sPCA: Scalable Principal Component Analysis for Big Data on Distributed Platforms

Tarek Elgamal[1] Maysam Yabandeh[2*] Ashraf Aboulnaga[1]
Waleed Mustafa[3*] Mohamed Hefeeda[1]
[1]Qatar Computing Research Institute, [2]Twitter, [3]NTG Clarity

ABSTRACT

Web sites, social networks, sensors, and scientific experiments currently generate massive amounts of data. Owners of this data strive to obtain insights from it, often by applying machine learning algorithms. Many machine learning algorithms, however, do not scale well to cope with the ever increasing volumes of data. To address this problem, we identify several optimizations that are crucial for scaling various machine learning algorithms in distributed settings. We apply these optimizations to the popular Principal Component Analysis (PCA) algorithm. PCA is an important tool in many areas including image processing, data visualization, information retrieval, and dimensionality reduction. We refer to the proposed optimized PCA algorithm as scalable PCA, or sPCA. sPCA achieves scalability via employing efficient large matrix operations, effectively leveraging matrix sparsity, and minimizing intermediate data. We implement sPCA on the widely-used MapReduce platform and on the memory-based Spark platform. We compare sPCA against the closest PCA implementations, which are the ones in Mahout/MapReduce and MLlib/Spark. Our experiments show that sPCA outperforms both Mahout-PCA and MLlib-PCA by wide margins in terms of accuracy, running time, and volume of intermediate data generated during the computation.

1. INTRODUCTION

Internet-scale web services collect terabytes of data from their users' activities such as clicks, visits, likes, and ratings. This data offers opportunities for extracting valuable insights about users and their interests which can enable service providers to improve their services and attract more customers. Making sense of tera-scale data is, however, a challenging task, because many current machine learning algorithms were designed for centralized computing systems where the entire dataset can fit in the memory of one computing node. This highlights the need for designing distributed machine learning algorithms that can process large volumes of data.

Distributed machine learning algorithms, however, introduce a new set of challenges. For example, most machine learning algorithms involve quite complex and inter-dependent computations, and dividing these computations among multiple computing nodes while preserving the accuracy and theoretical guarantees is a non-trivial task. More importantly, this division of load introduces a new problem, namely that partial results and intermediate data may need to be exchanged among computing nodes. If not carefully managed, this intermediate data may actually become the main bottleneck for scaling machine learning algorithms, regardless of the available number of computing nodes. This is in addition to the other common challenges in all distributed settings, such as scheduling tasks, handling failures, and balancing load.

In this paper, we take Principal Component Analysis (PCA) [23] as an important, complex machine learning algorithm to show several techniques that can be applied to address the challenges of big data analysis on distributed systems. PCA is a popular machine learning tool in many areas, including image processing [27], data visualization [20], compression [15], and information retrieval [5]. Moreover, since PCA reduces the dimensionality of the data, it is a key step in many other machine learning algorithms that do not perform well with high-dimensional data such as k-means clustering [14]. We start by conducting a thorough analysis of existing PCA algorithms and their scalability in distributed settings. Then, we design our distributed PCA algorithm starting from one of the current PCA algorithms that promises the best theoretical scalability. We propose a set of simple, but highly effective, optimizations that achieve substantial performance gains for the proposed distributed PCA algorithm. Most of these optimizations are applicable to other machine learning algorithms since they target primitive matrix operations commonly used in these algorithms, such as matrix multiplication, matrix mean-centering, and computing matrix norms.

Although traditional libraries such as ScaLAPACK [6] offer implementations for PCA and various other machine learning algorithms, they are targeted towards high-end HPC platforms. In contrast, although our proposed optimizations can also benefit HPC machine learning libraries, we focus on designing a scalable PCA algorithm for commodity distributed clusters that are available to almost all academic and industrial organizations. In addition, we consider recent distributed programming platforms that run on such clusters, such as MapReduce [10] and Spark [33]. These programming platforms offer many advantages over traditional ones such as MPI [4], including transparent handling of failures, load balancing, and task scheduling, which greatly facilitate the development of distributed code. There are currently multiple libraries that offer PCA for distributed clusters. Two quite popular examples are Mahout [2] on MapReduce and MLlib [3] on Spark. Our experiments, however, show that the PCA algorithms in these two libraries do not scale well to support big data analysis. For example, the imple-

[*] Work done while the author was at the Qatar Computing Research Institute.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2751520.

mentation of PCA in Mahout finished processing a 1 GB dataset in less than an hour on an 8-node cluster, where each node has 8 cores. When we applied the same algorithm on a dataset of 94 GB, we had to wait for *five days* for the algorithm to finish. Our algorithm, in contrast, finished in less than five hours. The situation is not much better for PCA in MLlib, which failed to process datasets with more than 6,000 dimensions.

This paper addresses the challenges of PCA for large-scale data, and makes the following contributions:

- Analysis of different methods for performing PCA and their limitations in handling large-scale datasets on distributed clusters. To the best of our knowledge, such a rigorous analysis was never done before, and it is crucial for selecting the proper PCA method for different environments and datasets.
- Design and implementation of an efficient PCA algorithm called *scalable PCA*, or *sPCA*, for large datasets on distributed commodity clusters. The design is general and can be implemented on different platforms. We implemented sPCA on the disk-based MapReduce and the memory-based Spark programming platforms.
- Extensive empirical study using large and diverse datasets to assess the performance of sPCA and compare it against other distributed PCA implementations. Our results show that sPCA can be several orders of magnitude faster than two solid and widely used state-of-the-art competitors: Mahout-PCA on MapReduce and MLlib-PCA on Spark. In addition, sPCA has better scalability and accuracy than these competitors. An important property of sPCA is that it generates a very small amount of intermediate data. This is useful for MapReduce since it means that sPCA has a low disk footprint, resulting in less disk and network I/O. For Spark, this property not only decreases network I/O, but also allows for the analysis of much larger datasets in the limited aggregate memory of the cluster. For example, on a dataset of more than 1.26 billion tweets from Twitter, sPCA generates 131 MB of intermediate data whereas Mahout-PCA generates 961 GB of intermediate data.

The rest of this paper is organized as follows. In Section 2, we present our analysis of different PCA methods in the literature. In Section 3, we present the proposed design of sPCA. We present our MapReduce and Spark implementations in Section 4. Section 5 presents our experimental evaluation. Section 6 summarizes the related work, and Section 7 concludes the paper.

2. ANALYSIS OF PCA ALGORITHMS

In this section, we analyze different methods for computing the principal components of a given dataset represented as a matrix. Although several PCA algorithms exist and are well known in the literature, to the best of our knowledge, they have never been analyzed and compared in a systematic manner, especially in the context of large-scale datasets and distributed processing environments. Due to space limitations, we only present the summary of our analysis. Detailed step-by-step derivations are presented in the companion technical report [17].

Distributed Execution Cost Model. We analyze all methods across two important metrics: time complexity and communication complexity. We consider the worst-case scenarios for both metrics. The time complexity is the upper bound on the number of computational steps needed by the algorithm to terminate. Some PCA algorithms run multiple iterations of the same code, where each iteration improves the accuracy of its predecessor by starting from a better initial state. The time complexity that we present is for a single iteration, as the number of iterations is typically bounded by a small constant.

During the distributed execution of a PCA algorithm, processing nodes may need to exchange data among each other, which we call intermediate data. The worst-case total size of the intermediate data is considered as the communication complexity. We note that most PCA algorithms work in multiple synchronous phases, and the intermediate data is exchanged at the end of each phase. That is, a phase must wait for the entire intermediate data produced by its predecessor phase to be received before its execution starts. Therefore, a large amount of intermediate data will introduce delays and increase the total execution time of the PCA algorithm, and hence the intermediate data can become a major bottleneck. The exact delay will depend on the cluster hardware (network topology, link speed, I/O speed, etc.) as well as the software platform used to manage the cluster and run the PCA code. Some software platforms, e.g., Hadoop/MapReduce, exchange intermediate data through the distributed storage system, while others, e.g., Spark, exchange data through shared virtual memory. For our analysis of communication complexity to be general, we consider the total number of bytes that need to be exchanged, and we abstract away the details of the underlying hardware/software architecture.

In addition, during our analysis, we identify the methods implemented in common libraries such as Mahout, MLlib, and ScaLAPACK. Mahout [2] is a collection of machine learning algorithms implemented on Hadoop MapReduce. MLlib [3] is a Spark implementation of some common machine learning algorithms. ScaLAPACK [6] is a library of linear algebra algorithms implemented for parallel distributed memory machines.

The notation we use in this paper is mostly consistent with Matlab's programing language. Variable names, including matrices, are composed of one or more letters. Multiplication is indicated with a star ($*$) between the variables. M' and M^{-1} are the transpose and inverse of matrix M, respectively. I is the identity matrix and $||M||_F^2 = \sum_{i=1}^{N}\sum_{j=1}^{D}(M_i^j)^2$ is the square of the Frobenius norm of the matrix M. Furthermore, M_i denotes row i of matrix M. We use M_i^j to refer to the jth element of vector M_i.

