenMP resultan en una implementación sencilla gracias al usar un recurso de memoria compartida.

Para implementarlas se usa una directiva que define su comportamiento seguido de unas clausulas como se ve en el **Código 6** donde se declara la región paralela [19].

Siguiendo la declaración de regiones compartidas y expansión escalar en la propuesta de paralelización, primeramente, como se ve en el **Código 5**, se declaran las nuevas variables *en_thread* y *en_res* donde se almacenarán los resultados parciales de cada proceso y se juntan en el resultado final respectivamente.

```
epsilon_neighbours_t *en;
int max_threads = omp_get_max_threads();
epsilon_neighbours_t en_thread[max_threads];
epsilon_neighbours_t *en_res = (epsilon_neighbours_t *) calloc(1,
                                         sizeof(epsilon_neighbours_t));
```

Código 5. Declaración de variables para recurso compartido.

Después de las declaraciones se usa la directiva *#pragma omp parallel* [20] en el **Código 6** la cual indica que la siguiente región será ejecutada concurrentemente por cada proceso en los distintos hilos disponibles en la máquina.

Para que se cumplan las restricciones de localidad de datos también hay que usar una serie de cláusulas que definirán si un recurso se replicará en cada proceso o será de uso compartido. Para ello se usan las siguientes clausulas:

- **Default(none):** Le dice al programa que se tiene que especificar la localidad de todas las variables independientemente.
- **Shared():** Variables que se usaran en un espacio de memoria compartido entre los procesos.
- **Private():** Crea una copia en el espacio de memoria privado de las variables sin inicializarlas.
- **FirstPrivate():** Crea una copia en el espacio de memoria privado inicializando las variables al último valor conocido por la ejecución en serie.

Una vez creadas las regiones paralelas, se reserva espacio en la memoria para las estructuras que almacenan las vecindades Epsilon encontradas por cada proceso que representara los resultados parciales de la ejecución.

```
#pragma omp parallel default(none) shared(dist, flag, en_thread,
en_res, points) private(en) firstprivate(index, epsilon, num_points)
{
      en = (epsilon_neighbours_t *) calloc(1,
                                           sizeof(epsilon_neighbours_t));
      ...
}
```

Código 6. Declaración de región paralela y la localidad de las variables.

Habiendo inicializado las variables y hechas las copias y repartos de localidades de estas, se procede a la ejecución paralela de las iteraciones del bucle vista en el **Código 7**. Seguirá el modelo de ejecución fork-join propuesto en el *apartado 3.2.4* que implícitamente tendrá una barrera al final.

Esta se realiza con la directiva *#pragma omp for* [21] que indica al compilador que tiene que repartir las iteraciones del bucle que viene seguido en porciones de datos del mismo tamaño asignado a los distintos procesos. Esta cláusula implícitamente cumplirá el requisito de descomponer los datos y realizar una asignación de procesos estática con distribución vertical propuestos en el *apartado 4.1.1*.

```
#pragma omp for
for (i = 0; i < num_points; i++)
{
      if(flag)
            continue;
      if (i == index)
            continue;
      if (dist(&points[index], &points[i]) > epsilon)
            continue;
      else
      {
            if (append_at_end(i, en) == FAILURE)
            {
                  flag = true;
            }
      }
}
```

Código 7. Declaración paralela del modelo de ejecución fork-join.

Una vez llegados al final de la cláusula para el bucle *for*, seguimos estando en la región paralela y es necesario que cada proceso acceda a la porción de memoria compartida y actualice los resultados de la ejecución visto en el **Código 8**.

Estas se cargan en el vector *en_thread* donde cada posición representa el identificador del proceso que actualiza su valor. Es necesario una sincronización por barrera al final de este fragmento con la directiva *#pragma omp barrier* [22]que bloquea la ejecución hasta que todos los procesos hayan llegado a ese punto ya que, acto seguido el proceso maestro juntara todos los resultados. Por lo que necesitamos que todos los procesos hayan acabado de cargar los resultados primero.

```
int thread_num = omp_get_thread_num();
en_thread[thread_num].head = en->head;
en_thread[thread_num].tail = en->tail;
en_thread[thread_num].num_members = en->num_members;
#pragma omp barrier
```

Código 8. Actualización de recursos de memoria compartida.

Una vez cargados todos los resultados a la memoria compartida del programa, se usa la directiva *#pragma omp master* [23] la cual indica que ese fragmento se realizara únicamente por el proceso maestro que se puede ver en el **Código 9**.

Este es el paso final en el cual el proceso maestro recoge los resultados de la variable compartida *en_thread* y los concatena en la variable declarada al principio de la función *en_res*. Esta es la que representa la totalidad de los vecinos Epsilon encontrados por cada proceso y la que se usará como retorno para la función.

```
#pragma omp master
{
	if(!flag)
	{
		bool first_encap = false;
		int num_threads = omp_get_num_threads();
		for (i=0; i<num_threads; ++i)
		{
			if(en_thread[i].head != NULL && en_thread[i].tail !=
					NULL && en_thread[i].num_members != 0)
			{
				if (!first_encap)
				{
					en_res -> head = en_thread[i].head;
					en_res -> tail = en_thread[i].tail;
					en_res -> num_members =
							en_thread[i].num_members;
					first_encap = true;
				}
				else
				{
					en_res->tail->next = en_thread[i].head;
					en_res->tail = en_thread[i].tail;
					en_res->num_members = en_res->num_members
							+ en_thread[i].num_members;
				}
			}
		}
	}
}
//fin de región paralela, sigue ejecución secuencial
```

Código 9. Concatenación de resultados en el proceso maestro.

Fuera de las regiones aquí indicadas, el algoritmo prosigue su ejecución serial como en el código original.

5.2 OpenMPI

Esta interfaz, al operar sobre máquinas distribuidas que disponen de memoria principal privada para cada una, partimos de la premisa de que la memoria total de la que disponemos será mucho mayor.

La complejidad espacial total del algoritmo será de $O(n * k)$, donde k son el número de nodos. Pero esta estará distribuida en la memoria privada de cada uno de ellos, por lo que no supondrá mayor problema.

Por ello, como primer paso, en vez de fragmentar los datos, el proceso raíz creará la estructura de datos principal y la replicará a cada uno de los nodos del sistema con una operación de broadcast[20] [24].

La programación en este estándar resulta distinta a lo anterior ya que distintas máquinas tienen que ejecutar el mismo código, por lo que hay que adaptar el algoritmo con claras diferencias de que parte del trabajo será realizado por cada tipo de nodo.

Esto se realiza mediante cláusulas condicionales en las cuales se consulta el identificador del nodo como se ve en el **Código 10**. Esta decidirá si tiene que ejecutar una parte u otra del algoritmo.

```
//Si el nodo tiene el rango raíz := 0
if(rank == 0)
{
	//Codigo para el nodo raíz
}else if(rank == 1)
	//Codigo para el nodo con identificador 1
}else{
	//Codigo ejecutado por el resto de los nodos
}
```

Código 10. Consulta de identificador del nodo.

Siguiendo el mismo análisis del *apartado 3.1* y su posterior propuesta de paralelización, las modificaciones para esta interfaz se basarán en que el proceso raíz realizará todas las operaciones de asignar los clusters a cada punto y el resto de los nodos cumplirán la función de coprocesadores que ayudarán al cálculo de la región de los vecinos.

Para asegurar que esta reestructuración se cumple desde el principio del algoritmo hasta el final de su ejecución, este cambio se introduce en la función DBSCAN, donde se separa el trabajo que realiza cada tipo de nodo como se ve en el **Código 11**.

```
void dbscan(point_t *points, unsigned int num_points, double epsilon,
unsigned int minpts, double (*dist)(point_t *a, point_t *b), int rank,
int procs)
{
	unsigned int i, cluster_id = 0;
	int terminate = 1;
	if(rank == 0)
	{
		for (i = 0; i < num_points; ++i)
		{
			if (points[i].cluster_id == UNCLASSIFIED)
			{
				if (expand(i, cluster_id, points, num_points,
			epsilon, minpts, dist, rank, procs) == CORE_POINT)
				{
					++cluster_id;
				}
			}
		}
```

[20] Operación por la que un nodo en la red, replica un dato a todos los demás nodos que pertenezcan a la misma red.

```
            //Señal de finalizacion para el resto de nodos
            MPI_get_epsilon_neighbours(-1, points, num_points, epsilon,
                                          dist, rank, procs);
      }
      else
      {
            epsilon_neighbours_t *en;
            while(en != -1)
            {
                  en = MPI_get_epsilon_neighbours(i, points,
                              num_points, epsilon, dist, rank, procs);
            }
      }
}
```

Código 11. Diferenciación de ejecución entre nodo raíz y otros nodos.

El nodo raíz proseguirá con la ejecución normal del algoritmo mientras el resto iterarán sobre la función de búsqueda de vecinos de un punto a la espera de la recepción del mensaje de finalización. Este mensaje indicará que ya se han procesado las distancias de todos los puntos y en este momento todos los nodos salen del bucle y terminan las ejecuciones del algoritmo.

Seguidamente se adaptará la función que realiza la consulta de la región de vecinos de un punto.

El proceso raíz realiza un envío del punto para el cual se tienen que encontrar los vecinos mediante una operación de broadcast.

La operación de paso de mensaje broadcast pertenece a la familia de mensajes bloqueantes, lo que quiere decir que, en esta implementación, los nodos no raíz que lleguen a este punto estarán a la espera de la recepción del mensaje. Por lo que no se invertirá tiempo de computación y resultará efectiva a nivel energético.

