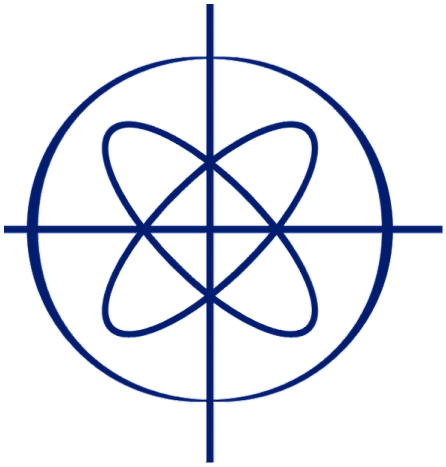


ITESO

UNIVERSIDAD JESUITA
EN GUADALAJARA

Sistemas Paralelos

Jose Luis Elvira Valenzuela

Nelson Victor Cruz Hernandez
MS705135

Algoritmo K-means

K-means

K -means es un método de agrupamiento, que tiene como objetivo la partición de un conjunto de *n* observaciones en *k* grupos en el que cada observación pertenece al grupo más cercano a la media. Es un método utilizado en minería de datos y machine learning. El algoritmo más común utiliza una técnica de refinamiento iterativo, también se le conoce como algoritmo de Lloyd, sobre todo en la comunidad informática.

Dado un conjunto inicial de *k* centroides $m_1^{(1)},\ldots,m_k^{(1)}$, el algoritmo continúa alternando entre dos pasos:

Paso de asignación: Asigna cada elemento del dataset al grupo o cluster con la media más cercana.

$$S_i^{(t)} = \{x_p : \|x_p - m_i^{(t)}\| \leq \|x_p - m_j^{(t)}\| \ \forall \ 1 \leq j \leq k\}$$

Donde cada x_p va exactamente dentro de un $S_i^{(t)}$, incluso aunque pudiera ir en dos de ellos.

Paso de actualización: Calcular los nuevos centroides como el centroide de las observaciones en el grupo.

$$\mathbf{m}_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{\mathbf{x}_j \in S_i^{(t)}} \mathbf{x}_j$$

El algoritmo se considera que ha convergido cuando las asignaciones ya no cambian.

El pseudocódigo se podría ver de la siguiente forma:

```
# Function: K Means
# -------------
# K-Means is an algorithm that takes in a dataset and a constant
# k and returns k centroids (which define clusters of data in the
# dataset which are similar to one another).
def kmeans(dataSet, k):

    # Initialize centroids randomly
    numFeatures = dataSet.getNumFeatures()
    centroids = getRandomCentroids(numFeatures, k)

    # Initialize book keeping vars.
    iterations = 0
    oldCentroids = None

    # Run the main k-means algorithm
    while not shouldStop(oldCentroids, centroids, iterations):
        # Save old centroids for convergence test. Book keeping.
        oldCentroids = centroids
        iterations += 1

        # Assign labels to each datapoint based on centroids
        labels = getLabels(dataSet, centroids)

        # Assign centroids based on datapoint labels
        centroids = getCentroids(dataSet, labels, k)

    # We can get the labels too by calling getLabels(dataSet, centroids)
    return centroids
```

Serial Approach

Para propósitos de la entrega el algoritmo fue implementado en lenguaje C, es posible ver la version serial en el siguiente repositorio.

NOTA: El algoritmo serial se encuentra en la rama de master

https://github.com/VictorCruzIsc/MachineLearningAlgorithms/tree/master/clustering

