Parameter Server for Distributed Machine Learning

Mu Li[1], **Li Zhou**[1], **Zichao Yang**[1], **Aaron Li**[1], **Fei Xia**[1],
David G. Andersen[1] **and Alexander Smola**[1,2]
[1]Carnegie Mellon University
[2]Google Strategic Technologies
{muli, lizhou, zichaoy, aaronli, feixia, dga}@cs.cmu.edu, alex@smola.org

Abstract

We propose a parameter server framework to solve distributed machine learning problems. Both data and workload are distributed into client nodes, while server nodes maintain globally shared parameters, which are represented as sparse vectors and matrices. The framework manages asynchronous data communications between clients and servers. Flexible consistency models, elastic scalability and fault tolerance are supported by this framework. We present algorithms and theoretical analysis for challenging nonconvex and nonsmooth problems. To demonstrate the scalability of the proposed framework, we show experimental results on real data with billions of parameters.

1 Introduction

Distributed optimization and inference is becoming popular for solving large scale machine learning problems. Using a cluster of machines overcomes the problem that no single machine can solve these problems sufficiently rapidly, due to the growth of data in both the number of observations and parameters. Implementing an efficient distributed algorithm, however, is not easy. Both intensive computational workloads and the volume of data communication demands careful system design.

It is worth noting that our system targets situations that go beyond the typical cluster-compute scenario where a modest number of homogeneous, exclusively-used, and highly reliable is *exclusively* available to the researcher. That is, we target cloud-computing situations where machines are possibly unreliable, jobs may get preempted, data may be lost, and where network latency and temporary workloads lead to a much more diverse performance profile. For instance, it is understood that synchronous operations may be significantly degraded due to occasional slowdowns, reboots, migrations, etc. of individual servers involved. In other words, we target real cloud computing scenarios applicable to Google, Baidu, Amazon, Microsoft, etc. rather than low utilization-rate, exclusive use, high performance supercomputer clusters. This requires a more robust approach to computation.

There exist several general purpose distributed machine learning systems. Mahout [5], based on Hadoop [1] and MLI [27], based on Spark [29], adopt the iterative MapReduce [14] framework. While Spark is substantially superior to Hadoop MapReduce due to its preservation of state and optimized execution strategy, both of these approaches use a *synchronous* iterative communication pattern. This makes them vulnerable to nonuniform performance distributions for iterative machine learning algorithms, i.e. machines that might happen to be slow at any given time. To overcome this limitation, distributed GraphLab [21] asynchronously schedules communication using a graph abstraction. It, however, lacks the elastic scalability of the map/reduce-based frameworks, and relies on coarse-grained snapshots for recovery. Moreover, global variables synchronization is not a first-class primitive. Of course, beyond these general frameworks, numerous systems have been developed that target specific applications, such as [3, 13, 24, 22, 28, 10, 15].

We found that many inference problems have a rather restricted structure in terms of their parametrization where considerable gains can be made by exploiting this design. For instance, generalized linear models typically use a single massive parameter vector, or topic models use an array of sparse vectors. In general, many relevant large-scale graphical models consist largely of a small number of plates, thus allowing for a repeated structure of a small number of components which are shared between observations and machines. This offers considerable efficiencies by performing these operations *in bulk* and by specializing synchronization primitives for the specific datatypes.

In this paper, we focus on the parameter server approach to distributed optimization. In this model, computational nodes are partitioned into clients and servers. Each client "owns" a portion of the data and workload, and the servers together maintain the globally shared parameters. This architectural idea has not new: It has been applied to several machine learning applications including latent variable models [26, 2, 17], distributed inference on graphs [3], and deep learning [13]. Our goal is to build a general purpose system with features only partially supported by previous work:[1]

Ease of use. The globally shared parameters are represented as (potentially sparse) vectors and matrices, which are more convenient data structures for machine learning applications than the widely used (key,value) store or tables. High-performance and convenient multi-threaded linear algebra operations, such as vector-matrix multiplication between parameters and local training data, are provided to facilitate developing applications.

