A database-based distributed computation architecture with Accumulo and D4M: an application of eigensolver for large sparse matrix

Yin Huang, Yelena Yesha, Shujia Zhou
Computer Science and Electrical Engineering
University of Maryland, Baltimore County
Baltimore, MD, 21250
Email: {yhuang9,yeyesha,szhou}@umbc.edu

***Abstract*—NoSQL distributed databases have been devised to tackle the challenges resulting from volume, velocity and variety of big data. Graph representation of datasets requires efficient distributed linear algebra operations for large sparse matrix constructed from big data. Storing the transformed matrix into the database not only speeds up the big data analysis process but also facilitates the computation because of indexing. The Hadoop based approach does not natively support iterative algorithms due to data shuffling during each iteration.**

This paper presents a novel database-based distributed computation architecture bridging the gap between Hadoop and HPC. The novelty results from exploring the indexing capability of D4M (Dynamic Distributed Dimensional Data Model) to support linear algebra operations in a distributed computation environment. The idea is to store input data and intermediate results in associative array format inside Accumulo table to facilitate the data sharing among working nodes. pMatlab is deployed as the parallel computation engine. Our proposed architecture is proved to be lighter, easier and faster than MapReduce based approach. One example application is calculating top k eigenvalues and eigenvectors for large sparse matrix. Experiments on Graph500 benchmark datasets demonstrate 2X speedup of our architecture as compared to HEIGEN (An eigensolver for billion-scale matrices using MapReduce).

***Keywords*—Big Data Analytics; Distributed Computation Architecture; Eigendecomposition; Apache Accumulo; D4M**

I. Introduction

Typical data analytics normally include the following pipeline: collecting data, querying data, analyzing data and report. The large volume of unstructured data has significantly complicated this pipeline. Typical example of unstructured data comes from social media such as Facebook, Twitter etc. First we need a system to store and query data of interest. Second, we transform raw data into numerical format for analysis. Third, we can apply machine learning algorithms to discover patterns for better decision making. Nowadays Hadoop [1] plays a fundamental role in tackling the challenges caused by increasing volume and velocity of unstructured data. The success of Hadoop relies on its two components: Hadoop Distributed File System (HDFS) for storing massive amount of data with redundancy for failure tolerance and MapReduce for batch processing. Hadoop, however, fails to provide an easy-to-use interface to retrieve data of interest for HDFS. Apache Hive [2] has been designed to facilitate querying and managing large datasets residing in distributed storage with SQL-like language. But the response time for random, real-time read/write access is relatively slow because of MapReduce. This leads to the development of distributed, scalable, high performance data storage and retrieval system such as Accumulo [3] and HBase [4]. Furthermore, more advanced algorithms for big data analytics typically require distributed linear algebra operations, which are not natively supported in Hadoop. Examples are spectral clustering for graph analysis [6], and large scale distributed deep networks [28]. Therefore, it is imperative to create a database-based distributed computation architecture which not only manages unstructured data but also supports high performance linear algebra operations.

High Performance Computing (HPC) community has been tackling the second challenges for decades but fails short in managing large unstructured data. The question is how can we integrate traditional HPC with Hadoop. In [27], Glenn Lockwood of the San Diego Supercomputer Center came up with a list of reasons why Hadoop remained on the fringe of HPC today. This paper serves to bridge the gap between Hadoop and HPC. The proposed architecture deploys NoSQL database, Accumulo for example, which is built on top of Hadoop, as underlying data management system. Meanwhile, pMatlab [16], a parallel MPI library for Matlab, is deployed as the parallel computation engine. D4M [9] serves as the data structure to retrieve data from Accumulo and to feed into pMatlab for computation in a distributed manner. For example, in many applications, it is intuitive to represent data as a graph to discover patterns. Graph representation has a wide range of applications from social sciences to physics and bioinformatics [5], [6], [7]. Take community discovery in social network [8] for instance, a sparse adjacency matrix for all Twitter users can be built to reflect their relationships. The adjacency matrix construction, however, requires complicated operations; moreover, to find users of similar interests, we need apply spectral decomposition on the matrix. Both of which are not efficiently supported in MapReduce based computing model because the matrix size easily exceeds million or billion. Recent work has focused on constructing graphs from the data stored in the D4M format and applying eigendecomposition to the modularity matrix [10].

978-1-4799-9926-2/15/$31.00 ©2015 IEEE

However, the authors in [10] only use D4M to construct adjacency matrix and store data on a single database node which will become the bottleneck as data size increases. Our architecture differs from theirs in that we extend D4M to numerical calculations for linear algebra operations. Furthermore, we parallelize the computation across the whole cluster instead of a single node to deal with million or billion scale matrices.

Sparse matrix has many real life applications. In general, real life applications such as social media tend to produce matrix with sparseness ranging from 0.01% to 1% . The biggest difference between sparse and dense matrix is the unbalanced distribution of entries. Such difference may cause significant performance problems in distributed computation where the distribution of work loads are imbalanced. pMatlab [16] provides such a parallel library that integrates MPI and Matlab. In pMatlab, the main process spawns working processes evenly among working nodes, which will cause synchronization problems between parallel operations due to uneven work load will lead to different completion times for all processes. Therefore, in order to implement iterative graph algorithms involving sparse matrix operations, we need modify pMatlab to differentiate main process from working processes so that main process waits for the working processes instead of doing computation. In addition, load balancing among working nodes plays a significant role in speeding up the performance. A scheduler is in need to achieve this goal.

