Performance Comparisons of the K-Means Algortihm

Mark Ward
Center for Data Science
New York University
maw627@nyu.edu

Abstract

The k-means algorithm is a widely used clustering technique. Here we will examine the performance of multiple implementations of the k-means algorithm in different settings. Our discussion will touch on the implementation of the algorithm in both python and C, and will also mention a 3rd party package for the k-means algorithm that is also written in C but provides python bindings. We will then turn to focus on the parallelization of the k-means algorithm and discuss both implementation and performance.

1. Introduction

The k-means algorithm is a well known method for partitioning n points in a d-dimensional space into k distinct clusters. The objective of the algorithm is to produce k cluster centers that minimize the sum of the squared distances between each point and its nearest cluster center. Solving this problem exactly is NP-hard, but k-means, also known as Lloyd's algorithm [4], is an efficient heuristic that is frequently used today in both academic and industrial applications.

The k-means algorithm has been known since the 1960s from Forgy [2] and MacQueen [5]. The original presentation of Lloyd's algorithm [4], which is often used in the k-means problem, was applied to pulse-code modulation (PCM), where one was interested in the quantization of voltage signals. Today, the algorithm finds a very wide range of applications, especially in the fields of data mining and information retrieval. K-means and various other clustering methods have been used in medical research in identifying cancerous cells [6]. Search engines return various forms of media that are relevant to a query and rely on being able to identify similarity between potential results which could include text documents or images. The k-means algorithm can be used in this field to identify similar items offline and hence improve the speed at which information is retrieved in response to a query [3]. Additionally, two applications that I have used the k-means algorithm for are in customer relationship management (CRM) and feature extraction in deep learning. In CRM, I have used k-means to understand and segment customers based on data collected about their interactions with email marketing campaigns which can be used to devise strategies on how to better target customers with different historical behavior. In deep learning and the creation of neural networks for object recognition in natural images I have used k-means to extract common features found in images that are used to provide a better initialization of the networks parameters to allow for faster convergence during training of the network.

1.1. The k-means Algorithm

As mentioned earlier, given a dataset $X \in \mathbb{R}^d$ of n points and a an integer $k > 0$, the k-means algorithm seeks to find a set $C \subseteq \mathbb{R}^d$ of k centers that minimize the potential function:

$$\phi(C) = \sum_{x \in X} \min_{c \in C} \|x - c\|^2 \tag{1}$$

We can then partition the data by assigning each data point to a cluster represented by $c_i \in C$ for which $\|x - c_i\|^2$ is minimal. The procedure for finding the set C is as follows [1]:

1. Arbitrarily choose an initial k centers $C = \{c_1, c_2, \ldots, c_k\}$.
2. For each $i \in \{1, \ldots, k\}$, set the cluster C_i to be the set of points in X that are closer to c_i than they are to c_j for all $j \neq i$.
3. For each $i \in \{1, \ldots, k\}$, set c_i to be the center of mass of all points in C_i: $c_i = \frac{1}{C_i} \sum_{x \in C_i} x_i$.
4. Repeat steps 2 and 3 util C no longer changes.

The algorithm guarantees that the potential function will decrease after application of steps 2 and 3 and is also guaranteed to converge as there are only a finite number of possible partitions of the data. But there are no performance guarantees that the best clustering will be found. Therefore, it is

common practice to run the algorithm many times with different initializations to find the set C that best minimizes the potential function. The final cluster centers that are found is highly dependent on the initialization of those centers. Typically the centers are chosen uniformly at random from X, but better initialization methods do exist [1] that are known to decrease the required runtime. Our discussion will focus though on the random initialization of these points.

2. Implementations

I was interested in examining the performance of multiple implementations of k-means in different languages and with different parallelization schemes. I wrote two versions of the algorithm in python. The first implementation was very straightforward using nested for loops over the data and the cluster centers to find the nearest center for each point. Python is notoriously slow though with for loops since it is a dynamic language that is also strongly typed, hence in each iteration of a for loop the interpreter does type checking of the variables which may slow code done. Therefore, a common practice in python is to use what is called a list comprehension to avoid for loops which can be much faster. In my second implementation I did not include any for loops and made heavy use of python's popular numerical library, numpy. I also wrote another version in python that used the machine learning package scikit-learn which has k-means and also provides the ability to use some parallelization. The interface is in python but all of the numerical work is done in C on the backend. Next, I created a version of the k-means algorithm in C that mirrored my python implementation. I did not use any external libraries or the math library, all code was written from scratch. Finally, I took the sequential C code and converted it to a parallel version that made use of MPI. I will now go in to more detail on the parallel version of k-means using MPI.

