Analysis of High-Performance Distributed ML at Scale through Parameter Server Consistency Models

Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho*, Garth Gibson and Eric P. Xing
School of Computer Science, Carnegie Mellon University
*Institute for Infocomm Research, A*STAR
wdai,abhimank,jinlianw,garth,epxing@cs.cmu.edu, hoqirong@gmail.com

Abstract

As Machine Learning (ML) applications embrace greater data size and model complexity, practitioners turn to distributed clusters to satisfy the increased computational and memory demands. Effective use of clusters for ML programs requires considerable expertise in writing distributed code, but existing highly-abstracted frameworks like Hadoop that pose low barriers to distributed-programming have not, in practice, matched the performance seen in highly specialized and advanced ML implementations. The recent Parameter Server (PS) paradigm is a middle ground between these extremes, allowing easy conversion of single-machine parallel ML programs into distributed ones, while maintaining high throughput through relaxed "consistency models" that allow asynchronous (and, hence, inconsistent) parameter reads. However, due to insufficient theoretical study, it is not clear which of these consistency models can really ensure correct ML algorithm output; at the same time, there remain many theoretically-motivated but undiscovered opportunities to maximize computational throughput. Inspired by this challenge, we study both the theoretical guarantees and empirical behavior of *iterative-convergent ML algorithms* in existing PS consistency models. We then use the gleaned insights to improve a consistency model using an "eager" PS communication mechanism, and implement it as a new PS system that enables ML programs to reach their solution more quickly.

Introduction

The surging data volumes generated by internet activity and scientific research (Dean et al. 2012) put tremendous pressure on Machine Learning (ML) methods to scale beyond the computation and memory limit of a single machine. With very large data sizes (Big Data), a single machine would require too much time to process complex ML models (Ahmed et al. 2012; Ho et al. 2013; Cipar et al. 2013; Cui et al. 2014; Li et al. 2014), and this motivates or even necessitates distributed-parallel computation over a cluster of machines. A popular approach is *data parallelism*, in which the data is partitioned and distributed across different machines, which then train the (shared) ML model using their local data. One of the primary challenges in data parallelism is designing software systems to share the ML model across a cluster of commodity machines, in a manner that is theoretically well-justified, empirically high-performance, and easy-to-program all at the same time — and a highly promising solution is the titular "Parameter Server" (PS) paradigm, which ML practitioners have been turning to in recent years (Ahmed et al. 2012; Ho et al. 2013; Cipar et al. 2013; Cui et al. 2014; Li et al. 2014).

Many general-purpose Parameter Server (PS) systems (Ho et al. 2013; Cipar et al. 2013; Cui et al. 2014) provide a Distributed Shared Memory (DSM) solution to the issue of Big Data in ML computation. DSM allows ML programmers to treat the entire cluster as a single memory pool, where every machine can read/write to any model parameter via a simple programming interface; this greatly facilitates the implementation of distributed ML programs, because programmers may treat a cluster like a "supercomputer" that can run thousands of computational threads, without worrying about low-level communication between machines. It should be noted that not all PS systems provide a DSM interface; some espouse an arguably less-convenient push/pull interface that requires users to explicitly decide which parts of the ML model need to be communicated (Li et al. 2014), in exchange for potentially more aggressive efficiency gains.

A major motivation behind the parameter server architecture is the opportunities it offers — without forcing users to use complicated distributed programming tools — to explore various relaxed consistency models for controlled asynchronous parallel execution of machine learning programs, in an effort to improve overall system efficiency. Central to the legitimacy of employing such inconsistent models — which in theory would cause erroneous outcomes — is the iterative-convergent nature of ML programs, which presents unique opportunities and challenges that do not manifest in traditional database applications. One of the most interesting opportunities is that iterative-convergent ML programs exhibit limited error-tolerance in the parameter states, a point that is confirmed by the theoretical and empirical success of stochastic subsampling (e.g. minibatch methods) or randomized algorithms, which are prominently represented in the large-scale ML literature (Ahmed et al. 2012; Ho et al. 2013; Gemulla et al. 2011; Kumar et al. 2014). The key challenge is to determine *why*

error tolerance allows inconsistent execution models to accelerate distributed-parallel execution of ML algorithms on Big Data (while still producing correct answers), which in turn teaches us *how* to construct better execution models that not only have superior theoretical properties, but also enable more efficient systems designs.

Recent works (Niu et al. 2011; Ho et al. 2013; Kumar et al. 2014) have introduced relaxed consistency models to trade off between parameter read accuracy and read throughput, and show promising speedups over fully-consistent models. Their success is underpinned by the aforementioned error-tolerant nature of ML, that "plays nicely" with relaxed synchronization guarantees — and in turn, relaxed synchronization allows system designers to achieve higher throughput, compared to fully-consistent models. These progresses notwithstanding, we still possess limited understanding of (1) how relaxed consistency affects the convergence rate and stability of distributed ML algorithms, and (2) what opportunities remain undiscovered or unexploited to seek better trade-offs between the performance of the distributed ML algorithm (how much progress it makes per iteration, which can be compromised by relaxed consistency), and the throughput of the PS system (how many iterations can be executed per second, which increases with relaxed consistency). Contemporary PS literature has focused most on system optimizations for PSes, using various heuristics like async relaxation (Chilimbi et al. 2014) and uneven update propagation based on parameter values (Li et al. 2014), but lacks a thorough formal analysis of such trade-offs. In this paper, we investigate these issues from an ML-theoretic standpoint, and provide principled insights on how system behaviors influence algorithmic outcomes, as well as insights on how to improve PS design thereupon. Concretely, we examine the theoretical and empirical behavior of PS consistency models from several interesting angles, such as the distribution of stale reads and the impact of staleness on solution stability. We then apply the learnt insights to design a more efficient consistency model and, accordingly, a new PS system that outperforms its predecessors.

We begin with an in-depth examination of a Value-Bounded Asynchronous Parallel (VAP) model, which we formulated to encapsulate an appealing idea that can be shown to represent an ideal, gold-standard parallelization strategy in terms of theoretical behavior (high progress per ML algorithm iteration). Unfortunately, although the basic idea of VAP was attempted in (Li et al. 2013), mathematically the VAP principle can be only approximated, because bounding the value of in-transit updates amounts to tight synchronization. Without well-defined system guarantees, theses system artifacts built on approximate VAP are difficult to formally characterize. We then propose an Eager Stale Synchronous Parallel (ESSP) model, a variant of the Stale Synchronous Parallel (SSP, a bounded-iteration model that is fundamentally different from VAP) model introduced in (Ho et al. 2013), and formally show that ESSP offers a comparable guarantee to that of the ideal VAP, but represents a practical and easily realizable PS scheme for data-parallelization. Specifically, we establish expectation and variance bounds for both ESSP and VAP, and show that ESSP attains qualitatively the same guarantees as VAP, but with an arguably easier-to-control parameter (staleness versus value-bound), and significantly stronger guarantees than that of the original SSP. The variance bounds in particular provide a deeper characterization of convergence (particularly solution stability) under SSP and VAP, which is not possible with existing PS theory that is focused only on expectation bounds (Ho et al. 2013). Upon this new theory, we develop an efficient implementation of ESSP and shows that it outperforms SSP in convergence (both real time and per iteration) by reducing the average staleness, consistent with our theoretical results. This ESSP implementation will be made available soon as part of the Petuum project (`www.petuum.org`), an open-source framework for distributed Machine Learning.

