USO DE OPENMP EN LOS ALGORITMOS K-MEANS Y FLOYD-WARSHALL

Clàudia Pàmies Vaqué, Joana Alonso Mayor

Arquitecturas Avanzadas 2025/26 - Juan Carlos Moure

ÍNDICE

INTRODUCCIÓN

El objetivo principal del trabajo realizado es comprender en profundidad el funcionamiento de algoritmos de aprendizaje no supervisado y estudiar cómo mejorar su rendimiento aplicando técnicas de optimización y paralelización.

Concretamente, se analizan y optimizan las implementaciones de K-means y Floyd-Warshall usando las herramientas y directrices aprendidas en el transcurso de la asignatura de Arquitecturas Avanzadas, incluyendo tanto opciones de compilación como las directrices de OpenMP para paralelizar el código en entornos de memoria compartida.

La decisión de realizar este trabajo se basa en la relevancia que tienen dichos algoritmos tanto a nivel practico como didáctico:

- K-means es un método estándar para la clusterizacion y tiene aplicaciones amplias en visión por computador y/o minería de datos.

- Floyd-Warshall se usa para poder calcular caminos mínimos entre todos los pares de nodos en grafos y nos sirve para poder comprender dependencias y optimizaciones en operaciones matriciales.

Este trabajo documenta la implementación secuencial (brindada en el Campus Virtual de la asignatura), la identificación de los principales cuellos de botella, aplica estrategias de optimización con OpenMP y cuantifica el impacto de dichas mejoras mediante medidas de rendimiento y análisis de escalabilidad. Finalmente, adjuntamos una pequeña discusión sobre sus limitaciones, problemas encontrados a lo largo del trabajo y posibles líneas de trabajo futuro.

MARCO TEÓRICO

Este punto del trabajo es usado para definir los conceptos y herramientas sobre los que se basa dicho trabajo: los algoritmos k-means y Floyd-Warshall, los principios de paralelizacion en memoria compartida y las directrices de OpenMP. El objetivo es identificar los puntos de coste computacional y las oportunidades de paralelismo, así como las posibles limitaciones (dependencias, acceso a memoria y/o sincronización).

3.1. K-means:

K-means es un algoritmo de aprendizaje no supervisado para la clusterización. Dado un set de datos representado con vectores, K-means busca particionar los puntos en M clústeres de manera que la suma de las distancias (normalmente euclidianas) entre cada punto y su centroide del clúster sea mínima.

La complejidad de la aplicación de este algoritmo es de O(n*k*d) por la fase de asignación y O(n*d) para la recomputación de centroides, donde n equivale al número de puntos, k las distancias que tenemos que calcular para cada punto y d la dimensión del espacio en el que trabajamos. El número de iteraciones que debe de realizar el programa depende de la inicialización y del dataset que otorguemos como input.

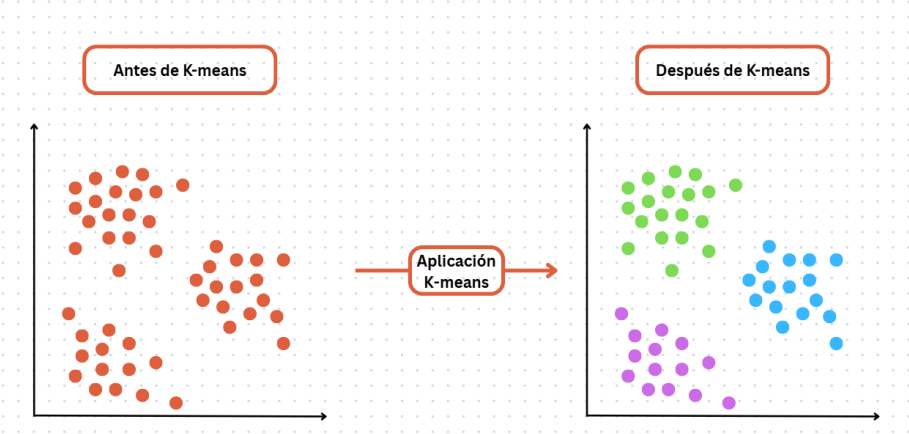
Consideramos necesaria la definición de centroide para definir más en profundidad el funcionamiento de dicho algoritmo. Los centroides (los cuales no necesariamente debe coincidir con un punto real del conjunto) son el centro geométrico (mediana) de los puntos asignados a cada clúster. El algoritmo opera iterativamente:

- Asigna cada punto al clúster más próximo
- Recalcula los centroides
- Repite estos pasos hasta la convergencia

Como resultado se obtiene una partición del espacio de datos en grupos lo más compactos posible respecto a la distancia.

