Heterogeneity-aware Distributed Parameter Servers

Jiawei Jiang† Bin Cui† Ce Zhang‡ Lele Yu†

†School of EECS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University
‡Department of Computer Science, ETH Zürich
†{blue.jwjiang, bin.cui, leleyu}@pku.edu.cn ‡ce.zhang@inf.ethz.ch

ABSTRACT

We study distributed machine learning in *heterogeneous environments* in this work. We first conduct a systematic study of existing systems running distributed stochastic gradient descent; we find that, although these systems work well in homogeneous environments, they can suffer performance degradation, sometimes up to 10×, in heterogeneous environments where stragglers are common because their synchronization protocols cannot fit a heterogeneous setting. Our first contribution is a heterogeneity-aware algorithm that uses a constant learning rate schedule for updates before adding them to the global parameter. This allows us to suppress stragglers' harm on robust convergence. As a further improvement, our second contribution is a more sophisticated learning rate schedule that takes into consideration the delayed information of each update. We theoretically prove the valid convergence of both approaches and implement a prototype system in the production cluster of our industrial partner Tencent Inc. We validate the performance of this prototype using a range of machine-learning workloads. Our prototype is 2-12× faster than other state-of-the-art systems, such as Spark, Petuum, and TensorFlow; and our proposed algorithm takes up to 6× fewer iterations to converge.

1. INTRODUCTION

Large-scale machine learning (ML) has become a crucial issue in both academia [28, 8] and industry [4, 24]. As the size of many industrial-scale datasets often comprises terabytes or even far more, it is often necessary to partition data over commodity machines and train ML models in a distributed way [42]. This problem has been intensively studied by the database and system communities [13], resulting in a range of popular systems such as Mahout [2], MLlib [3], Pregel [32], and GraphX [43]. These systems have two things in common. First, they all use a variant of *Stochastic Gradient Descent (SGD)* as their workhorse optimization algorithm due to its small memory cost and rapid convergence rate [8]. Second, most of these systems implicitly assume a *homogeneous* cluster, that is, workers possess similar computation and network capacity [5]. This assumption, however, does not hold for many of the cases we tried to support in real-world clusters.

Use Case 1 (Network and Computation Heterogeneity)**.** One of our industrial partners owns a YARN [40] cluster consisting of 8,800 commodity machines located in several data centers across the country. Data transmission between data centers is normally slower than within data centers. Even within one data center, though, heterogeneous situations still happen due to complex network topology, distinct capabilities of switches, and network congestion. Worse, different machines also contain different generations of CPUs. To get enough workers, jobs submitted by the analytics team often involve network and computation heterogeneity, which yields some slow workers, a phenomenon called *straggler*.

SIGMOD'17, May 14-19, 2017, Chicago, Illinois, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3035918.3035933

Use Case 2 (Resource Sharing)**.** For another industrial partner, many users share a cluster. Researchers and engineers submit 1.2 million applications in a single day, with at most 6,000 applications running simultaneously. Running jobs inevitably compete for scarce hardware resources on the same machine; therefore different instances of the same job often have very different execution time.

Use Case 3 (Spot Instances on the Cloud)**.** Amazon EC2 [1] provides a variety of instances for users, and often the cheapest way to get an HPC cluster there is to use spot instances. To minimize cost, a rented cluster often contains different types of instances (e.g., m4.large with two cores and c4.4xlarge with 16 cores). When one user tries to run a job on such a cluster, the workers equipped with inferior hardware generally need more time to compute the same amount of data than those with more advanced hardware. [1]

Motivated by these real-world cases, we conduct a systematic study of running distributed SGD over these *heterogeneous* clusters. The heterogeneity of the cluster is measured by the speed gap between the fastest worker and the slowest worker. In the presence of heterogeneity, we study the question that *what protocol should these machines use to communicate and synchronize?* Unsurprisingly, systems designed for homogeneous clusters can perform poorly in a heterogeneous setting. We first empirically study existing systems and find that this performance degradation is caused by the implicit assumption that all workers, regardless of speed, have the same impact to the global parameter via the *same global learning rate*. To solve this problem, we propose two heterogeneity-aware learning rate schedules—ConSGD, which uses a constant learning rate schedule, and DynSGD, which uses a dynamic learning rate schedule. They produce an improvement over state-of-the-art algorithms of 2-12×. We also prove a theoretical-convergence upper bound for our approaches. The prototype we developed has been deployed to train up to 200 GB of data over heterogeneous clusters. Motivated by the techniques in the database community, our implemented parameter server supports multi-version control, efficient parameter partition, and partition synchronization.

[1]There are some other problems in the cloud environment. For instance, the fault tolerance mechanism to address the transient nature of spot instances. But this work focuses on the straggler problem in the cloud environment.

Overview of Technical Contributions

We first provide a preview of distributed SGD to set the context and then describe each of our technical contributions.

Distributed SGD. We focus on training ML models with distributed SGD, which has been employed to train many ML models such as Logistic Regression (LR), Support Vector Machine (SVM), and Deep Neural Networks [49, 18]. The input is a *training dataset* that contains N tuples $\{x_i \in \mathbb{R}^n\}$ and their corresponding labels $\{y_i \in \mathbb{R}\}$. The goal is to find a model $w \in \mathbb{R}^n$ that minimizes the sum of some convex loss function f over all input tuples. To solve this problem, SGD scans the dataset while updating w every time it processes a tuple; we call each update an *iteration*. Once it makes a pass over the entire dataset, we say SGD has finished an *epoch*.

With distributed settings, one way to distribute the workload is via a *parameter server (PS)* architecture [7, 17, 16, 28]—in it, a global parameter w is stored on a parameter server and each worker has a parameter replica. Each worker processes tuples independently and proposes updates to the PS. There are variants of SGD that deal with parameter synchronization between the PS and workers. In this paper, we call the synchronization interval the *clock*.

Anatomy of Existing Systems. We systemically study existing ML systems in heterogeneous environments. They cover the spectrum of the reported popular distributed SGD variants.

BSP Systems: Bulk Synchronous Parallel. Systems like Spark [45], MLlib [3], and GraphX [43] synchronize distributed parameter replicas under the BSP protocol that one worker cannot continue to the next clock until the PS receives all parameter replicas and broadcasts a newly updated global parameter. In the presence of stragglers, the overall processing time is determined by the slowest worker. Thus, as the degree of heterogeneity increases by 2×, the performance degrades by up to 2× in our experiment.

ASP Systems: Asynchronous Parallel. Systems like DistBelief [17], Hogwild! [37], and TensorFlow [4] use an ASP protocol where workers proceed without waiting for each other, making ASP systems often faster than BSP systems in homogeneous clusters. However, when some stragglers are significantly slower than other machines, ASP cannot guarantee correct convergence [37, 47]. Empirically, we find that systems with ASP can be up to 10× slower when the heterogeneity increases.

SSP Systems: Stale Synchronous Parallel. Some systems [16, 41, 38] exploit the SSP [23] strategy where the fastest worker cannot exceed the slowest one more than a predefined *staleness*. Although this explicit bound on staleness is potentially a good candidate for a heterogeneous environment, existing systems still suffer from heterogeneity—in their implementations, the PS generally *adds* updates directly to the global parameter. In heterogeneous environments, the global parameter and the replica of a straggler can be inconsistent, thus the update generated by a straggler might push the global parameter to some place distant from the optimality at a time when the global parameter is already near the optimality. As a result, SSP can run up to 5.5× slower in a heterogeneous cluster.

Summary. We intuitively and empirically find that, although distributed SGD variants of existing ML systems are suitable for homogeneous environments where stragglers are infrequent, they fail to fit heterogeneous clusters. Intrinsically, the SSP protocol reveals its benefits in the context of large-scale ML.

Constant Learning Rate Schedule under SSP. Our first contribution is a simple algorithm, CONSGD, that improves performance over the standard SGD under the SSP protocol, namely SSPSGD. Unlike SSPSGD, which accumulatively adds local updates to the global parameter, CONSGD introduces a constant *global learning rate* and multiplies it to each local update before accumulation. The intuition behind CONSGD is that by choosing a global learning rate $\lambda_g \in (0, 1)$ we can reduce the disturbance caused by abnormal delayed updates from stragglers and can choose larger local learning rate to enhance the local convergence of the worker. To choose this constant global learning rate, we use a heuristic that sets the constant to be inversely proportional to the number of workers. We find that this nonparametric heuristic works well in practice and is theoretically sound. In practice, CONSGD can alleviate the unstable convergence problem of SSPSGD and thereby increase system efficiency. The empirical results in Section 7 show that CONSGD makes the system up to 6× faster than SSPSGD. In theory, we prove that CONSGD can asymptotically achieve the same upper convergence bound as SSPSGD.

Summary. We develop a variant of SSPSGD to restrain the harm of delayed updates and assure fast convergence meanwhile by decoupling the learning rate into the local learning rate of the worker and the global learning rate (a decay factor) of the PS. This strategy, despite its simplicity, significantly improves the performance and enjoys the same, if not better, theoretical guarantee.

Dynamic Learning Rate Schedule under SSP. CONSGD helps alleviate the impact of stragglers by choosing a constant global learning rate for all local updates from workers regardless of their speed. Our second contribution is a novel mechanism, which we call DYNSGD, to further improve performance over CONSGD using a dynamic global learning rate schedule that takes into consideration the delayed information of each update. We first define the *staleness* of local update and define a global learning rate schedule that gives a smaller weight to a larger staleness. We then prove theoretically that DYNSGD has a slightly tighter upper bound than CONSGD and SSPSGD. We next develop a data structure to efficiently support this dynamic schedule, one that only incurs 3% overhead in terms of storage space. Empirically, DYNSGD can be up to 12× faster than SSPSGD.

Summary. We improve over CONSGD by a dynamic learning rate schedule that can be efficiently calculated with an auxiliary data structure. Since DYNSGD can maintain individual learning rate for every update according to the stale level, it further reduces the damage of delayed updates to obtain more robust convergence.

Data Management. The proposed DYNSGD needs to dynamically revise the global learning rate of previous updates. Motivated by the techniques of the database community, we implement a parameter server with multi-version control to accomplish the revision operation, which makes our prototype different from existing parameter server platforms [16, 17]. To achieve balanced query distribution and fast range query, we adopt a hybrid range-hash strategy to partition parameters. In addition, due to non-deterministic transmission, different partitions of the parameter might be unsynchronized when the parameter update and parameter query happen simultaneously. Unfortunately, existing ASP and SSP based systems do not task into consideration this problem. Facilitated by our multi-version parameter server, we introduce a version-based mechanism to achieve the partition synchronization.

Overview. The rest of the paper is organized as follows. We introduce preliminary materials in Section 2 and present our study of existing systems in Section 3. We describe CONSGD and DYNSGD in Section 4 and Section 5, respectively. We describe our data management in Section 6 and present experimental results in Section 7. We describe related work in Section 8 and conclude in Section 9.

2. PRELIMINARIES

Here we introduce preliminaries related to distributed SGD and the PS architecture that is popular in state-of-the-art ML systems.

2.1 Data, Execution, & Performance Models

We focus here on *training* a subclass of ML workloads that can be formalized as follows. The input is a *training set* that includes *training samples* and their *labels*: $\{(x_i \in \mathbb{R}^n, y_i \in \mathbb{R})\}$. The goal is to find a model w that best predicts each y_i given x_i. More precisely, for a (convex) *loss function* f, the goal is to solve

$$\arg\min_w \sum_i f(x_i, y_i, w)$$

Data Model. One popular data model, which we also adopt, decouples the data used during training into *immutable data* and *mutable data*. The training samples and labels are immutable, while the mutable predictive model w that we want to find will be updated frequently. Previous work has studied different data layouts for storing both types of data [46] as matrices and vectors.