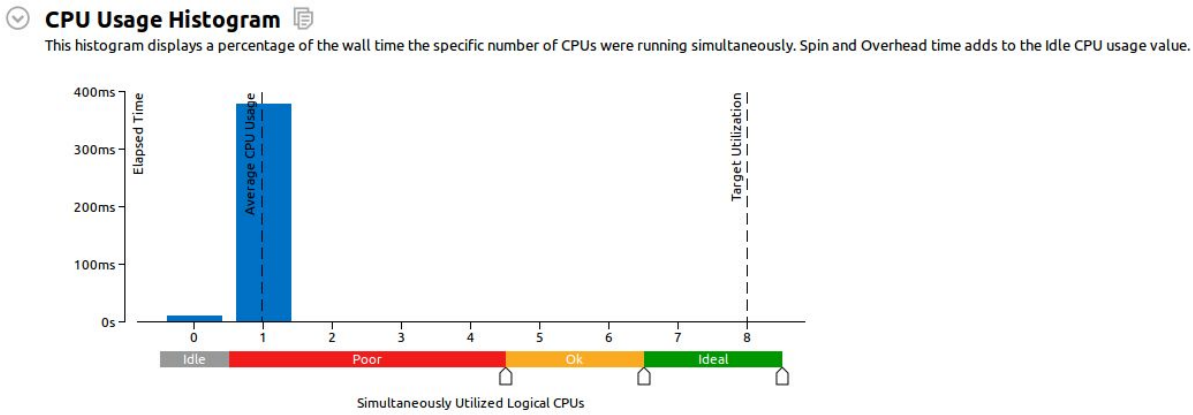
Es posible apreciar las funciones que permiten la ejecución del algoritmo.

```
int stopKmeans(Cluster *clusters, int totalClusters, int epochs);
void refreshCentroids(Cluster *clusters, int totalClusters);
```

```
void kmeans(Point *dataset, int totalElementsDataSet, Cluster *clusters, int totalClusters){
        int i,j, winner, epochs=0;
        int continueRunning = 1;
        while(continueRunning){
                cleanClusters(clusters, totalClusters);
                for(i=0; i<totalElementsDataSet; i++){
                        double distance = getDistance(dataset[i], clusters[0].currentCentroid);
                        winner = 0;
                        for(j=1; j<totalClusters; j++){
                                if(distance > getDistance(dataset[i], clusters[j].currentCentroid)){
                                        distance = getDistance(dataset[i], clusters[j].currentCentroid);
                                        winner = j;
                                }
                        }
                        clusters[winner].elements[clusters[winner].totalElements] = dataset[i];
                        clusters[winner].totalElements++;
                }
                refreshCentroids(clusters, K);
                epochs++;
                continueRunning = stopKmeans(clusters, K, epochs);
        }
        printClusters(clusters, K);
        printf("continueRunning: %d, Epochs: %d\n", continueRunning, epochs);
}
```

Parallel Approach

Serial Version - Basic Hotspots analysis



CPU Usage Histogram

El histograma de uso de CPU muestra que hay un largo camino por recorrer para lograr que el algoritmo de k-means que fue implementado sea optimo en cuestión de uso de CPU.

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle Poor Ok Ideal Over)	Spin Time	Overhead Time	Module	Function (Full)	Source File	Start Address
getDistance	139.971ms	0ms	0ms	skm	getDistance		0x400830
kmeans	87.998ms	0ms	0ms	skm	kmeans		0x400ef9
__pow	84.037ms	0ms	0ms	libm.so.6	__pow	w_pow.c	0x26990
__sqrt	48.001ms	0ms	0ms	libm.so.6	__sqrt	w_sqrt.c	0x26c50
[Import thunk pow]	19.994ms	0ms	0ms	skm	[Import thunk pow]		0x4005e0

Bottom-up

De acuerdo al analisis de hotspots en la parte de Bottom-up es posible apreciar que las funciones en las que el programa esta ocupando mas tiempo son getDistance(), kmeans(), _pow(), _sqrt()

Function	CPU Time: Total Effective Time by Utilization (Idle Poor Ok Ideal Over)	Spin Time	Ove.. Time	CPU Time: Self Effective Time by Utilization (Idle Poor Ok Ideal Over)	Spin Time	Ov.. Time	Modu
_start	100.0%	0.0%	0.0%	0ms	0ms	0ms	skm
main	100.0%	0.0%	0.0%	0ms	0ms	0ms	skm
kmeans	100.0%	0.0%	0.0%	87.998ms	0ms	0ms	skm
__libc_start_main	100.0%	0.0%	0.0%	0ms	0ms	0ms	libc.so.6
getDistance	76.8%	0.0%	0.0%	139.971ms	0ms	0ms	skm
__pow	22.1%	0.0%	0.0%	84.037ms	0ms	0ms	libm.so.6
__sqrt	12.6%	0.0%	0.0%	48.001ms	0ms	0ms	libm.so.6
[Import thunk pow]	5.3%	0.0%	0.0%	19.994ms	0ms	0ms	skm

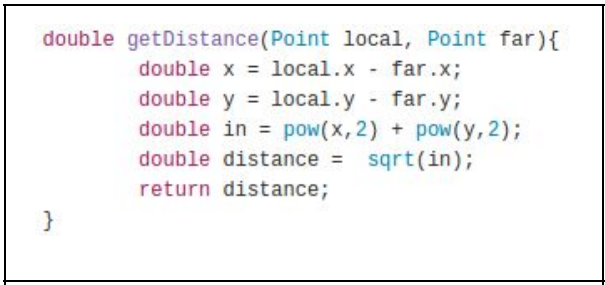
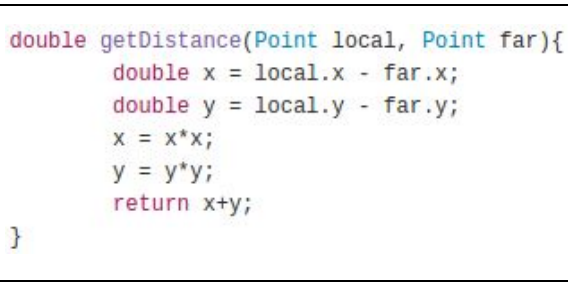
Caller/Callee

Esto es mas secillo verlo en la parte de Caller como se ve en la imagen de arriba

Analisis y optimizacion

getDistance() se encarga de calcular la distancia entre dos puntos a través de la fórmula de Euclidiana $\sqrt{(x-y)^2} = |x-y|$., para hacer esto está usando funciones de pow() y sqrt() que tienen un gran impacto en el uso de CPU por lo que para mejorar el desempeño es necesario encontrar un forma diferente de calcular dichos valores.
La forma de mejorar esto es evitar el pow() multiplicando los elementos por si mismos que a nivel computacional es mas barato, para la operación de sqrt() simplemente se evita ya que

por la naturaleza del problema, la distancia a uno de los centroides puede obtenerla sin necesidad de la raíz cuadrada.
La función optimizada queda de la siguiente forma.

	
Función antes de optimizar	Funcion despues de la optimización

kmeans() es la funcion que lleva a cabo todo el procesamiento del algoritmo.
Por la naturaleza del problema es necesario actualizar continuamente los elementos del dataset que pertenecen a cada cluster y por lo tanto el número de elementos que tiene cada cluster.
De forma serial el código tiene una sección crítica demasiado grande para atacar como un critical section por lo que es necesario encontrar una forma de deshacer esta condición de concurso.

Secciones críticas y variables que la causan

```
void kmeans(Point *dataset, int totalElementsDataSet, Cluster *clusters, int totalClusters){
        int i,j, winner, epochs=0;
        int continueRunning = 1;
        while(continueRunning){
                cleanClusters(clusters, totalClusters);
                for(i=0; i<totalElementsDataSet; i++){
                        double distance = getDistance(dataset[i], clusters[0].currentCentroid);
                        winner = 0;
                        for(j=1; j<totalClusters; j++){
                                if(distance > getDistance(dataset[i], clusters[j].currentCentroid)){
                                        distance = getDistance(dataset[i], clusters[j].currentCentroid);
                                        winner = j;
                                }
                        }
                        clusters[winner].elements[clusters[winner].totalElements] = dataset[i];
                        clusters[winner].totalElements++;
                }
                refreshCentroids(clusters, K);
                epochs++;
                continueRunning = stopKmeans(clusters, K, epochs);
        }
        printClusters(clusters, K);
        printf("continueRunning: %d, Epochs: %d\n", continueRunning, epochs);
}
```

Si se quiere paralelizar este método, la variable winner sera accedida por varios hilos y cuando se quiere acceder al arreglo de clusters con indice winner podriamos estar teniendo actualizaciones y aumentos a diferentes clusters ya que no podemos garantizar que winner es la misma variable a través de todos los hilos.
La forma de atacar esta sección crítica es crear una matriz de resultados en la cual cada hilo guardará el resultado que se tenga y solamente hara operaciones sobre su propia

estructura de datos, y posteriormente de forma manual, se hará la suma o réduction de todos estos resultados.

Variables para que cada hilo guarde sus resultados

```
int totalElementsMatrix[totalClusters][totalThreads];
double sumXMatrix[totalClusters][totalThreads];
double sumYMatrix[totalClusters][totalThreads];
memset(totalElementsMatrix, 0, sizeof(totalElementsMatrix[0][0]) * totalClusters * totalThreads);
memset(sumXMatrix, 0, sizeof(sumXMatrix[0][0]) * totalClusters * totalThreads);
memset(sumYMatrix, 0, sizeof(sumYMatrix[0][0]) * totalClusters * totalThreads);
```

Búsqueda por cada elemento del dataset a que cluster pertenece

```
#pragma omp parallel for private(threadId, winner)
                //Check which centroid is the nearest for the coordinate
                for(i=0; i<totalElementsDataSet; i++){
                        threadId = omp_get_thread_num();
                        winner = winnerCluster(dataset[i], clusters, totalClusters);
                        totalElementsMatrix[winner][threadId]++;
                        sumXMatrix[winner][threadId] += dataset[i].x;
                        sumYMatrix[winner][threadId] += dataset[i].y;
                        //printf("i:%d) W:%d | TID:%d | C:%d\n",i,winner,threadId, totalElementsMatrix[winner][threadId]);
                        whichClusterIn[i] = winner;
                }
```

Las matrices son accedidas en base al cluster y el hilo que está ejecutando el código en ese momento.

Reducción manual de los resultados (en c no es posible aplicar reduction a variables de tipo arreglos, esto solo es posible en fortran)

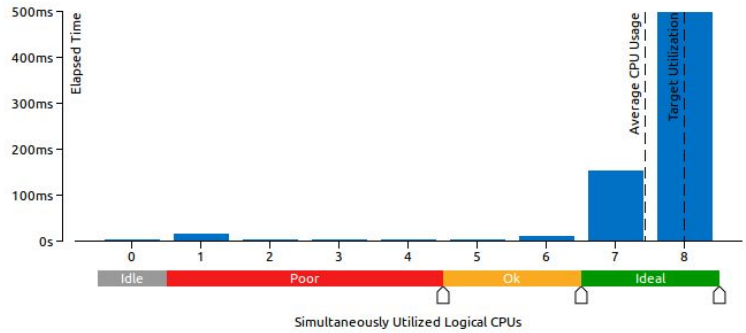
```
//Matrix reduction for totalElements, xSum, ySum
for(i=0; i<totalClusters; i++){
        for(j=0; j<totalThreads; j++){
                clusters[i].totalElements += totalElementsMatrix[i][j];
                clusters[i].xSum += sumXMatrix[i][j];
                clusters[i].ySum += sumYMatrix[i][j];
        }
}
```

Una vez aplicadas dichas mejoras se ejecuta el programa y se corren las herramientas de Parallel Studio dando los resultados mostrados a continuación.

Parallel Version - Basic Hotspots analysis

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



CPU Usage Histogram

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle / Poor / Ok / Ideal / Over)	Spin Time	Overhead Time	Module	Function (Full)	Source File	Start Address
winnerCluster	2.460s	0s	0s	skm	winnerCluster		0x400df7
getDistance	1.268s	0s	0s	skm	getDistance		0x4009b0
kmeans._omp_fn.1	1.116s	0s	0s	skm	kmeans._omp_fn.1		0x401b9c
func@0x9c70	0.242s	0s	0s	libgom...	func@0x9c70		0x9c70
func@0x9b10	0.036s	0s	0s	libgom...	func@0x9b10		0x9b10
func@0x874b	0.020s	0s	0s	libgom...	func@0x874b		0x874b
omp_get_thread_num	0.010s	0s	0s	libgom...	omp_get_thread_...		0x74f0
func@0x8c90	0.008s	0s	0s	libgom...	func@0x8c90		0x8c90

Bottom-up

Function	CPU Time: Total — Effective Time by Utilization (Idle / Poor / Ok / Ideal / Over)	Spin Time	Ove.. Time	CPU Time: Self — Effective Time by Utilization (Idle / Poor / Ok / Ideal / Over)	Spin Time	Ov.. Time	Mod
kmeans._omp_fn.1	94.1%	0.0%	0.0%	1.116s	0s	0s	skm
func@0x8290	87.2%	0.0%	0.0%	0s	0s	0s	libgomp.
start_thread	87.2%	0.0%	0.0%	0s	0s	0s	libpthrea
__clone	87.2%	0.0%	0.0%	0s	0s	0s	libc.so.6
winnerCluster	72.3%	0.0%	0.0%	2.460s	0s	0s	skm
getDistance	24.6%	0.0%	0.0%	1.268s	0s	0s	skm
_start	12.8%	0.0%	0.0%	0s	0s	0s	skm
main	12.8%	0.0%	0.0%	0s	0s	0s	skm
kmeans	12.8%	0.0%	0.0%	0s	0s	0s	skm
__libc_start_main	12.8%	0.0%	0.0%	0s	0s	0s	libc.so.6
func@0x9c70	4.7%	0.0%	0.0%	0.242s	0s	0s	libgomp.
func@0x9b10	0.7%	0.0%	0.0%	0.036s	0s	0s	libgomp.
func@0x874b	0.4%	0.0%	0.0%	0.020s	0s	0s	libgomp.
omp_get_thread_num	0.2%	0.0%	0.0%	0.010s	0s	0s	libgomp.
func@0x8c90	0.2%	0.0%	0.0%	0.008s	0s	0s	libgomp.

Caller/Callee

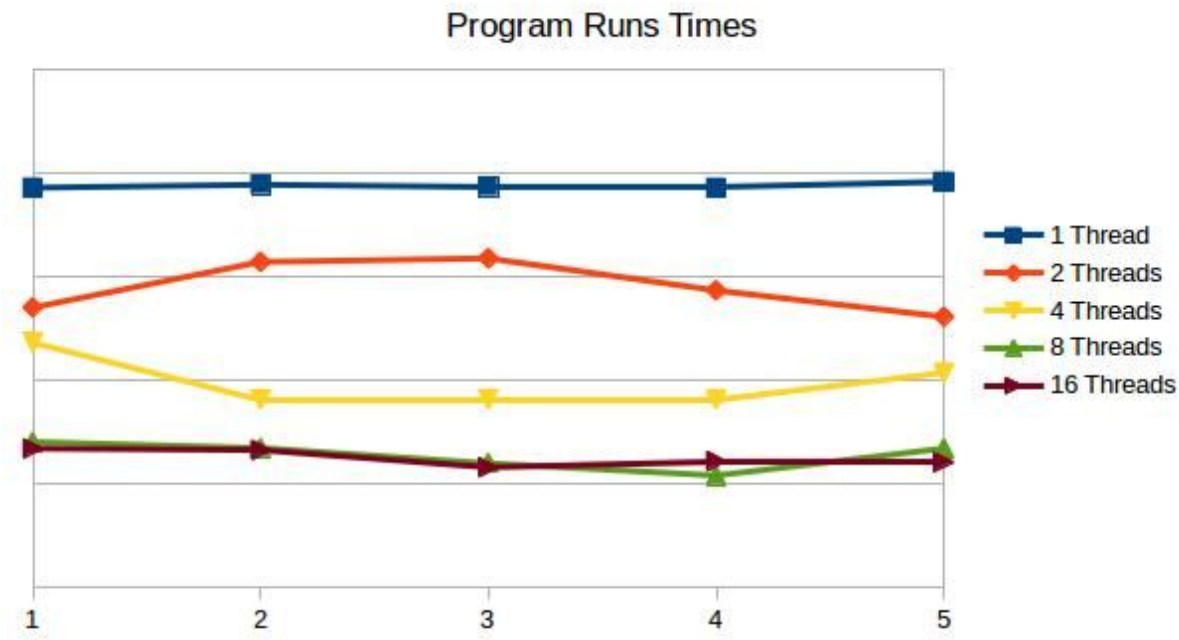
Cómo es posible apreciar la mejora fue bastante.

Resultados

Se corrieron 5 pruebas con diferente número de threads y los resultados en tiempo se guardaron en la siguiente tabla.

