Robust Large-Scale Machine Learning in the Cloud

Steffen Rendle, Dennis Fetterly, Eugene J. Shekita, Bor-yiing Su
Google Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
{srendle,fetterly,shekita,boryiingsu}@google.com

ABSTRACT

The convergence behavior of many distributed machine learning (ML) algorithms can be sensitive to the number of machines being used or to changes in the computing environment. As a result, scaling to a large number of machines can be challenging. In this paper, we describe a new scalable coordinate descent (SCD) algorithm for generalized linear models whose convergence behavior is always the same, regardless of how much SCD is scaled out and regardless of the computing environment. This makes SCD highly robust and enables it to scale to massive datasets on low-cost commodity servers. Experimental results on a real advertising dataset in Google are used to demonstrate SCD's cost effectiveness and scalability. Using Google's internal cloud, we show that SCD can provide near linear scaling using thousands of cores for 1 trillion training examples on a petabyte of compressed data. This represents 10,000x more training examples than the 'large-scale' Netflix prize dataset. We also show that SCD can learn a model for 20 billion training examples in two hours for about $10.

Keywords

Distributed Machine Learning; Coordinate Descent; Linear Regression

1. INTRODUCTION

Although distributed machine learning (ML) algorithms have been extensively studied [22, 12, 10, 9], scaling to a large number of machines can still be challenging. Most fast converging single machine algorithms update model parameters at a very high rate which makes them hard to distribute without compromises. For example, single-machine stochastic gradient descent (SGD) [7] updates model parameters after processing each training example, while coordinate descent (CD) [18] updates them after processing a single feature. Common approaches to distribute SGD or CD break the basic flow of the single-machine algorithm by letting updates occur with some delay or by batching [16, 22, 8, 23, 12]. However, this changes the convergence behavior of the algorithm, making it sensitive to the number of machines as well to the computing environment. As a result, scaling can become non-linear and the benefit from adding more machines can tail off early [16].

Because of these scaling problems, some authors [9] have argued that it is better to scale out ML algorithms using just a few 'fat' servers with lots of memory, networking cards, and GPUs. While this may be an appealing approach for some problems, it has obvious scaling limitations in terms of I/O bandwidth. Generally speaking, it is also more expensive than scaling out using low-cost commodity servers. GPUs in particular are not always a cost effective solution for sparse datasets.

In this paper, we describe a new scalable coordinate descent (SCD) algorithm for generalized linear models that does not suffer from the scaling problems outlined above. SCD is highly robust, having the same convergence behavior regardless of how much it is scaled out and regardless of the computing environment. This allows SCD to scale to thousands of cores and makes it well suited for running in a cloud environment with low-cost commodity servers. A key observation of our work is that, by using a natural partitioning of parameters into blocks, updates can be performed in parallel a block at a time without compromising convergence. In fact, on many real-world problems, SCD has the same convergence behavior as the popular single-machine coordinate descent algorithm.

In addition to the SCD algorithm, we describe a distributed system that addresses the specific challenges of scaling SCD in a cloud computing environment. Straggler handling ends up being the biggest challenge to achieving linear scaling with SCD. Experimental results using a real advertising dataset in Google are used to demonstrate SCD's cost effectiveness and scalability.

To summarize, the main contributions of this paper are as follows:

- We describe a new scalable coordinate descent algorithm (SCD) whose convergence behavior is always the same, regardless of how much SCD is scaled out and regardless of the computing environment.
- We describe a distributed system for SCD that can provide near linear scaling using thousands of cores.
- We provide experimental results from Google's internal cloud using low-cost preemptible virtual machines to show that SCD can solve large-scale ML problems with 1 trillion training examples.

KDD '16 August 13-17, 2016, San Francisco, CA, USA

ACM ISBN 978-1-4503-4232-2/16/08.
DOI: http://dx.doi.org/10.1145/2939672.2939790


2. PROBLEM STATEMENT

In this section, we discuss the machine learning and computing environments that are the focus of our work. Then we state the main goals of our work.

2.1 Machine Learning Environment

We focus on generalized linear models with large sparse datasets. Linear models are popular in industry for large-scale prediction tasks [24, 21, 12, 20] because of their high prediction quality and interpretability.

2.1.1 Linear Model

Let $\boldsymbol{x} \in \mathbb{R}^p$ be a feature vector and y an unknown variable of interest, e.g. click/non-click, rating, etc. A linear model $\hat{y} : \mathbb{R}^p \to \mathbb{R}$ assumes a linear dependency between the input feature vector $\mathbb{R}$ and the variable of interest y

$$\hat{y}(\boldsymbol{x}) = \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle = \sum_{i=1}^{p} \theta_i x_i \tag{1}$$

where $\boldsymbol{\theta} \in \mathbb{R}^p$ are the model parameters that parameterize the dependency. Using a link function, linear models can be generalized (GLM) [18] to prediction tasks such as logistic regression (binary classification), Poisson regression (count data), etc. Note that even though a linear model assumes a linear dependency in $\boldsymbol{x}$, non-linearity can be introduced by preprocessing $\boldsymbol{x}$, e.g., using polynomial expansions or log-transformations. Preprocessing allows linear models to be very expressive and highly non-linear in the original space. The remainder of the paper assumes that a proper preprocessing has been done and that $\boldsymbol{x}$ is the feature vector after transformation.

2.1.2 Optimization Task

ML algorithms learn the values of the model parameters $\boldsymbol{\theta}$ given a set S of labeled *training examples* $(\boldsymbol{x}, y)$. In matrix notation, $X \in \mathbb{R}^{|S| \times p}$ is the *design matrix* with labels $\boldsymbol{y} \in \mathbb{R}^{|S|}$. The optimization task is to find the model parameters that minimize a loss:

$$\underset{\boldsymbol{\theta} \in \mathbb{R}^p}{\operatorname{argmin}} \sum_{(\boldsymbol{x}, y) \in S} l(y, \hat{y}(\boldsymbol{x})) + \lambda ||\boldsymbol{\theta}||^2 \tag{2}$$

where l is a loss function that depends on the optimization task, e.g., using a squared loss $l(y, \hat{y}) := (y - \hat{y})^2$, or using a logistic loss $l(y, \hat{y}) := -y\hat{y} + \ln(1 + exp(\hat{y}))$. λ is a regularization value for generalization. For simplicity, we will assume L2 regularization, but all the results in this paper can be extended to L1 regularization.

