Large-Scale Matrix Factorization using MapReduce

Zhengguo Sun, Tao Li, and Naphtali Rishe
School of Computer Science
Florida International University
Miami, FL 33199
{zsun002, taoli, rishen}@cs.fiu.edu

Abstract—**Due to the popularity of nonnegative matrix factorization and the increasing availability of massive data sets, researchers are facing the problem of factorizing large-scale matrices of dimensions in the orders of millions. Recent research [11] has shown that it is feasible to factorize a million-by-million matrix with billions of nonzero elements on a MapReduce cluster. In this work, we present three different matrix multiplication implementations and scale up three types of nonnegative matrix factorizations on MapReduce. Experiments on both synthetic and real-world datasets show the excellent scalability of our proposed algorithms.**

Keywords: nonnegative matrix factorization; MapReduce

I. Introduction

Nonnegative matrix factorization (NMF) factorizes an input nonnegative matrix into nonnegative matrices of lower rank. Originally proposed for parts-of-whole interpretation of matrix factors, NMF has attracted a lot of attentions recently in data mining and machine learning communities. It is recently discovered that NMF can be used to solve challenging data mining and machine learning problems. For example, NMF with the sum of squared error cost function is equivalent to a relaxed K-means clustering, the most widely unsupervised learning algorithm [21][22]. In addition, NMF with the I-divergence cost function is equivalent to probabilistic latent semantic indexing, another unsupervised learning method popularly used in text analysis [23][24]. Consequently, NMF has been shown to be useful in a variety of applied settings, including environmetrics, chemometrics, pattern recognition, multimedia data analysis, text mining, web mining, and DNA gene expression analysis.

Due to the increasing availability of massive data sets, researchers are facing the problem of factorizing matrices of dimensions in the orders of millions. Recent research [11] has shown that it is possible to factorize such gigantic matrices within tens of hours using the MapReduce distributed programming platform. In this paper, we propose three different matrix multiplication implementations using MapReduce, based on which we successfully parallelize three different types of NMF algorithms on the MapReduce platform. We evaluate our algorithms on a cluster provided by Google and IBM through the NSF Cluster Exploratory (CLuE) program [17] using both synthetic and real data sets. Results have shown that the performance of our proposed algorithm is at least as good as previous efforts.

Different from the work by Liu et al. [11] whose implementations are directly built on the updating rules, we reduce the problem of NMF to a series of different matrix multiplication implementations on the MapReduce platform, which makes our algorithms applicable to most matrix factorization algorithms that are using the multiplicative updating scheme.

In summary, the key contributions of our work are summarized below:

- We propose three different matrix multiplication implementations on MapReduce.
- We scale up three different types of NMF algorithms on MapReduce based on our proposed matrix multiplication implementations.
- We implement and experimentally evaluate our algorithms using both synthetic and real data sets on a real-world cluster.

The rest of this paper is organized as follows. Section II defines three different types of nonnegative matrix factorization and introduces their updating algorithms respectively. Section III describes three schemes to perform matrix multiplication on MapReduce based on the properties of the matrices. These schemes are then used to implement the updating algorithms in Section II. We report the experimental evaluation in Section IV and briefly discuss related works in Section V. Finally, Section 0 concludes our work.

II. Nonnegative Matrix Factorization

Let $X \in \mathbb{R}^{+^{m\times n}}$ be the input data matrix, we define three different matrix factorizations and introduce their updating rules in the following.

A. NMF

The classical NMF [12] could be written as

$$X \approx FG^T,$$

where $F \in \mathbb{R}^{+^{m\times k}}, G \in \mathbb{R}^{+^{n\times k}}$. The updating rules for NMF are listed as follows:

$$F_{ik} \leftarrow F_{ik}\frac{(XG)_{ik}}{(FG^TG)_{ik}}, \quad (1)$$

$$G_{jk} \leftarrow G_{jk}\frac{(X^TF)_{jk}}{(GF^TF)_{jk}}. \quad (2)$$

B. Convex-NMF

In general, the basis vectors F has the meaning of cluster centroids. To enforce this geometry meaning, F can be restricted to be a convex combination of the input data points [25]. In other words, in addition to have F to be nonnegative, we restrict each column of F to lie within the space spanned by the columns of X, i.e.,

$$f_l = w_{1l}x_1 + \cdots + w_{nl}x_n = Xw_l, \text{ or } F = XW,$$

where f_l and x_i is a column vector of F and X, w_{il} is an element of $W, and\ w_{il} \geq 0$. We call this restricted form of factorization as Convex-NMF [25]. The updating rules of this factorization are

$$G_{ik} \leftarrow G_{ik}\sqrt{\frac{(X^TXW)_{ik}}{(GW^TX^TXW)_{ik}}}, \qquad (3)$$

$$W_{ik} \leftarrow W_{ik}\sqrt{\frac{(X^TXG)_{ik}}{(X^TXWG^TG)_{ik}}}. \qquad (4)$$