Spectral decomposition is of great importance in linear algebra and functional analysis for a wide variety of scientific and engineering applications, such as pattern recognition, image compression, and community discovery in social network. The essence of spectral decomposition is the calculation of eigenvalues and eigenvectors of a matrix. Recent platforms for eigensolvers based on MapReduce include Mahout and HEIGEN. Mahout implements Stochastic Singular Value Decomposition [22] (SSVD) algorithm while HEIGEN Lanczos-Selective Orthogonalization [24] (Lanczos-SO) algorithm. SSVD is reported to have a high memory footage therefore SSVD is not a good fit for large matrix. Lanczos-SO algorithm is a variant to Lanczos-No Orthogonalization (Lanczos-NO) which stifles the formation of duplicate eigenvectors due to the redundant copies of the outermost eigenvectors. Lanczos algorithms take an iterative approach to calculate eigenvalue and eigenvectors. MapReduce, however, is not adequate for iterative algorithms due to several respects. First, the shuffling of intermediate results between MapReduce jobs. Second, the unchanged input data must be reloaded and reprocessed at each iteration, wasting I/O, network bandwidth, and processor resources.

In this paper, we have devised a database-based distributed computation architecture. Our contributions are as follows.

1) The proposed architecture successfully bridges the gap between Hadoop and HPC.
2) We exploit the usage of D4M to numerical calculations for supporting linear algebra operations.
3) We tailor pMatlab source code to meet the need of iterative nature of machine learning algorithms for sparse matrices.
4) We build a scheduler by exploiting the statistical information from Accumulo table to achieve load balancing.

To evaluate the performance of our proposed architecture, we focus on spectral decomposition. Experimental results with Graph500 benchmark datasets demonstrate 2X speedup of proposed architecture as compared to HEIGEN [14].

The rest of the paper is organized as follows. Section 2 describes the components of architecture, discusses the data storage format and the graph construction procedure. Section 3 investigates Lanczos-SO algorithm in HEIGEN. Section 4 demonstrates our implementation of Lanczos-SO. Section 5 is for experimental results. Section 6 lists related work. Section 7 is a discussion and followed by a summary in section 8.

II. System architecture

Our system consists of the following 3 components: First, the bottom layer is Hadoop cluster and Accumulo database where the data are stored in the D4M format, which provides an easy to use interface for accessing subsets of data. We can thus build graphs representing various types of relationships in matrix format. After transforming unstructured data into matrix format, we directly store the matrix in Accumulo table for further processing. This significantly simplifies the data analysis pipeline because no matrix post processing is required. Second is the service layer containing Matlab and MatlabMPI, both of which provide the computation and communication resource to the upper layer. Third is the user layer where associative arrays query and store the data to be processed while pMatlab handles the parallel computation.

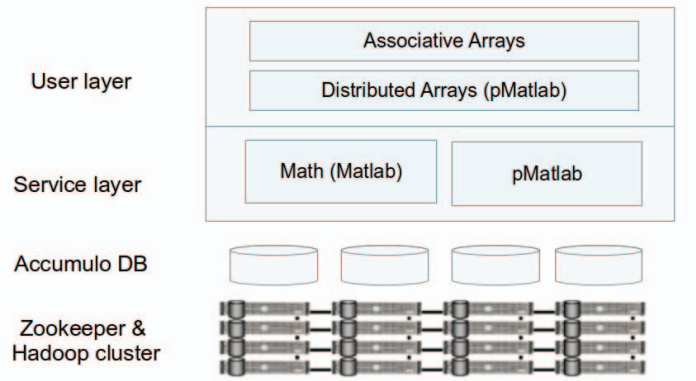

Fig. 1: System architecture: bottom layer is Hadoop cluster and Accumulo, second the service layer, third the user layer

A. Accumulo and D4M

Both Accumulo and HBase are examples of open-source distributed databases which offer scalable storage and retrieval, based on Google's BigTable design. BigTable design tackles challenges related to the volume, velocity and variety of data. Compared to HBase, Accumulo provides cell-level access control and features an architecture that leads to higher performance for parallel clients [23]. In addition, [29] shows Accumulo demonstrates high data ingestion rate and scales well. This is important because iterative algorithms involve multiple read/write operations for each iteration; the ingestion time and query response time determine the general performance.

In D4M, data are actually stored in a 2-dimensional associative array which provide a one-to-one mapping onto the tables in a triple store that makes the complex manipulations simple to code. Consider, for example, an associative array $Assoc('Tweet1','Status \mid 200')$ holding information about user's tweet Status. Associative array is a natural mapping from tabular key-value store to matrix coordinates by expanding the Tweet table.

TweetID	User	Status
Tweet1	Joe	200
Tweet2	Adam	200
Tweet3	Jane	301

TABLE I: A simple example with information about three tweets

Tweets in table I are stored in D4M format in table II. "1" simply means this cell exists and only non-zero cells are stored. $TedgeDeg$ in table III is the degree table which sums up the total number of entries for each column. We can rely on this table to understand the distribution of data and achieve load balancing. Our scheduler relies on this table to obtain the sparseness information.