2.1 Basic PCA

Given a matrix Y of size $N \times D$ (N rows and D columns), a PCA algorithm obtains d principal components ($d \leq D$) that explain the most variance (and hence information) of the data in matrix Y [23, 29]. To be useful in practice, d is chosen to be much smaller than D, that is $d \ll D$. The principal components can be used to get better insights about the data. For example, in the image processing domain, PCA is used to obtain the principal facial components whose linear combination could recreate any face in the image dataset [26]. In information retrieval, the principal components explain the principal terms in a set of documents [34]. In addition, PCA could be used as a dimensionality reduction technique [14] when dealing with high-dimensional data. For example, the data of a matrix Y can be mapped on the principal components, without losing much information. The resulting matrix X (of size $N \times d$) can be obtained using the following formula: $X = Y * C$, where C is a $D \times d$ matrix containing the d principal components as its columns. Since matrix X is much smaller than the original matrix Y, it can be used as input to other machine learning algorithms such as k-means clustering. A simple method to perform PCA is to compute the covariance matrix of the input matrix Y. Then, compute the eigen-decomposition of the covariance matrix, and choose the eigenvectors that correspond to the largest d eigenvalues. Our analysis shows that the computational cost of this method is dominated by the computation of the covariance matrix, which is $O(ND \times \min(N,D))$. This is a very high computational cost, and thus this method is not suitable for large datasets. In addition to the computational cost, this method requires generating a large and dense covariance matrix of size $D \times D$ which incurs substantial communication cost, making the method not suitable for large datasets. This method is implemented in MLlib [3], and we refer to this method as MLlib-PCA. The method is also implemented in RScaLAPACK, which is an add-on package for the widely-used R programing language. RScaLAPACK uses the parallel linear algebra routines implemented in the ScaLAPACK library.

2.2 Computing PCA Using SVD

Another approach to PCA is using singular value decomposition (SVD) [29]. SVD decomposes a matrix into three matrices:

$$Yc = U * \Sigma * V'.$$

When the input matrix is mean-centered, i.e., $Yc = Y - Ym$ (where Ym is a vector of all the column means of Y), V gives the principal components of Yc. For the sake of simplicity, we use $Y - Ym$ to indicate that vector Ym is subtracted from each row of matrix Y. Some libraries, such as Mahout, provide PCA by performing SVD on the mean-centered input matrix.

Several methods have been proposed to compute the SVD of a matrix. We describe the two most common methods: the first method is suitable for dense matrices [11] and the second is suitable for sparse matrices [22].

SVD for Dense Matrices. Golub and Kahan [19] introduced a two-step approach for computing SVD: convert the input matrix to a bidiagonal one and then perform SVD on the bidiagonal matrix. Demmel and Kahan [11] improved this approach by adding another step before bidiagonalization, which is QR decomposition. We refer to this method as SVD-Bidiag, which has the following three steps for a given matrix Y: (i) compute the QR decomposition of Y, which results in an orthogonal matrix Q and an upper triangular matrix R; (ii) transform R to a bidiagonal matrix B; and (iii) compute SVD on B.

The SVD-Bidiag algorithm is implemented in RScaLAPACK. Our analysis shows that the computational complexity of the SVD-Bidiag algorithm is dominated by the QR decomposition and bidiagonalization steps, and is given by $O(ND^2 + D^3)$. Therefore, the SVD-Bidiag algorithm is only suitable when D is small.

Next, we analyze the communication overhead of the SVD-Bidiag algorithm. The algorithm involves the three main steps mentioned above, and each produces intermediate data that needs to be communicated to different computing nodes to continue the computation. Specifically, the QR decomposition step results in two matrices, $N \times d$ matrix Q and $d \times D$ matrix R. Thus, the intermediate data for this step is $O(Nd + Dd)$. The bidiagonalization step of R results in three matrices: $d \times d$ matrix U_1, $d \times D$ matrix B, and $D \times D$ matrix V_1, which makes the intermediate data for this step $O(d^2 + Dd + D^2)$ = $O(D^2)$. The SVD computation on the bidiagonal matrix B results in three matrices of the same dimensions as the ones computed in the bidiagonalization step, and thus has the same order of intermediate data $O(D^2)$. Therefore, the maximum amount of intermediate data between any two of the three steps is $O(\max((N+D)d, D^2))$, which is substantial for large datasets. Therefore, our analysis reveals a serious issue with the SVD-Bidiag algorithm, namely the communication complexity, which will be the bottleneck for scalability if this algorithm is used for processing big data on a distributed platforms.

SVD for Sparse Matrices. SVD can be computed efficiently for sparse matrices using Lanczos' algorithm [22], which has a computational complexity of $O(Nz^2)$, where z is the number of non-zero dimensions (out of D dimensions). We refer to this method as SVD-Lanczos, and it is implemented in popular libraries such as Mahout and GraphLab [1]. The SVD-Lanczos algorithm, however, is not efficient for performing PCA on large datasets, because the matrix must be mean-centered in order to obtain the principal components as a result of SVD. Since in many applications the mean of the matrix is not zero, subtracting the mean from a sparse matrix substantially decreases its sparsity. In this case, z will approach the full dimensionality D, and the cost for computing PCA using SVD-Lanczos will be $O(ND^2)$, which is prohibitive for large datasets.

2.3 Computing PCA Using Stochastic SVD

Randomized sampling techniques have recently gained popularity in solving large-scale linear algebra problems. The work in [21] describes a randomized method to compute approximate decomposition of matrices, which is referred to as stochastic SVD (SSVD). SSVD has two steps: (i) it uses randomized techniques to compute a low-dimensional approximation of the input matrix, and (ii) it performs SVD on the approximation matrix. The accuracy of the results depends on the performance of the randomized techniques and the size of the approximation matrix. Accuracy can be improved through running the randomization step multiple times. Therefore, SSVD has the flexibility of trading off the accuracy of the results with the required computational resources.

Our analysis shows that the computational complexity of SSVD is dominated by the first step, which is $O(DNd)$. This is a much better complexity than the previous techniques, because d is typically much smaller than D and is usually a constant. However, SSVD requires exchanging multiple intermediate matrices, which may cause a problem for scalability. Our analysis shows that the amount of intermediate data can be up to $O(\max(Nd, d^2))$.

The Mahout library implements PCA using SSVD on the mean-centered input matrix. Mean-centering may convert a sparse matrix to a dense one. To circumvent this problem, Mahout augments its implementation of SSVD with a PCA option. When this option is used, the SSVD algorithm computes the mean but stores it separately from the original sparse input. It, nevertheless, propagates the mean to all the matrix operations that are part of SSVD. We call this algorithm Mahout-PCA. Mahout-PCA is a close algorithm to our work, and is one of the algorithms against which we compare our proposed sPCA.

2.4 Probabilistic PCA

Probabilistic PCA (PPCA) [32] is a probabilistic approach to computing principal components of a dataset. PPCA is the basis for our scalable PCA (sPCA), and thus we present it in some detail. In this probabilistic approach, PCA is presented as a latent (unobserved) variable model that seeks a linear relation between a D-dimensional observed data vector $\mathbf{y}$ and a d-dimensional latent variable $\mathbf{x}$. The model is defined by:

$$\mathbf{y} = \mathbf{C} * \mathbf{x} + \mu + \varepsilon,$$

where $\mathbf{C}$ is a $D \times d$ transformation matrix (i.e, the columns of $\mathbf{C}$ are the principal components), $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, μ is the vector mean of $\mathbf{y}$, and $\varepsilon \sim \mathcal{N}(\mathbf{0}, ss * \mathbf{I})$ is white noise to compensate for errors. The value ss used in ε is a scalar value representing the average variance and it is estimated from the data. $\mathcal{N}(\mathbf{u}, \Sigma)$ denotes the Normal distribution with $\mathbf{u}$ mean and Σ covariance matrix.

The work in [32] shows that, given N observations $\{\mathbf{y_r}\}_1^N$ as the input data, the log likelihood of data is given by:

$$\mathcal{L}(\{\mathbf{y_r}\}) = \sum_{r=1}^{N} \ln\{p(\mathbf{y_r})\}.$$

Method to Compute PCA	Time Complexity	Communication Complexity	Example Libraries
Eigen decomp. of covariance matrix	$O(ND \times \min(N,D))$	$O(D^2)$	MLlib-PCA (Spark), RScaLAPACK
SVD-Bidiag [11]	$O(ND^2 + D^3)$	$O(\max((N+D)d, D^2))$	RScaLAPACK
Stochastic SVD (SSVD) [21]	$O(NDd)$	$O(\max(Nd, d^2))$	Mahout-PCA (MapReduce)
Probabilistic PCA (PPCA) [32]	$O(NDd)$	$O(Dd)$	sPCA (our algorithm)

Table 1: Comparison of different methods for computing PCA of an $N \times D$ matrix to produce d principal components.

Thus, the *Maximum Likelihood Estimate (MLE)* of $\mathbf{C}$ is obtained by optimizing:

$$\underset{\mathbf{C}}{arg\,max}\,\mathscr{L}(\{\mathbf{y_r}\}). \tag{1}$$

The main idea behind the Probablistic PCA algorithm described in [32] is that the MLE solution of Equation (1) is equivalent to the solution of PCA. Moreover, [32] proposed an *Expectation Maximization (EM)* [12] algorithm to optimize the likelihood of Equation (1). EM is a well-known method to optimize the likelihood of models when a closed form solution does not exist. This algorithm is the basis for our sPCA algorithm and it will be described in detail later in this section and in the rest of the paper.