Taking a broader view of recent large-scale ML efforts, we see our work as complementing *model-parallel* efforts (Lee et al. 2014; Kumar et al. 2014) that directly partition ML parameters for parallel updates (as opposed to PS-supported data-parallellism, where every data partition updates the *whole* parameter state). While the theoretical underpinnings of model-parallelism seem quite different from the data-parallel analysis espoused in this paper, we conjecture that the two types of parallelism are in fact compatible, and can be practically applied, as well as theoretically analyzed, in tandem. Since data-parallelism solves the issue of large data, while model-parallelism addresses massive ML model sizes, it seems only natural that their combination should yield a highly-promising solution to the Big ML challenge, where both data and model sizes greatly exceed the computational and memory capacity of a single machine. We expect the results presented in this paper will provide a desirable theoretical and systems paradigm for studying this promising direction in future work.

Parameter Server and Data Parallelism

We now make precise the data-parallel computation model and the parameter server (PS) abstraction that facilitates large-scale data-parallel computation. In data-parallel ML the data set D is pre-partitioned or naturally stored on worker machines, indexed by $p = 1, ..., P$ (Fig. 1a). Let D_p be the p-th data partition, $A^{(t)}$ be the model parameters at clock t, the data-parallel computation executes the following update equation until some convergence criteria is met:

$$A^{(t)} = F(A^{(t-1)}, \sum_{p=1}^{P} \Delta(A^{(t-1)}, D_p))$$

where $\Delta()$ performs computation using full model state $A^{(t-1)}$ on data partition D_p. The sum of intermediate results from $\Delta()$ and current model state $A^{(t-1)}$ are aggregated by $F()$ to generate the next model state. For stochastic gradient (SGD) algorithms like the one used in this paper for matrix factorization, $\Delta()$ computes the gradient and the update is $A^{(t)} = A^{(t-1)} + \eta \sum_{p=1}^{P} \Delta(A^{(t-1)}, D_p)$ with η being the step size. The second example application in this paper is topic model (LDA) using collapsed Gibbs sampling. In this

case $\Delta()$ increments and decrements the sufficient statistics ("word-topic" counts) according to the sampled topic assignments. This update equation can also express variational EM algorithms $A^{(t)} = \sum_{p=1}^{P} \Delta(A^{(t-1)}, D_p)$. To achieve

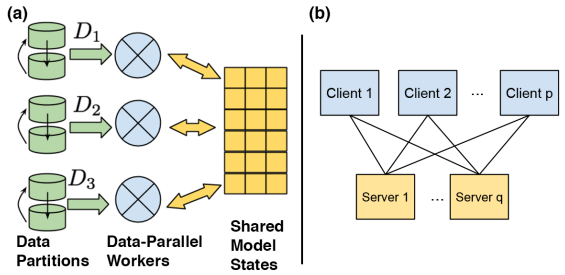

Figure 1: (a) Illustration of data parallelism. (b) Parameter server topology. Servers and clients interact via a bipartite topology. Note that this is the logical topology; physically the servers can collocate with the clients to utilize CPU on all machines.

data-parallel ML computation, we need to (1) make model state A available to all workers, and (2) accumulate the Δ updates from workers. A Parameter Server (PS) serves these needs by providing a distributed shared memory interface (a shared key-value store using a centralized storage model). The model parameters A are stored on the server, which can be distributed and thus not limited by a single machine's memory (Fig. 1b). The worker machines access the entire model state on servers via a key-value interface. This distributed shared memory provided by the PS can be easily retrofitted to existing single-machine multi-threaded ML programs, by simply replacing reads/updates to the model parameters with read/update calls to the PS. Because inter-machine communication over networks is several orders of magnitude slower than CPU-memory communication in both bandwidth and latency, one challenge in implementing an efficient PS is to design consistency models for efficient reads/updates of model parameters.

Consistency Models for Parameter Servers

A key idea for large-scale distributed ML is to carefully trade off parameter consistency for increased parameter read throughput (and thus faster algorithm execution), in a manner that guarantees the final output of an ML algorithm is still "correct" (meaning that it has reached a locally-optimal answer). This is possible because ML algorithms are *iterative-convergent* and *error-tolerant*: ML algorithms will converge to a local optimum even when there are errors in the algorithmic procedure itself (such as stochasticity in randomized methods).

In a distributed-parallel environment, multiple workers must simultaneously generate updates to shared global parameters. Hence, enforcing strong consistency (parameters updates are immediately reflected) quickly leads to frequent, time-consuming synchronization and thus very limited speed up from parallelization. One must therefore define a relaxed consistency model that enables low-synchronization parallelism while closely approximating the strong consistency of sequential execution. The insight is that, to an iterative-convergent ML algorithm, inconsistent parameter reads have essentially the same effect as errors due to the algorithmic procedure — implying that convergence to local optima can still happen even under inconsistent reads, *provided the degree of inconsistency is carefully controlled*. We now explain two possible PS consistency models, and the trade-offs they introduce.

The ideal but hard-to-realize Value-bounded Asynchronous Parallel (VAP) model

We first introduce Value-bounded Asynchronous Parallel (VAP), an ideal model that directly approximates strong consistency (e.g. in the sequential setting) by bounding the difference in *magnitude* between the strongly consistent view of values (i.e. values under the single-thread model) and the actual parameter views on the workers. Formally, let $\mathbf{x}$ represent all model parameters, and assume that each worker in the ML algorithm produces additive updates ($\mathbf{x} \leftarrow \mathbf{x} + \mathbf{u}$, where $\mathbf{u}$ is the update)[1]. Given P workers, we say that an update $\mathbf{u}$ is *in transit* if $\mathbf{u}$ has been seen by $P-1$ or fewer workers — in other words, it is yet visible by all workers. Update $\mathbf{u}$ is no longer in transit once seen by all workers. The VAP requires the following condition:

VAP condition: Let $\mathbf{u}_{p,i}$ be the updates from worker p that are in transit, and $\mathbf{u}_p := \sum_i \mathbf{u}_{p,i}$. VAP requires that, whenever any worker performs a computation involving the model variables $\mathbf{x}$, the condition $||\mathbf{u}_p||_\infty \leq v_{thr}$ holds for a specified (time-varying) value bound parameter v_{thr}. In other words, the aggregated in-transit updates from all workers cannot be too large.

