Streaming and Parallelized Coresets construction and its applications

Wei Ma
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA
weima@cmu.com

Max(Cong) Ma
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA
cma1@andrew.cmu.com

ABSTRACT

Coreset has been a promising topic in machine learning recently, as it reduces the size of original dataset without losing much information about it. In this project, we carefully reviewed the theory and framework of coresets construction for different machine learning tasks. We spot the limitations of the conceptual parallel computation framework for piratical implementation. Then we propose an asynchronized architecture for coresets construction to overcome the limitations of the conceptual framework. The proposed framework is implemented by Message Passing Interface (MPI) and is suitable for different coresets construction methods. The SVD based coreset for projective clustering and adaptive sampling based coresets for mixture model is implemented in this project. We ran the implemented coresets construction model on fake data to prove its efficiency. We also ran the coreset construction algorithm with several pop-ular image datasets, and demonstrated its accuracy by running thek-mean algorithm on both the original dataset and the coreset.

Keywords

Coresets, MPI, parallel computation, PCA, adaptive sampling

1. INTRODUCTION

Recent years have witnessed an explosion in the amount and dimension of data from various sources. Due to the huge "volume" and "velocity"[1] of the data being produced, learning from these datasets requires infeasible amount of computational power. Though various streaming algorithms are proposed to deal with large large amount data flow, the storage of the data is still a serious problem. Also, once we want to adjust the parameters from the established model, the model need be re-run on the whole dataset, making a difficulties on validate different models.

Different approaches are proposed to tackle the raised difficulties, pruning redundant data before learning algorithm is one of the promising methods to reduce computational time. So the idea becomes find a sketch of data that can represent the features of the whole dataset.

A coreset is defined as a sketch of the dataset which yields $(1+\varepsilon)$ approximation to the original dataset. It is originated from the researches of point sets measures, which indicate that a small number of data set S can approximate the measures of whole point sets P. Note S is not necessarily a subset of P, where we refer S is a strong coreset of P. Mathematically,

$$(1-\varepsilon)\mu(S) \leq \mu(P) \leq (1+\varepsilon)\mu(S) \tag{1}$$

where $\mu(\cdot)$ is some certain measure of the datasets and ε is the error bound. Coresets have great potentials to serve as a data reduction scheme. It also provides a generalized framework for streaming and parallelized computation of various classical machine learning methods such as k-means, PCA and mixture models.

The idea of coresets originated from the computational geometry, which tries to find a subset of data points to represent the whole data cloud. Coresets is a general concept that includes various construction methods such as sampling, feature extraction and ε-samples. It is already used in lots of distance based clustering methods such as k-means, k-median [7] and low rank approximation [4] methods. The construction methods are also different since we can re-generate the data points (strong coreset) and can also select "important" data points from original dataset (weak coreset).

This project investigates the problem of implementing an efficient coreset construction framework for different coreset variants. We review the the conceptual streaming and parallel coreset construction framework by [5, 6] and discover its limitations. We further extend the conceptual model to an asynchronized coresets updating framework which will full utilize the computation power under distributed settings. We implement the proposed framework using Message Passing Interface (MPI) and test it for different machine learning tasks. Finally we test our implementation on both fake and real datasets to validate the its efficiency and accuracy.

2. CORESETS VARIANTS

For different machine learning tasks, the proposed methods to construct coresets are different. In this project we discuss two types of coresets: one is for projective clustering using principal component analysis (PCA) [6] and the other is for mixture model using adaptive sampling [5]. Note the projective clustering is actually L^2-distance based and mixture model is log-likelihood based, which is also a weighted L^2 distance, so our project focuses on the L^2-distance based coresets. However, our implementation is friendly to all the coreset construction methods as far as the authors know.

2.1 (j,k)-coreset for projective clustering

A (j,k)-coreset for projective clustering is defined as a small set of points that yields a $(1+\varepsilon)$-approximation to the sum of squared

[1] http://www.gartner.com/newsroom/id/1731916

DOI: 10.1145/1235

distances from the n rows to any set of k affine subspaces, each of dimension at most j. For example, $(0, k)$-coreset is for k-means clustering and $(j, 1)$-coreset is for PCA analysis. Due to [6], we have the following theorem.

