A Comparison of Lasso-type Algorithms on Distributed Parallel Machine Learning Platforms

Jichuan Zeng, Haiqin Yang, Irwin King and Michael R. Lyu
Shenzhen Key Laboratory of Rich Media Big Data Analytics and Applications
Shenzhen Research Institute, The Chinese University of Hong Kong
Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong
{hqyang, king, lyu}@cse.cuhk.edu.hk

Abstract

Due to tremendous increase of data, scalability becomes a challenging issue for many modern machine learning algorithms. Various distributed machine learning platforms, e.g., GraphLab, Spark and Petuum, are proposed to tackle this issue. Lasso algorithm, as its effectiveness in performing regression tasks while selecting the important features simultaneously, has become a benchmark machine learning method deployed in these platforms. However, rare work discusses the performance of the Lasso algorithm systematically and many other lasso-type algorithms are not well-studied yet. In addition, how to relieve the "Ninja performance gap" between optimized code and most of these frameworks is a difficult task. To resolve the above tasks, we present a detailed deployment of the lasso-type algorithms in several state-of-the-art distributed machine learning platforms. We characterize the performance of the native implementation and identify the potential parts to reduce the performance gap. We give a comprehensive comparison on running time, easy-of-deployment and capability of handling big data, which will enable end-users to choose platforms based on their goals.

1 Introduction

Regression via Least Absolute Shrinkage and Selection Operator (Lasso) is a well-known statistical machine learning method [19]. This method aims to tackle the regression tasks while selecting the important features where the dimension of the data is highly larger than the number of samples. A motivating example comes from gene selection where the number of samples is in the order of $10^3 - 10^4$ due to the high cost of experimental treatment while the number of dimensions can be up to 10^9. Lasso regression has provided an effective solution to seek a linear effect of the most informative gene while discarding those irrelevant ones. Due to its effectiveness, Lasso has been extended widely. Many lasso-type algorithms such as elastic net [25], group lasso [22], sparse group lasso [6], and so on have been proposed and attained successfully in the applications, where data contain similar features, group features, and sparse group features.

As the number of data in application domains increases radically, computation scalability becomes a challenging issue of many machine learning algorithms, including lasso-type algorithms. To meet the requirements of real-world applications, various algorithms are proposed in the literature. One family of methods is to stream the examples to an online-learning algorithms [1]. Typical online learning algorithms for lasso-type models include truncation method [11], forward-backward splitting method [5], dual averaging method [20, 21], and etc. These methods may be insufficient and limited when the data are stored distributedly. Hence, parallelizing the batch-trained algorithms and executing distributively become another family of solutions. Currently, many platforms have been proposed and claimed to provide programming and runtime support for distributed/parallel machine learning algorithms, including GraphLab [12], Mahout [15], MLBase [10], and Petuum [4]. Other

systems such as Pregel [13], Giraph [3], Hama [18], Spark [24], Twitter Storm and Dryad [9] have been designed for general-purpose distributed platforms, but treat it as an important component. Now, Lasso regression has become a benchmark method deployed in these platforms.

However, there is no clear and complete comparison for the Lasso regression in these platforms. Moreover, many other lasso-type algorithms are not implemented and well-studied in these platforms yet. How to borrow the implementation idea of lasso and transfer it to other lasso-type algorithms is still non-trivial. In addition, the "Ninja performance gap" [17] between natively written graph code and well-tuned hand-optimized code is common and scales to be multiple orders of magnitude. How to relieve the "Ninja performance gap" in Lasso-type algorithms implementation is a challenging task. To this end, we start from Lasso and group Lasso, which are two representative lasso-type algorithms and test them in three distributed parallel platforms, Graphlab, Spark, Petuum. We select these platforms because they are currently popularly distributed parallel machine learning platforms which are easily deployed and very promising for iterative algorithm updating. We characterize the performance of the native implementation and identify the potential parts to reduce the performance gap. We give a comprehensive comparison on running time, convergence, and easy-of-deployment. We hope our efforts will grow into a widely used, standard benchmark for this sort of platforms. In the future, a implementor of a new or existing platform need only implement these codes and compare with our numbers.

The rest of the paper is organized as follows. Section 2 outlines the general framework of Lasso-type models, especially the solutions for Lasso and Group Lasso, respectively. Section 3 depicts how to separate the data in distributed machines. Section 4 summarizes three popular distributed parallel platforms and how to implement Lasso and Group Lasso accordingly. Section 5 provides a detailed comparison and discussion on the results. Finally, Section 6 concludes the whole investigation with brief discussion on future work.

2 Lasso-type Models

Suppose we are given a set of identically distributed samples $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, where $\mathbf{x}_i \in \mathbb{R}^d$ is the feature for the i-th sample and y_i is the i-th response, we try to find a linear model as follows:

$$y = \mathbf{x}^T\mathbf{w} + \epsilon \tag{1}$$

where $\mathbf{w} \in \mathbb{R}^d$ is the regression weight vector to be learned and ϵ is the error term, or noise, usually taking the gaussian distribution.

To seek the optimal solution in Eq. (1), one usually turn to the following regularized minimization framework:

$$\min_{\mathbf{w}\in\mathbb{R}^d} \quad \mathcal{F}(\mathbf{w}) = \sum_{i=1}^{N} L(\mathbf{x}_i^T\mathbf{w}, y_i) + \Omega_\lambda(\mathbf{w}) \tag{2}$$

where $L(\cdot)$ is a non-negative, usually convex loss function. $\Omega_\lambda(\mathbf{w})$ is a regularization term to control the model complexity while achieving a certain property. $\lambda \geq 0$ is a constant to trade-off the loss and the regularization.

In the following, we mainly focus on Lasso and Group Lasso as they are two typical lasso-type algorithms and widely applied in many real-world applications.

2.1 Lasso

The Lasso model [19] is an instance of ℓ_1 regularized loss minimization model which minimize the residual squared loss with a constraint on ℓ_1 norm of the regression coefficient, or equivalently, minimizing the following sum of the residual squared loss and the ℓ_1 norm of the regression coefficient:

$$\min_{\mathbf{w}\in\mathbb{R}^d} \quad \mathcal{F}_L(x) = \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_1 \tag{3}$$

where $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T \in \mathbb{R}^{N\times d}$ concatenates the features row-by-row. It should be noted that the key of Lasso is to encourage sparsity by the ℓ_1-norm on the weight, i.e., $\|\mathbf{w}\|_1$. We follow

the parallel coordinate descent method in [2] to solve the Lasso problem. That is, we choose a subset of coordinates P_t from $\{1, 2, \ldots, d\}$ uniformly at random and update the weight parallelly by

$$w_j \leftarrow w_j + \triangle w_j,$$

where $\triangle w_j$ is calculated by the following equation:

$$\triangle w_j = \max\{-w_j, -(\nabla \mathcal{F}_L(\mathbf{w}))_j/\beta\},$$

where $\beta > 0$ is a loss-dependent constant.

After that, we compute the collective update by

$$\Delta \mathbf{w} = \sum\nolimits_{i \in P_t} \triangle w_i.$$

It is noted that under a certain condition, the above updating can be converged [2].

2.2 Group Lasso

The Group Lasso [22] is an extension of Lasso to select factors in a group manner. Suppose the data consist of K groups, i.e., $\mathbf{X} = [\mathbf{X}_1, \ldots, \mathbf{X}_K]$, where $\mathbf{X}_k \in \mathbb{R}^{n \times d_k}$ denotes the feature vectors corresponding to the k-th group and d_k denotes the size of the feature in the k-th group. We have, $\sum_{k=1}^{K} d_k = d$. The goal of Group Lasso is to seek the weight by minimizing the following objective function:

$$\min_{\mathbf{w} \in \mathbb{R}^d} \quad \mathcal{F}_G(x) = \frac{1}{2}\|\sum_{k=1}^{K} \mathbf{X}_k \mathbf{w}_k - \mathbf{y}\|_2^2 + \lambda \sum_{k=1}^{K} \sqrt{d_k}\|\mathbf{w}_k\|_2 \tag{4}$$

where $\mathbf{w}_k \in \mathbb{R}^{d_k}$, $k = 1, \ldots, K$. We can see that if $d_k = 1$ for all groups, it reduce to the original Lasso model. Hence Group Lasso acts like a group-level Lasso, which selects a group of factors for producing accurate prediction.

Here, we adopt the Nesterov's methods [14] to evaluate the objective value and (sub)gradients at each iteration. The Nesterov's method updates two sequences of variables, $\mathbf{s}_j$ and $\mathbf{w}_j$, alternatively, where $\mathbf{s}_j$ is the sequence of search points, and $\mathbf{w}_j$ is the sequence of approximate solution points. $\mathbf{s}_j$ can can be computed through affine transformation of $\mathbf{w}_{j-1}$ and $\mathbf{w}_j$. Base on the (sub)gradient of $\mathbf{s}_j$, the approximate points $\mathbf{w}_j$ can be acquired by

$$\mathbf{s}_j = \mathbf{w}_j - \alpha_j(\mathbf{w}_j - \mathbf{w}_{j-1}) \tag{5}$$

$$\mathbf{w}_{j+1} = P_G(\mathbf{s}_j - \beta \nabla L(\mathbf{s}_j)) \tag{6}$$

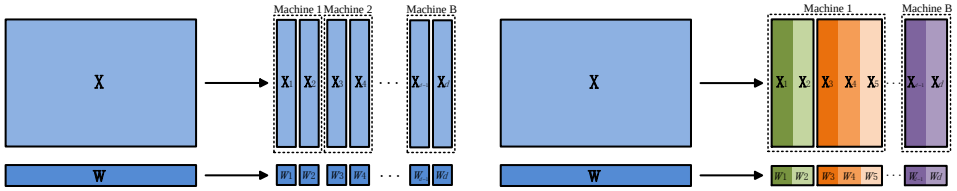
where $\alpha_i \in (-1, 1)$ is the affine coefficient, $\beta > 0$ is the gradient step size, P_G is the Euclidean projection of gradient search step $\mathbf{v}$ on the convex set G, that is, $P_G(v) = \min_{\mathbf{w} \in G} \frac{1}{2}\|\mathbf{w} - \mathbf{v}\|^2$.

3 Data Splitting

A key issue of large scale datasets is that the data is too large to store centrally, or when it is collected in a distributed manner. Hence, usually there are three ways to split the data distributedly [16]:

- row block splitting: the data $\mathbf{X}$ is partitioned into row blocks $\mathbf{X}_{(1)}, \mathbf{X}_{(2)}, \ldots, \mathbf{X}_{(M)}$. Stacking them forms $\mathbf{X}$.
- column block splitting: the data $\mathbf{X}$ is partitioned into column blocks $\mathbf{X} = [\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_N]$.
- general block splitting: the data $\mathbf{X}$ is partitioned into MN sub-blocks, where the rows are divided by M blocks and the columns are divided by N blocks. The (i, j)-th block is denoted by $\mathbf{X}_{i,j}$.

By taking into account the property of Lasso-type algorithms, we adopt the second strategy, i.e., column block, to split the data. Figure 1 gives an illustrated example of the splitting strategy of Lasso and Group Lasso, where in Lasso, the d columns of data are uniformly distributed in different nodes. While in Group Lasso, similar splitting is conducted, but data in a group will be resided only in one node, and each node will contain the same number of groups.



(a) Lasso (b) Group Lasso

Figure 1: Data splitting strategy. Fig. 1(a) shows the splitting strategy of Lasso, where the d columns/dimensions of data are uniformly distributed in different nodes. Fig. 1(b) illustrates splitting strategy of Group Lasso. Similarly, the K columns/groups of data are uniformly separated in different nodes, where in each group, the data will be resided in one node.

Hence, we can separate the objective function block-wisely, i.e., $\mathcal{F}(\mathbf{w}) = \sum_{b=1}^{B} \mathcal{F}_b(\mathbf{w}_b)$, where $\mathbf{w}_b$ is the b-th block of $\mathbf{w}$. More specifically, the regularization term is expressed as $\Omega_\lambda(\mathbf{w}) = \sum_{b=1}^{B} \Omega_\lambda(\mathbf{w}_b)$ while the loss function is expressed as $\mathcal{L}(\mathbf{X}\mathbf{w}, \mathbf{y}) = \frac{1}{2}\sum_{b=1}^{B} \|\mathbf{X}_b\mathbf{w}_b - y_b\|^2$. We can then execute and obtain $\mathbf{w}_b$ parallelly. Algorithm 1 depicts the implementation of a distributed solver for the Lasso-type models.

Algorithm 1 Distributed solver for Lasso-type models

Parameters: T, λ, n, d, B
Input: Observed matrix $\mathbf{X}$, and response vector $\mathbf{y}$
Output: The coefficient vector $\mathbf{w}$.
Initialize $\mathbf{X}_b$ is stored in node b, set $\mathbf{w} = 0$;
for $t = 1, 2, ...T$ **do**
  Update $\mathbf{X}\mathbf{w}$ by *Reduce* all nodes
  random sample s selected blocks from B blocks;
  for $b = \{1, \ldots, s\}$ parallelly **do**
    for each j in block b **do**
      $g_j \leftarrow \nabla\mathcal{L}(\mathbf{X}\mathbf{w}, \mathbf{y})$
      $w_j^{t+1} \leftarrow \mathbf{prox}_{\Omega_\lambda(\cdot)}(w_j^t - \delta g_j)$
    end for
  end for
end for

4 Distributed Parallel Machine Learning Platforms and Implementation

In the following, we present three popular distributed parallel machine learning platforms and depict how to implement Lasso and Group Lasso on them.

4.1 GraphLab

GraphLab [12] is a graph-based distributed parallel machine learning platform, which started from a project at Carnegie Mellon University in 2009. It consists of the following unique features:

- It provides a parallel-programming abstraction that is targeted for sparse iterative graph algorithms through a high-level programming interface, that is "vertex programming". Each GraphLab process is multi-threaded to fully use multi-core resources available on modern cluster nodes.
- It includes asynchronicity and allows for very appealing computational models, since when writing the code associated with a vertex, one need only consider the computation required to update the state of that vertex; one can more or less ignore the remainder of the computation.
- It supports reading from and writing to both Posix and HDFS file systems.

- It is unique in that its computational model is pull-based and asynchronous. Each vertex in the graph constantly requests data from its neighbors in order to update its own state.

Implementation of Lasso. We define the data vertices as the the storage of $\mathbf{X}$. Each data vertex stores one column of the data matrix $\mathbf{X}_i$ and the response $\mathbf{y}$ with the corresponding coefficient w_i, which consists of a triple $\langle \mathbf{X}_i, w_i, \mathbf{X}_i\mathbf{y} \rangle$. We then utilize `graph.load` to parallelly load data into distributed machines. The GraphLab computation begins with an initialization by performing `transform_vertices` over all the data vertices. And then we also use `graph.transform_vertices` to apply `Shoot` in each vertex (column), and update the w_i in it. At the end of one iteration, we gather the information of all the data vertices we need to compute collective objective.

In order to make the computation more efficient, we do not take the edges between the data vertices into account. Instead, we use a *broadcast* way to traverse all data vertices and use an *aggregate* method to record running states. By this way, we can reduce the memory consumption significantly.

Implementation of Group Lasso. We define two types of vertices: data vertices and group vertices. Again, the i-th data vertices are used to store a triple $\langle \mathbf{X}_i, w_i, \mathbf{X}_i\mathbf{y} \rangle$. The group vertices collect Euclidean projection P_G and other required statistics. A group vertex is associated with each data vertices.

We initialize each X_i by performing a `transform_vertices` operation over all the data vertices. Next, we use GraphLab's *gather-apply-scatter* abstraction to implement Nesterov's method. In gathering phase, the i-th group vertex collects the gradient search step $\mathbf{v_{G_i}}$ from the data vertices belong to the G_i. The data vertex collects the Euclidean projection P_{G_i} from corresponding group vertex. In the *applying* phase, the data vertex update its regression coefficient with new w_i value. The group vertices compute P_G based on $\mathbf{v_G}$.

4.2 Spark

Spark [24] is an open-source Hadoop MapReduce alike general-purpose cluster computing system which was developed by the AMP lab at University of California, Berkeley. Spark utilizes an abstraction called Resilient Distributed Datasets (RDDs) [23] to do in-memory data-processing under the framework of MapReduce. RDD is a read-only collection of objects providing a restricted form of shared memory, based on coarse-grained transformations (e.g., map, filter and join) over a set of data or existing RDDs. RDD also provides fault tolerance through its lineage (the transformation log). With the help of RDD's in-memory operations, Spark allows users to write parallel computations using a set of high-level operators, without concerning work distribution and fault tolerance.

Spark is written in Scala, which makes it feasible for nearly all JVM-based platforms such as Hadoop Yarn, Apache Mesos. Another excellent feature of Spark lays on the imitation of Scala's collections API and functional style, which providing beautiful codes and good support for statistical computing.

Implementation of Lasso. We begin by creating a RDD named `data`, which is read and parsed from data in disk storage. We split the data into $\mathbf{X}$ `x_rdd` and $\mathbf{y}$ `y_rdd` via `data.map`. The initialization of matrix is completed by `x_rdd.map`. Next we enter the main loop and perform Shoot algorithm which is the coordinate descent by `x_rdd.map(p=>Shoot).reduce`. In `Shoot`, $\mathbf{w}_j$ are updated, and broadcast to all Worker nodes through `sc.broadcast`. Most of the code of Lasso is run locally in Driver, except the initialization and shoot algortihm.

Implementation of Group Lasso. In our implantation of group Lasso, Spark is asked to manage and aggregate data at the group level, instead of the column level. Therefore, the difference from Lasso is this Spark implementation begins by creating a RDD `g_x_rdd`. This RDD stores, the group identifier and its associated list of columns of data. Then the initialization is applied to `g_x_rdd` similar to Lasso implementation. Now it comes to the main loop of Nesterov' method, we will employ two steps to in this process. Firstly, we compute the Euclidean projection P_G, which denoted as `g_proj_rdd` for each target group by a simple `g_x_rdd.map`. Then, the aggregates are used to update the value of $\mathbf{w}$ via `g_x_rdd.map(p=>update(g_proj_rdd)).reduce`.

4.3 Petuum

Petuum [8] is a framework for iterative-convergent distributed machine learning. Currently, they are parameter server for global parameters and variable scheduler for local variables. Petuum-PS supports novel consistency models such as bounded staleness, which achieve provably good results on iterative-convergent ML algorithms, that is Stale Synchronous Parallel (SSP) [8].

The most attractive aspect of Petuum is the SSP, SSP is a soft synchronization, which blocks the worker thread until all worker threads are within a specified window of the current iteration. SSP is a middle ground of Bulk Synchronous Parallel (BSP) which block until thread catch up, and Asynchronous, in which thread will go to next iteration without waiting. SSP can reduce the network waiting time as it purported.

Implementation of Lasso. We define a SSPtable which is a table-based interface of SSP parameter server, and used to store $\mathbf{w}$ in the parameter servers, which only contain one row, and the dimension of the SSPtable is the dimension of the predictor $\mathbf{X}$. And then, we initialize threads to process the coordinate descent. In one iteration, a thread will implement `Shoot`, and then check the objective, at the end of the iteration followed by a `TableGroup.Iteration` step to mark the clock increment of this thread.

Implementation of Group Lasso. In Group Lasso, we utilize a SSPtable to store $\mathbf{w}$ as we did in Lasso. Different from Lasso implementation, the groups in each node are process by worker threads alternatively. In each worker thread, it will compute current group's Euclidean projection P_G and update the corresponding $\mathbf{w_G}$ belongs to the group.

5 Experiments

In the following, we will evaluate the capability of processing large scale Lasso-type algorithms among the three platforms, Graphlab [12], Spark [24], and Petuum [4]. In order to give the reader a guide of how to write distributed parallel codes under big data platform, In addition to give a performance comparison, we will also focus on the easy-of-use among the platforms, we will show the differences and difficulties while implementing Lasso-type algorithms on platforms, and treat programmability as an important merit for the performance of platforms.

5.1 Datasets

Since the available real-world datasets are usually relative small, we generate several synthetic datasets to evaluate in the experiment. The datasets are generated as follows [7]: constructing different pairs (N, d) for data matrix $\mathbf{X}$ based on the sparsity and assigning the values of $\mathbf{X}$ following the standard Gaussian distribution. The original weight vector $\mathbf{w}_o$ is generated by randomly sampling non-zero features, where the values of the elements of the sampled entities are generated following the standard Gaussian distribution. The final response $\mathbf{y}$ is computed by $\mathbf{y} = \mathbf{X}\mathbf{w}_o + \epsilon$. For group Lasso, we construct the group index before sampling the original weight vector $\mathbf{w}_o$, and conduct samples according to the group index. Table 1 summarizes the statistics of the generated datasets for Lasso and Group Lasso.

5.2 Experimental Setup

All algorithms are run in multi-core servers with VMware ESXi on each server. For each virtual machine, we configured 4 cores (2.5 GHz) and 8 GB RAM running on Ubuntu 14.04. There are 8 VMs in our experimental cluster. All nodes are connected with each other by 1 Gigabit Ethernet. The version of the GCC is 4.8, and the version of JDK is 1.7. The version of Petuum is 0.21 and the parameter of staleness is set to 8. The version of GraphLab is 2.2. The version of Spark is 1.0. They are downloaded from their github repositories.

5.3 Experimental Results

Table 2 records the execution time in seconds required in the compared platforms. We have the following observations:

Table 1: Statistics of the synthetic datasets

Dataset	Lasso Datasets					Group Lasso Datasets				
Type	SD1	SD2	SD3	SD4	SD5	GD1	GD2	GD3	GD4	GD5
Size	5K*10K	5K*100K	5K*100K	5K*1M	5K*1M	5K*10K	5K*10K	5K*100K	5K*100K	5K*100K
Density	2%	2%	.2%	20%	100%	.2%	.2%	.2%	2%	20%
Groups						100	1K	1K	1K	10K

Table 2: Execution time in seconds for the three compared platforms

	Lasso					Group Lasso				
Platform	SD1	SD2	SD3	SD4	SD5	GD1	GD2	GD3	GD4	GD5
GraphaLab	62.3	242.7	354.3	1023.2	1632.9	16.3	62.3	109.7	157.3	579.2
Petuum	26.3	89.5	127.2	785.4	967.6	14.9	26.4	62.0	109.2	231.7
Spark	482.0	565.8	1563.6	2453.9	4512.6	417.4	489.4	529.4	709.9	1781.9

- The execution time in the three platforms increases gradually as the size of the datasets increases. All the compared platforms can handle these huge datasets, e.g., with one thousand samples, one million features, and five billion non-zero entities.
- Petuum is the most efficient as on average it only takes about half of the time consuming by Graphlab in Lasso and Group Lasso, while the time required by Spark is about three to seven times of that of Graphlab for Lasso and it is about three to twenty five times for Group Lasso. The reason lies that the parameter-server model of Petuum and memory-based storage are well suited for Stochastic Coordinate Descent (SCD), which repeatedly update the shared parameters in high frequency. In addition, the Stable Synchronous Parallel (SSP) can significantly accelerate the iteration speed. One thing to remain that, with the staleness or even the parallel threads increase, the convergence might not be guaranteed, since accumulation of the deviation brought from inconsistent updates.
- Graphlab's performance depends largely on your code design. Generally, Graphlab is quite sensitive to the number of groups as the time required can be increased dramatically. The increase of group vertex will bring a significant increase of the edges, which may increase the burden of computation.
- The maintenance of high dimension shared variables cost too much communication effort in Spark. Moreover, SCD only update a small portion of shared variables, which is also inefficient in Spark's iteration, because it could be the same cost as to update the whole variables for Spark. As a result, Spark's performance is unsatisfactory in our experiment. We also observe that Spark is not suitable to solve small scale datasets as communication time between drivers and workers dominates the computation time. Spark is not very sensitive to the dimension of datasets because of the support of sparse data in Spark v1.0.

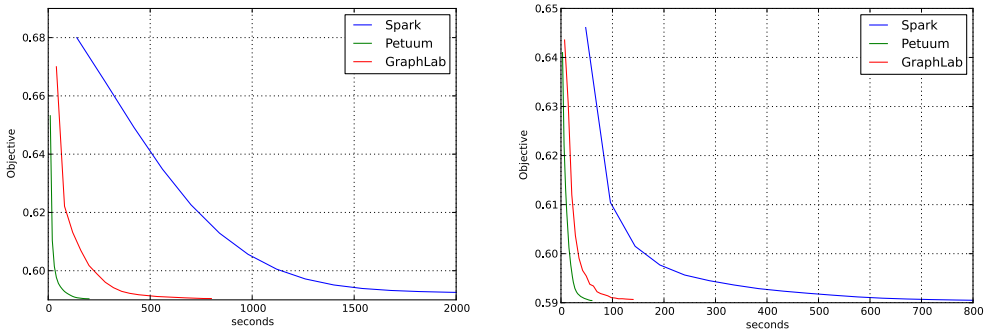
Fig. 2 shows the convergence curve of Lasso and Group Lasso on the three compared platforms. This again verifies that Petuum is much efficient than Graphlab while Spark is the lowest among three compared platforms.

Convenience of implementation. We evaluate the easy of implementation by examining the number of codes written for Lasso and Group Lasso for the compared platforms. From Table 3, we know that Spark enjoys much less code and is much convenient in deployment than that of Petuum and GraphLab. This owes to the implementation of scala-based functional style and the Driver-Worker runtime pattern. Because Petuum is still in early-stage development, you have to cope with lots of scripts to deploy your program on a cluster. What's worse, you need to distribute your complied code manually to the rest of your nodes in the cluster.

Table 3: Lines of codes written to implement Lasso and Group Lasso in the 3 compared platforms

Platform	Lasso	Group Lasso
GraphLab	389	465
Petuum	433	529
Spark	129	158

Ninja performance gap. Ninja performance gap is the performance gap between naively written parallelism unaware code and best-optimized code on modern multi-/many-core processors [17]. When we implement the first version of Lasso in GraphLab, we use edge in the data matrix, it works when the data size is not large, but it quickly runs out of memory when we load a relative big graph. We then redesign the implantation and find a another method to store the data, i.e., storing each column of the data as a vertex. This can significantly reduce the memory consumption and successfully load the entire graph. For Petuum, as it is developed in very early staged, there are lots of optimizations you can involve in your code, such as caching your update value for SSPtable to perform batch update or caching them until it meets some conditions, which could reduce SSPtable operation frequency. In Spark the Ninja performance can be effectively avoided, because the Map-Reduce programming model is quite straightforward, and the abstract data structure RDD is collection alike which are familiar to the developers.



(a) Lasso in SD3 (b) Group Lasso in GD3

Figure 2: Convergence comparison of Lasso and Group Lasso on Graphlab, Spark, and Petuum.

6 Conclusion

In this paper, we choose Lasso and Group Lasso, two typical and famous Lasso-type algorithms and compare the performance on three merited machine learning platforms, GraphLab, Spark, and Petuum, through generated huge-large datasets. The extensive results demonstrate that Petuum is a very efficient machine learning platform for executing Lasso and Group Lasso while Spark is quite slow in the execution. However, Spark has the advantages of easy implementation, which can avoid the Ninja performance gap.

We will continue our effort in the following interesting directions. First, we plan to deploy the implementation on truly big data real-world applications. Second, we intend to implement more Lasso-type algorithms and release the codes for advancing the development of the research community. Third, we will consider how to dispatch the data based on the status of machines to further improve different platforms.

Acknowledgment

The work described in this paper was fully supported by the National Grand Fundamental Research 973 Program of China (No. 2014CB340401 and No. 2014CB340405), the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK 413212 and CUHK 415113), and Microsoft Research Asia Regional Seed Fund in Big Data Research (Grant No. FY13-RES-SPONSOR-036).

References

[1] Léon Bottou. Online algorithms and stochastic approximations. In David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998.

[2] Joseph K. Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *ICML*, pages 321–328, 2011.

[3] Avery Ching. Large-scale graph processing infrastruction on hadoop. In *Hadoop Summit*, Santa Clara, USA:[sn], 2011.

[4] Wei Dai, Jinliang Wei, Xun Zheng, Jin Kyu Kim, Seunghak Lee, Junming Yin, Qirong Ho, and Eric P. Xing. Petuum: A framework for iterative-convergent distributed ml. *CoRR*, abs/1312.7651, 2013.

[5] John Duchi and Yoram Singer. Efficient learning using forward-backward splitting. *Journal of Machine Learning Research*, 10:2873–2898, 2009.

[6] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. A note on the group lasso and a sparse group lasso, 2010.

[7] Laurent El Ghaoui, Vivian Viallon, and Tarek Rabbani. Safe feature elimination in sparse supervised learning. *CoRR*, abs/1009.3515, 2010.

[8] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

[9] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[10] Tim Kraska, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, and Michael I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

[11] John Langford, Lihong Li, and Tong Zhang. Sparse online learning via truncated gradient. *Journal of Machine Learning Research*, 10:777–801, 2009.

[12] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[13] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *PODC*, page 6, 2009.

[14] Yu Nesterov. Introductory lectures on convex optimization: A basic course. Kluwer Academic Publishers, 2003.

[15] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. Mahout in action. In *Manning*. 2011.

[16] Zhimin Peng, Ming Yan, and Wotao Yin. Parallel and distributed sparse optimization. In *ACSSC*, pages 659–646, 2013.

[17] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *ISCA*, pages 440–451, 2012.

[18] Sangwon Seo, Edward J. Yoon, Jae-Hong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *CloudCom*, pages 721–726, 2010.

[19] Robert Tibshirani. Regression shrinkage and selection via the lasso. *J. Roy. Statist. Soc. Ser. B*, 58(1):267–288, 1996.

[20] Lin Xiao. Dual averaging method for regularized stochastic learning and online optimization. *Journal of Machine Learning Research*, 11:2543–2596, October 2010.

[21] Haiqin Yang, Zenglin Xu, Irwin King, and Michael R. Lyu. Online learning for group lasso. In *ICML*, pages 1191–1198, Haifa, Israel, 2010.

[22] M. Yuan and Y. Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society, Series B*, 68(1):49–67, 2006.

[23] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[24] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[25] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical, Society B*, 67:301–320, 2005.