TU/e
Eindhoven University of Technology

2IMA35 Massively Parallel Algorithms

Study number: 1750445, Jonas Brøndum
Study number: 1908995, Wai-Hong Anton Fu

Report 2
Coreset clustering on experimental data

January 16, 2023

Contents

1 Introduction

The purpose of this report is to study the performance of coreset clustering under different conditions, including the type of dataset, the size of the dataset, and the number of clusters.

A coreset clustering algorithm for parallel and non-parallel computation has been developed to further analyze any differences in performance. The datasets also include two different images that will be segmented using the same coreset clustering.

The code developed for this report is on this github repository.

2 Datasets

The datasets used are the images *lena* [2] and *baboon* [1]. Furthermore, a dataset consisting of generated blobs using the Scikit-learn library is used.

The synthesized data from the Scikit-learn library is varied in the sample size $[n \times 2]$.

$$n = [1000, 5000, 10000, 50000, 100000, 500000, 1000000] \tag{1}$$

The images are both RGB-images with dimensions $[512 \times 512 \times 3]$ (see figures 1 and 2).

Figure 1: Image, *lena*

Figure 2: Image, *baboon*

3 Method

An algorithm for the coreset construction was developed to work in parallel or non-parallel.

The outline of the algorithm mostly follows the pseudocode specified in the lecture notes for lecture 6 about k-means.

3.1 How the (k, ϵ)-coreset algorithm works

From the data, the points are given initial weights of 1. An initial k centers are calculated with k-means++. These k centers along with the points and the weights are sent to a CoresetConstruction algorithm that computes the coreset using the technique discussed in the class:

1. For each center, balls (and subsequently annulus) are calculated and points within constructed grid cells for each ball are charged to a single representative point in that particular cell.

2. Then the points and weights are saved for the following round in the parallel version, where this process is repeated.
3. The final clustering is done by running k-means on the final coreset. These centers are then used to label the original point-set.

For the image segmentation, the RGB values for each pixel are treated as 3D coordinates, whereon the coreset is computed.

3.2 The parallellization

The parallelization was implemented using PySpark. Each element is randomly sent to one arbitrary machine, which is then run in a binary tree-like structure. The first x machines calculate the coreset for their given subset, then a new coreset is calculated on $x/2$ machines and so forth until the tree reaches its root. In this experiment, 4 machines are used (thus 3 rounds are used, with 4, 2 and 1 machines).

4 Experiments and results

Three experiments were carried out to analyze the performance of the implemented algorithm.

1. Pure image segmentation performance analysis
2. Running time of parallel versus non-parallel
3. Running times and hyper-parameter analysis of image segmentation, for both parallel (k, ϵ)-coreset and vanilla k-means clustering

4.1 Experiment 1

The purpose of this experiment is to investigate how the algorithm performs on image segmentation. The algorithm was run with different k as seen in figures 3, 4 and 5.

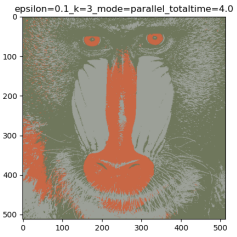

Figure 3: Image segmentation of *baboon* with $k = 3$

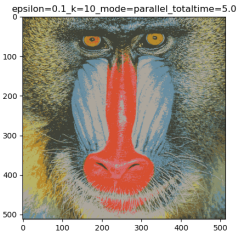

Figure 4: Image segmentation of *baboon* with $k = 10$

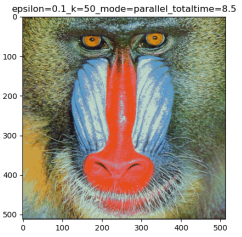

Figure 5: Image segmentation of *baboon* with $k = 50$

Firstly, it's clear that the image is segmented very well for any of the cluster sizes. Even for $k = 3$, the major characteristics of the image are a little unclear but still distinguishable. Additionally, it's seen that the gain in detail is diminished between $k = 10$ to $k = 50$

Secondly, the more cluster centers are implemented the more detail is added back to the image. At around $k = 50$ the image fully resembles the original (24-bit color) image.

Similarly, for *lena* the image segmentation can be seen in figure 6 for $k = 3$ and $k = 100$.

Furthermore, it's possible to look at the cost of the coresets that are used to segment both *baboon* and *lena* (fig. 7 and fig. 8).