THEOREM 1. *For every set B which is the union of k affine j-subspace of R^d, we can find a set Q in R^d and a weight function $w : Q \to [0, \infty)$ and a constant $c > 0$ such that*

$$(1-\varepsilon)\sum_{p\in P} dist^2(p,B) \le \sum_{p\in Q} w(p)dist^2(p,B) \le (1+\varepsilon)\sum_{p\in P} dist^2(p,B) \quad (2)$$

where

$$|Q| = \begin{cases} \mathcal{O}(j/\varepsilon) & \text{if } k = 1 \\ \mathcal{O}(k^2/\varepsilon^4) & \text{if } j = 0 \\ poly(2^k \log n, 1/\varepsilon) & \text{if } j = 1 \\ poly(2^{kj}, 1/\varepsilon) & \text{if } j, k > 1 \end{cases} \quad (3)$$

and dist(p, B) is denoted as the minimum Euclidean distance from point p to dataset B.

The theorem indicates that we can approximate the original dataset B by a fixed size dataset Q. Surprisingly, the size of Q is independent with the data dimension d, making it great potentials for dealing with high dimensional data. Also we can observe that the size of Q is at most of $(\log n)$, which means we can shrink the size of original dataset significantly.

It is also proved that Singular Value Decomposition (SVD) can be used to find such dataset Q, Q is actually a low-rand matrix approximation of B. Conceptual, we can use the top Q eigenvalues ant their corresponding eigen-vectors to approximate B, which is exactly the coreset Q.

2.2 Coresets for mixture model

Different from coresets construction for projective clustering which measures the data points by sum of squared distance, coresets for mixture model is measured by the weighted log-likelihood of the data. We define

$$\phi(D|\theta) = -\sum_{j=1}^{n} \ln \sum_{i=1}^{k} \frac{w_i}{Z(\theta)\sqrt{|2\pi\Sigma_i|}} \exp\left(-\frac{1}{2}(x_j-\mu_i)^T\Sigma_i^{-1}(x_j-\mu_i)\right) \quad (4)$$

where $Z(\theta) = \sum_{i=1}^{k} \frac{w_i}{\sqrt{|2\pi\Sigma_i|}}$ is a normalizer, k is number of Gaussian components, n is number of data, (μ_i, Σ_i) is the parameters for ith Gaussian component. Then it can be proved that we can find a weighted data set C such that with probability $1 - \delta$,

$$(1-\varepsilon)\phi(D|\theta) \le \phi(C|\theta) \le (1+\varepsilon)\phi(D|\theta) \quad (5)$$

The size of C is proven to be $\mathcal{O}(dk^3/\varepsilon)$, and it is independent with the number of original dataset n. This indicates that if we have enough number of data, we can preserve a good enough coreset and never need to update the coreset even new data come.

The construction of coreset is achieved by the adaptive sampling, which obtains a weak coreset of original dataset. The sampling concept is to find sparse data points that can represent the whole data set. As shown in Figure 1, basically we can sample one data point (red point) and then discard the data points (blue points) nearest to the sampled data points. There are more advanced techniques to ensure the good approximation, but in the project we will use this sampling methods. Similar weak coreset construction technique for k-means and k-median can be found in [7].

3. COMPUTATIONAL ARCHITECTURE

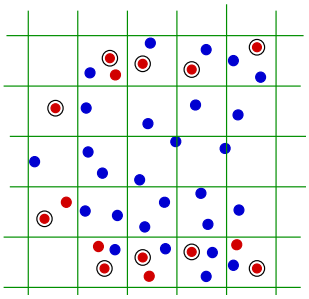

Figure 1: Illustration of adaptive sampling, from [2]

One exceptional feature of coreset is that it is close with union operation, which indicates that we can calculate the coresets for different data shards and then union them together to get the coreset for the whole dataset. Such feature makes coreset friendly to streaming/parallel computation framework. Streaming construction for coreset can be achieved naively, since we can store the calculated coresets in disk and finally merge them together. For more advanced techniques, since the size of coresets is fixed, we can store them all in memory, if the order of computation is managed properly [6].

A tree-based "merge-and-reduce" framework are proposed by different researchers [3, 1, 6], it conceptually illustrates the feasibility of streaming and parallel construction of coresets. We first review the tree-based construction process, then discuss its limitations for real implementation and then proposed an pratical framework for asynchronized construction of coresets.