TweetID	User| Joe	User| Adam	User| Jane	Status| 200	Status| 301
Tweet1	1			1	
Tweet2		1		1	
Tweet3			1		1

TABLE II: Tweets expanded in D4M schema

	User| Joe	User| Adam	User| Jane	Status| 200	Status| 301
Degree	1	1	1	2	1

TABLE III: TedgeDeg: a degree table containing the total number of entries of each column

D4M is originally designed to fast index and query string elements in a dataset. In this paper, we explore the query function of D4M to support numerical elements retrieval from a matrix table. To extract a subset of the data, we can index into ranges of the associative array just as is done with matrices in Matlab. D4M will return row, column and value vectors as string format. We first convert them to numbers and then rebuild the matrix by using Matlab *sparse* function. Fig. 2 shows an example of how D4M can be used to facilitate the retrieval of matrix elements as compared to Hadoop. In Fig. 2, a matrix has been stored in both Hadoop HDFS and Accumulo table. Normally we put text file with each line being a cell element of matrix to HDFS. In Accumulo, we store the original matrix into a table called *matrix* in associative array format. D4M provides a much easier and faster way to return data of interest than MapReduce. For example, to get the first row of matrix, a simple query like *matrix('1,',:)* is sufficient in D4M. For Hadoop, it takes a MapReduce task to scan the whole file system to find the result. Depending on the result size, the former takes around a few seconds while the latter takes a few minutes. In addition, our code size is also significantly shorter than HEIGEN. While HEIGEN has over two thousand lines of JAVA code, we have around three hundred lines. D4M makes the proposed architecture lighter, easier and faster than Hadoop.

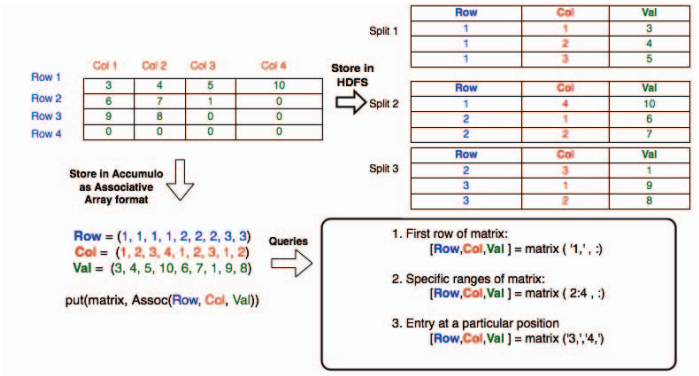

Fig. 2: D4M compare to Hadoop for matrix storage and manipulation

B. pMatlab

pMatlab is a parallel library of MPI for Matlab based on MatlabMPI for launching programs and communicating between processes, which thus provides process-level parallelization. pMatlab provides Single Program Multiple Data (SPMD) as parallel computation model. $Dmat$ object in pMatlab can help distribute a matrix across processes. Typical usage of pMatlab requires the programmer to specify the total number of processes in the whole cluster with a fixed number of machines. pMatlab will assign equal number of processes to each machine in the cluster. And each process is uniquely identified by a PID. The main process (PID=0) serves as the leader which spawns worker processes and will synchronize the computation when worker processes have completed their tasks. More details can be found in [16].

To schedule the job, It is straightforward to assign equal number of columns of matrix to each process. This naive approach, however, is not suitable for sparse matrix because imbalanced workloads will lead to undesirable performance. To overcome this problem, we have designed a scheduler to ensure that each process will handle more or less the same amount of workload. This scheduler is supported by degree table, $TedgeDeg$ table mentioned in previous section for example, which maintains the statistical information about the data distribution.

The fact that the main process will not start the synchronization stage until the command agg is executed will cause a problem. This problem arises when the main process has more workloads than other processes. Because the main process will ignore the synchronization signals from worker processes when it is still doing computation while worker processes have completed. Our solution is to modify MatlabMPI code to exclude the main process (PID=0) from doing any calculation as the main process has to wait for the completion of working processes (PID greater than 0). By excluding the main process, we ensure the synchronization of each parallel operation.

C. Graph Analysis

As long as we have constructed the graph, we could run different graph analysis algorithms. Spectral clustering is an unsupervised clustering approach which not only tolerates noisy data but also produces better accuracy than typical clustering algorithms such as k-means. In this paper, we focus

Algorithm 1. Lanczos -SO(Selective Orthogonalization)

Input: Matrix $A^{n\times n}$, random n-vector b, maximum number of steps m, error threshold ϵ
Output: Top k eigenvalues $\lambda[1..k]$, eigenvectors $Y^{n\times k}$

1: $\beta_0 \leftarrow 0$, $v_0 \leftarrow 0$, $v_1 \leftarrow b/||b||$;
2: **for** $i = 1..m$ **do**
3: $v \leftarrow Av_i$; // Find a new basis vector
4: $\alpha_i \leftarrow v_i^T v$;
5: $v \leftarrow v - \beta_{i-1}v_{i-1} - \alpha_i v_i$; // Orthogonalize against two previous basis vectors
6: $\beta_i \leftarrow ||v||$;
7: $T_i \leftarrow$ (build tri-diagonal matrix from α and β);
8: $QDQ^T \leftarrow EIG(T_i)$; // Eigen decomposition of T_i
9: **for** $j = 1..i$ **do**
10: **if** $\beta_i|Q[i,j]| \leq \sqrt{\epsilon}||T_i||$ **then**
11: $r \leftarrow V_i Q[:,j]$;
12: $v \leftarrow v - (r^T v)r$; // Selectively orthogonalize
13: **end if**
14: **end for**
15: **if** (v was selectively orthogonalized) **then**
16: $\beta_i \leftarrow ||v||$; // Recompute normalization constant β_i
17: **end if**
18: **if** $\beta_i = 0$ **then**
19: break for loop;
20: **end if**
21: $v_{i+1} \leftarrow v/\beta_i$;
22: **end for**
23: $T \leftarrow$ (build tri-diagonal matrix from α and β);
24: $QDQ^T \leftarrow EIG(T)$; // Eigen decomposition of T
25: $\lambda[1..k] \leftarrow$ top k diagonal elements of D; // Compute eigenvalues
26: $Y \leftarrow V_m Q_k$; // Compute eigenvectors. Q_k is the columns of Q corresponding to λ