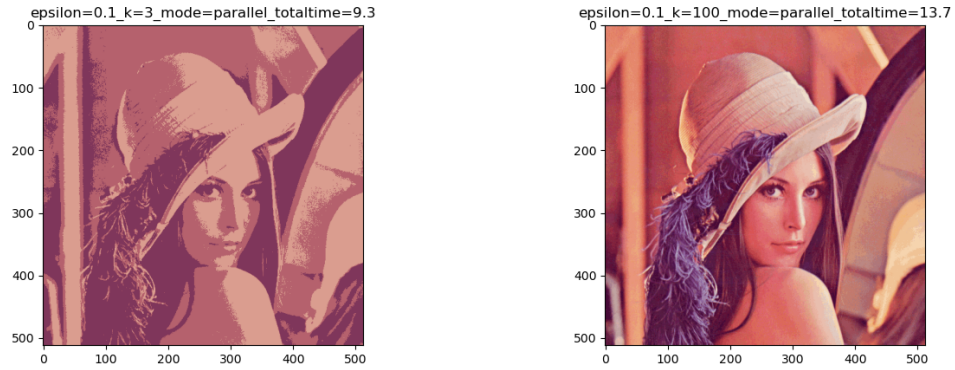

Figure 6: Image segmentation of the image, *lena*, with the parallel implementation of the coreset with $k = 3$ and $k = 100$ respectively.

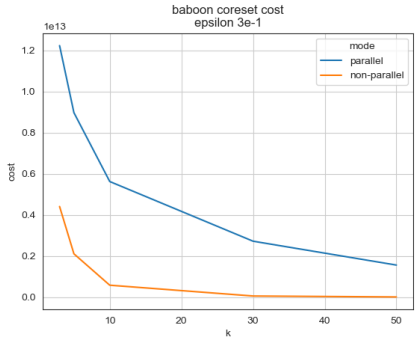

Figure 7: Cost comparison of the image *lena* between parallel and non-parallel.

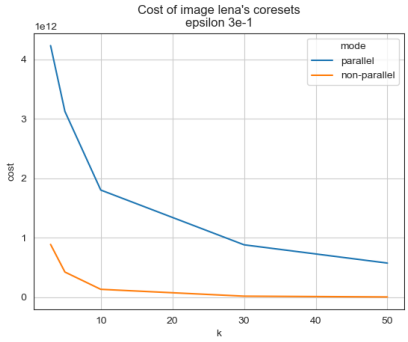

Figure 8: Cost comparison of the image *baboon* between parallel and non-parallel.

From the cost of both *baboon* and *lena* it's seen that as the amount of clusters k is increased the cost is decreased, which is to be expected.

Followingly, there are two important takeaways from the figures 7 and 8.

Firstly, the cost of *baboon* is approximately 3 times larger than the cost of *lena*. This is most likely due to the fact, that the color range of *lena* is smaller. *lena* is mostly a red-ish hue, while *baboon* includes many different colors. This suggests that the actual clusters that the colors represent are distributed more distinctly in *baboon*. This also means that clustering on *lena* is a more complex task.

Secondly, another important takeaway is that the cost of the non-parallel computation is about ∼3-4 times smaller than its parallel counterpart for each image. As described in section 3.2, the parallel algorithm runs in a tree-like structure, where each subpart of the coreset computed is merged with one another, therefore, it has a cumulative impact on ϵ_{total} (e.g. the total error). A way to reduce this discrepancy would be to adjust/decrease the ϵ for each round, to take this cumulative impact into account.

4.2 Experiment 2

The purpose of this experiment is to compare the parallelized (k, ϵ)-coreset to the non-parallelized coreset algorithm, figure 9 shows the running time for the two algorithms. It was expected to see a similar trend as to section 4.3, that the parallel is initially slower than non-parallel, but gradually

increases in efficiency as the sample size increases to some threshold. This was however not the case, as shown in figure 9. There are many reasons for this, perhaps the number of machines need to be tuned, or n-samples have to get really large for the effect to show. Some of the clusterings of this algorithm are shown in figures 10 and 11.

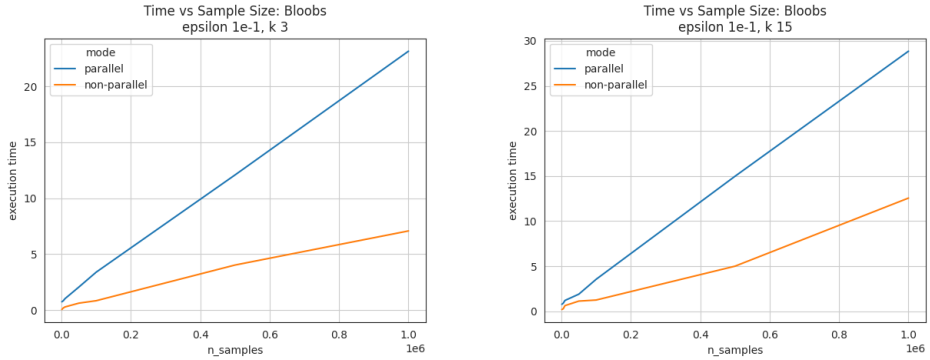

Figure 9: Running time against the sample size, for parallel (k, ϵ)-coreset and non-parallel coreset. These plot shows for $\epsilon = 0.1$ and $k = 3, k = 15$, respectively.