Execution Model. Different training algorithms usually imply different execution models. In this work, we focus on a widely used algorithm in the family of first-order optimization methods, namely the *SGD* algorithm. The execution of SGD consists of multiple *clocks*, each of which contains one pass over all training samples. Inside each clock, SGD scans each training sample x_i, calculates the gradient $\nabla f(x_i, y_i, w)$, and updates the model with

$$w \leftarrow w - \eta \nabla f(x_i, y_i, w)$$

where η is the *learning rate*. One improvement over this baseline approach is mini-batch SGD in which we calculate the gradient for a *subset* of training samples instead of just one of them [8, 29].

Performance Model. The performance of SGD can be decoupled as *statistical efficiency* and *hardware efficiency* [46]. Statistical efficiency measures how many clocks (iterations or batches) are needed until convergence to a tolerance, and hardware efficiency measures the wall-clock time each clock can be carried out.

2.2 Distributed SGD

Distributed SGD with data parallelism distributes the calculation onto different machines. Therefore, we need a coordinator to synchronize the solution returned by each worker. The PS architecture is a popular choice [17, 16, 28, 7, 12], as illustrated in Figure 1. There are two types of nodes: parameter servers, which together store a global copy of the parameter w, and workers, each of which maintains a parameter replica and uses its assigned data shard to update the local parameter. Periodically, each worker pulls the global parameter from the PS and pushes its local updates to the PS.

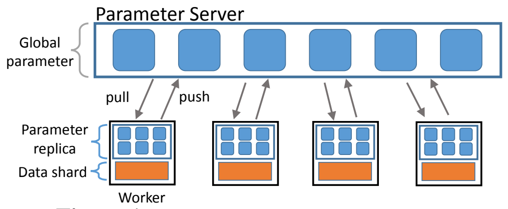


Figure 1: Architecture of Parameter Server

Stale Synchronous Parallel (SSP). There are different types of synchronizations between the parameter servers and workers. SSP is one popular protocol that also embraces many other popular coordination strategies as special cases. Algorithm 1 shows distributed SGD under the SSP protocol, namely SspSGD.

Worker: Each worker pushes its local update u_c^m to the PS (line 7). We use c_p to control the frequency of parameter request such that the worker does not need to request the global parameter if its local replica is not too stale. We modify c_p to the clock of the slowest worker every time we pull the global parameter.

Algorithm 1 SspSGD [23]

```
M: # workers, P: # parameter servers, N: # samples, C: # clocks
b: batch size, s: staleness threshold, w_0: initial parameter
Worker m = 1, ..., M:
1: Initialize $w_c^m \leftarrow Pull(m,0)$, $c_p \leftarrow 0$
2: for $c = 0$ to $C$:
3:     $u_c^m \leftarrow 0$
4:     for $batch = 1$ to $\frac{N}{bM}$:
5:         $u_c^m \leftarrow u_c^m - \eta \sum_{i=1}^{b} \nabla f(x_i, y_i, w_c^m)$
6:         $w_c^m \leftarrow w_c^m - \eta \sum_{i=1}^{b} \nabla f(x_i, y_i, w_c^m)$
7:     $Push(u_c^m)$
8:     if $c_p < c - s$:
9:         $(w_{c+1}^m, c_p) \leftarrow Pull(m, c+1)$
Parameter Server p = 1, ..., P:
1: Initialize $w \leftarrow w_0$, $c_{min} \leftarrow 0$
2: function $Push(u_c^m)$:
3:     $w \leftarrow w + u_c^m$
4:     if all the workers finish $c_{min}$:
5:         $c_{min} \leftarrow c_{min} + 1$
6: function $Pull(m, c)$:
7:     if $c \le c_{min} + s$:
8:         return $(w, c_{min})$
```

Parameter Server: The PS monitors the clock of the slowest worker, denoted by c_{min}. In the push function, if all the workers finish c_{min}, we increase it by one (line 4-5). When receiving a pull request from a worker, the PS sends the latest global parameter and c_{min}, but only if the worker satisfies the SSP constraint (line 7-8).

SSP is a flexible protocol in which we can vary the value of s to control the constraints degree. Specifically, choosing s=0 yields BSP, while ASP is obtained by choosing s=$+\infty$ and disabling c_p.

Since the parameter is partitioned across several nodes, the synchronization between different partitions should be considered when push and pull operations happen simultaneously. The partition synchronization is different from parameter synchronization tackled by BSP or SSP. We will elaborate this problem in Section 6.

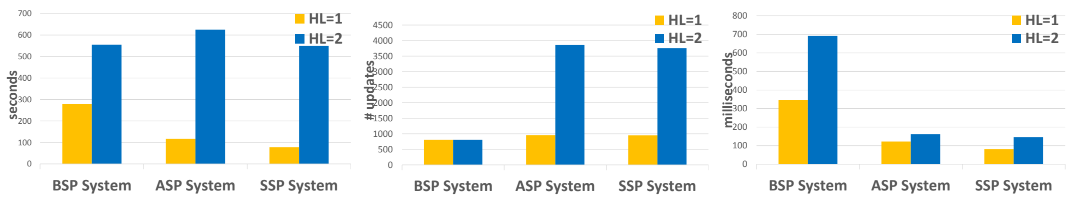
2.3 Modeling Heterogeneous Clusters

Modeling heterogeneous clusters has been a topic of interest for the distributed computing community [27, 44]. Because our algorithms do not need an explicit model of heterogeneity, we adopt a simple model. We decompose the run time of worker m into the computation time t_c^m, the transmission time t_t^m, and the waiting time t_w^m, where $t_c^m + t_t^m$ can represent the speed of worker m. Intuitively, a cluster is more heterogeneous if the speed gap between the fastest worker and the slowest worker is larger. As such, we define the heterogeneous level of a cluster by the ratio between the fastest worker f and the slowest worker s:

$$HL = \frac{t_c^f + t_t^f}{t_c^s + t_t^s} \tag{1}$$

3. ANATOMY OF EXISTING SYSTEMS

Here we study these questions: "*How do existing systems perform in heterogeneous clusters? If there is a performance degradation, what causes it?*" With distributed SGD, one major factor affecting overall performance is the synchronization protocol that the PS uses to consolidate incoming updates from the workers. Most of existing ML systems can be categorized into three types according to the synchronization protocols they use, namely BSP, ASP, and SSP. We pick the most popular systems for each type and then conduct an empirical study to validate the following hypothesis.



(a) total run time (b) # updates to converge (c) per-update time

Figure 2: Performance of existing systems in the presence of heterogeneity.

Hypothesis. When the heterogeneity of the cluster increases, all BSP, ASP, and SSP-based systems become slower due to slower statistical efficiency *and/or* hardware efficiency.

Metrics & Proxy. We measure heterogeneity via Eq. 1. We measure the end-to-end performance of the system as the wall-clock time a system needs to converge to a given tolerance. We further decouple the end-to-end performance into *statistical efficiency* and *hardware efficiency*. Statistical efficiency is measured as the number of updates the PS receives until convergence, and hardware efficiency is measured as the average time each update takes.

Protocol. The results of our study hold for a variety of datasets (see Section 7). In this section, we use a standard dataset [31] to illustrate these results. We train a logistic regression model on a 30-node cluster. The terminating objective is set at 0.2. In addition, if the evaluated system incorporates a PS architecture, we designate 10 machines to act as the PS. We activate the *sleep()* function in 20% workers to simulate the heterogeneous environment, and we increase the sleep time to increase heterogeneity. In addition, we use a grid search to tune the optimal hyperparameter.

Limitation. To control the level of heterogeneity, we artificially make *some workers* slower. In a real-world scenario, different workers will have different speeds, and this behavior is obviously more sophisticated than the study we conduct in this section. However, as we show in Section 7.3, even with clusters deployed by our industrial partner, we observe a similar phenomenon.

3.1 Systems Running BSP-based SGD

Result. We use Spark to evaluate the performance of a BSP system in a heterogeneous cluster. As shown in Figure 2, as the heterogeneous level increases, the statistical efficiency remains the same because each local parameter replica remains the same regardless of worker speed. On the other hand, the hardware efficiency decreases because the barrier of BSP stalls fast workers. Overall, the total run time increases $2\times$ when HL increases by $2\times$.

Discussion. Our empirical results suggest that, given homogeneous environments where all workers can finish a clock with comparable time, the BSP protocol works fine. Nevertheless, if the underlying environments are heterogeneous, some stragglers can be much slower than others. On this occasion, stragglers become the bottleneck and block the system.

3.2 Systems Running ASP-based SGD

Result. We use the ASP mode of Petuum [16] to evaluate the performance of ASP-based distributed SGD in a heterogeneous cluster. As shown in Figure 2, as the heterogeneous level increases from 1 to 2, the hardware efficiency slightly decreases since the stragglers need more time to generate updates. In addition, the statistical efficiency rapidly deteriorates, bringing at most a $4\times$ convergence degradation. Overall, the total run time increases by $6\times$.

Discussion. ASP systems cannot work well in heterogeneous environments when stragglers are prevailing. In theory, ASP-based distributed SGD cannot assure valid convergence when the speed gaps between workers are not bounded. We will analyze the reason for this performance decline together with SSP-based system.

3.3 Systems Running SSP-based SGD

As noted, the synchronization rule of BSP is too strict and ASP cannot guarantee correct convergence. The SSP protocol tries to balance these two strategies [23].

Result. We select Bösen to evaluate an SSP system's robustness against heterogeneity with s=10. As shown in Figure 2, the run time of the SSP system outperforms the BSP and ASP systems when HL=1 since SspSGD enhances hardware efficiency via decreasing synchronization frequency. But when HL increase to 2, the statistical efficiency declines by $4\times$ and the hardware efficiency declines by $1.8\times$. Overall, the total run time is $7\times$ slower.

Discussion. In the implementation of SspSGD in existing systems, the local updates of workers are directly added to the global parameter on the PS. Consequently, the global parameter can be abnormally modified by a delayed local update from a straggler.

We use the examples in Figures 3(a) and 3(b) to intuitively explain the reason. The initial parameter is w_0 and the circles represent the contour lines of objective function. $W1$, $W2$ and $W3$ push their local updates of the first and second clock to the PS, thus the global parameter becomes $(w_0{+}a{+}b{+}c{+}d{+}e{+}f)$, a place near the optimal point. However, $W4$, the straggler, is still running the first clock. The generated update g is abnormal to the current global parameter because g is computed upon the initial parameter. Thus, the global parameter is updated to a place far from the optimal point.

In other words, the accumulative approach of SspSGD impedes robust convergence when the parameter is approaching the optimal point because delayed local updates from stragglers might disturb the global parameter. Certainly, we can use very small learning rate to achieve more robust convergence. However, this strategy slows down the convergence on a single worker and makes SGD hard to converge to the optimality owing to too small local learning rate.

3.4 Summary

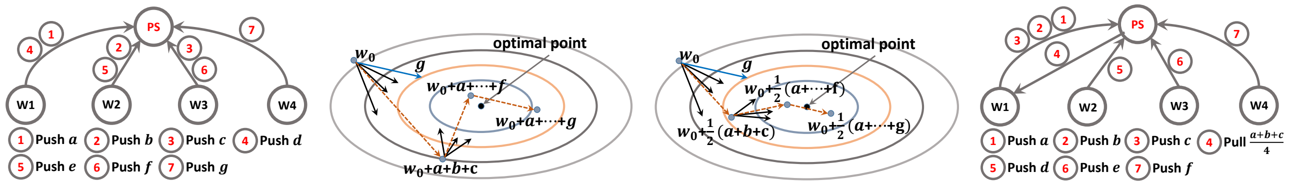
In general, the SSP protocol is a promising candidate owing to its efficient parameter exchange strategy. However, SspSGD encounters performance deterioration in a heterogeneous cluster because delayed updates from stragglers render the convergence unstable.