2.1. Parallelization of k-means with MPI

There is no dependence between data points during the k-means algorithm as one only needs knowledge of the cluster centers at each iteration to determine which cluster a point should be belong to. Therefore, I began by splitting the data into roughly even chunks across all processes. To do this when reading data from a text file, I used MPI_File_read_at_all to have each processor read in a specified amount of data beginning at a predetermined position in the file. Next, I had to initialize the cluster centers at random from the entire dataset. I had processor 0 collect all of the initial data from other processors when needed using basic MPI_Send and MPI_Recv, more detail on this is provided in the presentation slides. Once all the centers were initialized, I used MPI_Bcast to broadcast the initializations to each processor. Now each processor has its own chunk of the data, initialized centroids, and also a vector to count the number of points that belong to each cluster which is initialized to all zeros. Each processor will also track two other numbers that will be shared: a partial sum of the potential function and also δ which counts the number of points who were reassigned to a new cluster during an iteration. Therefore, each processor much keep track of which cluster each point belongs to in order to compute δ, but this information never needs to be communicated across the network.

Now that we have initialized everything each processor will find the nearest center for each point, add that point to a temporary vector for that center, and increase the count for that center as well. Once this is done we must calculate the new centers by communicating between all of the processors. I used an MPI_Allreduce on both the temporary sum of vectors belonging to each cluster as well as the counts. This efficiently gave all processors the data needed to compute the new centers. I also used two calls to MPI_Allreduce for the potential function sum and δ which are used to check if we have converged to a local minimum. This is repeated for T trials for multiple different random initializations of the centers in order to find the best clustering that minimizes the potential function.

2.2. Parallelization with Scikit-Learn

The scikit-learn package takes a different approach to parallelizing k-means, it takes advantage of the fact that each k-means trial can be computed independently. Therefore, it has each processor run the k-means algorithm multiple times on the data, but to do so it must copy the entire dataset to each processor.

3. Performance Analysis

Each implementation was run on multiple datasets with varying configurations. I created four dataset, each with points in $\mathbb{R}^10$, with varying sizes and number of clusters. Each cluster was drawn from a gaussian distribution where each component of the mean was in $[-10.0, 10.0]$ and had unit variance. The datasets had sizes of 10,000, 100,000, 1,000,000, and 10,000,000 points. I also used one real world dataset that I got from Kaggle [7] which provided 61878 data points with 93 dimensions where each point represented a product on an e-commerce website and the each dimension was a different attribute describing the product.

3.1. Runtime Comparisons

All implementations were run on the smaller dataset with 10,000 points. Each one was tested with different settings of $k \in [5, 50]$ in increments of 5. K-means was repeated for a large number of trails for each configuration. Since the number of iterations required for k-means to converge depends on the initialization, I normalized the runtimes to obtain Figure 1. Normalization allows us to make meaning

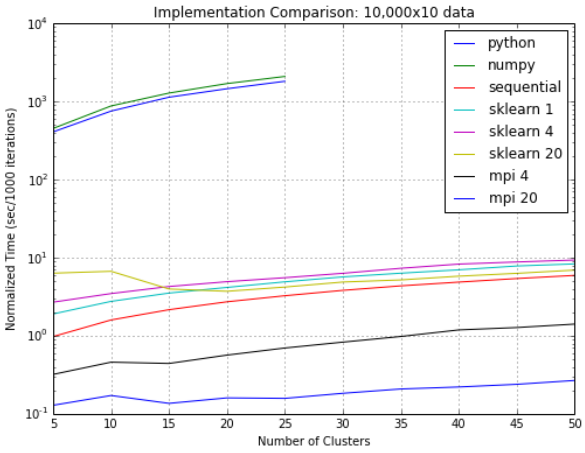

Figure 1. Plot showing normalized runtime comparisons across multiple implementations