Fig. 3: Lanczos-SO algorithm in [14]

on spectral decomposition techniques outlined in [14]. The primary goal of spectral decomposition is to find the top K eigenvalues and eigenvectors of our input graph. Fig. 3 is the Lanczos-SO algorithm that we are testing in proposed architecture. More details about Lanczos-SO can be found at [24]. The reason why we choose Lanczos-SO is as follows:

1) Lanczos method generally calculates top k largest eigenvalues.
2) Unlike Lanczos-NO, Lanczos-SO filters spurious eigenvalues given the selective reorthogonalizations (line 9-16).
3) The most costly operation is matrix-vector multiplication, which is computationally cheaper than matrix-matrix multiplication. HEIGEN is capable of dealing with billion-scale matrices.

III. Lanczos-SO in HEIGEN

HEIGEN implements Lanczos-SO algorithm based on Hadoop; every step in the algorithm is implemented as MapReduce jobs. MapReduce model is well-suited to a class of algorithms with an acyclic data flow, but it does not natively support iterative algorithms. Such algorithms can be expressed as multiple MapReduce jobs launched by a driver program, but this workaround imposes a performance penalty in every iteration because Hadoop MapReduce jobs must write their outputs to stable disk storage on completion therefore costly disk accesses for each iteration [15].

Consider, for example, line 3 in Fig. 3 the matrix and vector multiplication (MV) will read vector from previous output (line 21). Previous output, however, is actually stored to local disk where reducers are running. This incurs a lot of data exchange and thus slows down the whole calculation. In addition, the input matrix has to be reloaded every iteration in line 3 even though the data remain unchanged. It is shown in experimental section that MV takes almost 95% time of the whole computation.

To sum up, two major problems are as follows:

- The shuffling of intermediate results between MapReduce jobs.
- The unchanged input data must be reloaded and reprocessed at each iteration, wasting I/O, network bandwidth, and processor resources.

Both problems are caused by the fact that Hadoop relies on MapReduce to generate inputs for each slave node from HDFS in distributed computation environment which becomes inefficient when iterative reading occurs.

IV. Lanczos-SO with Accumulo and D4M

To address the problems listed in previous section, we describe how Accumulo and D4M can help speed up the computation.

Fig. 4 shows the steps for implementing Lanczos-SO using Accumulo and D4M. Step one, we run our scheduler based on the global data distribution table obtained in Accumulo when we upload input matrix. The goal is to find the row ranges for each process to achieve load balance. Step two, after we have obtained the ranges for each process, each process will copy corresponding row ranges of matrix to local disk. Step three, we start computing eigenvalues and eigenvectors in parallel. In contrast to HEIGEN, the first two steps are extra operations for better performance but they are executed only once.

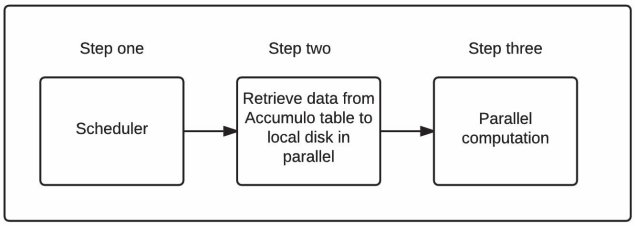

Fig. 4: Data processing steps for Lanczos-SO with Accumulo and D4M

Advantages of our proposed architecture are as follows:

1) We reduce the data shuffling to database read operations by writing intermediate results back to database. This significantly mitigates the traffic load when data shuffling occurs between iterations.
2) Unlike MPI based approach which requires message exchanges for data sharing, database-based approach eliminates the message passing among work nodes since all work nodes have access to the database.
3) By storing input matrix in D4M format, we can obtain the global distribution information which can be used to design a static scheduler for load balance. This significantly reduces the complexity for the architecture.
4) To avoid reloading the unchanged input data, we propose to load the data to working nodes' local disk,

since working node will operate on the same input data for each iteration.

Our scheduler works in the following way, first by dividing the total number of entries by the number of processes, we obtain the average load for each process.

$$Avg = \frac{Num\_Of\_Entries}{Num\_Of\_Processes} \quad (1)$$

Then we set marking ticks to divide the columns of matrix equally among working processes. These ticks are later adjusted so that average load is achieved for every working process.

Average main memory footage can be evaluated by the following formula.

$$MM = (\frac{\alpha * S^2}{N} + S) \times 3B \quad (2)$$

Where α is the sparseness of the matrix, S is the size of the matrix, N means the number of machines, 3 Byte for an associative array. For matrix with size 262,144, the main memory requirement is around $240MB$ with sparseness 1% and 8 machines. In real world applications, matrix sparseness generally ranges from 0.01% to 1%. In our experiment, we generate matrices with sparseness 1%.