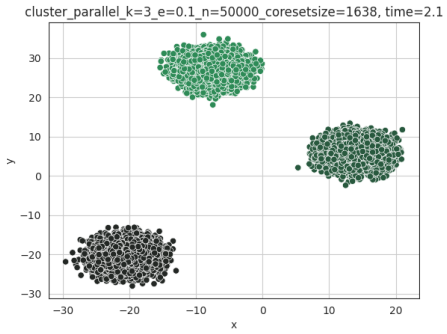

Figure 10: Blob clustering (parallel) with $k = 3$.

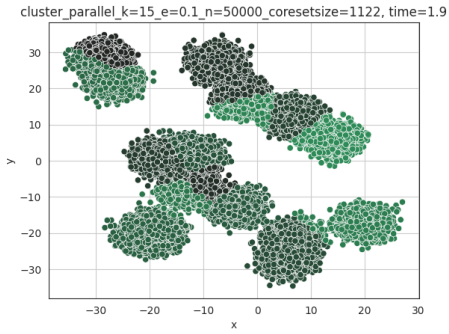

Figure 11: Blob clustering (parallel) with $k = 15$.

4.3 Experiment 3

The purpose of this experiment is to investigate the following in regard to image segmentation:

1. How the parallel (k, ϵ)-coreset performs, compared to not using the coreset (e.g. only using vanilla k-means).
2. How the (k, ϵ)-coreset algorithm performs on different datasets, *lena* and *baboon*.

The left image in figure 12 shows how the parallel (k, ϵ)-coreset performs against not using coreset at all. The running time for both (k, ϵ)-coreset and no-coreset increases linearly, note that the running time for parallel (k, ϵ)-coreset increase in a much smaller rate than no-coreset, s.t. it's faster than no-coreset when k is greater than 10.

Remember that k-means have a time complexity of $\mathcal{O}(nkd)$ per iteration and (k, ϵ)-coreset have $\mathcal{O}(\sqrt{n})$. Since (k, ϵ)-coreset reduces n, at some point, it is worth using it, despite the $\mathcal{O}(\sqrt{n})$ time consumptions, for this specific case, that point seems to be when $k = 10$

The right image in figure 12 shows the running time for *lena* and *baboon* over different epsilons. *Baboon* follows a trend that is expected, e.g. that the running time increases as epsilon gets larger. This makes

sense since the epsilon represents the error one is willing to tolerate, thus a higher epsilon should imply fewer grids when constructing the coreset, which should yield faster performance. However, *lena* does not follow this trend. This may be due to noise since *lena* does not seem to follow any specific trend.

What is interesting, however, is that *lena* seems to be much harder in general, to perform clustering on, in comparison to *baboon*. As the right image in figure 12 would suggest, *lena* might take up to double the time to perform image segmentation on, in comparison to *baboon*. The likely reason is, as briefly described in section 4.1, *lena* contains a smaller range of colors, which would result in cluster centers closer to each other, thus making it harder to cluster.

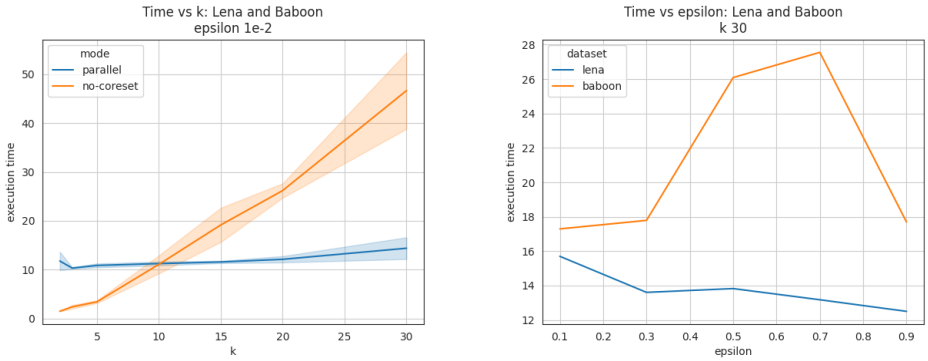

Figure 12: The left image is of the running time against k for parallel (k, ϵ)-coreset and vanilla k-means, for both *lena* and *baboon*. The right image is of running time against different epsilons for parallel (k, ϵ)-coreset, on two different datasets; *lena* and *baboon*.

5 Conclusion

In conclusion, the algorithm works very well in both image segmentation and the synthesized data (ie. blobs). Further investigation then showed that the parallel implementation didn't run faster than the non-parallel, which could be due to the sample size, poor implementation, or hyper-parameter tuning (for example the number of machines might be one important parameter). However, for different values of k, it was seen that the parallel (k, ϵ)-coreset algorithm seemed to outperform the vanilla k-mean algorithm as $k \geq 10$.

References

[1] Karri Chiranjeevi and Umaranjan Jena. Image compression based on vector quantization using cuckoo search optimization technique. *Ain Shams Engineering Journal*, 9, 10 2016.

[2] Wikipedia. Lenna, Jan 2023.