3.1 Conceptual tree based architecture

Figure 2 presents the structure of a tree-based coreset construction framework. Each node of the tree contains a coreset of a shard of whole dataset, and the root contains the coreset for the whole dataset. The coresets in each node is of size m, they can be constructed from bottom to top, and in each level each coreset can be constructed in parallel.

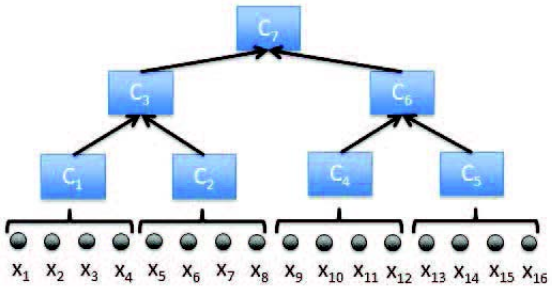

Figure 2: Tree construction for coresets, from [6]

If we define the leaf to be level 0, then for level i, the coreset at that level is the coreset for $\lceil n/2^i \rceil$ data points, so the tree has at most $\lceil \log n \rceil$ height. So for streaming algorithm, when a new coreset comes, we can identify its level, merge it to current coreset and send the merged coreset to the upper level. So for each new coreset, we need at most $\lceil \log n \rceil$, therefore the overall computational time is $\mathcal{O}\left(\lceil \log n \rceil \lceil \frac{n}{m} \rceil \alpha(m)\right)$, where $\alpha(m)$ is the computational time for merging two coresets with size m.

And for the parallel computation, we can first stream the data

with size m for each coreset at the lowest level, then we merge all the coresets at the same level, we keeping merging the coresets till we finally have one coreset at the top level. If we have M machines, the total computational time will be at most $\mathcal{O}\left(\lceil \log n \rceil \lceil \frac{n}{mM} \rceil \alpha(m)\right)$. Note if $M \geq n/m$, the computation in each level can be finished in one run.

To calculated the coresets for data shards and them merge them together through the tree-based architecture, we need to determine the proper size m of the coreset in each node so that we can ensure the accuracy of the merged coresets. Then we have the following theorem,

THEOREM 2. *To achieve $1+\varepsilon$ approximation through the tree based architecture coresets constrution methods, for each tree node we need to achieve $1+\varepsilon'$ accuracy, where $\varepsilon' = \mathcal{O}(\varepsilon/\log n)$, where n is the total number of the data points.*

As can be seen from the above theorem, we need a stricter criterion for the leaves so that we can get the desired approximation rates at the root, but the increase of ε is only of $\mathcal{O}(\log n)$. However when we want to design the real implementation, we find the tree based architecture has some limitations.

3.2 Limitations in practical implementation

As the tree based architecture is a all-reduce framework, it is not very friendly to Hadoop/Spark since both frameworks are suitale for the high I/O but lower computational intensity tasks, while for coresets construction I/O is low but computational burden is heavy. So it is more related to high performance computing (HPC), which is friendly to MPI, we will discuss the feature of MPI later. Also the tree based architecture has some limitations for practical distributed systems even using MPI, for example

- If the number of machines M is large, then at the higher level of the tree, some of the machine will not be assigned with task since the number of node is less than M.
- Since the number of node is too large, we need to store the computed coreset in hard disk and read them again for next computation. This involves heavy I/O and communication costs.
- The distributed systems is not only about multi-machine calculation, but also about multi-core calculation. It's not efficiently to use multi-process to read different data shard, so the proposed methods is not suitable for multi-core architecture.

To solve these limitations, we want to propose a framework that can construct the coreset in memory and make full use of the computational resources. After careful design, an asynchronized coreset construction framework which absorb the advantages of both streaming and parallel methods in tree-based setting is proposed. We argue that we do not necessarily need to construct the coreset for level i then level $i-1$, instead we can update them in an asynchronized manner.

3.3 Asynchronized architecture

In the asynchronized setting, we first fix the size of each coresets to be m, then each node need the memory $c(m) = m \times d \times$ sizeof$(float)$. If we have C memory can be used to store coresets in each core , we can assign $\lfloor C/c(m) \rfloor$ slots to store coresets. Note the slots is analogues to the tree node in the tree based architecture, each slot contains a coreset at certain level.