C. Tri-Factorization

Tri-factorization [14] is useful to simultaneously cluster the rows and the columns of the input data matrix X. We consider the following nonnegative 3-factor decomposition

$$X \approx FSG^T,$$

where $F \in \mathbb{R}^{+^{m\times k}}, S \in \mathbb{R}^{+^{k\times k}}, G \in \mathbb{R}^{+^{n\times k}}$. Note that S provides additional degrees of freedom such that the low-rank matrix representation remains accurate while F gives row clusters and G gives column clusters. The updating rules of tri-factorization are

$$G_{jk} \leftarrow G_{jk}\sqrt{\frac{(X^TFS)_{jk}}{(GG^TX^TFS)_{jk}}}, \qquad (5)$$

$$F_{ik} \leftarrow F_{ik}\sqrt{\frac{(XGS^T)_{ik}}{(FF^TXGS^T)_{ik}}}, \qquad (6)$$

$$S_{ik} \leftarrow S_{ik}\sqrt{\frac{(F^TXG)_{ik}}{(F^TFSG^TG)_{ik}}}. \qquad (7)$$

III. NMF USING MAPREDUCE

A. MapReduce Introduction

MapReduce is a programming model proposed by Google for processing and generating large data sets [6]. Inspired by two functional programming primitives, each MapReduce computation unit consists of two stages, namely, map stage and reduce stage. Input of key/value pairs are consumed by a map function defined by the user in the map stage. A set of intermediate key/value pairs are generated in the map stage and grouped by keys as the input to the reduce stage. User-defined reduce function merges all intermediate values associated with the same intermediate key and output the final result.

Despite its simplicity, the applicability of this model is not limited to processing and generating large data sets. Large-scale data clustering [8], graph computations [9] and many other problems have been shown to be suitable for MapReduce. In the following, we first discuss three different matrix multiplication implementations on MapReduce. Based on that, we next discuss how to scale up NMF on MapReduce. We give details on the steps to compute the classic NMF. We extend the algorithm to ConvexNMF and Tri-Factorization, by pointing out the difference between these two factorizations and NMF as their updating rules are similar. For the same reason, we only discuss how to update G in NMF.

B. Matrix Multiplication on MapReduce

The major operation in the updating rules of all three types of factorizations is matrix multiplication. Thus, efficient implementation of matrix multiplication on MapReduce is the key to scale up matrix factorization.

Assume $A \in \mathbb{R}^{+^{m\times n}}$ and $B \in \mathbb{R}^{+^{n\times k}}$. Traditionally, AB is defined as the inner product of the row vectors of A and column vectors of B. Thus,

$$AB = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix}\begin{pmatrix} b_1 & b_2 & \cdots & b_k \end{pmatrix} = \begin{pmatrix} a_1b_1 & \cdots & a_1b_k \\ \vdots & \ddots & \vdots \\ a_mb_1 & \cdots & a_mb_k \end{pmatrix}.$$

This kind of decomposition is useful if we could share B across all rows of A. Each row of AB could be computed in parallel. A MapReduce job MM-1 is formulized in TABLE I. to carry out this computation.

TABLE I. THREE MATRIX MULTIPLICATION SCHEMES

Scheme		**Input/Output**	
		input	***output***
MM-1	Map	$\langle i, a_{i*}\rangle$	$\langle i, a_{i*}B\rangle$
	Reduce	None	
MM-2	Map	$\langle i, a_{*i}\rangle$ or $\langle i, b_{i*}\rangle$	$\langle i, a_{*i}\rangle$ or $\langle i, b_{i*}\rangle$
	Reduce	$\langle i, [a_i, b_i]\rangle$	$[\langle j, c_{j*}\rangle]$ or $[\langle j, c_{*j}\rangle]$
	Map	$\langle j, c_j\rangle$	$\langle j, c_j\rangle$
	Reduce	$\langle j, [c_{j_1}c_{j_2}\cdots]\rangle$	$\langle j, c'_j\rangle$
MM-3	Map	$\langle i, a_{i*}\rangle$	$[\langle j, c_j\rangle]$
	Reduce	$\langle j, [c_{j_1}c_{j_2}\cdots]\rangle$	$\langle j, c'_j\rangle$

We use a_{i*} to denote a row vector with id i and a_{*i} to denote a column vector. We use a_i to denote either a row vector or column vector if it doesn't matter which one it is. a'_i is used to denote the final result aggregated from all the a_i `s. We use an angle bracket to denote a key/value pair and a square bracket to denote a list of key/value pairs.