Esta incluye el envío del mensaje de fin del algoritmo visto en el **Código 12** para salir del bucle de ejecución y poder realizar a posteriori las correspondientes gestiones y liberaciones de espacios de memoria antes de finalizar el algoritmo en todos los nodos.

```
epsilon_neighbours_t *MPI_get_epsilon_neighbours(unsigned int index,
point_t *points, unsigned int num_points, double epsilon, double
(*dist)(point_t *a, point_t *b), int rank, int procs)
{
      //final de algoritmo
      if(index == -1)
      {
            MPI_Bcast(&index, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
            return 0;
      }
      //recoge el indice para los procesos no root
      MPI_Bcast(&index, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
      //si es -1, fin de algoritmo
      if(index == -1)
      {
            return -1;
      }
      …
}
```

Código 12. Sincronización de procesos mediante broadcast.

Una vez determinado el punto sobre el que se opera, cada nodo calculará sus propias iteraciones sobre el resto de los puntos del conjunto de datos basándose en el rango identificador del que disponen como se ve en el **Código 13**. Estos se almacenarán en una estructura de vecindad privada que cada nodo creará por sí mismo.

De aquí en adelante, cada nodo computará sus iteraciones sobre el conjunto de los datos y este fragmento se realizará en paralelo.

```
epsilon_neighbours_t *en = (epsilon_neighbours_t *) calloc(1,
                                        sizeof(epsilon_neighbours_t));

unsigned int pointsDiv, pointsMod;
pointsDiv = num_points / (unsigned int)procs;
pointsMod = num_points % (unsigned int)procs;

unsigned int iterIni, iterFin;

iterIni = (pointsDiv * rank) + pointsMod;
iterFin = (pointsDiv * (rank + 1)) + pointsMod;
if(rank == 0)
{
      iterIni=0;
}
for (int i = iterIni; i < iterFin; ++i)
{
      if (i == index)
            continue;
      if (dist(&points[index], &points[i]) > epsilon)
            continue;
      else
      {
            if (append_at_end(i, en) == FAILURE)
            {
                  destroy_epsilon_neighbours(en);
                  en = NULL;
                  break;
            }
      }
}
```

Código 13. Asignación de datos mediante el rango del proceso y computación paralela.

Una vez terminado el cálculo hay que recolectar los datos con una operación del tipo Gather[21] [25]. En este algoritmo, los datos que puede encontrar cada nodo son de tamaño variable, por lo que la función concreta a usar es la GatherV[22] [26].

Al no disponer de la cantidad de vecinos que ha encontrado cada nodo, se tendrá que realizar previamente un mensaje Gather común para que el nodo raíz reciba la cantidad de

[21] Paso de mensaje por el cual un nodo especificado como parámetro de la función, recolecta una cantidad de datos fija del resto de nodos almacenándolos en un buffer.

[22] Paso de mensaje por el cual un nodo especificado como parámetro de la función, recolecta una cantidad de datos variable del resto de nodos almacenándolos en un buffer. Esta cantidad variable debe estar especificada de antemano antes de la recolección.

elementos que han encontrado el resto de los nodos. Esto es obligatorio ya que para la función GatherV es uno de los parámetros de la función.

Habiendo recibido los tamaños, la estructura de vecinos de cada nodo se transformará a un vector de tamaño fijo en el que solo se almacenará el índice que les pertenece en la lista enlazada de la estructura donde se almacenan los puntos. Esto se hace porque la estructura de vecinos es una estructura de punteros y estos no serán los mismos en la memoria principal de cada uno de los nodos.

```
MPI_Gather(&num_found, 1, MPI_INT,
            num_neighbours, 1, MPI_INT,
            0, MPI_COMM_WORLD);
unsigned int index_array[num_found];
unsigned int *index_list;
int total_recvs = 0;
if(rank != 0)
{
      node_t *h = en->head;

      for(int i=0; i < num_found; i++)
      {
            index_array[i] = h->index;
            h = h->next;
      }
      destroy_epsilon_neighbours(en);
}else
{
      for(int i=0; i<procs; i++)
      {
            iter_neighbours[i] = total_recvs;
            total_recvs = total_recvs + num_neighbours[i];
      }

      index_list = (unsigned int *) calloc(total_recvs, sizeof(unsigned
int));
}
```

Código 14. Envío al proceso raíz del número de resultados encontrados por cada nodo.

Una vez que el proceso raíz ha recibido la cantidad de vecinos que tiene cada nodo y los vectores están cargados y listos para ser enviados, se aplica la operación de reunión de datos de tamaños variables GatherV como se ve en el **Código 15**.

Esta recolectará en un buffer del nodo raíz todos los datos en el orden que marca el rango identificador del resto de los nodos. Aunque por la propia mecánica del algoritmo, no sería importante en que orden se recibiesen.

Una vez terminada la recepción, el nodo raíz concatenara todos los resultados en la estructura de región de vecinos para el punto especificado en su memoria privada.

```
MPI_Gatherv(index_array, num_found, MPI_UNSIGNED,
            index_list, num_neighbours, iter_neighbours, MPI_UNSIGNED,
            0, MPI_COMM_WORLD);

if(rank == 0)
{
      for(int i = iter_neighbours[1]; i < total_recvs; i++)
      {
            append_at_end(index_list[i], en);
      }
      free(index_list);
}
return en;
```

Código 15. Recolección de datos final y concatenación de resultados.

Introducidos estos cambios, solo queda asegurarse que las llamadas a la función que consulta la región de vecinos de cada punto se realicen en los mismos sitios que en el algoritmo original, es decir, al principio de las funciones de expansión y difusión del cluster. Y que esta sea la función adaptada a la ejecución en interfaz OpenMPI.

5.3 CUDA

Para adaptar el algoritmo a su implementación mediante el uso de la unidad de procesado grafico es necesario reestructurar el código de una manera que haya claras diferencias entre las funciones ejecutadas por el procesador y el dispositivo grafico ya que estas son unidades de procesamiento independientes que se comunican a través del bus de datos. En este contexto, al procesador se le asigna el nombre de anfitrión(host) y a la unidad de procesado grafico el de dispositivo(device).

Con el fin de lograr esta adaptación, primero tenemos que calcular el número de bloques y warps, denotados por dimBlock y dimGrid respectivamente en el **Código 16,** que se generarán en las llamadas a una función que realice el dispositivo. Estas cantidades se obtienen en función al número de puntos que se trata en cada ejecución. El número de bloques vendrá determinado por la constante *TILE* que se recomienda que sea divisible por 32, ya que los conjuntos de bloques tienen 32 hilos. El valor 1024 es generalmente el más usado ya que su eficacia esta comprobada extendidamente por los investigadores. El tamaño del grid será la división entre el número de puntos y esta constante.

Acto seguido, el anfitrión deberá indicar las variables que serán usadas por el dispositivo, cuanta memoria es necesaria para cada una de ellas y también es el encargado de enviar la señal para que esa cantidad de memoria sea reservada. Si hubiese datos que tienen que ser enviados desde el anfitrión al dispositivo o viceversa, se realizará la operación acorde también. En este caso se enviarán todos los puntos del conjunto de datos al dispositivo.

Esta memoria se reserva y el envío de datos se realiza antes de la llamada a la función DBSCAN porque así solo realizaremos este proceso una vez evitando comunicación innecesaria que repercutirá en el tiempo de ejecución. Cuando se ha completado, las nuevas variables se añadirán como parámetro de la función y sus descendientes para que puedan ser usados en las llamadas.

```
// Tamaño de los bloques
Int TILE = 1024;

// Declaracion numero de grids y bloques
dim3 dimBlock (TILE, 1, 1);
dim3 dimGrid (((num_points-1) / TILE) + 1, 1, 1);

// Cemory alloc para CUDA
cudaMalloc((void**)&CUDA_points, sizeof(point_t) * num_points);
// Copiar datos a la grafica
cudaMemcpy(CUDA_points, points, sizeof(point_t) * num_points,
                                              cudaMemcpyHostToDevice);
// Vector para guardar los vecinos de una iteracion
unsigned int *CUDA_neigh_array;
cudaMalloc((void**)&CUDA_neigh_array, sizeof(unsigned int) *
                                                        num_points);

dbscan(points, num_points, epsilon, minpts, euclidean_dist, dimBlock,
                              dimGrid, CUDA_points, CUDA_neigh_array);
```

Código 16. Cálculo de distribución de bloques, reserva de memoria y transferencia de datos a la GPU.

El siguiente paso será adaptar la función que calcula la región de vecinos de un punto dado. Como se determinó al inicio del algoritmo, el número de grids por el número de bloques nos genera una cantidad de hilos igual al número de puntos. Por lo que el cálculo que tiene que procesar cada hilo equivaldría al mismo que se realiza en una iteración dentro del bucle original.

Para ello hay que agregar el atributo *__global__* a esta función como se ve en el **Código 17**. Esto indica que es una función llamada desde el anfitrión y será ejecutada por cada uno de los hilos declarados. Para identificar cada una de las distintas iteraciones del algoritmo original, se usa la variable *tid*. Esta es el identificador del hilo actual y se obtiene del cálculo de la posición del hilo que tiene en cada bloque de ejecución.

