

Scalable Machine Learning Computing a Data Summarization Matrix with a Parallel Array DBMS

Carlos Ordonez · Yiqun Zhang ·
S. Lennart Johnsson

the date of receipt and acceptance should be inserted later

Abstract Big data analytics requires scalable (beyond RAM limits) and highly parallel (exploiting many CPU cores) processing of machine learning models, which in general involve heavy matrix manipulation. Array DBMSs represent a promising system to manipulate large matrices. With that motivation in mind, we present a high performance system exploiting a parallel array DBMS to evaluate a general, but compact, matrix summarization that benefits many machine learning models. We focus on two representative models: linear regression (supervised) and PCA (unsupervised). Our approach combines data summarization inside the parallel DBMS with further model computation in a mathematical language (e.g. R). We introduce a 2-phase algorithm which first computes a general data summary in parallel and then evaluates matrix equations with reduced intermediate matrices in main memory on one node. We present theory results characterizing speedup and time/space complexity. From a parallel data system perspective, we consider scale-up and scale-out in a shared-nothing architecture. In contrast to most big data analytic systems, our system is based on array operators programmed in C++, working directly on the Unix file system instead of Java or Scala running on HDFS mounted on top of Unix, resulting in much faster processing. Experiments compare our system with Spark (parallel) and R (single machine), showing orders of magnitude time improvement. We present parallel benchmarks varying number of threads and processing nodes. Our 2-phase approach should motivate analysts to exploit a parallel array DBMS for matrix summarization.

1 Introduction

As big data keeps growing, business and scientific goals become more challenging and organizations want to save money in infrastructure they push the need

to computing machine learning models in large parallel clusters. Cloud computing [1] is an established technology in big data analytics given its ability to scale out to a large number of processing nodes, providing ample RAM and practically unlimited disk storage. It can provide an affordable, yet powerful computing infrastructure with an easy-to-use software service. From the software side, there exists a long-time existing gap between mathematical packages and large-scale data processing platforms. On one hand, mathematical systems like R, Matlab and more recently Python, provide comprehensive libraries for machine learning and statistical computation, but none of them are designed to scale to large data sets that exceed even a single machine’s memory. On the other hand, parallel DBMSs (e.g. Teradata, Oracle, Vertica) and “big data” Hadoop systems (e.g. MapReduce [7], Spark [27]) are two prominent technologies that offer ample storage and parallel processing capabilities. However, even though there is significant research progress on data mining algorithms [13], computing machine learning models on both parallel systems remains a challenging problem. In the DBMS world UDFs have been shown a great mechanism to accelerate machine learning algorithms [4, 19], at the expense of customizing the computation to a specific DBMS. On the other hand, it is fair to say that R remains one of the important languages for machine learning computations [14]. Previous research attempted integrating R with Hadoop as well as DBMSs (e.g. DB2 Ricardo [6], SQL Server R suite and Vertica predictive analytics, among others). A tight integration takes advantage of R’s powerful mathematical capabilities and on the other hand, it leverages large-scale data processing capabilities from Hadoop and parallel DBMSs. Some of the examples from the MapReduce world are Apache Mahout, MLBase and Spark. All of them lack data management capabilities and optimizations available on a DBMS. At the same time, DBMSs like SQL Server, Vertica, Teradata, Netezza and Greenplum have incorporated machine learning algorithms inside their systems. Even though those extensions enable computations to perform in-DBMS analytics, they do not work well in parallel for iterative algorithms. That is, both worlds are getting closer. However, scalable parallel matrix computations remain difficult to evaluate inside a parallel DBMS. Our goal is to show a parallel DBMS can indeed help in a demanding matrix summarization common to several machine learning models.

Database systems research has emphasized that DBMSs with alternative storage mechanisms [16, 24, 26] can achieve orders of magnitude in performance improvement. SciDB [25] is an innovative parallel DBMS with array storage, capable of managing unlimited size matrices (i.e. as large as space on secondary storage) and well-integrated with R (with powerful mathematical capabilities). Such scalable architecture opens research possibilities to scale machine learning algorithms with parallel database-oriented processing and matrix sizes exceeding RAM capacity.

In this work, we explore how to leverage a parallel array DBMS (SciDB) capabilities to accelerate and scale the computation of fundamental machine learning models in a parallel cluster of computers. Our main research contributions include the following:

1. Parallel data loading into 2-dimensional arrays (i.e. matrices).
2. Parallel data summarization in one pass returning a comprehensive summarization matrix, called Γ .
3. Pushing vector-vector outer product computation to RAM, resulting in a fully parallel matrix multiplication, with no data transfer, just one parallel barrier, and minimal overhead.
4. Further acceleration of parallel summarization with multi-threaded processing or a GPU exploiting many processing units (cores).

The rest of the paper is organized as follows: Section 2 introduces mathematical definitions. Section 3 presents a parallel algorithm and analyzes its time complexity and speedup. Section 4 explains the technical details of our system. Section 5 presents a benchmark comparing our system with competing systems, including Spark, from the Big Data Hadoop world and R, a well-known statistical system. Related work is compared in Section 6. Finally, Section 7 summarizes conclusions and identifies directions for future work.

2 Definitions

2.1 Input Data Set and Output Model

This section can be skipped if the reader is familiar with machine learning models. The input for our machine learning models is a $d \times n$ matrix X . In mathematical terms, let $X = \{x_1, x_2, \dots, x_n\}$ be the input data set with n points, where each point x_i is a *column vector* in \mathbf{R}^d . The goal is to compute a machine learning model that we call Θ , minimizing some error or penalization metric (e.g. squared error).

Our system allows the computation of a wide range of linear (fundamental) models including principal component analysis (PCA), linear regression (LR), variable selection (VS), Naïve Bayes classifier, K-means (and EM) clustering, logistic regression and linear discriminant analysis. Those models involve many matrix computations, which are a good fit for an array DBMS. All the models we support can derive their computations from the data summarization matrix Γ . However, non-linear models such as SVMs, decision trees and deep neural networks are not currently supported, but we will study how parallel summarization can accelerate their computation. In this work we focus on PCA and LR. In PCA we take X "as is" with d dimensions. On the other hand, in linear regression (LR) X is augmented with an extra $(d + 1)$ th dimension with the output variable Y , resulting in a \mathbf{X} augmented $(d + 1) \times n$ matrix. We use $i = 1 \dots n$ and $j = 1 \dots d$ (or $j = 1 \dots d + 1$ if Y exists) as matrix subscripts. Notice that in order to simplify mathematical notation and present equations in a more intuitive form we use column vectors and column-oriented matrices.

2.2 Parallel DBMS

From a general parallel computing point of view, our algorithms assume a computer cluster system with distributed memory and disk, interconnected by a fast network. Let P be the number of processing nodes, where each node has its own memory and disk storage (in some diagrams and plots we use N instead of P). Nodes communicate with each other via message passing sending data. There is a master (central) lightweight process running on one of the working nodes, responsible for coordinating computations.

We now explain the parallel DBMS architecture in more depth. The parallel DBMS works on a parallel cluster of computer (nodes), under a fully distributed memory and disk architecture (i.e. shared-nothing) [11]. That is, each node has its own main memory and secondary storage and it cannot directly access another node storage. Therefore, all processing nodes communicate with each other transferring data. Before launching the parallel computation, executable operator code is replicated across all nodes. All physical operators automatically work in parallel on partitions of the data. All the nodes are called “worker” nodes and one of them has the additional role of serving as the coordinating (master) node. Notice there is no separate “master” node, unlike other parallel systems. This is due to the fact that the coordination workload is much lower than the worker CPU and I/O tasks and because it is easier to maintain multiple instances of the coordinating process on multiple nodes. It is assumed CPU and I/O processing is done locally and data transfer is avoided whenever possible [12]. When data transfer happens there are two major possibilities: (1) There is a large amount of data, which forces a data redistribution across the cluster. (2) There is a small amount of data that is transferred or copied to one node or a few nodes. In general, all workers send small amounts of data to the master node. This parallel architecture is independent from the storage mechanism. Currently, parallel DBMSs support the following storage mechanisms: by row, by column and as multidimensional arrays. In this paper, we focus on 2D arrays, which represent a perfect fit to store and read matrices..

SciDB, like most parallel SQL engines has a shared-nothing architecture. SciDB stores data as multi-dimensional arrays in chunks, instead of rows compared to traditional DBMSs, where each chunk is in effect a small array that is used as the I/O unit. SciDB preserves the most important features from traditional DBMSs like external algorithms, efficient I/O, concurrency control and parallel processing. From a mathematical perspective, SciDB offers a rich library with matrix operators and linear algebra functions. From a systems programming angle, SciDB allows the development of new array operators extending its basic set of operators. In our case, we extend SciDB with a powerful summarization operator.

3 Parallel Computation

In this section we develop theory that can be applied on large-scale parallel systems, including parallel DBMSs and “Big Data” systems running on HDFS. These theory results need further study on HPC systems (e.g. using MPI), where secondary storage is not necessarily shared-nothing.

3.1 Data Summarization Matrix

A fundamental optimization in our system is the data summarization matrix Γ , which can help substituting the data set in many intermediate machine learning model computations.

The Γ Matrix

As mentioned in Section 2.1, supervised models, like linear regression, use an *augmented* matrix \mathbf{X} (in bold face to distinguish it from X). Based on this augmented matrix, we introduce the augmented matrix \mathbf{Z} , created by appending \mathbf{X} with a row n -vector of 1s. Since X is $d \times n$, \mathbf{Z} $(d+2) \times n$, where the first row corresponds to a constant dimension with 1s and the last row corresponds to Y . Each column of \mathbf{Z} is a $(d+2)$ -dimensional vector z_i . Since PCA is an unsupervised (descriptive) model, we eliminate the last dimension, resulting in \mathbf{Z} being $(d+1) \times n$.

The Γ matrix is a generalization of scattered sufficient statistics [19] assembling them together in a single matrix and it is defined as:

$$\begin{aligned} \Gamma &= \begin{bmatrix} n & L^T & \mathbf{1}^T \cdot Y \\ L & Q & XY^T \\ Y \cdot \mathbf{1} & YX^T & YY^T \end{bmatrix} \\ &= \begin{bmatrix} n & \sum x_i^T & \sum y_i \\ \sum x_i & \sum x_i x_i^T & \sum x_i y_i \\ \sum y_i & \sum y_i x_i^T & \sum y_i^2 \end{bmatrix} \end{aligned}$$

By using the z_i vector introduced above, we now come to an important equation. Matrix Γ can be equivalently computed in two equivalent forms as follows:

$$\Gamma = \mathbf{Z} \cdot \mathbf{Z}^T \tag{1}$$

$$= \sum_{i=1}^n z_i \cdot z_i^T. \tag{2}$$

Each form has profound performance and parallel processing implications. The 1st form is based on matrix multiplication, whereas the 2nd form is a sum

of vector outer products. Both forms have different implications with respect to memory footprint, scalability and speed.

Matrix Γ contains a comprehensive, accurate and sufficient summary of X to efficiently compute the models defined above. PCA and LR require one Γ summarization matrix (non-diagonal),

PCA is derived by factorizing the variance-covariance matrix (simply called covariance matrix) or the correlation matrix with a symmetric singular value decomposition (SVD). The covariance matrix can be computed as

$$V = \frac{1}{n} \sum_i (x_i - \mu)(x_i - \mu)^T = Q/n - (L/n)(L/n)^T. \quad (3)$$

Intuitively, the covariance matrix centers the data set at the origin and measures how variables vary with respect to each other, in their original scales. Based on the covariance matrix, the correlation matrix is derived by computing pair-wise correlations ρ_{ab} with the more efficient form:

$$\rho_{ab} = \frac{V_{ab}}{\sigma_a \sigma_b} = (nQ_{ab} - L_a L_b) / (\sqrt{nQ_{aa} - L_a^2} \sqrt{nQ_{bb} - L_b^2}), \quad (4)$$

where σ_a, σ_b are the corresponding standard deviations. Intuitively, the correlation matrix is a normalized covariance matrix [14]. Then SVD factorization is solved as

$$V = U E U^T, \quad (5)$$

$$\rho = U E^T, \quad (6)$$

where U has the eigenvectors and E the squared eigenvalues (commonly called λ_j^2). The principal components are those eigenvectors having eigenvalues $\lambda_j \geq 1$. LR is solved using

$$\beta = (\mathbf{X}\mathbf{X}^T)^{-1} X Y^T = \mathbf{Q}^{-1} X Y^T, \quad (7)$$

via least squares optimization, where \mathbf{Q} is an augmented matrix which includes n, L, Q as submatrices.

3.2 Algorithm to Compute a Model on Large Data Sets

Our algorithm is designed to analyze large data sets on a modern parallel cluster. Recall from Section 3 P is the number of processing nodes (machines, processors). Our most important assumption is formalized as follows:

Assumption: We assume $d \ll n$ and $P \ll n$. But for d either possibility is acceptable: $d < P$ or $P > d$.

The methods to compute the machine learning model Θ are changed, with the following 2-phase algorithm:

- Phase 1: parallel across the P nodes. Compute Γ in parallel, reading X from secondary storage, building z_i in RAM, updating Γ in RAM with the sum of $z_i \cdot z_i^T$ vector outer products. This phase will return P local summarization matrices: $\Gamma^{[I]}$ for $I = 1 \dots P$.
- Phase 2: at master node, parallel within 1 node: Iterate method until convergence, exploiting Γ in main memory in intermediate matrix computations to compute Θ .

Phase 1 involves a parallel scan on X , where each partition is processed independently. Since Γ significantly reduces the size of matrix X from $O(dn)$ to the summarized representation taking space $O(d^2)$ Phase 2 can work on one node (i.e. the Coordinator/Master node) and Phase 2 can work only in main memory, without I/O operations on X . However, Phase 2 can still work in parallel in RAM with a multicore CPU (e.g. multi-threaded processing, vectorized operations) or with a GPU (with a large number of cores sharing memory) as we shall see below.

During Phase 1 each worker node I computes $\Gamma^{[I]}$ in parallel without exchanging messages with other workers. When all workers finish the coordinator node gathers worker results to compute the global summarization matrix:

$$\Gamma = \Gamma^{[1]} + \dots + \Gamma^{[P]},$$

having just $O(d^2)$ network communication overhead per processor to send $\Gamma^{[I]}$ to the master node. That is, there is a fully parallel computation for large n , coming from the fact that each $z_i \cdot z_i$ is evaluated independently on each processor. Given the additive properties of Γ the same algorithm is applied on each node $I = 1 \dots P$ to get $\Gamma^{[I]}$, combining all partial results $\Gamma = \sum_I \Gamma^{[I]}$ in the coordinator node.

3.3 Parallel Speedup and Communication Efficiency

We present an analysis on parallel speedup assuming P processors on the shared-nothing parallel architecture introduced in Section 2. Assuming X is a large matrix it is reasonable to assume $d < n$ and $n \rightarrow \infty$.

The first aspect to consider is partitioning the matrix for parallel processing; this is done when the data set is loaded into the parallel DBMS. As it is standard in parallel DBMSs, when the matrix is loaded each x_i in X is hashed to one of the P processors based on some hash function $h(i)$ applied to the point primary key: hashing by point id i . In this manner, the input matrix X will be evenly partitioned in a random fashion. We must mention that round-robin assignment could also work because values of i are known beforehand. Then each outer product $z_i \cdot z_i^T$ is computed in parallel in RAM, resulting in $O(d^2n/P)$ work per processor for a dense matrix and $O(dn/P)$ work per processor for a sparse matrix. Both dense and sparse matrix multiplication algorithms become more efficient as P increases or $n \rightarrow \infty$. We present the following speedup statement as a theorem given its importance for parallel processing:

Definition 1 Speedup: Let T_P be the processing time to compute Γ using P nodes, where $1 \leq P$. Parallel speedup s_P using P nodes is defined as $s_P = T_1/T_P$, where T_1 corresponds to sequential processing and T_P corresponds to the maximum processing time among the P nodes when they work in parallel (i.e. slowest node is a bottleneck).

Theorem 1 *Linear speedup: Assuming that $d \ll n$ and that Γ fits in main memory then our parallel algorithm gets increasingly closer to optimal speedup $s_P = T_1/T_P \approx O(P)$.*

Proof: For one node time is $T_1 = d^2n/2 = O(d^2n)$. Since n is large we can evenly partition it into P parts with n/P points. Therefore, the total computation time for P parallel processors is $T_P = (d^2/2) \cdot (n/P) + 2 \cdot d^2P$, where the first term corresponds to the parallel computation of Γ and the second term corresponds to sending P partial results to the coordinator (master) node and adding the P partial Γ s. Therefore, the total time is $T_P = (d^2/2)(n/P) + 2d^2P = O(d^2n + d^2P) = O(d^2(n + P))$. Since we assume $P \ll n$ then $n + P \approx n$. Therefore, computation of partial matrices dominates cost resulting in $T_P \approx O(d^2n/P)$ for a dense matrix and then $T_1/T_P \approx O(d^2n)/O(d^2n/P) = O(P)$. Analog reasoning shows the speedup is also $O(P)$ for sparse matrices: $T_1/T_P \approx O(dn)/O(dn/P) = O(P)$. \square

For completeness, we provide an analysis of a parallel system where P is large or n is small relative to P . That is, we assume $P = O(n)$. We must point out this scenario is highly unlikely with a parallel DBMS where P is tens or a few hundreds at most.

Lemma 1 *Communication bottleneck: Assume only one node can receive partial summarizations. Then time complexity with P messages to a single coordinator (master) node is $O(d^2n/P + Pd^2)$.*

Proof: Straightforward, based on the fact that the coordinator requires serialization from the P workers, becoming a sequential bottleneck. \square

Lemma 2 *Improved global summarization with balanced tree: Let P be the number of nodes (processing units). Assume partial results $\Gamma^{[I]}$ can be sent to other workers in a hierarchical (tree) fashion. Then $T(d, n, P) \geq O(d^2P + \log_2(P)d^2)$ (a lower bound for time complexity with P messages).*

Proof: We assume workers send messages to other workers in synchrony. 1: If $I \bmod 2 = 1$ then send $\Gamma^{[I+1]}$ to node I , requiring $(P/2)$ messages. 2: If $I \bmod 2^2 = 1$ then send $\Gamma^{[I+2^1]}$ to node I , requiring $(P/2^2)$ messages. and so on. Intuitively, we divide by 2 the number of messages at every communication round. Therefore, the total number of messages is $P/2 + P/4 + \dots + 1 \approx P = O(P)$. In summary, we just need synchronous processing for $\log_2(P)$ communication rounds which end when we get the global Γ . \square

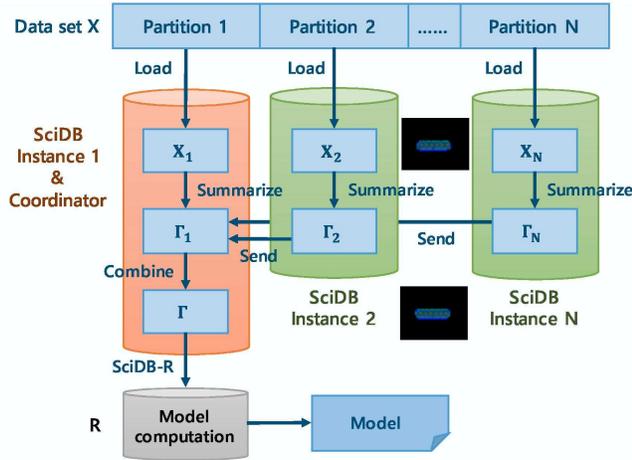


Fig. 1 Diagram of parallel system components (1 SciDB instance/node, N nodes).

4 Computing Machine Learning Models Pushing Summarization into a Parallel Array DBMS

4.1 System Architecture

The parallel computation is formally defined in Section 3. Here we provide an overview of the parallel DBMS from a systems perspective. Figure 1 illustrates our major system components in an N -node parallel cluster showing how data points in X are transferred from node to node (notice we use N instead of P in the diagram). In order to simplify the diagram each node in the cluster runs only one SciDB instance, but we want to point out it is feasible to spawn one thread per core (i.e. intra-node parallelism).

The data set file is first loaded into the DBMS. After the cluster has been set up, SciDB loads the data set into a two-dimensional array in parallel. The input matrix thereafter goes through the two-phase algorithm introduced in Section 3.2:

1. Parallel data summarization to get matrix Γ ;
2. Model Θ computation in RAM in the R language, iterating the numerical method until convergence.

During the data summarization phase the Γ matrix is computed in parallel using a novel user-defined array operator in the parallel DBMS (SciDB). As discussed above, the Γ matrix is much smaller than the original data set X . Therefore, it can easily fit in the main memory substituting X in several intermediate computations. We compute the model Θ exploiting the Γ matrix in R, using the SciDB-R library.

Since R is responsible for the computation of the mathematical model on one node it runs on the coordinator node. That is, it is unnecessary to run

R in every node. The R language runtime introduces a sequential processing bottleneck because it is single threaded given its functional programming roots. However, the time complexity of the R computation depends only on d , not on n . To solve SVD for PCA and least squares for LR time complexity in R is $O(d^3)$. We should emphasize that we use the R language to make adoption of our research prototype easier, but it is feasible to use some other host language like C++, Java or Python which allow faster multi-threaded processing.

4.2 Parallel Data Loading

Since an array DBMS is a transient platform to perform matrix computations it is necessary to provide a fast data loading mechanism. This is especially important to be competitive with Hadoop systems, in which loading generally just involves copying files to HDFS. The SciDB DBMS provides two mechanisms for loading data into arrays: (1) Converting the CSV file into a temporary file with a SciDB-specific array 2D text format for later loading, or (2) Loading the data file into a one-dimensional tuple array first and then go through a re-dimensioning process to transform the one-dimensional array (array of tuples) into a 2-d array (matrix). Since the second mechanism can be used directly with CSV files and does not require any programming it is the default used by most SciDB users. Unfortunately, both mechanisms are slow with a large parallel cluster. Specifically, both loading alternatives require 2 full disk reads and 2 full disk writes. (one full disk read from the CSV file, then one full disk write, using SciDB-specific text format or as a one-dimensional array, then 1 full disk read to read them back, finally, a full disk write into the matrix format). In addition, re-dimensioning requires reshaping d sets of 1D chunks and transforming them into a single set of 2D chunks with a different data partitioning. This implies reading disk blocks in one format, grouping and writing them in a different format and rehashing them to the P nodes. With this major bottleneck in mind, we developed a novel user-defined operator *load2d()* for direct parallel matrix loading into the parallel array DBMS.

We now explain our optimized loading operator in terms of matrix X , as an $n \times d$ array loaded in parallel using P nodes. Let n_B be the number of points stored on one chunk. We chose unspanned storage meaning that each point x_i fits completely inside the chunk. We set the physical chunk size for our matrix in SciDB to $n_B \times d$, where our default setting is $n_B = 10k$ which creates chunks in the range of MBs (ideal for SciDB). Intuitively, we partition the input rectangular matrix $d \times n$ into smaller $d \times n_B$ rectangles preserving the d side. As the first step, the operator computes how many chunks are needed based on n and n_B : $C_{total} = \lfloor \frac{n-1}{n_B} + 1 \rfloor$, and determine how many chunks each node will store: $C_{each} = \lfloor \frac{C_{total}-1}{P} + 1 \rfloor$. As the second step, the coordinator node scans the data file and determines the offset (line number) within the file where each thread will start reading based on C_{each} and n_B . Then the P nodes start reading the file in parallel without locking. We use a standard round-robin algorithm to distribute and write chunks to each node, like SciDB

does. This approach results in a significant acceleration. Our loading operator directly outputs the matrix in SciDB chunk format with optimal chunk size, avoiding any intermediate files on disk and the slow re-dimensioning process. To summarize, our parallel load operator can load significantly faster than the built-in SciDB loading function because: (1) It requires less I/O work: only 2 disk scans to read data and 1 disk scan to write the 2D array. (2) It saves significant CPU time by not re-dimensioning from 1D into 2D. (3) There is no data redistribution (shuffling) among the P nodes, which would add communication and double I/O overhead.

4.3 Parallel Summarization with new Array Operator

As seen in Section 3, the most important property of the Γ matrix is that it can be computed by a single matrix multiplication using \mathbf{Z} . As proven before, $\Gamma = \mathbf{Z}\mathbf{Z}^T$ can be more efficiently computed with this sum form assuming z_i fits in RAM and dynamically building z_i in RAM as well:

$$\Gamma = \sum_{i=1}^n z_i \cdot z_i^T. \quad (8)$$

We consider two major forms of parallelism to compute Γ based on the second expression:

- Inter-node parallelism.
- Intra-node parallelism.

Inter-node parallelism is best for a cloud environment. A benchmark will help understanding the scale-out properties of our system by increasing the number of nodes, assuming network transfer cost is higher than disk to main memory transfer. Intra-node parallelism is best for HPC-style nodes with many CPUs and plenty of RAM. Notice, however, that intra-node parallelism cannot scale well to analyze big data due to I/O cost. We will experimentally analyze the scale-up behavior by increasing the number of threads in a single node, assuming the cost to transfer data from disk to main memory gets amortized by interleaving I/O with CPU operations.

In inter-node parallelism there is network transmission and therefore this cost and overhead has considerable weight. Specifically, it is necessary to consider if network transmission is significantly faster or slower than disk transfer speed. On the other hand, in intra-node parallelism there is no network transmission, but it is necessary to interleave I/O with CPU multi-threaded processing, taking into account potential contention among threads for local resources like main memory (RAM) to consolidate partial results and access to a single disk. From a multicore CPU perspective, one thread per core may not achieve optimal performance given the contention for limited resources and periodic coordination. Therefore, the number of threads should be optimized. In the case of the parallel array DBMS such number is between 1 and the number of cores in the CPU.

4.4 Inter-node parallelism: across the cluster

Our Γ operator works fully in parallel with a partition schema of X into P subsets $X^{[1]} \cup X^{[2]} \cup \dots \cup X^{[P]} = X$, where we independently compute $\Gamma^{[I]}$ on $X^{[I]}$ for each node I (we use this notation to avoid confusion with matrix powers and matrix entries). In SciDB terms, each worker I will independently compute $\Gamma^{[I]}$. No synchronization between instances is needed once the computation starts therefore the parallelism is guaranteed. When all workers are done the coordinator node gathers all results and compute a global $\Gamma = \Gamma^{[1]} + \dots + \Gamma^{[P]}$ with $O(d^2)$ communication overhead per node. Notice $O(d^2) \ll O(dn)$. This is essentially a natural parallel computation, coming from the fact that we can push the actual multiplication of $z_i \cdot z_i^T$ into the array operator. Since this multiplication produces a symmetric matrix the operator computes only the upper or lower triangular submatrix.

4.5 Intra-node parallelism: multiple threads per node with a multicore CPU

From a systems perspective, we assume each array chunk (multidimensional block) is independently processed by one thread. Recall from Section 3.2 that X can be processed fully in parallel without communication during summarization. This fact also applies to a multicore CPU where main memory is partitioned depending on the number of threads. Each thread can keep a local Γ version without locking (i.e. without concurrent updates). Therefore, with a multicore CPU it is possible to increase speed by increasing floating point operations (FLOPS). In theory, if the CPU has Q cores then Q could independently process a partition of X doing I/O and FLOPS without synchronization, but there may exist some contention for local resources (cache, RAM, disk). It is assumed disk I/O is slow, but disk to RAM transfer is fast. A low number of threads, below Q , will leave some cores idle. On the other hand, if the number of threads goes beyond Q this configuration will introduce overhead because a thread will serve multiple partitions introducing many context switches. Once local Γ versions per thread are ready they can be sent directly to the coordinator node. That is, it is unnecessary to follow a 2-phase scheme, by first getting Γ on each node aggregating Γ from all local threads and then computing global Γ across all P nodes.

4.6 Intra-node parallelism: accelerating summarization with a GPU

Computing Γ by evaluating $\mathbf{Z}\mathbf{Z}^T$ is a bad idea because of the cost of transposing \mathbf{Z} row by row and storing \mathbf{Z}^T on disk. Instead, we evaluate the sum of vector outer products $\sum_{i=1}^n z_i \cdot z_i^T$, which is easier to update in parallel in RAM. Moreover, since Γ is symmetric, we only compute the lower triangle of the matrix. Our Γ matrix multiplication algorithm works fully in parallel with a partition of X into P subsets $X^{[1]} \cup X^{[2]} \cup \dots \cup X^{[P]} = X$, where

we independently compute $\Gamma^{[I]}$ on $X^{[I]}$ for each processor I . A fundamental aspect is to optimize computation when \mathbf{X} (and therefore \mathbf{Z}) is sparse: any multiplication by zero returns zero. Specifically, $n_{nz} \leq \sqrt{dn}$ can be used as a threshold to decide between a sparse or dense algorithm, where n_{nz} is the number of non-zero entries in X . In this paper we focus on parallelizing the dense matrix algorithm. On the other hand, computing Γ on sparse matrices with a GPU is challenging because tracking and partitioning non-zero entries reduces parallelism by adding more complex logic in the operator code; we leave this aspect for future work.

Our following discussion focuses on integration with an array DBMS, given its ability to manipulate matrices. From a systems perspective, we integrated our model computation using Γ with the SciDB array DBMS and the R language, with a 2-phase algorithm: (1) data summarization in one pass returning Γ ; (2) exploiting Γ in intermediate computations to compute model Θ . We emphasize z_i is assembled in RAM (we avoid materializing z_i to disk). Unlike standard UDFs in a traditional relational DBMS with SQL, which usually need to serialize a matrix into binary/string format then deserialize it in succeeding steps, our operators in SciDB return Γ directly in 2D array format, which is a big step forward compared to previous approaches. Finally, Phase 2 takes place in R on the master node leveraging R’s rich mathematical operators and functions. Even though Phases 2 does not run in parallel across nodes, it does not have a significant impact on performance because Γ is much smaller than the data set.

Parallel processing happens as follows. In SciDB, arrays are partitioned and stored as chunks and such chunks are only accessible by C++ iterators. In our previous work [21], we compute the vector outer product $z_i \cdot z_i^T$ as we scan the data set in the operator. We emphasize z_i is assembled in RAM (we avoid materializing z_i to disk). In general, interleaving I/O with floating point computation is not good for GPUs because it breaks the SIMD paradigm. In our GPU accelerated operator, we first extract matrix entries in each chunk into main memory. Then we transfer the in-memory subsets of X to GPU memory, one chunk at a time. Then the GPU computes the vector outer products $z_i \cdot z_i^T$ fully in parallel with its massive amount of processing units (cores). The sum is always maintained in the GPU memory. It takes $\log(n)$ time to accumulate n partial Γ s into one using the reduction operation. When the computation finishes for the whole data set, the Γ matrix is transferred back from GPU memory to CPU main memory. The C++ operator code is annotated with OpenACC directives to work with GPU. In our current GPU version, the CPU only does the I/O part. Since the DBMS becomes responsible for only I/O our approach also has promise in relational DBMSs.

5 Experimental Evaluation

We study efficiency and evaluate parallel processing comparing our DBMS-based solution with competing analytic systems: the R language and Spark.

Table 1 Data sets.

Data set	d	n	Description
NetIntrusion	38	4,940,217	Computer network intrusion detection (KDD 99 Cup)
YearSongPred	90	515,345	Predict year of song release (UCI Repository)

We present benchmark results on three systems: one server with a multicore CPU, a large parallel cluster running in the cloud and a server with a GPU (many cores) and a multicore CPU.

5.1 Data Sets

We used two real data sets to measure time complexity and compare different systems, described in Table 1. Both data sets are widely used in machine learning and data mining research, but they are small to evaluate parallel processing. Therefore, we decided to create “realistic” data sets by replicating rows and columns and rows to create challenging data sets. Since the Net-Intrusion data set is large and represents typical big data problems (sparse, coming from a log, predictive attributes) we use it as the base data set to build large matrices.

5.2 Evaluating Performance and Numeric Accuracy with Real Data Sets

We compare the performance of our system and evaluate accuracy with real data sets to compute our default model: linear regression (LR). Given modern servers, with large RAM, available data sets to compute machine learning models fit in RAM across the parallel cluster and they are adequate to test parallel speedup. Therefore, we have opted to compare performance on a single machine with R, a popular analytic language in machine learning and statistics. To provide a more complete comparison we also include Spark running on the same hardware (one node) although we caution Spark is better suited for parallel processing on multiple nodes, which we analyze in the next section.

Hardware and Software: We used one server with a 4-core CPU running at 2.1 GHz, 8 GB RAM and 1 TB disk. The operating system was Linux Ubuntu 14.1. R, SciDB and Spark were installed and tuned for this configuration. Notice the real data sets fit in RAM on all systems. In R we use the `lm()` function to fit a linear model. In R+DBMS we used the `solve()` function following Equation 7. In Spark we used its fastest linear regression function available in MLlib (based on gradient descent), with 20 iterations (the recommended default setting).

To present a complete picture, we evaluate model accuracy obtained by our DBMS+R solution using the R solution as reference: we compute the maximum relative error of β coefficients across all d dimensions taking the R solution as reference. For instance, for the first regression coefficient $\text{err}_1 = |\beta_1^{[DBMS+R]} - \beta_1^{[R]}| / \beta_1^{[R]}$. Table 3 shows such maximum relative error on the

Table 2 Comparing performance to compute linear regression (LR) model on one machine with real data sets; time in secs.

Data Set	d	n	R language	Spark	DBMS+R	rel. error
NetIntrusion	38	5M	44	515	12	< 1%
YearSongPred	90	500k	32	217	13	< 1%

last column: for both data sets accuracy was higher than 99% (floating point numbers match on at least 3 decimals).

Table 3 compares the R language matrix operators with the dense and sparse operators to compute Γ with the NetIntrusion data set, which is a sparse matrix. In the case of R, we use dense matrix multiplication `%*%` because it is the most common used in practice and because it produces an output matrix compatible with the built-in linear regression functions. We emphasize that R reads the input matrix and loads it into RAM before doing any computation. In order to understand the impact of matrix density (i.e. many zeroes) we created two versions of the NetIntrusion data set: for $d = 10$ we selected the 10 sparsest dimensions (highest fraction of zeroes) out of the original 38 dimensions, whereas for $d = 100$ we replicated all dimensions uniformly until reaching high dimensionality $d = 100$ (i.e. same density, but a much wider table). As can be seen, comparing dense matrix computations (apples to apples), the DBMS+R combination is much faster than the R language built-in matrix multiplication with a dense matrix storage. Moreover, R ends up failing due to reaching RAM capacity. Spark is one order of magnitude slower mainly due to overhead on a single node (i.e. it needs much more hardware). The main reasons for better performance are: (1) the DBMS reads the matrix in blocked binary form into RAM, whereas R parses the input text file line by line; (2) the DBMS avoids an expensive matrix transposition; (3) the DBMS operator directly transforms x_i into z_i in RAM and then multiplies z_i by itself, instead of extracting a matrix row and a matrix column and then multiplying them (even if transposition was avoided). The gap becomes much wider with the sparse matrix operator, being 2X-3X faster than the dense matrix operator. In short, the DBMS+R combination is faster than R even when the input matrix fits in RAM.

The comparison with our sparse operator is not fair to R (i.e. not comparing apples to apples), but manipulating sparse matrices in R is complicated. We must mention it is possible to compute Γ in R with faster and more memory-efficient mechanisms including data frames (to manipulate nested arrays), calling BLAS routines (part of the LAPACK library), vectorized operations (to skip loops), block-based I/O (reading the data set in blocks) and C code (e.g. Rcpp), all of which would require a significant programming effort. Nevertheless, it is unlikely R routines for sparse matrices could be much faster than our sparse summarization operator for large disk-resident matrices. We leave such study for future work.

Table 3 Comparing R language, Spark and DBMS+R to get summarization matrix; real data set NetIntrusion (replicated); dense operator (all dn entries included) and sparse operator (zero entries deleted); times in secs (0* means < 0.2 secs).

d	n	R dense	Spark dense	DBMS dense	DBMS sparse
10 density=12%	100k	1	16	0*	0*
	1M	5	35	1	1
	10M	50	258	9	2
	100M	fail	stop	110	53
100 density=27%	100k	6.5	177	3	2
	1M	44.1	1204	31	11
	10M	fail	stop	333	105
	100M	fail	stop	3424	1015

5.3 Inter-node Parallelism: Computing I on a Parallel Cluster

We compare our array DBMS cloud solution with Spark, currently the most popular analytic system from the Hadoop stack. We tried to be as fair as possible: both systems run on the same hardware, but we acknowledge SciDB benefits from data sets being pre-processed in matrix form. We analyze data summarization and model computation.

Cloud System Setup: We created a configuration script to automate the complicated system setup in parallel, which makes installing and running SciDB in the cloud as easy as running HDFS/Spark. Aside from a suitable Linux OS installation, packaged software components used as part of the system include: the SciDB database server, the R runtime, and the SciDB-R integration package. We also included several customized scripts in the system that can help with the cluster set up, parallel data loading from Amazon S3, data summarization and model computation. The user-friendly web GUI and close integration with R can greatly lower the entry barrier for using SciDB for complicated analysis, making the analytical processes a more flexible, extensible and interactive experience with all kinds of functions and libraries that R supports.

Parallel Load operator: As mentioned above, we developed a parallel data loading operator that makes our system competitive with Hadoop systems running on top of HDFS. In the interest of saving space, we did not conduct a benchmark comparing loading speed between our cloud system and Spark. But we should point out our parallel operator to load matrix X from a CSV file takes similar time to copying the CSV file from the Unix file system to HDFS plus creating data set in Spark’s RDD format. We would also like to point out that the time to load data is considered less important than the time to analyze data because it happens once.

Hardware: We use a parallel cluster in the cloud with 100 virtual nodes, where each node has 2 VCPUs (2 cores) running at 2 GHz each, 8GB RAM and 1 TB. Spark takes advantage of an additional coordinator node with 16 GB RAM (i.e. Spark uses $P + 1$ nodes). Notice this is an entry-level parallel database system configuration, not an HPC system which would have much

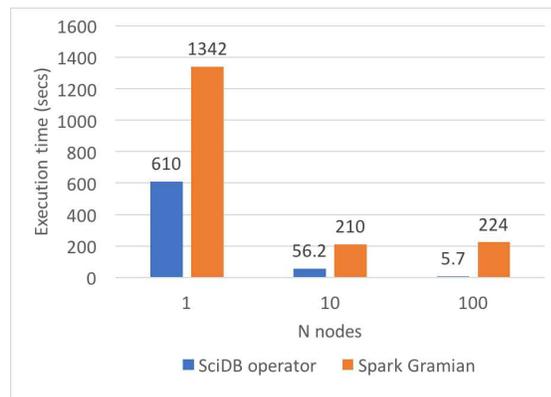


Fig. 2 Γ computation time varying number of nodes: array DBMS vs. Spark ($n = 100M, d = 38$).

larger RAM and many more cores. As mentioned before, we developed scripts for fast automatic cluster configuration.

In Figure 2 we compare both systems computing the Γ summarization matrix. That is, we compute the submatrices L, Q in Spark using its *gramian()* function, without actually computing the machine learning model. That is, we could simply export L, Q from Spark to R and compute the model in R in RAM. As can be seen, our system scales out better than Spark as P grows. In fact, Spark increases time when going from 10 to 100 nodes, which highlights a sequential bottleneck. At 100 nodes the gap between both systems is close to two orders of magnitude.

Figure 3 compares both systems with a fundamental model: linear regression (LR). This comparison is also important because LR is computed with Stochastic Gradient Descent (SGD) in Spark, whereas ours is based on data summarization. In Spark we used 20 iterations, which was the recommended setting in the user’s guide to get a stable solution. With one node the difference is more than one order of magnitude, whereas at 100 nodes our system is more than two orders of magnitude faster than Spark. We emphasize that even running Spark with one iteration, which would get an unstable solution, our system is 10X faster (i.e. $920/20 \approx 46$, compared to 6 seconds).

5.4 Intra-node parallelism: multiple threads per node

We now study scale-up varying the number of threads. The mapping of threads to cores is accomplished by varying the number of SciDB instances per node, configured at DBMS startup time on a local server with a 4-core CPU (i.e. our database server, not a cloud system with virtual CPUs), 4 GB RAM and 1 TB disk. Table 4 presents parallel scale-up experiments varying the number of threads. Our objective is to determine the best number of threads in the

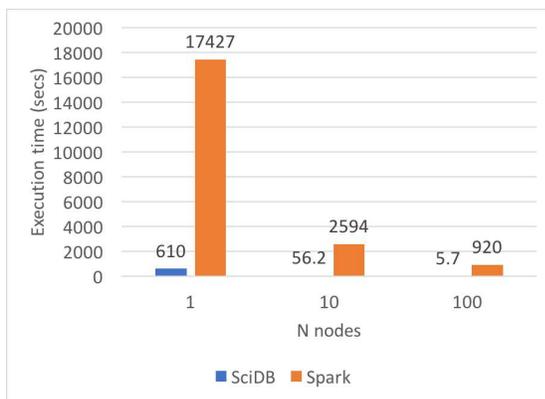


Fig. 3 Model computation time for LR varying number of nodes: array DBMS vs. Spark ($n = 100M, d = 38$).

Table 4 Intra-node parallelism: Γ computation with multi-threaded processing varying # threads ($d = 38, 1$ node, 4-core CPU); times in secs.

# threads	$n=100k$		$n=1M$		$n=100M$	
	time	speedup	time	speedup	time	speedup
1	1.4	1.0	13.0	1.0	146.6	1.0
2	0.7	2.0	6.8	1.9	82.8	1.8
4	0.8	1.8	6.2	2.1	60.8	2.4
8	1.0	1.4	5.3	2.4	52.1	2.8

4-core CPU. As the table shows, as the # of threads increases time decreases as well, but unfortunately at 4 threads speedup slightly decreases. Based on this evidence it is not worth going beyond 4 threads. Therefore, the optimal number of threads is 2, lower than the total number of cores. Our explanation follows. At the lower end one thread is not capable of interleaving FLOPs for Γ with I/O operations to scan X . At the other end, 4 threads introduce excessive overhead due to frequent thread context switches, and interleaving scan access to different addresses on the same disk. Since Γ is small it is feasible to maintain and incrementally update it in cache memory; we leave this aspect for future work. It is not worth going beyond the number of cores in the CPU, 4 in this case. The lesson learned is that if scale-up capacity has been reached the only alternative to increase performance is to scale out.

5.5 Intra-node Parallelism: accelerating Γ computation with a GPU

The previous experiments beg the question if processing can be further accelerated with GPUs. Since Spark does not offer off the shelf functions exploiting GPUs we cannot compare it. However, we emphasize that we established that our solution is orders of magnitude faster than Spark. Therefore, it is unlikely that Spark could be faster exploiting GPUs if our solution also uses GPUs.

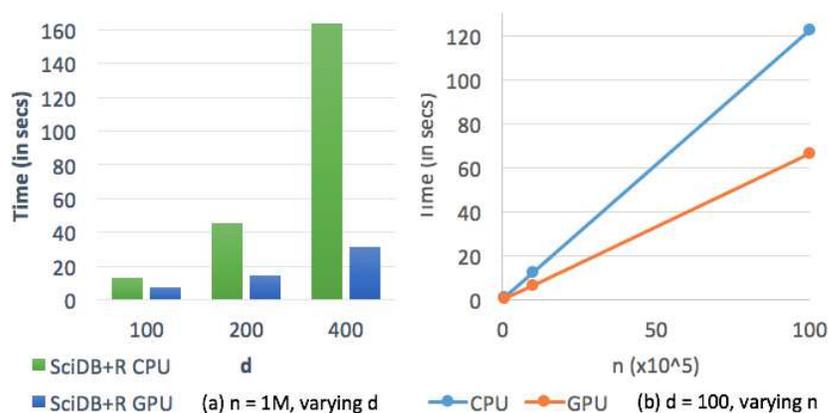


Fig. 4 T computation time: CPU vs. GPU.

Setup: In this section we examine how much time improvement GPU processing can bring to model computation using T . We ran our experiments on a parallel cluster with 4 nodes. Each node has an 8-core Intel Xeon E5-2670 processor, an NVIDIA GRID K520 GPU with 1,536 CUDA cores. The GPU card has 4GB of video memory, while the machine has 15 GB of main memory. On the software side, on those machines we installed Linux Ubuntu 14.04, which is currently the most reliable OS to run the SciDB array DBMS. The system is equipped with the latest NVIDIA GPU driver version 352.93. We also installed the PGI compilers for OpenACC and SciDB 15.7. We revised our CPU operator C++ code with OpenACC annotations marking key vector operations in the loops to be parallelized with GPU cores so that they are automatically distributed for parallelism. The data sets are synthetic, dense matrices with random numbers. We loaded the data sets into SciDB as arrays split into equal-sized chunks and evenly distributed across all parallel nodes, where each chunk holds 10,000 data points.

Figure 4 illustrates GPU impact on data summarization, which is the most time consuming step in our 2-phase algorithm. The bar figure shows the GPU has little impact at low d (we do not show times where $d < 100$ since the GPU has marginal impact), but the gap between CPU and GPU rapidly widens as d grows. On the other hand, the right plot shows the GPU has linear time complexity as n grows, an essential requirement given the I/O bottleneck to read X . Moreover, the acceleration w.r.t CPU remains constant.

We now analyze GPU impact on the overall model computation, considering machine learning models require iterative algorithms. Table 5 shows the overall impact of the GPU. As can be seen SciDB removes RAM limitations in the R runtime and it provides significant acceleration as d grows. The GPU provides further acceleration despite the fact the dense matrix operator is already highly optimized C++ code. The trend indicates the GPU becomes more effective as d grows. Acceleration is not optimal because there is over-

Table 5 Comparing computation of model Θ using R, DBMS+R, and DBMS+R+GPU; dense matrix operator; $P=1$ node (CPU=8 cores); times in secs.

n	d	model	CPU		GPU
			R	R+SciDB	R+SciDB
1M	100	PCA	29	14	8
1M	200	PCA	90	46	16
1M	400	PCA	fail	165	33
10M	100	PCA	fail	147	104
10M	200	PCA	fail	466	215
10M	400	PCA	fail	1598	455
10M	100	LR	fail	147	103
10M	200	LR	fail	464	212
10M	400	LR	fail	1594	451

head moving chunk data to contiguous memory space and transferring data to GPU memory and because the final sum phase needs to be synchronized. However, the higher d is, the more FLOP work done by parallel GPU cores.

6 Related Work

Data summarization to accelerate the computation of machine learning models, has received significant attention. We believe we are the first to reduce data set summarization to matrix multiplication, but computing matrix multiplication in parallel in a distributed memory database architecture has been studied before [10]. A similar, but significantly less general, data summarization was proposed in [28] to accelerate the computation of distance-based clustering: the sums of values and the sums of squared values. This is mathematically equivalent to a diagonal Q matrix extracted from Γ . In a ground-breaking paper [3] exploited such summaries as multidimensional sufficient statistics for the K-means and EM clustering algorithms. We were inspired by this approach. It is important to explain differences between [28] and [3]: data summaries were useful only for one model (clustering). Compared to our proposed matrix, both [28, 3] represent a (constrained) diagonal version of Γ because dimension independence is assumed (i.e. cross-products, covariances, correlations, are ignored) and there is a separate vector to capture L , computed separately. From an algorithmic perspective, our parallel summarization algorithm boils down to one parallel matrix multiplication, whereas those algorithms work with scattered aggregations. Caching and sharing intermediate computation is an important mechanism to accelerate data mining computations in parallel [22]. Our summarization matrix shares the same motivation, but instead of caching data in RAM like the operating system, we compute a summarization matrix which benefits many machine learning models. A much more general data summarization capturing up to the 4th moment of a probability distribution was introduced in [8], but it requires transforming variables (i.e. building bins for histograms), which are incompatible with most machine learning methods and which sacrifice numeric accuracy. Recent work [5] on streams

defends building specialized data structures like histograms and sketches. Parallel data summarization has received moderate attention because it has not been seen as a widely applicable technique. Reference [17] highlights the following parallel summarization techniques: sampling, incremental aggregation, matrix factorization and similarity joins. A competing technique to accelerate model computation is gradient descent [9,18]. Unfortunately, it has three major drawbacks: it does not translate well into a fully parallel computation (the gradient is sequentially updated), it requires carefully tuning a step size to update the solution (data set and model dependent) and the solution is approximate (not as accurate as summarization). Data summarization is related to parallel query processing [12], considered as an aggregation query.

There is large body of work on computing machine learning models in Hadoop “Big Data” systems, before with MapReduce [2] and currently with Spark [27]. On the other hand, computing models with parallel DBMSs has received less attention [15,23] because they are considered cumbersome and slower. PCA computation with the SVD numerical method was accelerated using parallel summarization in a parallel DBMS [20]. However, there are important differences: in this work sufficient statistics are scattered, whereas they are integrated into a single matrix in our current paper; there are no theory results on parallel processing; array storage is much better than relational storage for matrices; linear regression, a harder problem, was not considered; this work proposed UDFs for SQL instead of array operators; finally, GPUs were not considered. The F summarization matrix was proposed in [21], which introduces two parallel matrix-based algorithms for dense and sparse matrices, respectively. Later, [29] introduced an optimized UDF to compute the F matrix on a columnar DBMS, passing it to R for model computation. However, these papers did not explore the parallel array DBMS angle in depth, with a large number of nodes, multi-threaded processing and GPUs. We must point out that since matrix computations are CPU intensive, it was necessary to study how to further accelerate computation with GPUs. Another practical aspect we did not envision initially as a major limitation turned out to be a bottleneck: loading data into the array DBMS, especially in parallel. In this paper we tackled all these research issues: scale out parallelism, scale up parallelism, and parallel matrix loading. A careful performance benchmark comparison between our system, R alone and Spark rounds our contribution.

This article is a significant extension of [30]. We added important theory results on parallelism. From a systems perspective we now study and contrast intra-node and inter-node parallelism. We added experiments measuring parallel speedup and multi-threaded processing. Related work was significantly expanded, especially considering summarization and parallel processing.

7 Conclusions

We presented a system integrating a parallel array DBMS (SciDB) and a host analytic language (R). Unlike most big data analytic systems, our system is

not built on top of HDFS, but works in a traditional shared-nothing DBMS architecture. We studied the parallel computation of a general summarization matrix that benefits machine learning models. Specifically, we studied the computation of PCA and LR, two fundamental, representative, models. We showed data summarization is a common bottleneck in many matrix equations. We introduce a parallel, scalable, algorithm to compute such summarization matrix. We presented important speedup and communication theory results that characterize the efficiency and limits of our parallel algorithm. From a systems perspective, we considered inter-node and intra-node parallelism. Since matrices are generally manipulated as arrays in a traditional programming language we studied how to optimize its computation with a parallel array DBMS. We carefully considered an array DBMS has significantly different storage compared to traditional relational DBMSs based on SQL. Moreover, we studied how to further accelerate summarization with multi-threaded processing and GPUs. Benchmark experiments showed our system has linear speedup as the number of nodes increases. The GPU further accelerates the computation of the summarization matrix. On the other hand, our experiments show our system is orders of magnitude faster than Spark and R language runtime, to summarize the data set and to compute the same machine learning model.

There are many research issues. Our parallel summarization is ideal for an array DBMS because it is a matrix computation. However, it has potential applicability in a traditional parallel DBMS based on SQL, perhaps with some performance penalty. We need to further study the computation of the summarization matrix on sparse matrices. The summarization matrix is a great fit for multicore CPUs and GPUs, but we need to study in more depth how to compute vector outer products with vectorized execution. Given the wide adoption of Spark, instead of competing with it, we plan to integrate our algorithms with Spark to make Spark scale beyond main memory across the cluster. The main drawback about integrating systems is the need to move data between the DBMS and HDFS/Spark. Finally, a major limitation of our system, compared to Spark and MapReduce, is the lack of fault tolerance during processing: if a node fails the entire computation must be restarted. So we need to study how to dynamically support migrating summarization from a failing node to another node maintaining checkpoints.

Acknowledgments

This work concludes a long-time project, during which the first author visited MIT from 2013 to 2016. The first author thanks the guidance from Michael Stonebraker to move away from relational DBMSs to compute machine learning models in a scalable manner and to understand SciDB storage and processing mechanisms for large matrices.

References

1. M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.
2. A. Behm, V.R. Borkar, M.J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V.J. Tsotras. ASTERIX: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases (DAPD)*, 29(3):185–216, 2011.
3. P. Bradley, U. Fayyad, and C. Reina. Scaling clustering algorithms to large databases. In *Proc. ACM KDD Conference*, pages 9–15, 1998.
4. Q. Chen, M. Hsu, and R. Liu. Extend udf technology for integrated analytics. *Data Warehousing and Knowledge Discovery*, 5691:256–270, 2009.
5. G. Cormode. Compact summaries over large datasets. In *Proc. ACM PODS*, pages 157–158, 2015.
6. S. Das, Y. Sismanis, K.S. Beyer, R. Gemulla, P.J. Haas, and J. McPherson. RICARDO: integrating R and hadoop. In *Proc. ACM SIGMOD Conference*, pages 987–998, 2010.
7. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
8. W. DuMouchel, C. Volinski, T. Johnson, and D. Pregybon. Squashing flat files flatter. In *Proc. ACM KDD Conference*, 1999.
9. R. Gemulla, E. Nijkamp, P.J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proc. KDD*, pages 69–77, 2011.
10. D.V. Gucht, R. Williams, D.P. Woodruff, and Q. Zhang. The communication complexity of distributed set-joins with applications to matrix multiplication. In *Proc. ACM PODS*, pages 199–212, 2015.
11. A. Hameurlain and F. Morvan. Parallel relational database systems: Why, how and beyond. In *Proc. DEXA Conference*, pages 302–312, 1996.
12. A. Hameurlain and F. Morvan. CPU and incremental memory allocation in dynamic parallelization of SQL queries. *Parallel Comput.*, 28(4):525–556, 2002.
13. J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2006.
14. T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning*. Springer, New York, 1st edition, 2001.
15. J. Hellerstein, C. Re, F. Schoppmann, D.Z. Wang, E. Fratkin, A. Gorajek, K.S. Ng, and C. Welton. The MADlib analytics library or MAD skills, the SQL. *Proc. of VLDB*, 5(12):1700–1711, 2012.
16. A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.
17. F. Li and S. Nath. Scalable data summarization on big data. *Distributed and Parallel Databases*, 32(3):313–314, 2014.
18. J. Liu, S.J. Wright, C. Re, V. Bittorf, and S. Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. *The Journal of Machine Learning Research*, 16(1):285–322, 2015.
19. C. Ordonez. Statistical model computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(12):1752–1765, 2010.
20. C. Ordonez, N. Mohanam, and C. Garcia-Alvarado. PCA for large data sets with parallel data summarization. *Distributed and Parallel Databases*, 32(3):377–403, 2014.
21. C. Ordonez, Y. Zhang, and W. Cabrera. The Gamma matrix to summarize dense and sparse data sets for big data analytics. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(7):1906–1918, 2016.
22. S. Parthasarathy and S. Dworkadas. Shared state for distributed interactive data mining applications. *Distributed and Parallel Databases*, 11(2):129–155, 2002.
23. M. Stonebraker, D. Abadi, D.J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.
24. M. Stonebraker, J. Becla, D.J. DeWitt, K.T. Lim, D. Maier, O. Ratzesberger, and S.B. Zdonik. Requirements for science data bases and SciDB. In *Proc. CIDR Conference*, 2009.

25. M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.
26. M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it’s time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.
27. M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud USENIX Workshop*, 2010.
28. T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In *Proc. ACM SIGMOD Conference*, pages 103–114, 1996.
29. Y. Zhang, C. Ordonez, and W. Cabrera. Big data analytics integrating a parallel columnar DBMS and the R language. In *Proc. of IEEE CCGrid Conference*, 2016.
30. Y. Zhang, C. Ordonez, and L. Johnsson. A cloud system for machine learning exploiting a parallel array DBMS. In *Proc. DEXA Workshops (BDMICS)*, pages 22–26, 2017.