Tópicos especiales en Telemática
Similaridad de Textos

Juan David Pérez - Andrés Atehortúa
jperezp2@eafit.edu.co aatehor4@eafit.edu.co
Escuela de ingenierías
Universidad EAFIT

***Abstract*— En este documento se muestra el reporte investigativo con respecto a la práctica de computación de alto rendimiento, en el cual se abordara el proceso y conclusiones de dicho proyecto.**

keywords– HPC, Paralelización, Kmeans, Similaridad, Vectorización, Coseno, *Cluster*

I. INTRODUCCIÓN

A partir de este documento se busca plantear los resultados del proyecto 3 de HPC en la materia Tópicos especiales en telemática, donde se mostrará como la paralelización de algoritmos puede mejorar el desempeño y reducir el tiempo del mismo en el procesamiento de datos y el análisis de los mismos. El proyecto básicamente consiste en implementar un método de similaridad y kmeans para trabajar con documentos en el lenguaje de programación python, en este caso se implementó *cosine similarity* como método de similaridad.

II. MARCO TEÓRICO

Este proyecto comprende varios aspectos que se mencionaran a continuación, los cuales se integraron para llevar a cabo su desarrollo; uno de estos es un algoritmo de similaridad el cual se encargara de hallar la frecuencia con la cual se repiten ciertas palabras entre documentos. Para este caso se eligió *cosine similarity*, el cual es una medida entre 2 vectores en un espacio; dicha medida se obtiene al evaluar el coseno del ángulo entre dichos vectores. La función coseno esta dada por:

$$cos(\alpha) = \frac{A*B}{||A||*||B||}$$

Como se menciona anteriormente esta medida se aplica entre vectores para lo cual se deberá convertir los documentos en un tipo de matriz tf-idf (que despues se descompone en vectores), de tal manera que se pueda aplicar dicho algoritmo de similitud, para la modelación de un documento a un espacio vectorial lo primero que se hace es crear un diccionario con los términos presentes en el documento (eliminando las *stopwords*), posteriormente se utiliza la frecuencia de cada termino en el vocabulario que posee cada documento para la representación del mismo en el espacio vectorial, definición a partir de funciones

$$tf(t,d) = \sum_{x \in d} fr(x,t)$$

Donde d es el documento, t el número de veces que el termino esta en el documento y $fr(x,t)$ está definida como:

$$fr(x,t) = \begin{cases} 1 & \text{if } x = t \\ 0 & otherwise \end{cases}$$

Por lo tanto, la creación del vector que representa el documento está dada por

$$\vec{v_{d_n}} = (tf(t_1,d_n)), (tf(t_2,d_n)), ..., (tf(t_n,d_n))$$

Una vez se tiene el vector, el termino tf-idf (*term frequency – inverse document frequency*) weight busca solucionar el problema de que términos realmente le aportan al texto ya que en muchos casos los que más se repiten no son los que más aportan o son los más importantes. Para esto primero debemos normalizar el vector, a partir de la formula

$$\hat{v} = \frac{\vec{v}}{|||\vec{v}||_p}$$

Y luego aplicar el idf, el cual está definido como

$$idf(t) = log \frac{|D|}{1+|d:t \in d|}$$

Donde |D| es el espacio de documentos definido por D = d1,d2,...,dn, y d:t E d es el número de documentos donde el termino t aparece, cuando la función tf(t,d) <> 0, de este proceso sale un vector el cual se multiplica con una matriz identidad y posteriormente esta matriz se multiplica con una que sale en las primeras etapas del proceso con la frecuencia de los términos en cada documento. Después de esto se tiene una matriz tf-idf con el conjunto de documentos a ser comparados

luego de tener esto previamente calculado se procede a aplicar el algoritmo de k-means; este algoritmo es un método de cuantificación vectorial el cual tiene como objetivo dividir un conjunto de observaciones en n sub-grupos o *clusters*. Todo ello con el fin de agrupar los documentos que comparten una mayor similitud en n grupos, esto se puede llevar a aplicaciones de recomendaciones, buscadores, entre otras.

El algoritmo de k-means, requiere del número de *clusters* y un *dataset*, primero el algoritmo establece unos puntos centrales para los k *clusters*, que pueden ser aleatorios o partiendo del conjunto de datos, el algoritmo opera en

dos pasos básicamente, define los centros y posteriormente asigna los datos más cercanos a cada uno de estos puntos centrales basado en la distancia euclidiana por defecto, en nuestro caso trabajamos con *cosine similarity* al cual se resta 1 para hallar la distancia.