En esta implementación se declara una nueva variable para la comunicación entre el anfitrión y el dispositivo llamada *array_neigh*. Esta variable es un vector con el mismo tamaño que el número de puntos que representará el índice en que se encuentran los puntos vecinos dentro de una región. Los índices donde haya un punto vecino se marcarán con el valor uno y, en los que no, con el valor cero.

```
__global__ void CUDA_get_epsilon_neighbours(unsigned int index, point_t
*points, unsigned int num_points, double epsilon, unsigned int
*array_neigh)
{
      int tid = blockIdx.x*blockDim.x + threadIdx.x;

      if(tid < num_points)
      {
            double dist = CUDA_euclidean_dist(&points[index],
                                                  &points[tid]);

            if(dist <= epsilon && tid != index)
            {
                  array_neigh[tid] = 1;
            }else
            {
                  array_neigh[tid] = 0;
            }
```

```
        }
}
```

Código 17. Adaptación de consulta de región de vecinos para GPU.

Se decide adoptar esta implementación en vez de la creación de una estructura nueva y su transmisión como las usadas anteriormente ya que el uso y reserva de memoria dinámica desde dentro de un hilo del dispositivo es una operación muy costosa y a esta también habría que añadirle el tiempo de transmisión. Además, harían falta mecanismos de exclusión mutua que asegurasen que se escribiría en memoria principal del dispositivo solo por un hilo a la vez para no crear conflictos en los punteros de la estructura. Esto añadiría un gran coste por sobrecarga al tiempo de ejecución. De esta manera, se dispone de una variable sobre la cual los hilos actualizan el valor a cada llamada a la función.

Para el correcto funcionamiento de esta función, también hay que modificar la llamada a la función que calcula la distancia entre puntos. A esta hay que añadirle el atributo *__device__* como se ve en el **Código 18**. Esta indica que solo será llamada desde el entorno del dispositivo y será ejecutado solamente por el hilo que la llame.

```
__device__ float CUDA_euclidean_dist(point_t *a, point_t *b)
{
    return sqrt(pow(a->x - b->x, 2) + pow(a->y - b->y, 2) +
                                           pow(a->z - b->z, 2));
}
```

Código 18. Implementación local de función distancia en GPU.

Una vez adaptadas estas funciones, es necesario modificar las dos llamadas desde donde se realizan, en la funciones expansión y difusión del cluster. Primero se debe realizar la llamada a la función que será ejecutada por el dispositivo con el número de grids y bloques que se calculó previamente. Seguido de una función de sincronización que asegura que el anfitrión ejecutara el resto del código cuando el dispositivo haya terminado.

Esta sincronización es necesaria ya que hay que recolectar los datos de la nueva estructura declarada que representan los vecinos de la región encontrados por el dispositivo después de que este termine de actualizarla. Esto se realiza mediante una operación que copia los datos de una memoria a otra.

Una vez obtenidos los datos, se procede a la creación de la estructura de vecinos de un punto que serán procesados por el anfitrión en una ejecución en serie. Estos se obtienen iterando sobre los datos obtenidos por el dispositivo y actualizando la estructura en los índices donde haya el valor uno.

```
epsilon_neighbours_t *seeds = (epsilon_neighbours_t *) calloc(1,
                                      sizeof(epsilon_neighbours_t));
if (seeds == NULL)
{
        return FAILURE;
}

CUDA_get_epsilon_neighbours<<<grid, block>>>(index, CUDA_points,
                          num_points, epsilon, CUDA_neigh_array);
cudaDeviceSynchronize();

cudaMemcpy(neigh_array, CUDA_neigh_array, sizeof(unsigned int) *
                          num_points, cudaMemcpyDeviceToHost);

for (int i=0; i < num_points; i++)
```

```
{
	if(neigh_array[i] == 1)
	{
		append_at_end(i, seeds);
	}
}
```

Código 19. Lanzamiento de función en GPU, recepción de región de vecinos y adaptación al algoritmo en serie.

Después de estas modificaciones, el algoritmo prosigue de la misma manera como lo hace en su ejecución en serie en la implementación original.

Una vez procesados todos los puntos y asignados a un cluster, solo resta liberar los espacios de memoria que se reservaron en el dispositivo como se ve en el **código 20**. Esto es necesario hacerlo solo una vez ya que en esta implementación solo se reserva espacio para ellas una vez al principio del algoritmo y se opera sobre las mismas constantemente como se expuso al principio de este apartado.

```
cudaFree(CUDA_neigh_array);
cudaFree(CUDA_points);
```

Código 20. Liberación de memoria reservada en GPU.

6 Evaluación

En este apartado, primero se hará una breve descripción de las máquinas en las cuales se realizarán las ejecuciones de los algoritmos paralelizados para poder tener unas nociones de sus atributos.

La descripción no será muy profunda debido a que se comparará la ejecución paralela con la secuencial de la misma máquina y esto ya es suficiente para los datos que se necesitan para este análisis.

Luego se ejecutarán los algoritmos siguiendo las implementaciones del *apartado 5*. Estas se harán ,en el caso de los multiprocesadores y multicomputadoras, variando el número de hilos de ejecución y nodos a la vez que se varían la cantidad de puntos en el conjunto de datos que se procesan. Mientras que los parámetros Epsilon y el mínimo de puntos se mantendrán constantes como se estableció en el *apartado 3.1.1*, es decir, 2.5 y 6 respectivamente.

La cantidad de puntos variara en los siguientes valores: 50000, 100000, 150000, 200000, 250000.

La cantidad de procesos se variarán según las posibilidades que nos ofrezca la máquina en cuestión, pero estas partirán de uno y se aumentaran en potencias de dos, es decir: 1, 2, 4, 8, 16, 32, 64, 128…

Para la unidad de procesado gráfico, solo se variarán la cantidad de puntos ya que este solo consta de unidades de proceso vectoriales que realizan operaciones simples. Como se explicó en el *apartado 2.8*, la propia unidad ordena y asigna bloques de cálculo para las tareas que recibe en función de los parámetros delimitados por el usuario.

Una vez ejecutadas las pruebas, en base al tiempo de ejecución se realizarán unas métricas relativas en relación con el número de procesadores y la cantidad de datos. Estas estiman la efectividad con la que el algoritmo paralelizado aprovecha los recursos y son las que se usarán para determinar la escalabilidad de la implementación.

Estas métricas son:

- **Speed-Up**: Las veces que es más rápida la ejecución paralela en comparación a la secuencial, esta se obtiene dividiendo el tiempo de ejecución obtenido por el algoritmo secuencial con el tiempo de ejecución obtenido por el algoritmo paralelizado.

$$S(n) = {T_{ejecucion_secuencial}}/{T_{ejecucion_paralelo}}$$

- **Eficiencia**: Muestra la eficiencia con la que el algoritmo aprovecha los recursos en base al número de procesos que se usan. Está directamente relacionada con el Speed-Up y se obtiene dividiéndolo por el número de procesos. Esta es una de las métricas que determina la escalabilidad ya que, será escalable si sus valores se mantienen constantes en diagonal con el incremento del conjunto de datos y los procesos como se ve en la **Tabla 4**. Esta también se usa para determinar la razón por la cual se puede implementar una mejora en la paralelización. Pero para ello es necesario conocer la fracción serial del código, y para este caso no la podemos determinar ya que los valores del profiling marcan siempre cero.

$$E(n) = {S(n)}/{num_procesos}$$

Datos/n_procesos	1	4	8	16	32
64	1	0.80	0.57	0.33	0.17
192	1	0.92	0.80	0.60	0.38
320	1	0.95	0.87	0.71	0.50
512	1	0.97	0.91	0.80	0.62

Tabla 4. Muestra del valor constante en la eficiencia para un algoritmo escalable [32].

- **Isoeficiencia**: Esta determina como debe crecer el tamaño de los datos para mantener la eficiencia constante al aumentar el número de procesos y es una de las métricas más importantes al implementar un sistema paralelo. Si conseguimos un incremento de la eficiencia estable, este será uno de los parámetros constantes necesarios para su cálculo. La Isoeficiencia se obtiene multiplicando el tiempo de sobrecarga por la constante de eficiencia. Se establece que un sistema es altamente escalable si los incrementos de la Isoeficiencia son pequeños a medida que aumenta el número de procesos.

$$T_{sobrecarga}(n_{datos}, \; n_{procesos}) = n_{procesos} * T_{ejecucion_paralelo} - T_{ejecucion_secuencial}$$

$$k = {}^{E}/_{1 - E}$$

$$Isoeficiencia = k * T_{sobrecarga}(n_{datos}, \; n_{procesos})$$

6.1 Multiprocesador

6.1.1 *Máquina*

Para la evaluación de este código se dispone un multiprocesador de memoria principal compartida *AMD Ryzen Threadripper PRO 3995WX* [27] con arquitectura *ZEN2* ofrecida por el DEIM de la URV con el seudónimo ORCA. Esta dispone de 64 núcleos físicos y una tecnología multithreading de dos hilos por núcleo. Lo que nos da un total de 128 hilos de ejecución. Admite accesos a memoria tanto UMA como NUMA configurable por el puerto de conexión.

La memoria cache de este procesador se estructura como sigue:

Nivel	Tamaño
Nivel 1 privada de instrucciones	32KB por núcleo
Nivel 1 privada de datos	32KB por núcleo
Nivel 2 privada de datos e instrucciones	512KB por núcleo
Nivel 3 privada de datos e instrucciones	256MB para todos los núcleos

Tabla 5. Niveles de memoria cache para AMD Ryzen Threadripper.

El coste espacial del algoritmo como ya se expuso es de $O(n + k)$. Cada uno de los espacios de memoria en los que se almacenan cada uno de los puntos del conjunto de datos en el espacio vectorial de tres dimensiones ocupa 16 Bytes.

Con una cache privada de este tamaño, el algoritmo podría ser ejecutado, asumiendo el peor de los casos donde *k* toma el mismo valor de *n,* hasta para un conjunto de datos con una cantidad de ocho millones de puntos sin tener que intercambiar datos con la memoria principal.

