
# CIENCIA DE LA COMPUTACIÓN

## ESTRUCTURAS DE DATOS AVANZADA

---

# INFORME: K-MEANS

---

Carlos Eduardo Atencio Torres

April 2023

Autor:
Iris Rocio Curo Quispe

Semestre VI
2023-1

# Laboratorio - Comparación entre a implementación secuencial y paralela de K-means

Iris Rocio Curo Quispe
iris.curo@ucsp.edu.pe

***Abstract*—Este informe presenta un análisis de rendimiento de las implementaciones paralela y secuencial del algoritmo K-Means en el contexto de la agrupación de datos de punto flotante en 2D. Además, se desarrolló una función de visualización para mostrar los puntos utilizando un archivo HTML. El enfoque del estudio fue comparar las dos versiones del algoritmo para tamaños de cluster k = 2 y k = 5. Se evaluó el tiempo de ejecución y la eficiencia de ambas implementaciones para determinar cuál enfoque ofrece un mejor rendimiento en términos de velocidad y utilización de recursos. Los resultados indican que la versión secuencial del algoritmo superó a la implementación paralela, demostrando un rendimiento superior en los experimentos realizados. Estos hallazgos tienen implicaciones significativas para aplicaciones donde la eficiencia temporal es un factor crítico. El informe proporciona información sobre el proceso de implementación, discute los resultados obtenidos y destaca consideraciones clave para elegir la versión más adecuada del algoritmo K-Means en escenarios específicos de agrupación de puntos en 2D. Estos hallazgos sirven como referencia valiosa para tomar decisiones informadas en futuros proyectos relacionados con el análisis de datos y la agrupación de puntos.**

***Index Terms*—algoritmo K-Means, implementación paralela, implementación secuencial, agrupación de puntos en 2D, comparación de rendimiento, eficiencia temporal**

## I. Introducción

El algoritmo K-means es un algoritmo de aprendizaje no supervisado utilizado principalmente en el campo de la minería de datos y análisis de clústeres. Su objetivo principal es agrupar un conjunto de datos en k clústeres distintos, donde cada clúster representa un grupo de elementos similares [1].

El contexto de uso del algoritmo K-means es muy amplio y se aplica en diversas áreas, incluyendo:

- Segmentación de clientes: Agrupa a los clientes en diferentes grupos basados en su comportamiento de compra, preferencias o características demográficas.
- Análisis de imágenes: Agrupa píxeles de una imagen en diferentes clústeres, útil para segmentación de objetos o compresión de imágenes.
- Agrupación de documentos: Clasifica documentos en categorías o temas similares para la organización y recuperación de información.
- Análisis de datos científicos: Agrupa datos experimentales para identificar patrones o características similares en biología, física, astronomía, etc.
- Detección de anomalías: Identifica puntos atípicos o anomalías que no se ajustan a ningún clúster específico.

Estos son solo algunos ejemplos del contexto de uso del algoritmo K-means. En general, se utiliza en situaciones donde se busca agrupar datos en clústeres basados en su similitud o proximidad, lo que proporciona información valiosa para el análisis y la toma de decisiones [2].

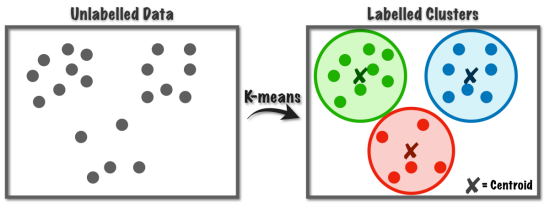


Fig. 1. Clusterización del K-means

## II. K-means

Para la implemntación del K-means se presenta sus dos versiones, una secuencial y otra paralela usando OpenMP. Ambas implementaciones presentan las mismas clases y funciones con ligeras variaciones. La implementación se puede ver en el sigueinte link de repositorio *k-means-Comparación*. La estructura *Color* representa un color con tres componentes: *r* (rojo), *g* (verde) y *b* (azul). Tiene un constructor que toma los valores de los componentes y los inicializa. La clase *Punto* representa un punto en un espacio bidimensional. Tiene dos miembros: *x* y *y*, que representan las coordenadas del punto en los ejes x e y, respectivamente. Tiene un constructor que toma los valores de *x* y *y* y los inicializa. La función *readPoints* es una función de plantilla que lee los puntos desde un archivo especificado por *nameF* y devuelve un vector con los puntos leídos. La función utiliza un objeto *std::ifstream* para abrir el archivo y leer los puntos línea por línea. Cada línea del archivo debe contener dos valores separados por espacios, que se interpretan como las coordenadas *x* e *y* de un punto. Los puntos leídos se agregan al vector *points* utilizando *emplace_back*. Finalmente, la función devuelve el vector con los puntos leídos.

La clase *KMeans* tiene los siguientes métodos:

- *ejecutar()*: Este método ejecuta el algoritmo K-means en paralelo. Toma como entrada un vector de puntos y asigna cada punto al cluster más cercano. Luego actualiza los centroides de los clusters iterativamente hasta que no haya cambios.

- *imprimirClusters() const*: Este método imprime los clusters resultantes. Muestra el número de puntos en cada cluster, así como las coordenadas de los centroides.
- *dibujarClusters() const*: Este método crea un archivo HTML que muestra los clusters en un mapa interactivo. Los centroides se representan como marcadores de mayor tamaño, mientras que los puntos individuales se representan como marcadores de menor tamaño, ambos con colores correspondientes a los clusters.

Los métodos privados de la clase *KMeans* son:

- *asignarPuntosAClusters()*: Este método asigna cada punto al cluster más cercano. Se realiza en paralelo, utilizando un bucle *for* con la directiva *omp parallel for* para distribuir los cálculos entre hilos.
- *inicializarClusters()*: Este método inicializa los clusters con centroides aleatorios. Los centroides se eligen de manera aleatoria a partir de los puntos proporcionados.
- *calcularDistancia()*: Este método calcula la distancia euclidiana entre dos puntos. Se utiliza para determinar la distancia entre un punto y el centroide de un cluster.
- *calcularDistancia()*: Este método calcula la distancia euclidiana entre dos puntos. Se utiliza para determinar la distancia entre un punto y el centroide de un cluster.

La función *dibujarPresentacion()* muestra una presentación en la consola antes de ejecutar el algoritmo. Simplemente muestra algunos títulos y animaciones de carga.

La función *Test()* es una función auxiliar que crea una instancia de *KMeans* y ejecuta el algoritmo con los puntos proporcionados y el número de clusters especificado. También mide y muestra el tiempo de ejecución.

*1) Implementación secuencial:*

```
// Clase Nodo
template<typename T>
class Nodo {
public:
    Punto<T> centroide;
    std::vector<Punto<T>> puntos;
    Color color; // color del cluster

    Nodo(const Punto<T>& centroide, const
        Color& color) : centroide(centroide),
        color(color) {}

    void agregarPunto(const Punto<T>& punto) {
        puntos.push_back(punto);
    }

    void limpiarPuntos() {
        puntos.clear();
    }
};

// Clase KMeans
template<typename T>
class KMeans {
private:
    int k;  // Nmero de clusters
    std::vector<Nodo<T>> clusters;

public:
    KMeans(int k_) : k(k_) {}


    void ejecutar(const std::vector<Punto<T>>&
         puntos);
    void imprimirClusters() const;
    void dibujarClusters() const;
private:
    void asignarPuntosAClusters(const std::
        vector<Punto<T>>& puntos);
    void inicializarClusters(const std::vector
        <Punto<T>>& puntos);
    T calcularDistancia(const Punto<T>& punto1
        , const Punto<T>& punto2);
};

template<typename T>
void KMeans<T>::ejecutar(const std::vector<
    Punto<T>>& puntos) {
        // Inicializar los clusters con
            centroides aleatorios

        //imprimirPuntos(puntos);

        inicializarClusters(puntos);

        bool cambios = true;
        while (cambios) {
            cambios = false;

            // Asignar cada punto al cluster
                ms cercano
            asignarPuntosAClusters(puntos);

            // Actualizar los centroides de
                los clusters
            for (auto& cluster : clusters) {
                if (!cluster.puntos.empty()) {
                    T sumX = 0, sumY = 0;
                    for (const auto& punto :
                        cluster.puntos) {
                        sumX += punto.x;
                        sumY += punto.y;
                    }
                    T nuevoCentroideX = sumX /
                         cluster.puntos.size()
                        ;
                    T nuevoCentroideY = sumY /
                         cluster.puntos.size()
                        ;
                    //cout<<"Cent: "<<
                        nuevoCentroideX << " ,
                         "<<nuevoCentroideY<<
                        endl;
                    cluster.centroide = Punto<
                        T>(nuevoCentroideX,
                        nuevoCentroideY);
                }
            }
        }
}

template<typename T>
void KMeans<T>::imprimirClusters() const {
        for (int i = 0; i < clusters.size();
            ++i) {
```

```
            std::cout << "Cluster " << i << ": Centroid = (" << std::fixed << std::setprecision(15) << clusters[i].centroide.x << ", " << clusters[i].centroide.y << ")" << std::endl;
            std::cout << "Points: ";
            std::cout<< "# Points: "<< clusters[i].puntos.size()<< endl;
            /*
            for (const auto& punto : clusters[i].puntos) {
                std::cout << "(" << std::fixed << std::setprecision(15) << punto.x << ", " << punto.y << ")" << std::endl;
            }
            */
            std::cout << std::endl;
        }
}

template<typename T>
void KMeans<T>::dibujarClusters() const {
    std::ofstream output("clusters.html");
    output << "<!DOCTYPE html>\n";
    output << "<html>\n";
    output << "<head>\n";
    output << "<meta charset=\"UTF-8\"/>\n";
    output << "<title>Puntos Coloreados con Zoom</title>\n";
    output << "<link rel=\"stylesheet\" href=\"https://unpkg.com/leaflet@1.2.0/dist/leaflet.css\" />\n";
    output << "<meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0, maximumscale=1.0, user-scalable=no\" />\n";
    output << "<script src=\"https://unpkg.com/leaflet@1.2.0/dist/leaflet.js\"></script>\n";
    output << "</head>\n";
    output << "<body>\n";
    output << "<div id=\"map\" style=\"width: 900px; height: 500px;\"></div>\n";
    output << "<script>\n";
    output << "var puntos = [\n";

    for (const auto& cluster : clusters) {
        output << "{ coordenadas: [" << cluster.centroide.x << ", " << cluster.centroide.y << "], color: 'rgb("
               << cluster.color.r << "," << cluster.color.g << "," << cluster.color.b << ")', centroide: true },\n";
        for (const auto& punto : cluster.puntos) {
            output << "{ coordenadas: [" << punto.x << ", " << punto.y << "], color: 'rgb("
                   << cluster.color.r << ","
                   << cluster.color.g << ","
                   << cluster.color.b << ")' },\n";
        }
    }

    output << "];\n";
    output << "var map = L.map('map').setView([0, 0], 1);\n";
    output << "var pointGroup = L.featureGroup().addTo(map);\n";

    output << "for (var i = 0; i < puntos.length; i++) {\n";
    output << "var punto = puntos[i];\n";
    output << "if (punto.centroide) {\n";
    output << "var marker = L.circleMarker(punto.coordenadas, {\n";
    output << "color: punto.color,\n";
    output << "fillColor: punto.color,\n";
    output << "fillOpacity: 1,\n";
    output << "radius: 8\n";
    output << "}).addTo(pointGroup);\n";
    output << "} else {\n";
    output << "var marker = L.circleMarker(punto.coordenadas, {\n";
    output << "color: punto.color,\n";
    output << "fillColor: punto.color,\n";
    output << "fillOpacity: 1,\n";
    output << "radius: 4\n";
    output << "}).addTo(pointGroup);\n";
    output << "}\n";
    output << "}\n";
    output << "map.fitBounds(pointGroup.getBounds());\n";
    output << "L.control.zoom().addTo(map);\n";
    output << "</script>\n";
    output << "</body>\n";
    output << "</html>\n";
    output.close();
}


template<typename T>
void KMeans<T>::asignarPuntosAClusters(const std::vector<Punto<T>>& puntos) {
    for (auto& cluster : clusters) {
        cluster.limpiarPuntos(); // Limpiar los puntos en el cluster antes de asignar nuevos puntos
    }

    for (const auto& punto : puntos) {
        int clusterActual = 0;
        T distanciaMinima = calcularDistancia(punto, clusters[0].centroide);

        for (int i = 1; i < clusters.size(); ++i) {
            T distancia = calcularDistancia(punto, clusters[i].centroide);
            if (distancia < distanciaMinima) {
                distanciaMinima = distancia;
                clusterActual = i;
            }
        }

```

```
162         clusters[clusterActual].agregarPunto(
                punto); // Asignar el punto al
                cluster ms cercano
163     }
164 }
165
166
167 template<typename T>
168 void KMeans<T>::inicializarClusters(const std
        ::vector<Punto<T>>& puntos) {
169     //cout<<"PS: "<<puntos.size()<<endl;
170     std::vector<int> indices(puntos.size());
171     std::iota(indices.begin(), indices.end(),
            0);
172     std::random_shuffle(indices.begin(),
            indices.end());
173
174     for (int i = 0; i < k; ++i) {
175         int r = rand() % 256; // Componente R
                aleatorio
176         int g = rand() % 256; // Componente G
                aleatorio
177         int b = rand() % 256; // Componente B
                aleatorio
178
179         clusters.emplace_back(Nodo<T>(puntos[
                indices[i]], Color(r, g, b)));
180     }
181 }
182
183
184 template<typename T>
185 T KMeans<T>::calcularDistancia(const Punto<T>&
        punto1, const Punto<T>& punto2) {
186         T dx = punto1.x - punto2.x;
187         T dy = punto1.y - punto2.y;
188         return std::sqrt(dx * dx + dy * dy);
189 }
```

*2) Implemnetación paralela:* El proceso de paralelización en este código se lleva a cabo utilizando la directiva `#pragma omp parallel for`, que indica al compilador que el bucle for siguiente se debe ejecutar en paralelo utilizando múltiples hilos. La paralelización mediante OpenMP reduce el tiempo de ejecución al distribuir la carga de trabajo entre múltiples hilos. La complejidad teórica del algoritmo no se modifica, sigue siendo O(n * k * d * iter), pero el tiempo de ejecución se reduce debido al paralelismo. Donde:

- n: Es el número de puntos en el conjunto de datos.
- k: Es el número de clusters que se desean generar.
- d: Es la dimensión de los puntos en el espacio de características.
- iter: Indica el número de iteraciones que realiza el algoritmo K-Means para converger hacia una solución.

Estos valores dependen del problema y del conjunto de datos utilizados. n y d están determinados por la naturaleza de los datos, k puede ser determinado por requisitos específicos o técnicas de selección de clusters, y el número de iteraciones se ajusta para lograr una convergencia adecuada.

En este caso, se utilizan dos bucles for paralelos en el código:

- El primer bucle for paralelo asigna cada punto al cluster más cercano. Cada hilo paralelo se encarga de asignar un subconjunto de puntos a clusters. Dado que el bucle itera sobre todos los puntos, se distribuye la carga de trabajo de manera equitativa entre los hilos.
- El segundo bucle for paralelo actualiza los centroides de los clusters. Al igual que el primer bucle, cada hilo paralelo se encarga de actualizar un subconjunto de clusters.

La complejidad del algoritmo K-Means sin paralelización es O(n * k * d * iter), donde n es el número de puntos, k es el número de clusters, d es la dimensión de los puntos y iter es el número de iteraciones. En cada iteración, se asignan todos los puntos a los clusters y se actualizan los centroides.

```
1  template<typename T>
2  void KMeans<T>::ejecutar(const std::vector<
       Punto<T>>& puntos) {
3      // Inicializar los clusters con centroides
           aleatorios
4      inicializarClusters(puntos);
5
6      bool cambios = true;
7      while (cambios) {
8          cambios = false;
9
10         // Asignar cada punto al cluster ms
               cercano en paralelo
11         #pragma omp parallel for
12         for (int i = 0; i < puntos.size(); ++i
               ) {
13             int clusterActual = 0;
14             T distanciaMinima =
                   calcularDistancia(puntos[i],
                   clusters[0].centroide);
15
16             for (int j = 1; j < clusters.size
                   (); ++j) {
17                 T distancia =
                       calcularDistancia(puntos[i
                       ], clusters[j].centroide);
18                 if (distancia <
                       distanciaMinima) {
19                     distanciaMinima =
                           distancia;
20                     clusterActual = j;
21                 }
22             }
23
24             #pragma omp critical
25             {
26                 clusters[clusterActual].
                       agregarPunto(puntos[i]);
                       // Asignar el punto al
                       cluster ms cercano
27             }
28         }
29
30         // Actualizar los centroides de los
               clusters en paralelo
31         #pragma omp parallel for
32         for (int i = 0; i < clusters.size();
               ++i) {
```

```
            if (!clusters[i].puntos.empty()) {
                T sumX = 0, sumY = 0;
                for (const auto& punto :
                    clusters[i].puntos) {
                    sumX += punto.x;
                    sumY += punto.y;
                }
                T nuevoCentroideX = sumX /
                    clusters[i].puntos.size();
                T nuevoCentroideY = sumY /
                    clusters[i].puntos.size();

                #pragma omp critical
                {
                    clusters[i].centroide =
                        Punto<T>(
                        nuevoCentroideX,
                        nuevoCentroideY);
                }
            }
        }
    }
}
```

.

## III. Comparación y Análisis

Para hacer las pruebas y ver el rendimiento de ambas implemntaciones, se usó la comparación a nivel de tiempo de ejecución. Para las pruebas se utilizó 2 archivos .txt, estos contenian números flotantes. El primer archivo de nombre `puntos_2_bloques.txt` es para encontrar los 2 *clústeres*, mientras que el archivo `puntos_5_bloques.txt` es para encontrar los 5 *clústeres*. En la figura 2 y 3 se muestra el resulta luego de aplicar el algoritmo de K-means para el *dataset* usado.

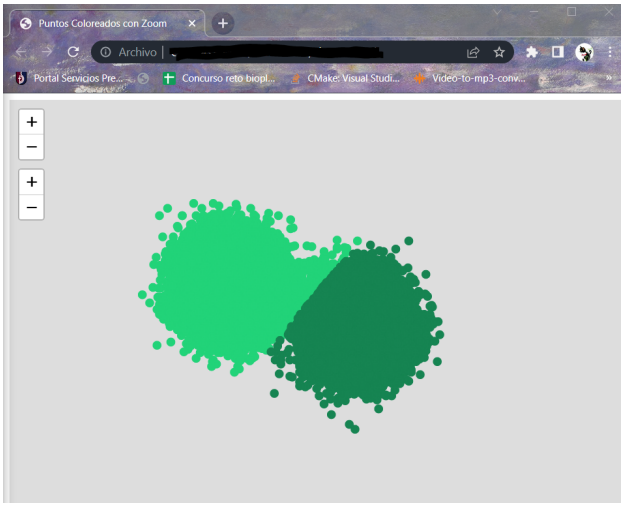


Fig. 2. Los 2 clústeres encontrados usando el algoritmo de K-means.

Como se puede apreciar tanto en la gráfica en la figura 5 y la tabla 4, El rendimiento a nivel de tiempo es mejor para la versión secuencial. La implementación presentada del algoritmo K-Means utiliza paralelización para asignar cada

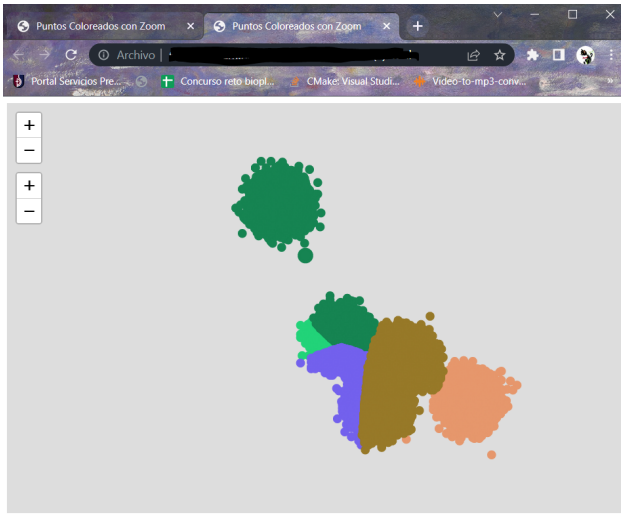


Fig. 3. Los 5 clústeres encontrados usando el algoritmo de K-means.

punto al cluster más cercano y actualizar los centroides de los clusters. Sin embargo, esta paralelización no logra mejorar el tiempo de ejecución y, de hecho, resulta en una mayor lentitud en comparación con la versión secuencial. Hay dos razones principales que contribuyen a esta desventaja en el rendimiento:

- Se utiliza una sección crítica (pragma omp critical) para garantizar la exclusión mutua al agregar puntos a los clusters y actualizar los centroides. Esto implica que solo un hilo puede realizar estas operaciones a la vez, anulando así los beneficios de la paralelización. La alta concurrencia necesaria para el K-Means no se aprovecha plenamente debido a esta restricción.
- El tamaño de los datos de entrada puede no ser lo suficientemente grande como para justificar el uso de paralelización. Los costos asociados con la creación y sincronización de hilos pueden superar los beneficios de dividir la carga de trabajo en múltiples hilos. Esto es especialmente cierto si los cálculos individuales realizados por cada hilo son relativamente simples.

| K | Secuencial | Paralela |
|---|---|---|
| 2 | 0.00681 | 0.0095529 |
| 5 | 0.006825 | 0.0094887 |

Fig. 4. Tabla que presenta el tiempo promedio obtenido luego de ejecutar la versión paralela y sencuencial para los 2 *dataset*.

Para determinar cuánto porcentaje es mejor un valor en comparación con otro, se hacen los siguientes cálculos:

$$\text{Porcentaje de mejora} = \left(\frac{\text{Valor antiguo} - \text{Valor nuevo}}{\text{Valor antiguo}}\right) \times 100 \quad (1)$$

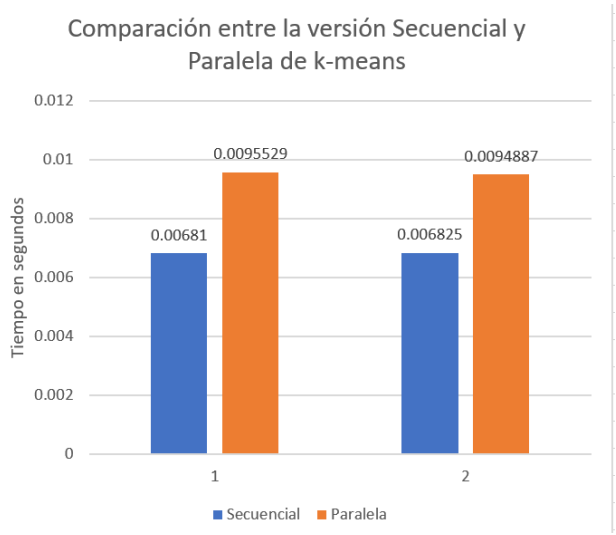


Fig. 5. Gráfica que presenta el tiempo promedio obtenido luego de ejecutar la versión paralela y sencuencial para los 2 *dataset* con k = 2 y k = 5.

Sustituyendo los valores proporcionados, el resultado sería:

$$\text{Porcentaje de mejora} = \left(\frac{0.0026637}{0.0094887}\right) \times 100 \approx 28.09\% \quad (2)$$

Por lo tanto, 0.006825 es aproximadamente un 28.09% mejor que 0.0094887.

## IV. Conclusiones

La implementación paralela del algoritmo K-Means usado, por los resultados obtenidos resulta más lenta en tiempo de ejecución en comparación con la versión secuencial debido a la falta de una paralelización efectiva y al costo adicional asociado con la sincronización de hilos. Para esta experimentación la versión secuencial es la más apropiada para calcular los clústeres.

## References

[1] Alguliyev, R. M., Aliguliyev, R. M., and Sukhostat, L. V. Parallel batch k-means for big data clustering. *Comput. & Ind. Eng. 152* (February 2021), 107023. Accessed: June 2, 2023.

[2] Contributors to Wikimedia projects. k-means clustering - Wikipedia. Wikipedia, the free encyclopedia, n.d. Accessed: June 2, 2023.