III. Análisis y diseño mediante PCAM incluyendo algoritmos y estructuras de datos y rendimiento analítico de la solución

Para la realizar la paralelización del proyecto se procedió a estudiar y analizar que partes del código serial era posible dividir; se vio que tareas o procedimientos tenían mayor comunicación entre si para realizar su proceso en un solo *rank* o núcleo. Se utilizo una arquitectura *master-slave* en la cual el *master* le envía una serie de tareas a los nodos esclavos para luego reunir la información que fue procesada.

En la sección de implementación se muestra en detalle el código requerido para lo que se menciono anteriormente, es importante resaltar que parte de la implementación es gracias a *sergeio* usuario de github, el cual posee un proyecto de *text clustering* en su cuenta.

IV. Implementación

La implementación de este proyecto se divide en dos partes (serial y paralelo), se anexa parte del desarrollo para tener una idea general de su implementación

A. Serial

Vectorizer

```
def cluster_paragraphs(paragraphs, num_clusters, filecontent):
    word_lists = make_word_lists(paragraphs)
    word_set = make_word_set(word_lists)
    word_vectors = make_word_vectors(word_set, word_lists)

    paragraph_map = dict(zip(map(str, word_vectors), filecontent))

    k_means = KMeans(num_clusters, word_vectors)
    k_means.main_loop()
return translator(k_means.clusters, paragraph_map)
```

K-means

```
def update_clusters(self):
        def closest_center_index(vector):
            similarity_to_vector = lambda center: similarity(center,vector)
            center = max(self.centers, key=similarity_to_vector)
            return self.centers.index(center)

        self.clusters = [[] for c in self.centers]
        for vector in self.vectors:
             index = closest_center_index(vector)
             self.clusters[index].append(vector)

    def update_centers(self):
        new_centers = []
        for cluster in self.clusters:
            center = [average(ci) for ci in zip(*cluster)]
            new_centers.append(center)

        if new_centers == self.centers:
            return False

        self.centers = new_centers
        return True

    def main_loop(self):
        self.update_clusters()
        while self.update_centers():
self.update_clusters()
```

Cosine

```
def similarity(v1, v2):
  return dot_product(v1, v2) / (magnitude(v1) * magnitude(v2) + .00000000001)
```

B. Paralelo

Vectorizer

```
def word_frequencies(word_vector):

    num_words = len(word_vector)
    frequencies = defaultdict(float)

    if rank == 0:
        data = word_vector
    else:
        data = None
    data = comm.bcast(data,root = 0)

    for word in data:
        frequencies[word] += 1.0/num_words
    return dict(frequencies)
```

K-means

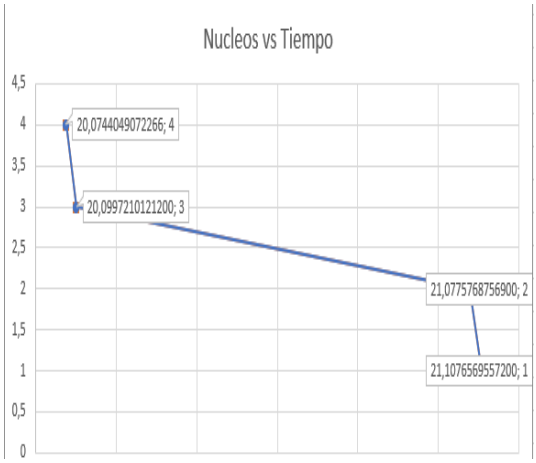
```
def update_clusters(self):

        def closest_center_index(vector):
            similarity_to_vector = lambda center: similarity(center,vector)
            center = max(self.centers, key=similarity_to_vector)
            return self.centers.index(center)

        self.clusters = [[] for c in self.centers]

        if rank == 0:
            info = self.clusters
        else:
            info = None
        info = comm.bcast(info,root=0)
        pos = rank
        while pos < len(self.vectors):
            vector = (self.vectors)[pos]
            index = closest_center_index(vector)
            if vector != None:
                info[index].append(vector)
            pos += size
        comm.Barrier()
self.clusters = info
```

V. Análisis de Resultados

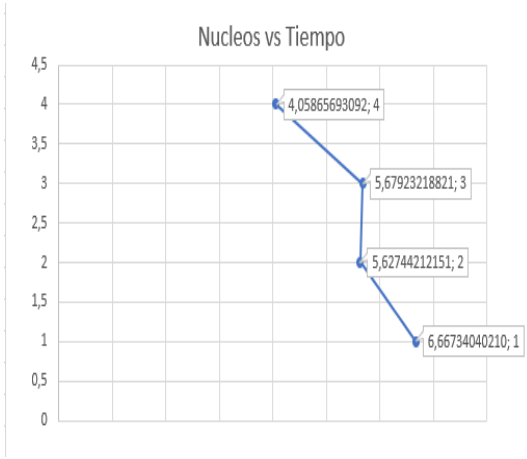
A continuación se muestra una tabla que ilustra el como a medida que la cantidad de núcleos decrece, el tiempo de ejecución del programa aumenta en esta gráfica se muestra el tiempo que tomo la ejecución del programa la cual esta medida en segundos y al lado de esta medida esta la cantidad de núcleos (donde el valor de 1 corresponde a la ejecución serial). Para dichos resultados se hizo uso de los siguientes parámetros:

- 20 documentos del *dataset* de **Gutenberg**
- Un valor K igual a 10 (Gráfica 1)
- Un valor K igual a 4 (Gráfica 2)

Gráfica 1



Gráfica 2



Nota: Todos los valores de salida están expresados en segundos

VI. Conclusiones

Mediante este proyecto se puede observar que la programación en paralelo a menudo escala con el tamaño del problema, por lo cual puede resolver problemas más grandes. En general se podría decir que es un medio para proporcionar concurrencia en problemas de computación, ademas de permitir un mayor *thoughput* y latencia.

Pero no se puede decir que todo son ventajas ya que también tiene ciertas desventajas, bien sean para la persona que realiza el programa o para el *hardware* o la aplicación en general, algunas de estas son:

- Complejidad
- Demanda de recursos
- Portabilidad

Referencias

[1] «Python: tf-idf-cosine: to find document similarity.» 2017, de Stack Overflow Sitio web: https://stackoverflow.com/questions/12118720/python-tf-idf-cosine-to-find-document-similarity.

[2] Thomas. (2017). «Python: tf-idf-cosine: to find document similarity.» 2017, de Stack Overflow Sitio web: https://stackoverflow.com/questions/41504454/calculate-cosine-similarity-of-all-possible-text-pairs-retrieved-from-4-mysql-ta.

[3] Coursera. (2017). «Distance metrics: Cosine similarity.» 2017, de Coursera Sitio web: https://www.coursera.org/learn/ml-clustering-and-retrieval/lecture/yyegc/distance-metrics-cosine-similarity.

[4] «Set - Unordered collections of unique elements,» Python, 2017. [En línea]. Disponible en: https://docs.python.org/2/library/sets.html. [Último acceso: 2017].

[5] «Google's Python Class,» Google, [En línea]. Disponible en: https://developers.google.com/edu/python/. [Último acceso: 2017].

[6] «Python: List Comprehensions,» [En línea]. Disponible en: http://www.secnetix.de/olli/Python/list_comprehensions.hawk. [Último acceso: 2017].

[7] «Similarity between two text documents,» stackoverflow, [En línea]. Disponible en: https://stackoverflow.com/questions/8897593/similarity-between-two-text-documents. [Último acceso: 2017].

[8] «Python: tf-idf-cosine: to find document similarity,» stackoverflow, [En línea]. Disponible en: https://stackoverflow.com/questions/12118720/python-tf-idf-cosine-to-find-document-similarity. [Último acceso: 2017].

[9] C. Perone, «Machine Learning :: Text feature extraction (tf-idf) – Part I,» 2011. [En línea]. Disponible en: http://blog.christianperone.com/2011/09/machine-learning-text-feature-extraction-tf-idf-part-i/.

[10] C. Perone, «Machine Learning :: Text feature extraction (tf-idf) – Part II,» 2011. [En línea]. Disponible en: http://blog.christianperone.com/2011/10/machine-learning-text-feature-extraction-tf-idf-part-ii/.

[11] C. Perone, «Machine Learning :: Cosine Similarity for Vector Space Models (Part III),» 2013. [En línea]. Disponible en: http://blog.christianperone.com/2013/09/machine-learning-cosine-similarity-for-vector-space-models-part-iii/.

[12] «Pairwise metrics, Affinities and Kernels,» scikit learn, [En línea]. Disponible en: http://scikit-learn.org/stable/modules/metrics.html#cosine-similarity.

[13] E. Fox, «k-means as coordinate descent,» University of Washington, [En línea]. Disponible en: https://www.coursera.org/learn/ml-clustering-and-retrieval/lecture/Fb58J/k-means-as-coordinate-descent.

[14] A. Trevino, «Introduction to K-means Clustering,» Data science, [En línea]. Disponible en: https://www.datascience.com/blog/k-means-clustering. [Último acceso: 2017].

[15] Sergeio, «Text clustering,» github, [En línea]. Disponible en: https://github.com/sergeio/text_clustering