Parallel k-Means++ Analysis

Patrick Williams
Wake Forest University
Winston-Salem, NC
willpj18@wfu.edu

Esteban Murillo
Wake Forest University
Winston-Salem, NC
murie18@wfu.edu

ABSTRACT

k-Means is one of the prime examples of the algorithms that are very simple yet very effective. Motivated by some real-life examples we had in previous classes, we wanted to improve the efficiency of the code we implemented in Python. The improvement of the code comes in two stages, first, we used a more efficient programming language (C++) that allows the algorithm to run faster. Then, the second stage corresponds to the parallelization of the code, which C++ allows with the help of the `OpenMP` API, something that Python does not have. Finally, we made use of the `DEAC Cluster` in order to run all our experiments.

Author Keywords
clustering; k-Means; parallel

CCS Concepts
•Information systems → Clustering; •Theory of computation → Shared memory algorithms;

INTRODUCTION

The *k-Means* clustering algorithm is one of the most widely used clustering algorithms because of its ease of use and effectiveness. As a result, there have been many attempts to parallelize the k-Means algorithm in the literature. Almost all the research we found has to do with real-life scenarios, where they are actually trying to perform cluster analysis on real data at the same time that they try to parallelize it in order to get results faster. Our project is different to theirs in the sense that we are not focusing that much on the data that we are clustering, instead of that, we focus ourselves on the classification of the data and how to make it faster. Since the data is not the center of our research, we only extract certain features from our dataset (Java source code). Later, we found that there were more efficient ways to deal with this, discussed in the ”Future work” section.

MOTIVATION

Our inspiration for this project comes from our Data Mining class, where we have been implementing several algorithms with some real-life applications. One that called our attention was precisely the *k-Means* algorithm that helped us perform cluster analysis on different datasets. Very soon we realized that the algorithm performs very well, but there are several “bottlenecks” to it. One example, is the feature extraction, which we realized was taking a long time. That is why we decided to parallelize the feature extraction.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

OVERVIEW OF THE *K-MEANS* ALGORITHM

k-Means algorithm is a type of unsupervised learning. The goal is to make cluster analysis so that we end up with different groups. This algorithm is going to place elements with similar characteristics together. It is important to extract or get relevant information from these elements so that the cluster analysis is more accurate. The way in which the aforementioned algorithm works is as follows:

- From all the options, choose k different samples and use them as centroids
- Compute the distance from each of the other samples to the centroids. This is usually done with *Euclidean Distance* or *Manhattan Distance*. It’s worth noting that in order to calculate distance, some features from the samples need to be extracted beforehand
- Assign each of the elements to the cluster that is represented by the centroid with the shortest distance to it
- Repeat the process until one of the following conditions is met:
  - Clusters do not change
  - Centroids do not change
  - Quality does not longer improve
  - Maximum amount of iterations is reached

OVERVIEW OF OUR IMPLEMENTATION

The main parameters of our implementation of *k-Means* are the following:

- d: The dimension of feature vectors
- n: The number of feature vectors
- k: The number of clusters to create
- i: The maximum number of iterations for the clustering algorithm

The first step in implementing a *k-Means* clustering algorithm is to choose the initial centroids. This step can be done in a number of ways, and our choice was the *k-Means++* algorithm, which pseudo-randomly chooses centroids from the set of data points with weighted probabilities based on the distances between points and existing centroids. This way, points that are farther away from existing centroids are more likely to be chosen. The implementation of the algorithm is shown in Algorithm 1.

The algorithm iterates once for each cluster, which includes iterating over all points and comparing to all existing centroid, which is at most k. The Euclidean distance is calculated for each pair of points, comparing d features. As a result, the complexity of the *k-Means++* algorithm we implemented is $O(k^2nd)$.

The *k-Means* algorithm we implemented is shown in in Algorithm 2. It iterates for at most i iterations, operating on each point, which iterates through every centroid and calculates the distance between them each time to assign it to a cluster. The complexity of this algorithm is $O(inkd)$, meaning the overall complexity of these two algorithms is $O(k^2nd + inkd)$.