V. Experimental Results

In order to test our architecture, we have set up a 16-nodes cluster connected with 10 Gbps InfiniBand switch with the following configuration. Each node has 2 quad-core CPUs with the model Intel(R) Xeon(R) CPU X5560 2.80GHz. Main memory is 24GB, L1d cache 32KB, L1i cache 32KB, L2 cache 256KB. Hadoop 2.2.0. Accumulo 1.5.2. Zookeeper 3.4.6. D4M, pMatlab, and Matlab 2010bSP2. CentOS 6.5 64-bit. We generate graph edges using the same 2×2 Kronecker algorithm as the Graph500 benchmark.

We test on the following various sizes of symmetric matrix: 4096×4096,8192×8192, $16,384 \times 16,384$, $65,536 \times 65,536$, $262,144 \times 262,144$, $524,288 \times 524,288$, $1,048,576 \times 1,048,576$. Five sets of experiments have been conducted in the same cluster: HEIGEN, proposed architecture with and without scheduler using 15 and 57 processes respectively. For last 4 experiments, the first process is the main one which spawns other working processes. Moreover, 14 out of 16 machines are working nodes while one is running HDFS namenode and the other running the main process. Such tests, therefore, equal to 1 process per machine and 4 processes per machine.

In Fig. 4, step one takes around 300 seconds for matrix with size of 1 million. The time for step two is proportional to the number of machines used in the cluster. For example, with 8 machines, it takes around 1100s for the same matrix, whereas it takes around 600s with 16 machines, which demonstrates scalable performance. We focus on step three, iterative calculation of eigenvalues and eigenvectors, which is analyzed below.

Fig. 5 shows the average running time of Lanczos-SO algorithm for one iteration on HEIGEN and proposed architecture. Our model has obtained almost 2X speed up as HEIGEN when

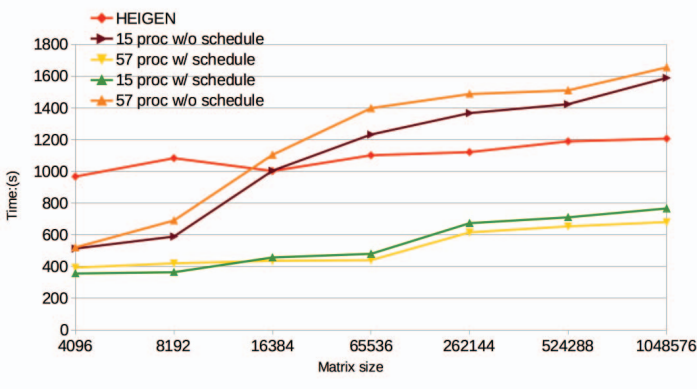

Fig. 5: Average running time of Lanczos-SO algorithm for one iteration on HEIGEN and proposed architecture. X-axis is the size of the matrix; Y-axis is the running time in seconds

the matrix size approaches one million. Fig. 6 gives more detail for the running time of each operation and Table IV shows the definition of each operation in Lanczos-SO algorithm.

MV:	matrix-vector multiplication	Line 3
Alpha:	dot product of two vectors to compute α	Line 4
OrtV:	orthogonalize against two previous basis vectors	Line 5
Beta:	normalization of a vector to compute β	Both line 6 and 16
UpdateV:	Update vector v_{i+1}	Line 21

TABLE IV: Operation definition in Lanczos-SO algorithm

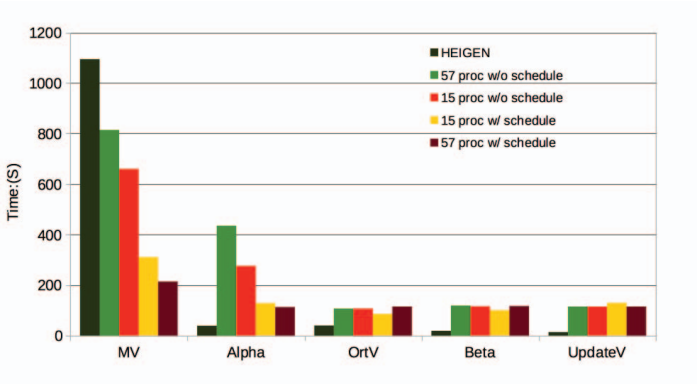

Fig. 6: Average running time for different Lanczos-SO operations on HEIGEN and proposed architecture for matrix with size of one million

Matrix-vector multiplication or MV dominates the calculation, proving Lanczos-SO algorithm is a data intensive computation rather than compute-intensive, disk I/O dominates most of the time. In Fig. 8, it is clearly shown that reading input matrix from disk dominates the MV for million-scale matrix, 210 seconds accounting for 70% of the whole time. In HEIGEN, almost 550 seconds is used to generate key/value pairs for matrix and vector; and another 550 seconds is used to shuffle the pairs from previous vector and complete the multiplication. Our computation model is almost 3X faster for MV, proving our distributed computation architecture is more efficient for data shuffling. As for other operations, HEIGEN is faster than our approach because all of these operations

only involve vector, scalar multiplication which requires small data exchange. MapReduce framework also provides good data locality for distribute computation. In addition, process synchronization and Matlab startup cause around 100s overhead while HEIGEN takes only a few seconds.