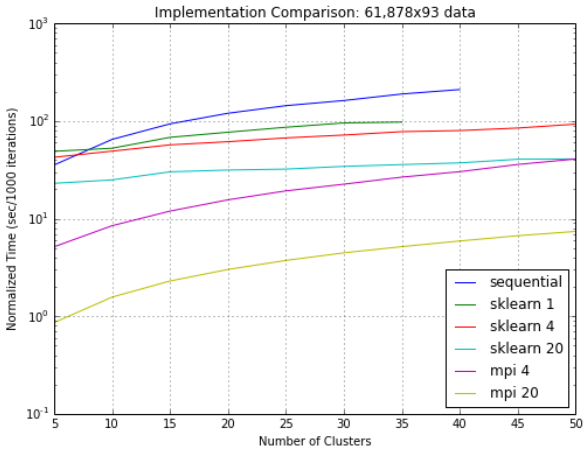

Figure 2. Plot showing normalized runtime comparisons for a larger dataset

comparisons between the implementations as any fluctuations in runtime due randomization is removed.

The first take away from Figure 1 is to note how slow the python implementations are in comparison to all of the others, two to three orders of magnitude slower. I was surprised to see the runtime of the two python implementations to be so close to one another. I would have expected the vectorized version to be much faster. As python is a high level language, it is very likely that in this version I was unaware of memory copies that were taking place behind the scenes that caused the performance to take a significant hit.

The next interesting thing to note is the performance of the scikit-learn package. This graph makes it very apparent that there is significant overhead in method used for parallelization. When 20 cores were used and the difficulty of the k-means task was low, ie. $k < 15$, then we see the runtime is slower than when 1 or 4 cores were used. Clearly this is due to the expensive nature of copying and transferring the entire dataset 20 times. Thus when the amount of data is not too large and/or the dimensionality and number of clusters is small, one does not see performance improvements by using multiple cores as the communication and memory overhead dominate over the computational complexity.

Finally, note that on this smaller task my sequential C version was able to beat out scikit-learn, and python, for all settings of the number of clusters. As expected, the MPI implementation was the fastest as the computational task is split among multiple processors and the number of bytes that must be communicated is relatively small compared to size of the entire dataset.

Next, Figure 2 shows a comparison on a larger dataset. We observe similar behavior in the sense that the tests performed show that the MPI version beats all of the others. But if we look at the far right side we may infer that the asymptotic performance of scikit-learn, as the problem difficulty increase, may prove to be faster than MPI. This makes sense though as scikit-learn performs one very expensive communication of data transfer at the beginning but then all computation is performed locally. On the other hand the MPI version will incur greater communication costs as the number of clusters increases. Therefore, we observe the line for the runtime of scikit-learn is much flatter than that of the MPI versions.

3.2. MPI Performance

Let us turn to focus on the performance of the MPI implementation of the k-means algorithm. Figure 3 plots the total runtime of the k-means algorithm as a function of the number of cluster centers that are searched for. The affect of increasing the number of centers is two-fold when examining parallelization with MPI. Not specific to the MPI implementation, increasing the number of centroids will increase the computational difficulty as we must search over more potential candidate clusters for each data point in each iteration. Therefore, doubling the number of centers will almost double the number of required FLOPs in each iteration of k-means. In relation to MPI, increasing the number of clusters will also increase communication cost as doubling the number of centers will also double the number of bytes that must be communicated over the network.

Each of the experiments ran the k-means algorithm for 500 trials on the $10,000,000$ sample data set. The experiments used 4, 8, 12, 16, or 20 MPI processes. The runtime for each is very similar when the number of clusters is small, but we see significant improvements as the number of clusters grows beyond 20. For example, when using 4 processes and 20 clusters the runtime is about 75 minutes and with 30 clusters the runtime is about 453 minutes, roughly 6 times

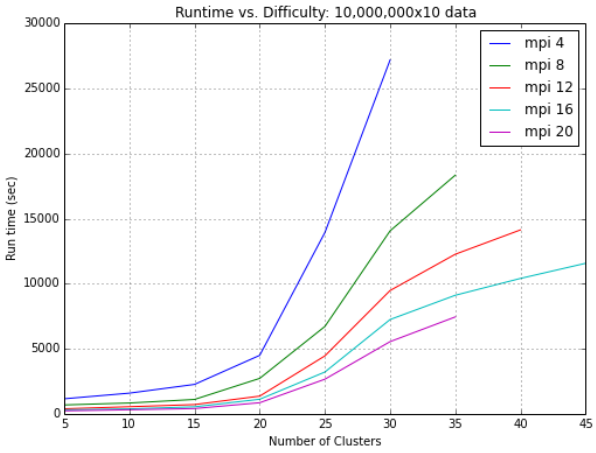

Figure 3. Runtime as a function of increasing the number of cluster centers on a large dataset of 10 million points.