El número de clústers k debe de definirse antes de la ejecución del algoritmo ya que dicho valor no se determina automáticamente.

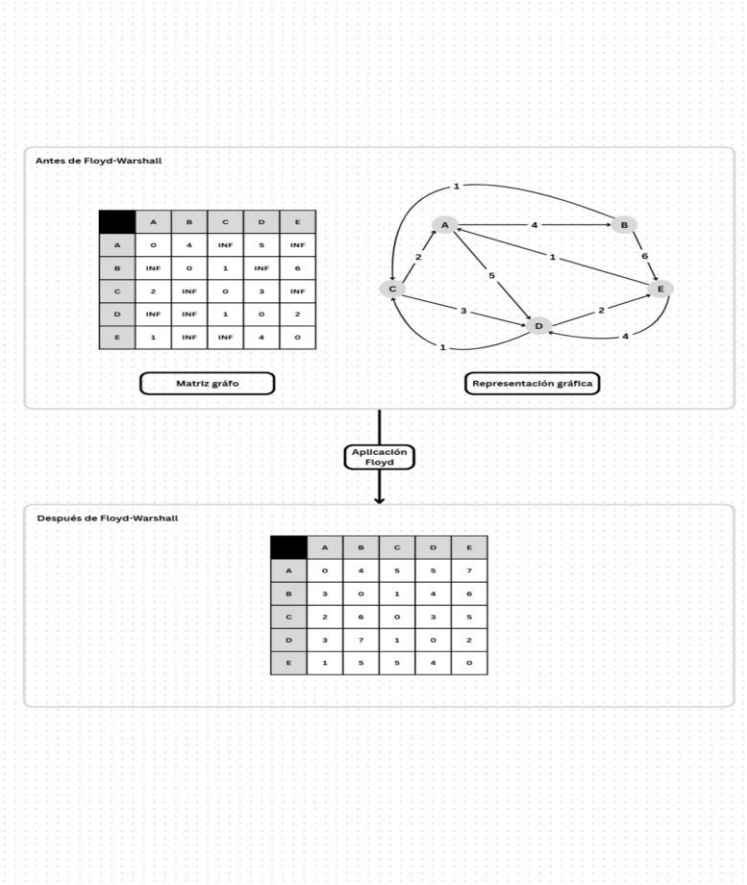
A pesar de ser un algoritmo muy utilizado dentro del campo de la minería de datos y el machine learning presenta ciertas limitaciones como, por ejemplo, no tiene un funcionamiento explícitamente optimo si los clusters no tienen forma específica o densidades muy distintas.



[Fig 1] Representación gráfica de la aplicación del algoritmo a un conjunto de puntos.

3.2. Algoritmo Floyd-Warshall:

El algoritmo de Floyd-Warshall calcula las distancias mínimas entre todos los pares de nodos de un grafo dirigido o sin dirigir ponderado. Es una solución basada en la programación dinámica con una representación típica mediante una matriz M de dimensiones NxN.



[Fig 2] Representación matricial y gráfica del resultado de la aplicación del algoritmo Floyd-Warshall a una matriz de 5x5

Su coste computacional es de O(n^3) a nivel temporal y O(n^2) a nivel de memoria. La dependencia principal es que la iteración en k es secuencial. Para calcular la iteración k se requieren los valores resultantes de iteraciones anteriores en k. Sin embargo, para una k fija, las actualizaciones de cada par de nodos (i,j) son independientes y, por lo tanto, paralelizables.