The performance of HEIGEN appears not to be affected by the size of matrix. The speculation is as follows:

- The matrix is too sparse.
- Data replication in HDFS helps HEIGEN distribute the sparse matrix equally across the data nodes.
- MapReduce framework provides good job scheduling.

RV:	Reading Vector
RM:	Reading Matrix
PP:	Post Processing: convert string into matrix
MUL:	Matrix Vector Multiplication
WB:	Writing result Back

TABLE V: Operation definition for MV implementation

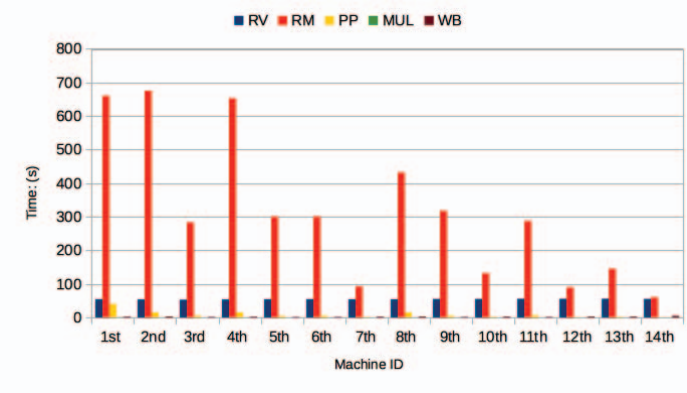

Fig. 7: Average running time of operations in MV by distributing columns equally into 14 machines for matrix with size of one million.

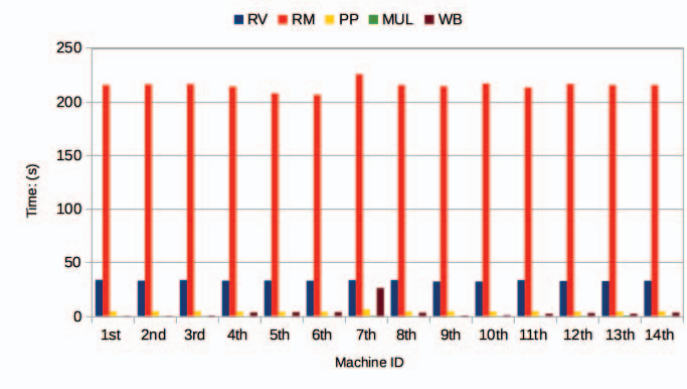

Fig. 8: Average running time of operations in MV by distributing work loads equally to working machines for matrix with size of one million.

Fig. 9 shows the statistical information about matrix with size of one million: x-axis shows the column number and y-axis the total number of non-zero entries in each column. Fig. 7 and Fig. 8 show the performance of MV for one iteration without and with scheduling for this matrix, respectively. Operation definition for MV implementation can be found in Table V. RM is the most costly operation and the shape in Fig. 7 is similar to the distribution in Fig. 9 because every machine is assigned the same length of rows from the input matrix. Due to the sparseness of the input matrix, performance of parallel processing is highly affected by the work load distribution. Fig. 8 shows 3X faster for RM because we have scheduled almost the same work load across the working processes.

VI. Related work

Most parallel large-scale eigensolvers are based on MPI with message passing. Examples are as follows: the work by Zhao et al. [30], PLAPACK [31], HPEC [32], PLANO [33], PREPACK [34], SCALABLTE [35]. However, they do not scale well to very large matrix, billion-scale for example [14]. One reason is the algorithm deployed contains matrix-matrix multiplication. For example, the parallel block lanczos algorithm deployed in HPEC requires matrix-matrix multiplication. The other reason lies in the communication when MPI is used with peer-to-peer networks since aggregation is costly and the network performance will be low. In our architecture, the communication is left to the database since all slave nodes write the output there and all nodes have access to the database.

Apache Spark provides an in-memory computation engine for large-scale data processing, and MLib (http://spark.apache.org/mllib/) is the machine learning library for Spark. Even though Spark outperforms Hadoop on iterative algorithms like logistic regression, MLib has its limitation in SVD for large matrix because the algorithm requires large matrix-matrix multiplication. Moreover, the performance of Spark degrades when RDDs size exceed the memory. Unlike our architecture, Spark offers no indexing capability to speed up data shuffling for iterative algorithms. We plan to investigate the performance of Spark to compare with our architecture. Apache Flink [36] is another example of platform for efficient, distributed, general-purpose data processing at its beginning stage, but currently there is no support for SVD. SystemML [37] expresses machine learning algorithms in a higher-level language and then compile and execute them in a MapReduce environment. The goal of SystemML is to make machine learning algorithms more adaptable and scalable on MapReduce environment. And our architecture brings together Hadoop and HPC. Cumulo [38] is designed for matrix-based big data analysis in the cloud. To support scalable linear algebra operation, Cumulo first preprocesses matrix as tiles (sub-matrices), and then runs MapReduce for computation. Our architecture offers an alternative novel way to MapReduce to conduct distribute linear algebra operations by indexing and querying matrix elements.

VII. Discussion

The reason that our computation model outperforms MapReduce lies in the fact that the former utilizes D4M to retrieve input data from NoSQL distribute database more efficiently than the latter which relies on mapper to generate key/value pairs from input matrix. In addition, the iterative nature of algorithm fits more the former.

