Clustering on Clusters

An analysis of K-Means clustering on distributed systems

Nicholas Livingstone
The University of New Mexico

December 16th, 2021

1 Introduction

Clustering methods are an important tool in data science and machine learning. K-means is one such algorithm that is currently one of the most popular clustering methods today. This algorithm can effectively cluster large groups of data well, but due to its iterative nature it can exhibit poor performance when run through a serial implementation. In the world of big data, it's critical to have efficient algorithms that can process lots of data quickly. This report will analyze a parallel implementation of k-means clustering on a distributed system and demonstrate how different types of data can effect its performance as well as potential bottlenecks and limitations.

2 What is Clustering

Clustering is a machine learning technique which groups a set of data points. Given a set of data points X, a clustering method will attempt to assign each point to a specified group. The core idea behind clustering is that we should expect two data points of the same group to share similar qualities, and conversely, data points of differing groups to have dissimilar qualities. An example of clustering can be seen on the right in Figure 1. The data points in the example have been clustered into three different groups: red, green, and blue. Clustering can be used for a variety of purposes. For example, clustering can be used to identify bank fraud by grouping suspicious transactions together or to recognize patterns in DNA.

Figure 1: Clustering example

3 K-Means Clustering

K-Means clustering was first presented in 1957 by Hugo Steinhaus [11]. It is a specific clustering method which utilizes the distance between data points to find the 'center' of a cluster called *centroids*. The name stems from k, which is a parameter representing the number of clusters the algorithm should generate. A demonstration of k-means can be seen in Figure 2. The algorithm works as follows. First, an initial set of centroids are chosen, this is represented by the red and blue 'X's in panel *b*. These centroids can be chosen randomly or procedurally (procedural initialization

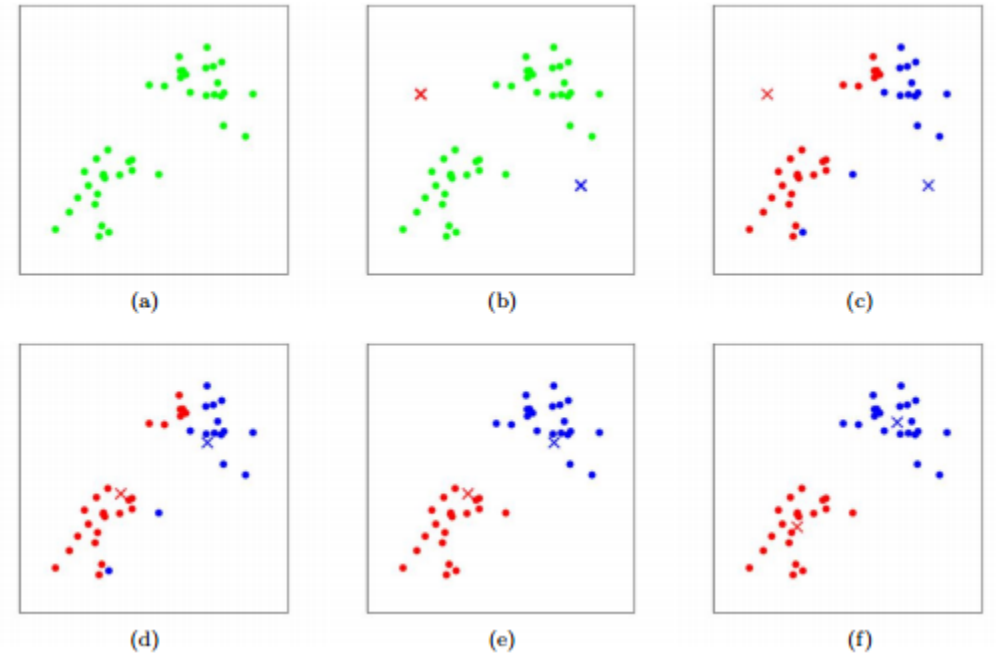


Figure 2: Example of K-Means Clustering

of centroids can improve how quickly the algorithm determines the final clusters). Next, each point of the data set is added to a group by finding the nearest centroid. In panel c, we can see that each point becomes colored based on the centroid it is nearest to. Then a new centroid is calculated by determining the average of all points in a cluster. Panel d shows the updated centroids which have shifted from their original position. This process is repeated iteratively until the centroids no longer move past a specific tolerance or the algorithm has reached a predefined number of maximum iterations and does not converge. Depending on the data set, it is possible for k-means to not converge. Often when the clusters are not well defined, not circular in shape, or overlap in strange ways [10].