In the following steps, we show how the likelihood term $\mathscr{L}(\{\mathbf{y_r}\})$ is derived. It is shown in [32] that the conditional distribution of $\mathbf{y}$ given $\mathbf{x}$ is:

$$p(\mathbf{y}|\mathbf{x}) = (2\pi * ss)^{-D/2}\exp[-\frac{1}{2ss}\|\mathbf{y} - \mathbf{C} * \mathbf{x} - \mu\|^2].$$

With the assumed prior distribution on $\mathbf{x}$ as:

$$p(\mathbf{x}) = (2\pi)^{-d/2}\exp[-\frac{1}{2}\mathbf{x}' * \mathbf{x}],$$

we can obtain the marginal distribution of $\mathbf{y}$, $p(\mathbf{y})$, by first obtaining the joint distribution $p(\mathbf{x},\mathbf{y})$,

$$p(\mathbf{x},\mathbf{y}) = p(\mathbf{y}|\mathbf{x})p(\mathbf{x}).$$

Then, we integrate the joint distribution over $\mathbf{x}$ to get

$$\begin{aligned} p(\mathbf{y}) &= \int p(\mathbf{y}|\mathbf{x})p(\mathbf{x})\mathrm{d}x \\ &= (2\pi)^{-D/2}|\mathbf{M}|^{-1/2}\exp[-\frac{1}{2}(\mathbf{y} - \mu)' * \mathbf{M}^{-1} * (\mathbf{y} - \mu)], \end{aligned}$$

which is a Normal distribution with mean μ and covariance matrix $\mathbf{M}$ defined by:

$$\mathbf{M} = ss * \mathbf{I} + \mathbf{C} * \mathbf{C}'.$$

Hence, the log likelihood of data is given by:

$$\begin{aligned} \mathscr{L}(\{\mathbf{y_r}\}) &= \sum_{r=1}^{N} \ln\{p(\mathbf{y_r})\}, \\ &= -\frac{N}{2}\{D * \ln(2\pi) + \ln|\mathbf{M}| + \mathrm{tr}(\mathbf{M}^{-1} * \mathbf{S})\}, \end{aligned}$$

where $\mathbf{S}$ is the sample covariance matrix of $\{\mathbf{y_r}\}$ given by

$$\mathbf{S} = \frac{1}{N}\sum_{r=1}^{N}(\mathbf{y_r} - \mu) * (\mathbf{y_r} - \mu)',$$

where $\mathrm{tr}(\mathbf{A})$ is the trace of matrix $\mathbf{A}$, which is the sum of elements on the diagonal. The work in [32] shows that the MLE solution of Equation (1) is equivalent to the solution of PCA, namely the eigenvectors of the sample covariance matrix $\mathbf{S}$, up to an arbitrary rotation matrix. In addition, PPCA offers two desirable properties. First, large datasets often have missing values. Since PPCA uses expectation maximization, the projections of principal components can be obtained even when some data values are missing. Second, multiple PPCA models can be combined as a probabilistic mixture for better accuracy and to express complex models.

Algorithm 1 PPCA (Matrix Y, int N, int D, int d)

```
$C = normrnd(D,d)$
$ss = normrnd(1,1)$
$Ym = columnMean(Y)$
$Yc = Y - Ym$
while not STOP_CONDITION do
    $M = C' * C + ss * I$
    $X = Yc * C * M^{-1}$
    $XtX = X' * X + ss * M^{-1}$
    $YtX = Yc' * X$
    $C = YtX/XtX$
    $ss2 = trace(XtX * C' * C)$
    $ss3 = \sum_{n=1}^{N} X_n * C' * Yc'_n$
    $ss = (||Yc||_F^2 + ss2 - 2 * ss3)/N/D$
end while
```

Algorithm 1 depicts the pseudo code of PPCA in [32]. In Algorithm 1, Y is the input matrix of size $N \times D$ and d is the desired number of principal components. In other words, matrix Y has the N observations $\mathbf{y_r}$ as its rows. The function $normrnd(r,c)$ gives a random matrix of size $r \times c$ with Normal distribution. The function $trace$ obtains the trace of the matrix. $||Yc||_F^2$ is the square of the Frobenius norm of the mean-centered input matrix. The algorithm requires computing many intermediate variables, among which we have X, the matrix that has N latent variables $\mathbf{x_r}$ as its rows. The algorithm initializes the transformation matrix C and the variance ss with random values. At each iteration, it improves the values of C and ss until it reaches the STOP_CONDITION. Our analysis shows that the time complexity of PPCA is $O(NDd)$. Section 3 uses Algorithm 1 as the starting point for the design of sPCA.

2.5 Summary of the Analysis

The summary of our analysis is shown in Table 1; details are given in [17]. As the table shows, the time complexities of the top two methods (eigen decomposition of covariance matrix and SVD of bi-diagonalized matrix) are a function of N (number of data points) multiplied by D^2 (number of dimensions of each data point), which is quite high for many datasets with a large number of dimensions. In addition, the communication complexities of these two methods are also quite high, especially for high dimensional datasets. Therefore, even if there are enough computing nodes to handle the high computational costs, the communication costs can still hinder the scalability of these two methods.

The last two methods in Table 1 (stochastic SVD and probabilistic PCA) have a more efficient time complexity of $O(ND)$, assuming that d is a relatively small constant, which is typically the case in many real applications. Thus, these two approaches are potential candidates for performing PCA for large datasets. Our analysis and experimental evaluation (in Section 5), however, reveal that even though the time complexity of stochastic SVD can be handled by employing more computing nodes, it can suffer from high communication complexity. For example, our experiments show that the high communications complexity of Mahout-PCA (which uses SSVD) prevents it from processing datasets in the order of tens of GBs. Therefore, based on our analysis, the most promising PCA approach for large datasets is the probabilistic PCA.

Next, we present our sPCA, which is based on probabilistic PCA. sPCA runs in a distributed environment in a way that minimizes the communication complexity while maintaining all the theoretical guarantees provided by the original probabilistic PCA on accuracy and time complexity. This makes sPCA suitable for large datsets. In addition, our design and optimization approach is useful in its own right to scale other machine learning algorithms.

3. DESIGN OF SPCA

In this section, we present the design of sPCA, our scalable implementation of PPCA for distributed platforms such as MapReduce and Spark. A naive approach for implementing PPCA is to have a distributed (e.g., MapReduce) job for each linear algebra operation in Algorithm 1. The dependency between these jobs is depicted in the job graph in Figure 1. Each node is labeled with the variable that the job produces. A link from node A to node B indicates that data of variable A must be computed before starting the job that computes variable B. Variables carried over from the previous iteration are distinguished with the index i. Variable Y is the input to the algorithm and does not change between iterations. The output (the principal components of Y) is in C_i.

This simple PPCA implementation works as follows. Matrix M is computed using matrix C_{i-1} and variance ss_{i-1} that are carried over from the previous iteration. For the first iteration ($i = 1$), C_0 and ss_0 are initialized randomly from a Normal distribution. Then, matrices M and C_{i-1} as well as the input matrix Y are used to generate the intermediate matrix X. Matrix X is used for three other computations. First, it is used together with the variance ss_{i-1} and the computed matrix M to create matrix XtX. The second consumer of X is its product with the transpose of input matrix Y (YtX). This matrix is divided by matrix XtX to produce the next version of principal components, C_i. The third consumer of X is $ss3$, part of the variance, which needs C_i and YtX that were computed in the last two steps. Eventually, the variance ss_i is updated for the next iteration based on 3 components: (i) Frobenius norm of the input matrix, (ii) $ss2$, which is the trace of the product of XtX and $C_i' * C_i$, and (iii) $ss3$, which is computed in the last step.

As depicted in Figure 1, there are many linear algebra operations per iteration and the naive approach results in poor performance. In the following, we present our proposed sPCA algorithm. We present our design as successive optimization ideas in separate subsections. Then, we put all optimizations together in the final subsection. We emphasize that our optimization ideas do not change any theoretical properties of PPCA.

We first note that not all operations in Figure 1 need to be performed in a distributed manner. In fact, after careful inspection of the algorithm and its various data structures, we found that only three jobs need to be computed in a distributed manner because they operate on large matrices that cannot fit in the memory of a single machine. These jobs are X, YtX, and $ss3$, and we highlight them in Figure 1 by dotted rectangles. All other operations can easily run on a single machine, even for very large datasets. Specifically, our implementation of sPCA has one main driver program, which implements the control flow, launches parallel operations for the three jobs X, YtX, and $ss3$, and executes all other operations locally.

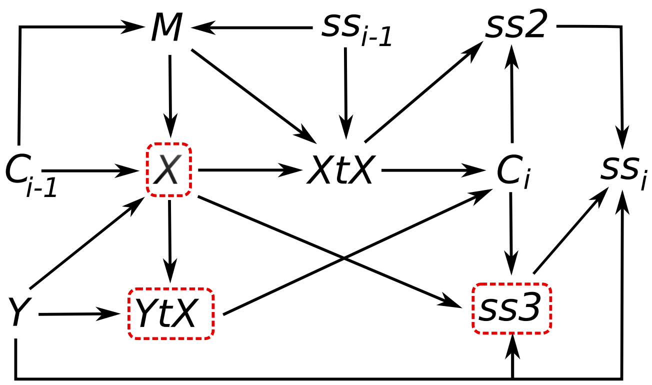

Figure 1: The job graph of PPCA. Nodes are labeled with variable names in Algorithm 1. Variables carried over from the previous iteration are indexed by $i-1$. Dotted rectangles indicate distributed jobs. The input to the algorithm is matrix Y and the output is C_i, which contains the principal components. For the first iteration ($i = 1$), C_{i-1} and ss_{i-1} are randomly initialized using Normal distributions.

3.1 Mean Propagation to Leverage Sparsity

The first optimization we propose is the mean propagation idea, which preserves and utilizes the sparsity of the input matrix Y. PPCA requires the input matrix to be mean-centered (denoted by Yc), meaning that the mean vector Ym must be subtracted from each row of the original matrix Y. Large matrices, however, are mostly sparse, with many zero elements. Sparse matrices can achieve a small disk and memory footprint by storing only non-zero elements, and performing operations only over non-zero elements. Subtracting the non-zero mean from the matrix would make many elements non-zero, so the advantage of sparsity is lost. The algorithm would incur much more (i) disk I/O operations, (ii) network I/O operations, and (iii) CPU time for operations that could otherwise be skipped for zero elements.

To avoid the problems of subtracting the mean, we keep the original matrix Y and the mean Ym in two separate data structures. We do not subtract the mean Ym from Y. Rather, we propagate the mean throughout the different matrix operations. For example, if the algorithm has a step like $Yc * C$, we change it to be:

$$\begin{aligned} Yc * C &= (Y - Ym) * C \\ &= Y * C - Ym * C. \end{aligned}$$

That is, the mean Ym is propagated and multiplied with C, and at the same time the sparse matrix Y is efficiently multiplied with C. We apply the same technique on all the algebraic matrix operations of Algorithm 1. This optimization is quite useful for algorithms that require a matrix to be mean-centered.

3.2 Minimizing Intermediate Data

As explained in Section 2, intermediate data can slow down the distributed execution of any PCA algorithm, because it needs to be transferred to other nodes for processing to continue. For example, at each iteration of running the basic PPCA on a 94 GB input dataset with 50 principal components, nearly 500 GB of intermediate data was created in our initial implementation, which was one of the main bottlenecks.

Through analysis and profiling of early implementations of sPCA, we found that the intermediate matrix X can potentially have large size. And as shown in Figure 1, X has to be fed to all the three

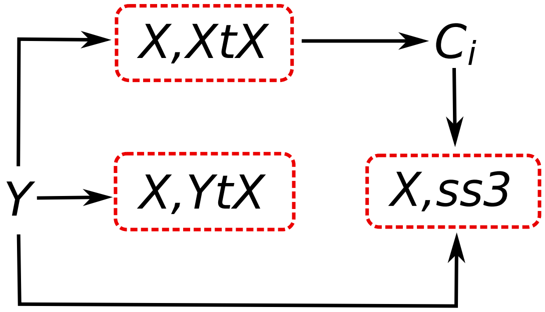

Figure 2: Part of the job graph in Figure 1 focusing only on distributed jobs. It shows our optimization of reducing intermediate data by redundantly computing X in the three jobs.

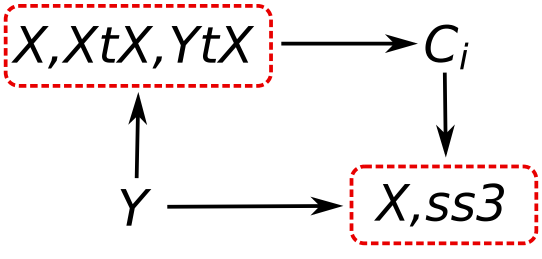

Figure 3: Final job graph of sPCA, based on Figure 2, and further optimized by merging two distributed jobs into one.

distributed jobs, so it can become a major scalability bottleneck. To minimize the size of the intermediate data X, we propose two ideas: redundant computation and distributed job consolidation. First, we note that while storing and exchanging X is expensive due to its large size, computing it is a relatively lightweight operation when we use in-memory matrix multiplication: each row of X requires multiplying a sparse row of Y with a small, in-memory, matrix $C*M^{-1}$ (Algorithm 1). To leverage this property, we redesign the algorithm by redundantly recomputing X at each job that consumes it as input. This approach essentially trades intermediate data footprint with computation. Figure 2 illustrates this optimization. The figure shows only the part of the original job graph in Figure 1 where matrix X is recomputed in each of the three distributed jobs.

Our second optimization is job consolidation, which means merging multiple distributed jobs into one in order to reduce the communication between these jobs. Since there is no dependency between the XtX and YtX jobs in Figure 2, we consolidate them into one job. This also reduces the number of times that X is redundantly computed. Figure 3 shows the final job graph of the distributed part of sPCA. This job graph is plugged into the job graph in Figure 1.

3.3 Efficient Matrix Multiplication

As shown in Algorithm 1, PPCA requires many matrix multiplications, which are expensive operations in a distributed setting. To appreciate the techniques that sPCA employs to overcome the inefficiency of matrix multiplication, we briefly explain different possible implementations of this operation. Semantically, the product of two matrices A of size $N\times D$ and B of size $D\times M$ is matrix $A*B$ of size $N\times M$ and can be defined as:

$$(A*B)_i^j = \sum_{k=1}^{D} A_i^k * B_k^j.$$

This computation requires many random accesses to the two matrices, which makes it inefficient when the two matrices are distributed. There are, nevertheless, variants of matrix multiplication that can be implemented efficiently. For example, if instead of $A*B$ we want to compute $A'*B$, then we can use the following equivalent formula:

$$(A'*B) = \sum_{i=1}^{D} (A_i)' * B_i, \tag{2}$$

which requires accessing one row at a time from A' and B. Some libraries, e.g., Mahout, use this approach for matrix multiplication. Mahout obtains the transpose of matrix A and then uses map-side join to multiply the corresponding rows from the two matrices. The result is then sent to reducers, which sum up the received partial matrices. This approach still requires an extra matrix transpose operation, as well as transferring a large amount of data between the mappers and reducers. The map-side join also requires nontrivial initialization time to align the partitions of the two matrices.

For sPCA, we seek a more efficient matrix multiplication operation. Notice that if matrix B can entirely fit in memory, we can benefit from the following equivalent equation:

$$(A*B)_i = A_i * B.$$

Using the above equation, matrix multiplication could be implemented by distributing only the first matrix among different nodes and loading the entire second matrix into the memory of each node. Specifically, we partition the large matrix, A, among multiple nodes and matrix B is loaded into the memory of each node. Each node reads a row from its partition of A, multiplies it with the in-memory matrix B, and produces one row of the result matrix. This approach does not require matrix transpose, and is more efficient.

An example of a matrix multiplication operation in Algorithm 1 that can benefit from in-memory matrix multiplication is the product of the input matrix Yc and matrix C, which is of size $D\times d$ (recall that d is typically small). For example, in our experiments with a 94 GB dataset, the size of matrix C was 30 MB, which can easily fit in memory.

It is, nevertheless, not always possible to benefit from this technique since the second matrix could be large. For example, in the PPCA algorithm, the calculation of matrix YtX requires a product between the transpose of Yc and X: $YtX = Yc'*X$. This operation was actually a bottleneck in our first prototype of sPCA. In the new design, however, since we generate matrix X on-demand, one row of X is generated at a time which allows for efficient implementation of matrix multiplication using Equation (2).

3.4 Efficient Frobenius Norm Computation

The PPCA algorithm requires computing the Frobenius norm of the mean-centered input matrix $Yc = Y - Ym$. Recall that we store the mean vector Ym separately from the matrix Y to avoid creating the dense matrix Yc. Applying the same technique, we can compute the rows of Yc online, right before computing the Frobenius norm. Algorithm 2 shows this approach.

Although Algorithm 2 has the advantage of requiring a small amount of memory to maintain only one dense row at a time, it still requires iterating over the dense row Y_d, which is much larger than the original sparse row Y_s. To solve this problem, we design Algorithm 3 which does not even require creating the dense vector. We note that many machine learning algorithms compute various norms of matrices. The proposed method for optimizing the computation of the Frobenius norm can be extended to other matrix norms using similar ideas. Thus, this simple optimization can benefit several other machine learning algorithms.

Algorithm 2 Frobenius-simple (Matrix Y, Vector Y_m) : double

```
for all $Y_s$ in $Y.rows$ do
    $Y_d = Y_s - Y_m$
    for all $Y_d^j$ in $Y_d$ do
        $sum += (Y_d^j)^2$
    end for
end for
return $sum$
```

Algorithm 3 Frobenius (Matrix Y, Vector Y_m) : double

```
for all $Y_m^j$ in $Y_m$ do
    $msum += (Y_m^j)^2$
end for
for all $Y_s$ in $Y.rows$ do
    for all $Y_s^j$ in $Y_s$ do
        $sum += (Y_s^j - Y_m^j)^2$
        $sum -= (Y_m^j)^2$
    end for
    $sum += msum$
end for
return $sum$
```

In this approach, we first compute the Frobenius norm of the mean matrix, which would be equal to the norm of the sparse matrix if all elements of the sparse matrix were zero. We then subtract the mean value only from non-zero elements and add the square of the results to the norm being computed. We also cancel the effect of considering the square of the mean for each non-zero element by subtracting it from the norm being computed.

3.5 sPCA: Putting it All Together

Algorithm 4 shows the pseudo code of sPCA including all the techniques that we described in the previous sections. The parts of the algorithm that are run with distributed jobs are highlighted with bold font. The C and ss variables are initialized with random values. Before starting the iterations, we run two lightweight jobs to compute the column mean and Frobenius norm of the input matrix. $YtXJob$ computes both XtX and YtX variables. It generates row r of X on demand using row r of the input matrix Y, the in-memory matrix CM, as well as the mean Ym and its effect on X (Xm). $ss3Job$ computes the third part of variance. Similar to $YtXJob$, $ss3Job$ generates the rows of X on-demand. The error computation and the condition $STOP\_CONDITION$ are described in Section 5.1

4. IMPLEMENTATION OF sPCA

In this section, we show that the design of sPCA and the optimizations it uses are not restricted to a specific platform; they are valid for the disk-based MapReduce and the in-memory Spark. The implementation described in this section is open source and available for download[1].

4.1 Implementation in MapReduce

This section provides a brief description on the MapReduce implementation of the two main parallel jobs in sPCA (YtXJob and ss3Job). These two jobs are run in each iteration of sPCA. There are another two MapReduce jobs (meanJob and FnormJob) which are lighter weight and run once before the main loop of the algorithm.

Algorithm 4 sPCA (Matrix Y, int N, int D, int d)

```
$C = normrnd(D,d)$
$ss = normrnd(1,1)$
$Ym = \mathbf{meanJob}(Y)$
$ss1 = \mathbf{FnormJob}(Y)$
while not STOP_CONDITION do
    $M = C' * C + ss * I$
    $CM = C * M^{-1}$
    $Xm = Ym * CM$
    $\{XtX, YtX\} = \mathbf{YtXJob}(Y, Ym, Xm, CM)$
    $XtX + = ss * M^{-1}$
    $C = YtX / XtX$
    $ss2 = trace(XtX * C' * C)$
    $ss3 = \mathbf{ss3Job}(Y, Ym, Xm, CM, C)$
    $ss = (ss1 + ss2 - 2 * ss3)/N/D$
end while
```

At a high level, programs in the MapReduce framework are divided into mappers, reducers, and optionally combiners. Mappers execute user-specified function on different parts of the dataset. The results are then sorted and directed to reducers, which will aggregate and produce the final results. Combiners can be applied to the mappers' output before feeding them to reducers.

In sPCA, the mapper of the YtXJob operates on each row of the input matrix Y, Y_i, and then generates X_i, the corresponding row of intermediate matrix X. It then uses Equation (2) to generate a partial result for XtX and YtX. The partial results have to be summed up in the combiners and eventually in the reducers to generate the full results. This, however, makes each mapper generate an entire dense matrix after processing each sparse row. To solve this problem, we make the mapper keep two in-memory matrices $XtX\text{-}p$ and $YtX\text{-}p$ to maintain the partial results and add them with the partial sums after processing each row. At the end of processing for a mapper, when the MapReduce framework calls the cleanup method, the mapper writes the entire partial matrices $XtX\text{-}p$ and $YtX\text{-}p$ to the output. We refer to this technique as using a *stateful combiner*. This technique results in much less load on the combiners, and saves CPU cycles.

To send both $XtX\text{-}p$ and $YtX\text{-}p$ to reducers we use a composite key. Since $XtX\text{-}p$ is of size $d \times d$, where d is the number of principal components and thus small, we define the composite key to send all the partial $XtX\text{-}p$ matrices to the same reducer. That reducer then sums them up in memory and writes the resulting XtX into HDFS. Matrix YtX, on the other hand, is generated using the normal output interface of reducers.

Similar to YtXJob, ss3Job generates the rows of X on demand. After producing each row X_i, it does the following computation:

$$X_i * C' * Y_i'. \qquad (3)$$

The default way to do this computation is to first perform $X_i * C'$ and then multiply the result by Y_i'. This, however, is not efficient, because vector Y_i' is sparse, and thus most of the work to compute elements in $(X_i * C')$ will be wasted since most of these elements will be multiplied with zero elements in Y_i'. Using the associativity property of matrix multiplication, we perform this operation as $X_i * (C' * Y_i')$, i.e., first multiply matrix C' with the sparse vector Y_i', and then obtain its dot product with X_i, which is efficient since both are of small size d. In addition, the mapper output of this job is a scalar, which reduces the amount of intermediate data.

[1] `https://github.com/Qatar-Computing-Research-Institute/sPCA`

Algorithm 5 YtXSparkJob (Matrix Y, Vector Ym ,Vector Xm, Matrix CM, int D, int d)

```
$YtXSum = spark.accumultor(newMatrix(D,d))$
$XtXSum = spark.accumulator(newMatrix(d,d))$
$Y.map\{Yi =>$                                    ▷ runs in parallel
    $Xi = Yi*CM - Ym*CM$
    $(YtX)i = Yi' * (Xi - Xm) - Ym' * (Xi - Xm)$
    $(XtX)i = Xi' * (Xi - Xm) - Xm' * (Xi - Xm)$
    $YtXSum.add((YtX)i)$
    $XtXSum.add((XtX)i)\}$
YtX=YtXSum.value()
XtX=XtXSum.value()
```

4.2 Implementation in Spark

Spark [33] provides two main abstractions for parallel programming: *resilient distributed datasets (RDDs)* and parallel operations on these datasets. An RDD is a collection of records that can be operated on in parallel. An RDD is partitioned across multiple machines and users can control its persistence (e.g., cache in memory or store on disk) and its partitioning (e.g., partition by key). Developers typically define one or more RDDs through *transformations* on data in stable storage. Examples of transformations include *map* (which returns a new distributed dataset formed by passing each element of the source through a user-defined function) and *filter* (which returns a new dataset formed by selecting those elements of the source on which a user-defined function returns true.). Developers can then use these RDDs in *actions*, which are operations that return a value to the application or export data to a storage system. Examples of actions include *count* (which returns the number of elements in the dataset), *collect* (which returns the elements themselves), and *save* (which outputs the dataset to a storage system).

sPCA is designed to leverage the in-memory computations provided by Spark through making the input matrix Y persistent in the memory of the cluster nodes and performing distributed operations on it repeatedly. This approach translates to much less disk and network I/O. The disk I/O is limited to the amount of data that does not fit in the aggregate memory of the cluster.

Algorithm 5 presents the pseudo code of the YtXJob implemented on Spark. The code makes use of a special type of variables provided by Spark called *accumulators*. Accumulators are variables that workers can only add to using an associative operation, and that only the driver can read. The *map* operation of the YtXJob operates on each row of the input matrix Y, Y_i, and then generates X_i, the corresponding row of intermediate matrix X. It then uses Equation (2) to generate the partial result XtX_i and YtX_i. We note that the partial results are summed up in the same map operation using the accumulators $XtXSum$ and $YtXSum$, thus eliminating the need for reduce operations and achieving good scalability. The results of the accumulators are read later in the driver program after all map tasks finish execution. We note that YtX_i is the product of the sparse vector Y_i' of length $D \times 1$ and the vector X_i of length $1 \times d$ and hence, YtX_i is a sparse $D \times d$ matrix. In order to make use of this sparsity we only pass the indices of the sparse entries of YtX_i to the accumulator *YtXSum*. This results in significant improvement in the running time since the complexity of this operation was reduced from $O(D \times d)$ to $O(z \times d)$, where z is the number of non-zero dimensions (out of D dimensions).

The implementation of the ss3Job in Spark follows the same steps described for the MapReduce implementation (Section 4.1) to optimize the computation described in Equation (3).

5. EVALUATION

In this section, we present a rigorous evaluation of sPCA comparing it against the closest algorithms using multiple real datasets from different domains.

Algorithms Compared. We compare four methods for computing PCA:

- sPCA-MapReduce: sPCA implementation on MapReduce,
- sPCA-Spark: sPCA implementation on Spark,
- Mahout-PCA: PCA implementation in Mahout [2] on MapReduce, and
- MLlib-PCA: PCA implementation in MLlib [3] on Spark.

Both the Mahout-PCA and MLlib-PCA implementations are quite popular and optimized, and we found them to be the best options for their respective platforms. For uniform comparison, all PCA algorithms compute 50 principal components.

Datasets. We use four real datasets, which are quite diverse in terms of the domain they come from, size, number of dimensions in the data, sparsity, and ranges/types of values for each data item. We use various subsets of the datasets to assess the scalability and performance of the considered PCA algorithms with increasing data sizes. The datasets are:

- *Tweets:* A large set of tweets from the Twitter social network. We construct a matrix such that the rows represent the tweets and the columns represent all words that appear in each tweet. The matrix is of size $1,264,812,931 \times 71,503$, and each element is either 1 or 0, where 1 means the corresponding word appeared in that tweet, and 0 means otherwise. When we store only the non-zero elements, this matrix occupies about 94 GB.
- *Bio-Text:* A set of 8 million biomedical documents collected from the U.S. National Library of Medicine, which is the largest medical library in the world. The collection includes books, journals, and technical reports on medicine and related sciences. We construct a matrix from this dataset, where the rows represent the documents and the columns represent the distinct words in each document. The matrix size is $8,200,000 \times 141,043$, and each element is either 1 or 0, where 1 means the corresponding word appeared in that document, and 0 means otherwise. The non-zero elements of this matrix occupy about 4.9 GB.
- *Diabetes:* This dataset is collected from 353 patients. A urine sample is taken from each patient. Then, a magnetic field is applied on the urine samples. The nuclear magnetic resonance (NMR) is then measured for the molecules (metabolites) in the urine. NMR is a phenomenon where the molecules absorb and re-emit electromagnetic radiation. This energy is at a specific resonance frequency that depends on the strength of the magnetic field and the magnetic properties of the atoms. The data represents the magnitude of this energy at each frequency. The magnitude is measured at 65,669 different frequencies for each patient. Thus, we construct a matrix of size $353 \times 65,669$, where rows represent patients and columns represent sample frequencies. Unlike previous datasets that have binary elements, the elements in this dataset are real values representing the magnitude of radiation at different frequencies.
- *Images:* A dataset of 160 million data vectors. These vectors are visual features extracted from 1 million images downloaded from ImageNet [13]. From each image, we extract an average of 160 SIFT [25] features, where each SIFT feature is a vector of 128 dimenstions. This results in a dataset of 160 million vectors. The matrix is dense and of size $160,000,000 \times 128$, where rows represent the data vectors, columns represent dimensions of each vector, and each element is a real value that represents the texture of some part of an image.

Dataset	Size	sPCA-Spark	MLlib-PCA	sPCA-MapReduce	Mahout-PCA
Tweets	$1.26B \times 2K$	708	822	$3,900$	$29,160$
	$1.26B \times 6K$	$1,260$	$2,196$	$10,080$	$97,920$
	$1.26B \times 71.5K$	$5,940$	Fail	$16,200$	$430,200$
Bio-Text	$8.2M \times 2K$	48	102	$1,050$	$2,280$
	$8.2M \times 10K$	114	Fail	$1,290$	$6,240$
	$8.2M \times 14K$	516	Fail	$1,740$	$8,580$
Diabetes	$353 \times 2K$	20	55	540	720
	$353 \times 10K$	30	Fail	720	$1,680$
	$353 \times 65.7K$	156	Fail	960	$3,300$
Images	$160M \times 128$	$7,800$	660	$12,600$	$117,700$

Table 2: Comparison of running time (in sec) for sPCA on both Spark (sPCA-Spark) and MapReduce (sPCA-MapReduce) against the closest counterparts on Spark (MLlib-PCA) and MapReduce (Mahout-PCA).

Performance Metrics. We consider three performance metrics: accuracy, running time to achieve a target accuracy, and size of the intermediate data.

We measure the accuracy by computing the 1-Norm of the reconstruction error, which is given by: $e = ||Y - X * C^{-1}||_1$. Although this provides a common way to compare the accuracy of different algorithms, the reconstruction error is a big, dense matrix which is costly to store and process. We reduce the cost of storage by computing the error row by row, avoiding the need to store the large reconstruction matrix in the file system. Nevertheless, iterating over the resulting dense rows is still time consuming. We reduce this time by measuring the error only on a random subset of the rows, Yr. To have a unique way to interpret the measured error, independent of the sampling rate or matrix size, we report the norm of the reconstruction error divided by the norm of the matrix made up of the randomly selected rows, which is:

$$e = ||Yr - Xr * C^{-1}||_1 / ||Yr||_1.$$

In addition, we measure the ideal accuracy that can be achieved with 50 principal components after a large number of iterations. After each iteration, we report the percentage of the ideal accuracy that is achieved.

The intermediate data size is the amount of data generated by each algorithm during its execution. We note that in many cases the intermediate data generated by the algorithm far exceeds the size of the input data, and thus becomes a major bottleneck.

Cluster Specifications. We run the experiments on the Amazon EC2 cloud. We created a cluster of 8 Amazon EC2 m3.2xlarge instances, where each node has 8 cores and 32 GB of memory. The cluster runs Linux Red Hat 4.6.3. Amazon Hadoop distribution 0.20.205 and Apache Spark 1.0 were installed on the cluster. The Amazon Hadoop distribution is based on Apache Hadoop, with patches and improvements added that make it work efficiently with Amazon Web Services (AWS).

Experiments Conducted. We first compare the running times of all algorithms on the four datasets. Then, we conduct detailed evaluation and comparison on MapReduce and Spark, separately. Finally, we isolate and study the effect of each of the proposed optimizations on the performance of sPCA.

5.1 Comparison of All Algorithms

We measure the running time of all algorithms on the four datasets, and for each dataset we choose several sizes. A representative sample of our results is shown in Table 2. We note that MLlib-PCA is a deterministic algorithm that terminates after performing a fixed number of matrix operations, unlike sPCA and Mahout-PCA which are iterative algorithms that keep refining the principal components until they reach a target accuracy. Hence, we compare the running time of MLlib-PCA with that of sPCA and Mahout-PCA based on the time needed for sPCA and Mahout-PCA to reach at least 95% of the ideal accuracy. We also limit the number of iterations to 10.

We make three observations on the results in Table 2. First, sPCA outperforms the other two algorithms by wide margins in most of the cases. For example, on the MapReduce platform, for a large dataset of tweets of size $1.26B \times 71.5K$, sPCA-MapReduce finishes in less than 5 hours ($16,200$ sec), while Mahout-PCA takes almost 5 days ($430,200$ sec) to finish. The second observation is that MLlib-PCA fails to compute the principal components for high dimensional datasets. This is because MLlib-PCA requires storing a $D \times D$ covariance matrix in the memory of one machine, and hence the algorithm fails when the size of this matrix exceeds the available memory of one machine (not the aggregate memory in the cluster). In our experiments with 32 GB memory machines, MLlib-PCA fails when D exceeds 6,000. Even in the cases that MLlib-PCA succeeds to produce results, it can take about twice the time of our sPCA-Spark. The third observation is that MLlib-PCA outperfroms other approaches in the specific case of a low-dimensional and dense matrix such as the Images dataset. The Images dataset has relatively low dimensionality (128 dimensions). In this case, MLlib-PCA computes a 128×128 intermediate matrix and then does further computations on one machine, so it finishes faster than other approaches, including sPCA-Spark. It is not a problem for such datasets that MLlib-PCA does not leverage sparsity (since the matrix is dense) and fails when the dimensionality is high (since the dimensionality is low).

To summarize, our results show that sPCA offers much better scalability and performance than its competitors on both MapReduce and Spark.

5.2 Detailed Evaluation on MapReduce

In this section, we present an in-depth comparison of sPCA-MapReduce and Mahout-PCA.

Accuracy. The accuracy of both sPCA-MapReduce and Mahout-PCA algorithms depends on the number of iterations that they run. We run both algorithms on different datasets and measure the accuracy after each iteration. Two sample results are shown in Figures 4 and 5, for the *Bio-Text* and *Tweets* datasets, respectively. Figure 4 shows that sPCA reaches 93% accuracy after 715 sec, in the second iteration. In addition, sPCA converges fast, in less than 1,500 sec. Mahout-PCA takes much longer to converge: more than 5,000 sec.

Figure 5 reports the accuracy for the larger *Tweets* dataset. The figure shows that sPCA achieves much higher accuracy than Mahout-PCA. For example, at time 1,000 sec, the accuracy of

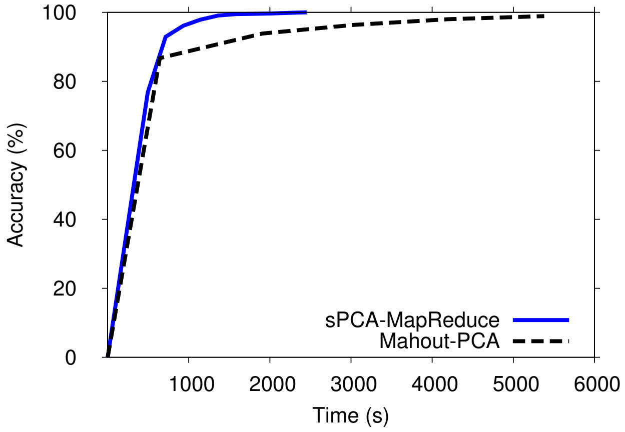

Figure 4: Accuracy vs. time on the *Bio-Text* dataset.

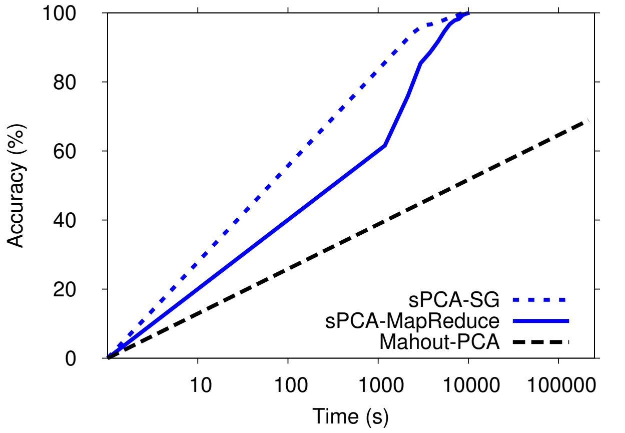

Figure 5: Accuracy vs. time on the *Tweets* dataset. The x axis is in log scale.

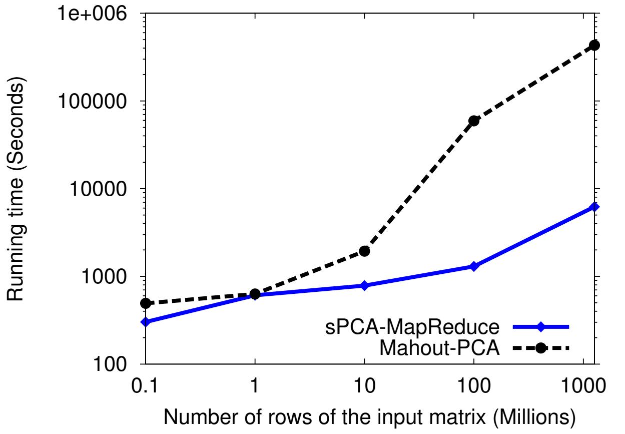

Figure 6: Time to reach 95% of the ideal accuracy on the *Tweets* dataset. We vary the number of rows (tweets) in each experiment. The x and y axes are in log scale.

sPCA is at least 20% higher than that of Mahout-PCA, and the accuracy gap keeps increasing with time. sPCA achieves almost 100% accuracy before 10,000 sec, whereas the accuracy of Mahout-PCA reaches up to 70% after more than 259,000 sec of running. That is, after running for about 26 times longer, Mahout-PCA achieves up to 30% less accuracy than sPCA.

sPCA starts with random initialization for the variance (*ss*) and the principal components (*C*), and improves upon them at each iteration. If we feed the algorithm with *smart guesses* for these variables, it converges faster. We use this optimization by first running the algorithm on a much smaller sample matrix, randomly selected from the original input. We then feed the resulting *ss* and *C* variables to the algorithm to be run on the original dataset. This operation is quite useful for processing large data sets as the time to compute the smart guesses is offset by the much larger savings in time to reach the target accuracy. We refer to sPCA by sPCA-SG when this initialization process is used. The results on the *Tweets* dataset indicate that this initialization technique adds 527 sec of delay. However, as shown in Figure 5, the technique produces a much higher accuracy compared to that of without the optimization. We note that Mahout-PCA cannot use this optimization because Mahout-PCA requires a large random matrix that has the same number of rows (1.26 billion) as the input matrix. In contrast, in sPCA a small $D \times d$ random matrix is initialized which does not depend on the number of rows N, so sPCA can be run easily on a small number of rows before running on the whole input matrix.

Time to Achieve Target Accuracy. We compare sPCA-MapReduce and Mahout-PCA based on the time needed to reach 95% of the ideal accuracy. We vary the size of the input dataset and measure the time needed for both sPCA-MapReduce and Mahout-PCA to achieve 95% accuracy. A sample of our results is shown in Figure 6 for the *Tweets* dataset. Other results are similar. In this figure, we vary the number of rows in the input matrix, but we use the same number of columns, namely the full 71,503 columns of the dataset. The results in Figure 6 show that the running times for both algorithms are close for small datasets (i.e., up to 10 million rows). For larger datasets, however, sPCA-MapReduce reaches 95% accuracy two orders of magnitude faster than Mahout-PCA. The reason for this is that the benefits of our optimizations in sPCA pay off better when we scale to larger datasets. More importantly, unlike Mahout-PCA, the running time of sPCA-MapReduce increases at a much smaller rate as the size of the input dataset increases, which allows it to scale well.

Intermediate Data Size. We measure the size of intermediate data generated by sPCA-MapReduce and Mahout-PCA. Our results (figures not shown due to space limitations) show that sPCA-MapReduce generates much smaller intermediate data in all cases compared to Mahout-PCA. For example, for the *Bio-Text* dataset, Mahout-PCA generates 8 GB of intermediate data, whereas sPCA-MapReduce generates only 240 MB, a factor of 35x reduction. This property pays off even more when we scale to larger datasets. Our results show that for the *Tweets* dataset, Mahout-PCA generates 961 GB of intermediate data, whereas sPCA-MapReduce produces 131 MB of such data, a factor of 3,511x reduction. Notice that sPCA-MapReduce generates less intermediate data as a fraction of the dataset size for the larger *Tweets* dataset since it has fewer columns compared to the *Bio-Text* dataset.

Analysis of sPCA and Mahout-PCA Jobs. We analyze the individual jobs of sPCA and Mahout-PCA in terms of running time of each job and the amount of data generated. This analysis helps in understanding the performance differences observed in the previous sections. Our analysis shows the following: For sPCA, we notice that although the *Tweets* dataset is larger than the *Bio-Text* dataset by a factor of 20x, the durations of the jobs increase by a factor less than 4. This is because the overheads of the Hadoop framework and job initialization have a larger relative impact in the smaller case. More importantly, the execution time depends also on the sparsity of the matrix. Although the *Tweets* dataset is 20x larger in size, it is much sparser.

On the other hand, Mahout-PCA's jobs are significantly slower in relative terms when the input size increases. For example, the execution time of the job in Mahout-PCA (*Bt* job) corresponding to our YtX job increases by a factor of 654x when we increase the data size 20x by switching from the *Bio-Text* to the *Tweets* dataset. Most of this time is spent in the mappers. To understand why Mahout-PCA suffers from this inefficiency, we looked at the mappers' output data and we observed that the mappers produce 15.6x more output for the *Tweets* dataset than for *Bio-Text*, resulting in 4 terabytes of data. The combiners, therefore, are overloaded with a large amount of input. This mapper output size is extremely large compared to the aggregate output size of the mappers of the YtX job in sPCA, which increases by only 2.3x times when we switch from *Bio-Text* to *Tweets*. This moderate mapper output size contributes to the scalability of sPCA.

5.3 Detailed Evaluation on Spark

In this section, we compare sPCA-Spark with MLlib-PCA. To the best our knowledge, MLlib-PCA is the only available Spark implementation of PCA that was added starting from Spark version 1.0.0, which is the version we use in our experiments.

Time to Achieve Target Accuracy. As described in Section 2.1, MLlib-PCA is a deterministic algorithm that terminates after performing a fixed number of matrix operations. We compare the running time of MLlib-PCA with that of sPCA-Spark based on the time needed for sPCA-Spark to reach at least 95% of the ideal accuracy. We run multiple experiments using the *Tweets* dataset. In each experiment, we use the same number of rows, but we vary the number of columns, and we measure the total running time for both algorithms. We plot the results in Figure 7. The results show that MLlib-PCA fails when the number of columns D exceeds 6,000. As discussed before, this is due to the fact that MLlib-PCA loads a $D \times D$ covariance matrix in the memory of one machine. Hence, the algorithm is not scalable except up to a few thousand columns. On the other hand, sPCA-Spark requires a small $O(D \times d)$ matrix to be stored in memory, and d is typically a small constant. This important difference makes sPCA-Spark much more scalable than MLlib-sPCA.

Regarding the running time, the figure shows that sPCA is much faster than MLlib-PCA. For example, the running time of sPCA is nearly half of MLlib-PCA for $D = 6,000$ and the difference in speed increases with increasing the number of columns. This happens because MLlib-PCA performs dense matrix operations on the covariance matrix. Since, the covariance matrix has D^2 elements, the running time of MLlib-PCA increases quadratically with D, unlike sPCA in which there is a linear relationship between the running time and the number of input dimensions D.

Finally, we observe that the running time of sPCA-Spark does not increase with the same factor as the input size. For example, the running time increases by a factor of 10x with increasing the input size by a factor of 70x. This gain is due to the efficient use of sparse matrices in sPCA.

Intermediate Data Size. Intermediate data in Spark can be created in different ways. It could be (i) RDDs distributed in the memory or the disks of different machines in the cluster, or (ii) intermediate data loaded in the memory of the master machine which is the machine that runs the driver program and handles the workflow of the Spark jobs (launching distributed jobs, aggregating results, etc.). Since both sPCA-Spark and MLlib-PCA cache only one RDD in the aggregate memory of the cluster and this RDD is used for the input matrix, other intermediate data is loaded in the driver program. We therefore measure the amount of memory consumed by the process that runs the driver program for both algorithms.

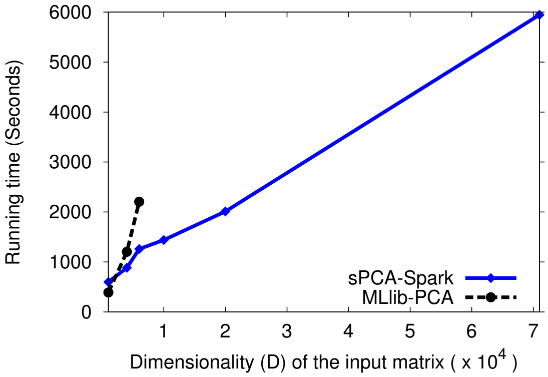

Figure 7: Time to reach 95% of the ideal accuracy on the *Tweets* dataset. We vary the number of columns in each experiment.

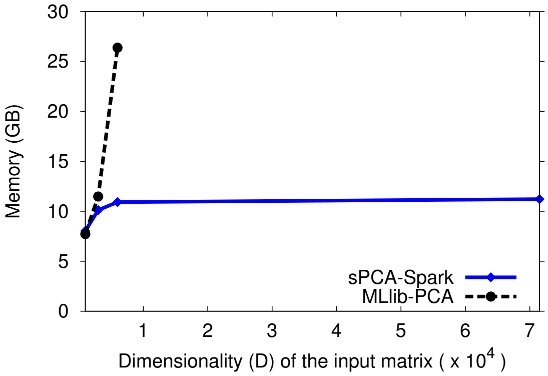

Figure 8: Memory consumption of sPCA-Spark vs. MLLib-PCA on different input sizes for the *Tweets* dataset.

We monitor the memory used by the Java process that runs the driver program in intervals of 5 seconds using the JVM utilities *jmap* and *jstat* and we report the maximum resident memory throughout the running time of the process. Figure 8 compares the memory consumption of both algorithms for the *Tweets* dataset. The results show that the memory consumption of sPCA is almost constant. However, the memory consumption of MLlib-PCA increases drastically with increasing the number of columns. For example, MLlib-PCA consumes more than 26 GB of memory for an input matrix of 6,000 columns. This explains the results shown in Figure 7 and shows why MLlib-PCA fails to process more than 6,000 columns on a machine with 32 GB of memory.

5.4 Effect of Individual Optimizations

In Section 3, we presented our design as successive optimization ideas. Then, we put all the optimizations together to form the final sPCA algorithm. In this section, we analyze how much each of these optimizations contributes to the speedup achieved by sPCA. We analyze the three core optimization ideas: mean propagation to

	Mean Prop.	Intermed. Data	Frobenius
W/ Opt.	2	3	0.4
W/O Opt.	5,400	2,640	102

Table 3: Running time (in sec) for the three main distributed operations with and without the proposed optimizations.

	16 cores	32 cores	64 cores
Running Time	22,680	11,640	5,940
Speedup	1	1.95	3.82

Table 4: Running time (in sec) and speedup of running sPCA on clusters of different sizes.

leverage sparsity (Section 3.1), minimizing intermediate data (Section 3.2), and optimizing the computation of the Frobenius norm (Section 3.4). These optimizations are used in the operations on lines 7, 8, and 13 of Algorithm 1, respectively. Each operation corresponds to one optimization and they are all distributed operations. Therefore, we use these operations to test the optimizations by comparing the optimized and unoptimized versions of the operations.

We use a subset of the *Tweets* dataset consisting of 100,000 rows, and we measure the running time of each operation with and without applying the optimization. The experiments are done using sPCA-Spark and the results are shown in Table 3. The results show that the careful design and optimization ideas of sPCA provide us with orders of magnitude speedup over the unoptimized implementation. The results also show that mean propagation is the optimization that provides the biggest benefit out of the three optimizations. This is because it preserves the sparsity of the input matrix, which has a major effect on performance. The second most important optimization is minimizing the intermediate data. The results show that it takes 3 seconds to compute matrices X and XtX from the input matrix Y compared to 44 minutes needed to compute matrix XtX from the stored large matrix X. The Frobenius norm optimization in Algorithm 3 is faster than the simple implementation in Algorithm 2 by a factor of 270x.

5.5 Speedup

In this section, we analyze the performance of sPCA-Spark on the *Tweets* dataset when running it on clusters of different sizes (16, 32, and 64 cores, corresponding to 2, 4, and 8 nodes). Table 4 shows the running time and speedup with increasing the number of cores. We measure the speedup as $S = T_{16\ cores}/T_{n\ cores}$, where $T_{16\ cores}$ is the running time of sPCA-Spark on the smallest cluster (16 cores), and $T_{n\ cores}$ is the running time of sPCA on a cluster with n cores. The results in Table 4 show that the careful design of sPCA in addition to using Spark, which reduces the communication overhead, result in a linear, almost-ideal speedup (i.e., a low distributed systems penalty).

6. RELATED WORK

Implementing efficient machine learning algorithms on big data is an active field of research. Many ongoing works approach the problem from different perspectives. Several recent works [8, 16] attempt to leverage the observation that the iterative nature of machine learning algorithms does not perfectly match the MapReduce framework. Such works usually (i) add language support to enable the developer to express the iterations, and (ii) provide compiler support to leverage the knowledge about iterations for better scheduling and caching policies. HaLoop [8] extends the MapReduce programming model with the notion of iteration. The knowledge of iteration is then taken into consideration to affect scheduling (running on local data obtained from the previous iteration) and caching policies (caching the output if it is going to be used in the next iteration). Twister [16] suggests modifications to the MapReduce framework to make it efficient for machine learning algorithms. A different approach to iterative machine learning is adopted by Hogwild! [28]. Hogwild! parallelizes the stochastic gradient descent (SGD) algorithm on a shared memory machine by running SGD without locks. Interestingly, convergence is still guaranteed. SGD is useful for many machine learning tasks, but it cannot be used to compute PCA.

Some related works take a top-down approach and define the minimum language requirements to express machine learning algorithms on top of distributed systems [7, 24]. Borkar et al. [7] use DataLog to define a language expressive enough to cover many of the existing machine learning algorithms. They argue that the general query optimization techniques in the database literature could be applied to compile the declarative programs into efficient executions plans. MLbase [24] argues for a DBMS approach for machine learning algorithms, in which the algorithm is expressed in an expressive language (similar to SQL) and MLbase takes care of optimization and query planning. A similar approach is adopted by SystemML [18], in which the user expresses a computation in a language similar to R, and the system automatically compiles the computation to an optimized workflow of MapReduce jobs. The SciDB system [31] focuses on parallel array processing for scientific workloads. The main focus of SciDB is effective storage and retrieval of arrays in cluster environments [30], and a computation like PCA would be an application on top of SciDB.

In this paper, we take a bottom-up approach: we study the bottlenecks in a complex machine learning algorithm and provide solutions for each one. The insights and rules that we presented in this paper could be leveraged by any of the above systems.

Chu et al. [9] list many machine learning algorithms that can be parallelized on multiple cores using MapReduce. For PCA, they suggest using the classic technique of first obtaining the covariance matrix, and then computing its eigenvectors. Then, they show that the covariance matrix can efficiently be computed in the MapReduce model using only one pass on the data. Afterwards, they use a centralized algorithm to obtain the eigenvectors. The disadvantage of this approach is that it requires storing the covariance matrix in the memory of one machine. Although this is possible in the case of “thin” matrices that have a small number of dimensions, it is not a feasible solution for matrices with large dimensionality, which we target. However, we employ their approach for computing the covariance matrix in sPCA when we compute matrix XtX. We presented a comprehensive overview of computing PCA in Section 2.

7. CONCLUSION

In this paper, we analyzed different methods for computing the principal components of an input matrix, which is referred to as principal component analysis (PCA). Our analysis indicated that all current algorithms for PCA have significant computation or communication bottlenecks that prevent them from scaling to large datasets. We presented a scalable design and implementation for PCA, which we call sPCA. sPCA is based on the probabilistic PCA (PPCA) algorithm [32], and it employs several optimizations to support large datasets on distributed clusters. We implemented sPCA on the MapReduce and Spark platforms and showed that it significantly outperforms the closest counterparts on both platforms.

8. REFERENCES

[1] Graphlab: http://graphlab.org/.

[2] Mahout machine learning library: http://mahout.apache.org/.

[3] MLlib machine learning library: https://spark.apache.org/mllib/.

[4] MPI Forum: http://www.mpi-forum.org.

[5] M. W. Berry, S. T. Dumais, and G. W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4), 1995.

[6] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1997.

[7] V. R. Borkar, Y. Bu, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Declarative systems for large-scale machine learning. *IEEE Data Eng. Bull.*, 35(2), 2012.

[8] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow. (PVLDB)*, 3(1-2), 2010.

[9] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Proc. Conf. on Neural Information Processing Systems (NIPS)*, 2006.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[11] J. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Comput*, 11(5), 1990.

[12] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1), 1977.

[13] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[14] C. Ding and X. He. K-means clustering via principal component analysis. In *Proc. Int. Conf. on Machine Learning (ICML)*, 2004.

[15] Q. Du and J. E. Fowler. Hyperspectral image compression using JPEG2000 and principal component analysis. *IEEE Geoscience and Remote Sensing Letters*, 4(2), 2007.

[16] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proc. Int. Symp. on High-Performance Parallel and Distributed Computing (HPDC)*, 2010.

[17] T. Elgamal and M. Hefeeda. Analysis of PCA algorithms in distributed environments. Technical Report arXiv:1503.05214.

[18] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on mapreduce. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2011.

[19] G. Golub and C. E. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5), 1970.

[20] A. N. Gorban, B. Kgl, D. C. Wunsch, and A. Zinovyev. *Principal Manifolds for Data Visualization and Dimension Reduction*. Springer Publishing Company, Incorporated, 2007.

[21] N. P. Halko. *Randomized methods for computing low-rank approximations of matrices*. PhD thesis, University of Colorado, 2012.

[22] V. Hernandez, J. Roman, and A. Tomas. A robust and efficient parallel SVD solver based on restarted Lanczos bidiagonalization. *Electronic Transactions on Numerical Analysis*, 31, 2008.

[23] I. Jolliffe. Principal component analysis. 1986. 1986.

[24] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A distributed machine-learning system. In *Proc. Conf. on Innovative Data Systems Research (CIDR)*, 2013.

[25] D. G. Lowe. Object recognition from local scale-invariant features. In *Proc. Int. Conf. on Computer Vision (ICCV)*, 1999.

[26] M. Nixon and A. S. Aguado. *Feature Extraction & Image Processing*. Academic Press, 2nd edition, 2008.

[27] J. M. Porta, J. J. Verbeek, and B. J. Kröse. Active appearance-based robot localization using stereo vision. *Autonomous Robots*, 18(1), 2005.

[28] B. Recht, C. Re, S. J. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proc. Conf. on Neural Information Processing Systems (NIPS)*, 2011.

[29] L. I. Smith. A tutorial on principal components analysis. *Cornell University*, 2002.

[30] E. Soroush, M. Balazinska, and D. L. Wang. ArrayStore: A storage manager for complex parallel array processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2011.

[31] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of SciDB. In *Proc. Scientific and Statistical Database Management Int. Conf. (SSDBM)*, 2011.

[32] M. E. Tipping and C. M. Bishop. Mixtures of probabilistic principal component analysers. *Neural Computation*, 11(2), 1999.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX Conf. on Networked Systems Design and Implementation (NSDI)*, NSDI'12. USENIX Association, 2012.

[34] H. Zou, T. Hastie, and R. Tibshirani. Sparse principal component analysis. *Journal of computational and graphical statistics*, 15(2), 2006.