2.1.3 Large-Scale Learning and Sparse Datasets

We focus on datasets with potentially trillions of training examples, that is, $|S| \in \mathcal{O}(10^{12})$. Consequently, the training data does not fit in memory. We also focus on models with billions of features, that is, $p \in \mathcal{O}(10^9)$. The model will usually fit in memory, but our proposed solution can also handle models that are larger than memory.

Typically, a training dataset with lots of features will also have a lot of sparsity. There might be billions of features, but only a small number (e.g., hundreds) of non-zero features per example. The reason for high sparsity is usually because of one-hot encoded categorical variables. For example, part of the feature vector $\boldsymbol{x}$ might contain a country variable that is represented by a binary vector with as many entries as there are countries. But each binary vector will contain only one non-zero entry, corresponding to the selected country. For variables like countries, there might be only hundreds of entries, whereas for variables like video ids or user ids, there might be billions of entries leading to very high sparsity. Note that our approach is not limited to categorical variables but supports any kind of real-valued feature vector $\boldsymbol{x} \in \mathbb{R}^p$.

Let $N_Z(\boldsymbol{x})$ be the number of non-zeros in a feature vector $\boldsymbol{x}$ or design matrix X. ML algorithms can make use of the sparsity and usually have a runtime in the order of $N_Z(X)$ instead of $|S| \times p$.

2.2 Computing Environment

Cloud computing has become a cost effective solution for many applications, particularly for batch applications that need to scale with their data size. Although our work can be applied to any large-scale distributed computing environment, we focus on a cloud with the following properties:

- *Shared Machines*: To increase utilization, each physical machine can be shared by multiple virtual machines (VMs).
- *Distributed File System*: A fault tolerant distributed file system (DFS), such as the Google File System [19], is available to store data. In addition to training data, the DFS can be used to save any application state needed for fault tolerance.
- *Preemptible VMs*: The cloud scheduler is free to preempt a low-priority VM in favor of a higher-priority one. All of a VM's state is lost when it is preempted. However, a notification is sent to a VM before it is preempted. This includes a grace period that is long enough for the application to save its state to the DFS for fault tolerance, if necessary.
- *Machine Failures*: Physical machines can fail without any notification. Failures are assumed to be rare, but do happen. Consequently, long-running jobs need to checkpoint their state to the DFS for fault tolerance.

Other than preemptible VMs, these are fairly standard properties for most modern clouds. Preemptible VMs are available on Google's Cloud Platform (GCP) [1] and as "spot instances" from Amazon Web Services (AWS) [2]. They are an attractive way to lower costs for long-running batch jobs. For example, a preemptible VM is about 70% less expensive than a standard VM on GCP.

In terms of cost, a cloud with the above properties is particularly appealing for running distributed ML jobs. However, scaling a distributed ML algorithm on such a cloud becomes even more challenging. This is because there can be wide variations in machine performance because of contention for physical resources such as CPU and networking, or contention for software resources such as access to the DFS. Preemptions also create headaches. All these things can negatively impact convergence behavior.

2.3 Goals of Distributed Learning

In terms of the learning algorithm and its distributed system design, the main goals of our work are as follows:

1. *Robust Distribution*: The algorithm's convergence behavior should always be the same, regardless of how

much it is scaled out and regardless of the computing environment.

2. *Linear Scale-out*: If the number of training examples $|S|$ as well as the number of computing resources grow by a factor of M, the time to solve the learning problem should stay constant. This is also known as "weak scaling".

3. *Linear Speed-up*: If the number of computing resources grows by a factor of M, the same learning problem should be solved M times faster in wall time. This is also known as "strong scaling".

The learning algorithm and distributed system that we describe in this paper satisfies all three of these goals for a wide range of problem sizes and scaling factors.

3. LEARNING ALGORITHMS

In this section, we review the popular coordinate descent (CD) algorithm [18, 26, 11], which is inherently restricted to a single machine. Then we describe our new scalable coordinate descent (SCD) algorithm. Note that this section focuses on the abstract SCD algorithm. In Section 4, we describe SCD's distributed system design and implementation.

3.1 Coordinate Descent (CD)

CD looks at a single model parameter or *coordinate* θ_j at a time, assuming that the values of other model parameters $\boldsymbol{\theta} \setminus \theta_j$ are known and fixed. Under this assumption, the optimum for θ_j has a closed form solution:

$$\theta_j^* = \frac{T_j}{T_j' + \lambda}. \tag{3}$$

For linear regression, the *sufficient statistics* T_j and T_j' are:

$$T_j := \sum_{(\boldsymbol{x},y)\in S} x_j \left(y - \sum_{i \neq j} \theta_i \, x_i \right), \quad T_j' := \sum_{(\boldsymbol{x},y)\in S} x_j^2. \tag{4}$$

To simplify the discussion, we use linear regression in all the algorithm descriptions, but other loss functions can be handled similarly, such as logistic regression using a second-order Taylor expansion [18]. For sparse data, the computation for T_j and T_j' can be accelerated by only iterating over examples $\boldsymbol{x}$ where $x_j \neq 0$.

From this analysis follows Algorithm 1. As shown, CD iterates over one model parameter θ_j at a time. In each iteration, the sufficient statistics T_j and T_j' are aggregated (lines 8-11), and the local optimum θ_j^* is calculated (line 12). To make the cost of computing T_j independent of the other features $\boldsymbol{x} \setminus x_j$, the current prediction $\hat{y}(\boldsymbol{x})$ of each example is precomputed. Using $\hat{y}(\boldsymbol{x})$, the computation of T_j simplifies to:

$$T_j = \sum_{(\boldsymbol{x},y)\in S} x_j \; (y - \hat{y}(\boldsymbol{x}) + \theta_j \, x_j) \, . \tag{5}$$

The precomputed prediction $\hat{y}(\boldsymbol{x})$ for each example also needs to be updated each iteration (lines 13-15). Finally, the model parameter is updated with its local optimum (line 16).

CD is known as a fast-converging algorithm for problems that fit on a single machine [11]. Variations on Algorithm 1 include cyclic coordinate descent [18] and stochastic coordinate descent [26].

Algorithm 1 Coordinate Descent

```
1: procedure CD(S)
2:     ŷ ← (0, ..., 0)                    ▷ Precomputed predictions
3:     θ ← (0, ..., 0)
4:     repeat
5:         for j ∈ {1, ..., p} do
6:             T_j ← 0
7:             T'_j ← 0
8:             for (x, y) ∈ S where x_j ≠ 0 do
9:                 T_j ← T_j + x_j (y − ŷ(x) + θ_j x_j)
10:                T'_j ← T'_j + x_j^2
11:            end for
12:            θ*_j ← T_j / (T'_j + λ)
13:            for (x, y) ∈ S where x_j ≠ 0 do
14:                ŷ(x) ← ŷ(x) + x_j(θ*_j − θ_j)
15:            end for
16:            θ_j ← θ*_j
17:        end for
18:    until converged
19: end procedure
```

3.1.1 CD Analysis

Recall that CD processes one parameter at a time. We define a CD *iteration* as the update of one parameter and an *epoch* as a pass over all parameters. The amount of computation per iteration is on average $\mathcal{O}(N_Z(X)/p)$ and per epoch $\mathcal{O}(N_Z(X))$.

A straightforward distributed version of Algorithm 1 would require a system-wide synchronization *barrier* just after aggregating the sufficient statistics T_j and T_j' (lines 8-11). In general, distributing work only pays off if the overhead in terms of communication and barriers is small compared to the amount of work that is parallelized. However, with a large number p of parameters and a high sparsity ($N_Z(\boldsymbol{x}) \ll p$) per feature vector $\boldsymbol{x}$, there is relatively little work to be done in an iteration of CD. Consequently, a straightforward distributed version of Algorithm 1 would neither scale-out nor speed-up.

3.2 Scalable Coordinate Descent (SCD)

SCD increases the amount of work per iteration by carefully partitioning the parameters and by iterating a block of parameters at a time. Scaling is achieved by computing the sufficient statistics over examples in parallel across machines. Robustness is achieved by keeping the partition fixed and independent of the number of machines. Fast convergence is achieved by a clever partitioning of parameters.

We start with discussing the algorithm, assuming a partition is given and later show how to choose a good partition.

3.2.1 Robust and Scalable Algorithm

Let $\mathcal{P}$ be a partition of feature or parameter indices $\{1, .., p\}$. We refer to $B \in \mathcal{P}$ as a *block*. We denote the subset of parameters $\boldsymbol{\theta}$ associated with the block B by $\boldsymbol{\theta}^B$, the subvector of the feature vector $\boldsymbol{x}$ by $\boldsymbol{x}^B$, and the submatrix of the design matrix X by X^B. The basic flow of SCD (see Algorithm 2) is similar to CD, but instead of iterating over one model parameter at a time, SCD iterates over one block B of parameters at a time (line 7). In each iteration, partial sums for the sufficient statistics T and T' are computed in parallel (lines 8-13). These are aggregated across machines

Algorithm 2 Scalable Coordinate Descent

```
 1: procedure SCD(S, P)
 2:     θ ← (0, ..., 0)
 3:     ŷ ← (0, ..., 0)                    ▷ Precomputed predictions
 4:     repeat
 5:         T ← (0, ..., 0)
 6:         T' ← (0, ..., 0)
 7:         B ← SelectBlock(P)
 8:         for (x, y) ∈ S do ▷ In parallel, across machines
 9:             for j ∈ B where x_j ≠ 0 do
10:                 T_j ← T_j + x_j (y − ŷ(x) + θ_j x_j)
11:                 T'_j ← T'_j + x_j^2
12:             end for
13:         end for
14:         Aggregate T and T' across machines.
15:         for j ∈ B do
16:             θ*_j ← (1 − α) θ_j + α T_j / (T'_j + λ)
17:         end for
18:         for (x, y) ∈ S do ▷ In parallel, across machines
19:             for j ∈ B where x_j ≠ 0 do
20:                 ŷ(x) ← ŷ(x) + x_j(θ*_j − θ_j)
21:             end for
22:         end for
23:         for j ∈ B do
24:             θ_j ← θ*_j
25:         end for
26:     until converged
27: end procedure
```

(line 14), and the new value θ^*_j of each parameter in B is calculated (lines 15-17).

Because the sufficient statistics of several features have been computed in parallel, the independence assumptions of the CD update step (eq. 3) are violated. To ensure convergence, we use the common line-search method to find a step size $\alpha \in [0, 1]$ and update each parameter with:

$$\theta^*_j = (1 - \alpha)\,\theta_j + \alpha \frac{T_j}{T'_j + \lambda}. \tag{6}$$

Obviously, the smaller the step size α, the slower the convergence in comparison to CD. In Section 3.2.3, we will show that by using a clever partitioning, SCD can usually take optimal steps, in which case $\alpha = 1$. After the new value has been calculated, the precomputed predictions $\hat{y}(\boldsymbol{x})$ for each example are updated (lines 18-22). Finally, each model parameter in B is updated with its new value (lines 23-25).

In Algorithm 2, it is important to note that parallel execution is done over examples, i.e., *sharding* is by example. This means that each machine is responsible for computing the partial sufficient statistics T and T' for all the parameters $\boldsymbol{\theta}^B$ in the selected block B but only for a subset of examples. Consequently, the more machines, the faster sufficient statistics are computed. As long as the amount of work for a block is large enough, SCD scales with more machines. Moreover, as the partition is fixed and does not change with the number of machines, SCD is robust.

3.2.2 Optimal Updates with Pure Blocks

So far we have discussed the robustness and scalability, now we focus on convergence speed. The key idea is to partition the model parameters into what we call *pure blocks* of independent parameters. We prove that parallel updates within a pure block are equivalent to processing the updates sequentially and consequently, full update steps $\alpha = 1$ can be taken.

Definition 1 (Pure Block). *A block B is* pure *iff the feature subvector $\boldsymbol{x}^B$ of every example $(\boldsymbol{x}, y) \in S$ has at most one non-zero entry:*

$$B \text{ pure} :\Leftrightarrow \forall (\boldsymbol{x}, y) \in S : N_Z(\boldsymbol{x}^B) \leq 1.$$

Similarly, a partition $\mathcal{P}$ of $\{1, \ldots, p\}$ is *pure* if all the blocks in the partition $B \in \mathcal{P}$ are pure.

Lemma 1 (Independence of Updates). *All parameter updates for a pure block are independent.*

Proof. The well known closed form solution of a regularized least-squares problem is $\boldsymbol{\theta} = (X^t\,X + \lambda\,I)^{-1}\,X\,\boldsymbol{y}$. Let $\tilde{B} := \{1, \ldots, p\} \setminus B$. Consequently, the closed form solution for the parameters of the block B is $\boldsymbol{\theta}^B = ((X^B)^t\,X^B + \lambda\,I)^{-1}\,X^B\,(\boldsymbol{y} - X^{\tilde{B}}\,\boldsymbol{\theta}^{\tilde{B}})$. Because B is pure, the Gramian $(X^B)^t\,X^B$ is a diagonal matrix $\text{diag}(T'_1, T'_2, \ldots)$. It follows that the standard CD update (eq. 3) is identical to the joined closed form solution for all parameters $\boldsymbol{\theta}^B$ of the block. This means that, within a pure block B, the update of one parameter does not influence the update of another parameter. □

The upshot of Lemma 1 is that, within a pure block, processing parameter updates in parallel is equivalent to processing the updates sequentially. In other words, SCD on pure partitions is equivalent to CD while allowing scaling.

Note that purity and robustness are two different concepts. SCD is robust no matter if a pure or impure partition has been chosen. Pure partitions are preferred because optimal step sizes can be taken.

3.2.3 Generating a Partition

Now we describe how to construct a good partition for a training set $S = (X, \boldsymbol{y})$. Based on our analysis, a good partition has two properties: (1) pure blocks for convergence speed and (2) a large amount of work per block for system speed. Both properties can easily be met individually but at first glance fulfilling both looks hard. However, most real-world datasets have a natural partition with good properties. As mentioned in Section 2.1.3, input feature vectors $\boldsymbol{x}$ are usually generated from several variables. We suggest to partition the features by variable. For each variable v, a block is constructed that corresponds to the features generated from v. The properties of this partition are: (1) For many variable types the resulting blocks are pure. This includes categorical variables, a cross product of categorical variables, bucketized numerical variables, dense numerical variables, etc. (2) For these variables each resulting block is of equal computational complexity with $N_Z(X^B) = |S|$. To summarize, a natural partition by underlying variable often has all desired properties. Many real-world datasets are composed of these variable types. In fact, these are the most common variable types in regression datasets at Google.

Some datasets contain variable types with suboptimal properties. An example is a set-valued variable, such as the genres of a movie, where several genres can be assigned to one movie. In this case a split by variable would result in an *impure* block. We propose splitting the features of such a

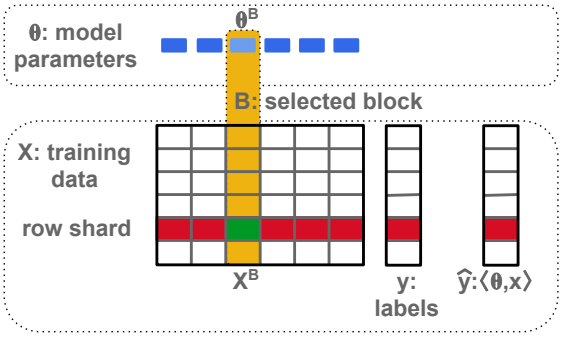

Figure 1: Data is sharded in both the feature and example dimensions. Each cell in the grid is stored as a separate file that holds the data of many examples for one block.

variable v into $N_Z(X^v)/|S|$ blocks. Splitting the features this way means there will be one active feature per block on average. However, there can still be training examples with more than one active feature. That means the blocks of a set variable can be impure and a step size $\alpha < 1$ is used.

3.2.4 SCD Analysis

An SCD *iteration* processes one block B of model parameters at a time. The computational complexity for an iteration is $\mathcal{O}(N_z(X^B))$. Updating all parameters, that is, one SCD *epoch*, takes $|\mathcal{P}|$ iterations, so the overall complexity of one epoch is $\mathcal{O}(N_Z(X))$. Although it is not shown in Algorithm 2, a small number of synchronization barriers are required in each SCD iteration. More will be said about this in Section 4 when SCD's implementation is described. Consequently, the number of barriers per SCD epoch is $\mathcal{O}(|\mathcal{P}|)$. Compared to CD, SCD decreases the number of barriers from $\mathcal{O}(p)$ to $\mathcal{O}(|\mathcal{P}|)$. On a large sparse dataset, this can mean hundreds of barriers instead of billions of barriers. The decrease in the number of barriers, along with an increase in the amount of work per barrier, is what makes it feasible to distribute SCD over a large number of machines.

The SCD algorithm meets the three goals of Section 2.3. First, SCD is robust in the sense that it performs exactly the same update computations no matter how many machines it uses. Moreover, the outcome of each SCD update step is deterministic and unaffected by the computing environment. Second, the increase of work per iteration allows SCD to scale linearly – at least in theory. In practice, various system overheads and stragglers become a limiting factor. More will be said about this shortly.

4. DISTRIBUTED SCD SYSTEM

In this section, we describe a distributed system that addresses the specific challenges of scaling SCD in a cloud computing environment.

4.1 Storage Format

As shown in Figure 1, the training data $S = (X, \boldsymbol{y})$ for SCD is sharded in both the feature and example dimensions. It is sharded in the feature dimension using the block partition $\mathcal{P}$ (i.e., the sharding follows $\mathcal{P}$), while it is sharded in the example dimension using *row sharding*. Feature sharding enables SCD to process the data corresponding to one block independently of the other blocks, while example sharding enables SCD to process different shards of X^B in parallel.

The remaining data is sharded as follows. The model parameters $\boldsymbol{\theta}$ are sharded in just the feature dimension (following $\mathcal{P}$), while the labels $\boldsymbol{y}$ and predictions $\hat{\boldsymbol{y}}$ are sharded in just the example dimension (following the row sharding).

As noted in Figure 1, each cell in the grid is stored in a separate file. For example, if there were $p = 10^9$ features partitioned into $|\mathcal{P}| = 100$ blocks and $|S| = 10^{10}$ examples sharded in 10^4 rows with 10^6 examples each, then the design matrix X would be stored in $10^4 * 100 = 10^6$ files.

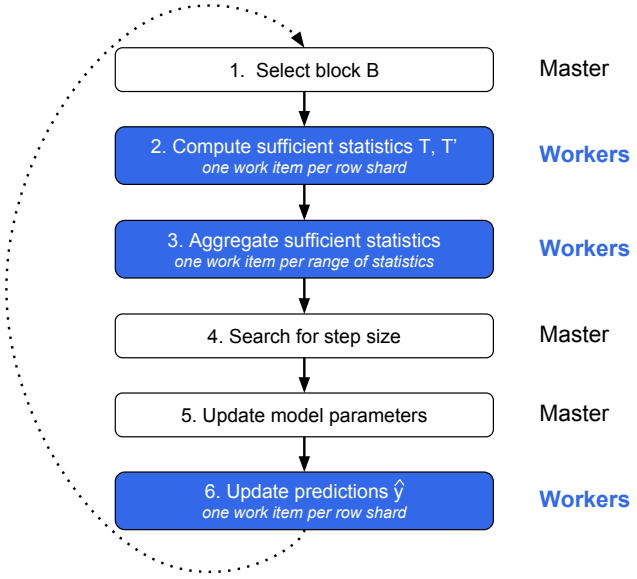

Figure 2: The system flow for one iteration of the SCD algorithm. The computationally expensive steps are distributed over the workers.

4.2 System Architecture

Our system architecture is based on a single *master* and multiple *workers*. The master acts as the orchestrator and is responsible for assigning work, while the workers execute the computationally expensive parts of the SCD algorithm. To assign work, the master hands out *work items* to workers. Each work item corresponds to a small unit of work that usually takes only a few hundred milliseconds to execute. Both the master and workers are multi-threaded to take advantage of multi-core parallelism.

A single master obviously becomes a scaling bottleneck at some point. But as the performance experiments in Section 5 will show, even a single master can scale to hundreds of workers with thousands of cores.

4.3 System Flow

Recall that the main loop of the SCD algorithm iterates over one block of parameters at a time (see Algorithm 2, lines 4-26). One iteration of this loop translates into the following steps:

1. Select block: The master selects a block B based on various heuristics that try to estimate which block will have the largest impact on convergence.

2. Compute sufficient statistics: The workers compute the partial sums of sufficient statistics T and T' over the row shards of block B. These partial sums are stored in the memory of the workers.

3. Aggregate sufficient statistics: The sufficient statistics of the workers are aggregated and sent back to the master.

4. Search for step size: The master chooses a step size according to the line-search method described in Section 3.2.1.

5. Update model parameters: The master updates the model parameters $\boldsymbol{\theta}$ of block B using the chosen step size and the sufficient statistics.

6. Update predictions: The workers update their predictions $\hat{\boldsymbol{y}}$ to reflect the change in the model parameters.

The steps in this flow are depicted in Figure 2. The master executes the steps sequentially, either by performing the step itself (steps 1, 4, and 5) or by assigning work items to the workers for the steps that are computationally expensive (steps 2, 3, and 6). The latter are distributed over the workers and executed in parallel.

The sequential execution of the steps effectively creates a system-wide barrier for the workers after steps 2, 3, and 6. Typically, a whole iteration of SCD takes less than a minute, which means there is a system-wide barrier every few seconds. Handling stragglers, preempted and/or failed workers and executing a system-wide barrier at this rate is challenging. The steps assigned to workers are now described in more detail.

4.3.1 Computing Sufficient Statistics

The workers compute the partial sums for the sufficient statistics over row shards of the selected block (see Algorithm 2, lines 8-13). Each worker is multi-threaded and the threads execute work items in parallel. Each work item refers to one row shard in this case. Given a work item, a worker thread needs the training data, labels $\boldsymbol{y}$, and predictions $\hat{\boldsymbol{y}}$ for the corresponding row shard in order to compute the statistics. These can be found in the worker's cache, in another worker's cache, or in the DFS. Section 4.4.2 will provide more detail about the caches maintained by each worker.

Sufficient statistics are stored in the memory of each worker using a thread-local store for the 100K most-frequent features and a worker-level store (over all threads) for the remaining features. This two-level scheme improves hardware cache locality and also enables lock-free updates [25] of the statistics. Update collisions can occur in the worker-level store, but they have a very low probability of happening since it is only used for infrequent features.

4.3.2 Aggregating Sufficient Statistics

After the partial sums for the sufficient statistics have been computed by the workers, they need to be aggregated across workers (see Algorithm 2, line 14). If each worker sent back its statistics to the master, it would create a scaling bottleneck due to the TCP Incast problem [13]. Instead, the statistics are partitioned into ranges and the aggregation is distributed among the workers (see Figure 3).

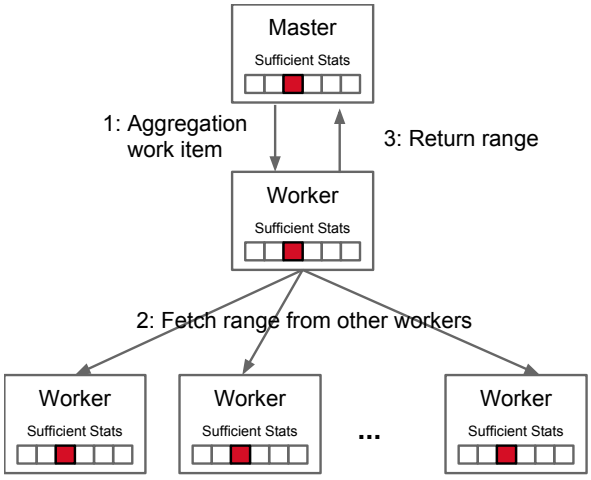

Figure 3: Aggregating Sufficient Statistics: The statistics are partitioned into ranges. For a particular range, an 'aggregator' worker collects the statistics from the other 'leaf' workers and sends the summation to the master. A worker can simultaneously be an aggregator and leaf for different ranges.

Each work item refers to one range of the sufficient statistics in this case. For load balancing, each range is sized to be about 128KB. The worker thread assigned a particular work item becomes the aggregator for the corresponding range of statistics. It collects the statistics for the range from the other workers, performs the aggregation, then sends the aggregated range to the master. Multiple ranges are aggregated in parallel across the workers to take advantage of all the available networking bandwidth.

Let $|W|$ be the number of workers, and let $r > \frac{|B|}{|W|}$ be the number of aggregation ranges, where $|B|$ is the number of features in a block. The size of each range is $\frac{|B|}{r}$. Each range creates $|W|\,\frac{|B|}{r}$ inbound network traffic for the aggregator, $\frac{|B|}{r}$ outbound traffic for each (leaf) worker, and $\frac{|B|}{r}$ inbound traffic for the master. This means that, over all ranges, each worker has $|B|$ inbound and outbound network traffic, while the master has $|B|$ inbound traffic. Consequently, the network traffic of the workers and the master remains constant, no matter how many workers there are.

4.3.3 Updating Predictions

After the master updates the model parameters, the predictions $\hat{\boldsymbol{y}}$ need to be updated to reflect the change in the model parameters (see Algorithm 2, lines 18-22). Each work item refers to one row shard in this case. Given a work item, a worker thread needs the training data and labels $\boldsymbol{y}$ for the corresponding row shard in order to update the predictions. Again, these can be found in the worker's cache, in another worker's cache, or in the DFS.

4.4 Straggler Handling

In theory, SCD should scale perfectly. However, in practice, various overheads and stragglers become a limiting factor. Up to about 1K workers, stragglers are by far the biggest limiting factor. After that, the single master design becomes the limiting factor.

Recall that there is a system-wide barrier for the workers after steps 2, 3, 6 in the SCD system flow (see Figure 2). Therefore, the time it takes to execute these steps is gated by the slowest worker, that is, the worst straggler. It is well known that straggler effects get amplified the more a system is scaled-out [15]. Stragglers are especially challenging in SCD because there is a barrier every few seconds. Moreover, the more SCD is sped-up, the shorter the time between barriers.

Stragglers are usually caused by variations in CPU, networking, or DFS performance. Dynamic load balancing is the main mechanism used to deal with stragglers. It eliminates most of the stragglers caused by CPU and networking performance. However, because of tail latencies, load balancing alone is not sufficient to handle some DFS stragglers. Caching and prefetching are added to deal with these tail latencies.

4.4.1 Dynamic Load Balancing

The DFS enables any worker to work on any work item. Any time a worker thread is idle, it asks the master for a new work item. This results in dynamic load balancing similar to that in a multithreaded program [6], where faster workers get assigned more work items, and vice versa for slower workers. To facilitate load balancing, the system is configured so that there are at least four work items per worker thread.

4.4.2 Caching

In a cloud computing environment with shared access to the DFS, tail latencies can be as bad as several seconds. The best way to mitigate these tail latencies is to avoid the DFS as much as possible using caching. Files for the training data, labels $\boldsymbol{y}$, and predictions $\hat{\boldsymbol{y}}$ are cached in each worker using an LRU eviction strategy. Compressed files are cached in memory and decompressed when a worker needs to access them.

To improve caching, the master tries to assign the same row shard to a given worker in each iteration of its main loop. If a row shard ends up being "stolen" by a different worker because of load balancing, the new worker can avoid accessing the DFS by requesting the row shard's files from the old worker's cache. When this happens, hedged-requests [15] are used to avoid a worker that is slow to respond. A request is sent to both the old worker and the DFS. The first request to finish is the "winner" and the "loser" is canceled.

4.4.3 Prefetching

Using compression, even a small cache for the labels $\boldsymbol{y}$ and predictions $\hat{\boldsymbol{y}}$ can be highly effective. However, caching is less effective for the training data. This is because each iteration of the SCD algorithm accesses the training data associated with a different block B. As a result, the cache hit rate for training data tends to be much lower. This means that workers have to frequently access the DFS for training data. To minimize the impact of tail latencies in this case, training data for the next iteration is prefetched. As a result, even a very slow DFS access has no effect as long as it is shorter than a whole iteration.

4.5 Dealing with VM Preemptions

Recall that a notification is sent to a VM before it is preempted. This includes a grace period that is long enough for the application to save its state to the DFS for fault tolerance, if necessary. We use this grace period to drain a worker that will be preempted. When a worker is notified that it will be preempted, it simply stops asking for new work items. As a result, other workers end up stealing all the preempted worker's row shards and associated data.

The master is usually configured using a standard VM to prevent preemptions. But even if it is configured with a preemptible VM, there is only one master and many workers, so the chances of it being preempted are low. If the master is preempted, it is treated as a machine failure (see below).

4.6 Dealing with Machine Failures

Machine failures are very rare but do happen. The predictions $\hat{\boldsymbol{y}}$ for a row shard are only written to the DFS when they are evicted from a worker's cache. Consequently, predictions can be lost if a worker fails. When this happens, a recovery phase orchestrated by the master simply recomputes the lost predictions.

Of course, the master can also fail. At the end of each iteration, the master takes a checkpoint of its state to the DFS which includes the current value of the model parameters $\boldsymbol{\theta}$. If the master fails and is restarted, it uses the last checkpoint to recovery its state and resume from where it left off.

5. EXPERIMENTS

In this section, we investigate the performance of SCD on a large-scale advertising dataset. The dataset has 1.7 billion parameters ($p = 1.7{*}10^9$) with a pure partition in $|\mathcal{P}| = 194$ blocks. We experimented with several scales of the dataset from 20 billion examples with 4 trillion non-zero elements in the design matrix up to one trillion examples ($|S| = 10^{12}$) and 200 trillion non-zero elements ($N_Z(X) = 2 * 10^{14}$).

CD is considered a fast solver for linear models (e.g., [11, 18]) and is a popular choice for single machine implementations, e.g., LIBLINEAR [3] or glmnet [4]. As shown in Section 3.2.2, on pure partitions, SCD produces the same models with an identical convergence behavior as CD and thus, we focus on runtime and scaling questions. In particular, we investigate the scale-out and speed-up of SCD.

All experiments are run in the internal Google cloud [27] using low-priority, preemptible VMs. Each VM uses 8 cores and less than 30GB of memory. A comparable preemptible VM in Google's external cloud is the *n1-standard-8* with an hourly price of $0.12. The workers are overthreaded by a ratio of 2:1 to hide I/O latency.

5.1 Scale-out

As defined in Section 2.3, scale-out refers to a system's behavior when the problem as well as the number of machines grows. We experimented with $M \in \{1, 2, 4, 8, 16, 32, 50\}$ scales of the dataset and increased the number of machines accordingly by the same factor M. For each of the M scale-out variants, we ran SCD for one epoch, i.e., 194 iterations, and report the average iteration time. The baseline 1x experiment uses 20 machines for 20 billion training examples, the 2x scale-out 40 machines for 40 billion training examples, etc. Note that on the 50x scale, the dataset consists of 1 trillion examples which is 10,000x more data points than the 'large-scale' Netflix prize dataset [5]. The compressed 50x dataset takes about one petabyte on the DFS including standard replication. Figure 4 shows the scale-out behavior

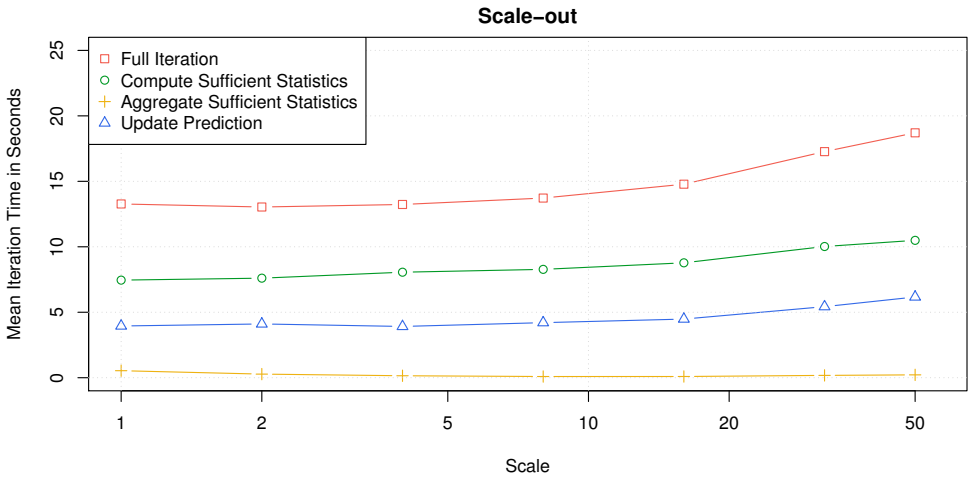

Figure 4: Scale-out behavior of SCD: 1x corresponds to 20 billion training examples and 20 machines, 2x to 40 billion examples and 40 machines, and 50x to 1 trillion examples and 1000 machines. The average total time for an iteration as well as the average time for the main phases of the SCD algorithm are shown.

of SCD. Compared to a perfect linear scaling, SCD shows less than 10% degradation for a 16x (=1600%) scale-out and about 35% degradation for a 50x scale-out. One of the main reasons for the degradation are stragglers caused by the very short barriers of SCD. The larger the data set, and consequently the more files, the higher the chance to hit a severe straggler. E.g., in the *update prediction* phase, the system orchestrates a thousand workers with 8000 cores and hundreds of thousands of work items in about 6 seconds. We conclude that SCD allows near linear scale-out to extremely large datasets.

5.2 Speed-up

The second gain from a distributed system is speed-up, i.e., increasing the number of machines while the problem size stays the same. We study the speed-up of our system on 20 billion training examples and vary the number of machines from 1 to 128 workers. Again, we run each variant of the system for one epoch and report the mean iteration time. As SCD is robust, every configuration learns exactly the same model.

The left plot of Figure 5 shows the average iteration time versus the number of workers. As can be seen, SCD provides a close to linear speed-up behavior. Actually, all configurations outperform the theoretical linear speed-up of a single worker. The reason for the super-linear speed-up is that the amount of data that can be cached increases with more workers. For example, if one worker has 1 GB of memory for caching label data and there are 10 GB of label files, then with one worker at most 10% of the label requests are cache hits, whereas with 32 workers, overall 32 GB of labels can be cached, which results in a much higher cache hit rate. In Figure 5, the super-linear speed-up has its optimum with 16 to 32 workers and moves closer to a linear speed-up with more workers. This is expected because the more machines, the faster the iterations, the shorter the barriers and consequently, the stronger straggler effects. For example, having a straggler of 1 second might not have any effect on a run with 32 machines because barriers are every 3 seconds, whereas for 64 machines, where a perfect speed-up means 1.5 second barriers, the same straggler has more impact. More aggressive prefetching, e.g., two iterations ahead, might solve this tail latency issue.

Speeding-up a system implies getting the results faster when using more resources. If the cost for a resource is constant as in common cloud environments, a theoretical linear speed-up implies getting the result faster with exactly the same cost. In reality, speed-ups are not exactly linear and additional costs can occur. The right plot of Figure 5 shows the time vs. cost trade-off for speeding up the SCD algorithm. The number of resources was varied from 1 to 128 workers and the plot shows the time to converge, i.e., in this case running SCD for five epochs, vs. the costs of the machines. For instance, running SCD for five epochs with 128 machines takes about one hour whereas the time with one worker is about 140 hours. The cost for running 128 workers for about one hour is about $16 whereas the cost for running one worker for 140 hours is about $17. Because SCD is robust, this means SCD can get the same model 100x faster with the same resource bill as a single machine. Factoring in the cost for the master, running SCD with 128 workers is actually much cheaper. In particular, the same result is achieved 100x faster with 2x less cost. If cost alone should be minimized, the optimal choice would be using 16 workers which costs less than $10 and requires about five hours. Spending a dollar more and running with 32 machines, would give the results in 2.5 hours.

Besides the resource costs, learning models is also associated with other costs such as the time (=salary) of the end user who is waiting for the results, the number of experiments that can be explored, etc. That means reducing the runtime is usually much more valuable than the computation costs. Factoring in such other costs, SCD provides a substantial speed-up at marginally higher compute costs. Finally, we want to highlight, that running very large datasets on low-priority cloud environments is very inexpensive, e.g., running the 20 billion example version of the *advertising* dataset to convergence would cost about $10 in the Google cloud. Given the value that such a model provides when it is applied, the costs are many orders of magnitude smaller.

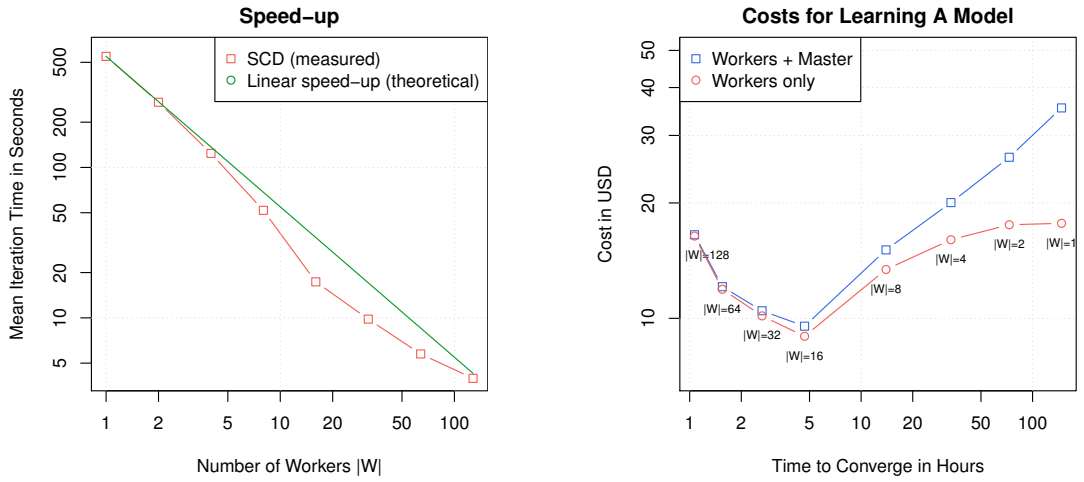

Figure 5: Speed-up behavior of SCD: Increasing the number of workers $|W|$ decreases the runtime close to linearly. The super linear speed-up is due to a larger overall cache size when running with more machines. The right plot shows the time vs. cost trade-off for learning the advertisement model to convergence with a varying number of workers $|W|$.

6. RELATED WORK

Block or parallel coordinate descent (BCD), e.g., [8, 23], and SCD share the idea of updating several coordinates in parallel but differ in important aspects. BCD [8, 23] assigns features to computing resources, e.g., cores or machines, and computes the sufficient statistics in parallel. This has consequences on both robustness and scalability.

