Parallelized Spectral Clustering on Apache Spark

Wenxuan Cai
University of California
Berkeley, CA 94720
wenxuancai@berkeley.edu

Yaohui Ye
University of California
Berkeley, CA 94720
yeyh@berkeley.edu

Quan Peng
University of California
Berkeley, CA 94720
quan1992@berkeley.edu

ABSTRACT

Clustering is one of the most popular techniques adopted in the industry and research area to detect group structure in the dataset. k-means and Spectral Clustering are two algorithms widely used for grouping similar subsets of data within a large dataset. k-means clustering is simple and fast, but depends heavily on the initialization and is likely to stuck on the local optimum. Spectral Clustering often outperforms k-means by using eigenvectors of the affinity matrix to project data into lower dimensions, and utilizes k-means clustering only on the last step of the algorithm. However, Spectral Clustering is computationally expensive and even the simple implementation has a serious memory bottleneck. Thus, in this project, we tried to run Spectral Clustering in parallel on Spark, with purpose to gain more insights into the performance of Spectral Clustering and study Spark, the most popular fast and general-purpose cluster computing system. In this project we experienced with Spectral Clustering on MNIST digit image dataset [18] and compared clustering speedup on different number of nodes. Specifically, we parallelized different steps of Spectral Clustering and got a reasonably good speedup on 10,000 image dataset. Moreover, we tried different optimizations and measured how much they improved the clustering performance. For k-means, we tried smart initialization techniques such as KM-2 [21] and BF [4]. For Spectral Clustering, we applied subsampling at the beginning to group data points by a distortion minimizing transformation, and conducted Spectral Clustering on the preprocessed dataset [21]. This optimization significantly reduced the time and memory needed by the algorithm while retaining comparable clustering accuracy. The combination of parallelism and algorithm optimizations provided us with a lot of speedup. From the project we got to learn how Spark works to achieve efficient parallelism, how to set up Spark cluster, and a nice way to parallelize Spectral Clustering on large dataset.

Categories and Subject Descriptors

H.4 [**Machine Learning**]: Clustering; D.1.3 [**Concurrent Programming**]: Parallel programming

Keywords

Machine Learning, k-means, Spectral Clustering, Spark

1. INTRODUCTION

Clustering is one of the most widely used techniques for exploratory data analysis, with applications ranging from statistics, computer science, biology to social sciences and psychology [20]. Typical applications include graph partition, speech separation, and image segmentation. k-means is a clustering algorithm easy to implement. However, as we have worked with k-means over time, the simplest method doesn't always give the best clustering result. The performance of k-means can vary significantly depending on the initialization method. A lot of optimization techniques exist to solve this problem. For example, Hartigan proposed to run k-means for multiple time with random initializations and pick the best result [10]. Sampling-based KM-2 [21] suggests to break the k-means into two steps. Before doing k-means on the entire data set, it would first run a quick k-means on a subset of data to pre-selected initialization centroids.

While k-means provides reasonable clustering performance on most problems, Spectral Clustering, one of the most popular modern clustering algorithms, outperforms the traditional clustering algorithms such as k-means in finding group structure. The reason is that traditional k-means clustering only considers distances to cluster centroids, but Spectral Clustering utilizes the eigenvalues of the affinity matrix to perform dimensionality reduction and works with distance between all pairs of points. Spectral clustering has good performance on small data set but limited applicability to large-scale problems due to its computational complexity $O(n^3)$ in general, with n data points. The reason is simple. Spectral

Clustering requires to compute the similarity matrix involving all data points, which is going to be an n by n matrix. After that, doing the eigenvector decomposition is an operation of $O(n^3)$ complexity. This cubic runtime makes the algorithm prohibitively expensive on dataset of million level. In this project, we first implemented the basic Spectral Clustering algorithm. After that, we tried to run the algorithm in parallel on Spark to speedup runtime on large dataset. Moreover, we adopted the optimization technique from paper Fast Approximate Spectral Clustering [21] to speedup our Spectral Clustering. Given the complexity of the algorithm, it will be time-consuming to run on the large real world dataset. We followed the idea ofčitechen2011parallel to take usage of the parallelism and algorithmic optimization would allow us to train efficiently on more data and improve the result. Actually, we got comparable accuracy and faster runtime after applying all techniques. The remainder of the report is organized as follows. In Section 2, we will introduce the details of Spectral Clustering. In section 3, we describe the ways that we parallelize different stages of Spark. In section 4, we give an overview of Spark and our architecture. In section 5, we evaluate our performances on testing set. We present the future work in section 6 and conclusion in Section 7.