Also we need to build two hashmaps S and F to keep track of the information of each coreset slot and the infomation of each file reader handlers, the information for slots includes: level, status, location and the information for file includes: status, current pointer. The status indicates whether the slots/file is empty, being edited or storing the valid coreset, location of the newest data since the data is in the memory of one machine, current pointer indicate the location where current file is read at. The size of H is $3\times$#slots$\times$sizeof(int) and size of F is $2 \times$ #files $\times$ sizeof(int). Note we only need to synchronized on H and F, the memory of real slots will not be synchronized unless necessary.

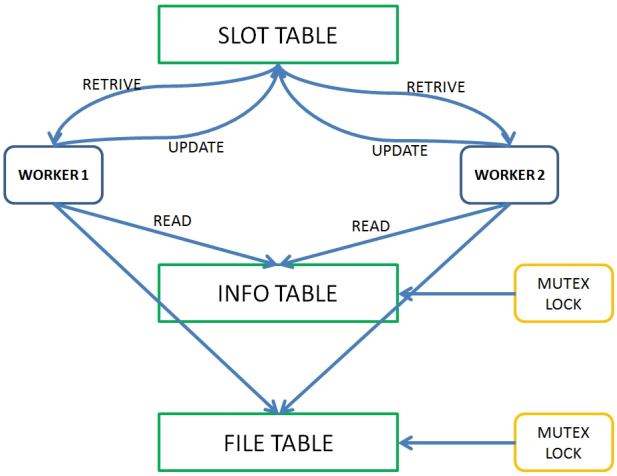

Figure 3: Asynchronized architecture for coresets

We assume our distributed setting is a multi-machine and multi-core environment, and each machine contains one shard of dataset. Idealy, we want one core in each machine reads the data and other cores merge the slots. So for each process for any of the machine, it will conduct one of the following tasks:

- if exists some slots are empty, read from data shard and construct a coreset with lowest level, store the coreset locally and update the infomation table S and F.
- if exists two valid coresets at the same level, merge them to one new coresets, store the new coreset locally and update the hash table S.
- if all the data shard are read through and there is no two valid coresets at the same level, merge two coresets that have lowest levels, store the new coreset locally and update the hash table S.

An illustration of the asynchronized architecture is presented in Figure 3, as can be seen the worker will automatically decide what to do based on the information in FILE TABLE F nad INFO TABLE S, then it will retrive data from the slot table and merge the data, finally it will update F and S to share the information with other workers. Note the worker will only update S since will will change the location to its own memory, it will not update the slots in the remote memory. This procedure will largely reduce the communication time since the INFO TABLE is much smaller.

By keeping conducting the three tasks, there will be only one valid coreset remaining in the coreset slots when all the workers have nothing to do, and it will be the coreset for the whole dataset. As can be seen, the asynchronized architecture for coreset construction combines the data reading and coreset merging tasks together,

and each process will dynamically determine which tack to conduct.

3.4 Synchronization issues

For the synchronization problem of this architecture, as we mentioned above, we only need to synchronize the INFO TABLE S and FILE TABLE F among different processes. The coresets will be stored in different cores, they will be communicated only when necessary. For each worker, it should block S and F at very beginning and decide what to do, then it should update the table to indicate it is now in charge of some slots. For example, if worker 1, it decides to read data from file 1, then it will update the status of file 1 to be busy and also the corresponding slot to be busy. After merging/reading the slots, the worker should also update the F and S to release control of the slots and provide the new information to other workers. Since the blocking only happens on updating the table S and F, both table are small (hundreds bytes), so the blocking time is negligible. Note all the block should be mutual exclusive blocking rather than read/write blocking.

3.5 Discussion

We raise some points that are important for the implementation:

- There are some implications to choose the priority of these three tasks if more than one task is available. The intuition suggests that the top priority should be given to reading the data from file. After the experiments on the real data, we find reading data is much faster than merging the slots, so we can read as much data as we can and then use the full computational power to merge the slots.

- We need to notice that the third task is used for dealing with the boundary conditions when the number of slots is not power of 2, but it will be conducted during the execution due to the different race conditions of the workers. The third task actually violates the Theorem 2 since the slots are combined aggressively, therefore we need to be aware of the choice of the total number of slots to reduce the occurrence of third task. The more slots we have, the less opportunity for third task to happen.