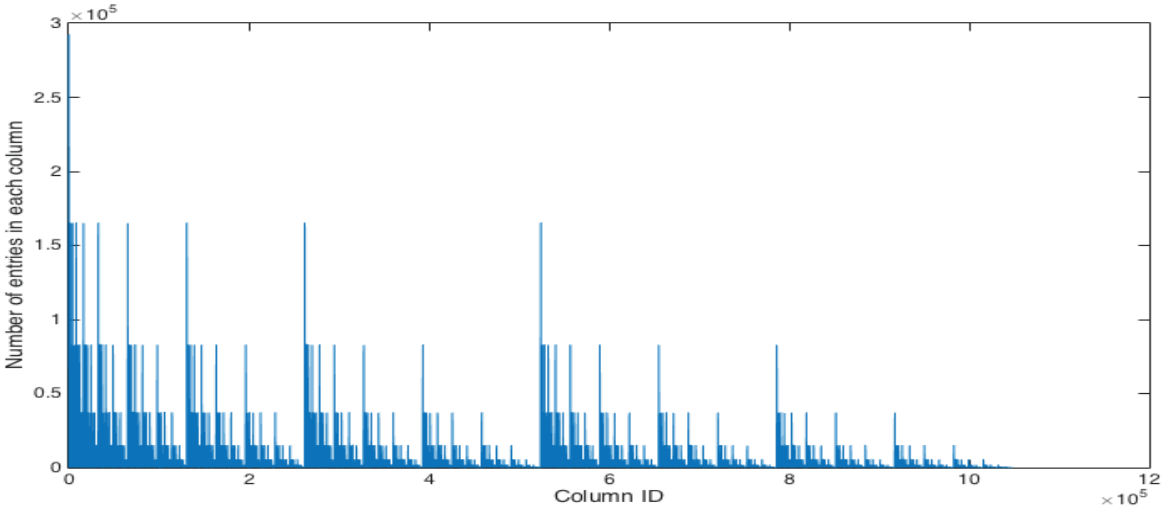

Fig. 9: Statistical information obtained from Accumulo table for non-zero entries distribution for matrix with size 1048576*1048576

Disadvantage of pMatlab is the lack of a computation framework which schedules the computation based on data locality like MapReduce. Such scheduler, however, can be implemented using Accumulo and D4M. The former provides global distribution information about sparse matrix whereas the latter offers easy-and-fast mechanism to retrieve data. In addition, pMatlab is less robust than MapReduce. Had one working process failed, the whole computation would have stopped.

Moreover, presented architecture has the potential to be extended to similar machine learning algorithms which are iterative. Consider, deep networks, for example, or autoencoder more specifically, a typical feedforward neural network which aims to learn a compressed, distributed representation (encoding) of a dataset. The idea of distributed representation is similar to find the top k eigenvalues and eigenvectors in Lanczos-SO algorithm. One obvious difference is autoencoder adapts a non-linear iterative approach to calculate a weight matrix which is the ultimate learning object. Therefore, extending our architecture to deep networks is one of our future works.

In future work, we will also consider implementing in-memory computation similar to Spark [26] which deploys Resilient Distributed Datasets [25] (RDDs) for in-memory computations on large clusters. One problem arises when the size of RDD exceeds the memory capacity. Unlike Accumulo which provides statistical information regarding input matrix, the size of RDDs is hard to estimate in advance because RDDs are constructed by transforming files in HDFS using operators such as *map*, *filter* etc and details of how input data are split among data nodes in HDFS is hidden from programmer. By storing input matrix in memory like RDDs, the performance of our architecture is expected to be improved as shown in our experiment, reading input matrix from disk is dominating the whole computation.

VIII. Summary

In this paper, we have created a database-based distributed computation architecture which brings together Hadoop and HPC supported by Accumulo and D4M. We deploy NoSQL distributed database Accumulo to manage large unstructured data. Meanwhile, pMatlab, a parallel MPI version for Matlab, is used as the parallel computation engine. D4M serves as the data structure to facilitate the distributed computation. To evaluate, we investigate and analyze MapReduce implementation of iterative graph algorithms with a focus on Lanczos-SO algorithm. Experimental results with Graph500 benchmark datasets demonstrate 2X speedup as compared to HEIGEN. In the future we will first consider implementing deep networks and second focus on integrating with Spark for better performance.

Acknowledgment

The authors would like to thank IBM/CAS Toronto for supporting Yin Huang with a CAS fellowship. We would also like to thank NIST/SSD Information Systems Group for providing support to conduct this Big Data Analytics computation. We are also grateful to CHMPR for providing the IBM iDataPlex bluewave computational resources to conduct these data intensive experiments. In particular, we wish to acknowledge Dr. John Dorband for training one of the authors as a system administrator to establish the Hadoop based ecosystem. And we would also like to thank Prof Xian-He Sun from Department of Computer Science at the Illinois Institute of Technology for providing cluster resource.

References

[1] Apache Hadoop https://hadoop.apache.org/

[2] Apache Hive https://hive.apache.org/

[3] Apache Accumulo https://accumulo.apache.org/

[4] Apache HBase http://hbase.apache.org/

[5] M. Newman. Fast algorithm for detecting community structure in networks. Physical Review E, 69, 066133 (2004)

[6] A. Ng, M. Jordan, and Y. Weiss. On spectral clustering: analysis and an algorithm. In Advances in Neural Information Processing Systems 14 2002, pp. 849-856.