2. BASIC CLUSTERING ALGORITHMS

In this section we introduce the details of different stages of Spectral Clustering algorithms.

2.1 Spectral Clustering

The goal of Spectral Clustering is to partition the data into k disjoint classes such that each point will be assigned to a single class. A good partition would break the data into several loosely connected components, while similarity within the component is high.

Basically, the Spectral Clustering is a flexible class of clustering procedures, which makes usage of the eigenvalue of the similarity matrix of the input data to perform dimensionality reduction before clustering in lower dimensions. Employing the eigenvector decomposition, Spectral Clustering is able to beat k-means when detecting group structures in large dataset.

Before providing the algorithm in pseudocode, we now present the algorithm briefly and introduce our notations. Basically, Spectral Clustering can be divided into three steps, and each step possesses full potential for parallelization: Laplacian Matrix Construction, Eigenvector Decomposition, and k-means Clustering. Given n data points $x_1, x_2, \cdots, x_n$ in R^d, the first step is to construct the affinity matrix W, where W_{ij} is the distance between x_i and x_j. In our case, we use the Euclidian distance directly. Thus $W_{ij} = \sqrt{\sum_{k=1}^{d}(x_{ik} - x_{jk})^2}$. As we compute the affinity matrix W, we then build the diagonal degree matrix D where $D_{ii} = \sum_{k=1}^{n} W_{ik}$. With degree matrix, we compute the Laplacian Matrix $L = D - W$. After that, we take the first k eigenvectors $u_1, u_2, \cdots, u_k$ of L, as matrix $U \in R^{n \times d}$. The last step is to use each row of U as the a lower dimensional representation of x_i, and do a k-means clustering. Written in pseudocode, it is shown in Algorithm 1.

Algorithm 1 Spectral Clustering

1: Construct affinity matrix W from data set $x_1, x_2, \cdots, x_n$
2: Construct degree matrix D from W
3: $L \leftarrow D - W$
4: Build $U \in R^{n \times d}$ where the columns are the first k eigenvectors of L
5: Use the i^{th} row U_i of U to represent x_i, do k-means clusterings on $U_1, U_2, \cdots, U_n$

2.1.1 Bottleneck

It can proved be proved that mathematically Spectral Clustering solves the Ncut problem. The algorithm is not hard to code, but there are two serious problems with the naive implementations.

1. The memory overhead. Like PCA, spectral clustering takes usage of the spectrum of the affinity matrix to project data points into lower dimensions. Constructing affinity matrix on a single node requires $O(n^2)$ computation since it needs to process over all pairs of points. Consequently, we need to store the affinity matrix, which takes $O(n^2)$ space. Storing a dense matrix on a single machine RAM becomes impossible when n goes to level of 100k. Using 32 bit integer, storing such an affinity matrix takes $(100,000^2) * 4\text{B} = 40\text{G}$ memory.

2. The time complexity of Spectral Clustering is $O(n^3)$, which comes from the requirement to perform eigenvector decomposition on the $O(n^2)$ affinity matrix. To give the reader a quantified idea of this time lower bound, we ran the simple Spectral Clustering on the 10,000 images from MNIST digit dataset [14]. It took more than 30 minutes to do the eigenvector decomposition using the numpy library. Thus, Spectral Clustering is computationally expensive for large datasets, which limits its application to large-scale problems.

2.1.2 Solution

For the memory overhead, a lot of solutions exist to handle the memory bottleneck. Three typical ideas are

1. Zero out W_{ij} if it is smaller than a certain threshold ϵ

2. For each point x_i, only store the k nearest neighbours in the affinity matrix

3. Use Nystrom approximation to store a dense submatrix [8]

In our project, we chose the second solution. In theory, the smaller the k, the worse the result because we throw away more info. k in k-nn is a tunable parameter, and we picked 500 which both fitted the memory of our machine and gave a good performance.

