Clustering

Cristian David Suaza Cárdenas - csuazaca@eafit.edu.co
Andrés Felipe García Granados - agarci45@eafit.edu.co
Jeniffer María Palacio Sandoval - jmpalaci@eafit.edu.co

EAFIT

Palabras clave— k-means, distancia euclidiana, dataset, cluster, pcam, mpi, python, serial, paralelo

I. Resumen

En el presente documento se plantea una solución al Proyecto3 de la materia Tópicos Especiales en Telemática. Éste consiste en implementar un algoritmo paralelo que permita agrupar diferentes documentos, específicamente se hablará del algoritmo ***k-means***, el cual toma como base la ***distancia euclidiana*** que es una métrica de similitud entre documentos y que de hecho se escoge ésta debido a que es la que utiliza por defecto k-means.

II. Introducción

El mundo de hoy se puede decir que es inconcebible sin la tecnología, pues gracias a dispositivos como los computadores, celulares, entre otros, podemos buscar todo tipo de información que a diario fluye por medio de internet. Es tanta la información que se transmite por este medio, que resulta necesario para los creadores de buscadores como Google, Firefox, Microsoft Edge, o cualquier otra plataforma para buscar algo de un interés particular, utilizar algoritmos que agrupen dicha información y de esta manera poder mostrarle al usuario lo más relacionado posible con la búsqueda realizada por éste.

III. Marco teórico

Se cuenta con un conjunto de documentos desordenados, los cuales requieren ser organizados en un número de clusters dependiendo del grado de similitud entre estos documentos. El primer paso a realizar es determinar qué tan similares son cada uno de los documentos del dataset. Existen varios algoritmos que ayudan a determinar la similitud entre dos documentos, entre ellos están: *Distancia Euclidiana, Similitud Coseno, Coeficiente de Jaccard, Coeficiente de Correlación de Pearson, Divergencia de Kullback-Leibler.*

Luego de determinar qué tan similares son los documentos, se debe utilizar un algoritmo de agrupamiento como *k-means* para clasificarlos en un número de k clusters.

IV. Análisis y Diseño mediante PCAM

Para el diseño del algoritmo paralelo se utiliza la metodología **PCAM**, la cual consiste en Partición, Comunicación, Aglomeración y Mapeo.

La implementación de este algoritmo paralelo se realiza en el entorno de desarrollo *PyCharm Edu 4.0.2* mediante *MPI.*

La **Partición** se realizó por medio de *Descomposición de Dominio* en la cual la aplicación es común en todos los procesadores pero éstos trabajan en diferentes porciones de datos, esto es, hay 4 procesadores que ejecutan la misma tarea pero sobre diferentes datos.

En la **Comunicación** se asignan las tareas a los distintos procesadores de la siguiente manera: se tiene una matriz cuadrada de tamaño n donde se almacenarán las distancias entre los documentos. La n se dividirá entre 4 procesadores, es decir, cada procesador utilizará $\frac{1}{4}$ de la matriz para calcular las distancias correspondientes entre los documentos. De esta manera, se emplea el *balanceo de carga*, ya que se distribuirá la carga para los 4 procesadores.

En la **Aglomeración** se combinan tareas pequeñas con una tarea un poco más grande, es decir, hay 4 procesadores en el que cada uno halla un promedio de distancias para que luego un quinto procesador coja esos promedios y los junte en uno solo con el fin de hallar nuevos centroides y realizar el algoritmo de *k-means* que es la tarea más grande dentro de la implementación.

En el **Mapeo** se implementa un *mapeo dinámico descentralizado* donde cada procesador tiene su pool de tareas y se comunica con otros procesadores dependiendo de sus necesidades. De esta manera, se asignan tareas pequeñas a 5 procesadores y se asigna una tarea un poco más grande a un sexto procesador, es decir, 4 procesadores realizan la tarea cada uno de hallar las distancias entre documentos para $\frac{1}{4}$ de la matriz, lo cual se menciona anteriormente en el apartado de la Comunicación. Hay un quinto procesador que se encarga de juntar esas partes de la matriz halladas por los procesadores anteriores para conformar la matriz de distancias entre documentos. Por último, se destina un procesador a realizar una tarea un poco más grande, la cual consiste en la implementación de *k-means*, donde se calculan nuevos centroides y se va agrupando los documentos de acuerdo a la similitud entre ellos. Esto se hace hasta que los centroides converjan.

V. Implementación

A continuación se muestra cada una de las librerías y funciones requeridas para llevar a cabo la implementación del algoritmo tanto serial como paralelo:

- Se importan las siguientes librerías:
  - *glob* - retorna una lista con los nombres de los archivos que cumplan un patrón dado. En este caso son todos los archivos que sean *.txt*
  - *math* - provee acceso a funciones matemáticas como por ejemplo: sacar raíz cuadrada de alguna expresión.
  - *randint* - módulo para generar números random.
  - *MPI* - provee las funciones necesarias para el paso de mensajes, lo cual permite aprovechar múltiples procesadores.
  - *time* - provee múltiples funciones relacionadas con el tiempo, en este caso se utiliza para ver el tiempo de ejecución del programa.

- En un arreglo se guardan las palabras vacías [3] (stop words[1]), para omitirlas durante la ejecución del algoritmo de similitud entre 2 documentos. Se utilizaron las palabras vacías del inglés debido a que el los documentos del dataset utilizado están en este idioma.

- Se define la función *countWords*[2] para contar el número de veces que aparece cada una de las palabras dentro del documento que se pasa como argumento:

```
def countWords(filename):
    file=open(filename,"r+")
    wordCounter={}
    for word in file.read().split():
        if stop_words.__contains__(
            word.lower())==False:
            if word not in wordCounter
                :
                wordCounter[word] = 1
            else:
                wordCounter[word] += 1
        file.close()
    return wordCounter
```

- Se define la función *similFiles* para comparar qué tan similares son 2 documentos, esto es, se buscan las palabras del par de documentos que se pasan como argumentos, se realiza una intersección para buscar las palabras comúnes en ambos documentos y mirar las veces que éstas aparecen en cada uno de ellos, por último se hace la distancia euclidiana con los vectores resultantes de las operaciones anteriores:

```
def similFiles(file1, file2):
    countWords1 = countWords(file1)
    countWords2 = countWords(file2)
    keys_1 = set(countWords1.keys())
    keys_2 = set(countWords2.keys())
    inter = keys_1 & keys_2
    vect1 = {}
    vect2 = {}
    if inter.__len__()==0:
        return 100
    else:
        for w in inter:
            vect1[w] = countWords1[w]
            vect2[w] = countWords2[w]
        return euclideanDistance(vect1
            , vect2)
```

- Se implementa la *distancia euclidiana* [2] entre 2 documentos con la fórmula:

$$d_E = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

Algoritmo:

```
def euclideanDistance(vect1, vect2):
    sum = 0
    for cont in vect1:
        sum += math.pow((vect1[cont] -
            vect2[cont]), 2)
    sum = math.sqrt(sum)
    return sum
```

- Se desarrolla la función *distCent* que se encarga de hallar la distancia entre centroides y los promedios de distancias de los documentos, en otras palabras, se produce una matriz clusters la cual es el resultado de la resta entre los centroides y los promedios de distancias; estos promedios de distancias están almacenados en un arreglo.

```
def distCent(filas, columnas, clusters
    , centroides):
    for c in range(filas):
        for j in range(columnas):
            clusters[c][j]= math.fabs(
                distAverage[j]-
                centroides[c])
    return clusters
```

- La función *minor* se encarga de seleccionar el índice del menor valor en cada una de las columnas de la matriz clusters, la cual es retornada por la función *distCent*. Cada uno de estos índices se almacenan en un arreglo llamado vMinor y son los que ayudan a identificar a qué cluster pertenece cada documento.

[1] stop words: palabras que no tienen sentido por sí solas

[2] Tomado de https://stackoverflow.com/questions/21107505/word-count-from-a-txt-file-program compartido por Paolo Forgia

```
def minor(clusters, k, filas):
    vMinor = []
    minor = 0
    aux = 0
    for j in range(filas):
        for i in range(k):
            if i==0:
                minor=clusters[i][j]
                aux=i
            elif clusters[i][j] <
                minor:
                minor=clusters[i][j]
                aux=i
        vMinor.insert(j, aux)
    return vMinor
```

- La función *K-Means* [7] retorna los nuevos centroides, los cuales son hallados a partir de la matriz clusters. Dicho de otra forma, en cada fila del cluster está la resta entre el centroide y cada una de las distancias promedio de los documentos del dataset, luego estas distancias obtenidas se agrupan en los cluster correspondientes de acuerdo a los índices almacenados en el arreglo vMinor. Por último se realiza el promedio de las distancias que hay en cada cluster y este resultado determinará los nuevos centroides.

```
def kmeans (clusters, k, filas):
    vMinor = minor(clusters, k, filas)
    centroides = []
    for i in range(k):
        sum = 0
        for j in range(vMinor.__len__
            ()):
            if vMinor[j]==i:
                sum += distAverage[j]
        if vMinor.count(i) !=0:
            prom = sum / vMinor.count(
                i)
            centroides.insert(i, prom)
        else:
            centroides.insert(i, 0)
    return centroides
```

- Como se explicó anteriormente en el Análisis y Diseño mediante PCAM, para paralelizar el código se utiliza la librería *MPI*, con la cual se distribuyen las tareas en diferentes procesadores con el fin de reducir los tiempos durante el análisis de cada uno de los documentos con los demás documentos del dataset.

  A continuación se observa la tarea que realiza el *rank1, rank2, rank3* y *rank4* en paralelo, donde cada uno de ellos trabaja con la misma matriz pero sobre diferentes porciones de código, es decir, cada *rank* calcula la distancia entre documentos y también calcula el promedio de las filas sobre $\frac{1}{4}$ de la matriz:

```
if rank == 1:
    data1 = comm.recv(source=0)
    ds1 = distAverage
    for i in range(distances.__len__()
        /4):
        sum = 0
        for j in range(distances.
            __len__()):
            if i != j:
                distances[i][j]=
                    similFiles(filelist
                    [i],filelist[j])
                sum += distances[i][j]
        prom = sum / j
        ds1.insert(i, prom)
    data1 = distances[0:distances.
        __len__()/4]
    comm.send(data1, dest=5)
    comm.send(ds1, dest=6)
```

Para los demás *rank* lo único que cambia es el rango de la matriz sobre el que se va a trabajar, es decir, sobre otro $\frac{1}{4}$ de la matriz.

El *rank5* se encarga de juntar cada una de las porciones de la matriz que calcularon los anteriores *ranks*, lo que da como resultado la matriz de distancias entre documentos y que será usada para formar un arreglo de promedio de distancias entre documentos:

```
if rank == 5:
    data1 = comm.recv(source=1)
    data2 = comm.recv(source=2)
    data3 = comm.recv(source=3)
    data4 = comm.recv(source=4)
    distances = data1+data2+data3+
        data4
    fn = time()
    te = fn - ini
    print "tiempo de ejecucion ", te
```

El *rank6* se encarga de juntar cada una de las porciones de los promedios hallados por los *ranks* anteriores, lo que da como resultado un arreglo de promedio de distancias. Mediante la resta entre los centroides y cada promedio del arreglo de promedio de distancias, se forma la matriz clusters mencionada anteriormente, por medio de la cual se realiza todo el proceso de *k-means* hasta que los centroides converjan. Una vez estos converjen, se muestra el resultado final con las distancias agrupadas en los clusters correspondientes.

```
if rank == 6:
    ds1 = comm.recv(source=1)
```

```
ds2 = comm.recv(source=2)
ds3 = comm.recv(source=3)
ds4 = comm.recv(source=4)
distAverage = ds1+ds2+ds3+ds4

print "distAverage ", distAverage
for i in range(k):
    a = randint(1,distAverage.
        __len__()-1)
    if i == 0:
        centroides.insert(i,
            distAverage[a])
    else:
        while (centroides.
            __contains__(
            distAverage[a]))== True
            :
            a = randint(1,
                distAverage.__len__
                ()-1)
        centroides.insert(i,
            distAverage[a])

print "centroides ", centroides

for i in range(20):
    print "\n"
    clusters = distCent(k,
        distAverage.__len__(),
        clusters, centroides)
    aux = centroides
    centroides = kmeans(clusters,
        k, distAverage.__len__())
    print "estos son los nuevos
        centroides"
    for i in range(k):
        print centroides[i]
    if aux == centroides:
        break
print "\n"
print "estas son las distancias
    entre documentos"
for i in range(k):
    print clusters[i]

vMinor = minor(clusters,k,
    distAverage.__len__())
print "\n"
print vMinor
for i in range(k):
    print "\n"
    print "En El Cluster ", i
    for j in range(vMinor.__len__
        ()):
        if vMinor[j]==i:
            print distAverage[j]
```

VI. Análisis de resultados

En la siguiente tabla se muestran los resultados obtenidos en distintas ejecuciones. Cada una de las columnas se interpreta asi:

- Dataset - Conjunto de documentos utilizados
- Documentos - Número de documentos en el dataset
- Serial - Tiempo de ejecución (número entero) en segundos con el algoritmo serial
- Paralelo - Tiempo de ejecución (número entero) en segundos después de paralelizar el código

Dataset	Documentos	Serial(s)	Paralelo(s)
Gutenberg	100	1922	1091
Gutenberg	70	1197	680
Gutenberg	50	513	326
Gutenberg	30	165	107
Gutenberg	20	74	47

* Todas las ejecuciones se realizaron con el dataset de Gutenberg, 7 procesadores y un k aleatorio entre 8 y 12.

VII. Conclusiones

- Se logra comprender las bases para la programación paralela, ya que aunque en este proyecto sólo se utilizaron las funciones de *recv()* y *send()*, también se consigue entender el funcionamiento de otras importantes funciones que ofrece **MPI** [6] como *bcast(), gather()* y *scatter()*.

- Con los resultados obtenidos se pudo tener un acercamiento a lo que es Computación de Alto Rendimiento, ya que se observa una gran mejora luego de paralelizar el código, lo que nos lleva a pensar que si se utilizan las estrategias, herramientas e infraestructura adecuada en problemas mucho más grandes donde se requieran muchos recursos computacionales, éstos se pueden resolver en tiempos considerables.

- Se pudo observar limitaciones en cuanto a nivel de algoritmia y hardware, dado que con respecto a lo primero se observa que si no se emplean las estrategias o algoritmos adecuados, el programa va a tomar mayor tiempo; igualmente a nivel de hardware, el rendimiento de las aplicaciones depende de la calidad de las máquinas, es decir, entre mejor sea ésta, mejor se va a comportar o por el contrario el rendimiento va a disminuir.

- Se logra comprender la ventaja de utilizar la metodología PCAM durante el análisis y diseño

de algoritmos paralelos, puesto que observamos que ésta nos ayudó a identificar con mayor facilidad la distribución de tareas entre los diferentes procesadores.

- Se pudo observar la mejora de un algoritmo paralelo frente a uno serial, pues el tiempo de ejecución se redujo considerablemente.

Referencias

[1] Anna Huang. *Similarity Measures for Text Document Clustering.* Department of Computer Science. The University of Waikato, Hamilton, New Zealand.

[2] Distancia euclidiana - Wikipedia, la enciclopedia libre. *Distancia euclidiana* Enero 26, 2016
`https://es.wikipedia.org/wiki/Distancia_euclidiana`

[3] Stop Words
`http://snowball.tartarus.org/algorithms/english/stop.txt`

[4] Ricardo Moya. *K-means en Python y Scikit-learn, con ejemplos.* Mayo 29, 2016
`https://jarroba.com/k-means-python-scikit-learn-ejemplos/`

[5] K-means - Wikipedia, la enciclopedia libre. *k-means.* Agosto 25, 2017
`https://es.wikipedia.org/wiki/K-means`

[6] Lisandro Dalsin. *Tutorial – MPI for Python 2.0.0 documentation.* 2015
`https://mpi4py.scipy.org/docs/usrman/tutorial.html`

[7] AVINASH YADAV. *Kmeans Clustering Solved Example with Java Code.* Abril 15, 2017
`https://www.youtube.com/watch?v=JSJIolgFYqw&t=849s`