[7] M. Newman and M. Girvan. Finding and evaluating community structure in networks. Physical Review E, 69, 026113 (2004).

[8] Yin Huang, Han Dong, Yelena Yesha, Shujia Zhou. A Scalable System for Community Discovery in Twitter During Hurricane Sandy. 14th IEEE/ACM International Symposium Cluster, Cloud and Grid Computing (CCGrid), May 2014

[9] J. Kepner, Massive database analysis on the cloud with D4M, in Proc. HPEC Workshop, 2011.

[10] B.A. Miller, N. Arcolano, M.S. Bear d, N.T. Bliss, J. Kepner, M.C. Schmidt, and P.J. Wolfe, A Scalable Signal Processing Architecture for Massive Graph Analysis, 37th IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Kyoto, Japan, Mar 2012

[11] Apache Giraph http://giraph.apache.org/

[12] Apache Hama https://hama.apache.org/

[13] L.N. Trefethen and D. Bau III, Numberical Linear Algebra, SIAM, 1997.

[14] U Kang, Breandan Meeder, Evangelos E. Papalexakis, and Christos Faloutsos, HEigen: Spectral Analysis for Billion-Scale graphs, IEEE Transactions on knowledge and data engineering, VOL. 26, No.2, Feb 2014.

[15] A. Dave, W. Lu, J. Jackson, and R. Barga, "CloudClustering: Toward an iterative data processing pattern on the cloud," in Proc. the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, 2011, pp. 1132-1137.

[16] Travinin Bliss, N., Kepner, J.: pMatlab parallel Matlab library. Int. J. High Perform. Comput. Appl. 21(3), 336359 (2007).

[17] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz, A. McCabe, P. Michaleas, A. Prout, A. Reuther, A. Rosa, and C. Yee, Dynamic Distributed Dimensional Data Model (D4M) Database and Computation System, Proceedings of the 2012 IEEE International Conference on Acoustics, Speech and Signal Processing, 2012, pp. 53495352

[18] J.Kepner, Parallel Matlab for Multicore and Multinode computers, SIAM Press, Philadelphia, 2009

[19] N. Bliss and J. Kepner, pMatlab Parallel Matlab Library, International Journal of High Performance Computing Applications: Special Issue on High Level Programming Languages and Modesl, J.Kepner and H. Zima (editors), Winter 2006 (November)

[20] J. Kepner and S. Ahalt, MatlabMPI, Journal of Parallel and Distributed Computing, vol. 64, issue 8, August, 2004

[21] N. Bliss, R. Bond, H. Kim, A. Reuther, and J. Kepner, Interactive Grid Computing at Lincoln Laboratory, Lincoln Laboratory Journal, vol. 16, no. 1, 2006.

[22] Nathan P. Halko, Randomized methods for computing low-rank approximations of matrices, Ph. D., Department of Applied Mathematics, University of Colorado. 2012.

[23] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. Lopez, G. Gibson, A. Fuchs, and B. Rinaldi, YCSB++: benchmarking and performance debugging advanced features in scalable table stores, in Proceedings of the 2nd ACM Symposium on Cloud Computing. ACM, 2011, p. 9.

[24] B. N. Parlett and D. S. Scott, The Lanczos algorithm with selective orthogonalization, Mathematics of Computation, 33:217-238, 1979.

[25] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI, 2012.

[26] Apache Spark https://spark.apache.org/

[27] Glenn Lockwood, "Hadoop's Uncomfortable Fit in HPC" http://glennklockwood.blogspot.com/2014/05/hadoops-uncomfortable-fit-i -hpc.html

[28] J. Dean, G.S. Corrado, R. Monga, K. Chen, M. Devin, Q.V. Le, M.Z. Mao, M.A. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks." In NIPS, 2012.

[29] J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, P. Michaleas, J. Mullen, A. Prout et al., "Achieving 100,000,000 database inserts per second using Accumulo and D4M, IEEE High Performance Extreme Computing, 2014.

[30] Y. Zhao, X. Chi, and Q. Cheng. An implementation of parallel eigenvalue computation using dual-level hybrid parallelism. Lecture Notes in Computer Science, 2007.

[31] P. Alpatov, G. Baker, C. Edward, J. Gunnels, G. Morrow, J. Overfelt, R. van de Gejin, and Y.-J. Wu. Plapack: Parallel linear algebra package design overview. SC97, 1997.

[32] M. R. Guarracino, F. Perla, and P. Zanetti. A parallel block lanczos algorithm and its implementation for the evaluation of some eigenvalues of large sparse symmetric matrices on multicomputers. Int. J. Appl. Math. Comput. Sci., 2006.

[33] K. Wu and H. Simon. A parallel lanczos method for symmetric generalized eigenvalue problems. Computing and Visualization in Science, 1999.

[34] J. L. R.B., S. D.C., and Y. C. Arpack users guide: Solution of large-scale eigenvalue problems with implicitly restarted arnoldi methods. SIAM, 1998.

[35] L. Blackford, J. Choi, A. Cleary, E. DAzevedo, J. Demmel, and I. Dhillon. Scalapack userss guide. SIAM, 1997.

[36] Apache Flink, https://flink.apache.org/

[37] A. Ghoting, R. Krishnamurthy, E. Pednault, et al. SystemML: Declarative machine learning on MapReduce. In ICDE, pages 231242, 2011.

[38] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In SIGMOD, 2013.