In order to speedup the slow eigenvector decomposition, we adopted the idea from Michael Jordan's paper Fast Approximate Spectral Clustering [21] to do a downsampling at the beginning to significantly reduce the size of the affinity matrix. Basically, after choosing a sampling ratio α, we first ran a fast k-means on the data set to find $m = \alpha n$ centroids $y_1, y_2, \cdots, y_m$. Then, each x_i will be represented by the centroid closest to it. Then we performed the Spectral Clustering on $y_1, y_2, \cdots, y_m$, and recovered the cluster membership for each x_i by looking up the cluster membership of the corresponding y centroid. This optimization turned out to work well, with a little drop in accuracy because we lost some information during the downsampling. Besides that, it significantly reduced the time and memory usage of the Spectral Clustering algorithm.

2.2 k-means

k-means is one of the most common clustering techniques. It aims at partitioning n observations into k clusters. The first step of this algorithm is to randomly choose k points from the dataset as our initial cluster centroids. The second step is to take each point of the dataset and associate it to its nearest centroid. The third step is re-calculating the centroids for each cluster that we get from last step by taking the means of all points within that cluster. After calculating the k new centroids, we calculate the new binding of each point and its nearest new centroid and start a new iteration from the second step. The algorithm stops when all centroids stop moving.

The accuracy of k-means is highly dependent on the initialization method, since if the initial centroids aren't well selected, the algorithm could be stuck on local optimum. Various approaches for selecting initial centroids have been proposed [11, 2, 13], and for our project, we used three different initialization methods, KM-1 [10], KM-2 [21] and BF [4], which will be described in following parts.

2.2.1 Optimization - KM-1

The first optimization is to simply run the algorithm for several times. For each time, we use randomly initialized centroids and calculate the accuracy after convergence. After repeating these runs, we choose the one with the highest accuracy as our final result.

2.2.2 Optimization - KM-2

KM-2 is the second optimization, which consists of two stages of k-means. During the first stage, we sample a fraction of the data uniformly at random and run k-means with k clusters on the sampled data. The second stage uses the centroids obtained from the first stage as the initial centroids and run k-means. Basically, the idea is to obtain good initial centroids so that fewer iterations and restarts are needed for the second stage.

2.2.3 Optimization - BF

The Bradley and Fayyad algorithm (BF) [4] consists of three stages of k-means. At first, we run k-means on randomly selected subsets of the entire data set for several times. The output centroids of all individual runs form a new data set. In the second stage, we run k-means on the new data set and the obtained centroids are used as the initial cluster centers for the third stage of k-means.

3. PARALLEL SPECTRAL CLUSTERING

As we generalized Spectral Clustering into 3 steps: Laplacian Matrix Construction, Eigenvector Decomposition, k-means, we can use different method to parallelize each step.

3.1 Parallel Laplacian Matrix Construction

In order to speed up the affinity matrix construction, we adopted idea from Wen-Yen's paper [5] and used map and reduce operation on Spark to perform computation distributedly.

MapReduce is a parallel computing framework designed by Google [6]. Basically, it divides the computation jobs into two steps - the map phase and reduce phase. Programmer can pass in specific function to each phase as long as it follows the general rule of MapReduce framework. In the map phase, the function needs to map the input into key-value pairs, which will be shuffled afterwards and passed to the designated reducer. In the reduce phase, key-value pairs which share the same key are collected by a single reducer. The reduce function is responsible for processing these values and generate the final result. MapReduce offers an abstract programming model which is able to organize thousands of machines to accomplish big parallel jobs. It also provides a fault tolerance mechanism that starts redundant job on another node once it detects any job failure. In general, MapReduce is widely used in noniterative algorithms. In our case, computing Laplacian Matrix is an independent and noniterative job, which fits well into the specialty of MapReduce.

Spark provides an easy programming interface for launching MapReduce jobs. By passing a lambda expression, Spark is able to distribute RDD across partitions and collets result according to the user-specific reduce function.