- When retrieving the data from other processor, the worker should read the INFO TABLE S first and then decide to get the newest data from which worker. The reason is that each worker has a version of the slots data, but only one is the newest, so don't use the old slots.

In next section we will discuss in detail how we implement the model and how we organize the codes.

4. IMPLEMENTATION DETAILS

In last section, we discuss the concept of the asynchronized architecture for coreset construction. In this section we will discuss the detail of how the concept is realized. Generally, we use Open MPI C++ as the programming framework and interface based design as the code organization pattern, we will discuss them separately.

4.1 MPI Features

The proposed asynchronized architecture can be implemented by Message Passing Interface (MPI). Specifically, we use its C++ implementation Open MPI[2] to implement the proposed framework in our project. Open MPI provides bunch of powerful methods for message communication between processes. Especially, we use one-sided communication model in MPI 2.0 protocol, which allows remote memory access and shared file handlers.

Remote memory access is the most important feature for MPI 2.0 protocol, which allows one-sided communication. That is to say, processor 1 can open a memory window to other processors, then other processors can visit that memory window without interrupting the processor 1. This is different from the Send/Recv model in MPI 1.0 since the data retrieval process is non-blocking. As shown in Figure 4, MPI 2.0 uses Get/Put to retrieve/update data from/to other processes, and other memory can only access/update the memory inside the window.

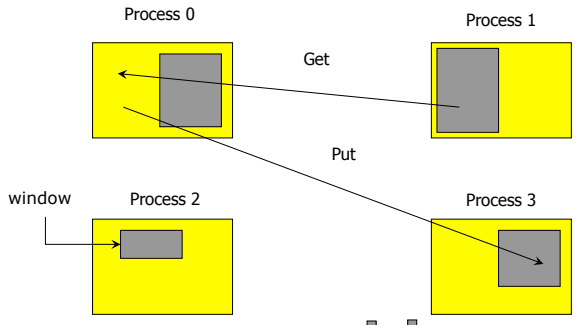

Figure 4: MPI One-sided communication, from William Gropp

In our implement ion, all the slots retrievals are implemented by Get and all the TABLE update are implemented by Put. There are corresponding lock/unlock function for each window since the Get/Put process is non-blocking.

4.2 Interface based code structure

Our codes are written in a mixed C/C++ style, the class is used to describe worker and dataset. The whole implementation is interface based, which means that we can easily add need coreset construction methods in the implementation. Here we simply describe the major components of the implementaiton.

- *coreset.cpp*: It is the main function of the implementation, it will intiallize the MPI and read the configuration files. It will allocate the memory for each processor and keep record of the main logic.

- *worker.h*: The worker class describe the feature of each worker, it will automatically decide what to do by reading the INFO TABLE and conduct the task and finally decide whether to terminate itself or not.

- *data.h*: Data class is an abstact layer of the real data, it can process data from different files, and worker can simply get data by calling "fillslot".

- *combiner.h*: Combiner describes how we merge two slots into one. It is the core operation in the coreset construction, but it can be replaced by ohter methods very easily.

In our project we will implement the SVD and adaptive sampling for projective clustering and mixture Gaussian models. Several open sourced libraries are used for example Eigen[3] for matrix computation and Redsvd[4] for fast singular value decomposition. In

[2]https://www.open-mpi.org/

[3]http://eigen.tuxfamily.org

[4]https://github.com/ntessore/redsvd-h/blob/master/include/RedSVD/RedSVD-h

following sections we will conduct different experiments to show the efficiency and accuracy of our implementation.

5. FAKE DATA TEST

In this section we test our implmentation on a generated faka dataset, the dataset is dense and each entry is a random number from 0 to 100. The experiments are conducted on a server machine with Intel(R) Xeon(R) CPU L5420@2.50GHz, 8 cores, 64-bit, 16GB memory. We run the test on different methods, dimension d, dataset size n, coreset size m, slot size x and number of processor np, the results are presented in Table 1.