First, the more computing resources, the more model parameters are updated in parallel with BCD. Consequently, BCD is not robust with respect to our goal in Section 2.3. In contrast to BCD, SCD parallelizes over examples while keeping the partitioning constant, which makes SCD robust.

Second, the frequency of a feature being non-zero is usually non-uniform distributed. Typically, the distribution is Zipfian, e.g., a popular video occurs in much more examples than a niche video. Moreover, imbalance occurs not only within the same variable, e.g., popular vs. non popular videos, but also over variables, e.g., high vs. low cardinality variables. For example, in the advertisement data set of Section 5, the bias feature occurs in 1 trillion example, other frequent features appear billions or millions of times but most features appear only a few times. Consequently, a parallel execution where each feature is a unit of work is impossible to load balance even in a perfect distributed environment. This means BCD will not have linear scaling properties in sparse datasets. In contrast to this, SCD parallelizes work over examples and does not care about the number of features that are updated in parallel. For example, the block sizes in Section 5 range from 1 feature per block to 240,000,000 features per block but in each block the number of non-zeros is constant $N_Z(X^B) \approx |S|$. That means, row shards are balanced in terms of computational costs.

Third, we introduce pure partitions which allow us to scale the popular single-machine CD algorithm without compromising convergence. Finally, we demonstrate the effectiveness of SCD on truly large-scale datasets. The distributed BCD algorithm [23] was evaluated on less than 10M examples using 400 machines. This dataset is 100,000 times smaller than our datasets. A single core in our system processes 10M examples in less than a second end-to-end including all synchronization work.

Another related line of work is Sibyl [10], a widely used ML platform in Google that distributes the parallel boosting algorithm [14]. Parallel boosting updates all features in one iteration and is comparable to SCD with a single non-pure block $B = \{1, \ldots, p\}$. The parallelism is achieved similar to SCD over examples. This makes Sibyl both robust and scalable. Parallel boosting fits very well a standard framework such as MapReduce [17] because iterations are long. The downside is that it requires many more epochs to converge than SCD and the memory consumption is much higher because all parameters and statistics are in each mapper. On various datasets, SCD showed an improvement of one to two orders of magnitude in convergence speed and computing resources. We omit these results due to space restrictions.

The orthogonal approach to CD is stochastic gradient descent (SGD) that iterates over examples. As described in Section 1, the single-machine SGD algorithm [7] accesses and updates parameters at a very high rate. This makes standard SGD inherently a single-machine algorithm. Many distributed SGD modifications have been proposed, e.g., using combinations of delayed updates and batching (e.g. [22, 12, 16]). Distributed SGD algorithms do not scale linearly and the behavior is more complex to control when changing the number of machines. This makes them harder to use than the SCD system – especially in a shared cloud environment where the system behavior is less predictable. In consequence, distributed SGD does not meet the design goals of robustness and linear scaling that motivated our work (see Section 2.3).

7. CONCLUSION

In this paper, we described a new scalable coordinate descent (SCD) algorithm for generalized linear models. SCD is highly robust, having the same convergence behavior regardless of how much it is scaled out and regardless of the

computing environment. This allows SCD to scale to thousands of cores and makes it well suited for running in a cloud environment with low-cost commodity servers. On many real-world problems, SCD has the same convergence behavior as the popular single-machine coordinate descent algorithm. In addition to the SCD algorithm, we described a distributed system that addresses the specific challenges of scaling SCD in a cloud computing environment. Using Google's internal cloud, we showed that SCD can provide near linear scaling using thousands of cores for 1 trillion training examples on a petabyte of compressed data.

8. ACKNOWLEDGMENTS

We would like to thank our colleagues Tushar Chandra, Mike Gunter, Judah Jacobson, Gus Katsiapis, Lukasz Lew, Alex Passos, Tal Shaked, and Greg Steuck of the Sibyl team for their insightful discussions about the SCD algorithm and system design.

9. REFERENCES

[1] https://cloud.google.com/.

[2] https://aws.amazon.com/.

[3] https://www.csie.ntu.edu.tw/~cjlin/liblinear/.

[4] https://cran.r-project.org/web/packages/glmnet/.

[5] J. Bennet and S. Lanning. The Netflix prize. In *KDD Cup and Workshop*, 2007.

[6] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[7] L. Bottou. Stochastic learning. In *Advanced Lectures on Machine Learning*, Lecture Notes in Artificial Intelligence, LNAI 3176, pages 146–168. Springer Verlag, 2004.

[8] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *Proceedings of the International Conference on Machine Learning (ICML 2011)*, June 2011.

[9] J. Canny and H. Zhao. Big data analytics with small footprint: Squaring the cloud. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 95–103, 2013.

[10] T. Chandra. Sibyl: A system for large scale machine learning at Google. *Keynote talk at the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[11] K.-W. Chang, C.-J. Hsieh, and C.-J. Lin. Coordinate descent method for large-scale l2-loss linear support vector machines. *J. Mach. Learn. Res.*, 9:1369–1398, June 2008.

[12] O. Chapelle, E. Manavoglu, and R. Rosales. Simple and scalable response prediction for display advertising. *ACM Trans. Intell. Syst. Technol.*, 5(4):61:1–61:34, Dec. 2014.

[13] Y. Chen, R. Griffit, D. Zats, and R. H. Katz. Understanding TCP incast and its implications for big data workloads. Technical report, DTIC Document, 2012.

[14] M. Collins, R. E. Schapire, and Y. Singer. Logistic regression, Adaboost and Bregman distances. *Mach. Learn.*, 48(1-3):253–285, Sept. 2002.

[15] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[16] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. A. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, pages 1223–1231. 2012.

[17] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[18] J. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22, 2010.

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, 2003.

[20] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-scale bayesian click-through rate prediction for sponsored search advertising in microsoft's bing search engine. In *Proceedings of the 27th International Conference on Machine Learning*, pages 13–20, 2010.

[21] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, and J. Q. n. Candela. Practical lessons from predicting clicks on ads at Facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, ADKDD'14, pages 5:1–5:9, 2014.

[22] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 583–598, 2014.

[23] D. Mahajan, S. S. Keerthi, and S. Sundararajan. A distributed block coordinate descent method for training l1 regularized linear classifiers. *CoRR*, abs/1405.4544, 2014.

[24] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica. Ad click prediction: A view from the trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1222–1230, 2013.

[25] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*, pages 693–701. 2011.

[26] S. Shalev-Shwartz and A. Tewari. Stochastic methods for l1-regularized loss minimization. *J. Mach. Learn. Res.*, 12:1865–1892, July 2011.

[27] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015.