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

Arquitecturas Avanzadas 2025/26 - UAB

ÍNDICE

1. Introducción 2
2. Marco teórico 3
    2.1. Algoritmo K-means 3
    2.2. Algoritmo Floyd-Warshall 5
    2.3. Modelo de Paralizacion 6
3. Introducción a OpenMP 7
4. State of the Art 8
5. Entorno Experimental 9
    5.1. Maquinaria y Sistema 9
    5.2. Compilación del Programa 9
    5.3. Dimensiones del Problema 11
    5.4. Metodología de Medidas 11
6. Implementacion Secuencial 12
    6.1. Dependencias K-means Secuencial 12
    6.2. Dependencias Floyd-Warshall Secuential 13
    6.3. Resultados Secuenciales 13
7. Desarrollo de las Optimizaciones 14
    7.1. K-means 14
    7.2. Floyd-Warshall 15
        7.2.1. Motivación 15
        7.2.2. Desarrollo 15
        7.2.3. Mejoras en caché 16
8. Resultados 16
    8.1.K-means 16
    8.2.Floyd-Warshall 17
9. Conclusión 18
10. Bibliografia 20
11. Apéndice 21
    11.1. Enlace GitHub 21
    11.2. Tablas 21
    11.3. Apuntes en Bruto 22
        11.3.1. Redactados 22
        11.3.2. Imagenes Usadas 24

1. 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 clusterización 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 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, se adjunta una pequeña discusión sobre sus limitaciones, problemas encontrados a lo largo del trabajo y posibles líneas de trabajo futuro.

2. MARCO TEÓRICO

Este apartado introduce los conceptos fundamentales necesarios para la comprensión de las optimizaciones aplicadas posteriormente. Se describen los algortimos K-means y Floyd-Warshall, así como los principios de paralelización en memoria compartida.

El objectivo no es unicamente describir los algoritmos, sino identificar desde el punto de vista arquitectónico:

- Dónde se concentra el coste computacional
- Qué dependencias de datos existen
- Qué oportunidades de paralelismo y optimización de memoria se pueden realizar

2.1. K-means:

K-means es un algoritmo de aprendizaje no supervisado para la clusterización (repartición de puntos en grupos). 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 el centroide del clúster al que pertenezca 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 el número de centroides y D la dimensión del espacio en el que se trabaja.

El número de iteraciones que debe de realizar el programa depende de la inicialización de centroides y del dataset que se otorgue como input.

Para proceder con la explicación del algoritmo, es necesaria la definición de centroide. Los centroides (los cuales no necesariamente deben coincidir con un punto real del conjunto) son el centro geométrico (mediana) de los puntos asignados a cada clúster.

El algoritmo opera iterativamente:

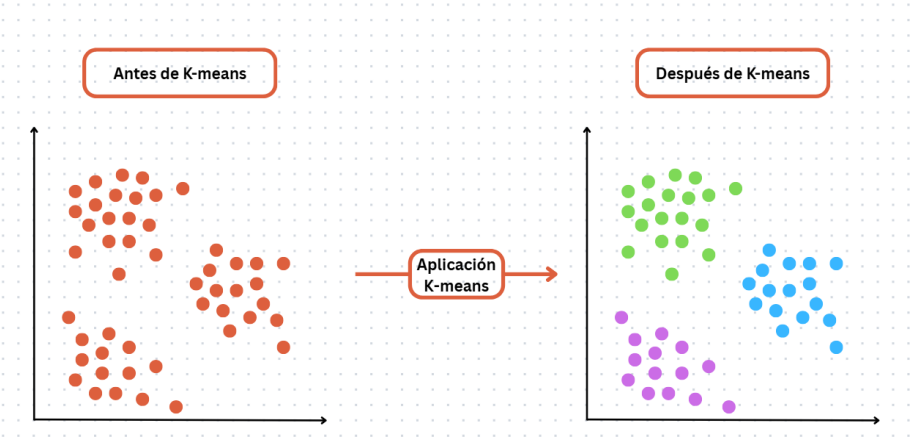
- Asigna cada punto al clúster más próximo
- Recalcula los centroides
- Repite estos pasos hasta la convergencia

Como resultado se obtiene una partición del espacio de datos en grupos lo más compactos con un número similar de puntos por grupo.

El número de clústeres 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 cómo por ejemplo que no tiene un funcionamiento explícitamente optimo si los clústeres no tienen forma específica o densidades muy distintas.

En un campo dónde los puntos no se pueden definir claramente en distintos grupos, este algoritmo no tendrá un funcionamiento debido.

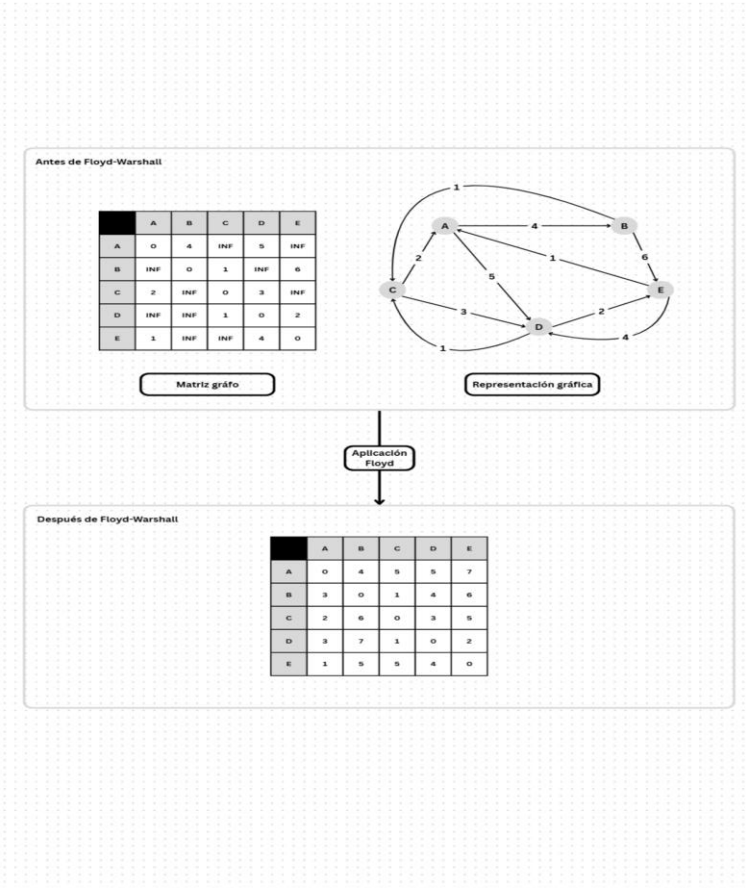


[Fig. 1] Representación gráfica de la aplicación del algoritmo a un conjunto de puntos.

Des de un punto de vista arquitectónico, el interé de este algoritmo reside en que el coste crece principalmente con N, lo que lo convierte en un buen candidato para paralelismo a nivel de datos.

2.2. 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() a nivel temporal y O() a nivel de espacial. La dependencia principal se encuentra en las iteraciones regidas por la variable k . Para calcular dicha iteración 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 al rendimiento del programa. Principalmente se pueden identificar 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 se accede 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 se realiza el orden de los bucles tal que: for(k)

for(i) for(j), entonces el acceso a elementos almacenados en memoria es de manera contigua, así se explota la caché más óptimamente.

Estos primeros puntos son observables sin la necesidad de perfilar el código, a pesar de eso, para poder entender con profundidad la carga de trabajo a lo largo del algoritmo y cómo esta está distribuida para poder focalizar el esfuerzo a optimizar las partes del código más críticas, es necesaria la perfilación del algoritmo. En el apartado de "Entorno Experimental", se desarrolla con más profundidad la perfilación del programa (y, también, la perfilación de K-Means).

Des de un punto de vista arquitectónico, este algoritma resulta interesante porque:

- Trabaja sobre matrizes de gran tamaño que no caben en caché

- Recorre repetidamente toda la matriz en cada iteración de k

- Presenta un patrón de acceso a la memória que pensaliza el rendimiento si no se optimiza la localidad.

2.3. Modelo de paralelización:

El modelo de paralelización considerado en este trabajo es de memoria compartida basada en directivas (en este caso, de OpenMP). En este modelo, todos los threads de un proceso comparten el espacio de direcciones y pueden acceder a los mismos datos. Contamos con dos threads por cada core (en el caso del entorno experimental usado para la realización de este documento, 6 cores con 2 threads para cada núcleo, 12 hilos en total, el procesador usado se especifica en el apartado Entorno Experimental). Cada núcleo físico se asocia a una caché L1 y L2 privada y el nivel de caché L3 es compartido entre los núcleos. Por lo tanto, cada thread lógico perteneciente a un mismo core, comparte L1 y L2 con el otro thread de su mismo núcleo.

Esta jerarquía descrita puede tener implicaciones sobre la localidad de los datos, como, por ejemplo, que dos threads intenten modificar una misma variable a la vez.

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 línea de caché donde está dicha variable, no solo la variable).

- **Overhead de sincronización**: Es necesario minimizar la sincronización dentro de los bucles que "llevan" más tiempo de ejecución 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 a que otros hilos terminen su trabajo, sin realizar ninguna tarea. Haciendo uso de schedule en OpenMP podemos solucionar este problema.

3. Introducción a OpenMP:

Antes de aplicar las técnicas de paralelización a los algoritmos estudiados, se introduce brevemente qué es OpenMP ya que, como se ha comentado previamente, es el modelo de programación paralela usado en este trabajo.

OpenMP proporciona un conjunto de directivas que permiten expresar paralelismo a nivel de bucle y datos en arquitecturas de memoria compartida. Mediante estas directivas se pueden gestionar threads, repartir la carga de trabajo, sincronizar los accesos a memoria y controlar el rendimiento de manera eficiente.

A continuación, se presentan las funcionalidades de OpenMP que se han aplicado para la realización del trabajo. Cabe indicar que, para la realización de la versión final de los algoritmos, adjuntados en el repositorio de GitHub, hay directrices que se han terminado descartando, pero que se han usado durante la implementación de los algoritmos con OpenMP inicialmente y han servido para entender con más profundidad su funcionamiento. Además, la comprensión de las directrices no usadas es útil para futuros trabajos con dichos algoritmos ya que la versión actual no pretende ser la final.

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

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

· **atomic y critical**: Para proteger actualizaciones concretas. Nos aseguramos de que la actualización de una variable no queda interrumpida. 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()**: **omp_get_thread_num()** nos indica en que thread estamos y **omp_get_num_threads()** con cuantos threads estamos ejecutando el programa.

Además, también se deben 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 se debe usar atomic cuando se realizan operaciones simples.

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

4. STATE OF ART

Este apartado resume brevemente el estado actual y las técnicas estándares utilizadas para resolver los problemas abordados por los algoritmos K-means y Floyd-Warshall más allá de las implementaciones básicas que se han realizado a lo largo del proyecto.

K-means:

El algoritmo estándar utilizado hoy en día se basa en la heurística de Lloyd (1982) pero ha evolucionado para solucionar sus principales problemas como, por ejemplo, el coste computacional en grandes volúmenes de datos.

- Inicialización (K-means++):
  La implementación con la que se trabaja en este proyecto inicializa los centroides aleatoriamente, lo que puede llevar ejecuciones menos óptimas. El estándar actual es utilizar K-means++ (Arthur & Vassilvitski, 2007), dónde se eligen los centros iniciales basándose en una probabilidad proporcional a la distancia, garantizando así una convergencia más rápida y precisa.

- Cálculo (Elkan & Hamerly):
  Para evitar calcular la distáncia de cada punto a todos los centroides en cada iteración, se utilizan algoritmos basadoas en la desigualdad triangular. Esto permite evitar cálculos innecesarios si se sabe que un punto está “muy” lejos de un centroide.

- Big Data (Mini-Batch K-means):
  En conjuntos de datos muy elevados (que no caben en memória) se utiliza la variante Mini-Batch, que actualiza los centroides utilizando pequeñas muestras aleatorias de los datos en lugar del conjunto completo. A pesar de que así se pierde precisión, se gana mucha más velocidad.

Floyd-Warshall:

- Grafos dispersos (Johnson & Dijkstra):

Para grafos donde el número de aristas es bajo (es decir, dispersos), no se usa Floyd-Warshall, si no que se suele ejecutar el algoritmo de Dijkstra o Johnson repetidamente para cada nodo.

- Optimización de memoria (Blocked FW):
  La tecnica estándar actual para el uso de Floyd-Warshall es Floyd-Warshall por bloques (Tiling), debido a que su cuello de botella es el acceso a memoria. Se subdivide la matriz en bloques para maximizar la reutilización de datos en la memoria caché (generalmente L1 y L2, pero depende del procesador con el que se trabaje).

- Computación de alto rendimiento (GPU):
  Actualmente, las implementaciones más rápidas de este algoritmo se realizan en GPUs (usando, por ejemplo, CUDA), ya que la independencia de cálculos dentro de cada bloque permite explotar el paralelismo, hecho que el elevado número de núcleos en GPUs le es útil.

5. ENTORNO EXPERIMENTAL

5.1. Maquinaria y sistema:

Processador	12th Gen Intel(R) Core(TM) i5-12400	Freq. base: 2500 MHz	6 cores, 2 hilos por core
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 se ha usado el comando “lscpu” y un script de bash adjuntado (en la carpeta de trabajo/repositorio de GitHub) llamado “config.sh”.

Para ejecutar el script de bash se deben de dar permisos junto el comando “chmod +x config.sh” para permitir la ejecución del archivo.

5.2. Compilación del programa:

Se ha 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 se han usado los siguientes flags de compilación:

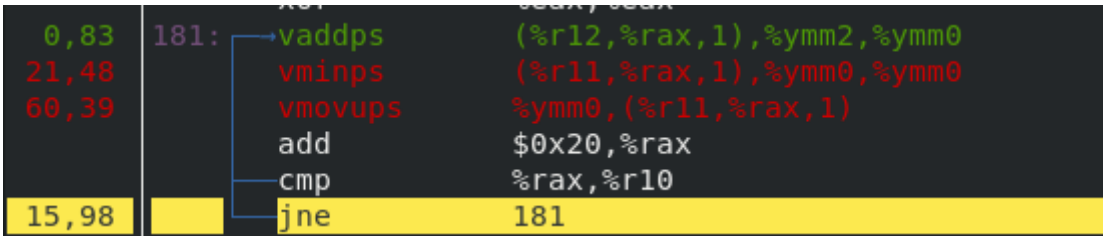
-O3: El flag –O3 del compilador gcc activa todas las optimizaciones especificadas por –O2 y otros flags como, por ejemplo, -floop-unroll. Se ha decidido usar este flag en vez de –O2 o -Ofast porque el resultado funcional es el mismo y, además, tiene una mejoría (muy pequeña) a nivel temporal. Pero se 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 de procesador. 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 ha sido usado para la realización de pruebas despues de la paralelización el programa. Para la versión secuencial se ha usado solo –O3 y -march=native (a parte del flag –o para determinar el nombre de salida del binario).

Todas las opciones de compilación se encuentran en el Makefile

Para poder conseguir el ensamblador de los códigos se ha hecho uso de los comandos de perfilado de programa “perf stat”, “perf record” y “perf report”. A continuación, se adjunta el ensamblador de Floyd-Warshall, usado para comprender que partes del código implican más trabajo, comprobar que el compilador vectoriza debidamente las operaciones que se realizan y asegurar que la operación “min” se traduce con “min” en vez de un seguido de operaciones de comparación, restas y saltos:



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

El perfilado de los códigos es importante para tener en cuenta el tiempo necesario para las operaciones que se realizan en cada algoritmo además de entender que operaciones requieren un mayor esfuerzo y dónde se encuentran los cuellos de botella. Esto permite priorizar optimizaciones efectivas y evitar gastos de tiempo en optimizaciones menores con un impacto muy pequeño.

5.3. Dimensiones del problema:

A continuación, se presentan los tamaños de problema con que se ha trabajado durante las distintas pruebas:

K-means:

Número de puntos N: 1K, 10K, 100K, 1M

Dimensionalidad D: 1000

Numero de centroides C: 4

Número de puntos	1k	10k	100k	1M
Dimensionalidad D	1k	1k	1k	1k
Número de centroides C	4	4	4	4

Para trabajos futuros se pretende modificar tanto la dimensionalidad como el número de centroides con los que se ha trabajado.

Floyd-Warshall:

Tamaño de la matriz N: 3k

Densidad simulada de la matriz: En el codigo inicial propuesto es un 65% densa, 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 pero para trabajos futuros se pretende variar el tamaño.

Este tamaño de matriz es suficientemente grande para que no quepa en caché L2 y que el acceso a memoria sea el principal factor limitante del rendimiento.

5.4. Metodología de medidas:

Para hacer la recolección de estadísticas, comprobando que el código sigue teniendo el mismo resultado funcional se han seguido las siguientes indicaciones:

· Control del número de threads (variando el número de 1 a 12).

· Uso de warm-up (ejecución de dos iteraciones de calentamiento antes de recojer los datos para estabilizar las cachés y políticas del sistema).

· Repeticiones: Para cada configuración se han realizado 5 repeticiones.

· Uso de perf stat, perf record/report para el analisis de regiones críticas del programa además de la recolección de datos (tiempo de ejecución del programa después de cada cambio mayor y resultado funcional numérico).

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

Para asegurar que todos estos puntos se realizan de manera regular para todas las ejecuciones y para facilitarnos la tarea de análisis se ha usado un script de ejecución (adjuntado adjuntado bajo el nombre de "ejec.sh") el cual hace uso del comando hyperfine. Dicho comando requiere una cierta programación en el lenguaje Rust, ya que la herramienta está escrita en este lenguaje. Si no se quiere usar el comando mencionado, se puede realizar un script en bash que tenga una ejecución similar a "hyperfine".

Cómo trabajo futuro se realizará la implementación de este script, para facilitar el acceso al trabajo realizado.

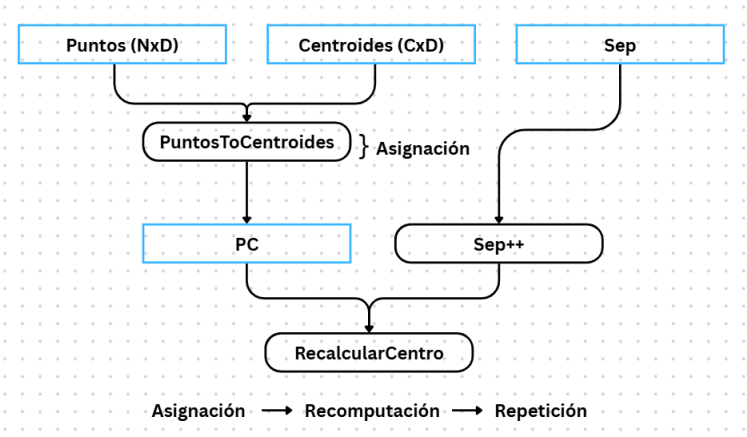
6. IMPLEMENTACIÓN SEQÜENCIAL

A continuación, se presenta la explicación de la implementación de los algoritmos explicados secuencialmente. Este apartado es necesario para entender dónde están realmente las dependencias y los cuellos de botella de cada programa.

Teniendo en cuenta que:

[Fig. 4] Leyenda grafos de dependéncia

5.1. Dependencias K-means secuencial:

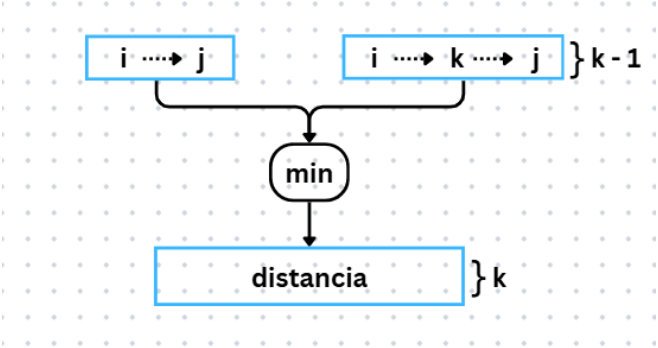


[Fig. 5] Dependencias K-means

Dónde:

- Puntos son los puntos con los que trabajamos para calcular los distintos clústeres.
- Centroides son los respectivos centros para cada grupo.
- Sep es el contador que nos indica los puntos que tenemos por grupo.
- PC es una estructura que nos guarda que puntos van a qué clúster.

5.2. Dependencias Floyd-Warshall secuencial:



[Fig. 6] Dependéncias Floyd-Warshall

Dónde:

- i -> j es el peso del camino des de i hasta j.
- i -> k -> j es el peso del camino des de i hasta j pasando por k.
- distancia es el peso minimo de los dos caminos anteriores.

6.3. Resultados sequenciales:

A continuación, se presentan los resultados de aplicar distintos flags de compilación, para así mostrar la decisión de aplicar o no aplicar dichos flags.

K-means:

Flags compilacion	Escalar n = 1M	Paralelizado n = 1M
-O3	2.524s	~
-O3 -march=native	2.568s	~

Dónde Kmeansmod.c hace referencia al código de K-means con el uso de OpenMP, haciendo uso de una k = 1M, D = 1k, C = 4.

En este caso, podemos observar que practicamente no hay escalado entre la versión secuencial y la versión paralelizada con OpenMP. Esto se debe a que para cada punto de K-means realizamos las siguentes tareas:

1. Leer punto
2. Leer todos los centroides
3. Calcular distancias
4. Escribir asignación
5. Realizar reducciones para recomputar centroides

Este proceso requiere muchas lecturas en memória y muy pocos cálculos. Añadir más hilos no añade más ancho de banda de memoria.

Floyd-Warshall:

Con un tiempo incial, sin flags de compilación de aproximadamente 77 segundos:

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.

7. DESARROLLO DE LAS OPTIMIZACIONES

Para ambos algoritmos se ha hecho uso de directrices de OpenMP descritas anteriormente para distribuir el trabajo entre los distintos cores y threads, además de tener en cuenta el uso de las cachés privadas de cada núcleo.

A continuación, se describe la lógica seguida para cada algoritmo:

7.1. 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, haciendo uso de las directrizes de OpenMP:

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

Dónde se puede ver una mejora de tiempo, pero sólo es apreciable cuando el tamaño de trabajo aumenta considerablemente.

7.2. Floyd-Warshall:

7.2.1. Motivación:

En la implementación secuencial, para cada valor de k se recorre toda la matriz de tamaño NxN. Para una matriz de 3000x3000 floats, esto implica trabajar sobre, aproximadamente, 36MB de datos repetidamente. Esta mida excede la capacidad de las cachés L1 y L2, lo que genera una gran cantidad de fallos en caché.

Por lo tanto, la pregunta principal no debe de ser solamente "¿Como podemos paralelizar Floyd-Warshall?" sino que también se debe de plantear como reducir accesos a memória principal.

Ambas dudas se pueden resolver con el bloqueo de matrizes.

Haciendo subbloqueo de matrizes conseguimos que los bloques quepan en caché y, además, nos abren la posibilidad de parlalelizar ciertas partes de k, hecho que no sería posible sin hacer bloques.

7.2.2. Desarrollo:

El tamaño de cada bloque fruto de la subdivisión de la matriz se ha decidido que sea de 8x8. El tamaño decidido viene dado por el tamaño de línea de caché (64 bytes), el tamaño de las variables float (4 bytes) y el tamaño total de la matriz principal (3000x3000), a la que se asocia al nombre “M”.

La matriz termina dividida en N / b bloques.

Para cada bloque diagonal k ([k,k]) se reakuzab tres fases distintas:

Fase 1 - *floyd_warshall_in_place(B[k][k], B[k][k], B[k][k])*:

Este paso fija todos los caminos mínimos que pasan solamente por los vértices dentro de este bloque.

Esta fase no se puede paralelizar, ya que, si se paralelizase, no se estarian cubriendo las dependencias necesarias cómo para obtener el resultado funcional esperado.

Fase 2 - Actualización de la fila y columna del bloque:

Se propagan los cambios en los caminos que pasan por el bloque diagonal hacia el resto de los bloques de la misma fila y columna. Estos bloques son independientes entre si y, por consiguiente, esta parte del programa es paralelizable parcialmente.

Fase 3 - Actualización del resto de bloques:

En esta fase, cada bloque [i,j] depende solamente de los bloques [i,k] y [k,j], los cuales ya han sido actualizados en la fase anterior. Todos los de esta fase son paralelizables.

7.2.3. ¿Hay una mejora en caché?

La función floyd_warshall_in_place trabaja exclusivamente en matrices de tamaño pequeño. Durante el el triple bucle interno, los datos A, B y C se reutilizan muchas veces y estos datos se quedan en caché. Así se consiguen reducir los accesos a memoria principal.

Este cambio en el patrón de acceso debería de mejorar mucho más el rendimiento que no la paralelización del algoritmo por si sola.

8. RESULTADOS

8.1. K-means:

Teniendo en cuenta los tiempos mostrados en el punto 7.1. del apartado “Desarrollo de Optimizaciones”:

n = 1k	0.003 / 0.004 = 0.75	-25%

n = 10k	0.016 / 0.013 = 1.23	+23%
n = 100k	0.123 / 0.145 = 0.85	-15%
n = 1M	3.55 / 2.680 = 1.32	+32%

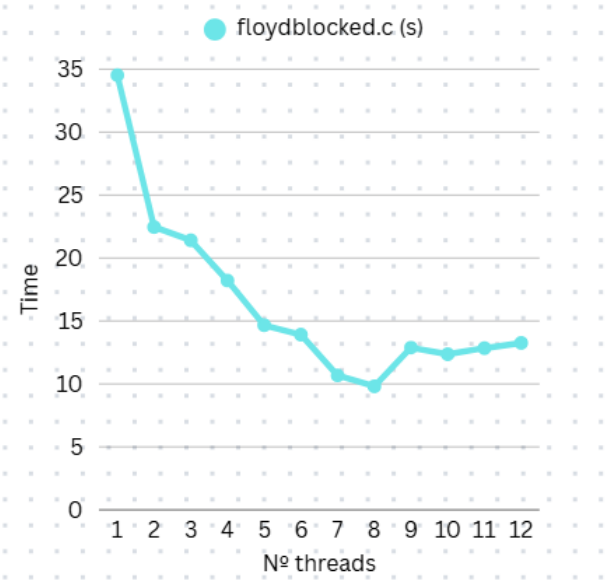
En tamaños pequeños, la versión paralela tiene sobrecostes (overhead) y puede ir más lenta, para tamaños mayores (en el caso de n = 1M) se consigue una mejora, pero solo de ~1.3x.

Estas conclusiones están sujetas a la mejora del algoritmo próximamente y, por lo tanto, no son definitivas.

8.2. Floyd-Warshall:

El algoritmo de Floyd-Warshall inicial presentaba un tiempo, sin la aplicación de ningún flag, de 77.44s.

A continuación, se adjunta un grafico representando la variación de tiempo de la versión de Floyd-Warshall con la matriz subblocada, dependiendo del numero de threads con el que se trabaje:



[Graph 1] Relación tiempo dependiendo del nº de hilos

Teniendo en cuenta que el algoritmo de Floyd-Warshall inicial presentaba un tiempo, sin la aplicación de ningún flag durante la compilación de este, de 77.44s, se presentan los siguientes valores de SpeedUp para la versión de Floyd-Warshall sub-blocada y paralelizada con OpenMP:

Threads	SpeedUp (77 / T)

1	2.23x
2	3.43x
3	3.60x
4	4.23x
5	5.25x
6	5.53x
7	7.22x
8	7.85x
9	5.98x
10	6.23x
11	6.00x
12	5.81x

El mejor SpeedUp conseguido es con 8 threads (~7.85x). A partir de 8 hilos, el rendimiento empeora, este patrón suele suceder en códigos memory-bound con OpenMP.

Se debe a una saturación en memoria, overhead de sincronización y, además, podria ser consecuencia de un false sharing, pero no es una conclusión definitiva, debido a que falta experimentación con este código.

9. CONCLUSIÓN

El trabajo realizado ha permitido analizar, implementar y evaluar estrategias de optimización y paralelizacion mediante OpenMP en dos algoritmos fundamentales: K-means y Floyd-Warshall. A través de la experimentación y el perfilado de código se han extraído conclusiones sobre la importancia de adaptar el software a la arquitectura subyacente.

En primer lugar, se ha podido ver que K-means se comporta como un algoritmo limitado por la memoria. El proceso requiere muchas lecturas de datos para realizar comparaciones con los centroides, pero relativamente pocos cálculos complejos.

Debido a esto, la paralelización apenas ofreció mejoras en tamaños de problemas pequeños.

En cambio, con el algoritmo de Floyd-Warshall, la implementación secuencial sufría una falta de localidad espacial y temporal, provocando un exceso de fallos de caché al recorrer repetidamente matrices grandes.

Finalmente, haciendo uso de tiling y paralelismo en OpenMP se lograd reducir el tiempo de ejecución de unos 77 segundos a 9.8 segundos, alcanzando un SpeedUp máximo de 7.85x.

A partir de 8 hilos se observa que el rendimiento decae, lo que sugiere una saturación del ancho de banda o un overhead de sincronización que supera las ganancias del paralelismo. Estas conclusiones no son definitivas, debido a que este proyecto aún está en proceso de finalizarse.

Des de un punto de vista metodológico se ha dado importancia a la compilación con los flags correspondientes y al perfilado debido de los códigos.

En definitiva, este proyecto ilustra que la paralelización efectiva no consiste únicamente en añadir directivas de OpenMP, requiere un análisis más profundo para comprender como se organiza la jerarquía de memoria, la gestión de dependencias y la computación de cada uno de los algoritmos.

Floyd-Warshall se beneficia del paralelismo de gran grueso y la localidad de datos, en cambio, K-means requiere un enfoque de estrategias distinto para poder superar sus limitaciones de ancho de banda.

10. BIBLIOGRAFIA Y WEBGRAFIA

IBM. (s. d.). *K-means clustering*. IBM Think. https://www.ibm.com/think/topics/k-means-clustering

GeeksforGeeks. (s. d.). *Floyd Warshall algorithm (DP-16)*. https://www.geeksforgeeks.org/dsa/floyd-warshall-algorithm-dp-16/

OpenMP Architecture Review Board. (s. d.). *OpenMP API specification*. https://www.openmp.org

Woods, C. (s. d.). *Beginning OpenMP*. http://chryswoods.com/beginning_openmp/index.html

Stack Overflow. (2012). *What's the difference between static and dynamic schedule in OpenMP?*

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

GNU Project. (s. d.). *GCC optimization options*. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Things and Stuff. (s. d.). *The Floyd–Warshall algorithm*. https://things-and-stuff.art/papers/floyd-warshall.pdf

Niederman, J. (2023). *Examples of floating point problems*. https://jvns.ca/blog/2023/01/13/examples-of-floating-point-problems/

Demmel, J., Nguyen, H. D., & Ahrens, J. (2014). *Parallel reproducible summation*. University of California, Berkeley. https://aspire.eecs.berkeley.edu/wp/wp-content/uploads/2014/07/Parallel-Reproducible-Summation.pdf

Karasik, O. (2024). *Implementing blocked Floyd–Warshall algorithm for solving all-pairs shortest path problem in C*. https://olegkarasik.wordpress.com/2024/09/26/implementing-blocked-floyd-warshall-algorithm-for-solving-all-pairs-shortest-path-problem-in-c/

GraphPad Software. (s. d.). *K-means clustering: Statistical checklist*. https://www.graphpad.com/guides/prism/latest/statistics/stat_checklist_kmeans_clustering.htm

Intel. (s. d.). *Intel® Core™ i5-12400 processor specifications*. https://www.intel.com/content/www/us/en/products/sku/134586/intel-core-i512400-processor-18m-cache-up-to-4-40-ghz/specifications.html

11. ANNEXO:

11.1. Enlace GitHub:

Para tener acceso a todos los codigos y scripts mencionados a lo largo de este informe, se puede acceder al siguiente repositorio de GitHub, dónde se encuentran los algoritmos.

https://github.com/clauudus/OpenMP-Unsupervised_ML

11.2. Tablas con tiempos:

11.2.1. K-means tabla paralelizada:

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

Esta tabla se ha dejado en blanco ya que aún se está trabajando con el algoritmo de K-means.

11.2.2. Floyd-Warshall tabla paralelizada:

Nº Threads	**floydblocked.c**

1	34.518 s
2	22.459s
3	21.406s
4	18.209s
5	14.663s
6	13.914s
7	10.671s
8	9.807s
9	12.877s
10	12.358s
11	12.839s
12	13.255s

Disclimer: Los datos no han sido obtenidos des de la misma maquina con la que se trabajó en un principio. Temporalmente estan escritos en el informe para tener una orientación sobre el funcionamiento de los algoritmos. Proximamente se repetirá el experimento en una maquina estable nuevamente (ya que estos datos son fruto de una virtualización de la maquina inicial), cambiando así la información del entrono experimental y los resultados obtenidos.

11.3. Apuntes en Bruto:

11.3.1. Redactados (apuntes en bruto sobre comportamientos no esperados):

Como el cálculo de Dik no siempre es secuencial, pueden surgir ciertos problemas debido a las operaciones con float. A continuación, se adjunta una explicación tratando dicho tema. Más adelante, se descartó la posibilidad de que el cálculo de Dik ya que, haciendo pequeños cambios en el código que no mostraba el resultado funcional debido, se consiguió el resultado (relativamente) esperado.

Cabe remarcar que el resultado fue relativamente esperado porque, a pesar de conseguir la implementación del código, no se consiguió una mejora en el tiempo.

Este trabajo sigue estando en funcionamiento, para así seguir explorando las dificultades presentadas y poder encontrar, finalmente, una optimización que, tal y como indica su nombre, sea optima.

Los apartados de Annexo estan aún en bruto, son notas que se tomaron a lo largo del trabajo para uso personal, pero se consideran útiles para la realización de otras tareas no relacionadas con K-means o Floyd-Warshall.

¿Qué 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 aritmético:

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

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 tolerância 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 (). 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.

11.3.2. Imágenes usadas para la comprensión de los problemas:

N = 3000

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

        12.011,06 msec task-clock:u              #    1,000 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
              661      page-faults:u             #   55,033 /sec
   52.537.201.736      cpu_core/cycles:u/        #    4,374 G/sec
  217.370.194.306      cpu_core/instructions:u/  #   18,098 G/sec
   54.297.711.403      cpu_core/branches:u/      #    4,521 G/sec
      220.524.419      cpu_core/branch-misses:u/ #   18,360 M/sec

     12,012834953 seconds time elapsed

     11,994542000 seconds user
      0,001997000 seconds sys
```

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

        12.362,99 msec task-clock:u              #    1,000 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
              660      page-faults:u             #   53,385 /sec
   52.863.708.821      cpu_core/cycles:u/        #    4,276 G/sec
  217.385.932.661      cpu_core/instructions:u/  #   17,584 G/sec
   54.299.961.766      cpu_core/branches:u/      #    4,392 G/sec
      220.294.558      cpu_core/branch-misses:u/ #   17,819 M/sec

     12,364537757 seconds time elapsed

     12,345776000 seconds user
      0,000998000 seconds sys
```

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

         4.666,54 msec task-clock:u              #    1,000 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
              660      page-faults:u             #  141,433 /sec
   19.801.407.152      cpu_core/cycles:u/        #    4,243 G/sec
   25.209.385.035      cpu_core/instructions:u/  #    5,402 G/sec
    7.200.308.187      cpu_core/branches:u/      #    1,543 G/sec
       88.670.315      cpu_core/branch-misses:u/ #   19,001 M/sec

      4,667657143 seconds time elapsed

      4,655241000 seconds user
      0,002995000 seconds sys
```

N = 500

```
CheckSum: 4583051

 Performance counter stats for 'floydmod':

            28,14 msec task-clock:u              #    0,980 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
              291      page-faults:u             #   10,340 K/sec
      111.117.881      cpu_core/cycles:u/        #    3,948 G/sec
      154.626.904      cpu_core/instructions:u/  #    5,494 G/sec
       42.884.683      cpu_core/branches:u/      #    1,524 G/sec
        1.541.509      cpu_core/branch-misses:u/ #   54,773 M/sec

      0,028715677 seconds time elapsed

      0,027512000 seconds user
      0,000982000 seconds sys
```

N = 1000

```
[aa-23@aolin13 rendimiento]$ gcc -O3 -march=native floydmod.c -o floydmod
[aa-23@aolin13 rendimiento]$ perf stat floydmod

CheckSum: 8580678

 Performance counter stats for 'floydmod':

           165,24 msec task-clock:u              #    0,996 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
              515      page-faults:u             #    3,117 K/sec
      691.004.442      cpu_core/cycles:u/        #    4,182 G/sec
    1.043.884.801      cpu_core/instructions:u/  #    6,317 G/sec
      296.402.373      cpu_core/branches:u/      #    1,794 G/sec
        7.501.381      cpu_core/branch-misses:u/ #   45,396 M/sec

      0,165843205 seconds time elapsed

      0,161341000 seconds user
      0,003980000 seconds sys
```

N = 2000

```
CheckSum: 6466361

 Performance counter stats for 'floydmod':

         1.100,04 msec task-clock:u              #    0,999 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
              891      page-faults:u             #  809,967 /sec
    4.678.744.921      cpu_core/cycles:u/        #    4,253 G/sec
    7.700.247.012      cpu_core/instructions:u/  #    7,000 G/sec
    2.198.153.157      cpu_core/branches:u/      #    1,998 G/sec
       37.903.942      cpu_core/branch-misses:u/ #   34,457 M/sec