Efficiency. Communication between nodes is asynchronous. Importantly, synchronization does not block computation. This framework allows the algorithm designer to balance algorithmic convergence rate and system efficiency, where the best trade-off depends on data, algorithm, and hardware.

Elastic Scalability. New nodes can be added without restarting the running framework. This property is desirable, e.g. for streaming sketches or when deploying a parameter server as an online service that must remain available for a long time. We use a distributed hash table [9] to allow new server nodes to be dynamically inserted into the set at any time.

Fault Tolerance and Durability. Conversely, node failure is inevitable, particularly at large scale using commodity servers. For instance, an MTBF (mean time between failure) of 3 years amounts to one failure per day on 1,000 nodes. Scheduler pre-emption can significantly increase this rate on industrial deployments.
We use an optimized data replication architecture that efficiently stores data on multiple server nodes to enable fast (in less than 1 second) recovery from node failure. Moreover, since client nodes are independent from each other, new clients can be started automatically when one fails in the same fashion as MapReduce is capable of rescheduling new mappers.

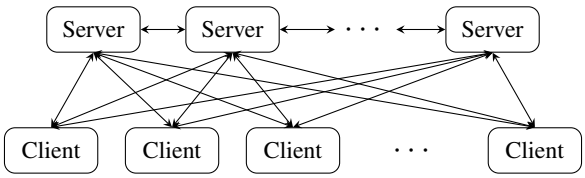


Figure 1: Communication pattern between clients and servers. Clients process data while servers synchronize parameters and perform global updates. Note that most code is shared between clients and servers, the main difference being the manner in which they update parameters.

2 Architecture

2.1 Overview

The parameter server architecture, shown above, has two classes of nodes: The *server* nodes maintain a partition of the globally shared parameters (machine local parameters are not synchronized by default). They communicate with each other to replicate and/or to migrate parameters for reliability and scaling. The *client* nodes perform the bulk of the computation; the *server* nodes mainly

[1]The C++ codes are available at http://parameterserver.org/

perform bookkeeping and global aggregation steps. Each client typically stores locally a portion of the training data, computing local statistics such as gradients. Clients communicate only with the server nodes, updating and retrieving the shared parameters. Clients may be added or removed; doing so requires transmitting the appropriate portion of the training dataset to the new machine(s) and querying the respective set of parameters.

The parameter servers may simultaneously support several independent parameter vectors (i.e. channels) for different algorithms. This is useful, e.g., when the servers may be storing parameters for an operational model being actively queried by some nodes, while also being used to train a new model for future use, using a different set of client nodes. Such an approach greatly simplifies model updates and deployments since all that is required is for the clients to switch channels.

2.2 Application Examples

Our model is easiest understood by discussing a number of superficially diverse use cases that all fit into the same framework. It is understood that specific problems can be considerably more varied and complex than these examples.

Risk minimization by distributed subgradient iterations. The objective is to solve optimization problems of the form

$$F(w) = \sum_{i=1}^{n} \ell(x_i, y_i, w) + \Omega(w). \tag{1}$$

Here $\ell(x_i, y_i, w)$ is a loss function, such as a regression or classification error that depends on the data x_i, labels y_i and the parameters w only via nonzero terms in x_i. The optimization algorithms typically iteratively computes the first-order gradient of $F(w)$, which fits into the client and server architecture: The globally shared parameters w are maintained by servers. The clients in parallel store the training data and compute its gradient: Each client takes a set of pairs (x_i, y_i), and the entries of w needed to calculate the revised gradient. During inference, these local gradients are aggregated by servers. New updated w values are sent back to the clients.

Risk minimization by parameter synchronization. The objective is identical to the above scenario. However, in this case local parameter updates are carried out at the client side and communication with the server is for parameter synchronization only, e.g. using a distributed variant of ADMM [8].

Distributed Gibbs Sampler. Latent variable models typically infer auxiliary unobserved variables Z from observed data X in a generative setting. For instance, in Latent Dirichlet Allocation (LDA), the goal is to explain observed documents by a mixture of topics [7]. Collapsed Gibbs sampling [16] is a widely used inference algorithm, which iteratively counts the statistics on (document,word), (document,topic), and (word,topic) and then reassigns topics to words based on the conditional probabilities. In the parameter server setting, documents are partitioned to clients so that the first two statistics can be computed locally. The globally shared word-to-topic assignments are then maintained by server nodes [26, 2] while the clients send state change updates to the server.

Deep Learning. Deep learning essentially iterates several nonlinear function classes. While the function classes themselves are fairly compactly described, inference on large amounts of data nonetheless requires parallelization of the set of observations. [13] describe two complementary (synchronous) variants: they decompose the set of variables over several machines and compute different parts of the objective function respectively. Secondly they decompose the observations over different machines.

Sketches. Typically data sketches [12, 6] are designed to perform well for a given time interval (e.g. by counting how many items were observed since the initialization of the sketch) rather than storing a full frequency distribution. Moreover, they are engineered for single machine storage. The use of consistent hashing allows us to distribute the event stream over multiple machines, thus increasing throughput and accuracy.

3 Interface

3.1 Key-Value Vectors

A major difference to existing approaches is that we assume that the index set of keys is ordered and potentially dense. This allows us to use vector semantics and to send larger amounts of data in bulk rather than dealing with individual (key,value) pairs. Moreover, it simplifies memory management, network traffice, and it allows us to dispose of having to store a separate index set for dense vectors.

Note that this approach is a strict superset of what typical (key,value) servers provide. For the sake of concreteness assume that the parameter server holds only one such vector. The parameter server presents the shared parameters as (sparse) vectors to clients *and* servers. Applications may treat this data as either a vector/matrix or as a set of key-value pairs, whichever is more convenient. Individual data entries can be accessed or modified using their keys, for example the `feature_id` in risk minimization problems, or the combination of `word_id` and `topic_id` in LDA. Clients or servers can also perform linear algebra operations on entire vectors, such as addition $w + u$, finding the 2-norm $\|w\|_2$, and more general operations $\alpha Ax + \beta y$, as encoded in the Level 1 BLAS subroutines. In addition, parameters can easily interact with the local training data if they are also vectors or matrices.

Beyond convenience, this interface design leads to efficient code. Taking advantage of the structure of the vectors and matrices, execution within linear algebra operations is optimized for space and time locality, as is well explored by libraries such as BLAS/Lapack [4]. It is also easier to efficiently multi-thread the internal implementation of these operators and take advantage of SIMD/vector support for both sparse and dense vectors.

3.2 Push and Pull

Data communication between nodes is captured by two operations, push and pull. The former sends local modified data entries of the shared parameters to others, while the latter retrieves remote modifications. Applications can specify whether to use either a new local value w_k or new local modification $w_k - w_k^{(\text{synced})}$ for communication. In other words, the response to a push or pull request is problem specific.

The parameter server minimizes network traffic by sending only needed data. For example, each server node typically maintains only a segment of the shared parameters. When a client pushes, the framework finds all locally updated data entries, then sends each entry to the server node that maintains the key for this entry. On the receiving side, clients often need only a subset of the shared parameters. Upon receiving a pull request from a client, a server returns only entries for the specific keys needed by the client — a list either included in the pull request or pre-negotiated with the server to further reduce traffic.[2]

Both push and pull operations are non-blocking. The caller (typically the computational thread) inserts its requests into queues, and then resumes computation. Separate I/O threads managed by the framework perform the actual network communication. This asynchronous communication results in a data consistency model that we explain in Section 4.1, and analyze theoretically in Section 5.

3.3 User-Defined Functions on the Server

Beyond aggregating data from clients, server nodes can execute user-defined functions. These can be beneficial because the server nodes often have more complete or up-to-date information about the shared parameters. For example, consider proximal gradient methods for solving risk minimization, as will be discussed further in Section 5. At each iteration, this algorithm first aggregates the gradients of the loss function and then computes a new w by solving a proximal operator associated with the gradients and the regularizer. For instance for ℓ_1 regularizers this is the soft-shrinkage operator. Using a server-side function to solve the proximal operator on the server nodes instead of the client nodes reduces the amount of data that must be moved between nodes. Likewise, in the context of sketching, the clients perform hardly any operation and the lion's share of work occurs in the servers.

[2] By omitting a key list and send values only we can double the network throughput. This is easily achieved, e.g. by transmitting only the checksum of a range of keys rather than the actual keys.

4 Toward Scale and Reliability

4.1 Consistency Model

Asynchronous communication improves the system efficiency via paralleling the usage of both CPU, disk and network bandwidth. However, it brings data inconsistency between nodes and potentially slows down the convergence rate of the optimization algorithms. The best trade-off between system efficiency and algorithm convergence rate usually depends on a variety of factors, including the algorithm's sensitivity to data inconsistency, feature correlation in training data, and capacity difference of hardware components. Instead of binding ourselves to a particular strategy, the parameter server provides flexible data consistency models for applications to select:

Best Effort. In this case the parameter server will not stall regardless of the availability of resources. For instance, [26] describe such a system. However, this is only recommendable whenever the underlying algorithms are robust with regard to delays.

Maximal Delayed Time. When a maximal delayed time τ for the push operation is set, a new push call will be blocked until all previous push call τ time ago have been finished. In other words, if we use the iteration number as the (logic) time and set $\tau = 2$, then calling push at iteration 4 will be blocked if any push operation before iteration 3 has not been finished yet, namely the network packages associated with that operation have not been successfully sent yet. Thus, if $\tau = 0$, we get the bulk synchronous parallel model, where each push call will be blocked until the data have been sent.

Also note that for an infinite delay $\tau = \infty$, we have the best-effect model [26]. The same applies for the pull operations.

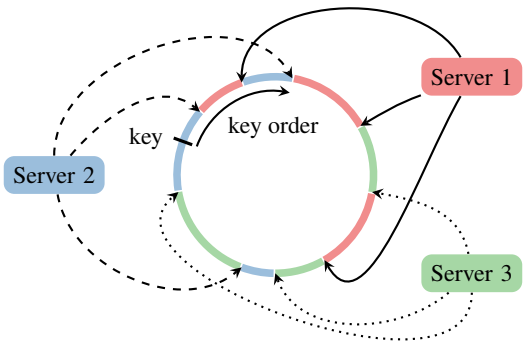
User-defined Filters. The Parameter server supports user-defined filters for selective synchronization. One example is the *significantly modified* filter, which only pushes entries that have been changed more than by a significant amount, e.g.

$$|w_k - w_k^{(\text{synced})}| > \Delta.$$

An intuitive choice is using a large Δ at the beginning, and then continuously decreasing Δ when approaching a solution.

4.2 Elastic Scalability and Fault Tolerance

We use the key-value pairs viewpoint of the shared parameters. The basic idea comes from distributed hash tables [9, 25], where both key-value pairs and server nodes are inserted into the hash ring. Each node manages the key segment starting with its insertion point to the next point by other nodes in the anticlockwise direction, which is called the anticlockwise neighbor. In the example shown on the right, the server nodes manages segments of the same color. Different to performing key discovery and routing as [18], we use a DHT for assignment and we store the mapping from key segments to nodes in Paxos [19], as implemented in Zookeeper. Note that a physical node is inserted $\log p$ times in the form of virtual nodes to facilitate load-balancing.



Each key segment is then duplicated into the k anticlockwise neighbor server nodes for fault tolerance. If $k = 1$, then the segment with the mark in the example will be duplicated at Server 3. A new node comes is first randomly (via a hash function) inserted into the ring, and then takes the key segments from its clockwise neighbors. On the other hand, if a node is removed or if it fails, its segments will be served by its nearest anticlockwise neighbors, who already own a duplicated copy if $k > 0$. To recover a failed node, we just insert a node back into the failed node's previous positions and then request the segment data from its anticlockwise neighbors.

5 Theoretical Analysis

5.1 Nonconvex and nonsmooth Optimization

We provide convergence analysis for the following nonconvex optimization problems

$$\text{minimize } F(w) := f(w) + h(w) \text{ for } w \in \mathcal{X}. \tag{2}$$

Here $f : \mathbb{R}^p \to \mathbb{R}$ is continuously differentiable but not necessary convex and $h : \mathbb{R}^p \to \mathbb{R} \cup \{\infty\}$ is LSC, convex but possibly nonsmooth.