4. CONSGD

Motivated by our empirical study in Section 3, we present ConSGD, a simple but novel algorithm, to tackle the performance degradation caused by heterogeneity. We describe the basic version of ConSGD and an efficient hyperparameter-free heuristic. We then discuss the theoretical guarantee and limitations.

Algorithm. The basic idea behind ConSGD is simple—instead of accumulating local updates to the global parameter (line 3 of PS in Algorithm 1), we introduce a *global learning rate* $\lambda_g \in (0, 1)$, which can be seen as a decay factor, for this local update

$$w \leftarrow w + \lambda_g u_c^m.$$



(a) A synchronization example (b) Why SSPSGD cannot work well (c) Why CONSGD works better (d) Limitations of CONSGD

Figure 3: Illustration of SSPSGD and CONSGD over Heterogeneous Clusters.

As readers might expect, this change is very easy to implement given an SSP system. In fact, we only need to change a single line. However, as we will see in Section 7, this simple change can make the system up to 6× faster in presence of heterogeneity.

Intuition. Figure 3(c) illustrates why this simple twist works with λ_g=0.5. The update sequence is presented in Figure 3(a). When the delayed local update of $W4$ is pushed to the PS, we add $g/2$ to the global parameter instead of g in Figure 3(b). Therefore, the disturbance induced by the straggler is reduced, and hence more robust convergence is acquired. Note that if s=+∞ in SSP, this is equivalent to choosing a lower learning rate for ASP when asynchrony increases. This is a well-known trick for ASP. CONSGD can simply be seen as a generalization of this idea to SSP. But unlike ASP and SSPSGD, *we can use larger local learning for* CONSGD to improve the convergence speed on a single worker.

Hyperparameter-free Heuristic. One design decision to make is how to choose the global hyperparameter λ_g. Although it is possible to grid-search the optimal setting, this strategy incurs significant overhead. Our approach is a simple but theoretically sound heuristic that sets $\lambda_g=\frac{1}{M}$. The intuition is that BSP multiplies local updates by $\frac{1}{M}$ as $w_{c+1}=\frac{1}{M}\sum_{i=1}^{M}(w_c+u_c^i)=w_c+\frac{1}{M}\sum_{i=1}^{M}u_c^i$. Since $\lambda_g=\frac{1}{M}$ works for BSP, we reasonably guess that it works for SSP as well. As we will see in Section 7, this heuristic works across a range of datasets and only incurs at most a 1.2× penalty compared with an optimal global learning rate obtained by grid search.

Theoretical Analysis. Here we present the theorem concerning the upper bound of CONSGD's regret defined below with the hyperparameter-free heuristic and leave the proof to Appendix A.

Regret: We define the minimizer of $f(w)$ as w^*, and our goal is to find the bound of the regret $R[W]$ associated with a parameter sequence W=$\{w_1, w_2, ..., w_T\}$. T refers to the total processing clocks of all the workers, i.e., $C \times M$.

$$R[W] := \frac{1}{T}\sum_{t=1}^{T} f_t(w_t) - f(w^*), \;\; where\ f(w) := \frac{1}{T}\sum_{t=1}^{T} f_t(w)$$

We state two standard assumptions that the theoretical statement of convergence rate (clock vs. objective value) depends on.

Assumption 1. (L-Lipschitz function) For convex function $f_t(w)$, the subdifferentials $\|\nabla f_t(w)\|$ are bounded by some constant L.

Assumption 2. (Bounded diameter) For any $w, w' \in \mathbb{R}^n$ and some constant $F > 0$, $D(w\|w') = \frac{1}{2}\|w - w'\|^2 \leq F^2$ holds.

Proposition 1 (SSPSGD [23])**.** Define $u_t := -\eta_t \nabla f_t(\widetilde{w}_t) = -\eta_t \widetilde{g}_t$, $\eta_t = \frac{\sigma}{\sqrt{t}}$ where $\sigma = \frac{F}{L\sqrt{2(s+1)M}}$. Under Assumption 1, Assumption 2 and the technical conditions defined in Appendix A, we attain the bound of the regret:

$$R[W] \leq 4FL\sqrt{\frac{2(s+1)M}{T}} \qquad (2)$$

Theorem 1 (CONSGD)**.** Define $u_t := -\eta_t \nabla f_t(\widetilde{w}_t) = -\eta_t \widetilde{g}_t$, $\eta_t = \frac{\sigma}{\sqrt{t}}$ where $\sigma = \frac{F}{L\sqrt{2(s+1)M}}$. Under the same technical conditions as in Proposition 1, we attain the bound of the regret:

$$R[W] \leq (M+3)FL\sqrt{\frac{2(s+1)M}{T}} \qquad (3)$$

If $\sigma = \frac{MF}{L\sqrt{2(s+1)M}}$, we attain the bound of the regret:

$$R[W] \leq 3FL\sqrt{\frac{2(s+1)M}{T}} \qquad (4)$$

The upper bound we are able to prove for CONSGD is slightly tighter than SSPSGD; however, this does not tell us much about the relative performance of CONSGD and SSPSGD, which can only be answered via empirical validation. As we will see in Section 7.4, CONSGD can be up to 6× faster in convergence rate.

Discussion of Limitations. CONSGD cannot completely surmount the heterogeneity caused by stragglers since it employs a constant global learning rate λ_g. Figure 3(d) gives an example to show the limitation of CONSGD. Supposing that $W1$ is the fastest and it completes three clocks before the other three workers push their first update. The local parameter of $W1$ is (w_0+a+b+c). Nevertheless, when $W1$ pulls the global parameter, the local parameter is revised to $w_0+\frac{1}{4}(a+b+c)$. In this case, the local parameter of $W1$ experiences a large shrink. Therefore, although CONSGD relieves the unstable convergence problem, it might slow down the convergence of fast workers to some extent. The theoretical reason is that the number of local updates of clock c at a certain moment is smaller than M. Therefore, the expectation of the global parameter is not equal to the expectation of the local parameter. This kind of inconsistency necessarily damages convergence robustness.

Note that since DYNSGD proposed below outperforms CONSGD in most cases according to our empirical results, we consider that CONSGD is a building block for DYNSGD.

5. DYNSGD

CONSGD uses a single global learning rate for all workers regardless of their speed. In this section, we propose a novel mechanism (DYNSGD) to take advantage of a per-worker learning rate, design a data structure to dynamically maintain this learning rate for each worker, and prove theoretical convergence.

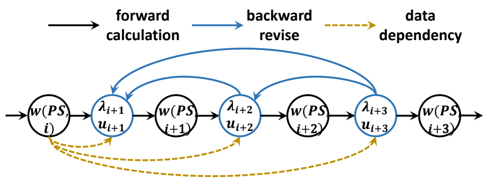
5.1 DYNSGD Algorithm

The fundamental reason for the convergence instability of SSPSGD and CONSGD is that both methods directly add local updates or decayed ones to the global parameter, leaving the local update's delayed information out of consideration.

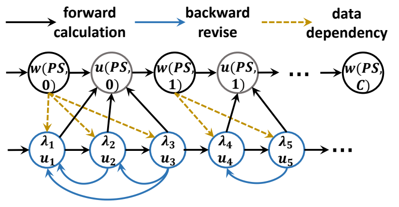
Abstract Model of a Parameter Server. We first describe an abstract model for a parameter server to provide context for describing the DYNSGD algorithm. We model a parameter server PS as a tuple $(\mathcal{S}, f_{clock}, f_{server}, \lambda, w_0)$, where $\mathcal{S}$ is a *stream of updates* $\{u_1u_2...\}$. Each update u_i comes from a certain machine at a certain clock (defined more precisely in the next paragraph), and the function $f_{server}(i)$ and $f_{clock}(i)$ store this information. We call the function λ a *learning rate schedule* that, for each update u_i, returns the learning rate $\lambda(i)$. w_0 is the initial model.

A pairing of a parameter server PS and a time i uniquely defines a global parameter. We call this mapping the *materialization* of a parameter server, denoted as $w(PS, i)$:

$$w(PS, i) = w_0 + \sum_{j=1}^{i} \lambda(j)u_j$$



(a) Calculation graph of DYNSGD



(b) Improved calculation graph of DYNSGD

Figure 4: Abstraction of DYNSGD

Staleness of Updates. We formalize the delay of each local update, and we call this the *staleness* of a local update. Let u_i be an update, function $f_{server}(i)$ returns the machine that proposed u_i; and $f_{clock}(i)$ returns an index such that the update u_i is calculated from the model defined by the materialization $w(PS, f_{clock}(i))$. Given this definition, we can define the staleness of an update u_i as

$$staleness(u_i) = |\{u_j : f_{clock}(j) = f_{clock}(i)\}|.$$

Intuitively, we define the staleness of an update u_i as the number of updates that rely on the same model replica.[2] Intuitively, if a single model replica is used by K different updates, then K–1 of them are stale—this motivates our definition of staleness.

DYNSGD Algorithm. Given the above model, we can describe our DYNSGD algorithm in a succinct way. We choose a learning rate schedule λ that is inversely proportional to the staleness of each update—the larger the staleness, the smaller the learning rate. As we show later, a theoretically sound choice, which we adopt, is

$$\lambda(i) = \frac{1}{staleness(u_i)}.$$

Calculation Graph. Given all the $w(PS, i)$ and u_i, we can use a graph to formalize the calculation procedure since a data dependency exists between one update and the parameter replica it relies on. Figure 4(a) presents an example where u_{i+1}, u_{i+2} and u_{i+3} are calculated with $w(PS, i)$. When u_{i+2} arrives at the PS after u_{i+1}, $staleness(u_{i+1})$ changes since $f_{clock}(u_{i+1})$=$f_{clock}(u_{i+2})$; therefore we should revise λ_{i+1} backward. Similarly, we should revise both λ_{i+1} and λ_{i+2} backward with the arrival of u_{i+3}.

According to the *i.i.d.* property of the training data, the expectations of updates are equal: $\mathbb{E}[u_{i+1}]$=$\mathbb{E}[u_{i+2}]$=$\mathbb{E}[u_{i+3}]$. Thus the three versions of the global parameter have the same expectation:

$$\mathbb{E}[w(PS, i{+}3)] = \mathbb{E}[w(PS, i) + \frac{u_{i+1}{+}u_{i+2}{+}u_{i+3}}{staleness(u_{i+3})}]$$

$$=\mathbb{E}[w(PS, i)] + \mathbb{E}[\frac{u_{i+1}{+}u_{i+2}{+}u_{i+3}}{3}] = \mathbb{E}[w(PS, i)] + \mathbb{E}[u_{i+3}]$$

$$=\mathbb{E}[w(PS, i{+}1)] = \mathbb{E}[w(PS, i{+}2)]$$

The above discussion demonstrates why DYNSGD can work. On the one hand, it guarantees the local parameter replica can stay consistent with the global parameter. On the other hand, rather than directly adding updates to the global parameter, we use a dynamic learning rate schedule to revise the global learning rate of former updates for the purpose of reducing the harm of delayed updates.