6.1.2 *Resultados*

Una vez completados todos los estudios previos, se procede a ejecutar el código implementado en el estándar OpenMP sobre la máquina multiprocesador.

Primero se presentan los resultados en la gráfica que se aprecia en la **Figura 19**.

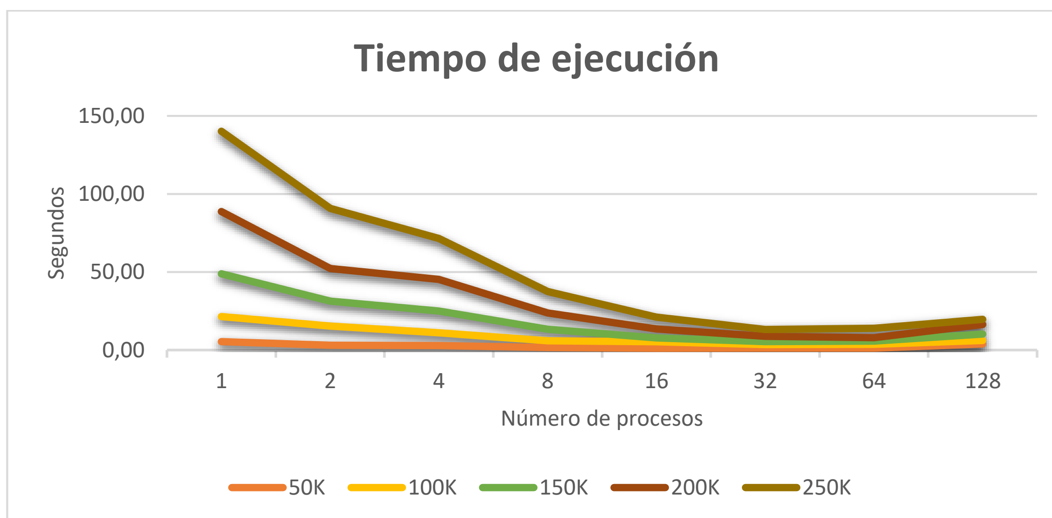


Figura 19. Tiempos de ejecución de conjuntos de datos de distintos tamaños según el número de procesos.

A primera vista, parece que la ejecución paralela cumple el objetivo propuesto de rebajar el tiempo de ejecución del algoritmo a medida que se aumentan el número de procesos con los que se trabaja.

Los conjuntos de datos más pequeños tienen una mejora más controlada debido al coste de lanzar los procesos y sincronizarlos. De la misma manera que la mejora parece constante hasta alcanzar los 64 procesos.

A los 128 se aprecia un notable empeoramiento ya que se reparten conjuntos de datos más pequeños entre muchos procesos. Esto hace que aumente el coste de sincronización y comunicación además de aumentar la parte serial en el momento de juntar los resultados parciales.

Seguidamente se mostrará en la **Figura 20** los Speed-Up obtenidos en comparación a los tiempos de ejecución.

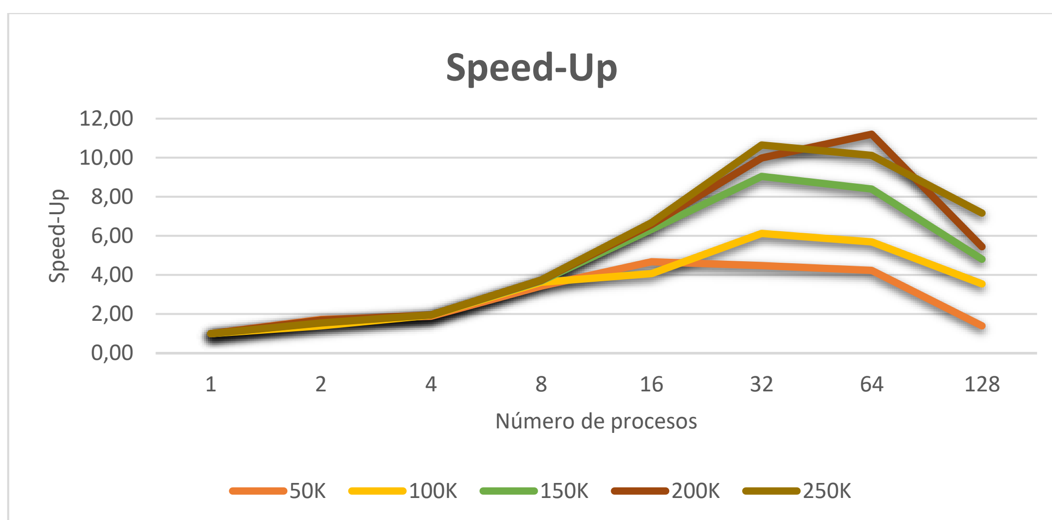


Figura 20. Speed-Up obtenido sobre conjuntos de datos de distintos tamaños según el número de procesos.

Gracias a estos datos podemos tener una visión más clara de la mejora obtenida con esta implementación paralela. Todas consiguen incrementar el Speed-Up de una manera constante hasta los 32 procesos. Aumentando los procesos a partir de este punto hace que la eficiencia del algoritmo decrezca. Por lo que a priori parece que este sistema es escalable hasta 32-64 procesos dependiendo del número de datos de entrada.

Vistos los resultados del Speed-Up, se procede a calcular la eficiencia del algoritmo en función al número de datos y procesos para poder determinar su escalabilidad.

Numero procesos / Numero Datos	50K	100K	150K	200K	250K
1	1	1	1	1	1
2	0,8622	0,6979	0,7819	0,8510	0,7719
4	0,4719	0,4842	0,4880	0,4895	0,4906
8	0,4283	0,4527	0,4611	0,4656	0,4682
16	0,2924	0,2543	0,3959	0,4111	0,4171
32	0,1401	0,1914	0,2826	0,3119	0,3328
64	0,0662	0,0891	0,1311	0,1751	0,1580
128	0,0109	0,0277	0,0375	0,0426	0,0560

Tabla 6. Eficiencia obtenida sobre conjuntos de datos de distintos tamaños según el número procesos.

Obtenidos los resultados que se ven en la **Tabla 6**, se debe determinar si podemos considerar algún valor constante en diagonal que determinara si este algoritmo es escalable.

Para esto se marcarán los recuadros que podemos considerar que están dentro de un rango de valores constantes en la **Tabla 6**. Con este conjunto de datos se aprecia que la escalabilidad es hasta 16 procesos, o 32 si lo podemos considerar dentro del rango de la constante.

También podemos observar que, para todos los resultados, la eficiencia aumenta a medida que aumenta el número de datos en el conjunto. Esto nos indica que posiblemente el algoritmo será mucho más escalable si se aumentan significativamente la cantidad de datos.

Después del análisis de la eficiencia y viendo que podemos considerar que disponemos de una constante, la cual fijaremos en el valor 0.46, se procederá al cálculo de la isoeficiencia que se presentará en la **Tabla 7**.

Procesos/Datos	**50K**	**100K**	**150K**	**200K**	**250K**
1	0	0	0	0	0
2	4.68	18.46	41.33	76.74	121.44
4	5.15	19.54	43.84	79.18	124.51
8	6.14	22.17	48.84	87.13	136.22
16	11.14	53.80	63.76	108.76	167.56
32	28.25	77.48	106.06	167.49	240.4
64	64.94	187.55	276.91	357.48	639.06
128	415.96	644.23	1072.01	1707.09	2023.42

Tabla 7. Isoeficiencia obtenida sobre conjuntos de datos de distintos tamaños según el número de procesos.

Esta tabla nos muestra el resultado definitivo para cada conjunto de datos, entre cuantos procesos debemos separar su ejecución. Mientras el crecimiento de la isoeficiencia sea estable y, lo más pequeña posible, el sistema será escalable.

Vemos que para cada conjunto de datos el crecimiento lo podemos considerar estable hasta los 32 procesos. Confirmándonos la suposición hecha referente a los valores de Speed-Up. Con este conjunto de datos, el algoritmo será escalable hasta 32 procesos. A demás de abrir la posibilidad de que lo sea aún más a medida que crece el volumen de datos.

6.2 Multicomputadora

6.2.1 *Máquina*

En un principio, para las pruebas en la multicomputadora se debían realizar en la máquina ODROIDS (con 64 nodos y 256 núcleos) y la máquina POP (con 8 nodos y 32 núcleos) proporcionada por el DEIM de la URV.

Debido a problemas con conexión al servidor que proporciona estos servicios, las pruebas en esta computadora se deberán anular por la imposibilidad de solucionar el problema a los días cercanos a la fecha límite para este proyecto.

El estándar OpenMPI no solo es de uso en multicomputadoras, este también permite su ejecución en multiprocesadores haciendo que los datos en vez de compartirse a través de memoria principal mediante el bus de datos, se haga con la misma mecánica, pero a través del paso de mensajes que establecen sus protocolos.

Es por ello por lo que las pruebas se realizaran en una máquina de uso doméstico. Que tendrá un rango de nodos más limitados, pero servirán para poder realizar pruebas y análisis de la optimización del algoritmo.

Con esta definición y en condiciones normales, las pruebas también se podrían realizar en el multiprocesador del apartado anterior. Pero al ser máquinas en un servidor con un uso especifico, no se pueden instalar los paquetes necesarios para la ejecución de programas en estándar OpenMPI.

El multiprocesador usado será el *AMD Ryzen 7 6800HS* [28] con arquitectura ZEN3+ que dispone de ocho núcleos. Estos serán el mismo número de nodos de los que dispondrá la emulación de nuestro sistema multicomputador.