4 Parallel K-Means

4.1 Implementation

The parallel implementation of k-means shares much in common with the serial version with a few minor changes. Assume n points each with d dimensions and P processes, with a given input X and k clusters. Centroid initialization will occur only on the root process and these centroids are broadcasted to all other processes. Next, n/P points of X will be scattered to each process. Then each process will identify the group membership of each locally contained point. The new local centroids are computed from these groups. Following this an *AllReduce()* which broadcasts each local centroid to every other process and sums them locally. Each process then calculates the new global centroids by finding the average of these centroids. Similarly to the serial approach, this process repeats until convergence occurs or the max number of iterations is reached. A full outline of the method can be found in Algorithm 1.

4.2 Analysis

We should expect that the performance of the parallel implementation to perform better than the serial approach, as it now distributes computations across multiple processes and does not require that the original data be communicated. The key step in this algorithm which allows for a parallel implementation is the *AllReduce()*. However, this key step is also the source of potential bottlenecks. The first bottleneck is caused by communication costs. In this function call, we must send a total of $k \times (p-1) \times d$ values among all process. This implies that our communication costs will be dependent on the number of processes, clusters, and the dimension of our original data. Additionally, the *AllReduce()* is a blocking call. Meaning that, each process must identify membership of all points and compute the new local centroids in each iteration before the processes communicate. This can cause performance drops depending on the characteristics of the original data set, specifically this depends on the sparsity of the data. If the data exhibits high sparsity it can cause poor load balancing. Some processes may end up completing local computations much faster than others causing some processes to sit idly waiting for others to complete.

5 Methods

5.1 Data and Testing

To test the performance of parallel k-means, multiple data sets were chosen from the *Benchmark Suite for Clustering Algorithms* with varying amounts data points as well as different values of k [7]. A list of the datasets selected can be found in Table 1. Originally, data sets were to be selected to test the affects of sparsity on performance. However, data that was consistent in all other parameters beyond sparsity for clustering were unable to found and could not be tested against. The parallel method was tested over 2, 4, 8, and 16 processes. The average runtime of 100 runs of the programs was taken as result values. Additionally, to avoid skewing of data by randomly generated centroids, the clustering methods utilized kmeans++ to initialize the centroids.

Table 1: Data sets Used.

Name	n	k	d
dense	200	2	2
tetra	400	4	3
twodiamonds	800	2	2
wingnut	1016	2	2
h2mg_2_10	2048	2	2
a1	3000	20	2
engytime	4096	2	2
a2	5250	35	2

5.2 Technologies used

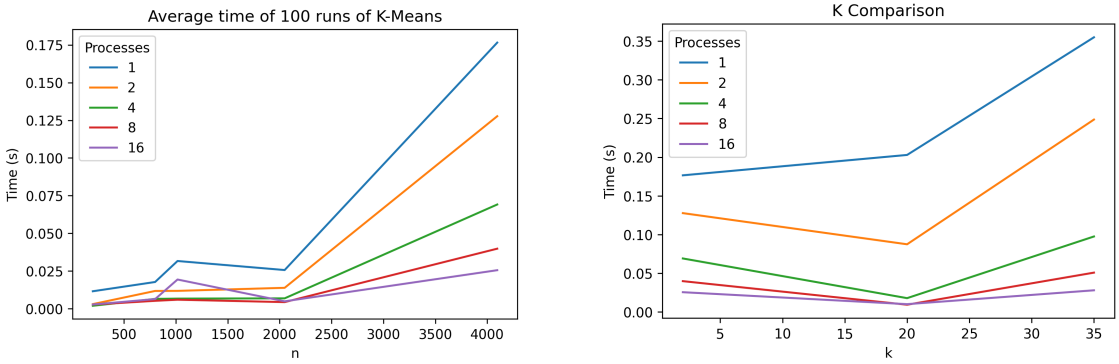
The algorithms were implemented using the anaconda distribution of python [2] with the mpi4py [5][6][3][4], sci-kit learn [9], scipy[12], and numpy libraries [8]. The test were run on the wheeler cluster of the Center for Advanced Research and Computing at the University of New Mexico[1].

6 Results

Figure 3 shows the results of running k-means for different processes over different values of n (the number of points in the input data) and k (the number of clusters). In both graphs, a process count of 1 represents the serial implementation of the algorithm.