[2]There are other possible ways to define the staleness. For example, one can define the staleness of u_i as the longest sequence of updates from other machines immediately before u_i. Our definition of staleness is one that facilitates our description of our algorithm.

5.2 Theoretical Guarantee of DYNSGD

On the basis of Assumption 1 and Assumption 2, the following theorem gives the convergence bound of DYNSGD.

Theorem 2. (DYNSGD) Define $u_t := -\eta_t \nabla f_t(\widetilde{w}_t) = -\eta_t \widetilde{g}_t$, $\eta_t = \frac{\sigma}{\sqrt{t}}$ where $\sigma = \frac{F}{L\sqrt{2(s+1)M}}$, and $\mu = \mathbb{E}[staleness(u_i)]$. We attain the bound of the regret's expectation:

$$\mathbb{E}[R[W]] \leq (\mu + 3)FL\sqrt{\frac{2(s+1)M}{T}} \tag{5}$$

The proof of Theorem 2 is presented in Appendix B. It demonstrates that the upper convergence bound of DYNSGD is asymptotically the same as CONSGD in Eq.(3) with the same learning rate, while being better in the constant term since $1 \leq \mu \leq M$.

5.3 Implementation of DYNSGD

The challenge in implementing DYNSGD is how to calculate $staleness(-)$ efficiently. We can store every $w(PS, i)$ and u_i to accomplish this purpose; however, the cost of this scheme is too expensive. Alternatively, we introduce a data structure to efficiently calculate this function when given a stream of updates.

Data Structure. In Figure 4(a), since $w(PS, i{+}1)$, $w(PS, i{+}2)$ and $w(PS, i{+}3)$ own the same expectation, they can be seen as a same version of the global parameter at different moments. Therefore, we can merge them to a single node to obtain the improved calculation graph of DYNSGD shown in Figure 4(b). There are three major components in the proposed data structure, we describe them below individually and summarize the notations in Table 1.

Notations	Descriptions
$u(PS, v)$	the global update of version v
$V(m)$	the version of the local update from worker m
$S(v)$	the staleness of the local update stamped with version v

Table 1: Notations used in DYNSGD

Global update: There are C versions of the global parameter, and $w(PS, i{+}1)$=$w(PS, i)$+$u(PS, i)$ where $u(PS, i)$ denotes a *global update* on the PS that will be applied to $w(PS, i)$. For example, $u(PS, 0)$ is a "summary" of u_1, u_2 and u_3. Let c_{max} denote the clock of the fastest worker, the global parameter is composed of the initial parameter and multi-version global updates:

$$w = w_0 + \sum_{v=0}^{c_{max}-1} u(PS, v) \tag{6}$$

Version of local update: An update from worker m is stamped with a "version", denoted by $V(m)$, to be mapped to the corresponding global update.

Staleness of local update: According to our definition of the staleness, the staleness of a local update is the number of local updates sharing the same version. We use $S(v)$ to denote the staleness of the local update u_c^m stamped with version $V(m)$:

$$staleness(u_c^m) = S(V(m))$$

Algorithm Details. Here we elaborate on the processing details of DYNSGD. The pseudo code is presented in Algorithm 2. We also provide a revision example in Appendix C. As the process running on each worker remains the same, we focus on the "push" and "pull" operations that happen on the PS.

Push operation: When worker m pushes a local update u_c^m to the PS, the push function of the PS operates as follows:

1. Get the stamped version of u_c^m, denoted by v. (line 3)
2. Get the staleness of u_c^m, denoted by d. (line 4)
3. The global learning rate of u_c^m is $\frac{1}{d}$. Meanwhile, for previous local updates stamped with version v, we need to revise their global learning rate from $\frac{1}{d-1}$ to $\frac{1}{d}$. The variation of $u(PS, v)$ is (line 5):

$$\Delta u = \frac{1}{d}u_c^m + \frac{d{-}1}{d}u(PS, v) - u(PS, v) = \frac{1}{d}(u_c^m - u(PS, v))$$

Algorithm 2 DYNSGD

```
$c_{min}$: clock of the slowest worker, $c_{max}$: clock of the fastest worker
Parameter Server
 1: Initialize $c_{min} \leftarrow 0$, $V \leftarrow 0$, $S \leftarrow 1$, global parameter $w \leftarrow w_0$
 2: function $Push(u_c^m)$:
 3:     $v \leftarrow V(m)$
 4:     $d \leftarrow S(v)$
 5:     $\Delta u = \frac{1}{d}(u_c^m - u(PS, v))$
 6:     $w \leftarrow w + \Delta u$
 7:     $u(PS, v) \leftarrow u(PS, v) + \Delta u$
 8:     $S(v) \leftarrow S(v) + 1$
 9:     $V(m) \leftarrow V(m) + 1$
10:     if all the items in $V$ is larger than v:
11:         clear $u(PS, v)$ and $S(v)$
12:     if all the workers finish $c_{min}$:
13:         $c_{min} \leftarrow c_{min} + 1$
14:     if $c + 1 > c_{max}$:
15:         $c_{max} \leftarrow c$
16: function $Pull(m, c)$:
17:     if $c \leq c_{min} + s$:
18:         $V(m) \leftarrow c_{max}$
19:         return ($w$,$c_{min}$)
```

4. Revise the global parameter and the global update with Δu:

5. Increase $V(m)$ and $S(v)$ by 1. (line 8-9)

6. If all the items in V is larger than v meaning that $u(PS, v)$ will not be updated afterwards, we clean the corresponding memory of $u(PS, v)$ and $S(v)$ to reduce the memory cost. (line 10-11)

7. Increase c_{min} if all the workers finish c_{min}. (line 12-13)

8. Increase the maximal clock if c+1 $> c_{max}$. (line 14-15)

Pull operation: When worker m pulls the global parameter from the PS before clock c+1, if worker m is authorized under the SSP constraint, the PS return w and c_{min}. We also modify $V(m)$, the stamped version of worker m's update, to c_{max} since there are currently c_{max} versions of global update. (line 17-19)

Space Cost. CONSGD introduces no extra space cost compared with SSPSGD, but DYNSGD inevitably needs more memory. The space cost of $V(-)$ and $S(-)$, which store integer numbers, is marginal since M and C are normally less than 1000 in practice. The majority of the space cost is derived from the multi-version global updates. The memory overhead of global updates on one machine is measured by ρ, bounded by Theorem 3.

Theorem 3. Assuming that a parameter requires r bytes of memory, the parameter is averaged and partitioned over P parameter servers. Under the SSP constraint, the upper bound of ρ is $\frac{r}{P}(s$+1$)$.

Proof. The global parameter needs r bytes, so that each machine needs $\frac{r}{P}$ bytes. Each global update occupies the same space as the global parameter. Owing to the eviction mechanism in the push function, the number of global updates is $(c_{max}$–c_{min}+1$)$, thus:

$$\rho = \frac{r}{P}(c_{max} - c_{min} + 1) \tag{7}$$

Since c_{max}–$c_{min} \leq s$, $\rho \leq \frac{r}{P}(s$+1$)$. □

As we will show in the experimental section, with considerably large parameter (58 million) and staleness (40), DYNSGD only induces 3% additional memory usage. One major merit of the PS architecture is the high scalability. For larger parameter, we can increase the number of parameter servers to reduce the space cost.

Practical Optimization. We further implement an optimization to reduce the space cost. During the training process of SGD, different dimensions of a parameter converge at quite different rates owing to data skew [11]. Many dimensions might reach nearly complete convergence after a few clocks and then barely change afterwards. Making use of this phenomenon, we filter extraordinarily small figures (say less than 10^{-6}) in the updates before each worker sends them to the PS. On the PS, if the sparsity of one global update is less than 50%, we change the storage layout from dense format to sparse format. The storage format will be described below. As we will show in the experiment, this mechanism can significantly reduce the memory usage.

6. DATA MANAGEMENT

As mentioned in Section 2, our targeted data model includes immutable training samples/labels, and mutable model parameters. In this section, we describe the mechanism we employ to handle the mutable parameter and the immutable data in a distributed setting.

Data Storage. The training data and parameter are stored as vectors in our implementation. We use several vectors to represent the matrix data. If the data is in dense format, we store the value of each dimension. If the data is in sparse format, we store the ordered indexes and the corresponding values of non-zero entries [20].

Training Data Replication. In most distributed machine learning systems, each worker handles a subset of the whole training data. The *sharding* approach [37, 49] partitions the dataset and assigns one subset to each worker. An alternative to *sharding* is the *full replication* approach that replicates the whole dataset for each worker. As studies in [46], *full replication* takes more time per epoch than *sharding*, while uses fewer epochs to converge.

The full replication strategy is inefficient when the volume of training dataset exceeds the processing capability of one machine. Therefore, we choose the sharding strategy. It is optional to execute random data reassigning for each clock, however, as studied by [34], executing the data randomization is a time-consuming process that is not guaranteed to result in a significantly improvement of the convergence rate. Consequently, we choose to perform the data randomization once during the data-loading phase.

Parameter Versioning. As described in our implementation of DYNSGD, we store multi-version global updates and conduct version control on the PS. Existing parameter server systems [28, 16] cannot support this requirement. Therefore, we develop a new parameter server infrastructure with a multi-version mechanism borrowing the version control technique from the database community [25]. We provide three user-defined functions to apply an update to one version of the parameter.

1. A *map function* which maps an update to one version.

2. An *update function* which performs actual update on one version of the parameter.

3. An *expire function* which deletes one version when satisfying some condition. For example, in the implementation of DYNSGD, when the PS has received all the updates stamped with version v, we delete the v-th version of the global update.

Parameter Partition. Adopting the PS architecture, it is a prerequisite to design how to partition a parameter to several parameter servers. The problem of data partitioning across a set of nodes has been extensively studied by the database community [6, 15, 36]. The most widely used approaches are round-robin partition (successively partition), range partition (partition based on the values of the key), and hash partition (partition with a hash function). Hash partition achieves more balanced query distribution, while range partition facilitates range queries [15].

In the context of PS architecture, the parameter queries happen in diverse patterns according to specific implementations of machine learning algorithms. Some tasks need to get the whole parameter, while others need a portion of the parameter [18]. To achieve a trade-off between query balance and fast range query, we adopt a hybrid strategy, called range-hash partition [19]. We first partition the parameter to several ranges based on the indexes, and then use hash partition to assign the partitions to distributed nodes [28].

Partition Synchronization. Since the parameter partitions are pushed and pulled simultaneously by several workers, a key question is how to synchronize different partitions. Taking Figure 5 as an example where the number refers to the order of operations, the first worker pushes three partitions to the PS while the third worker pulls the parameter. In this case, since the third partition arrives late, the third worker loses this partition pushed by the first worker. With unsynchronized partitions, it may cause potential convergence degradation for the same reason explained in Section 3.3.

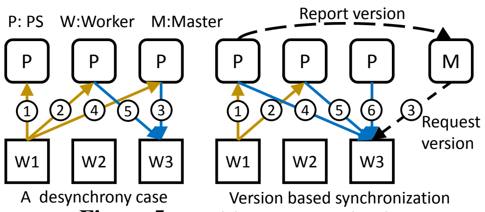


Figure 5: Partition synchronization

BSP protocol uses strict barrier to prevent this kind of desynchrony. While existing ASP and SSP implementations choose a best-effort scheme without partition synchronization mechanism.

Making use of our multi-version management of the parameter, we design a version-based method to address this problem. As shown in Figure 5, a master node supervises the parameter servers and the workers. Each parameter server reports the current version of its partition to the master periodically. Before pulling the global parameter, the worker requests the "stable version", i.e., the lowest version of all the partitions, from the master. Then, the worker pulls the partitions of the stable version from the parameter servers. As our empirical results in Appendix E show, this synchronization strategy brings a 9% improvement.