La memoria cache de este procesador se estructura como sigue:

Nivel 1 privada de instrucciones	32KB por núcleo
Nivel 1 privada de datos	32KB por núcleo
Nivel 2 privada de datos e instrucciones	512KB por núcleo
Nivel 3 privada de datos e instrucciones	16MB para todos los núcleos

Tabla 8. Niveles de memoria cache para AMD Ryzen 7 6800HS.

Aunque estas memorias solo serán relevantes en la porción de ejecución serial que tendrá que calcular cada nodo ya que los mecanismos de paso de mensajes se realizan mediante memoria principal en este estándar.

6.2.2 *Resultados*

Para la prueba en multicomputadora, debida a las limitaciones físicas de la propia máquina, la variación del número de nodos no se hará en potencias de dos, se harán desde uno hasta ocho con un incremento a cada rango ya que este procesador dispone de ocho núcleos.

De este modo, aunque no tan concretos como en el apartado anterior, obtendremos unos resultados más amplios sobre los cuales hacer el análisis.

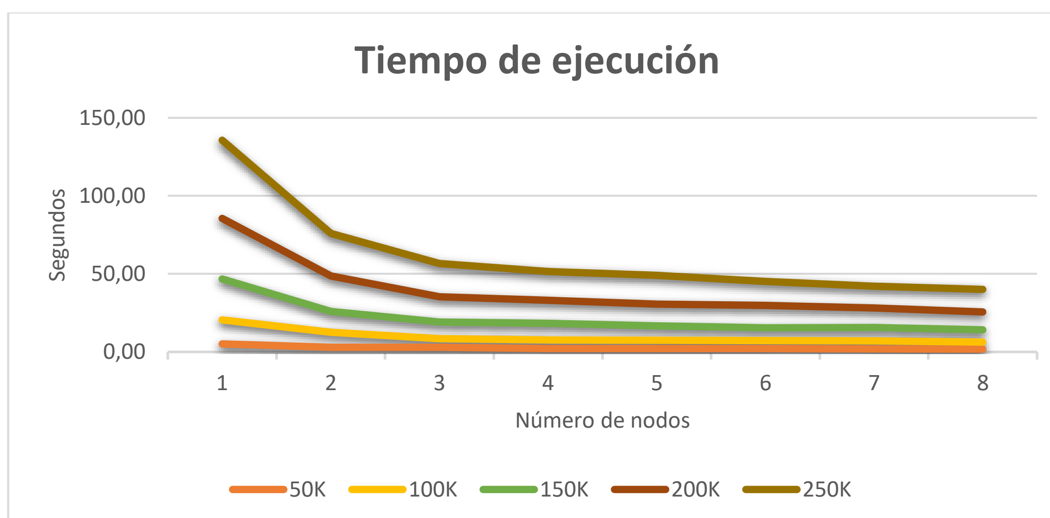


Figura 21. Tiempos de ejecución de conjuntos de datos de distintos tamaños según el número nodos en el sistema.

Se ve que las mejoras, a medida que aumentan el número de nodos en la red, son constantes con el aumento del número de datos. Aunque a priori parece que con el aumento de nodos la mejora sea más lenta, un caso similar pasa con el apartado anterior en la **Figura 19**, donde la mejora en el tiempo de ejecución es más contenida hasta que obtiene un notable salto de 8 a 16 hilos de ejecución.

En el análisis de las multicomputadoras es especialmente importante mantener atención de hasta dónde llega la mejora porque al tratarse de paralelización mediante pasos

de mensajes, estos pueden generar mucha más sobrecarga de comunicación afectando de manera negativa al algoritmo si se abusa de ellos.

Para ello se usan de referencia las demás métricas que darán una visión más objetiva de si es ese el caso. Para ello, ahora se mostrará la **Figura 22** con los Speed-Up obtenidos en comparación a los tiempos de ejecución.

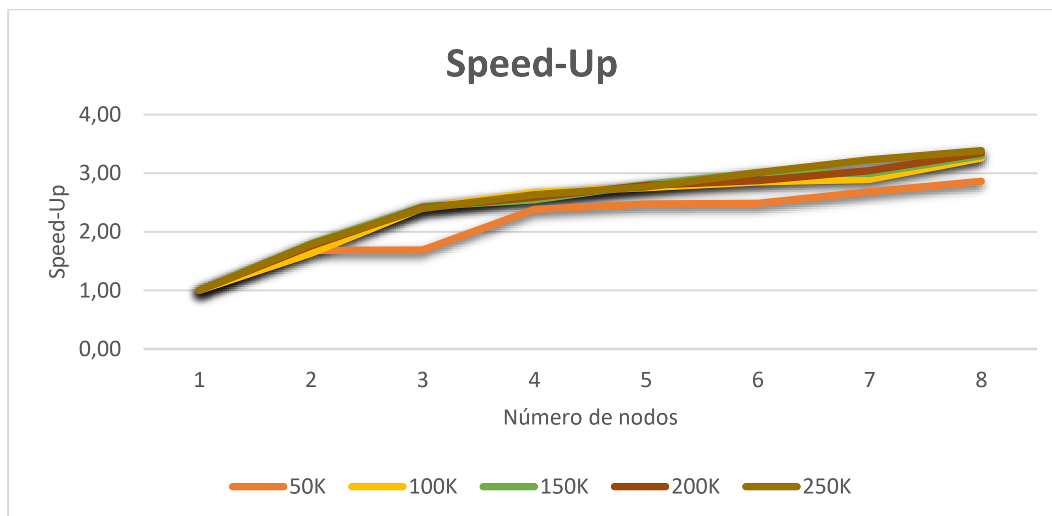


Figura 22. Speed-Up obtenido sobre conjuntos de datos de distintos tamaños según el número de nodos en el sistema.

Gracias a esta gráfica, podemos ver que la mejora en el tiempo de ejecución es constante dentro del rango de numero de nodos que disponemos para esta prueba. Por lo que parece que los pasos de mensaje no tienen una influencia preocupante en esta implementación.

Vistos los resultados del Speed-Up, se procede a calcular la eficiencia del algoritmo en función al número de datos y procesos para poder determinar su escalabilidad.

Numero procesos / Numero Datos	50K	100K	150K	200K	250K
1	1	1	1	1	1
2	0,8415	0,8162	0,8995	0,8801	0,8940
3	0,5628	0,8038	0,8098	0,8053	0,7992
4	0,5961	0,6680	0,6368	0,6477	0,6581
5	0,4952	0,5501	0,5618	0,5582	0,5531
6	0,4147	0,4750	0,5019	0,4793	0,5015
7	0,3832	0,4135	0,4280	0,4345	0,4614
8	0,3576	0,4071	0,4123	0,4177	0,4232

Tabla 9. Eficiencia obtenida sobre conjuntos de datos de distintos tamaños según el número de nodos en el sistema.

Obtenidos los resultados que se ven en la **Tabla 9**, se debe determinar si podemos considerar algún valor constante en diagonal que determinará si este algoritmo es escalable.

De igual manera que en el apartado anterior, se marcarán los recuadros con un color más oscuro que se podrían considerar dentro de un rango de valor constante.

Obtenemos una línea diagonal de valores constantes, pero para la multicomputadora es mucho más contenida que el multiprocesador. Además, los valores de eficiencia no aumentan mucho con el aumento de los datos. Estos ya son potenciales indicativos de que la escalabilidad no será muy grande.

Para poder determinar la isoeficiencia de esta implementación, se considerará como valor constante para su cálculo la eficiencia de 0.59.

Procesos/Datos	50K	100K	150K	200K	250K
1	0	0	0	0	0
2	0,97	4,62	5,22	11,65	16,09
3	4	5,01	10,98	20,68	34,09
4	3,49	10,2	26,66	46,53	70,47
5	5,25	16,78	36,46	67,7	109,6
6	7,27	22,68	46,38	92,95	134,83
7	8,29	29,11	62,46	111,36	158,35
8	9,25	29,88	66,62	119,25	184,91

Tabla 10. Isoeficiencia obtenida sobre conjuntos de datos de distintos tamaños según el número de nodos en el sistema.

El objetivo es obtener, para cada conjunto de datos, un crecimiento pequeño y estable de los valores de la isoeficiencia a medida que se aumentan el número de nodos. Esto nos indicará la escalabilidad del sistema.

En este caso, vemos que el aumento es progresivo para conjuntos de datos más pequeños. A medida que los aumentamos, este crecimiento se hace mucho más abrupto. Por lo que la escalabilidad de este algoritmo en sistemas multicomputador la concluiremos como muy pequeña.

Rompiendo con la presuposición llegada en base al Speed-Up, probablemente debido al retardo del gran número de mensajes que debe procesar el sistema que es igualmente proporcional al número de datos en el conjunto.

Seguirá aprovechando los recursos disponibles, pero para distribuciones mucho más pequeñas.

6.2.3 *Propuesta alternativa*

Debido a que las máquinas multicomputadoras disponen de una memoria principal mucho mayor para todo el conjunto, una propuesta alternativa de implementación seria que se repartiesen iteraciones sobre el conjunto de los datos entre los nodos y estos se precalculasen. El nodo raíz debería pedir conjuntos de datos a medida que los necesitase de otros nodos.

Dependiendo de las máquinas a usar y el volumen de los datos a tratar, se podrían almacenar todos en un solo paso de mensajes Gather o se tendrían que intercambiar conjuntos entre ellos a medida que los necesitasen.