As can be seen from TABLE I. no reduce phase is needed for MM-1. The input to the map phase is the row vectors of A. When the size of B gets large enough, this scheme will fail since the cost of sharing a large matrix across many nodes in a cluster is too expensive.

Alternatively, we could decompose A into column vectors and B into row vectors. Thus,

$$AB = (a_1 \quad a_2 \quad \cdots \quad a_n)\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \sum a_i \otimes b_i.$$

The implication of this decomposition is that if we could partition A and B in such a way that the corresponding columns of A and rows of B are on the same machine, we could compute the outer product in parallel and merge them together to obtain AB. This scheme fits the MapReduce model perfectly. No data sharing is required as in the previous scheme. We could use two MapReduce jobs to implement this scheme MM-2, which is shown in TABLE I.

Now let's examine these two jobs of MM-2 in detail. The first job does multiplication and the second job does summation. The map phase of the multiplication job accepts a column vector a_{*i} or a row vector b_{i*}, and emits an intermediate pair with column id or row id as key and the column or row itself as value. Column and row vectors with the same id are shuffled to the same machine as the input for the reduce phase. In the reduce phase, we compute the outer product of a_{*i} and b_{i*} and, depending on the context in which this multiplication is used, we either output a bunch of row vectors or column vectors, which is denoted in TABLE I. as $[\langle j, c_{j*}\rangle]$ or $[\langle j, c_{*j}\rangle]$. These row or column vectors are partial results of the vector. We merge them into the final vector in the summation job. The map phase of the summation job is an identity function, which just outputs the input key/value pairs for shuffling. The reduce phase of this job merges all the row vectors with the same id or column vectors with the same id and emits a final row vector or column vector.

This scheme can be used to compute the product of any matrices, no matter it is dense or sparse, as long as the left matrix is partitioned into column vectors and the right matrix is partitioned into row vectors. Under certain circumstance, the two MapReduce jobs can be merged into one. For example, consider the matrix multiplication A^TA. Assume A is partitioned in row vector, then A^T in column vector is the same as A in row vector. Thus, we could compute A^TA using only one MapReduce job as shown in MM-3 of TABLE I. where multiplication is conducted in map phase and summation is performed in reduce phase. This shortcut saves one MapReduce job and turns out to be very useful in matrix factorization, where multiplications like A^TA appear frequently.

C. NMF

We partition X, F and G into row vectors, which renders the following view:

$$X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} F = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{pmatrix} \; and \; G = \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{pmatrix},$$

where x_i is a n-dimensional row vector and f_i and g_i are k dimensional row vectors. All of them are stored as sets of $\langle i, x_i\rangle$, $\langle i, f_i\rangle$ and $\langle i, g_i\rangle$ key/value pairs. There are three steps to update G. The first step is to compute the numerator X^TF in (2). The second step is to compute the denominator GF^TF in (2). And the last step is to update G. Fig. 1 shows the flow chart of updating G. We discuss them in the following one by one.

1) Computing $A = X^TF$

Since we partition X and F in row vectors, X^TF is ready for multiplication. Using the scheme MM-2 in TABLE I. , we could compute X^TF with two MapReduce jobs.

2) Computing $B = GF^TF$

There are two ways to compute GF^TF. The first one is to compute GF^T first and GF^TF afterwards. This requires 4 MapReduce jobs, 2 for each multiplication. The second one is much faster. By using MM-3 to compute F^TF first, and using MM-1 to compute GF^TF, only two jobs are needed. Noting that the result of F^TF is a $k \times k$ matrix which is reasonably small to be fit into memory for computing GF^TF with G.

3) Updating G

Once both X^TF and GF^TF have been computed; only one job is needed to update F, which is summarized as follow:

- Map: Map $\langle i, g_i\rangle$, $\langle i, a_i\rangle$ and $\langle i, b_i\rangle$ as they are
- Reduce: Take $\langle i, [g_i, a_i, b_i]\rangle$ and emit$\langle i, g_i'\rangle$, where $g_i' = g_{i,j}\frac{a_{i,j}}{b_{i,j}}$.