In order to implement this mechanism, we slightly modify Algorithm 2. We do not revise the global parameter once receiving an update. Instead, we add the v-th version global update to the global parameter if this version expires. In the push operation, we return the sum of the global parameter and the active global updates.

7. PERFORMANCE EVALUATION

7.1 Experiment Setup

Prototype System Implementation. Here we construct a prototype system to assess the proposed algorithms. We describe the detailed design of this prototype in Appendix D. At a high level, this prototype, programmed in Java and deployed on YARN [40], follows the classical data-parallel parameter-server architecture. We use the Netty framework to conduct the message passing. We store the training data in HDFS and design a data-splitter module to partition data. In addition, we provide an easy-to-use data-reading API with *memory*, *disk*, and *memory-and-disk* storage levels. Note that, we use the prototype as a vehicle to study the performance tradeoff of different approaches. Since our methods can be integrated into existing systems, we will continue this task in future work. We believe this future work is orthogonal to the contribution of this paper, which does not rely on a specific system.

Datasets and ML Models. As described in Table 2, the first chosen dataset is a malicious URL dataset [31]. The second one is a large-scale proprietary CTR (click-through rate) dataset from our industrial partner Tencent Inc., one of the largest Internet companies in the world. CTR is an ad-click prediction dataset which consists of ad attributes, advertiser attributes, user attributes, etc. For ML models, we choose ℓ_2-regularized Logistic Regression (LR) and Support Vector Machine (SVM) [8, 29, 9].

Dataset	size	# instance	# features	# non-zero entries
URL	4GB	2.4M	3.2M	500
CTR	200GB	300M	58M	100

Table 2: Datasets for evaluation

Metrics. To measure the end-to-end performance, we adopt the wall-clock time a system needs to converge to a given tolerance and we do not count the time used for data loading and result outputting. We further decompose the end-to-end performance as the *statistical efficiency* and the *hardware efficiency* defined in Section 3.

On a single worker, we preserve the objective value of every clock as $F=\{f_1, f_2, ..., f_C\}$. We use the *mean value* of the last five items as the minimal objective value, denoted by min_{obj}. Since we typically terminate SGD if the objective value barely changes [29], we use the *variance* of the last five items in F, denoted by var_{obj}, to judge whether SGD steadily converges.

Experiment Setting. We compare our prototype system with Spark, Petuum, and TensorFlow. The BSP-based Spark [45] system implements PSGD [49]. We use Bösen, the data-parallel subsystem of Petuum, which supports all the three synchronous protocols. TensorFlow [4] is a recently popular ML system supporting BSP and ASP protocols. Since TensorFlow has no automatic infrastructure for parameter partition, the users have to manually partition the parameter and hand-code the placement of each partition. These systems are compiled with gcc 4.8.4, Java 1.7, Python 2.7, and Scala 2.10. We use two clusters. **Cluster-1** is a 40-node cluster in which each machine is equipped with 64GB RAM, 24 cores, and 1GB Ethernet. We use this cluster with manually injected stragglers to assess different methods under the same heterogeneity condition. **Cluster-2** contains 400 machines with the same hardware as Cluster-1. However, Cluster-2 is shared by many users, and serves 100+ jobs simultaneously. This naturally heterogenous cluster is used to evaluate the effect of naturally-occurring straggler [21].

Protocol. When tuning the learning rate, we grid-search the optimal value. We tried both a fixed learning rate η=σ and a decayed learning rate η_c=$\frac{\sigma}{\sqrt{\alpha c+1}}$ where α=0.2. Following [23], we set the batch size as 10% of the data. To further investigate the convergence robustness of the proposed approaches in relation to the relevant factors of SGD—including the learning rate (η), the staleness (s), and the number of workers (M)—we vary one factor while fixing others to assess whether statistical efficiency is influenced. The thresholds of convergence are set to be 0.2 for the URL dataset and 0.02 for the CTR dataset. We choose these numbers since the predictive accuracy achieves 90% of the optimality. In order to quantify the impact of heterogeneity, we use the same mechanism used in Section 3 to obtain different heterogeneous levels.

7.2 End-to-End Comparison

We first use Cluster-1 to validate that existing systems running distributed SGD under the BSP, ASP, and SSP protocols all suffer from performance degradation in a heterogeneous cluster.

Overall Result. As shown in Table 3, DynSGD always converges in less time than the competitors, followed by ConSGD. In a homogeneous environment, the speedup is not as clear. In a heterogeneous setting, DynSGD can be at most 12× faster than Spark, Petuum, and TensorFlow. Moreover, DynSGD obtains the best hardware efficiency because it can converge in fewer clocks; thus stragglers have fewer negative effects on faster workers.

BSP System. We compare Spark, Petuum, and TensorFlow under the BSP protocol. It is obvious that Petuum and TensorFlow outperform Spark substantially—at most 5.47× faster on the large-scale dataset CTR. This performance improvement primarily derives from the communication acceleration brought by the param-

Metrics	Setting			Spark	Petuum	TF	Petuum	TF	Petuum	ConSGD	DynSGD	Petuum	ConSGD	DynSGD
				BSP			ASP		SSP, s=3			SSP, s=10		
run time	LR	URL	HL=1	280	134	172	117	141	87	85	**82**	78	78	**73**
			HL=2	555	267	347	625	725	250	196	**147**	549	116	**93**
		CTR	HL=1	2352	462	639	748	897	570	456	**342**	646	532	**456**
			HL=2	5198	966	1142	4964	5254	2233	1414	**546**	4501	798	**714**
	SVM	URL	HL=1	188	84	103	88	122	60	55	**51**	62	52	**44**
			HL=2	381	178	226	476	554	188	133	**102**	454	84	**60**
		CTR	HL=1	2829	544	660	480	711	528	539	**360**	544	479	**416**
			HL=2	6040	1104	1436	6120	6944	3690	1907	**873**	6534	1036	**555**
# updates	LR	URL	HL=1	**810**	810	810	955	937	847	862	**833**	949	948	**887**
			HL=2	**810**	810	810	3858	3901	1243	1062	**851**	3756	1144	**891**
		CTR	HL=1	**180**	180	180	317	303	278	229	**164**	492	406	**353**
			HL=2	**180**	180	180	1964	1936	606	404	**204**	2035	517	**453**
	SVM	URL	HL=1	**570**	570	570	826	848	752	687	**628**	969	824	**692**
			HL=2	**570**	570	570	3318	3276	1128	822	**639**	3518	986	**758**
		CTR	HL=1	**240**	240	240	366	393	322	318	**229**	467	447	**387**
			HL=2	**240**	240	240	2757	2781	890	511	**253**	2912	773	**434**
per-update time	LR	URL	HL=1	0.345	0.165	0.212	0.122	0.15	0.103	**0.098**	**0.098**	**0.082**	0.083	**0.082**
			HL=2	0.691	0.33	0.428	0.162	0.185	0.201	0.184	**0.172**	0.146	0.102	**0.1**
		CTR	HL=1	13.0	2.57	3.55	2.36	2.96	2.05	1.99	**1.98**	1.31	1.3	**1.29**
			HL=2	28.88	5.37	6.34	2.53	2.71	3.68	3.5	**3.33**	2.21	1.58	**1.54**
	SVM	URL	HL=1	0.329	0.148	0.18	0.107	0.144	**0.08**	0.081	**0.08**	0.064	0.063	**0.061**
			HL=2	0.659	0.313	0.396	0.143	0.169	0.166	0.162	**0.160**	0.129	0.085	**0.079**
		CTR	HL=1	11.79	2.27	2.75	1.31	1.81	1.64	1.69	**1.57**	1.16	1.07	**1.05**
			HL=2	25.17	4.6	5.98	2.22	2.50	4.15	3.72	**3.45**	2.24	1.34	**1.29**

Table 3: End-to-End Comparison. TF is TensorFlow in abbreviation. Statistical efficiency is measured by the number of updates received by the PS until convergence to the threshold, and hardware efficiency is measured by per-update time. Run time and per-update time are in seconds. We take three runs and report the average (standard deviation for all numbers $<$ 10% of the mean).

eter server. On the medium-sized dataset URL, Petuum can be at most 2.23× faster—a relatively lower speedup compared with the result of CTR. It demonstrates that the PS architecture reveals its benefits more explicitly with a larger dataset and a larger parameter. Note that Petuum outperforms TensorFlow because Petuum has a more efficient parameter server implementation, while TensorFlow lacks a mature distributed parameter server infrastructure.

ASP System. We choose Petuum and TensorFlow as representatives to investigate ASP systems. When HL=1, although needing more updates to converge (with lower statistical efficiency in other words), ASP systems can obtain comparable performance, even better in some cases, than the BSP system through delicate tuning because the ASP system obtains a higher hardware efficiency (lower per-update time). However, when HL=2, Petuum under the ASP protocol can be at most 5.54× slower than its BSP counterpart. This disaster indicates that ASP cannot work well on a heterogeneous cluster since it is extremely hard to converge.

SSP System. Here we evaluate Petuum and our prototype under the SSP protocol with both small and large staleness.

Small staleness: With small staleness (s=3) and HL=1, Petuum requires less run time than its BSP version in most cases— at most, it is 1.54× faster, albeit it usually needs more updates. The performance improvement comes from the improvement in hardware efficiency since SspSGD does not need to get the global parameter every clock. Unfortunately, Pettum is 7× slower in a heterogeneous setting because the loss of statistical efficiency exceeds the benefit of hardware efficiency. Even worse, with a small staleness, faster workers must wait for the stragglers frequently, which decreases the hardware efficiency as well.

ConSGD and DynSGD, however, are able to work on heterogeneous clusters. They can converge in fewer clocks, therefore, the frequency with which faster workers have to wait for the stragglers decreases. Unsurprisingly, their per-update time is 3.5 and 3.33 seconds, less than 3.85 seconds of Petuum. On LR and CTR, they are 1.47× and 4.09× faster than Petuum in the run time.

Large staleness: Given a large staleness (s=10), the performance of Petuum further degrades in a heterogeneous setting. For example, on SVM and CTR, Petuum needs 6.24× more updates than in a homogeneous cluster. Hence the run time increases from 544 seconds to 6,534 seconds by 12×, even worse than Spark.

In contrast, ConSGD and DynSGD behave much better in a heterogeneous cluster. The reasons are twofold. On the one hand, their statistical efficiencies are evidently better than SspSGD such that they only need 26.5% and 14.9% updates on SVM and CTR. Also, since they can converge in fewer clocks, the hindrance of stragglers is weakened, and thus their hardware efficiencies also exceed that of Petuum. Overall, they can be 6.3× and 11.77× faster than Petuum in the presence of stragglers.

Optimal staleness: Some readers might wonder how SspSGD behaves when the staleness and the learning rate are both delicately tuned. Note that our prototype beats Petuum even with optimal staleness. Taking the results on LR and URL dataset as an example, we find that even if we try really hard to tune and get the optimal s=3, DynSGD is still 1.7× faster than Petuum.

Summary. The performance of state-of-the-art systems degrades in heterogeneous settings since their employed distributed SGD algorithms all face a severe degradation of the statistical efficiency.

Generally, the SSP protocol is a promising candidate, especially for large models, since it can squeeze parameter communications. However, SspSGD is unable to converge steadily with heterogeneity. To solve this problem, ConSGD and DynSGD can overcome the damage of stragglers and outperform existing distributed SGD algorithms under the BSP, ASP, and SSP protocols.

7.3 Results over Production Cluster

In the above experiment, we artificially create some stragglers to engineer a heterogeneous environment. To verify that the above phenomena occur in a real productive cluster, we conduct more experiments on Cluster-2 — a heterogeneous production cluster of Tencent Inc. In order to verify the heterogeneity of this cluster, we run Petuum under the ASP protocol and present the per-clock time of each worker in Figure 6. As can be observed, this cluster encounters both computation and network heterogeneities. The fastest worker can be 2× faster than the slowest one.

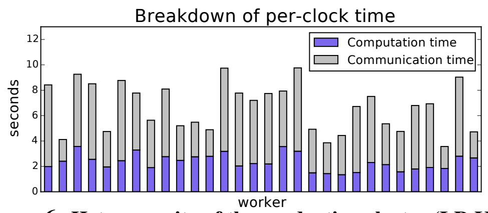


Figure 6: Heterogeneity of the production cluster (LR,URL)

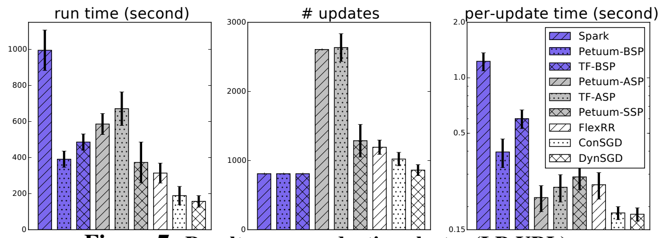


Figure 7: Results over production cluster (LR,URL)

We compare our prototype with Spark, BSP-based Petuum, BSP-based TensorFlow (TF), ASP-based Petuum, ASP-based TensorFlow, and SSP-based Petuum. In addition, we also consider an SSPSGD variant, called FlexRR [21]. FlexRR reassigns the training data among the workers to help stragglers catch up with others[3]. The staleness threshold is set to be three for all the SSP variants.

Figure 7 shows the results of LR and the URL dataset. Spark needs 995 seconds to converge, worse than 555 seconds in Table 3, due to more congested network in the productive cluster. The BSP-based Petuum and TensorFlow are 2× faster than Spark. ASP-based Petuum and TensorFlow, despite higher hardware efficiencies, need more time to converge than their BSP-based versions.

The SSP-based Petuum is 1.57× faster than its ASP version, and FlexRR further brings a 20% improvement which is similar to their reported results [21]. Although FlexRR can address the computation heterogeneity of the workers, it cannot solve the heterogeneity of the network. In other words, if the straggler is caused by slow communication rather than slow computation speed, moving some of its training data to other workers cannot solve the bottleneck. In fact, in this case, the extra data movement might cause more congested network that renders the straggler slower. Our methods, however, can cope with different kinds of heterogeneities in real clusters. Therefore, CONSGD and DYNSGD significantly outperform FlexRR — 1.66× and 2.1× faster respectively.

These results indicate that the performance degradation observed in Section 7.2 also happens in real clusters, sometimes to a greater extent. We also assess the proposed strategy of partition synchronization in this production cluster. Due to the space constraint, we present the results in Appendix E.

7.4 Robustness and Sensitivity

We validate that DYNSGD is robust to different choices of hyperparameters. As in end-to-end experiments, we find that the key to improving overall performance under the SSP protocol is to enhance statistical efficiency. Hence, we focus on the convergence behaviors of SSPSGD and our proposed SGD variants on our prototype system in Cluster-1 (HL=2). Due to the space constraint, we only present the results of LR on the CTR dataset in the following part. The other results are similar.

[3]Since there is no open-source code for FlexRR, we choose an approach superior to FlexRR. Each worker uses the memory-and-disk storage module we implement to load the entire dataset. The faster workers can read the subset of the stragglers, and the stragglers can handle a portion of their subsets. This variant surpasses FlexRR by preventing the cost of data movement among workers. To detect the stragglers, we use a parameter on the PS to record the time usages of the workers. Following the setting of FlexRR, we consider one worker as a straggler if it is 20% slower than the fastest, and we assign 5% data of the straggler to the fastest worker at each clock.

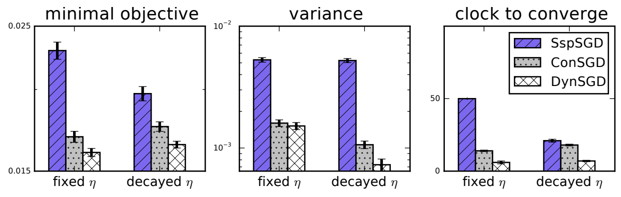


Figure 8: Best convergence criteria (LR,CTR,s=3,M=30,10% batch)

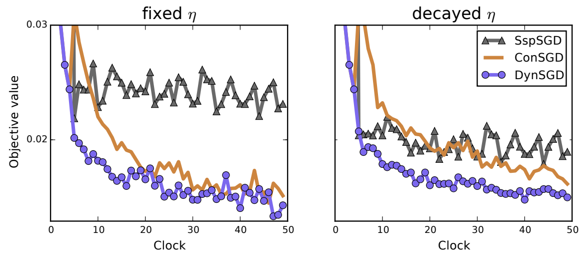


Figure 9: Best convergence curve (LR,CTR,s=3,M=30,10% batch)

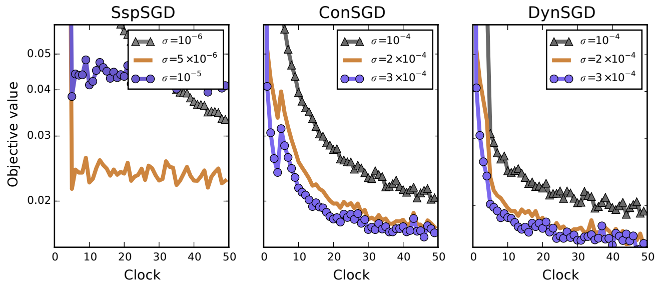


Figure 10: Impact of learning rate (LR,CTR,s=3,M=30,10% batch)

7.4.1 *Robustness and Convergence Rate*

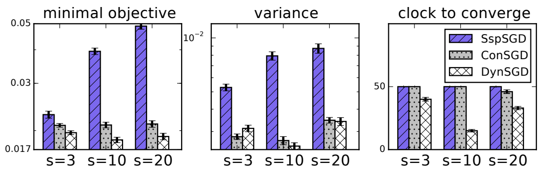
Choosing s=3, M=30 and a 10% mini-batch size, we grid-search for both the fixed and the decayed learning rate. We terminate the training process at clock 50. The best convergence curve and criteria of the fastest worker are presented in Figure 8 and Figure 9.

Fixed Learning Rate. Adopting fixed η, we obtain the optimal learning rates σ=5 × 10^{-6} for SSPSGD and σ=3 × 10^{-4} for CONSGD and DYNSGD, which verifies the statement in Section 3.3 that SSPSGD prefers very small local learning rate.

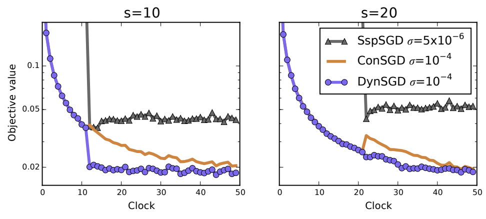
$\min_{obj}$ and var_{obj}: The minimal objective value (min_{obj}) of SSPSGD is 0.02294, while CONSGD and DYNSGD yield 27.51% and 30.17% improvements, respectively. Furthermore, the variance of objective value (var_{obj}) of SSPSGD is about 3.5× larger than that of CONSGD and DYNSGD, meaning that the objective value is endured with drastic oscillation.

Convergence rate: We can observe in Figure 9 that DYNSGD achieves the fastest convergence rate (clock vs. objective value). It only needs six clocks to converge to the threshold, while CONSGD needs 14 clocks. Unfortunately, SSPSGD fails to converge to the threshold within 50 clocks. The objective value decreases slowly before the first synchronization clock and sharply decreases at clock 5; however, it oscillates afterwards and cannot converge to the optimality. CONSGD converges quite steadily and is able to reach a much lower objective value. Although the objective value suffers a rebound at clock 5 as we have explained in Section 4, overall, CONSGD achieves fairly good performance taking into consideration its robustness and a fast convergence rate. DYNSGD achieves the best performance as shown in Figure 8 and Figure 9.

Decayed Learning Rate. Here we employ decayed learning rate $\eta_c=\frac{\sigma}{\sqrt{\alpha c+1}}$ where α=0.2. Similar to the above analysis,



(a) Convergence criteria



(b) Convergence curves

Figure 11: Impact of staleness (LR,CTR,M=30,10%batch)

though reaching a lower objective value than the fixed learning rate, min_{obj} of SspSGD is still 12.31% and 19.67% higher than the other two. The oscillation of SspSGD is not alleviated, and its var_{obj} is 4.9× and 7.17× larger than the other two algorithms, respectively. Meanwhile, SspSGD needs 21 clocks to converge to the threshold, while DynSGD only needs 7 clocks. In summary, DynSGD surpasses the other two methods as it achieves both robustness and a rapid convergence rate.

7.4.2 Impact of Learning Rate

Here we explore the impact of the learning rate η=σ by varying it in a moderate range. As shown in Figure 10, for SspSGD, a moderate change of σ leads to severe performance degradation. Unfortunately, although the mini-batch SGD is supposed to be robust against a reasonable change of learning rate, SspSGD violates this advantageous property. For ConSGD and DynSGD, when the learning rate varies slightly, they converge steadily, meaning they are robust to the modification of the learning rate.

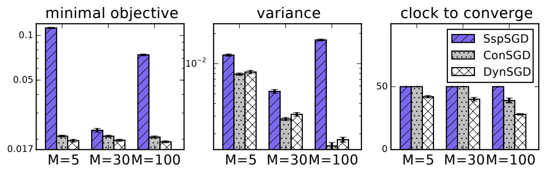
7.4.3 Impact of Staleness

We now study how staleness affects performance. We choose an appropriate learning rate of η=5 × 10^{-6} for SspSGD and η=10^{-4} for the other two algorithms since they achieve comparable min_{obj} according to the results in Figure 11. We use this as a baseline and vary the value of staleness while fixing the other relevant factors.

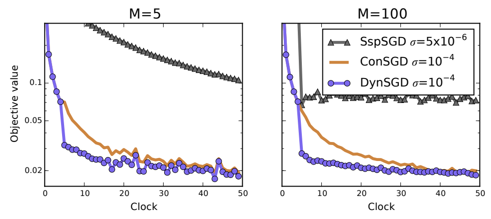
For SspSGD, its convergence is acceptable when s=3, reaching a minimal objective value of 0.02294. However, the growth of staleness significantly increases min_{obj} and var_{obj}. In addition, before the first synchronization clock, the convergence rate is very slow since one worker cannot benefit from other workers' updates. As for ConSGD and DynSGD, they sustain modest effects when we increase the staleness. Although the objective value of ConSGD slightly increases at the first synchronization clock, the overall convergence is stable. DynSGD still converges faster than ConSGD and demands fewer clocks to converge to the threshold.

7.4.4 Impact of Cluster Scale

Here we study the impact of varying the number of workers. We choose the same learning rate as Section 7.4.3. The results are presented in Figure 12. For SspSGD, if M increases from 30 to 100, var_{obj} increases from 0.00529 to 0.01722 since more updates within a clock amplify the harm of stragglers. Therefore SspSGD cannot converge to the optimality and min_{obj} increases from 0.02294 to 0.07398. In contrast, the decline of M from 30 to five renders the convergence rate very slow as a result of fewer updates during a clock, as well as increases min_{obj} to as large



(a) Convergence criteria



(b) Convergence curves

Figure 12: Impact of cluster scale (LR,CTR,s=3,10%batch)

as 0.11234 after 50 clocks. As for ConSGD and DynSGD, the convergence rate is not obviously influenced with a distinct M and no significant growth of min_{obj} occurs.

7.4.5 Choice of Global Learning Rate for ConSGD

For ConSGD, we use a simple heuristic to choose the constant global learning rate — λ_g=$\frac{1}{M}$. Although this choice works fine in practice, we wonder what the optimal choice of the global learning rate might be. Therefore, we use a grid search to find the optimal number ($\{1, 0.9, ..., 0.1, 0.01\}$) using the optimal learning rate σ=3 × 10^{-4} obtained in Section 7.4.1. As shown in Table 4, ConSGD obtains the best performance when λ_g=0.1. It converges to the threshold in 11 clocks, 1.27× faster than λ_g=$\frac{1}{M}$. But ConSGD is still slower than DynSGD. More importantly, a grid search for the optimal local learning rate η and the optimal global learning rate λ_g is arguably time-consuming.

	λ_g	min_{obj}	var_{obj}	clock to converge
ConSGD	0.1	**0.01657** ±0.00045	0.0022 ±0.00027	**11**±2
	$\frac{1}{M}$	0.01693 ±0.00033	0.0016 ±0.0001	14±1

Table 4: Optimal global learning rate (LR,CTR,M=30,s=3,10%batch)

7.5 Overhead of DynSGD

As explained in Section 5, DynSGD inevitably incurs extra space cost. Here we proceed to monitor average memory and average CPU usages of workers and parameter servers, presented in Figure 13. Since the hardware overheads of workers are hardly affected by different factors, we present one representative result.

The memory and CPU usages of parameter servers are much lower than those of workers. This suggests that the workers are more resource-constrained than parameter servers. DynSGD induces moderately higher memory usage than SspSGD. When the staleness is small (s=3), the space cost is imperceptible. When the staleness is considerably large (s=40), the average memory usage increases 3.02%, which is a relatively minor increase.

7.6 Summary of Performance Comparison

We demonstrate that existing ML systems running distributed SGD under the BSP, ASP, and SSP protocols cannot perform well in heterogeneous environments. Their performance can degrade substantially—at most 12× slower. It is problematic for ASP and SspSGD to steadily converge to the optimality. Furthermore, SspSGD is sensitive to the modification of relevant factors. If these factors change, practitioners have to search for another proper

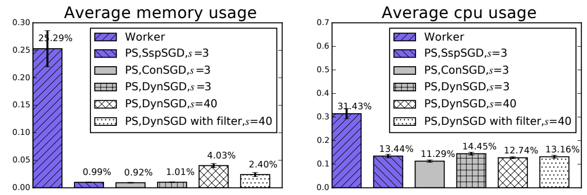


Figure 13: Hardware overhead (LR,CTR,M=30,10% batch)

learning rate. In practice, frequent seeking for the appropriate learning rate incurs tremendous time and resource wastes.

To the contrary, our proposed heterogeneity-aware distributed SGD algorithms, which leverage learning rate schedulers, work well in the presence of stragglers. We validate their effectiveness and efficiency in our prototype, noting that they can converge quite robustly and the modification of related factors barely harms the convergence. Overall, DYNSGD achieves the best performance as it accomplishes stable convergence, a fast convergence rate, and robustness regarding hyperparameters and configurations.

8. RELATED WORK

SGD is the workhorse algorithm for modern large-scale ML problems because of its high convergence rate, low memory cost, and robustness against noise with the batch scheme. There have been many works that deal with the parallelization of SGD. Langford [48] presented a Round-Robin method that puts the threads in order and lets each thread update the parameter successively. Recht [37] introduced a lock-free approach to parallelizing SGD via allowing the threads to access the shared parameters. Dimmwitted [46] studied the tradeoff space of access and replication methods to parallelize SGD in the NUMA architecture.

Unfortunately, the aforementioned algorithms are limited to a single machine. Zinkevich [49] proposed a distributed SGD algorithm under the BSP protocol that runs independent instances on several machines and calculates the average of distributed parameters replicas. But the existence of barrier under the BSP constraint results in an obstructed network and unnecessary waiting for slow workers. Ho [23] overcame the defects of the BSP protocol by introducing the SSP protocol. It allows the employment of a stale parameter; thus workers do not need to request the global parameter at every clock. Recently, a data reassigning method is proposed to speed up stragglers [21]. However, this approach induces non-negligible network overhead, and fails to solve the network heterogeneity. When suffering the network heterogeneity, the straggler still runs slowly even if we move part of its data to others. Worse, the data movement might aggravate the communication bottleneck of the stragglers. An orthogonal technique [35] can reduce the convergence imbalance via searching for the optimal step size in a candidate set on different workers. But its BSP-based distributed implementation can be stalled by the stragglers. Moreover, this method is inefficient for large parameters since it stores and calculates various versions of the parameter on each worker.

Another related research field is the distributed ML systems that leverage the above-mentioned synchronization protocols and parallel SGD algorithms. The database community has developed a lot of ML systems, such as SystemML [20], SimSQL [10], Vertica-R [33], FactorizedLearning [26], and MADlib [22], in which ML analytics is expressed in a higher-level language and is compiled and executed in a MapReduce environment. Other prevailing distributed ML systems include Mahout [2], MLlib [3], Pregel [32], GraphLab [30], and GraphX [43]. These systems establish a file system, a single coordinator, a message-passing mechanism, coarse-grained immutable RDDs [45], or built-in consistency mechanisms to synchronize parameters. These schemes either slow down the processing speed or reveal inefficiency when applying the fine-grained operations of iterative ML algorithms. Therefore, a more efficient parameter communication architecture, i.e., the PS architecture [7, 17, 39], was introduced to partition the parameter over machines. A range of platforms, such as ParallelLDA [39], YahooLDA [39], DistBelief [17], Petuum [16], and TensorFlow [4], incorporate the PS architecture and ASP-based distributed SGD. TensorFlow [4] can be executed on heterogeneous hardware, but its targeted heterogeneity refers to the ability of running on various devices, including mobile phones, tablets, commodity machines, and GPUs. This kind of heterogeneity is different from our targeted heterogeneity which leads to stragglers and unstable convergence. Since TensorFlow currently only supports BSP and ASP protocols, employing asynchronous updates leads to sub-optimal training performance, while employing synchronous updates will be as slow as the slowest replica[4]. What is more, TensorFlow currently lacks automatic parameter partition and efficient distributed deployment. Recently, the second-generation PS systems, such as Petuum [16] and Bösen [41], were developed to support the SSP protocol. When running SGD, they generally add the local updates of workers to the global parameter. This accumulative strategy weakens the robustness of SGD.

The implementation of the PS architecture involves the method that partitions immutable data and mutable parameter across a set of machines. Sharding and full replication are both employed by existing works to partition immutable data. As studied in [46], sharding increases the variance of the estimate we form on each node, while full replication increases the space cost. The database community has proposed various strategies to partition mutable data over distributed nodes, such as round-robin partition, range partition, and hash partition. Range partition accelerates range query as it minimizes the number of multi-sited transactions, while hash partition obtains better load balance for random queries [15]. Some other works propose hybrid partition strategies [19] to combine different partition methods according to specific scenarios.

9. CONCLUSION

We presented a systematic study of ML platforms running distributed SGD in a heterogeneous environment. We empirically and intuitively analyzed how the performance of existing systems can deteriorate up to 10× when facing heterogeneity. To solve this problem, we proposed two heterogeneity-aware distributed SGD algorithms under the SSP protocol. The first one employed a constant learning rate schedule for local updates when updating the global parameter to alleviate the unstable convergence caused by stragglers. Through considering the delayed information of local updates, we proposed the second one, a dynamic learning rate schedule, to further improve convergence robustness. Besides, we also prove the valid convergence of our approaches in theory.

We designed an efficient data structure for the introduced method and implemented a prototype system. We found that our prototype can be up to 12× faster than state-of-the-art systems. Moreover, our heterogeneity-aware algorithms were able to achieve significantly stable convergence, robustness against relevant hyperparameters, and a rapid convergence rate (at most 6× faster).

Acknowledgements

This research is supported by the National Natural Science Foundation of China under Grant No. 61572039, 973 program under No. 2014CB340405, Shenzhen Gov Research Project JCYJ20151014093505032, and Tecent Research Grant (PKU).

[4]https://www.tensorflow.org/versions/r0.11/tutorials/deep_cnn/index.html

10. REFERENCES

[1] Amazon ec2. https://aws.amazon.com/ec2/.

[2] Mahout project. http://mahout.apache.org/.

[3] Spark mllib. http://spark.apache.org/mllib/.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[5] A. Agarwal and J. C. Duchi. Distributed delayed stochastic optimization. In *NIPS*, pages 873–881, 2011.

[6] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, pages 359–370, 2004.

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

[8] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*, pages 177–186. 2010.

[9] L. Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. 2012.

[10] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued markov chains using simsql. In *SIGMOD*, pages 637–648, 2013.

[11] D. W. Cheung, S. D. Lee, and Y. Xiao. Effect of data skewness and workload balance in parallel data mining. *TKDE*, 14(3):498–514, 2002.

[12] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, 2014.

[13] B. Cui, H. Mei, and B. C. Ooi. Big data: the driver for innovation in databases. *National Science Review*, 1(1):27–30, 2014.

[14] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*, pages 37–48, 2014.

[15] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *VLDB*, 3(1-2):48–57, 2010.

[16] W. Dai, J. Wei, J. K. Kim, S. Lee, J. Yin, Q. Ho, and E. P. Xing. Petuum: A framework for iterative-convergent distributed ml. *arXiv preprint arXiv:1312.7651*, 2013.

[17] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *NIPS*, pages 1223–1231, 2012.

[18] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *SIGKDD*, pages 69–77, 2011.

[19] S. Ghandeharizadeh and D. J. DeWitt. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *VLDB*, pages 481–492, 1990.

[20] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

[21] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Addressing the straggler problem for iterative convergent parallel ml. In *SoCC*, pages 98–111, 2016.

[22] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library: or mad skills, the sql. *VLDB*, 5(12):1700–1711, 2012.

[23] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

[24] Y. Huang, B. Cui, J. Jiang, K. Hong, W. Zhang, and Y. Xie. Real-time video recommendation exploration. In *SIGMOD*, pages 35–46, 2016.

[25] W. Kim and F. H. Lochovsky. *Object-oriented concepts, databases, and applications*. 1989.

[26] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984, 2015.

[27] A. Lastovetsky, I.-H. Mkwawa, and M. O'Flynn. An accurate communication model of a heterogeneous cluster based on a switch-enabled ethernet network. In *ICPADS*, page 6, 2006.

[28] 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 *OSDI*, pages 583–598, 2014.

[29] M. Li, T. Zhang, Y. Chen, and A. J. Smola. Efficient mini-batch training for stochastic optimization. In *SIGKDD*, pages 661–670, 2014.

[30] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.

[31] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Identifying suspicious urls: an application of large-scale online learning. In *ICML*, pages 681–688, 2009.

[32] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[33] S. Prasad, A. Fard, V. Gupta, J. Martinez, J. LeFevre, V. Xu, M. Hsu, and I. Roy. Large-scale predictive analytics in vertica: fast data transfer, distributed model creation, and in-database prediction. In *SIGMOD*, pages 1657–1668, 2015.