El código de Floyd-Warshall proporcionado en el Campus Virtual contiene ciertas decisiones concretas que afectan a al rendimiento del programa. Principalmente encontramos las siguientes:

- La matriz se inicializa con valores aleatorios generando así una matriz parcialmente densa, lo que implica que vamos a tener una carga mayor de trabajo operacional.
- El orden de los bucles perjudica la localidad de caché, ya que estamos accediendo a la matriz con un layout row-major [for(k) for(j) for (i)], esta manera de ordenar los bucles implica un acceso no adyacente en memoria. Si realizamos el orden de los bucles tal que: for(k) for(i) for(j), entonces accedemos a elementos almacenados en memoria de manera contigua, así podemos explotar la caché más óptimamente.

Estos primeros puntos son observables sin la necesidad de perfilar el código, más adelante, en el apartado del marco práctico (INSERIR PÁGINA), dónde explicamos las optimizaciones realizadas para obtener una mejoría del tiempo demostramos, mostrando los datos de los experimetos realizados que, tal y como hemos podido observar, el rendimiento del programa se ve afectado por ellos.

3.3. Modelo de paralelización:

El modelo de paralelización considerado en este trabajo es el de memoria compartida con threads (programación con OpenMP). En este modelo, todos los threads comparten el espacio de direcciones y tienen cachés privadas, hecho que nos permite paralelizar bucles de manera sincronizada.

Los problemas y consideraciones fundamentales para el desarrollo de este documento han sido:

- **Race conditions**: Suceden cuando distintos threads leen y escriben el mismo dato sin sincronizar. Debemos identificar variables globales modificadas concurrentemente y protegerlas con reducciones, operaciones atómicas o mecanismos para agregar localmente a los threads dichas variables.
- **False sharin**g: Nos encontramos con este problema en el momento en que variables independientes que son accedidas por distintos threads quedan en la misma línea de caché (bloque de datos contiguo en memoria principal, cuando tratamos de acceder con un thread a una variable, la CPU carga toda la linea de caché donde está dicha variable, no solo la variable).
- **Overhead de sincronización**: Es necesario minimizar la sincronización dentro de los bucles que "se llevan" más tiempo por problemas de sincronización entre threads.
- **Load Imbalance**: Si las unidades de trabajo tienen cargas de trabajo desiguales algunos threads terminarán antes y se podrá dar el caso de que estén mucho tiempo esperando,

sin realizar ninguna tarea. Haciendo uso de schedule en OpenMP podemos solucionar este problema.

3.4. Introducción a OpenMP:

Antes de aplicar las técnicas de paralelización a los algoritmos estudiados, consideramos necesaria la introducción (breve) de OpenMP ya que, como hemos comentado previamente, es el modelo de programación paralela usado en este trabajo.

OpenMP nos proporciona un conjunto de directivas sencillas que permiten expressar paralelismo a nivel de bucle y datos en arquitecturas de memoria compartida. Mediante éstas directivas podemos crear y gestionar threads, repartir la carga de trabajo, sincronizar los accesos a memória y controlar el rendimiento de manera eficiente.

A continuación, presentamos las funcionalidades de OpenMP más relevantes tratadas en clase. Debemos especificar que hay algunas funcionalidades descritas a continuación que no hemos usado para nuestra paralelización, pero consideramos importante explicar ya que, para trabajos futuros y/o futuras optimizaciones en algoritmos similares a los trabajados, podrían ser usadas:

· **#pragma omp parallel**: Crea un equipo de threads.

· **#pragma omp for**: Distribuye iteraciones de un bucle entre threads. Se puede combinar (**#pragma omp parallel for**)

· **atomic y critical**: Para proteger actualizaciones concretas. Atomic es más ligero, pero solo lo usamos para operaciones atómicas simples.

· **schedule(static|dynamic|guided[,chunk]**: Controla como se asignan iteraciones, **static** es baja sobrecarga y **dynamic** mejora el balance con coste variable.

· **collapse(n)**: Combina n bucles anudados en una sola iteración para repartir mejor la carga de trabajo.

· **simd**: Fuerza vectorización con SIMD (Single Instruction, Multiple Data).