Because of the $O(n^2)$ memory overhead and the importance of using a sparse eigensolver, we chose to store only top 500

neighbours for each data point in the affinity matrix. In this case, with p computing nodes, each node would store n/p rows of affinity matrix. For the computing work, since for each data point we needed to compute the distances to all other points, we partitioned the work as n/p for each node. In the map phase, based on the index i of the mapper, it scanned through in/p to $(i+1)n/p$ data points and emitted the intermediate result with same key. Thus, each reducer could get a balanced amount of key/value pairs and perform the distance computations. In the reduce phase, each reducer would iterate through all its data points, and compute the distances to all other points. This required some internodes communication and Spark handled that for us just like programming a distributed hash table in UPC.

In order to take the top 500 neighbours for a data point, the simplest solution was to compute all the n distances, sort them, and pick the top 500. This solved the problem but consumed lots of unnecessary memory to store all the n distances, and this approach became expensive once n gets large. Our solution here was to use a max heap, and maintained the size to 500. In general, we kept n/p max heaps. Whenever we read a remote points, we scanned through the local points and put distances into corresponding max heap. This approach took advantage of the locality and turned out to be much faster than fetching all remote points once for each local data point. Moreover, when computing distance

$$||x_i - x_j|| = ||x_i||^2 + ||x_j||^2 - 2x_i^T x_j$$

we cached the value of $|x_i||^2$ and $||x_j||^2$ and reuse them later. It gave us a much better $O(d)$ runtime.

Since k nearest neighbour is not a symmetric relationship, the affinity matrix we constructed was not symmetric, which caused problem when passing to the sparse eigensolver later. We added a simple fix there that we copied each entry to its symmetric position in the matrix. After we computed the affinity matrix W, we constructed the diagonal degree matrix D and Laplacian matrix L as

$$D_{ii} = \sum_{j=1}^{n} W_{ij}$$

$$L = D - W$$

3.2 Eigenvector Decomposition

As we have the Laplacian matrix L, the next step is to find its eigenvectors to project data points into lower dimensions. A quick research through the existing sparse eigensolvers online pointed us to PARPACK [7], a parallel ARPACK implementation based on MPI.

MPI, known as message passing interface, is a cross platform message passing library for writing parallel programming [9]. At the beginning, an MPI program will be loaded into the memory, and each processor / process will gain a unique ID. MPI provides a universal protocol for programs to exchange messages and data. Different processes can communicate and synchronize with each other by calling MPI routines. Typical MPI functions are send, recv, broadcast, and reduce. PARPACK scales well on multiple nodes using MPI.

However, transplanting PARPACK onto Spark would not be an easy thing to do, so we put our attentions to the existing MLlib [17] integrated with Spark, and we found the SVD implementation in Spark. SVD, short for Singular Value Decomposition, factorizes a matrix X into the products of three matrices

$$X = U\Sigma V^T$$

where U is the left singular matrix, V is the right singular matrix, and Σ is a diagonal matrix of eigenvalues. The advantage of using Spark SVD is that we can specify the number of rank k we want, so we don't have to compute all the eigenvectors and use a lower rank approximation to the original matrix. This can save storage, cut the communication cost, and reduce overfitting on aspect of machine learning. We used the SVD library call directly on Spark to compute the eigenvalue and eigenvectors of Laplacian matrix L. The current Spark SVD implementation in mllib is partially distributed. As we found from the source and document, the matrix-vector multiplication $X^T Xv$ is computed in a distributed way, but the eigenvalues and eigenvectors are computed on the driver. Assume for a n by n matrix X, and we want the top k eigenvectors, we need to fit a n by k matrix into driver's memory. Actually, behind the screen, it calls the ARPACK to compute eigen-decomposition of $X^T X$ on the driver node. Although Spark SVD is not fully distributed at current version, we got enough memory on drive for our Amazon instance to finish the eigen-decomposition.

3.3 Parallel k-means

Once we get the first k eigenvectors of the Laplacian matrix, we use k means to process the matrix and each row of the matrix is regarded as one data point in the algorithm.

To parallelize k-means clustering, we first need to randomly select cluster centers from all points and then distribute all data points to different machines. At this point, each machine has a portion of the whole data set and a copy of cluster centroids. During each iteration, a node only needs to compute the new centroids information with local data thus parallelizing the algorithm. The model is very similar to MPI model as mentioned in [5], but the implementation of parallelization in Spark is making communication easier than MPI.

4. SPARK OVERVIEW

Map/Reduce was first presented by Google in 2002 [6] and has been widely used for processing large data sets in a distributed environment. However, when writing our own MR programs in Hadoop or R, we found it difficult both because

of the complicated configuration of the system and the inflexibility for different problems. We always need to build specialized systems for different use cases, which limits its applications. Specialized systems, including Pregel [16] and GraphLab [15], were built to deal with some of the limitations of traditional Map/Reduce.

Hosted under Apache Open Source Licence, Spark is a fast and general-purpose clustering computer system which efficiently schedules parallelled jobs into distributed computation nodes and utilizes the in-memory cache to reduce IO communication overhead incurred in traditional Map/Reduce frameworks such as Hadoop [22]. The goal of Spark is to generalize Map/Reduce to support new applications within the same engine.

We chose Spark to be our parallelization tool for a couple of reasons. On the one hand, a lot of machine learning problems require iterative processing over training set to learn the best parameters, which typically applies the same function repeatedly. Traditional Map/Reduce framework saves the intermediate results onto disks and reads them back later if needed, which incurs unnecessary IO overhead. Spark, using fast in-memory cache for those intermediate values, avoids the performance penalty of data reload and speeds up the entire training process [22]. On the other hand, we want to take the opportunity to experiment with this state-of-the-art technique and learn about how it works.

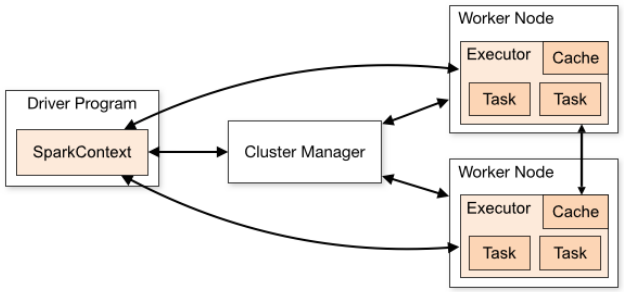

Figure 1: Spark Architecture [19]

Figure 1 shows the architecture of a Spark system. Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in the main program [19]. Spark uses a Master-Slave model, where the master node stores the SparkContext object and worker nodes (i.e., slave) are responsible for running computations and storing data for the application. When we want to run a Spark task, we can just submit it to the master node, and the SparkContext will send the tasks for the workers to run.

The most essential abstraction introduced by Spark is resilient distributed datasets (RDD), which is a collection of objects partitioned across different machines that enables Spark to perform efficient in-memory computation for iterative and interactive algorithms, which helped us a lot when running the experiments.

5. EVALUATION

In this sector we present the dataset we tested on and the results from running different clustering algorithms with different numbers of computing nodes.

During evaluation, we used MNIST digit image dataset from [18]. In short, it consists of hand-written digit images in grey scale. Each image is 28 * 28, so we have 784 dimensions in total. Particularly, each dimension is a pixel of grey scale between 0 and 255. The 10,000 images we picked contains all digits from 0 to 9. Figure 2 gives the visualization of random images of different digits.

Figure 2: Visualization of digit images

5.1 Environment

We set up a Spark cluster of 8 nodes on Amazon Spot instances. One of these nodes is the Master node, which manages the tasks. The other 7 nodes are Workers, which are actually processes/processors that run computations and store data for the application. The spot instances we rent were R3.xlarge High Frequency Intel Xeon E5-2670 v2 (Ivy Bridge) Processors [1], each of which has 4 VCPU and 27 GB of RAM.

5.2 Results

We run our parallelized program with different number of workers, ranging from 1 to 7.

5.2.1 Parallel Laplacian Matrix Construction

In Table 1, we show the running time and speedup of the matrix construction with different node numbers on the MNIST digit dataset. In Figure 4 we show the running time plot on different number of nodes and in Figure 3 we show the speedup plot. We could observe that the matrix construction is expensive on a single node, and we gained around 5.3x speedup when processing on 7 nodes.

5.2.2 Parallel k-means

Table 2 shows the running time and speedup of k-means clustering algorithm with different numbers of worker nodes. The speedup plot for the second part of the optimization, k-means clustering, is shown in Figure 5 and the corresponding time plot is shown in Figure 6. We gained around 2.5x

Nodes	Running time	Speedup
1	1626.77s	-
2	856.19s	1.9
3	580.99s	2.8
4	464.79s	3.5
5	396.77s	4.1
6	346.12s	4.7
7	306.93s	5.3

Table 1: Running time and Speedup of parallel Laplacian Matrix Construction on Spark

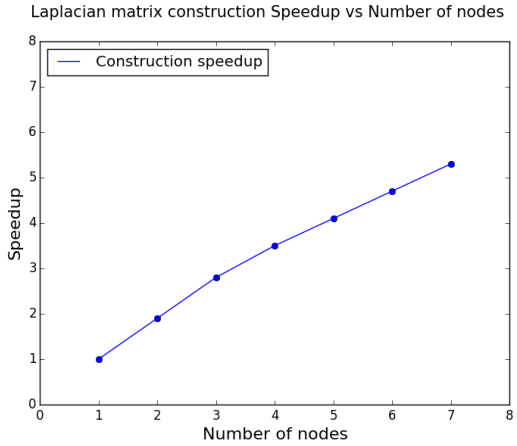

Figure 3: Laplacian matrix construction speedup vs number of nodes

speedup of the algorithm with 7 nodes. Even though the speedup is still almost linear to the number of nodes, comparing with last experiment, it's actually further from the ideal speedup. We used our own k-means implementation, instead of using the MLlib one, thus it's possible that we failed to fully exploit the ability of Spark clusters.

Nodes	Running time	Speedup
1	27.35s	-
2	21.20s	1.3
3	15.83s	1.7
4	14.12s	1.9
5	13.52s	2.0
6	12.08s	2.3
7	11.00s	2.5

Table 2: Running time and Speedup of parallel k-means on Spark

5.3 Centroids visualization

Figure 7 shows the visualization of the cluster centroids obtained from our Spectral Clustering algorithm, which successfully separated the boundry between different digit images.

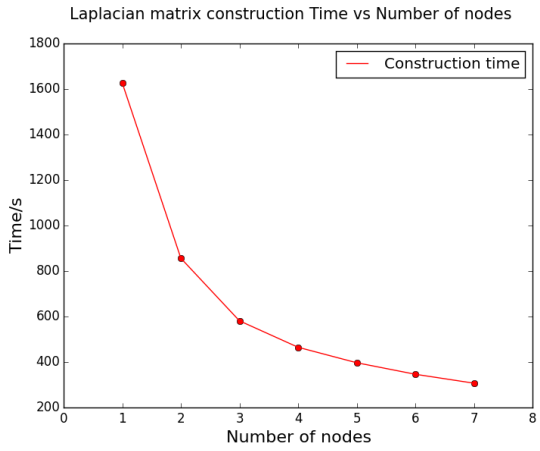

Figure 4: Laplacian matrix construction time vs number of nodes

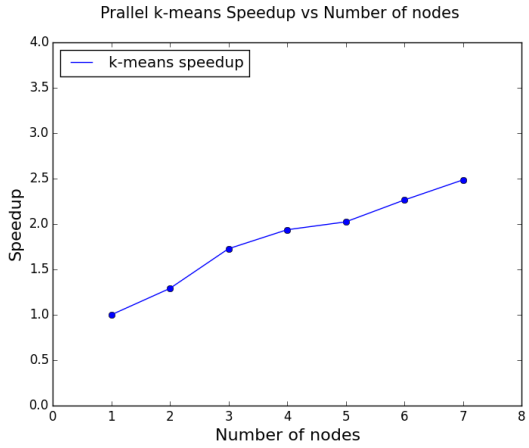

Figure 5: k-means convergence speedup vs number of nodes

6. FUTURE WORK

Currently we use Euclidean distance in the affinity matrix. It provides a symmetric estimation of similarity, but is not the only choice for the distance metric. Other similarity metrics, such as Gaussian Kernel [12], can be used to smooth the affinity estimation. Gaussian Kernel is defined as

$$K(x, y) = \frac{||x - y||^2}{2\sigma^2}$$

σ is a tunable parameters and it plays the role of smoothing the decision boundary. Adding the kernel could give us more complicated affinity metric and improve the performance.

For the eigenvector decomposition, we simply used the unnormalized Laplacian Matrix $L = D - W$. Actually, there exists other versions of the normalized Laplacian Matrix such as $L_{sym} = D^{-1/2}LD^{-1/2}$ and $L_{rw} = D^{-1}L$. As described by their names, L_{sym} is a symmetric matrix, and L_{rw} is closely related to random walk. The normalized

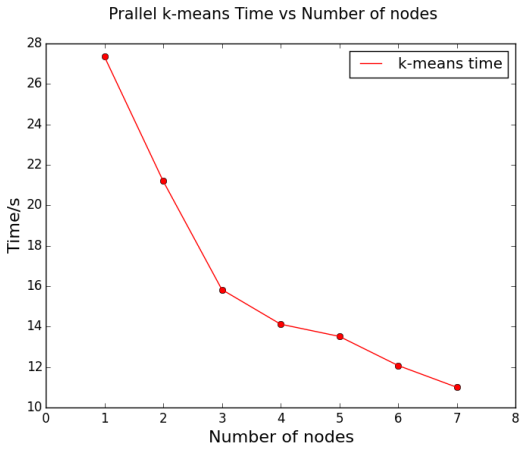

Figure 6: k-means convergence time vs number of nodes

Figure 7: Visualization of cluster centroids

Laplacian matrix has some extra properties compared to unnormalized one. Adding a distributed matrix multiplication into our Spectral Clustering algorithm could help us explore these properties and see the influence on the final result.

Lastly, we could extend the project by applying parallel Spectral Clustering to other datasets such as bag-of-word [14] to cluster related documents and compared results with other NLP models such as LDA [3].

7. CONCLUSION

In this project we experienced with Spectral Clustering algorithm. We started from the basic version of algorithm and parallelized the process on Spark. In short, compared to k-means which provides a fast clustering centroids searching technique, Spectral Clustering gives much better results but has memory and runtime bottleneck for large dataset. The memory bottleneck happens when constructing the affinity matrix. We showed that using k nearest neighbour could effectively reduce the memory overhead, and other solutions exist such as threshold pruning or Nystrom approximation. We successfully parallelized the Spectral Clustering to get speedup on multiple nodes on Spark. In this project, we parallelized each stage of the algorithm: we used MapReduce to build the affinity matrix in parallel; for the eigenvector decomposition, we took advantage of the existing Spark MLlib SVD routine, which internally used ARPACK for fast computation; we distributed the computations of k-means on multiple nodes to get a faster converge. We also learned a lot of Spark during the project. It provided us with a friendly interface for running distributed jobs, and a good abstraction and scheduling of parallel workloads.

8. REFERENCES

[1] Amazon. Amazon spot instance. [Online; accessed on 10-May-2015].

[2] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[3] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[4] P. S. Bradley and U. M. Fayyad. Refining initial points for k-means clustering. In *ICML*, volume 98, pages 91–99. Citeseer, 1998.

[5] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, and E. Y. Chang. Parallel spectral clustering in distributed systems. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(3):568–586, 2011.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] Y. Ding, G. Zhu, C. Cui, J. Zhou, and L. Tao. A parallel implementation of singular value decomposition based on map-reduce and parpack. In *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, volume 2, pages 739–741. IEEE, 2011.

[8] C. Fowlkes, S. Belongie, F. Chung, and J. Malik. Spectral grouping using the nystrom method. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(2):214–225, 2004.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[10] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Applied statistics*, pages 100–108, 1979.

[11] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(7):881–892, 2002.

[12] S. S. Keerthi and C.-J. Lin. Asymptotic behaviors of support vector machines with gaussian kernel. *Neural computation*, 15(7):1667–1689, 2003.

[13] S. S. Khan and A. Ahmad. Cluster center initialization algorithm for k-means clustering. *Pattern recognition letters*, 25(11):1293–1302, 2004.

[14] M. Lichman. UCI machine learning repository, 2013.

[15] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[16] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[17] MLlib. Spark mllib. [Online; accessed on 10-May-2015].

[18] MNIST. Mnist digit image dataset. [Online; accessed on 10-May-2015].

[19] Spark. Spark cluster overview. [Online; accessed on 10-May-2015].

[20] U. Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[21] D. Yan, L. Huang, and M. I. Jordan. Fast approximate spectral clustering. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 907–916. ACM, 2009.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.