En el peor de los casos, donde tuviésemos muchos puntos juntos y se formase un gran cluster para todos ellos, tendríamos una complejidad espacial de $O(n * n)$, haciendo inviable una implementación con Gather. Si tratásemos el conjunto de datos más pequeño del cual aquí disponemos, el nodo raíz debería almacenar una estructura en su memoria principal una cantidad de 40GB de datos, lo que lo hace imposible para la maquina donde se ejecutaron las pruebas. Aunque si sería posible su implementación en grandes servidores que por lo general disponen de una cantidad más significativa de memoria principal.

Para máquinas más pequeñas como la que se usó en este proyecto, incluso en el escenario del intercambio de los conjuntos de datos, teniendo ocho nodos en el sistema, el raíz debería tener una memoria de por lo menos 10GB para poder almacenar tanto sus datos como los del nodo que se los enviase.

Asumiendo un contexto real en que los clusters serán homogéneos y no ocuparan estructuras de datos tan grandes, los protocolos de envío y recepción de OpenMPI admiten un tamaño máximo de buffer de 4GB de datos por transmisión. Para la estructura de datos en el espacio vectorial de tres dimensiones, esto nos permitiría hacer el envío de aproximadamente un total de 250 millones de puntos en total que fuesen vecinos de otros.

Aunque esto es lo que admite la función, la velocidad de transmisión dependerá de la red de interconexión que se implemente en la multicomputadora. Pero sería una solución válida de todos modos.

Estas son propuestas válidas en casos concretos, por lo que no cubrirían todos los posibles escenarios donde se puedan dar en un uso real. Por lo que estas propuestas las consideraríamos posibles pero pendientes de estudio.

6.3 Unidad de procesado grafico

6.3.1 *Máquina*

Para la unidad de procesado grafico se hará uso del dispositivo *NVIDIA GeForce GTX 1060* [29] con una arquitectura PASCAL y 6GB de memoria dedicada con un ancho de banda para la comunicación de esta de 192GB / segundo. Dispone de 1280 unidades de procesado vectoriales, que son también los que marcan el grado de concurrencia y lo que consideraremos como el número de hilos físicos disponibles.

Esta se usará en combinación con el procesador del apartado anterior que emulaba a la multicomputadora. La conexión que requiere el dispositivo grafico con el procesador usa una tecnología *PCIe 3*, por lo que su ancho de banda por el bus de datos será de poco menos de 1GB / segundo de envío y recepción.

Para un envío o recepción es una velocidad muy considerable, pero la dificultad en este algoritmo reside en el alto número de comunicaciones que se deben realizar y pueden afectar negativamente a los resultados de la implementación.

6.3.2 *Resultados*

Como ya se comentó al inicio del *apartado 6.1*, como es la propia unidad la que distribuye los bloques para ejecutar las tareas, para estas pruebas solo se variará el número de puntos en el conjunto de datos. Estos se compararán con la ejecución en serie del algoritmo mediante el uso del procesador únicamente.

Con ello obtenemos los siguientes resultados:

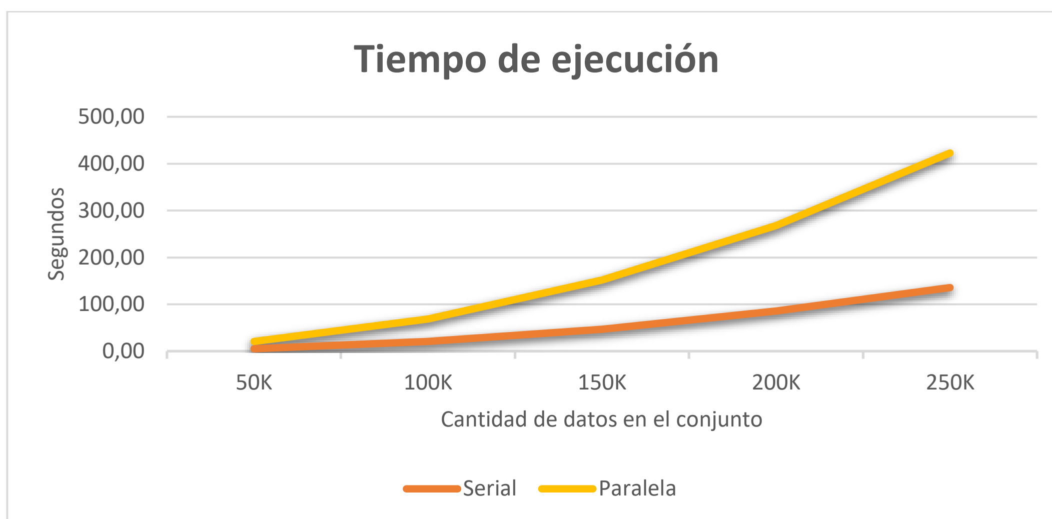


Figura 23. Tiempos de ejecución de conjuntos de datos de distintos tamaños para ejecución en GPU.

Se puede ver que el tiempo de ejecución en paralelo en combinación con la unidad de procesado gráfico tiene un tiempo claramente superior al del algoritmo en serie en todas las variaciones de cantidades de datos.

Esto es debido posiblemente al alto coste generado por la sobrecarga en las comunicaciones entre el anfitrión y el dispositivo. Para esta implementación, el algoritmo debe realizar el mismo número de transferencias de datos como puntos hay en el conjunto. Estas transferencias al tener que hacerse de memoria a memoria principal tienen un elevado coste que repercuten en su eficiencia.

Debido a los resultados negativos, no se procederá al cálculo de las demás métricas porque el tiempo de ejecución paralelo crece de manera exponencial más rápido que en serie. Por lo que no será de ayuda conocer la escalabilidad del algoritmo viendo desde un principio que ya no lo será de provecho para el estudio.

6.3.3 *Propuesta alternativa*

Para probar de ofrecer una solución a este problema y que se pueda conseguir un grado de paralelización eficiente mediante dispositivos de procesado grafico para este algoritmo, se propondrán y hará un breve análisis de algunas técnicas alternativas de implementar este algoritmo.

Una alternativa seria en esta misma implementación que, en el vector que almacena los índices, en vez de escribir una bandera en el índice deseado, guardar los índices en posiciones contiguas. Así se reducirían las iteraciones en la parte serial del algoritmo por parte del procesador ya que solo se iteraría tantas veces como número de vecinos encontrados.

Aunque para ello haría falta un mecanismo de exclusión mutua que asegurase que solo un hilo escribiese cuando un vecino es encontrado para la región.

Esta implementación se realizó y probo. Se obtuvieron resultados peores que los del apartado anterior debido a que estos mecanismos de sincronización son más costosos que iterar en serie sobre los datos recibidos.

Ante esta situación, otra opción sería la de implementar un precálculo de los vecinos como se propuso en el apartado referente a la evaluación de la multicomputadora. Pero incurriríamos en los mismos problemas espaciales.

Aunque con esta técnica surge un problema adicional. Cuando un hilo debe reservar un espacio de memoria, es necesario que previo a la ejecución del código en el dispositivo, se especifique que tamaño tendrá la memoria que pueda reservar cada hilo. En este caso resulta muy complejo determinarlo cuando cada hilo debe encontrar un número no determinado de puntos.

Para comprobar esto, se realizó una implementación de esta técnica. La ejecución de esta versión daba errores de memoria para un conjunto de datos de 50000 puntos y superiores. Por lo que también se descartará.

Con el objetivo de reducir las comunicaciones, se ofrece también la posibilidad de no solo copiar los datos a la unidad de procesado gráfico, si no todas las tareas completas del algoritmo.

De esta manera, un hilo ejecutaría la parte serial y todos los demás calcularían la región del punto indicado a cada iteración. Una vez terminada la ejecución, solo se copiaría el conjunto de puntos con el cluster asignado a la memoria principal del procesador.

Esta técnica también se implementó y obtuvo tiempos de ejecución aún mayores. Aunque las comunicaciones se hayan reducido al mínimo, los núcleos en una unidad de procesado grafico están diseñados para ejecutar tareas sencillas. Pueden ejecutar más complejas también, pero con esto se agrandaría mucho más el número de instrucciones que tiene que ejecutar y los intercambios entre memoria cache y principal que debe realizar.

Como última propuesta adicional, se expone la posibilidad de dividir el espacio de puntos en regiones para poder dividir la carga computacional y asignación de clusters según las regiones. Posteriormente comprobando que puntos de regiones son contiguas y modificando los clusters.

Esta es una implementación que demostró ser efectiva para un conjunto de datos en un espacio vectorial de dos dimensiones. Para uno de tres, como el que estudiamos en este proyecto, los investigadores Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel y Xiaowei XU, han demostrado que en un espacio vectorial igual o superior a tres dimensiones, la computación de este algoritmo se vuelve insostenible por el alto aumento de coste, aunque sea para conjuntos de datos contenidos [8].

Por lo que concluiremos como un algoritmo que no se puede optimizar mediante el uso de una unidad de procesado gráfico.

7 Conclusiones

7.1 Limitaciones de optimización

Durante el transcurso del desarrollo de este proyecto he tenido la oportunidad de realizar una investigación más intensiva sobre algoritmos de Machine Learning. Estos algoritmos son de mucha utilidad y tienen un gran potencial, pero, la optimización de estos puede resultar muy compleja y no efectiva en algunos casos.

Es por ello por lo que la paralelización escogida como el método de mejora para el algoritmo DBSCAN ha supuesto un gran reto tanto de investigación, como de análisis y puesta en práctica de los conocimientos adquiridos a lo largo del transcurso académico universitario.

Cuanto más se agranda el algoritmo y más funciones se tienen que usar, más costoso se hace poder realizar el análisis con precisión. Una vez completado este análisis, aquí he podido concluir que proponer un algoritmo paralelo requiere de mucha más dedicación de la que parecía en un inicio.