We consider the proximal gradient methods [11, 23]. Given a closed proper convex function $h(w) : \mathbb{R}^p \to \mathbb{R} \cup \{+\infty\}$, define the proximal operator as

$$\text{Prox}_\gamma(x) = \underset{y \in \mathcal{X}}{\text{argmin}} \; h(y) + \frac{1}{2\gamma} \|x - y\|^2 .$$

To minimize the composite objective function $f(w)+h(w)$, proximal gradient methods update w in two steps: a forward step performing steepest gradient descent on f and a backward step, carrying out projection using h. For a given learning rate $\gamma_t > 0$ at iteration t this can be written as

$$w(t+1) = \text{Prox}_{\gamma_t} \left[w(t) - \gamma_t \nabla f(w(t)) \right] \quad \text{for } t \in \mathbb{N} \tag{3}$$

5.2 Asynchronous and Convergence Guarantee

We treat the server nodes as a single unit to simplify the discussion. This simplification does not affect the correctness of the analysis. However, we assume that the prox operator $\text{Prox}_\gamma(x)$ can be carried out independently over the partitions induced by a distributed representation of the parameters. For instance, for an ℓ_1 penalty this is trivially true since the prox operator acts coordinate-wise.

First, assume data are partitioned into m clients so that we can rewrite $f(w) = \sum_{i=1}^m f_i(w)$. Next, in every iteration, each client i simultaneously computes the local gradient ∇f_i and then pushes it to the server. Note that each client runs at its own pace without synchronization at the beginning of each iteration. At the same time the server gathers the updates from clients and push the new value of w back after solving the proximal operator.

We analyze the combination of the "maximal delayed time" model with a "significantly-modified" filter. Assume a maximal delay τ iterations is allowed for the former. In other words, if client i has value $w(t_i)$ at iteration t, then we have $t - \tau \leq t_i \leq t$. For the latter, assume values are sent only if their absolute local modifications are large than Δ_t. Thus, at iteration t, the gradient received by the server from client i will be

$$G_i(t) = \nabla f_i(w(t_i) + \sigma_w(t_i)) + \sigma_{\nabla_i}(t).$$

Here $w(t_i)$ is the delayed copy of parameters at client i, and σ_w and σ_∇ are error due to small changed values are filtered, which satisfy $\|\sigma_w(t_i)\|_\infty \leq \Delta_{t_i}$ and $\|\sigma_\Delta(t)\|_\infty \leq \Delta_t$. The inexact gradient used by the server to compute $w(t+1)$ is then $s(t) = \sum_{i=1}^m G_i(t)$.

The following Theorem 2 indicates that if proper learning rate and Δ_t are chosen, then this algorithm is guaranteed to converge to a stationary point under the week consistency model discussed above. The theorem needs the following assumption.

Assumption 1 (Lipschitz Continuity) *There exists positive constant L_i such that $\|\nabla f_i(x) - \nabla f_i(y)\| \leq L_i \|x - y\|$ for any $x, y \in \mathcal{X}$ and all $i = 1, \ldots, m$.*

Theorem 2 *Assume Assumption 1 and denote by $L = \sum_{i=1}^m L_i$. Let τ be the maximal delay and $\Delta_t = \mathcal{O}(\frac{1}{t})$ for the significant-modified filter. For any $\epsilon > 0$, the asynchronous implementation will converge to a stationary point if the learning rate γ_t satisfies $\gamma_t \leq ((1+\tau)L + \epsilon)^{-1}$ for all $t > 0$.*

Proof. Denote by $\delta(t) = w(t+1) - w(t)$, we first upper bound the change of F from iteration t to $t+1$. Note that $w(t+1) = \text{Prox}_{\gamma_t}(w(t) - \gamma_t s(t))$, take derivatives at the both sides of the proximal

operator definition we obtain $\frac{1}{\gamma_t}(w(t)-w(t+1))-s(t)\in\partial h(w(t+1))$. By the convexity of h,

$$\begin{aligned}h(w(t+1))-h(w(t)) &\le \left\langle \frac{1}{\gamma_t}(w(t)-w(t+1))-s(t), w(t+1)-w(t)\right\rangle \\ &= -\frac{1}{\gamma_t}\|\delta(t)\|^2-\langle s(t),\delta(t)\rangle \end{aligned} \tag{4}$$

On the other hand, apply Assumption 1

$$f(w(t+1))-f(w(t)) = \sum_{i=1}^m f_i(w(t+1))-f_i(w(t)) \tag{5}$$

$$\le \sum_{i=1}^m \langle \delta(t), \nabla f_i(w(t))\rangle + L_i\|\delta(t)\|^2 \tag{6}$$

$$= \langle \delta(t), \nabla f(w(t))\rangle + L\|\delta(t)\|^2 \tag{7}$$

Combining (4) and (7), we have

$$\begin{aligned}F(w(t+1))-F(w(t)) &\le \left(L-\frac{1}{\gamma_t}\right)\|\delta(t)\|^2+\langle\delta(t),\nabla f(w(t))-s(t)\rangle \\ &\le \left(L-\frac{1}{\gamma_t}\right)\|\delta(t)\|^2+\|\delta(t)\|\,\|\nabla f(w(t))-s(t)\| \end{aligned} \tag{8}$$

Next we upper bound the different between the gradient $\nabla f(w(t))$ and the inexact gradient $s(t)$. By the way we compute $s(t)$,

$$\begin{aligned}\|\nabla f(w(t))-s(t)\| &= \left\|\sum_{i=1}^m \nabla f_i(w(t)) - \nabla f_i\left(w(t_i)+\sigma_w(t_i)\right)-\sigma_{\nabla_i}(t)\right\| \\ &\le \sum_{i=1}^m\sum_{j=1}^{t-t_i}\|\nabla f_i(w(t-j+1))-\nabla f_i(w(t-j))\| \\ &\quad +\|\nabla f_i(w(t_i))-\nabla f_i(w(t_i)+\sigma_w(t_i))\|+\|\sigma_{\nabla_i}(t)\| \quad \text{(triangle inequality)} \\ &\le \sum_{i=1}^m\sum_{j=1}^{t-t_i} L_i\|\delta(t-j)\|+L_i\|\sigma_w(t_i)\|+L_i\|\sigma_{\nabla_i}(t)\| \quad \text{(Assumption 1)} \\ &\le \sum_{i=1}^m\sum_{j=1}^{\tau} L_i\|\delta(t-j)\|+2L_i\sqrt{p}\Delta_{t-\tau} \quad \text{(delay} \le \tau \text{ and } \|w\|\le\sqrt{p}\|w\|_\infty) \\ &= \sum_{j=1}^{\tau} L\|\delta(t-j)\|+2L\sqrt{p}\Delta_{t-\tau} \end{aligned} \tag{9}$$

Substituting (9) into (8), we have

$$\begin{aligned}&F(w(t+1))-F(w(t)) \\ &\le \left(L-\frac{1}{\gamma_t}\right)\|\delta(t)\|^2+\sum_{j=1}^{\tau} L\|\delta(t)\|\,\|\delta(t-j)\|+\|\delta(t)\|\,2L\sqrt{p}\Delta_{t-\tau} \\ &\le \left(L+\frac{\tau L}{2}+\frac{\epsilon}{2}-\frac{1}{\gamma_t}\right)\|\delta(t)\|^2+\sum_{t=1}^{\tau}\frac{L}{2}\|\delta(t-j)\|^2+2L^2p\Delta_{t-\tau}^2 \end{aligned} \tag{10}$$

Summing over t we have the following chain

$$F(w(T+1))-F(w(0)) \le \sum_{t=0}^{T}\left((1+\tau)L+\frac{\epsilon}{2}-\frac{1}{\gamma_t}\right)\|\delta(t)\|^2+2L^2p\Delta_{t-\tau}^2 \tag{11}$$

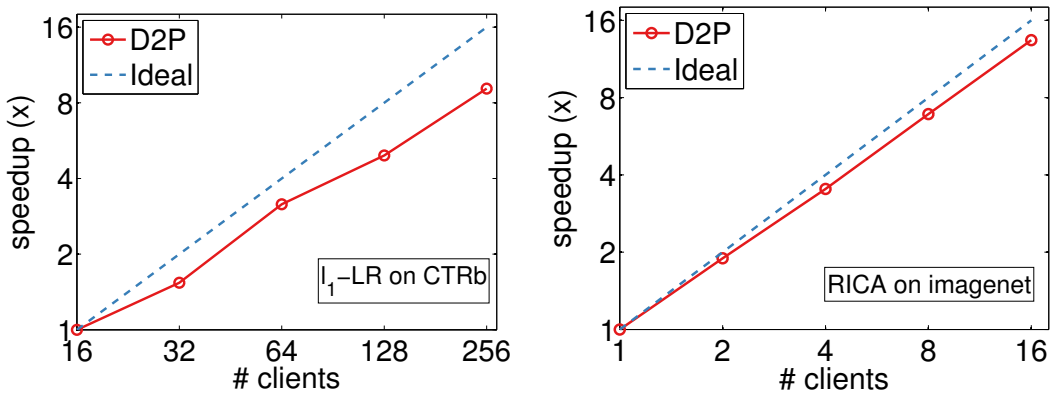


Figure 2: Linear scalability for two problems and associated decompositions. Left: optimization over 31 billion attributes in a binary classification problem. Right: problem decomposition in RICA on image net. Both instances show essentially a perfect speedup over two orders of magnitude.

Denote by $c(t) = \frac{1}{\gamma_t} - (1+\tau)L - \frac{\epsilon}{2}$, since $\gamma_t \leq \frac{1}{(1+\tau)L+\epsilon}$ for all t, then all $c(t) \geq \frac{\epsilon}{2} > 0$. So

$$\frac{\epsilon}{2}\sum_{t=0}^{T}\|\delta(t)\|^2 \leq \sum_{t=0}^{T} c(t)\,\|\delta(t)\|^2 \leq F(x(0)) - F(x(T+1)) + \sum_{t=1}^{T} 2L^2 p\Delta_{t-\tau}^2 \tag{12}$$

for any T. Since $\Delta_t = O(\frac{1}{t})$, and by the fact that $1 + \frac{1}{2^2} + \frac{1}{3^2} + \ldots = \frac{\pi^2}{6}$. Then the RHS of (12) is constant when $T \to \infty$, which implies $\lim_{t\to\infty}\delta(t) \to 0$. So $\lim_{t\to\infty}\mathrm{Prox}_{\gamma_t}(w(t)) - w(t) \to 0$, thus we find a local minimal point.

■

Intuitively, the difference between $w(t-\tau)$ and $w(t)$ will be small when approaching a stationary point. Besides, since Δ_t is decreasing to zero, so are the values in σ_w and σ_{∇_i}. The inexact gradient $s(t)$, therefore, would be a good approximation of the true gradient $\nabla f(w(t))$, so the convergence results of proximal gradient methods can be applied.

6 Experimental Results

In this section we solve two challenging problems — ℓ_1-regularized logistic regression (ℓ_1-LR) with sparse training data:

$$\min_{w\in\mathbb{R}^p}\sum_{i=1}^{n}\log(1+\exp(-y_i\,\langle x_i, w\rangle)) + \lambda\|w\|_1$$

and reconstruction ICA (RICA) with dense training data:

$$\min_{W\in\mathbb{R}^{\ell\times p}}\sum_{i=1}^{n}\frac{1}{2}\left\|WW^\top x_i - x_i\right\|_2^2 + \lambda\left\|Wx_i\right\|_1.$$

The former is convex, but the latter is highly nonconvex. We use an asynchronous proximal gradient method. The proximal operator has closed-form solution for ℓ_1-LR, i.e. the soft-shrinkage operator. We solve it by ADMM [8] for RICA. More details are presented in [20].

We run ℓ_1-LR and RICA on clusters with 256 AMD CPU cores and 16 Tesla K20 graphical cards, respectively. A range of sparse and dense datasets have been evaluated. CTRb and imagenet are the largest two. The former is a sparse Ads click-through dataset from a major search company. It contains 0.34B observations, 2.2B unique features, and 31B nonzero entries. The latter has been obtained from `http://www.image-net.org` with 100K images resized into 100x100 pixels.

Due to the space constraint, we only report the scalability results on these two datasets with maximal delay $\tau = 4$. For RICA we use 1 Billion parameters $W \in \mathbb{R}^{10^6 \times 10^4}$. We measure the scalability, shown in Figure 2, on the speedup of training time to achieve the same convergence precision. A ninefold speedup is observed when increasing the clients by 16 times for ℓ_1-LR, while we see a 13.5 fold acceleration for RICA.

7 Conclusion

In this paper, we described a parameter server framework to solve distributed machine learning problems. This framework is easy to use: Globally shared parameters can be used as local sparse vectors or matrices to perform linear algebra operations with local training data. It is efficient: All communication is asynchronous and flexible consistent models are supported to balance the trade-off between system efficient and fast algorithm convergence rate. Furthermore, it provides elastic scalability and fault tolerant aiming for stable long term deployment. We present convergence analysis under weak data consistency requirement. Last, we show experiments of two challenging tasks on real datasets with billions of variables to demonstrate the linear scalability.

References

[1] Apache hadoop, 2009. http://hadoop.apache.org/core/.

[2] Amr Ahmed, Mohamed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of The 5th ACM International Conference on Web Search and Data Mining (WSDM)*, 2012.

[3] Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J. Smola. Distributed large-scale natural graph factorization. In *World Wide Web Conference*, Rio de Janeiro, 2013.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, second edition, 1995.

[5] Apache Foundation. Mahout project, 2012. http://mahout.apache.org.

[6] R. Berinde, G. Cormode, P. Indyk, and M.J. Strauss. Space-optimal heavy hitters with strong error bounds. In J. Paredaens and J. Su, editors, *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS*, pages 157–166. ACM, 2009.

[7] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, January 2003.

[8] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–123, 2010.

[9] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *Peer-to-peer systems II*, pages 80–87. Springer, 2003.

[10] W.Y. Chen, Y. Song, H. Bai, C.J. Lin, and E.Y. Chang. Parallel spectral clustering in distributed systems. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(3):568–586, 2011.

[11] P. L. Combettes and J. C. Pesquet. Proximal splitting methods in signal processing. In *Fixed-Point Algorithms for Inverse Problems in Science and Engineering*, pages 185–212. Springer, 2011.

[12] G. Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *SDM*, 2005.

[13] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. Large scale distributed deep networks. In *Neural Information Processing Systems*, 2012.

[14] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[15] John Duchi, Alekh Agarwal, and Martin Wainwright. Distributed dual averaging in networks. In *Advances in Neural Information Processing Systems 23*, 2010.

[16] T.L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101:5228–5235, 2004.

[17] Q. Ho, J. Cipar, H. Cui, S. Lee, J. Kim, P. Gibbons, G. Gibson, G. Ganger, and E. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[18] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Symposium on the Theory of Computing STOC*, pages 654–663, New York, May 1997. Association for Computing Machinery.

[19] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[20] M. Li, D. G. Andersen, and A. J. Smola. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning*, 2013.

[21] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *PVLDB*, 2012.

[22] N. Parikh and S. Boyd. Graph projection block splitting for distributed optimization, 2012. submitted.

[23] N. Parikh and S. Boyd. Proximal algorithms. *To appear in Foundations and Trends in Optimization*, 2013.

[24] P. Richtarik and M. Takac. Distributed coordinate descent method for learning with big data. Technical report, 2013.

[25] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.

[26] A. J. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Very Large Databases (VLDB)*, 2010.

[27] E. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. Mli: An api for distributed machine learning. 2013.

[28] Christina Teflioudi, Faraz Makari, and Rainer Gemulla. Distributed matrix completion. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 655–664. IEEE, 2012.

[29] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Fast and interactive analytics over hadoop data with spark. *USENIX ;login:*, 37(4):45–51, August 2012.