Method	d	n	m	x	np	Running time(s)
SVD	100	1000	100	20	1	4.99
SVD	100	1000	100	20	2	4.364
SVD	100	1000	100	20	4	1.884
SVD	100	1000	100	20	8	2.476
SVD	100	1000	100	20	16	6.58
SVD	100	1000	100	20	32	22.75
ADS	100	1000	100	20	1	0.276
ADS	100	1000	100	20	2	0.236
ADS	100	1000	100	20	4	0.235
ADS	100	1000	100	20	8	0.93
ADS	100	1000	100	20	16	4.00
ADS	100	1000	100	20	32	10.83
SVD	100	10000	100	20	1	48.389
SVD	100	10000	100	20	2	30.647
SVD	100	10000	100	20	4	18.579
SVD	100	10000	100	20	8	8.788
SVD	100	10000	100	20	16	24.649
SVD	100	10000	100	20	32	38.132
ADS	100	10000	100	20	1	1.805
ADS	100	10000	100	20	2	1.068
ADS	100	10000	100	20	4	0.697
ADS	100	10000	100	20	8	1.577
ADS	100	10000	100	20	16	23.706
ADS	100	10000	100	20	32	75.89
SVD	100	100000	100	20	1	286.19
SVD	100	100000	100	20	2	160.8
SVD	100	100000	100	20	4	91.13
SVD	100	100000	100	20	8	57.33
SVD	100	100000	100	20	16	86.25
SVD	100	100000	100	20	32	136.53
ADS	100	100000	100	20	1	16.91
ADS	100	100000	100	20	2	8.73
ADS	100	100000	100	20	4	4.53
ADS	100	100000	100	20	8	6.67
ADS	100	100000	100	20	16	206.7
ADS	100	100000	100	20	32	635.6
SVD	100	100000	100	10	8	70.96
SVD	100	100000	100	50	8	66.12
SVD	100	100000	100	70	8	69.21
SVD	100	100000	100	100	8	84.9
SVD	100	100000	100	200	8	139.91

Table 1: Fake data test on differnt dataset, number of processors, slot sizes

We also visualize some of the results in Figure 5
We can raise the folloing oberservations:

- Running MPI with 8 processors will achieve the best performance, this exactly matched the number of cores of the machine. IF the number of processor exceed the number of cores, the execution time will increase due to the overhead and extra scheduling costs.

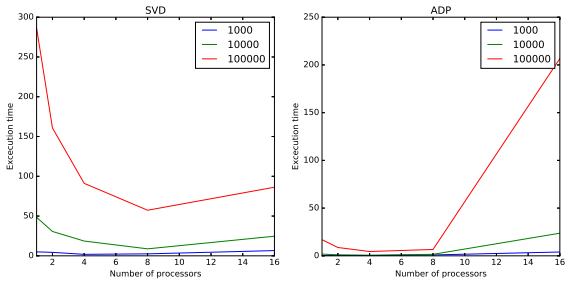

Figure 5: Runing time on different data set

- The adaptive sampling (ADS) method runs much faster than SVD since SVD requires large computational resources when dimension is high.

- The parallel framework has little benefit for the small dataset, but will significantly reduce the execution time of large dataset.

- Increasing the number of slots will increase the execution time since the communication/blocking costs will increase. But as we discussed earlier, more slots will lead to more accurate coreset, so there exists a trade-off on the choice of slots number.

6. EXPERIMENTS ON REAL DATA

In order to demonstrate the accuracy of coreset and also to visualize how it actually works, we run the coreset construction model on different image datasets. Here we present the result obtained from running on MINST[5] and CIFAR-10[6].

6.1 MNIST

The MNIST dataset contains images of handwritten digits, which has 60,000 training examples and 10,000 testing examples. In this project we used a subset of the dataset, and has 1797 examples. It contains roughly the same number of exampels for each digits.

We run both the SVD and ADS algorithms with coreset size 30. We then picked the top 10 rows from the coresets and visualize them below. The result returned from both algorithms are all recognizable digits. There are a few things to note here.

- SVD is an algorithm that constructs weak coresets, which means it selects examples of the original dataset that're most represetable, and ADS is an algorithm that constructs strong coresets, which means it generates result based on the dataset. Therefore, the ADS coreset seems more blured.

- The original data is in order while the generated coreset by ADS is not, which indicates may because it is easy to distinguish such as 8. But for SVD, the feature and order of each digit are maintained.

- ADS coreset contains nearly every digits, whereas SVD coreset has repeated 3 in it. However, this doesn't necessarily mean SVD is worse than ADS. In fact, when we later

[5]http://yann.lecun.com/exdb/mnist/
[6]https://www.cs.toronto.edu/ kriz/cifar.html

compare the k-mean centroid distance between the original dataset and the coreset, SVD coreset is closer to the orignal dataset.