Y no solo la dedicación es necesaria, también es de gran ayuda o incluso imperativo disponer de un conjunto de herramientas y máquinas adecuadas para poder realizar las pruebas necesarias con diferentes heurísticas y situaciones de uso real. Pudiendo asegurar un análisis más certero de la optimización propuesta y una implementación correcta.

Las propuestas expuestas en esta memoria no son las únicas ni las mejores. Hay una infinidad de métodos de optimizar este algoritmo y otros tantos mediante la paralelización. Pero las propuestas que se investigaron aquí se han hecho enfocadas a abarcar las distribuciones del mayor número de máquinas posibles. Para que se pueda ejecutar tanto en una supercomputadora, como en un ordenador con unas capacidades muy limitadas con una mejora satisfactoria y unos resultados correctos.

Por lo que de este apartado puedo expresar mi gran voluntad de haber dispuesto de más tiempo para poder desarrollar esta idea, dándome la posibilidad de investigar más a fondo sobre propuestas válidas para multicomputadoras y unidades de procesamiento gráfico. Y la emoción de poder llegar a hacerlo algún día.

7.2 Aprendizaje

Desarrollando esta propuesta de paralelización, he tenido la oportunidad de hacer uso de mucho conocimiento y técnicas aprendidas durante las lecciones. Pero también la de aprender nuevos métodos de análisis y de programación.

La lección, posiblemente más importante aprendida en este proyecto, es la importancia de realizar los análisis y comprobaciones necesarias antes de proceder a la propia programación. Ya que es un excelente método de ahorrar trabajo innecesario y posibles nervios en el caso de invertir mucho tiempo en implementar algo que no resulta mejor que lo original.

Este trabajo previo a la implementación también es de gran ayuda para la comprensión del problema. Expande mucho más su horizonte y acota el abanico de posibles opciones a algunas pocas que pueden resultar más efectivas.

Gracias a estas comprobaciones y sus implementaciones, he podido aprender nuevas heurísticas referentes a las optimizaciones locales del algoritmo que, aunque pequeñas, pueden presentar algunas mejoras.

Durante el diseño también he podido desarrollar la habilidad de ser más consciente de que no todo el hardware es igual. En algunos casos hay que amoldarse a la situación actual, pero siempre con el objetivo de globalizar la implementación al máximo número de casos posibles.

La cantidad de procesos activos que puede tener una maquina y el diseño de las memorias internas juegan un papel fundamental en el momento de tomar las decisiones de diseño. Gracias a los conocimientos sobre ellos he podido proponer una paralelización eficiente, descartando las que no lo son y probar algunas de las que tenía dudas como en el caso de ejecutar todo el algoritmo en la unidad de procesado gráfico.

Por las evaluaciones he podido resaltar la importancia de saber interpretar bien los resultados. Una buena comprensión de estos hace poder determinar con precisión si se está tomando un buen camino o es mejor desechar esta opción y tomar otro.

Con la interpretación de los resultados no solo se pueden llegar a estas conclusiones, sino que también se puede expandir un conjunto relativamente pequeño de resultados a posibilidades más amplias mediante la comprensión de la teoría aplicada. Gracias a estas he podido determinar no solo la escalabilidad del sistema, si no cual es la combinación de parámetros paralelos y máquinas a usar para cada tamaño de conjunto de datos.

7.3 Futuro

Gracias a la información previa recopilada referente al campo de la inteligencia artificial y el análisis del algoritmo DBSCAN y otros tantos que he tenido la oportunidad de ojear para la recopilación de ideas, he logrado aumentar el interés por este campo de trabajo.

Teniendo en cuenta que la inteligencia artificial es un campo que esta aun en desarrollo y, parece que lo seguirá estando durante un tiempo largo si es que alcanza un límite, es un ámbito en el que merece la pena centrar el esfuerzo.

Durante este proyecto, se ha podido germinar la idea de una posible pasión que desarrollar en el futuro para poder aportar nuevas ideas y mejoras que puedan llegar a ser de gran utilidad, llegar a ser implementadas para distintos ámbitos y, sobre todo, puedan llegar a facilitar las situaciones y el uso a profesionales y usuarios.

Como conclusión global, estoy muy satisfecho con lo aprendido en este proyecto y con ilusión de poder proseguirlo algún día o poder profundizar más en este campo de estudios.

8 Referencias

8.1 Bibliografía

[1] Libro: Geoffrey Hinton, Terrence J. Sejnowski, Unsupervised Learning and Map Formation: Foundations of Neural Computation, MIT Press, 1999.

[2] Libro: Jain A, Dubes R, Algorithms for clustering data, Prentice-Hall Inc., 1988.

[3] Artículo de revista: Xu R, Wunsch D, Survey of clustering algorithms, IEEE Trans Neural Network, 2005, 16, 645-678

[4] Informe de revista: Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, In Proceedings of the 2nd ACM International Conference on Knowledge Discovery and Data Mining (KDD), 1996, 2, 226-261.

[5] Libro: Han J, Kamber M, Pei J., Data Mining: Concepts and Techniques, San Francisco (US): Morgan-Kauffman, 2012.

[6] Informe de revista: I. E. Scarinci, P. Pérez, M. Valente, Algoritmo heuristico de segmentacion de imagenes PET utilizando tecnicas de intel·ligència artificial, Anales AFA, 2020, 31, 4, 165-171.

[7] Informe de revista: Ruth Reátegui , Edgar Daz , Ronald Campozano2, Análisis de conglomerados para identificar grupos de pacientes diabéticos en base a síntomas, RISTI, 2019, 27, 3, 384, 394.

[8] Informe de Revista: Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, Xiaowei XU, DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN, ACM Trans. Database Syst, 2017, 42, 3, 21.

8.2 Recursos web

[9] H.Antonio Vazquez Brust. *Machine learning para detectar centros de actividad urbana.* https://bitsandbricks.github.io/post/dbscan-machine-learning-para-detectar-centros-de-actividad-urbana/ [Consulta: 13/06/2022].

[10] Dongkuan Xu, Yingjie Tian. *A comprehensive survey of clustering algorithms.* https://link.springer.com/article/10.1007/s40745-015-0040-1 [Consulta: 15/06/2022].

[11] Gagarine Yaikhom. The DBSCAN clustering algorythm. https://github.com/gyaikhom/dbscan [Consulta: 15/06/2022].

[12] *2014 SIGKDD Test of Time Award*. https://www.kdd.org/News/view/2014-sigkdd-test-of-time-award [Consulta: 14/06/2022].

[13] *Como funciona el clustering basado en densidad.* https://pro.arcgis.com/es/pro-app/latest/tool-reference/spatial-statistics/how-density-based-clustering-works.htm [Consulta: 20/06/2022].

[14] *¿Cuáles son los conceptos de minería de datos más importantes?* https://quesignificado.org/cuales-son-los-conceptos-de-mineria-de-datos-mas-importantes/ [Consulta: 20/06/2022].

[15] *OpenMP*. https://es.wikipedia.org/wiki/OpenMP [Consulta: 28/06/2022].

[16] *General information about the Open MPI project.* https://www.open-mpi.org/faq/?category=general [Consulta: 17/07/2022].

[17] *NVIDIA CUDA C Programming guide version 4.2*. https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf [Consulta: 04/08/2022].

[18] Dominik Göddeke and Robert Strzodka. Scientific computing with GPUs. https://campusvirtual.urv.cat/pluginfile.php/4245072/mod_resource/content/1/04-hardware.pdf [Consulta: 05/08/2022].

[19] *OPENMP API Specification 2.1 Directive Format*. https://www.openmp.org/spec-html/5.0/openmpse9.html#x31-310002.1 [Consulta: 22/06/2022].

[20] *Parallel Construct.* https://www.openmp.org/spec-html/5.0/openmpse14.html#x54-800002.6 [Consulta: 22/06/2022].

[21] *Pragma omp for:* https://www.ibm.com/docs/en/xcfbg/121.141?topic=processing-pragma-omp [Consulta: 22/06/2022].

[22] *Pragma omp barrier:* https://www.ibm.com/docs/en/xl-c-aix/11.1.0?topic=processing-pragma-omp-barrier [Consulta: 24/06/2022].

[23] *Parallel master construct:* https://www.openmp.org/spec-html/5.0/openmpsu68.html#x95-3290002.13.6 [Consulta: 24/06/2022].

[24] *MPI_Bcast.* https://www.mpich.org/static/docs/v3.1/www3/MPI_Bcast.html [Consulta: 18/07/2022].

[25] *MPI_Gather.* https://www.mpich.org/static/docs/v3.3/www3/MPI_Gather.html [Consulta: 19/07/2022].

[26] *MPI_Gatherv.* https://www.mpich.org/static/docs/v3.3/www3/MPI_Gatherv.html [Consulta: 22/07/2022].

[27] *Procesadores AMD Ryzen Threadripper PRO 3995WX.* https://www.amd.com/es/products/cpu/amd-ryzen-threadripper-pro-3995wx [Consulta: 24/08/2022].

[28] *Procesadores moviles AMD Ryzen 7 6800HS.* https://www.amd.com/es/products/apu/amd-ryzen-7-6800hs [Consulta: 02/09/2022].

[29] *GeForce GTX 1060 Specifications.* https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1060/specifications/ [Consulta: 25/08/2022].

[30] *El sistema operativo GNU.* https://www.gnu.org/home.es.html [Consulta: 10/06/2022].