To analyze VAP, we must identify algorithmic properties common to ML algorithms. Broadly speaking, most ML algorithms are either *optimization-based* or *sampling-based*. Within the former, many Big Data ML algorithms are stochastic gradient descent-based (SGD), because SGD allows each worker to operate on its own data partition (i.e. no need to transfer data between workers), while the algorithm parameters are globally shared (hence a PS is necessary). SGD's popularity makes it a good choice for grounding our analysis — in later section, we show that VAP approximates strong consistency well in these senses: (1) SGD with VAP errors converges *in expectation* to an optimum; (2) the parameter *variance* decreases in successive iterations, guaranteeing the quality and stability of the final result.

While being theoretically attractive, the VAP condition is too strong to be implemented efficiently in practice: before any worker can perform computation on $\mathbf{x}$, it must ensure that the in-transit updates from all other workers sum to at most v_{thr} component-wise due to the max-norm. This poses a chicken-and-egg conundrum: for a worker to ensure the VAP condition holds, it needs to know the updates from all other workers — which, in general, requires the same amount of communication as strong consistency, defeating the purpose of VAP. While it may be possible to relax the VAP condition for specific problem structures, in general, value-bounds are difficult to achieve for a generic PS.

[1]This is common in algorithms such as gradient descent ($\mathbf{u}$ being the gradient) and sampling methods.

Eager Stale Synchronous Parallel (ESSP)

To design a consistency model that is practically efficient while providing correctness guarantees, we consider an iteration-based consistency model called Stale Synchronous Parallel (SSP) (Ho et al. 2013) that can be efficiently implemented in PS. At a high level, SSP imposes bounds on *clock*, which represents some unit of work in an ML algorithm, akin to iteration. Given P workers, SSP assigns each worker a clock c_p that is initially zero. Then, each worker repeats the following operations: (1) perform computation using shared parameters $\mathbf{x}$ stored in the PS, (2) make additive updates $\mathbf{u}$ to the PS, and (3) advance its own clock c_p by 1. The SSP model limits fast workers' progress so that the clock difference between the fastest and slowest worker is $\leq s$, s being a specified staleness parameter. This is achieved via:

SSP Condition (informal): Let c be the clock of the fastest workers. They may not make further progress until all other workers' updates $\mathbf{u}_p$ that were made at clocks at or before $c - s - 1$ become visible.

We present the formal condition in the next section. Crucially, there are multiple update communication strategies that can meet the SSP condition. We present *Eager SSP (ESSP)* as a class of implementations that eagerly propagate the updates to reduce empirical staleness beyond required by SSP. ESSP does not provide new guarantees beyond SSP, but we show that by reducing the average staleness ESSP achieves faster convergence theoretically and empirically.

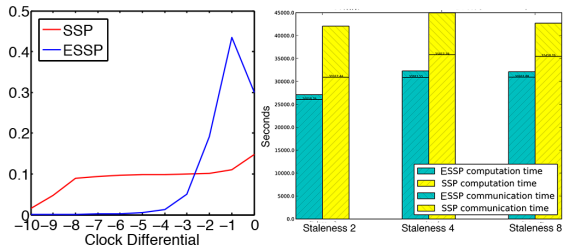

Figure 2: **Left:** Empirical staleness distribution from matrix factorization. X-axis is (parameter age - local clock), y-axis is the normalized observation count. Note that on Bulk Synchronous Parallel (BSP) system such as Map-Reduce, the staleness is always -1. We use rank 100 for matrix factorization, and each clock is 1% minibatch. The experiment is run on a 64 node cluster. **Right:** Communication and Computation breakdown for LDA. The lower part of the bars are computation, and the upper part is communication.

To show that ESSP reduces the staleness of parameter read, we can empirically measure the staleness of parameter reads during PS algorithm execution. Fig. 2 (left) shows the distribution of parameter staleness observed in matrix factorization implemented on SSP and ESSP. Our measure of staleness is a "clock differential": when a worker reads a parameter, the read parameters reflect updates from other worker 0 or more clocks behind. Clock differential simply measures this clock difference. Under SSP, the distribution of clock differentials is nearly uniform, because SSP "waits until the last minute" to update the local parameter cache. On the other hand, ESSP frequently updates the local parameter caches via its eager communication, which reduces the negative tail in clock differential distribution — this improved staleness profile is ESSP's most salient advantage over SSP. In the sequel, we will show that better staleness profiles lead to faster ML algorithm convergence, by proving new, stronger convergence bounds based on average staleness and the staleness distributions (unlike the simpler worst-case bounds in (Ho et al. 2013)).

Our analyses and experiments show that ESSP combines the strengths of VAP and SSP: (1) ESSP achieves strong theoretical properties comparable to VAP; (2) ESSP can be efficiently implemented, with excellent empirical performance on two ML algorithms: matrix completion using SGD, and topic modeling using sampling. We also show that ESSP achieves higher throughput than SSP, thanks to system optimizations exploiting ESSP's aggressive scheduling.

Theoretical Analysis

In this section, we theoretically analyze VAP and ESSP, and show how they affect ML algorithm convergence. For space reasons, all proofs are placed in the appendix. We ground our analysis on ML algorithms in the stochastic gradient descent (SGD) family (due to its high popularity for Big Data), and prove the convergence of SGD under VAP and ESSP. We now explain SGD in the context of a matrix completion problem.

SGD for Low Rank Matrix Factorization

Matrix completion involves decomposing an $N \times M$ matrix D into two low rank matrices $L \in \mathbb{R}^{N \times K}$ and $R \in \mathbb{R}^{K \times M}$ such that $LR \approx D$ gives the prediction of missing entries in D, where $K << \min\{M, N\}$ is a user-specified rank. The ℓ_2-penalized optimization problem is:

$$\min_{L,R} \sum_{(i,j) \in D_{obs}} ||D_{ij} - \sum_{k=1}^{K} L_{ik} R_{kj}||^2 + \lambda(||L||_F^2 + ||R||_F^2)$$

where $||\cdot||_F$ is the Frobenius norm and λ is the regularization parameter. The stochastic gradient updates for each observed entry $D_{ij} \in D_{obs}$ are

$$L_{i*} \leftarrow L_{i*} + \gamma(e_{ij} R_{*j}^\top - \lambda L_{i*})$$
$$R_{*j}^\top \leftarrow R_{*j} + \gamma(e_{ij} L_{i*}^\top - \lambda R_{*j})$$

where L_{i*}, R_{*j} are row and column of L, R respectively, and $L_{i*} R_{*j}$ is the vector product. $e_{ij} = D_{ij} - L_{i*} R_{*j}$. Since L, R are being updated by each gradient, we put them in the parameter server to allow all workers access them and make additive updates. The data D_{obs} are partitioned into worker nodes and stored locally.