Algorithm 1 Choose initial centroids for clustering using *k-Means++*

```
choose first centroid at random
for each remaining centroid do
    for each feature vector $v$ in parallel do
        for each existing centroid $c$ do
            calculate distance between $v$ and $c$
        end for
        find the distance $d$ between $v$ and its closest centroid
    end for
    calculate $s$ = sum of all $d^2$ values
    for each feature vector $v$ in parallel do
        calculate the weight of $v$: $w = d^2/s$
    end for
    randomly choose centroid with weighted probabilities $w$
end for
```

Algorithm 2 *k-Means* Clustering

```
for $i$ iterations do
    for each feature vector $v$ do
        for each centroid $c$ in parallel do
            calculate distance between $v$ and $c$
        end for
        assign $v$ to the closest centroid
    end for
    for each cluster in parallel do
        calculate centroid of cluster
    end for
end for
```

FACED OBSTACLES

One of the several obstacles we faced was the "translation" of the code. We found that things were not as easy to implement as the first time when written in Python. This is mainly because the data cannot be that easily manipulated in C++, so several utilities functions had to be written in order to handle the data correctly. On the other hand, parallelization of the code does not come easy and we had to put some thought to the actual parallelization of the algorithm, since there are some steps that require some kind of logic in order to parallelize. Finally, the last obstacle that we faced is the dataset we used. Of course that the *k-Means* is agnostic to the data that it classifies, but we still wanted to make some relevant analysis, so we decided to perform the cluster analysis on source code obtained from the Internet. In the beginning, we ran our code with a dataset comprising of 100 files, but of course this proved not to be big enough in order to see any significant speed-ups, that is why we decided to look somewhere else for a bigger dataset, one that allowed us to have a more realistic approach with real-life data.

BACKGROUND WORK

After some research we found out that the *k-Means* is not anything new and that it is a topic that has been researched a lot already. We took advantage of it and read some of the research that follow similar ideas as our project. For example, [4, 2] are two reseach papers that make use of the algorithm in question. But we feel that there are two papers in particular that do an interesting analysis on top of having some real-life applications. The first one is [1], which has a more in-depth analysis and even parallelized their code with both interfaces introduced in our class (`OpenMP` and `MPI`). According to the results presented in this paper, the obtained speed-ups are very promising, getting close to 2x speed-ups every time they doubled to number of threads. Lastly, and even though this is not the main focus of our project, the reason they were performing cluster analysis was to be able to process hyperspectral images, a type of image shot by specialized cameras that are very large in size and that in some cases can go over 500mb.

We also came across other type of interesting literature, such as [3], where they propose a very interesting approach to how to cluster. Their approach consists on splitting the dataset into smaller datasets and let the threads handle the smaller datasets. In this way, the outliers can be handled in a better way and they even claim their accuracy goes up. Then again, this is not something that could be said for every dataset, but for the ones that meet certain conditions.

Strong scaling results

The results of our strong scaling experiment is shown in Table 1 and Figure 1. Results showed a maximum speedup of about 4 with 8 processors, then after using more than 8 processors speedup and efficiency dropped. This could be because of a performance bug, but may also have to do with the overhead involved with spawning the threads.

Weak scaling results

Our weak scaling experiment results are shown in Figure 2 and Table 2. These results showed poorer performance for lower thread counts, showing a significant discrepancy between observed speedup and efficiency vs. ideal values.