[31] Transparencias de Computación paralela y masiva. *Paral·lelisme.* https://campusvirtual.urv.cat/pluginfile.php/4245040/mod_resource/content/0/Transparencies/Tema_I_Lab_CPM.pdf [Consulta: 21/06/2022].

[32] Transparencias de Arquitecturas Paralelas II. *Arquitecturas especiales.* https://campusvirtual.urv.cat/pluginfile.php/4245049/mod_resource/content/0/Transparencies/2013_T4p.pdf [30/06/2022].

[33] Transparencias de arquitectura de computadores. *Analisis de processadores paralelos.* https://campusvirtual.urv.cat/pluginfile.php/3973599/mod_resource/content/1/AC_Tema3_Paralelos.pdf [Consulta: 14/08/2022]

[34] Transparencias de computación paralela y masiva. *Multiprocessadors Memòria Compartida.* https://campusvirtual.urv.cat/pluginfile.php/4245042/mod_resource/content/0/Transparencies/Tema2_1_CPM_1213.pdf [Consulta: 20/06/2022]

8.3 Software

[35] Visual Studio Code. https://code.visualstudio.com/ [Consulta: 10/06/2022].

[36] WSL2. https://docs.microsoft.com/es-es/windows/wsl/install [Consulta: 10/06/2022].

[37] GCC. https://gcc.gnu.org/ [Consulta: 10/06/2022].

[38] G++. https://packages.ubuntu.com/bionic/g [Consulta: 10/06/2022].

[39] Comprobación de compatibilidad con CUDA. https://developer.nvidia.com/cuda-gpus [Consulta: 05/08/2022].

[40] CUDA toolkit. https://developer.nvidia.com/cuda-toolkit [Consulta: 05/08/2022].

Anexos

Anexo A. Archivos necesarios

En este apartado se presentan los archivos necesarios para la compilación y la ejecución del algoritmo. Estos son:

Código

Se adjuntan en este trabajo los códigos necesarios para ejecutar el programa. Tanto la original en serie como las modificadas para optimizar el algoritmo a una ejecución paralela. Estos archivos son:

- **Versión original**: dbscan.c
- **OpenMP**: dbscan_OMP.c
- **OpenMPI**: dbscan_MPI.c
- **CUDA**: dbscan_CUDA.cu

Ejecutables

Una vez compilados estos códigos se generan archivos ejecutables. Aunque en el *Anexo B* se explica como generarlos, también se adjuntarán en este trabajo:

- **Versión original**: dbscan
- **OpenMP**: dbscan_OMP
- **OpenMPI**: dbscan_MPI
- **CUDA**: dbscan_CUDA

Conjunto de datos

Estos archivos son los que representan el conjunto de datos de entrada para ser procesados por el algoritmo.

En ellos, la primera fila indica los parámetros de entrada Epsilon, mínimo de puntos y puntos totales respectivamente. El resto de las filas son todos los puntos en el espacio vectorial de tres dimensiones donde cada fila representa a un punto.

Se adjunta la siguiente lista con todos los archivos usados para el análisis en esta memoria. Donde el primer archivo es usado para comprobar los resultados de la implementación en un conjunto de datos pequeño y el resto para comprobar el tiempo de ejecución:

- 53.dat
- 50000.dat
- 100000.dat
- 150000.dat
- 200000.dat
- 250000.dat

Anexo B. Instalaciones necesarias

Para poder ejecutar los algoritmos, se describirán los programas y librerías necesarias y la manera de instalarlos.

Los estándares OpenMP y OpenMPI están muy extendidos al uso de casi todos los procesadores que existen actualmente. Y como se explicó en esta memoria, un procesador puede realizar la función de multicomputadora para ejecutar programas en este estándar. Por lo que ambos se pueden ejecutar en todos ellos y los requisitos de hardware se darán por satisfechos.

En el caso de la ejecución en CUDA, sí que es necesario comprobar que el dispositivo de procesado grafico es compatible previamente con esta librería. Esta comprobación también se explicará en este apartado.

Instalación de programas base

Para la ejecución de todos ellos serán necesarias las siguientes herramientas:

- Entorno de trabajo Linux o alguno que lo emule. En este proyecto se usó el entorno de desarrollo de programas *Visual Studio Code* [35] con la instalación del subsistema de Linux en Windows *WSL2* [36] con distribución Ubuntu que permitía el uso de su terminal.
- Disponer de herramientas de compilación para los lenguajes de programación C y C++. Se usará el compilador proporcionado por GNU[30], el GCC [37] para compilación en C y el proporcionado por la propia distribución de Ubuntu, el G++ [38] para la compilación en C++. Para instalarlos se usarán los siguientes comandos en la terminal de WSL2:
    - *sudo apt install gcc*
    - *sudo apt install g++*

Instalación de librerías OpenMP

La API OpenMP es una librería para el lenguaje de programación. Por lo que para poder hacer uso de ella y compilarla solo habrá que ejecutar un comando en el terminal para instalarla:

- *sudo apt-get install libomp-dev*

Una vez instalado el paquete, para que este sea funcional, habrá que incluir su cabecera en el código: *"omp.h"*.

Instalación de librerías OpenMPI

El estándar OpenMPI es del mismo modo una librería, pero también requiere un compilador propio para que se puedan compilar los algoritmos escritos con su sintaxis.

Aunque del mismo modo la instalación se realizará mediante un solo comando en el terminal que instalará todos los paquetes necesarios:

- *sudo apt-get install mpich*

También se deberá añadir a la cabecera de su librería en los códigos que lo usen: *"mpi.h"*.

Instalación de librerías CUDA

Primero se debe comprobar que el dispositivo de procesado grafico del que disponemos sea compatible con el entorno de desarrollo de CUDA [39].

Si nuestro dispositivo aparece como compatible, se debe proceder a la instalación del paquete de herramientas [40] en el sistema operativo base, ya sea Windows o Linux. Ya que el acceso a las comunicaciones con el dispositivo estará bloqueado hasta que el sistema operativo base no los autorice mediante este conjunto de herramientas.

Una vez instaladas las herramientas, de la misma manera que en los otros estándares, solo habrá que instalar las librerías y paquetes necesarios mediante el terminal para poder disponer de la herramienta de compilación de CUDA mediante el comando:

- *sudo apt install cuda*

Anexo C. Compilación y ejecución

En este apartado se describirá la manera y los comandos usados para la compilación y la ejecución de todas las versiones del algoritmo implementadas.

Algoritmo Secuencial

Para la compilación del algoritmo original se usará el compilador GCC con las siguientes banderas:

- **-lm**: Indica al compilador que se usara la librería "math.h" para la implementación.
- **-O3**: Indica el nivel de optimización que debe aplicar el compilador.
- **-o**: Indica el nombre del archivo ejecutable generado.

Así pues, el comando a usar quedaría de la siguiente manera:

- *gcc -O3 -o dbscan dbscan.c -lm*

Para su ejecución solo se deberá lanzar el programa de manera habitual introduciendo el conjunto de datos deseado mediante el terminal:

- *./dbscan < 53.dat*

Paralelización mediante OpenMP

Esta compilación es prácticamente idéntica a la usada para el algoritmo secuencial debido a que OpenMP es una librería del propio lenguaje. Para ello, al mismo comando se le añadirá la bandera **-fopenmp** que indicará que se están usando su librería y directivas:

- *gcc -fopenmp -O3 -o dbscan_OMP dbscan_OMP.c -lm*

El procedimiento para su ejecución es idéntico al del algoritmo secuencial.

- *./dbscan_OMP < 53.dat*

Paralelización mediante OpenMPI

La compilación en OpenMPI es propia, pero mediante el compilador GCC. Es por ello por lo que se usan las mismas banderas y parámetros que en el algoritmo secuencial, pero el comando se cambia por **mpicc**:

- *mpicc -O3 -o dbscan_MPI dbscan_MPI.c -lm*

Cuando se genera el archivo, también cambia la manera de ejecutarlo. No se ejecuta de la manera habitual mediante el terminal. Se debe hacer mediante un comando específico del conjunto de herramientas de este estándar. También se añadirá la bandera **-np**, que especificará el número de nodos en la multicomputadora con los que se procesará:

- *mpirun -np N dbscan_MPI < 53.dat*

Donde *N* es el número de nodos que se quieren asignar a la ejecución.

Paralelización mediante CUDA

Del mismo modo que OpenMPI, CUDA necesita una compilación propia para poder transferir las operaciones y datos con el dispositivo de procesado gráfico. Para ello se usarán casi los mismos parámetros, pero cambiando el comando a **nvcc**, quedándonos así el siguiente comando que debe ser introducido por el terminal:

- *nvcc -O3 -o dbscan_CUDA dbscan_CUDA.cu*

Para esta compilación no es necesario incluir la librería “math.h” en el código ni su bandera en el comando ya que las librerías de CUDA disponen de sus propias operaciones equivalentes.

Aunque su ejecución sí que se realiza directamente desde el archivo ejecutable creado por el compilador:

- *./dbscan_CUDA < 53.dat*

Anexo D. Archivos de salida

Una vez terminado el algoritmo de cualquiera de las versiones, el código genera un nuevo archivo en el que se guardan los puntos del archivo de entrada añadiendo a que cluster pertenecen.

El archivo generado por el algoritmo ejecutado en serie debe contener los mismos datos que cualquiera de los generados por las ejecuciones paralelas. Es de este modo que se comprueba que la optimización mediante paralelización ha sido correcta.

Los archivos siguen los siguientes formatos para los distintos estándares de programación:

- **Versión original**: dbResults.dat
- **OpenMP**: dbResultsOPM.dat
- **OpenMPI**: dbResultsMPI.dat
- **CUDA**: dbResultsCUDA.dat