VAP We formally introduce the VAP computation model: given P workers that produce updates at regular intervals which we call "clocks", let $\mathbf{u}_{p,c} \in \mathbb{R}^n$ be the update from worker p at clock c applied to the system state $\mathbf{x} \in \mathbb{R}^n$ via $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{u}_{p,c}$. Consider the update sequence $\hat{\mathbf{u}}_t$ that orders the updates based on the global time-stamp they are generated. We can define "real-time sequence" $\hat{\mathbf{x}}_t$ as

$$\hat{\mathbf{x}}_t := \mathbf{x}_0 + \sum_{t'=1}^{t} \hat{\mathbf{u}}_{t'}$$

assuming all workers start from the agreed-upon initial state $\mathbf{x}_0$. (Note that $\hat{\mathbf{x}}_t$ is different from the parameter server view as the updates from different workers can arrive the server in a different order due to network.) Let $\breve{\mathbf{x}}_t$ be the noisy view some worker w sees when generating update $\hat{\mathbf{u}}_t$, i.e., $\hat{\mathbf{u}}_t := G(\breve{\mathbf{x}}_t)$ for some function G. The VAP condition guarantees

$$||\breve{\mathbf{x}}_t - \hat{\mathbf{x}}_t||_\infty \leq v_t = \frac{v_0}{\sqrt{t}} \tag{1}$$

where we require the value bound v_t to shrink over time from the initial bound v_0. Notice that $\breve{\mathbf{x}}_t - \hat{\mathbf{x}}_t$ is exactly the updates *in transit* w.r.t. worker w. We make mild assumptions to avoid pathological cases.[2]

Theorem 1 *(SGD under VAP, convergence in expectation) Given convex function $f(\boldsymbol{x}) = \sum_{t=1}^{T} f_t(\boldsymbol{x})$ such that components f_t are also convex. We search for minimizer $\boldsymbol{x}^*$ via gradient descent on each component ∇f_t with step-size $\breve{\eta}_t$ close to $\eta_t = \frac{\eta}{\sqrt{t}}$ such that the update $\hat{\boldsymbol{u}}_t = -\breve{\eta}_t \nabla f_t(\breve{\boldsymbol{x}}_t)$ is computed on noisy view $\breve{\boldsymbol{x}}_t$. The VAP bound follows the decreasing v_t described above. Under suitable conditions (f_t are L-Lipschitz and bounded diameter $D(x\|x') \leq F^2$),*

$$R[X] := \sum_{t=1}^{T} f_t(\breve{\mathbf{x}}_t) - f(\boldsymbol{x}^*) = \mathcal{O}(\sqrt{T})$$

and thus $\frac{R[X]}{T} \to 0$ as $T \to \infty$.

Theorem 1 implies that the worker's noisy VAP view $\breve{\mathbf{x}}_t$ converges to the global optimum $\mathbf{x}^*$, as measured by f, in expectation at the rate $\mathcal{O}(T^{-1/2})$. The analysis is similar to (Ho et al. 2013), but we use the real-time sequence $\hat{\mathbf{x}}_t$ as our reference sequence and VAP condition instead of SSP. Loosely speaking, Theorem 1 shows that VAP execution is unbiased. We now present a new bound on the variance.

Theorem 2 *(SGD under VAP, bounded variance) Assuming $f(\boldsymbol{x})$, $\breve{\eta}_t$, and v_t similar to theorem 1 above, and $f(\boldsymbol{x})$ has bounded and invertible Hessian, Ω^* defined at optimal point $\boldsymbol{x}^*$. Let $\mathrm{Var}_t := \mathbb{E}[\breve{\boldsymbol{x}}_t^2] - \mathbb{E}[\breve{\boldsymbol{x}}_t]^2$ (Var_t is the sum of component-wise variance[3]), and $\breve{\boldsymbol{g}}_t = \nabla f_t(\breve{\boldsymbol{x}}_t)$ is the gradient, then:*

$$\mathrm{Var}_{t+1} = \mathrm{Var}_t - 2cov(\hat{\boldsymbol{x}}_t, \mathbb{E}^{\Delta_t}[\breve{\boldsymbol{g}}_t]) + \mathcal{O}(\delta_t) \tag{2}$$

$$+ \mathcal{O}(\breve{\eta}_t^2 \rho_t^2) + \mathcal{O}^*_{\delta_t} \tag{3}$$

near the optima $\boldsymbol{x}^*$. The covariance $cov(\boldsymbol{a}, \boldsymbol{b}) := \mathbb{E}[\boldsymbol{a}^T \boldsymbol{b}] - \mathbb{E}[\boldsymbol{a}^T]\mathbb{E}[\boldsymbol{b}]$ uses inner product. $\delta_t = ||\boldsymbol{\delta}_t||_\infty$ and $\boldsymbol{\delta}_t = \breve{\boldsymbol{x}}_t - \hat{\boldsymbol{x}}_t$. $\rho_t = ||\breve{\boldsymbol{x}}_t - \boldsymbol{x}^*||$. Δ_t is a random variable capturing the randomness of update $\hat{\boldsymbol{u}}_t = -\eta_t \breve{\boldsymbol{g}}_t$ conditioned on $\hat{\boldsymbol{x}}_t$ (see appendix).

[2]To avoid pathological cases where a worker is delayed indefinitely, we assume that each worker's updates are finitely apart in sequence $\hat{\mathbf{u}}_t$. In other words, all workers generate updates with sufficient frequency. For SGD, we further assume that each worker updates its step-sizes sufficiently often that the local step-size $\breve{\eta}_t = \frac{\eta}{\sqrt{t-r}}$ for some bounded drift $r \geq 0$ and thus $\breve{\eta}_t$ is close to the global step size schedule $\eta_t = \frac{\eta}{\sqrt{t}}$.

[3]$\mathrm{Var}_t = \sum_{i=1}^{d} \mathbb{E}[\breve{x}_{ti}^2] - \mathbb{E}[\breve{x}_{ti}]^2$

$cov(\hat{\mathbf{x}}_t, \mathbb{E}^{\Delta_t}[\breve{\mathbf{g}}_t]) \geq 0$ in general as the change in $\mathbf{x}_t$ and average gradient $\mathbb{E}^{\Delta_t}[\breve{\mathbf{g}}_t]$ are of the same direction. Theorem 2 implies that under VAP the variance decreases in successive iterations for sufficiently small δ_t, which can be controlled via VAP threshold v_t. However, the VAP condition requires tight synchronization as $\delta_t \to 0$. This motivates our following analysis of the SSP model.