This finishes the update of G. F is updated in the same manner, except that when computing XG, we need X to be partitioned into column vectors, whereas it is stored as row vectors. Thus, an additional MapReduce job is needed to transform X from row vectors to column vectors. This won't affect the overall performance, since only one such job is needed for the entire updating cycle, which usually consists of a few iterations.

D. Convex-NMF and Tri-Factorization

Convex-NMF and Tri-Factorization are essentially computed in the same manner as NMF, namely, to update a matrix, first compute the numerator of the updating rule, and then compute the denominator, and finally update the original matrix. In the following, we focus on their difference with NMF on the updating rules, rather than the details.

1) Convex-NMF

We partition X, G and W into row vectors as we did in NMF. One notable pattern of the updating rules of Convex-NMF is that X^TXW has appeared three times, two in (3) and one in (4). Thus, it is beneficial to first compute X^TXW and use the result later.

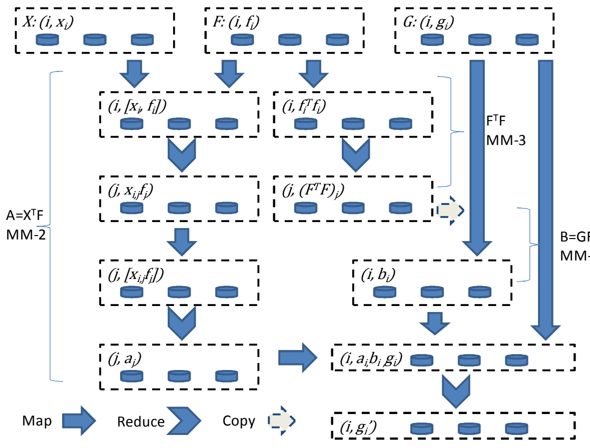

Figure 1. Computing $G_{jk} \leftarrow G_{jk} \frac{(X^TF)_{jk}}{(GF^TF)_{jk}}$.

Again, we could use the MM-3 to compute X^TX first and X^TXW afterwards. However, this time we decide not to do that, because the result of X^TX would be a huge dense matrix that could introduce a lot of data shuffling in later computation. We choose to compute XW first and X^TXW afterwards using MM-2 in TABLE I. .

After computing XW, we use two additional MM-2s to compute the denominator in (3). The numerator in (4) is computed in the same manner as X^TXW. When computing the denominator in (4), we first use MM-3 to compute G^TG, and then take the result of X^TXW and do a MM-1 job to finish the computation.

2) *Tri-Factorization*

Again, we partition X, G, F and S into row vector. Repeated patterns like X^TFS in (5) and XGS^T in (6) are utilized to reduce the computation. To update G, we first compute the numerator X^TFS in (5) using MM-2 ($A = X^TF$) and MM-1 ($B = AS$). The result is then used to compute the denominator in (5) using MM-2 ($C = G^TB$) and MM-1 (GC). F is updated similarly.

The updating rule of S has its own structure and S is updated slightly differently. First of all, the numerator of S is computed using two MM-2s ($A = F^TX$ and AG). We use MM-3 to compute both $B = F^TF$ and $C = G^TG$ in (7). And then two MM-1 operations are used to compute $D = BS$ and DC. Finally, an update job is used to update the new value of S.

IV. EXPERIMENTS

We use the open source implementation of MapReduce – Hadoop [15] in our experiments. All three factorization algorithms have been implemented in Java for Hadoop.

A. *Experiments Setup*

All the experiments were conducted on a large shared cluster of approximately 500 machines provided by Google and IBM [18]. Each machine has two single-core 2.8GHz Xeon processors, 8 GB memory, two 400 GB hard drives, and one gigabit Ethernet connection. Hadoop 0.20.1 runs on one 64-bit Xen virtual machine running on top of the above physical hardware with access to all the memory (minus some overhead), all the execution threads of the processors and all of the disks. The VM runs CentOS 5.3 and the host operating system runs Fedora Core 8.

Both synthetic and real data sets were used in the experiments. Synthetic data sets were generated by a random matrix generator, which generates a matrix $X \in \mathbb{R}^{+^{m \times n}}$ with sparsity δ on given parameters m, n and δ. We varied m, n and δ to see how the performance varies. While the number of participating machines in the map phase cannot be set directly, we varied the number of machines R in the reduce phase to see how more participating machines could improve the performance. Unless explicitly pointed out, all the reported time is for one iteration of the algorithms.