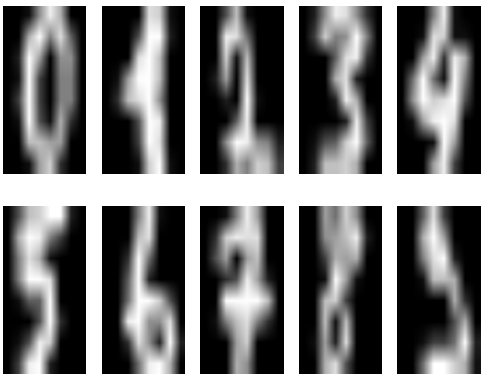

Figure 6: Example of MNIST data

Figure 7: Top rows of ADS coreset on MNIST

We investigated how the coreset size affects its performance, and we found that increasing the coreset size will improve its performance generally, that in this example, smaller distance between the k-mean centroids. This is what we expected. Because larger coreset is able to preserve more information about the original dataset. The graph below shows the decreasing trend of the centroids distance with increasing coreset size. The algorithm has some randomness inside so the graph isn't a perfect straight line. In addition, we see that SVD always outperform ADS no matter how large the coreset size is.

6.2 CIFAR-10

The CIFAR-10 are labeled subset of 80 million tiny images dataset. In this project, we used a subset of the data, which has 10,000 examples. It consists of 32x32 colour images in 10 classes, with 1000 per class.

Again, we run both the SVD and ADS algorithm with coreset size 30. We then picked the top 10 rows from the coreset and visualize them below. Unlike the results of MNIST, which all resembled the digits, the SVD coresets of CIFAR-10 are messy and not

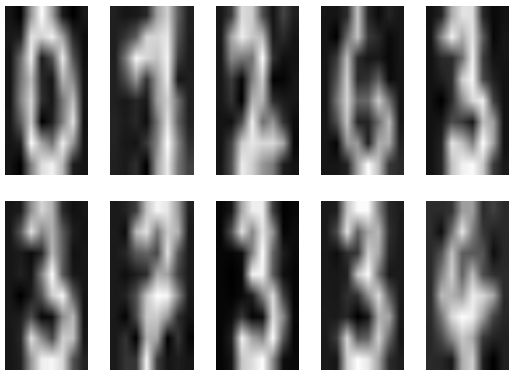

Figure 8: Top rows of SVD coreset on MNIST

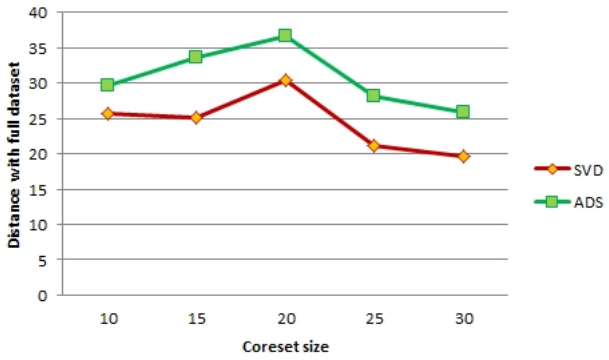

Figure 9: Performance vs. Coreset size

recognizable. We think this is due to the more complicated structure of the image of CIFAR-10, that even the closest images don't resemble each other. In fact, vanilla k-mean works pretty well on the MNIST, achieving 95% accuracy, but it only achieves less than 60% on the CIFAR-10 dataset. This shows that a vannila square-distance based algorithm isn't suitable for this problem. However, we can still use other machine learning algorithms on the coreset, like Gaussian-Mixture, whose calculation is based on likelihood that is also preserved in the coreset.

7. SUMMARY

Coresets have been an interesting topic in machine learning recently, as it can be used to reduce the size of original dataset without losing much information about it. An important property of coreset that stimulates this project is that the coresets are closed under union, thus we can construct the coreset of a batch of data in parallel, then aggregate them in an all-reduce manner.

In this project, we carefully reviewed the theory and concepts of coresets construction for different machine learning tasks. We spot the limitations of the conceptual parallel computation framework for practical implementation. We propose an asynchronized architecture for coreset construction to overcome the limitations of the conceptual framework.

We implemented the proposed coresets construction framework by Open MPI C++, which successfully utilizes the power of parallelism, as we show that with an increasing number of cores involved in the computation, the speed of computation increases significantly. We take advantage of the advance feature of MPI 2.0

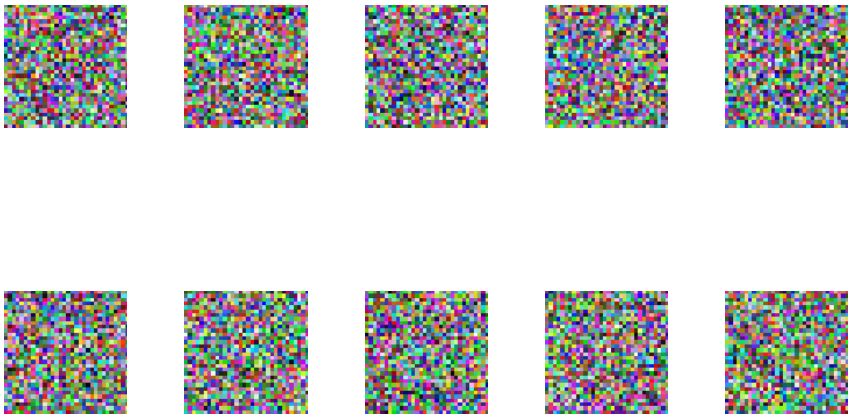

Figure 10: Top rows of SVD coreset on CIFAR-10

and largely reduce the blocking/communication costs.

We also ran the coresets construction algorithm with several popular image datasets, and demonstrated its accuracy by running the k-mean algorithm on both the original dataset and the coresets. The results indicates the great potential of coresets as a data reduction and preservation tool in the future. We hope our project can enlighten researches on developing practical framework of coresets construction.

Acknowledgments

We would like to thank the instructor of the course Prof. William Cohen for his advice on this project, and Prof. William Gropp's online lecture notes about high performance computing in MPI.

8. REFERENCES

[1] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Approximating extent measures of points. *Journal of the ACM (JACM)*, 51(4):606–635, 2004.

[2] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Geometric approximation via coresets. *Combinatorial and computational geometry*, 52:1–30, 2005.

[3] J. L. Bentley and J. B. Saxe. Decomposable searching problems i. static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[4] A. Deshpande, L. Rademacher, S. Vempala, and G. Wang. Matrix approximation and projective clustering via volume sampling. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 1117–1126. Society for Industrial and Applied Mathematics, 2006.

[5] D. Feldman, M. Faulkner, and A. Krause. Scalable training of mixture models via coresets. In *Advances in neural information processing systems*, pages 2142–2150, 2011.

[6] D. Feldman, M. Schmidt, and C. Sohler. Turning big data into tiny data: Constant-size coresets for k-means, pca and projective clustering. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1434–1453. Society for Industrial and Applied Mathematics, 2013.

[7] S. Har-Peled and S. Mazumdar. On coresets for k-means and k-median clustering. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 291–300. ACM, 2004.

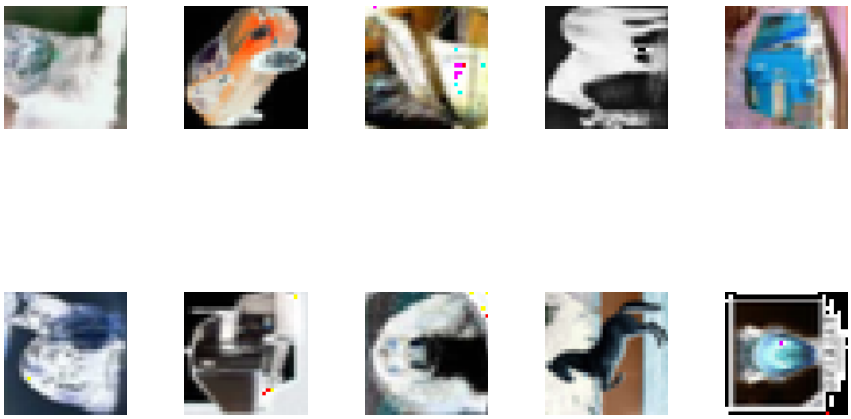

Figure 11: Top rows of ADS coreset on CIFAR-10