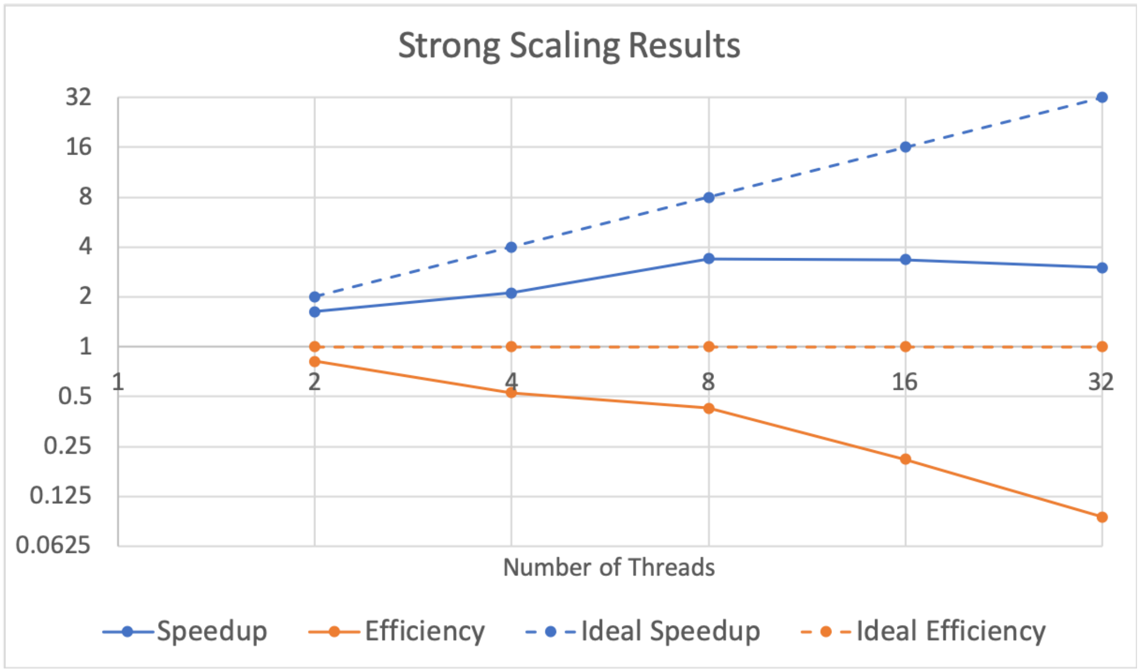

Figure 1. Strong scaling results with *k = 16* and *NumFiles = 13000*

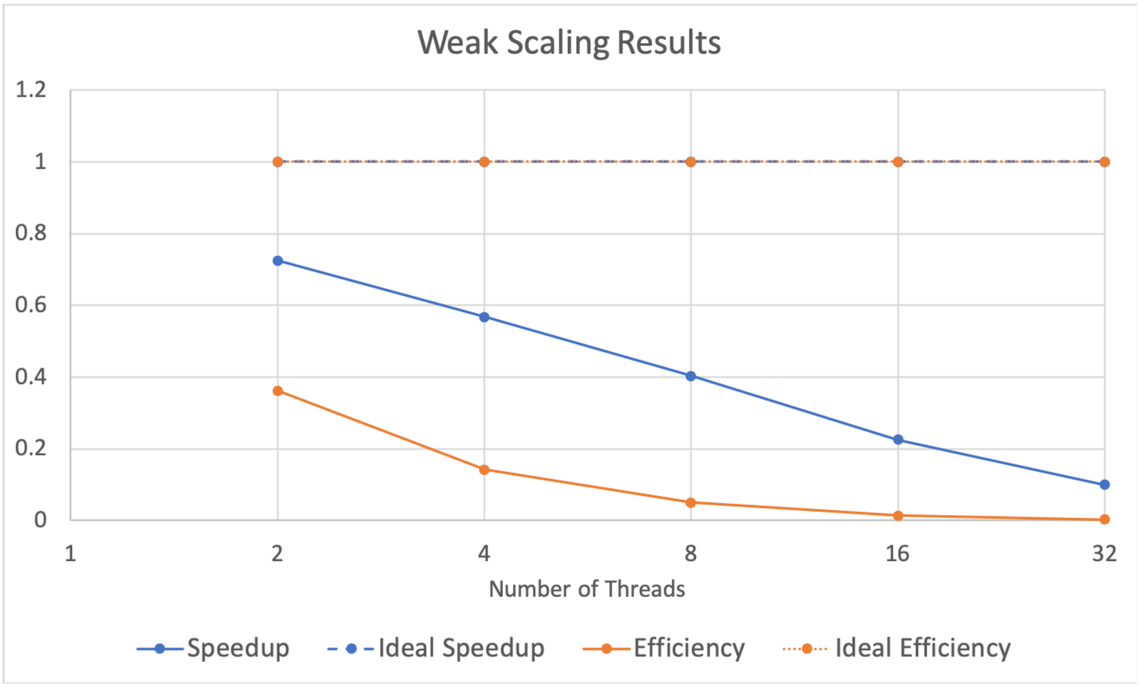

Figure 2. Weak scaling results with *k = 16*

Threads	Time (s)	Speedup	Efficiency
1	6.40425	N/A	N/A
2	3.92769	1.63053	0.81526
4	3.028	2.11500	0.52875
8	1.88471	3.39800	0.42475
16	1.91569	3.34305	0.20894
32	2.13241	3.00329	0.09385

Table 1. Strong scaling results with *k = 16* and *NumFiles = 13000*

Number of Files	Threads	Time (s)	Speedup	Efficiency
500	1	0.26666	N/A	N/A
1000	2	0.36804	0.72456	0.36228
2000	4	0.47010	0.56725	0.14181
4000	8	0.66115	0.40333	0.05041
8000	16	1.18502	0.22503	0.01406
13000	32	2.69181	0.09906	0.00309

Table 2. Weak scaling results with *k = 16*

FUTURE WORK

As part of our future work we would like to get more significant speed-ups for our code. We also believe that changing the data structures used to model the data could also result in some significant improvement overall. And while we are aware that this will not change our efficiency or speed-ups, we feel this might improve efficiency, not in terms of parallelism but in terms of the code's structure. Moreover, we would like to experiment on the choosing of k to see which is the value that yields the most accurate results. But then again, that is something we can only achieve if we perform the cluster analysis on real data and not randomized one.

NOTES

It is important to mention that the code was ran using the `DEAC Cluster` that has been available to us. This of course, has proven to be a very valuable resource since we did not have all the necessary tools in order to run our code using `OpenMP`. In order to see the implementation of the code, refer to our GitHub repository, provided in [5].

CONCLUSION