· **omp_get_thread_num()**, **omp_get_num_threads()**: Funciones útiles para la comprensión de OpenMP. **omp_get_thread_num()** nos indica en que thread estamos y **omp_get_num_threads()** con cuantos threads estamos ejecutando el programa.

Además, también debemos tener en cuenta distintas prácticas como:

· Especificar el ámbito de variables (private, shared, firstprivate o ninguna) para evitar confusiones y/o errores de race.

· Solo debemos usar atomic cuando realizamos operaciones simples.

· El uso de collapse nos ayuda a evitar overheads ya que nos sirve para fusionar distintos bucles anidados en un único espacio y así repartir el trabajo más equitativamente.

STATE OF ART

K-means:

Floyd-Warshall:

ENTORNO EXPERIMENTAL

4.1. Maquinaria y sistema:

Processador	12th Gen Intel(R) Core(TM) i5-12400	Freq. base: 2500 MHz	12 cores (2 hilos x núcleo)
Memoria	46 Gi, RAM		
Cachés	48K (L1d), 32K (L1i)	1280K (L2)	18432K (L3)
Sistema Operativo	Rocky Linux 8.19 (Green Obsidian)		

Para poder obtener esta información hemos usado el comando “lscpu” y un script de bash que hemos adjuntado (en la carpeta del trabajo/repositorio de GitHub) llamado “config.sh”.

Para ejecutar el script de bash hemos tenido que dar permisos junto el comando “chmod +x config.sh”.

4.2. Compilación del programa:

Compilador usado y versión:

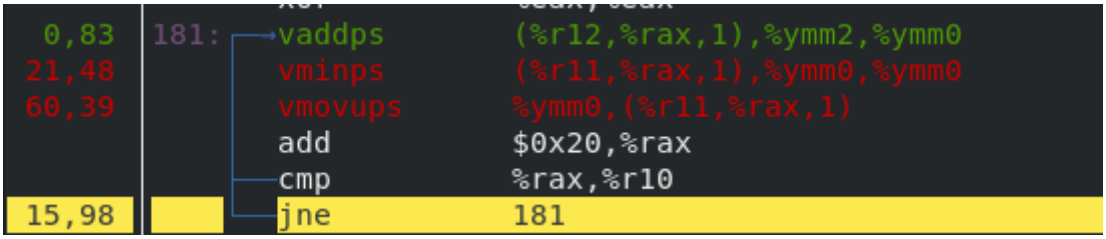
Hemos usado **gcc** (Red Hat 13.2.1-6) ya que los códigos presentados se han escrito en C, la elección de la versión del compilador es debida a que esta versión tiende a identificar correctamente cuando nos encontramos en un condicional para encontrar un mínimo y un máximo y, en consecuencia, al generar el binario lo expresa debidamente.

Para conseguir el binario de Floyd-Warshall hemos usado los siguientes flags de compilación:

-O3: PROVAR AMB –O2 i –OFAST El flag –O3 del compilador gcc activa todas las optimizaciones especificadas por –O2 y otros flags como, por ejemplo, -floop-unroll. Hemos decidido usar este flag en vez de –O2 porque el resultado funcional es el mismo y, además, va más rápido. Pero se debe tener en cuenta que, dependiendo del programa, el flag –O2 podría generar un binario que se ejecutara más rápido.

-march=native: Aplicando este flag, nos aseguramos de que la compilación esté lo más optimizada posible para nuestra arquitectura. Dependiendo de la arquitectura que se use, la ejecución del programa puede ser más rápida o lenta (o similar) a la obtenida y presentada en este documento.

–fopenmp: Este flag también ha sido usado, pero solo en el momento de paralelizar el programa. Para la versión secuencial hemos usado solo –O3 y –march=native (a parte del flag –o para determinar el nombre de salida del binario).



[Fig. 3] Ensemblador Floyd-Warshall modificado y compilado con los flags especificados, bucle más interno del programa

4.3 Dimensiones del problema:

K-means:

Número de puntos N: 1K, 10K, 100K

Dimensionalidad D: 3, 10, 50

Numero de centroides C: 4, 16, 64

Floyd-Warshall:

Tamaño de la matriz N: 500, 1K, 2K, 3K

Densidad simulada de la matriz: En nuestro código un 65%, esto implica que, un 65% de la matriz tendrá números determinados a partir de la semilla elegida, el resto de los valores tomarán el valor INF (variable global dónde se ha hardcodeado el valor 999999).

La primera parte experimental de Floyd-Warshall ha sido realizada usando un tamaño de matriz de 3000 x 3000.

4.4. Metodología de medidas:

Para hacer la recolección de estadísticas y comprobar que el código va más rápido y sigue teniendo el mismo resultado hemos seguido las siguientes indicaciones:

· Control del número de threads.

· Uso de warm-up (ejecutar dos iteraciones de calentamiento antes de recojerlos datos para estabilizar las cachés y politicas del sistema).

· Repeticiones: Para cada configuración hemos realizado 5 repeticiones.

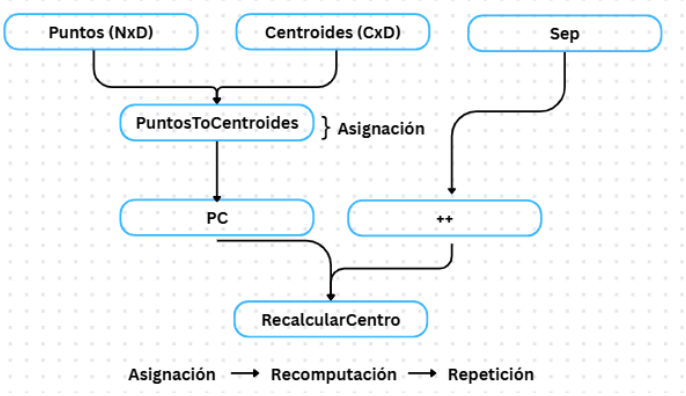
· Uso de perf stat, perf record/report para el analisis de regiones críticas del programa además de la recolección de datos.

· Hacer uso de la variable **checksum** para comprobar que el resultado funcional sigue siendo el mismo a pesar de los cambios realizados.

Para asegurarnos de que todos estos puntos se realizan de manera regular para todas las ejecuciones y para facilitarnos la taerea de analisis hemos usado el script de ejecución (adjuntado como ejec.sh) el cual hace uso del comando COMANDO USADO PREGUNTAR A CARLOS.

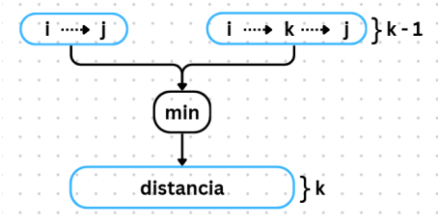
IMPLEMENTACIÓN SEQÜENCIAL

5.1. Dependencias K-means sequencial:



[Fig. 4] Dependencias K-means

5.2. Dependencias Floyd-Warshall sequencial:



[Fig. 5] Dependéncias Floyd-Warshall

5.3. Resultados sequenciales y paralelos:

K-means:

Flags compilacion	Kmeans.c	Kmeansmod.c
-O3		
-O3 -march=native		

Dónde Kmeansmod.c hace referencia al código de K-means con el uso de OpenMP.

Floyd-Warshall:

Flags compilación	Floyd.c	Floydmod.c	FloydBlocked.c
-O3	23.59 s	12.25 s	9.20 s
-O3 –march=native	23.82 s	4.57 s	8.70 s

Dónde Floydmod.c y FloydBlocked.c hacen uso de OpenMP.

1. DESARROLLO DE LAS OPTIMIZACIONES

Para ambos algoritmos hemos hecho el uso debido de las directrices de OpenMP descritas anteriormente para distribuir el trabajo entre los distintos cores de la mejor manera posible.

Además, debemos de entender con profundidad el uso de las cachés privadas de cada core. A continuación, describimos la lógica seguida para cada algoritmo:

6.1. K-means:

En el caso de K-means, nos encontramos con el hecho de que lo que hace que el algoritmo tarde más a encontrar una solución al problema propuesto es el tamaño de los datos de entrada del programa. En la siguiente tabla se ilustra dicho hecho, cambiando solamente el número de puntos que debemos computar:

Con una entrada estable de Puntos tridimensionales que se deben de dividir en 4 centroides distintos y con una Dimensión de 1000:

n = 1k	n = 10k	n = 100k	n = 1M
0.003s	0.016s	0.123s	3.55s

Y, con la versión paralelizada, gracias al uso de las directrizes de OpenMP:

n = 1k	n = 10k	n = 100k	n = 1M
0.004s	0.013s	0.145s	2.680s

Dónde podemos ver que sí que paralelizamos, pero sólo es apreciable cuando el tamaño de trabajo aumenta considerablemente.

6.2. Floyd-Warshall:

Para poder encabir de la mejor manera la matriz usada, la hemos subdividido en matrices más pequeñas, en concreto de tamaño 8x8. Esta decisión ha sido tomada a raíz del tamaño de línea de caché (de 64 bytes), el tamaño de las variables float (4 bytes) y el tamaño total de la matriz principal (3000x3000), a la que llamaremos "M" para realizar la explicación del desarrollo de este punto más accesible.

- La matriz "M" tiene todas las distancias entre vertices.
- La partimos en bloques más pequeños de 8x8.

De normal, Floyd-Warshall tiene tres bucles (i, j, k) y actualiza M[i,j] = min(M[i,j], M[i,k] + M[k,j]). Esta comprueba caminos que pasan por cada posible intermediario de k.

Para hacer bloques de la matriz "M" desarollamos una función que recibe tres bloques de 8x8 y realiza las operaciones de Floyd-Warshall dentro de dicho bloque (al que llamaremos "B").

- Encontramos los caminos minimos con: B1[i,j] = min(B1[i,j], B2[i,k] + B3[k,j].

Para cada índice de bloque (al que llamaremos "m", y que se conoce como diagonal, que es qualquier bloque situado en la diagonal principal de la matriz de bloques, es decir, con la misma fila y columna de bloque):

- **Recalculamos el bloque diagonal** B[m,m] con la función desarrollada function(B[m,m], B[m,m], B[m,m]). Realizando este paso nos aseguramos de "fijar" todos los caminos que pasan por los vertices dentre de este bloque.
- **Actualizamos bloques de la misma fila y columna,** llamando function(B[i,m], B[i,m], B[m, m] para las columnas y las filas. Así nos aseguramos de propagar el efecto conseguido en el anterior punto.
- **Actualizamos el resto de bloques,** llamando function(B[i, j], B[i, m]], B[m, j].

Estos pasos se ejecutan para cada m (es decir, para cada bloque diagonal), así conseguimos realizar Floyd-Warshall por bloques.

De esta manera conseguimos ir más rápido, ya que nos caben todos los datos con los que estamos trabajando dentro de caché. Este hecho implica tener accesos de lectura y escritura mucho más rápidos que no si tubiesemos que ir hasta memoria principal.

¿Que problemas nos encontramos al hacer sub-bloques?

- Si usamos *float*:

El problema principal que nos encontramos al usar variables de tipo float es que el orden de cálculos dado por la división de matrizes nos abre la posibilidad de que el resultado funcional difiera, aunque el algoritmo, en esencia, continue siendo el mismo.

Dentro del mundo matemático:

$(a + b) + c = a + (b + c)$

En cambio, con floats, nos encontramos con que:

$(a + b) + c \neq a + (b + c)$

Este problema nos surge porque float tiene una precisión limitada de 32 bits, lo que implica que cada operación realiza redondeos. Este suceso se puede dar cuando:

- Hacemos sub-bloques.
- Paralelizamos con OpenMP.
- Cambiamos el orden de bucles.

Por este hecho, nos encontramos con que en la versión secuencial del código checksum nos retorna, aproximadamente 5M, con la versión blocada y paralelizada, 4M.

- Si usamos *int*:

Si se requieren operaciones exactas con un resultado determinista entonces podemos hacer uso de variables de tipo int. Así nos aseguramos de un resultado de checksum constante en el caso de que los cambios en el algoritmo sean correctos y una paralelización segura.

A pesar de que nos presente ciertas ventajas, tiene sus inconvenientes como, por ejemplo, sólo es posible si trabajamos con pesos enteros (lo que equivale a no poder representar pesos reales) y con operaciones que no nos presenten overflow.

Cabe remarcar que sí que es posible realizar la misma tarea que con variables de tipo float haciendo uso de uint64_t, pero despúes de realizar ciertos benchmarks para comprobar su rendimiento, descartamos la posibilidad de realizarlo así debido a que nos representaba una desmejora considerable de rendimiento y tiempo. Se puede encontrar el código usado adjuntado en la carpeta de benchmarks Floyd-Warshall.

Si nos encontramos frente a un problema con distancias enteras, entonces int seria una bona opción a explorar.

¿Como podemos encontrar una tolerancia correcta?

Si decidimos trabajar con floats entonces no podemos esperar una igualdad exacta entre el resultado sin hacer sub-matrices y el algoritmo dónde sí que se hace. Comparar la variable checksum en los distintos casos no es una buena práctica a modo de "sanity-check". En vez de esta práctica podemos obtar por una toleráncia basada en magnitudes relativas y absolutas:

$$| a - b | <= atol + rtol * max(| a |, | b |)$$

Dónde:

- atol = tolerancia absoluta (para así evitar problemas cuando los valores son muy pequeños. Si la diferencia es menor que este valor, lo consideramos 0.
- rtol = tolerancia relativa (proporcional al tamaño de los datos). Nos indica que acepta una diferencia proporcional al tamaño de los numeros comparados.

Ambas tolerancias suelen tomar un valor de 1e – 6 para operaciones de tipo float.

Se debe de tener en cuenta que estos valores dependen de la magnitud de las distancias y del nombre de operaciones (N^3). Si las distancias son grandes como, por ejemplo, de 1e3 o 1e6 augmenta atol o rtol de manera proporcional.

Esta práctica la usamos en este caso porque consideramos que no necesitamos una precisión extrema, así hemos podemos realizar las comprobaciones necesarias para así determinar si la variación del resultado funcional es debido a un error de programación o fruto de la propiedad de la representación float y el orden de cálculo.

En el caso de que se requiera una reproductibilidad bitwise entonces se debe de plantear usar otro tipo de datos (como uint64_t) o no hacer uso de recursos de paralelismo/subdividir la matriz.

¿Qué solución se ha implementado?

- Cálculos en doubles para los temporales, para así reducir errores. Guardado de los datos en float.
- Evitación de sumas con "INF".
- Uso de rtol + atol para la tolerancia de error.

2. RESULTADOS

7.1. K-means:

Nº Threads	kmeans.c	kmeansmod.c
1		
2		

3		
4		
5		
6		
11		
12		

(añadir grafico) (añadir SpeedUp)

7.2. Floyd-Warshall:

El algoritmo de Floyd-Warshall inicial presenta un tiempo, sin la aplicación de ningún flag, de 77.44s. Teniendo en cuenta esta información, se desarolla la siguiente tabla:

Nº Threads	**floyd.c**	**floydmod.c**
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		

BIBLIOGRAFIA Y WEBGRAFIA

https://www.ibm.com/think/topics/k-means-clustering

https://www.geeksforgeeks.org/dsa/floyd-warshall-algorithm-dp-16/

https://www.openmp.org

http://chryswoods.com/beginning_openmp/index.html

https://stackoverflow.com/questions/10850155/whats-the-difference-between-static-and-dynamic-schedule-in-openmp

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

https://things-and-stuff.art/papers/floyd-warshall.pdf

https://jvns.ca/blog/2023/01/13/examples-of-floating-point-problems/

https://aspire.eecs.berkeley.edu/wp/wp-content/uploads/2014/07/Parallel-Reproducible-Summation.pdf

https://olegkarasik.wordpress.com/2024/09/26/implementing-blocked-floyd-warshall-algorithm-for-solving-all-pairs-shortest-path-problem-in-c/

https://www.graphpad.com/guides/prism/latest/statistics/stat_checklist_kmeans_clustering.htm