SSP We return to the (p, c) index. Under the SSP worker p at clock c only has access to a noisy view $\tilde{\mathbf{x}}_{p,c}$ of the system state ($\tilde{\mathbf{x}}$ is different from the noisy view in VAP $\breve{\mathbf{x}}$). Update $\mathbf{u}_{p,c} = G(\tilde{\mathbf{x}}_{p,c})$ is computed on the noisy view $\tilde{\mathbf{x}}_{p,c}$ for some function $G()$. Assuming all workers start from the agreed-upon initial state $\mathbf{x}_0$, the SSP condition is:

SSP Bounded-Staleness (formal): For a fixed staleness s, the noisy state $\tilde{\mathbf{x}}_{p,c}$ is equal to

$$\tilde{\mathbf{x}}_{p,c} = \mathbf{x}_0 + \underbrace{\left[\sum_{c'=1}^{c-s-1} \sum_{p'=1}^{P} \mathbf{u}_{p',c'}\right]}_{\text{guaranteed pre-window updates}} + \underbrace{\left[\sum_{(p',c') \in \mathcal{S}_{p,c}} \mathbf{u}_{p',c'}\right]}_{\text{best-effort in-window updates}},$$

for some $\mathcal{S}_{p,c} \subseteq \mathcal{W}_{p,c} = \{1, ..., P\} \times \{c - s, ..., c + s - 1\}$ which is some subset of updates in the $2s$ window issued by all P workers during clock $c-s$ to $c+s-1$. The noisy view consists of (1) guaranteed pre-window updates for clock 1 to $c - s - 1$, and (2) best-effort updates indexed by $\mathcal{S}_{p,c}$.[4] We introduce a clock-major index t:

$$\tilde{\mathbf{x}}_t := \tilde{\mathbf{x}}_{(t \bmod P), \lfloor t/P \rfloor} \qquad \mathbf{u}_t := \mathbf{u}_{(t \bmod P), \lfloor t/P \rfloor}$$

and analogously for $\mathcal{S}_t$ and $\mathcal{W}_t$. We can now define a reference sequence (distinct from $\hat{\mathbf{x}}_t$ in VAP) which we informally refers to as the "true" sequence:

$$\mathbf{x}_t = \mathbf{x}_0 + \sum_{t'=0}^{t} \mathbf{u}_{t'} \tag{4}$$

The sum loops over workers (t mod P) and clocks $\lfloor t/P \rfloor$. Notice that this sequence is unrelated to the server view.

Theorem 3 *(SGD under SSP, convergence in expectation (Ho et al. 2013), Theorem 1) Given convex function $f(\boldsymbol{x}) = \sum_{t=1}^{T} f_t(\boldsymbol{x})$ with suitable conditions as in Theorem 1, we use gradient descent with updates $\boldsymbol{u}_t = -\eta_t \nabla f_t(\tilde{\boldsymbol{x}}_t)$ generated from noisy view $\tilde{\boldsymbol{x}}_t$ and $\eta_t = \frac{\eta}{\sqrt{t}}$. Then*

$$R[X] := \sum_{t=1}^{T} f_t(\tilde{\boldsymbol{x}}_t) - f(\boldsymbol{x}^*) = \mathcal{O}(\sqrt{T})$$

and thus $\frac{R[X]}{T} \to 0$ as $T \to \infty$.

Theorem 3 is the SSP-counterpart of Theorem 1. The analysis of Theorem 3 only uses the worst-case SSP bounds. However, in practice many updates are much less stale than the SSP bound. In Fig 2 (left) both implementations have only a small portion of updates with maximum staleness.

[4]In contrast to (Ho et al. 2013), we do not assume read-my-write.

We now use moment statistics to further characterize the convergence. We begin by decomposing $\tilde{\mathbf{x}}_t$. Let $\bar{u}_t := \frac{1}{P(2s+1)} \sum_{t' \in \mathcal{W}_t} ||\mathbf{u}_{t'}||_2$ be the average of ℓ_2 of the updates. We can write the noisy view $\tilde{\mathbf{x}}_t$ as

$$\tilde{\mathbf{x}}_t = \mathbf{x}_t + \bar{u}_t \boldsymbol{\gamma}_t \tag{5}$$

where $\boldsymbol{\gamma}_t \in \mathbb{R}^d$ is a vector of random variables whose randomness lies in the network communication. Note that the decomposition in eq. 5 is always possible since $\bar{u}_t = 0$ iff $\mathbf{u}_{t'} = \mathbf{0}$ for all updates $\mathbf{u}_{t'}$ in the $2s$ window. Using SSP we can bound $\bar{u}_t$ and $\boldsymbol{\gamma}_t$:

Lemma 4 *$\bar{u}_t \leq \frac{\eta}{\sqrt{t}} L$ and $\gamma_t := ||\boldsymbol{\gamma}_t||_2 \leq P(2s+1)$.*

Therefore $\mu_\gamma = \mathbb{E}[\gamma_t]$ and $\sigma_\gamma = var(\gamma_t)$ are well-defined. We now provide an exponential tail-bound characterizing convergence in finite steps.

Theorem 5 *(SGD under SSP, convergence in probability) Given convex function $f(\boldsymbol{x}) = \sum_{t=1}^{T} f_t(\boldsymbol{x})$ such that components f_t are also convex. We search for minimizer $\boldsymbol{x}^*$ via gradient descent on each component ∇f_t under SSP with staleness parameter s and P workers. Let $\boldsymbol{u}_t := -\eta_t \nabla_t f_t(\tilde{\boldsymbol{x}}_t)$ with $\eta_t = \frac{\eta}{\sqrt{t}}$. Under suitable conditions (f_t are L-Lipschitz and bounded divergence $D(x||x') \leq F^2$), we have*

$$P\left[\frac{R[X]}{T} - \frac{1}{\sqrt{T}}\left(\eta L^2 + \frac{F^2}{\eta} + 2\eta L^2 \mu_\gamma\right) \geq \tau\right] \leq \exp\left\{\frac{-T\tau^2}{2\bar{\eta}_T \sigma_\gamma + \frac{2}{3}\eta L^2 (2s+1) P \tau}\right\}$$

where $R[X] := \sum_{t=1}^{T} f_t(\tilde{x}_t) - f(x^*)$, and $\bar{\eta}_T = \frac{\eta^2 L^4 (\ln T + 1)}{T} = o(T)$.

This means that $\frac{R[X]}{T}$ converges to $O(T^{-1/2})$ in probability with an exponential tail-bound. Also note that the convergence is faster for smaller μ_γ and σ_γ.