From 3(a), it's clear that as the number of processes increases we see an increase in performance. In general, every parallel run was faster than the serial performance. However, until $n \approx 2000$ the

performance of each parallel method does not vary greatly. It's not until we reach larger data sizes that a difference in performance between different process counts becomes clearly evident. In fact, there is actually a brief decline in performance for a small data input when using 16 processes. This can can be explained by the communication overhead mentioned earlier in section 4.2 in which the number of values being communicated increases as we increase the number of processes. Although not present, it can be hypothesized that this issue will become worse if we increased to 32 processes or even more. Figure 3(b) shows the affects of different values of k on k-means. As k gets large, it is evident that communication overheads increase overall runtime



(a) K-Means over values of n

(b) K-Means over different values of k

Figure 3: Graphs of Test Results

7 Conclusion

As shown in the results of this report, a parallel implementation of K-Means is a viable option for clustering large well defined data sets as it consistently outperforms the traditional serial approach. I would encourage future research to analyze the potential limitations caused by sparse matrices and how the issues with regards to load-balancing can be minimized. Potentially a pre-analysis of the data could be conducted as a solution in order to determine optimal scattering of data among processes. This could involve the reordering of the matrices as present in parallel sparse matrix solves or utilizing a stratified distribution in which each process receives an equally sparse section of the original input data. Nonetheless, the parallel method does an effective job of distributing computation across lots processes with only minor communication costs in comparison to other parallel algorithms.

Acknowledgment

Code used for this analysis is provided in a repository at https://github.com/nicholaslivingstone/parallel-kmeans.

References

[1] [n.d.]. CARC systems information :: Center for Advanced Research Computing | The University of New Mexico. https://carc.unm.edu/systems/Systems1.html

[2] 2020. Anaconda Software Distribution. https://docs.anaconda.com/

[3] Lisandro Dalcin and Yao-Lung L. Fang. 2021. mpi4py: Status Update After 12 Years of Development. *Computing in Science & Engineering* 23, 4 (July 2021), 47–54. https://doi.org/10.1109/MCSE.2021.3083216

[4] Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. 2011. Parallel distributed computing using Python. *Advances in Water Resources* 34, 9 (Sept. 2011), 1124–1139. https://doi.org/10.1016/j.advwatres.2011.04.013

[5] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. 2005. MPI for Python. *J. Parallel and Distrib. Comput.* 65, 9 (Sept. 2005), 1108–1115. https://doi.org/10.1016/j.jpdc.2005.03.010

[6] Lisandro Dalcín, Rodrigo Paz, Mario Storti, and Jorge D'Elía. 2008. MPI for Python: Performance improvements and MPI-2 extensions. *J. Parallel and Distrib. Comput.* 68, 5 (May 2008), 655–662. https://doi.org/10.1016/j.jpdc.2007.09.005

[7] Marek Gagolewski et al. 2020. Benchmark Suite for Clustering Algorithms – Version 1. https://doi.org/10.5281/zenodo.3815066

[8] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[10] Yordan P. Raykov, Alexis Boukouvalas, Fahd Baig, and Max A. Little. 2016. What to Do When K-Means Clustering Fails: A Simple yet Principled Alternative Algorithm. *PLOS ONE* 11, 9 (Sep 2016), e0162259. https://doi.org/10.1371/journal.pone.0162259

[11] Hugo Steinhaus. 1957. Sur la division des corps matériels en parties. *Bull. Acad. Pol. Sci., Cl. III* 4 (1957), 801–804.

[12] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Algorithm 1 Parallel KMeans

Assume n points and P processes

procedure PKMEANS(X, k, tol, max_iter)
    $c_0 \leftarrow$ INITCENTROIDS(X, k) ▷ On root process
    $c_p old \leftarrow$BROADCAST(c_0)
    $X_p \leftarrow$ SCATTER(n/P elements of x)
    $dist \leftarrow$ Array$[n/P]$ $[k]$
    $label \leftarrow$ Array$[n/P]$
    $c_p new \leftarrow$ Array$[k]$
    $iter \leftarrow 0$

    while $iter < max\_iter$ **do**
        for $x_i \in X_p$ **do**
            for $c_j \in c_p old$ **do**
                $dist[i][j] \leftarrow$ DISTANCE(x_i, c_j)
                $label[i] \leftarrow$ ARGMIN($dist[i][]$) ▷ Closest clusteroid
            end for
        end for

        $c_p new \leftarrow$ COMPUTENEWCENTROIDS ($dist, label$)
        $c_p new \leftarrow$ ALLREDUCE($c_p new$) $/P$ ▷ mean of local clusteroids

        if $abs(c_p new - c_p old) \leq tol$ **then**
            Return $c_p new$
        end if

        $c_p old \leftarrow c_p new$
        $iter \leftarrow iter + 1$
    end while
Return $\emptyset$ ▷ Did not converge
end procedure