From implementing *k-Means++* we have learned that we could have been a little bit more efficient about our project just by focusing more on the actual clustering. This means, that we now realize that we spent a significant amount of time trying to optimize something that was not all that important. Also, as part of our future work, we would like to further test the algorithm itself by using pre-made or randomized values instead of focusing on feature extraction, since the clustering algorithm itself does not rely on the extraction of the data. By doing this we could easily simulate large datasets without the hassle of getting the dataset or extracting the features of the files. Either way, we are pleased with the observed results. We got significant speed-ups for the *k-Means++* that could further be improved by changing the data structures that were used in our project from vectors to arrays.

REFERENCES

[1] Thomas L. Boggs. 2010. Parallel Implementation of the k-means Clustering Algorithm for Unsupervised Classification of Hyperspectral Imagery Release 1 . 0.

[2] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. 2002. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24, 7 (July 2002), 881–892. DOI:http://dx.doi.org/10.1109/TPAMI.2002.1017616

[3] D. S. B. Naik, S. D. Kumar, and S. V. Ramakrishna. 2013. Parallel processing of enhanced K-means using OpenMP. In *2013 IEEE International Conference on Computational Intelligence and Computing Research*. 1–4. DOI: http://dx.doi.org/10.1109/ICCIC.2013.6724291

[4] Emanuele Torti, Giordana Florimbi, Francesca Castelli, Samuel Ortega, Himar Fabelo, Gustavo Marrero Callico, Margarita Marrero-Martin, and Francesco Leporati. 2018. Parallel K-Means Clustering for Brain Cancer Detection Using Hyperspectral Images. *Electronics* 7 (10 2018), 283. DOI: http://dx.doi.org/10.3390/electronics7110283

[5] P. Williams and E. Murillo. 2019. Parallel K-Means. https://github.com/PWilliams520/Parallel_K-Means. (2019).