We need a few mild assumptions on the staleness γ_t in order to derive variance bound:

Assumption 1 *γ_t are i.i.d. random variable with well-defined mean μ_γ and variance σ_γ.*

Assumption 2 *γ_t is independent of $\boldsymbol{x}_t$ and $\boldsymbol{u}_t$.*

Assumption 1 is satisfied by Lemma 4, while Assumption 2 is valid since γ_t are only influenced by the computational load and network bandwidth at each machine, which are themselves independent of the actual values of the computation ($\mathbf{u}_t$ and $\mathbf{x}_t$). We now present an SSP variance bound.

Theorem 6 *(SGD under SSP, decreasing variance) Given the setup in Theorem 5 and assumption 1-2. Further assume that $f(\boldsymbol{x})$ has bounded and invertible Hessian Ω^* at optimum $\boldsymbol{x}^*$ and γ_t is bounded. Let $\mathrm{Var}_t := \mathbb{E}[\tilde{\boldsymbol{x}}_t^2] - \mathbb{E}[\tilde{\boldsymbol{x}}_t]^2$, $\boldsymbol{g}_t = \nabla f_t(\tilde{\boldsymbol{x}}_t)$ then for $\tilde{\boldsymbol{x}}_t$ near the optima $\boldsymbol{x}^*$ such that $\rho_t = ||\tilde{\boldsymbol{x}}_t - \boldsymbol{x}^*||$ and $\xi_t = ||\boldsymbol{g}_t|| - ||\boldsymbol{g}_{t+1}||$ are small:*

$$\mathrm{Var}_{t+1} = \mathrm{Var}_t - 2\eta_t cov(\boldsymbol{x}_t, \mathbb{E}^{\Delta_t}[\boldsymbol{g}_t]) + \mathcal{O}(\eta_t \xi_t) \tag{6}$$

$$+ \mathcal{O}(\eta_t^2 \rho_t^2) + \mathcal{O}^*_{\gamma_t} \tag{7}$$

where covariance $cov(\boldsymbol{a}, \boldsymbol{b}) := \mathbb{E}[\boldsymbol{a}^T \boldsymbol{b}] - \mathbb{E}[\boldsymbol{a}^T]\mathbb{E}[\boldsymbol{b}]$ uses inner product. $\mathcal{O}^*_{\gamma_t}$ are high order (≥ 5th) terms involving $\gamma_t = ||\boldsymbol{\gamma}_t||_\infty$. Δ_t is a random variable capturing the randomness of update $\boldsymbol{u}_t$ conditioned on $\boldsymbol{x}_t$ (see appendix).

As argued before, $cov(\mathbf{x}_t, \mathbb{E}^{\Delta_t}[\boldsymbol{g}_t]) \geq 0$ in general. Therefore the theorem implies that Var_t monotonically decreases over time when SGD is close to an optima.

Comparison of VAP and ESSP

From Theorem 2 and 6 we see that both VAP and (E)SSP decrease the variance of the model parameters. However, VAP convergence is much more sensitive to its tuning parameter (the VAP threshold) than (E)SSP, whose tuning parameter is the staleness s. This is evident from the $O(\delta_t)$ term in Eq. 3, which is bounded by the VAP threshold. In contrast, (E)SSP's variance only involves staleness γ_t in high order terms $\mathcal{O}^*_{\gamma_t}$ (Eq. 7), where γ_t is bounded by staleness. This implies that staleness-induced variance vanishes quickly in (E)SSP. The main reason for (E)SSP's weak dependency on staleness is because it "factors in" the SGD step size: as the algorithm approaches an optimum, the updates automatically become more fine-grained (i.e. their magnitude decreases), which is conducive for lowering variance. On the other hand, the VAP threshold forces a minimum size on updates, and without adjusting this threshold, the VAP updates cannot become more fine-grained.

An intuitive analogy is that of postmen: VAP is like a postman who only ever delivers mail above a certain weight threshold δ. (E)SSP is like a postman who delivers mail late, but no later than s days. Intuitively, the (E)SSP postman is more reliable than the VAP postman due to his regularity. The only way for the VAP postman to be reliable, is to decrease the weight threshold. This is important when the algorithm approaches convergence, because the algorithm's updates become diminishingly small. However, there are two drawbacks to decreasing δ: first, much like step-size tuning, it must be done at a carefully controlled rate — this requires either specific knowledge about the ML problem, or a sophisticated, automatic scheme (that may also be domain-specific). Second, as δ decreases, VAP produces more frequent communication and reduces the throughput.

In contrast to VAP, ESSP does not suffer as much from these drawbacks, because: (1) the SSP family has a weaker theoretical dependency on the staleness threshold (than VAP does on its value-bound threshold), thus it is usually unnecessary to decrease the staleness as the ML algorithm approaches convergence. This is evidenced by (Ho et al. 2013), which achieved stable convergence even though they did not decrease staleness gradually during ML algorithm execution. (2) Because ESSP proactively pushes out fresh parameter values, the distribution of stale reads is usually close to zero-staleness, regardless of the actual staleness threshold used (see Fig. 2) — hence, fine-grained tuning of the staleness threshold is rarely necessary under ESSP.

ESSPTable: An efficient ESSP System

Our theory suggests that an ESSP implementation using eager parameter updates should outperform a SSP implementation using stalest parameters, when network

bandwidth is sufficient. To verify this, we implement ESSP in Parameter Server (PS), which we call ESSPTable.

PS Interface: ESSPTable organizes the model parameters as tables, where each parameter is assigned with a row ID and column ID. Rows are the unit of parameter communication between servers and clients; ML applications can use one of the default types or customize the row data-structure for maximal flexibility. For example, in LDA, rows are word-topic vectors. In MF, rows in a table are rows in the factor matrices.

In ESSPTable each computation thread is regarded as a worker by the system. The computation threads execute application logic and access the global parameters stored in ESSPTable through a key-value store interface—read a table-row via `GET` and write via `INC` (increment, i.e. addition). Once a computation thread completes a clock tick, it notifies the system via `CLOCK`, which increments the worker's clock by 1. As required by the SSP consistency , a `READ` issued by a worker at clock c is guaranteed to observe all updates generated in clock $[0, c-s-1]$, where s is the user-defined staleness threshold.

Ensuring Consistency: The ESSPTable client library caches previously accessed parameters. When a computation thread issues a `GET` request, it first checks the local cache for the requested parameters. If the requested parameter is not found in local cache, a read request is sent to the server and a call-back is registered at the server.

Each parameter in the client local cache is associated with a clock c_i. $c_i = t$ means that all updates from all workers generated before clock t have already been applied to this parameter. c_i is compared with the worker's clock c_p. Only if $c_i > c_p - s$, the requested parameter is returned to the worker. Otherwise, the reader thread is blocked until an up-to-date parameter is received from the server.