The blog data was collected by an NEC in-house blog crawler. Given seeds of manually picked highly ranked blogs, the crawler discovered blogs that are densely connected with the seeds, resulting in an expanded set of blogs that communicate with each other [19][26]. The data set is represented as a sparse matrix that is of dimension 274,679 by 5,304. Each row is a blog entry and the columns contain word frequencies in the blogs. TABLE II. summarizes the characteristics of the data sets used in the experiments.

In the following sections, we first examine the computation cost of each component in (2). Then, we examine the scalability of the algorithms with the size of the input matrices. We also present how the performance varies w.r.t δ, k and R in section IV.C. Finally, we evaluate the algorithms on the real data set.

B. *Computation Cost*

The computation cost of each MapReduce job in updating G is shown in TABLE III. We recorded the data shuffled between Map phase and Reduce phase in MB and the total elapsed time in seconds for each job. We choose $m = 2^{22}, n = 2^{12}, k = 8, \delta = 2^{-7}$ for the matrix being factorized in this experiment, which is typical for a large data set.

As can be seen from TABLE III. , the major cost in updating G is the computation of X^TF and F^TF , which accounts for 87 percent of the total elapsed time. This is understandable because the inner dimension of both of the two multiplications is in the order of million. Another notable pattern is the imbalance in the size of data shuffling. The multiplication job of X^TF shuffled more than 2 GB of data, while F^TF only shuffled 7 KB. This is because we used MM-2 to compute X^TF, whose Map phase only read the input for grouping. Thus, both X^T and F were shuffled. On the other hand, we used MM-3 to compute F^TF, where the multiplication was performed in Map phase and only the small resulting $k \times k$ matrix was shuffled.

C. *Scalability and Performance w.r.t δ, k and R*

We also report the elapsed time of one iteration for all three algorithms in 0 In these experiments, we fixed $k = 8, \delta = 2^{-7}, R = 8$ and $n = 2^{11}$ and varied m from 2^{12} to 2^{24}.

TABLE II. DATA SETS DESCRIPTION

Data Sets	m	n	δ	Size(MB)
Synthetic data sets	2^{12}~2^{24}	2^{11}~2^{12}	2^{-10}~2^{-5}	1~6605
Real data set	274,649	5,304	0.008	47

As can be seen from Fig. 2, the time doesn't change much as m increases from 2^{12} to 2^{18}. This is because the computation power hasn't been saturated on this scale. The main cost is the overhead of starting MapReduce jobs. Starting from 2^{18}, we observe the sub linear scalability for all of the algorithms. This is due to the appropriate application of different multiplication implementations for different components in the updating rules, which minimizes the communication cost and maximizes data locality and parallelism.

Fig. 3 shows how the performance varies w.r.t the sparsity δ. As δ goes from 2^{-10} to 2^{-5}, the number of nonzero elements in the matrix increases from 16 million to 600 million, and the elapsed time also increases in proportion to that.

Fig. 4 reveals the linearity between elapsed time and the dimensionality of k. As k doubles from 8 to 128, the elapsed time gradually increases from 10 minutes to 40 minutes. The slope is smaller than 1, which is good for large data sets.

Finally, we plot the speedup achieved by doubling the number of machines in the reduce phases of MM-2 and MM-3 in Fig. 5. Two series are plotted. One shows the speedup for one single iteration of the NMF algorithm; the other shows the speedup achieved by the affected MapReduce jobs only, that is, MM-2 and MM-3. There are jobs that don't benefit from adding more machines, such as MM-1 and the updating of the original matrix. Thus, the speedup for one single iteration is smaller than that of MM-2 and MM-3.

D. Experiments with real data set

Finally, we ran our algorithm on the real world blog data set. The result is reported in Fig. 6. We divided this data set into five partitions and ran our algorithms on 20%, 40%, 60%, 80%, and the whole data set. Again, we observed linear scalability with regarding to this data set. Due to the relatively small size of this data set compared to the synthetic data sets, most of the elapsed time is spent in starting and cleaning up MapReduce jobs, which results a near-flat line in Fig. 6.

V. RELATED WORK

Two particular lines of research are related to our work. One is existing works in parallel matrix multiplication and NMF. The other is using MapReduce in large-scale data mining and machine learning.

A. Parallel Matrix Multiplication and NMF

Because of the importance of matrix multiplication as a basic operation in linear algebra, parallel matrix multiplication has been studied extensively. One of the most popular algorithm might be Cannon's algorithm [1]. Although most of the algorithms assume special data layout and are tied to a particular parallel architecture, some basic ideas still could be applied to MapReduce. Algorithms designed for single instruction multiple data (SIMD) systems are of particular interests, for the resembalance between MapReduce and SIMD. For example, the MM-2 scheme we proposed in Section III.B could also be implemented directly on an SIMD system by replacing a machine in the cluster with a processor in the system. From a relational algebra perspective, what MM-2 does is a join on the column id of the left matrix and the row id of the right matrix. More detailed discussions could be found in [2] [3].

Because of the popularity of NMF, there are many works in trying to parallelize it [4][5]. Since data sharing and communication are no longer light-weight in distributed clusters like MapReduce, those methods cannot be ported directly to MapReduce.

B. Large-scale data mining using MapReduce

Although the initial purpose of MapReduce is to perform large-scale data processing [6], it turns out that this model is much more expressive than that [6]. Chu et al. investigated the possibility to implement machine learning algorithms using MapReduce on multicore [7]. Their conclusion is that a variety of learning algorithms that fit the statistic query model [30] could be parallelized using MapReduce. An open source project Mahout [13] has been started to port those algorithms to Hadoop. Papadimitriou and Sun have done a case study in data mining on co-clustering [8] towards petabyte scale of data using MapReduce. MapReduce has also been used in many other tasks of data mining and machine learning. Those works include but not limited to Kang et al. on graph mining [9], Panda et al. on tree ensembles [20], Liu et al. on bayesian browsing model [10], and Chen et al. on behavioral targeting [16].

In particular, Liu et al. successfully scaled up the classical NMF [12] for web-scale dyadic data analysis on MapReduce [11]. They assumed the matrix to be factorized is stored as $(i, j, A_{i,j})$ tuples that are spread across machines and proposed different partitions for the factors. Although the techniques they used are different, they could be reduced to one of the matrix multiplication schemes we proposed in Section III.B.

TABLE III. COMPUTATION COST OF UPDATING G IN NMF

Component		$m = 2^{22}, n = 2^{12}, k = 8, \delta = 2^{-7}$ and $R = 8$	
		Shuffle(MB)	Time(sec)
$X^T F$	Multiplication	2099	242
	Summation	62	96
$F^T F$		0.007	141
$GF^T F$		0	26
Update G		1.4	45

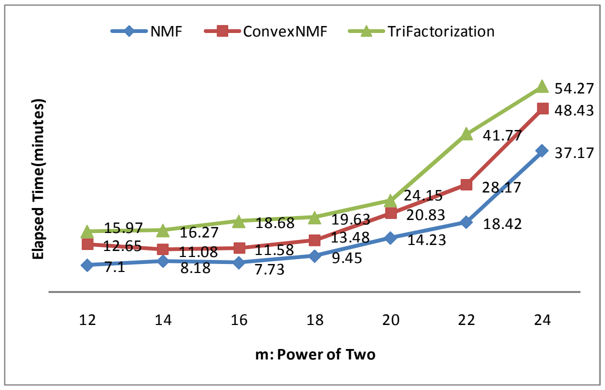

Figure 2. Elapsed Time w.r.t m.

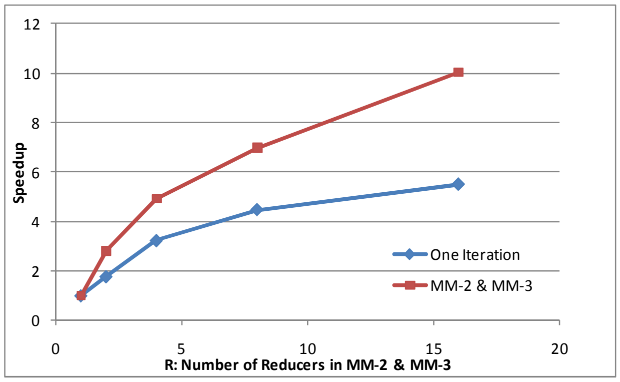

Figure 5. Speedup w.r.t R.

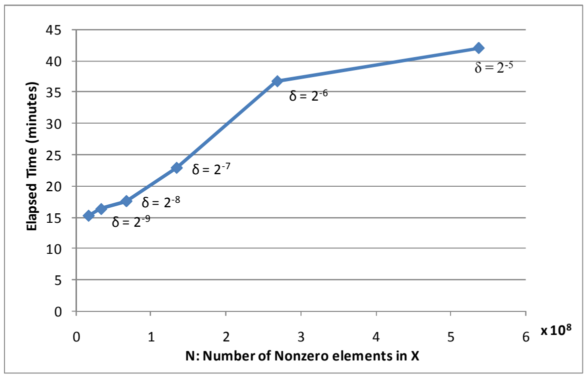

Figure 3. Elapsed Time w.r.t δ

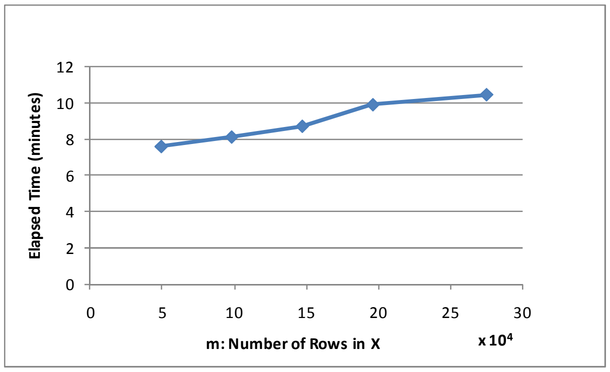

Figure 6. Elapsed time w.r.t m for real data set.

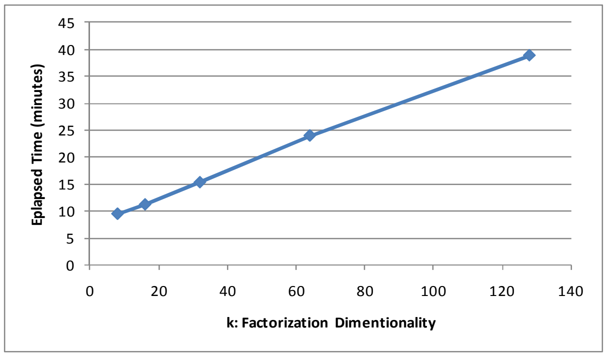

Figure 4. Elapsed Time w.r.t k.

VI. CONCLUSION

In this paper, we presented three different implementations of matrix multiplication on MapReduce depending on the properties of the matrices. Based on that, we successfully scaled up three different types of nonnegative matrix factorization algorithms. Matrices of dimension million-by-thousand with millions of nonzero elements can be factorized within several hours on a MapReduce cluster.

There are many avenues for future work on large-scale matrix factorization. First, our current work is focused on the multiplicative updating rules used in NMF which could be reduced to applications of different matrix multiplications. One interesting direction is to investigate schemes for scaling up other NMF algorithm such as alternating non-negative least squares [27] and projective gradient descent [28]. Second, recently tensor factorization, as a generalization of matrix factorization, has attracted a lot of research attention [29]. It is thus interesting to study schemes for scaling up large-scale tensor factorization. Last but not least, we would also like to explore various applications of NMF on large-scale data sets.

ACKNOWLEDGMENT

This research was supported in part by NSF grants IIS-0837716, CNS-0821345, HRD-0833093, IIP-0829576, IIP-0931517, CNS-1057661, IIS-1052625, CNS-0959985.

REFERENCES

[1] Lynn Elliot Cannon, “A cellular computer to implement the kalman filter algorithm,” Technical report, Ph.D. Thesis, Montana State University, 14 July 1969.

[2] Gene H. Golub and Charles F. Van Loan, Matrix Computations. 3rd ed, The Johns Hopkins University Press, 1996.

[3] J. J. MODI, Parallel Algorithms and Matrix Computation. Oxford: Clarendon Press, 1988.

[4] S. A. Robila and L. G. Maciak, “A parallel unmixing algorithm for hyperspectral images,” Intelligent Robots and Computer Vision XXIV, 2006.

[5] E. Batternberg and D. Wessel, “Accelarating Non-Negative Matrix Factorization for Audio Source Separation on Multi-core and Many-core Architectures,” 10th International Society for Music Information Retrieval Conference (ISMIR 2009).

[6] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (San Francisco, CA, December 06 - 08, 2004). Operating Systems Design and Implementation. USENIX Association, Berkeley, CA, 10-10.

[7] C. Chu et al, “MapReduce for Machine Learning on Multicore,” the Twentieth Annual Conference on Neural Information Processing Systems, 2006.

[8] S. Papadimitriou and J. Sun, “DisCo: Distributed Co-clustering with Map-Reduce: A Case Study towards Petabyte-Scale End-to-End Mining,” Proceedings of the 2008 Eighth IEEE international Conference on Data Mining (December 15 - 19, 2008), ICDM, IEEE Computer Society, Washington, DC, 2008, 512-521, doi:10.1109/ICDM.2008.142.

[9] U. Kang, C. Tsourakakis, and C. Faloutsos, “PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations,” the IEEE International Conference on Data Mining 2009.

[10] C. Liu, F. Guo, and C. Faloutsos, “BBM: bayesian browsing model from petabyte-scale data,” Proceedings of the 15th ACM SIGKDD international Conference on Knowledge Discovery and Data Mining (Paris, France, June 28 - July 01, 2009), KDD '09, ACM, New York, NY, 2009, 537-546, doi:10.1145/1557019.1557081.

[11] C. Liu, H. Yang, J. Fan, L. He, and Y. Wang, “Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce,” Proceedings of the 19th international Conference on World Wide Web (Raleigh, North Carolina, USA, April 26 - 30, 2010), WWW '10. ACM, New York, NY, 2010, 681-690, doi:10.1145/1772690.1772760.

[12] D. D. Lee and H. S. Seung, “Algorithms for Non-Negative Matrix Factorization,” NIPS, 2000.

[13] http://mahout.apache.org/

[14] C. Ding, T. Li, W. Peng, and H. Park, “Orthogonal nonnegative matrix tri-factorization for clustering,” Proc SIGKDD Int’l Conf. on Knowledge Discovery and Data Mining, 2006.

[15] http://hadoop.apache.org/

[16] Y. Chen, D. Pavlov, and J. F. Canny, “Large-scale behavioral targeting,” Proceedings of the 15th ACM SIGKDD international Conference on Knowledge Discovery and Data Mining (Paris, France, June 28 - July 01, 2009), KDD '09, ACM, New York, NY, 2009, 209-218, doi:10.1145/1557019.1557048.

[17] NSF Cluster Exploratory Program, http://www.nsf.gov/pubs/2008/nsf08560/nsf08560.htm

[18] Google&IBM Academic Cluster Computing Initiative, http://www.google.com/intl/en/press/pressrel/20071008_ibm_univ.html

[19] H. Ning, W. Xu, Y. Chi, Y. Gong, and T. Huang, “Incremental spectral clustering with application to monitoring of evolving blog communities,” SIAM Int. Conf. on Data Mining, 2007.

[20] B. Panda, J. Herbach, S. Basu, and R. Baryado, “PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce,” the 35th International Conference on Very Large Data Bases, 2009.

[21] C. Ding, X. He, and H.D. Simon, “On the equivalence of nonnegative matrix factorization and spectral clustering,” Proc. SIAM Data Mining Conf, 2005.

[22] T. Li and C. Ding, “The Relationships among Various Nonnegative Matrix Factorization Methods for Clustering,” Proceedings of the 2006 IEEE International Conference on Data Mining (ICDM 2006), Pages 362-371, 2006.

[23] E. Gaussier and C. Goutte, “Relation between PLSA and NMF and implications,” Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval, August 15-19, 2005.

[24] C. Ding, T. Li, and W. Peng, “On the Equivalence Between Nonnegative Matrix Factorization and Probabilistic Latent Semantic Indexing,” Computational Statistics and Data Analysis, 52(8): 3913-3927, 2008.

[25] C. Ding, T. Li, and M. I. Jordan, “Convex and Semi-Nonnegative Matrix Factorizations,” IEEE Transactions on Pattern Analysis and Machine Intelligence, 32(1): 45-55, 2010.

[26] Y. Chi, X. Song, D. Zhou, K. Hino, and B. L. Tseng, “On evolutionary spectral clustering,” ACM Trans. Knowl. Discov. Data 3, 4 (Nov. 2009), 1-30. doi:10.1145/1631162.1631165.

[27] M. B. Berry, M. Browne, A. N. Langville, V. P. Pauca, and R. J. Plemmons, “Algorithms and applications for approximate nonnegative matrix factorization,” Computational Statistics & Data Analysis Volume 52, Issue 1, 15 September 2007, Pages 155-173.

[28] Chih-Jen Lin, “On the Convergence of Multiplicative Update Algorithms for Nonnegative Matrix Factorization,” IEEE Transactions on Neural Networks 18 (6): 1589-1596, 2007.

[29] A. Shashua, and T. Hazan, “Non-negative tensor factorization with applications to statistics and computer vision,” Proceedings of the 22nd international Conference on Machine Learning (ICML '05), Pages 792-799, 2005.

[30] M. Kearns, “Efficient noise-tolerant learning from statistical queries,” J. ACM 45, 6, 983-1006, 1998.