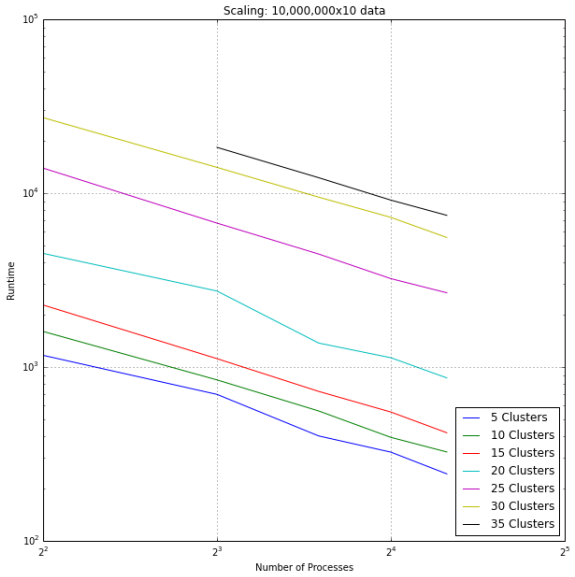

Figure 4. Strong scaling plot with log-log axes

slower. Then when using 20 processes and 20 clusters the runtime is about 14 minutes and with 30 clusters the runtime is about 92 minutes. Therefore, we see a nice scaling behavior as we use 5 times the number of processors we get a speedup close to 5 times as well. Precisely, with 20 clusters the speedup with 20 cores is about 4.89 times faster than with 4 cores and with 30 clusters the speedup is 5.21 times. This tells us that the computational cost outweighs that of the communication cost as there is more of a speedup when the number of clusters increases.

Finally, in Figure 4 we can see the strong scaling behavior of the k-means algorithm with MPI parallelization. The problem size was held constant with $n = 10,000,000$ and experimented with different numbers of clusters and MPI processes. As the plot is in general linear for all of the experiments with log-log axes we can conclude that algorithm exhibits strong scaling as the runtime decreases nicely as we increasing the computing power.

4. Conclusion

Through this project I learned a lot about parallelization practices and the inherent tradeoffs. For problems of different sizes and different complexities there may be some parallel schemes that will provide a greater speedup over others. My study mainly focused on parallelization with MPI in C but made performance comparisons to a variety of other implementations as well. While running my experiments on the HPC cluster at NYU I ran in to a few issues as the size of my data increased. It appeared that some of my MPI processes had possibly failed mid job or had relocated to a new node in the middle of the computation. I did not figure out a solution to this problem, but I am sure combing through the documentation on the NYU HPC website would provide some guidance. Overall, I was very impressed by the speedup I was able to obtain with my parallelization with MPI. What I did not get to complete in time for this report which I was very interested in doing was trying to improve the performance of the python versions. To do this, I wanted to to test out a python calls to pure C code and also an OpenCL interface. Further work that I would be interested in performing is creating a hybrid implementation that uses both MPI and OpenMP to increase the parallelization per node.

References

[1] Arthur, David, and Sergei Vassilvitskii. ”k-means++: The advantages of careful seeding.” Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2007.

[2] Forgy, Edward W. ”Cluster analysis of multivariate data: efficiency versus interpretability of classifications.” Biometrics 21 (1965): 768-769.

[3] Liu, Ting, Charles Rosenberg, and Henry A. Rowley. ”Clustering billions of images with large scale nearest neighbor search.” Applications of Computer Vision, 2007. WACV’07. IEEE Workshop on. IEEE, 2007.

[4] Lloyd, Stuart. ”Least squares quantization in PCM.” Information Theory, IEEE Transactions on 28.2 (1982): 129-137.

[5] MacQueen, James. "Some methods for classification and analysis of multivariate observations." Proceedings of the fifth Berkeley symposium on mathematical statistics and probability. Vol. 1. No. 14. 1967.

[6] Wang, X. Y., and Jon M. Garibaldi. "A comparison of fuzzy and non-fuzzy clustering techniques in cancer diagnosis." Proceedings of the 2nd International Conference in Computational Intelligence in Medicine and Healthcare, BIOPATTERN Conference, Costa da Caparica, Lisbon, Portugal. 2005.

[7] https://www.kaggle.com/c/otto-group-product-classification-challenge