Communication Protocol: The updates generated by computation threads are coalesced since they are commutative and associative. These updates are sent to the server at the end of each clock tick. The server sends updated parameters to the client through call-backs. When a client request a table-row for the first time, it registers a call-back on the server. This is the only time the client makes read request to the server. Subsequently, when a server table's clock advances from getting the clock tick from all clients, it pushes out the table-rows to the respective registered clients. This differs from the SSPTable in (Ho et al. 2013) where the server passively sends out updates upon client's read request (which happens each time a client's local cache becomes too stale). The call-back mechanism exploits the fact that computation threads often revisit the same parameters in iterative-convergent algorithms, and thus the server can push out table-rows to registered clients without clients' explicit request. Our server-push model causes more eager communication and thus lower empirical staleness than SSPTable in (Ho et al. 2013) as shown in Fig. 2 (left).

We empirically observed that the time needed to communicate the coalesced updates accumulated in one clock is usually less than the computation time. Thus computation threads usually observe parameters with staleness 1 regardless of the user-specified staleness threshold s. That relieves the burden of staleness tuning. Also, since the server pushes out updated parameters to registered clients in batches, it reduces the overall latency from sending each parameter separately upon clients' requests (which is the case in SSPTable). This improvement is shown in our experiments.

Experiments

We show that ESSP improves the speed and quality of convergence (versus SSP) for collapsed gibbs sampling in topic model and stochastic gradient descent (SGD) in matrix factorization. Furthermore, ESSP is robust against the staleness setting, relieving the user from worrying about an additional tuning parameter. The experimental setups are:

- **ML Models and algorithms:** LDA topic modeling (using a sparsified collapsed Gibbs sampling in (Yao, Mimno, and McCallum 2009)) and Matrix Factorization (using stochastic gradient descent (Koren 2009)). Both algorithms are implemented using ESSPTable's interface. For LDA we use 50% minibatch in each `Clock()` call, and we use log-likelihood as measure of training quality. For MF we use 1% and 10% minibatch in each `Clock()` and record the squared loss instead of the ℓ_2-penalized objective. The step size for MF is chosen to be large while the algorithm still converges with staleness 0.
- **Datasets** Topic model: New York Times ($N = 100m$ tokens, $V = 100k$ vocabularies, and $K = 100$ topics). Matrix factorization: Netflix dataset ($480k$ by $18k$ matrix with $100m$ nonzeros.) Unless stated otherwise, we use rank $K = 100$ and regularization parameter $\lambda = 0.1$.
- **Compute cluster** Matrix factorization experiments were run on 64 nodes, each with 2 cores and 16GB RAM, connected via 1Gbps ethernet. LDA experiments were run on 8 nodes, each with 64 cores and 128GB memory, connected via 1Gbps ethernet.

Speed of Convergence: Figure 3 shows the objective over iteration and time for LDA and matrix factorization. In both cases ESSP converges faster or comparable to SSP with respect to iteration and run time. The speed up over iteration is due to the reduced staleness as shown in the staleness profile (Figure 2, left). This is consistent with the fact that in SSP, computation making use of fresh data makes more progress (Ho et al. 2013). Also it is worth pointing out that large staleness values help SSP substantially but much less so for ESSP because ESSP is less sensitive to staleness.

Robustness to Staleness: One important tuning knob in SGD-type of algorithms are the step size. Step sizes that are too small leads to slow convergence, while step sizes that are too large cause divergence. The problem of step size tuning is aggravated in the distributed settings, where staleness could aggregate the updates in a non-deterministic manner, causing unpredictable performance (dependent on network congestion and the machine speeds). In the case of MF, SSP diverges under high staleness, as staleness effectively increases the step size. However, ESSP is robust across all investigated staleness values due to the concentrated staleness profile (Fig. 2, left). For some high SSP staleness, the convergence is "shaky" due to the variance introduced by staleness. ESSP produces lower variance for all staleness settings, consistent with our theoretical analyses. This

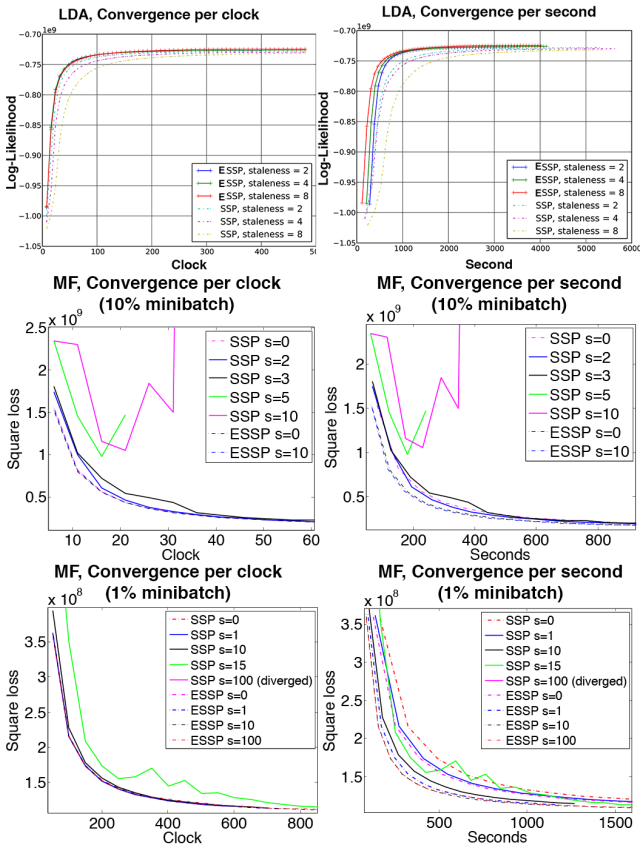

Figure 3: **Experimental Results.** The convergence speed per iteration and per second for LDA and MF.

improvement largely reduced the need for user to tune the staleness parameter introduced in SSP.

System Opportunity In addition to faster convergence per iteration, ESSP provides opportunities for system to optimize the communication. By sending updates preemptively, ESSP not only reduces the staleness but also reduces the chance of client threads being blocked to wait for updates; in essence, ESSP is a more "pipelined" version of SSP. Figure 2 (right) shows the breakdown of communication and compuation time for varying staleness. This contributes to the overall convergence per second in Figure 3, where ESSP has larger margin of speed gain over SSP than the convergence per iteration.