      1,100664252 seconds time elapsed

      1,097944000 seconds user
      0,000998000 seconds sys
```

Openmp N = 3000:

1

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

         4.639,40 msec task-clock:u              #    1,000 CPUs utilized
               0      context-switches:u        #    0,000 /sec
               0      cpu-migrations:u          #    0,000 /sec
             681      page-faults:u             #  146,786 /sec
   19.806.050.846      cpu_core/cycles:u/        #    4,269 G/sec
   25.185.169.572      cpu_core/instructions:u/  #    5,429 G/sec
    7.200.774.136      cpu_core/branches:u/      #    1,552 G/sec
       89.371.962      cpu_core/branch-misses:u/ #   19,264 M/sec

      4,640391663 seconds time elapsed

      4,625812000 seconds user
      0,005881000 seconds sys
```

2

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

         5.383,64 msec task-clock:u              #    1,957 CPUs utilized
               0      context-switches:u        #    0,000 /sec
               0      cpu-migrations:u          #    0,000 /sec
             683      page-faults:u             #  126,866 /sec
   22.465.253.756      cpu_core/cycles:u/        #    4,173 G/sec
   25.191.794.186      cpu_core/instructions:u/  #    4,679 G/sec
    7.202.643.168      cpu_core/branches:u/      #    1,338 G/sec
       91.123.334      cpu_core/branch-misses:u/ #   16,926 M/sec

      2,750778644 seconds time elapsed

      5,365259000 seconds user
      0,006990000 seconds sys
```

3

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

         6.363,87 msec task-clock:u              #    2,893 CPUs utilized
               0      context-switches:u        #    0,000 /sec
               0      cpu-migrations:u          #    0,000 /sec
             685      page-faults:u             #  107,639 /sec
   26.154.220.864      cpu_core/cycles:u/        #    4,110 G/sec
   25.198.032.601      cpu_core/instructions:u/  #    3,960 G/sec
    7.204.321.071      cpu_core/branches:u/      #    1,132 G/sec
       92.248.615      cpu_core/branch-misses:u/ #   14,496 M/sec

      2,199938231 seconds time elapsed

      6,341348000 seconds user
      0,006972000 seconds sys
```

4

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

          7.532,36 msec task-clock:u              #    3,819 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
               689      page-faults:u             #   91,472 /sec
    29.930.089.056      cpu_core/cycles:u/        #    3,974 G/sec
    25.208.377.404      cpu_core/instructions:u/  #    3,347 G/sec
     7.207.172.514      cpu_core/branches:u/      #  956,828 M/sec
        92.933.154      cpu_core/branch-misses:u/ #   12,338 M/sec

       1,972295465 seconds time elapsed

       7,498816000 seconds user
       0,008901000 seconds sys
```

5

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

          9.234,35 msec task-clock:u              #    4,760 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
               690      page-faults:u             #   74,721 /sec
    36.638.799.898      cpu_core/cycles:u/        #    3,968 G/sec
    25.241.694.732      cpu_core/instructions:u/  #    2,733 G/sec
     7.216.587.389      cpu_core/branches:u/      #  781,494 M/sec
        93.669.132      cpu_core/branch-misses:u/ #   10,144 M/sec

       1,939912569 seconds time elapsed

       9,191684000 seconds user
       0,007992000 seconds sys
```

6

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

          9.894,40 msec task-clock:u              #    5,663 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
               693      page-faults:u             #   70,040 /sec
    39.223.712.115      cpu_core/cycles:u/        #    3,964 G/sec
    25.259.301.980      cpu_core/instructions:u/  #    2,553 G/sec
     7.221.513.300      cpu_core/branches:u/      #  729,858 M/sec
        94.183.643      cpu_core/branch-misses:u/ #    9,519 M/sec

       1,747076398 seconds time elapsed

       9,840723000 seconds user
       0,009921000 seconds sys
```

7

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

         12.247,98 msec task-clock:u              #    6,617 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
               694      page-faults:u             #   56,662 /sec
    48.586.715.268      cpu_core/cycles:u/        #    3,967 G/sec
    25.549.185.447      cpu_core/instructions:u/  #    2,086 G/sec
     7.304.235.778      cpu_core/branches:u/      #  596,362 M/sec
        94.572.311      cpu_core/branch-misses:u/ #    7,721 M/sec

       1,850899690 seconds time elapsed

      12,195719000 seconds user
       0,002929000 seconds sys
```

8

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

        13.161,31 msec task-clock:u              #    7,523 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
              695      page-faults:u             #   52,806 /sec
   52.168.053.835      cpu_core/cycles:u/        #    3,964 G/sec
   25.495.021.540      cpu_core/instructions:u/  #    1,937 G/sec
    7.288.653.731      cpu_core/branches:u/      #  553,794 M/sec
       94.860.346      cpu_core/branch-misses:u/ #    7,208 M/sec

      1,749424165 seconds time elapsed

     13,087036000 seconds user
      0,016920000 seconds sys
```

9

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

        13.910,12 msec task-clock:u              #    8,435 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
              697      page-faults:u             #   50,107 /sec
   55.143.917.111      cpu_core/cycles:u/        #    3,964 G/sec
   25.476.426.779      cpu_core/instructions:u/  #    1,832 G/sec
    7.283.237.618      cpu_core/branches:u/      #  523,593 M/sec
       95.188.818      cpu_core/branch-misses:u/ #    6,843 M/sec

      1,649186968 seconds time elapsed

     13,837635000 seconds user
      0,009928000 seconds sys
```

10

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

        15.643,26 msec task-clock:u              #    9,374 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
              702      page-faults:u             #   44,876 /sec
   61.941.647.781      cpu_core/cycles:u/        #    3,960 G/sec
   25.413.731.500      cpu_core/instructions:u/  #    1,625 G/sec
    7.265.219.673      cpu_core/branches:u/      #  464,431 M/sec
       95.654.044      cpu_core/branch-misses:u/ #    6,115 M/sec

      1,668848322 seconds time elapsed

     15,541273000 seconds user
      0,018739000 seconds sys
```

11

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

        16.239,77 msec task-clock:u              #   10,261 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
              703      page-faults:u             #   43,289 /sec
   64.296.830.483      cpu_core/cycles:u/        #    3,959 G/sec
   25.354.150.915      cpu_core/instructions:u/  #    1,561 G/sec
    7.248.094.791      cpu_core/branches:u/      #  446,318 M/sec
       95.883.782      cpu_core/branch-misses:u/ #    5,904 M/sec

      1,582732012 seconds time elapsed

     16,141227000 seconds user
      0,010748000 seconds sys
```

12

```
CheckSum: 5271068

 Performance counter stats for 'floydmod':

        19.412,37 msec task-clock:u              #   11,158 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
              707      page-faults:u             #   36,420 /sec
   76.824.271.651      cpu_core/cycles:u/        #    3,957 G/sec
   25.556.634.674      cpu_core/instructions:u/  #    1,317 G/sec
    7.305.840.533      cpu_core/branches:u/      #  376,350 M/sec
       96.071.285      cpu_core/branch-misses:u/ #    4,949 M/sec

      1,739809308 seconds time elapsed

     19,293813000 seconds user
      0,017807000 seconds sys
```

K-MEANS

-O2

```
K-means, 1000 Puntos con 3 Dimensiones y 4 Centroides
El espacio de puntos es de Dimensión: 1000
Numero de iteraciones necesarias: 19

 Performance counter stats for 'k-means':

             0,45 msec task-clock:u              #    0,428 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
               54      page-faults:u             #  119,532 K/sec
        1.133.517      cpu_core/cycles:u/        #    2,509 G/sec
        4.846.721      cpu_core/instructions:u/  #   10,729 G/sec
          697.360      cpu_core/branches:u/      #    1,544 G/sec
            4.615      cpu_core/branch-misses:u/ #   10,216 M/sec

      0,001054650 seconds time elapsed

      0,000815000 seconds user
      0,000000000 seconds sys
```

-O3

```
K-means, 1000 Puntos con 3 Dimensiones y 4 Centroides
El espacio de puntos es de Dimensión: 1000
Numero de iteraciones necesarias: 19

 Performance counter stats for 'k-means':

             0,45 msec task-clock:u              #    0,347 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
               56      page-faults:u             #  125,503 K/sec
        1.098.790      cpu_core/cycles:u/        #    2,463 G/sec
        4.754.492      cpu_core/instructions:u/  #   10,655 G/sec
          669.155      cpu_core/branches:u/      #    1,500 G/sec
            4.591      cpu_core/branch-misses:u/ #   10,289 M/sec

      0,001284243 seconds time elapsed

      0,000000000 seconds user
      0,000768000 seconds sys
```

```
K-means, 10000 Puntos con 3 Dimensiones y 4 Centroides
El espacio de puntos es de Dimensión: 1000
Numero de iteraciones necesarias: 30

 Performance counter stats for 'k-means':

             3,22 msec task-clock:u              #    0,822 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
               92      page-faults:u             #   28,576 K/sec
       12.004.175      cpu_core/cycles:u/        #    3,729 G/sec
       70.229.590      cpu_core/instructions:u/  #   21,814 G/sec
        9.399.993      cpu_core/branches:u/      #    2,920 G/sec
            5.795      cpu_core/branch-misses:u/ #    1,800 M/sec

      0,003918067 seconds time elapsed

      0,002762000 seconds user
      0,000818000 seconds sys
```

```
K-means, 100000 Puntos con 3 Dimensiones y 4 Centroides
El espacio de puntos es de Dimensión: 1000
Numero de iteraciones necesarias: 47

 Performance counter stats for 'k-means':

            45,34 msec task-clock:u              #    0,988 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
              463      page-faults:u             #   10,211 K/sec
      183.217.526      cpu_core/cycles:u/        #    4,041 G/sec
    1.085.853.007      cpu_core/instructions:u/  #   23,947 G/sec
      143.603.501      cpu_core/branches:u/      #    3,167 G/sec
           15.118      cpu_core/branch-misses:u/ #  333,405 K/sec

      0,045908102 seconds time elapsed

      0,043664000 seconds user
      0,001980000 seconds sys
```

```
K-means, 100000 Puntos con 50 Dimensiones y 64 Centroides
El espacio de puntos es de Dimensión: 1000
Numero de iteraciones necesarias: 437

 Performance counter stats for 'k-means':

        69.533,05 msec task-clock:u              #    1,000 CPUs utilized
                0      context-switches:u        #    0,000 /sec
                0      cpu-migrations:u          #    0,000 /sec
              862      page-faults:u             #   12,397 /sec
  298.479.684.456      cpu_core/cycles:u/        #    4,293 G/sec
1.040.886.547.155      cpu_core/instructions:u/  #   14,970 G/sec
  148.704.838.312      cpu_core/branches:u/      #    2,139 G/sec
       46.539.543      cpu_core/branch-misses:u/ #  669,315 K/sec

     69,538826096 seconds time elapsed

     69,481753000 seconds user
      0,004982000 seconds sys
```

```
K-means, 100000 Puntos con 50 Dimensiones y 64 Centroides
El espacio de puntos es de Dimensión: 1000
Numero de iteraciones necesarias: 437

 Performance counter stats for 'k-means':

         51.086,70 msec task-clock:u              #    1,000 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
               363      page-faults:u             #    7,106 /sec
   223.690.151.160      cpu_core/cycles:u/        #    4,379 G/sec
 1.187.942.555.684      cpu_core/instructions:u/  #   23,253 G/sec
   148.814.348.995      cpu_core/branches:u/      #    2,913 G/sec
        61.640.804      cpu_core/branch-misses:u/ #    1,207 M/sec

      51,091055070 seconds time elapsed

      51,053633000 seconds user
       0,004993000 seconds sys
```

```
K-means, 100000 Puntos con 50 Dimensiones y 64 Centroides
El espacio de puntos es de Dimensión: 1000
Numero de iteraciones necesarias: 437

 Performance counter stats for 'k-means':

         68.049,91 msec task-clock:u              #    1,000 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
               879      page-faults:u             #   12,917 /sec
   297.877.647.281      cpu_core/cycles:u/        #    4,377 G/sec
 1.040.886.545.712      cpu_core/instructions:u/  #   15,296 G/sec
   148.704.836.869      cpu_core/branches:u/      #    2,185 G/sec
        44.683.131      cpu_core/branch-misses:u/ #  656,623 K/sec

      68,055006388 seconds time elapsed

      68,009287000 seconds user
       0,002998000 seconds sys
```

```
K-means, 1000 Puntos con 3 Dimensiones y 4 Centroides
El espacio de puntos es de Dimensión: 1000
Numero de iteraciones necesarias: 19

real    0m0,006s
user    0m0,002s
sys     0m0,003s
[clau@clauvm k-means]$ nano openmp.c
[clau@clauvm k-means]$ gcc openmp.c -O3 -march=native -o openmp
[clau@clauvm k-means]$ time ./openmp
K-means, 1000 Puntos con 3 Dimensiones y 4 Centroides
El espacio de puntos es de Dimensión: 1000
Numero de iteraciones necesarias: 19

real    0m0,005s
user    0m0,004s
sys     0m0,002s
```

K-means escalar y openMP con 4 centroides, 1000 puntos y 3 dimensiones.