	1	2	3	4	5
1 Thread	1.930329	1.944611	1.934531	1.929269	1.958543
2 Threads	1.351534	1.571757	1.588697	1.434128	1.306285
4 Threads	1.181073	0.907307	0.908163	0.90791	1.036185
8 Threads	0.701501	0.672372	0.598465	0.539581	0.67058
16 Threads	0.670627	0.663032	0.57914	0.60859	0.604007

Resultados de múltiples experimentos con diferente número de threads



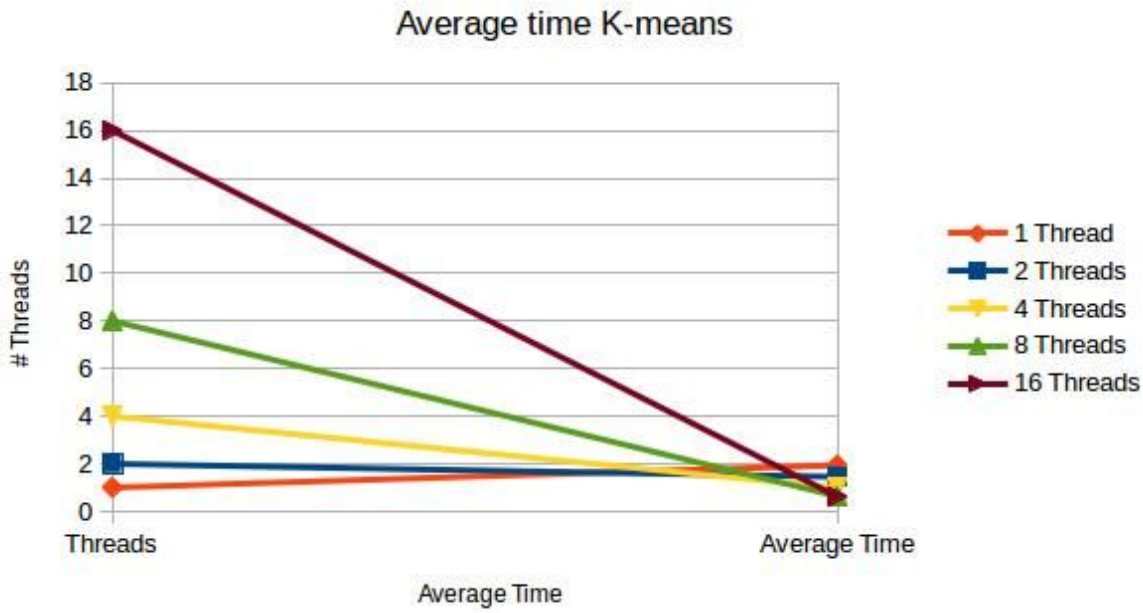
Resultados estables a través de los experimentos

Al graficar la tabla anterior es posible apreciar que los resultados son estables y de primera instancia se aprecia que el resultado con 8 y 16 threads es casi el mismo por lo que seria mejor solo ejecutarlo con 8 hilos.

La siguiente tabla muestra el promedio de los experimentos anteriores contra el número de hilos con los que fue probado de tal forma que se pueda verificar que hubo una mejora al ejecutar el algoritmo de forma paralela.

	Samples	Threads	Average Time
1	400000	1	1.9394566
2	400001	2	1.4504802
3	400002	4	0.9881276
4	400003	8	0.6364998
5	400004	16	0.6250792

Promedios de tiempos



Promedios de tiempos gráficamente

En la gráfica es posible apreciar una mejora en el tiempo de ejecución del algoritmo

Mejoras

De acuerdo al análisis de Parallel Amplifier las mejoras en rendimiento que es posible son pocas o nulas, sin embargo las mejoras podrían ser en optimizacion de codigo, principalmente en recibir parámetros para el número de muestras, y clusters o el aumento de muestras que se pueden analizar.
Además de agregar las funcionalidades de inicialización para Forgy y Partición Aleatoria.

Descarga y ejecución del código

https://github.com/VictorCruzIsc/MachineLearningAlgorithms

Instrucciones de descarga y ejecución para Linux:
Desde github

```
mkdir project
cd project/
git clone https://github.com/VictorCruzIsc/MachineLearningAlgorithms.git
Cloning into 'MachineLearningAlgorithms'...
cd MachineLearningAlgorithms
cd clustering
git checkout parallelVersion
make
./skm
```

Local
Descomprimir kmeans.zip

```
make
./skm
```

NOTA: Para hacer modificaciones de cluster o muestras es necesario abrir el archivo serialKmeans.h, N es el numero de muestras, K es el numero de clusters.