Related Work and Discussion

Existing software that is tailored towards *distributed* (rather than merely single-machine parallel), scalable ML can be roughly grouped into two categories: general-purpose, programmable libraries or frameworks such as GraphLab (Low et al. 2010) and Parameter Servers (PSes) (Ho et al. 2013; Li et al. 2013), or special-purpose solvers tailored to specific categories of ML applications: CCD++ (Yu et al. 2012) for Matrix Factorization, Vowpal Wabbit for regression/classification problems via stochastic optimization (Langford, Li, and Strehl 2007), and Yahoo LDA as well as Google plda for topic modeling (Wang et al. 2009).

The primary differences between the general-purpose frameworks (including this work) and the special-purpose solvers are (1) the former are user-programmable and can be extended to handle arbitrary ML applications, while the latter are non-programmable and restricted to predefined ML applications; (2) because the former must support arbitrary ML programs, their focus is on improving the "systems" code (notably, communication and synchronization protocols) to increase the efficiency of *all ML algorithms*, particularly through the careful design of consistency models (graph consistency in GraphLab; iteration/value-bounded consistency in PSes) — in contrast, the special-purpose systems combine both systems code improvements and *algorithmic* (i.e. mathematical) improvements tailor-made for their specific category of ML applications.

As a paper about general-purpose distributed ML, we focus on consistency models and systems code, and we deliberately use (relatively) simple algorithms for our benchmark applications, for two reasons: (1) to provide a fair comparison, we must match the code/algorithmic complexity of the benchmarks for other frameworks like GraphLab and SSP PS (Ho et al. 2013) (2) a general-purpose ML framework should not depend on highly-specialized algorithmic techniques tailored only to specific ML categories. General-purpose frameworks should *democratize distributed ML* in a way that special-purpose solvers cannot, by enabling those ML applications that have been under-served by the distributed ML research community to benefit from cluster computing. Since our benchmark applications are kept algorithmically simple, they are unlikely to beat the special-purpose solvers in running time — but we note that many algorithmic techniques featured in those solvers can be applied to our benchmark applications due to the general-purpose nature of PS programming.

In (Li et al. 2013), the authors propose and implement a PS consistency model that has similar theoretical guarantees to the ideal VAP model presented herein. However, we note that their implementation does not strictly enforce the conditions of their consistency model. Their implementation implicitly assumes zero latency for transmission over network, while in a real cluster, there could be arbitrarily long network delay. In their system, reads do not wait for delayed updates, so a worker may compute with highly inconsistent parameters in the case of congested network.

On the wider subject of Big Data, Hadoop (Borthakur 2007) and Spark (Zaharia et al. 2010) are popular programming frameworks which ML applications have been developed on. To our knowledge, there is no work showing that Hadoop or Spark have superior ML algorithm performance compared to frameworks designed for ML like GraphLab and PSes (let alone the special-purpose solvers). The main difference is that Hadoop/Spark only feature strict consistency, and do not support flexible consistency models like graph- or bounded-consistency; but Hadoop and Spark ensure program portability, reliability and fault tolerance at a level that GraphLab and PSes have yet to match.

Acknowledgement

This work is supported by DARPA XDATA FA87501220324, NSF IIS1447676, and ONR N000141410684.

References

Ahmed, A.; Aly, M.; Gonzalez, J.; Narayanamurthy, S.; and Smola, A. J. 2012. Scalable inference in latent variable models. In *WSDM*, 123–132.

Borthakur, D. 2007. The hadoop distributed file system: Architecture and design. *Hadoop Project Website* 11:21.

Chilimbi, T.; Suzue, Y.; Apacible, J.; and Kalyanaraman, K. 2014. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 571–582. Broomfield, CO: USENIX Association.

Cipar, J.; Ho, Q.; Kim, J. K.; Lee, S.; Ganger, G. R.; Gibson, G.; Keeton, K.; and Xing, E. 2013. Solving the straggler problem with bounded staleness. In *HotOS '13*. Usenix.

Cui, H.; Cipar, J.; Ho, Q.; Kim, J. K.; Lee, S.; Kumar, A.; Wei, J.; Dai, W.; Ganger, G. R.; Gibbons, P. B.; Gibson, G. A.; and Xing, E. P. 2014. Exploiting bounded staleness to speed up big data analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 37–48. Philadelphia, PA: USENIX Association.

Dean, J.; Corrado, G.; Monga, R.; Chen, K.; Devin, M.; Le, Q.; Mao, M.; Ranzato, M.; Senior, A.; Tucker, P.; Yang, K.; and Ng, A. 2012. Large scale distributed deep networks. In *NIPS 2012*.

Gemulla, R.; Nijkamp, E.; Haas, P. J.; and Sismanis, Y. 2011. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, 69–77. ACM.

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

Koren, Y. 2009. Matrix factorization techniques for recommender systems. *IEEE Computer* 42(8):30–37.

Kumar, A.; Beutel, A.; Ho, Q.; and Xing, E. P. 2014. Fugue: Slow-worker-agnostic distributed learning for big models on big data. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, 531–539.

Langford, J.; Li, L.; and Strehl, A. 2007. Vowpal wabbit online learning project.

Lee, S.; Kim, J. K.; Zheng, X.; Ho, Q.; Gibson, G. A.; and Xing, E. P. 2014. Primitives for dynamic big model parallelism. In *Advances in Neural Information Processing Systems (NIPS)*.

Li, M.; Yang, L. Z. Z.; Xia, A. L. F.; Andersen, D. G.; and Smola, A. 2013. Parameter server for distributed machine learning. *NIPS workshop*.

Li, M.; Andersen, D. G.; Park, J. W.; Smola, A. J.; Ahmed, A.; Josifovski, V.; Long, J.; Shekita, E. J.; and Su, B.-Y. 2014. Scaling distributed machine learning with the parameter server. In *Operating Systems Design and Implementation (OSDI)*.

Low, Y.; Gonzalez, J.; Kyrola, A.; Bickson, D.; Guestrin, C.; and Hellerstein, J. M. 2010. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*.

Niu, F.; Recht, B.; Ré, C.; and Wright, S. J. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*.

Wang, Y.; Bai, H.; Stanton, M.; Chen, W.-Y.; and Chang, E. Y. 2009. Plda: Parallel latent dirichlet allocation for large-scale applications. In *Proceedings of the 5th International Conference on Algorithmic Aspects in Information and Management*, AAIM '09, 301–314. Berlin, Heidelberg: Springer-Verlag.

Yao, L.; Mimno, D.; and McCallum, A. 2009. Efficient methods for topic model inference on streaming document collections. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, 937–946. New York, NY, USA: ACM.

Yu, H.-F.; Hsieh, C.-J.; Si, S.; and Dhillon, I. S. 2012. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *ICDM*, 765–774.

Zaharia, M.; Chowdhury, N. M. M.; Franklin, M.; Shenker, S.; and Stoica, I. 2010. Spark: Cluster computing with working sets. Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley.