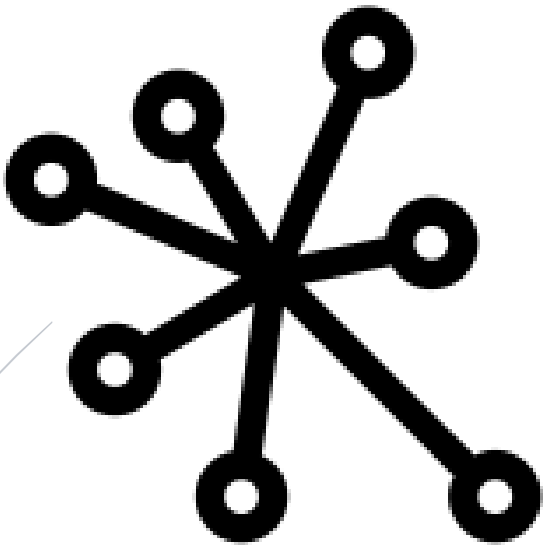
7-3-2024

HITO 1

Computación de Altas Prestaciones



Luis Javier Horcajada Torres
Antonio Domínguez Martín
MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

Índice

1. Introducción

El objetivo de estas prácticas de laboratorio consiste en el análisis y la posterior reducción del tiempo de ejecución de los algoritmos de clustering en entornos de gran volumen de datos utilizando técnicas de paralelización y lenguajes que den soporte a las mismas como son MPI y/o OpenMP.

Para ello, en este primer hito se partirá de una implementación secuencial de base del algoritmo k-means el cual se paralelizará con diferentes técnicas y/o tecnologías posteriormente y estudiar el efecto de cada una de ellas y extraer conclusiones sobre las pruebas realizadas.

Las ejecuciones de los dos algoritmos han sido realizadas sobre un ordenador portátil MSI-GL75-Leopard-10SEK, procesador i7-10750H+HM470, memoria DDR IV 8GB*2 (2666MHz) y gráfica Nvidia GeForce® RTX 2060, GDDR6 6GB.

2. Introducir un control de tiempos de ejecución en el código con el fin de analizar los tiempos de ejecución del algoritmo dependiendo de ciertos parámetros, sobre todo el número de clusters.

A partir del código original del archivo *kmeans.py* se ha realizado la medición de los tiempos tanto del tiempo inicial como final (cuya resta es el tiempo total de ejecución del algoritmo) gracias a la dependencia time y su método time.time(). Cabe mencionar que los tiempos han sido redondeados a las centésimas.

```
def run_kmeans(pixels, num_clusters):
    start_time = time.time()

    # Configurar los criterios para k-means
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.2)

    # Ejecutar k-means
    _, labels, centers = cv2.kmeans(pixels, num_clusters, None, criteria, 10,
cv2.KMEANS_RANDOM_CENTERS)

    end_time = time.time()
    execution_time = end_time - start_time

    return labels, centers, execution_time
```

Como se puede observar, se ha creado un método dónde se ejecuta el algoritmo, de esta manera mejoramos la legibilidad del código y su calidad. Adicionalmente, se puede ejecutar el código con un número de clusters distintos en una única ejecución.

A continuación, exponemos los tiempos obtenidos al ejecutar este código con distintos valores de clusters.

Primera ejecución

Nº de clusters	Tiempo de ejecución
2	11.71s
3	19.73s
4	29.90s
5	36.07s

Segunda ejecución

Nº de clusters	Tiempo de ejecución
2	10.74s
3	19.73s
4	30.06s
5	36.00s

Tercera ejecución

Nº de clusters	Tiempo de ejecución
2	10.55s
3	19.77s
4	29.81s
5	35.50s

Tiempo medio de las ejecuciones

Nº de clusters	Tiempo de ejecución
2	11.00s
3	19.74s
4	29.92s
5	35.50s

Esta prueba también se ha realizado sobre un computador Windows cuyo resultado es el siguiente:

Nº de clusters	Tiempo de ejecución
2	31.98s
3	117.5s
4	217.52s
5	570.43s

Es importante destacar que este ordenador Windows, es considerablemente superior al portátil usado en las pruebas anteriores, por lo tanto, deducimos que las librerías principales (**numpy, opencv**) están mejor optimizadas para una distribución Linux.

3. Sustituir la función de realización de k-means que proviene de la librería de OpenCV por otra solución propia (implementada desde cero, aunque puede no ser totalmente original) que permita modificar y analizar el código fuente de cada función y de esta forma observar el coste temporal de cada una de las partes del algoritmo. Elegir alguna de las tres heurísticas que se utilizan para implementar k-means.

A la hora de abordar la implementación de nuestro k-means teníamos tres opciones:

- **Algoritmo de Lloyd:** Es el más sencillo y el primero que surgió, divide los datos en K grupos minimizando la varianza, se inicializan los centroides de manera aleatoria lo que hace que su eficiencia no sea óptima y sea aleatoria.
- **Algoritmo de MacQueen:** Es una variante del algoritmo de Lloyd y está orientado para grandes conjuntos de datos, actualiza los centroides incrementalmente.
- **Algoritmo de Hartingan-Wong:** Mejora la calidad de los grupos al permitir cambios múltiples por iteración siendo más eficiente, pero a la vez es más costoso computacionalmente**.**

Se ha decidido utilizar el algoritmo de Lloyd ya que es el más común, también es el original y es el más similar al algoritmo integrado en la propia librería opencv de Python, por lo que la comparación sería más justa.

Adicionalmente también se ha optado por modularizar el código y también permitir que el usuario indique los clusters con los que quiere realizar la ejecución para mejorar su calidad.

3.1. Parseo de argumentos

```
def argparser():
    parser = argparse.ArgumentParser()
    parser.add_argument("-k", "--kmeans", dest="K", type=int, required=True, help="Kmeans size")
    args = parser.parse_args()
    K = args.K
    return K
```

3.2. Toma de tiempos

```
def run_kmeans(data, K):
    print(f'\nEmpezando kmeans con {K} clusters...')
    start_time = time.time()
    labels, centroids = kmeans(data, K)
    end  time = time.time()
    execution_time = end_time - start_time
    return labels, centroids, execution_time
```

3.3. K-means

Antes de mostrar el código, nos gustaría indicar que para la implementación del código nos hemos basado en pseudocódigos publicados en internet, concretamente en el siguiente.

Algorithm 1 Lloyd

1: choose k as the number of centroids
2: randomly assign all points to a centroid
3: calculate the centroids as mean of their assigned points
4: **repeat**
5: reassign each datapoint to its closest centroid
6: recalculate centroids as mean over their assigned datapoints
7: **until** convergence

[1] Figura 1

```
def kmeans(data, K):
    # Se inicializan los centroides aleatoriamente eligiendo K puntos de los datos
    indices = np.random.choice(data.shape[0], K, replace=False)
    centroids = data[indices]

    # Copia de centroides para su posterior comparación
    prev_centroids = centroids.copy()

    max_iterations = 100
    tolerance = 0.2
    for i in range(max_iterations):
        # Se asignan los datos a cada centroide según su distancia
        distances = np.linalg.norm(data[:, np.newaxis] - centroids, axis=2)
        labels = np.argmin(distances, axis=1)

        # Se actualiza la posición de cada centroide en función de la cantidad de datos de cada uno
        centroids = np.zeros((K, data.shape[1]))
        for j in range(K):
            cluster_points = data[labels == j]
            if len(cluster_points) > 0:
                centroids[j] = np.mean(cluster_points, axis=0)

        # Se comprueba la distancia con los centroides anteriores
        if np.linalg.norm(centroids - prev_centroids) < tolerance:
            break

        prev_centroids = centroids.copy()

    return labels, centroids
```

El funcionamiento del código es el siguiente:

- Se inicializan los centroides iniciales, cogiendo K píxeles (siendo K el número de clusters) y sus respectivos datos de manera aleatoria.

- Se calculan las distancias de cada punto a los centroides, y se asigna cada punto al centroide más cercano.
- Se recorre cada clúster reasignando los centroides en función de la media de los puntos de cada clúster.
- Se comprueba la distancia de los nuevos centroides con los anteriores, si la distancia es mejor que la tolerancia se termina el algoritmo.

4. Realizar un estudio de la solución general propuesta (con detalle rendimiento variando alguno de los parámetros, principalmente el número de clusters.

A partir del código del archivo *kmeansPropio.py* se ha realizado la medición de los tiempos tanto del tiempo inicial como final (cuya resta es el tiempo total de ejecución del algoritmo) gracias a la dependencia time y su método time.time(). Cabe mencionar que los tiempos han sido redondeados a las centésimas.

Estas ejecuciones también han sido realizadas con el mismo equipo del punto 1.

Primera ejecución

Nº de clusters	**Tiempo de ejecución**
2	11.74s
3	13.55s
4	36.54s
5	84.92s

Segunda ejecución

Nº de clusters	**Tiempo de ejecución**
2	08.98s
3	22.33s
4	41,21s
5	84.93s

Tercera ejecución

Nº de clusters	**Tiempo de ejecución**
2	10.13s
3	35.26s
4	36.53s
5	99.53s

Tiempo medio de las ejecuciones

Nº de clusters	**Tiempo de ejecución**
2	10.28s
3	23,71s
4	38,09s
5	89,79s

Comparación de tiempos (K-means propio/K-means librería)

Nº de clusters	Tiempo de ejecución
2	7% más rápido
3	20.19% más lento
4	27.29% más lento
5	153.05% más lento

Se puede observar que la librería oficial es considerablemente más rápida a mayor número de clústeres que nuestra implementación.