[34] C. Qin and F. Rusu. Scalable i/o-bound parallel incremental gradient descent for big data analytics in glade. In *DanaC*, pages 16–20, 2013.

[35] C. Qin and F. Rusu. Speculative approximations for terascale distributed gradient descent optimization. In *DanaC*, 2015.

[36] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *SIGMOD*, pages 558–569, 2002.

[37] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[38] X. Shi, B. Cui, Y. Shao, and Y. Tong. Tornado: A system for real-time iterative analysis over evolving data. In *SIGMOD*, pages 417–430, 2016.

[39] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *PVLDB*, 3(1-2):703–710, 2010.

[40] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *SOCC*, page 5, 2013.

[41] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SOCC*, pages 381–394, 2015.

[42] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding. Data mining with big data. *TKDE*, 26(1):97–107, 2014.

[43] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *GRADES*, page 2, 2013.

[44] N. Xu, B. Cui, L. Chen, Z. Huang, and Y. Shao. Heterogeneous environment aware streaming graph partitioning. *TKDE*, 27(6):1560–1572, 2015.

[45] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2, 2012.

[46] C. Zhang and C. Ré. Dimmwitted: A study of main-memory statistical analytics. *PVLDB*, 7(12):1283–1294, 2014.

[47] S.-Y. Zhao and W.-J. Li. Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee. In *AAAI*, 2016.

[48] M. Zinkevich, J. Langford, and A. J. Smola. Slow learners are fast. In *NIPS*, pages 2331–2339, 2009.

[49] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.

APPENDIX

A. PROOF OF CONSGD'S CONVERGENCE BOUND

We first provide the formal statement of CONSGD under the SSP protocol. Without loss of generality, we set the batch size $b = 1$ for analysis convenience. Since there is no barrier at the end of each clock, different workers hold different and noisy versions of the parameter w at a specific moment. We use $\widetilde{w}_c^m$ to denote the "noisy" parameter read by worker m at clock c. Following the analysis of [23] and [48], Lemma 1 defines the noisy parameter:

Lemma 1. (CONSGD) We set a fixed staleness $s > 0$ for the SSP protocol. Adopting CONSGD, the noisy parameter $\widetilde{w}_c^m$ can be decomposed into:

$$\widetilde{w}_c^m = w_0 + \frac{1}{M}\left(\sum_{c'=1}^{c-s-1}\sum_{m'=1}^{M} u_{c'}^{m'} + \sum_{c'=c-s}^{c-1} u_{c'}^{m} + \sum_{(c',m')\in H} u_{c'}^{m'}\right) \tag{8}$$

Here w_0 refers to the initial parameter and H is a subset of the local updates whose clocks are within $[c–s, c+s–1]$. The subset H excludes the updates of worker m.

In summary, $\widetilde{w}_c^m$ contains three groups of local updates:

1. Updates from clock 1 to $c–s–1$ which the PS has all received.

2. Updates on worker m from clock $c–s$ to $c–1$. This part refers to the local updates of worker m.

3. Updates in H. The clocks range from $c–s$ to $c+s–1$, and updates of a certain clock partly arrive the PS. Since the second part includes worker m's local updates, we should rule them out from H.

In order to analyze the convergence rate, we define a reference sequence of the parameter w called the "restrained" parameter:

$$w_t = w_0 + \frac{1}{M}\sum_{t'=0}^{t} u_{t'}, \;\; where \;\; u_t := u_{t/M}^{t\%M} \tag{9}$$

w_t is generated based on a Round-Robin mechanism. We put all the workers in order and sum all the updates successively. At every clock, we sum the updates from worker 1 to worker M. Specifically, at clock t/M, we sum the updates from worker 1 to worker $t\%M$. Lemma 2 defines the difference between the "restrained" parameter w_t and the "noisy" parameter $\widetilde{w}_c^m$.

Lemma 2. (CONSGD) Adopting CONSGD under the SSP constraint, the difference between w_t and $\widetilde{w}_t$ is:

$$\widetilde{w}_t = w_t - \frac{1}{M}\Big[\sum_{i\in D_t} u_i\Big] + \frac{1}{M}\Big[\sum_{i\in E_t} u_i\Big] \tag{10}$$

Here D_t refers to the subset of delayed updates which arrive the PS late from $(t–(s+1)M)$ to $(t–1)$, and E_t refers to the subset of early updates which arrive the PS in advance from $(t+1)$ to $(t+(s–1)M)$. We assert that $|D_t|+|E_t| \le s(M–1)$, $min(D_t \cup E_t) \ge max(1, t–(s+1)M)$, and $max(D_t \cup E_t) \le t+(s–1)M$.

Proof. Eq.(10) is obtained by comparing Eq.(8) and Eq.(9). Apparently, updates in D_t are within $[t–(s+1)M, t–1]$, and updates in E_t are within $[t+1, t+(s–1)M]$. Therefore, the minimal t in $D_t \cup E_t$ is not smaller than $max(1, t–(s+1)M)$, and the maximal t in $D_t \cup E_t$ is not larger than $t+(s–1)M$. We claim that $max(D_t \cup E_t)–min(D_t \cup E_t) \le sM$ since $c_{max}–c_{min} \le s$. Moreover, local updates of worker $t\%M$ should be evicted from D_t and E_t, thus $|D_t|+|E_t| \le sM–s=s(M–1)$. Compared with [23], we decrease this upper bound from $2s(M–1)$ to $s(M–1)$. □

Before proving Theorem 1, we give the following lemma to help bounding the instantaneous regret at a certain moment.

Lemma 3. (CONSGD) For all $t > 0$, if $w \in \mathbb{R}^n$, the following equation holds:

$$\langle \widetilde{w}_t – w^*, \widetilde{g}_t\rangle = \frac{\eta_t}{2M}\|\widetilde{g}_t\|^2 + \frac{M}{\eta_t}[D(w^*\|w_t) – D(w^*\|w_{t+1})] + \frac{1}{M}\Big[\sum_{i\in D_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle – \sum_{i\in E_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle\Big]$$

Proof. According to the definition of $D(w\|w')$ in Assumption 2,

$$D(w^*\|w_{t+1}) – D(w^*\|w_t) = \frac{1}{2}\|w^* – w_t + w_t – w_{t+1}\|^2 – \frac{1}{2}\|w^* – w_t\|^2$$
$$= \frac{1}{2}\|w^* – w_t + \frac{1}{M}\eta_t\widetilde{g}_t\|^2 – \frac{1}{2}\|w^* – w_t\|^2$$
$$= \frac{\eta_t^2}{2M^2}\|\widetilde{g}_t\|^2 – \frac{\eta_t}{M}\langle\widetilde{w}_t – w^*, \widetilde{g}_t\rangle – \frac{\eta_t}{M}\langle w_t – \widetilde{w}_t, \widetilde{g}_t\rangle$$

Applying Lemma 2 yields:

$$\langle w_t – \widetilde{w}_t, \widetilde{g}_t\rangle = \langle\frac{1}{M}\Big[\sum_{i\in D_t} u_i – \sum_{i\in E_t} u_i\Big], \widetilde{g}_t\rangle$$
$$= –\frac{1}{M}\sum_{i\in D_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle + \frac{1}{M}\sum_{i\in E_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle$$

Then, we obtain:

$$\frac{M}{\eta_t}[D(w^*\|w_{t+1}) – D(w^*\|w_t)]$$
$$= \frac{\eta_t}{2M}\|\widetilde{g}_t\|^2 – \langle\widetilde{w}_t – w^*, \widetilde{g}_t\rangle + \frac{1}{M}\Big[\sum_{i\in D_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle – \sum_{i\in E_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle\Big]$$

Move $\langle\widetilde{w}_t – w^*, \widetilde{g}_t\rangle$ to the left side:

$$\langle \widetilde{w}_t – w^*, \widetilde{g}_t\rangle = \frac{\eta_t}{2M}\|\widetilde{g}_t\|^2 + \frac{M}{\eta_t}[D(w^*\|w_t) – D(w^*\|w_{t+1})] + \frac{1}{M}\Big[\sum_{i\in D_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle – \sum_{i\in E_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle\Big]$$

□

We can prove Theorem 1 via employing Lemma 3.

Proof. Since $f_t(w)$ is convex, we get the upper bound of $R[W]$:

$$R[W] := \frac{1}{T}\sum_{t=1}^{T}[f_t(\widetilde{w}_t) – f_t(w^*)] \le \frac{1}{T}\sum_{t=1}^{T}\langle\widetilde{g}_t, \widetilde{w}_t – w^*\rangle$$
$$= \frac{1}{MT}\sum_{t=1}^{T}\frac{1}{2}\eta_t\|\widetilde{g}_t\|^2 + \frac{1}{MT}\sum_{t=1}^{T}\Big[\sum_{i\in D_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle – \sum_{i\in E_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle\Big]$$
$$+ \frac{M}{T}\Big[\frac{D(w^*\|w_1)}{\eta_1} – \frac{D(w^*\|w_{T+1})}{\eta_T} + \sum_{t=2}^{T}[D(w^*\|w_t)(\frac{1}{\eta_t} – \frac{1}{\eta_{t–1}})]\Big]$$

We can bound the first term by the L-Lipschitz property in Assumption 1:

$$\sum_{t=1}^{T}\frac{1}{2}\eta_t\|\widetilde{g}_t\|^2 \le \sum_{t=1}^{T}\frac{\sigma}{2\sqrt{t}}L^2 \le \sigma L^2\int_0^T\frac{1}{2\sqrt{t}} = \sigma L^2\sqrt{T}$$

The bounded diameter property in Assumption 2 yields:

$$\frac{D(w^*\|w_1)}{\eta_1} – \frac{D(w^*\|w_{T+1})}{\eta_T} + \sum_{t=2}^{T}[D(w^*\|w_t)(\frac{1}{\eta_t} – \frac{1}{\eta_{t–1}})]$$
$$\le \frac{F^2}{\sigma} + F^2\sum_{t=2}^{T}(\frac{\sqrt{t}}{\sigma} – \frac{\sqrt{t–1}}{\sigma}) = \frac{F^2}{\sigma}\sqrt{T}$$

Then we get the inequation below on the basis of Lemma 2:

$$\sum_{t=1}^{T}[\sum_{i\in D_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle-\sum_{i\in E_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle]$$
$$\leq\sum_{t=1}^{T}(|D_t|+|E_t|)\eta_{min(D_t\cup E_t)}L^2\quad(since\ \eta_{t-1}>\eta_t)$$
$$\leq\sum_{t=1}^{T}(|D_t|+|E_t|)\eta_{max(1,t-(s+1)M)}L^2$$
$$(since\ min(D_t\cup E_t)\geq max(1,t-(s+1)M))$$
$$=L^2[\sum_{t=1}^{(s+1)M}(|D_t|+|E_t|)\sigma+\sum_{t=(s+1)M+1}^{T}(|D_t|+|E_t|)\frac{\sigma}{\sqrt{t-(s+1)M}}]$$
$$\leq\sigma L^2[\sum_{t=1}^{(s+1)M}s(M-1)+\sum_{t=(s+1)M+1}^{T}s(M-1)\frac{\sigma}{\sqrt{t-(s+1)M}}]$$
$$\leq\sigma L^2 s(M-1)[(s+1)M+2\sqrt{T-(s+1)M}]$$
$$\leq\sigma L^2(s+1)^2M^2+2\sigma L^2(s+1)M\sqrt{T}$$

Applying the above inequations, we achieve:

$$R[W]\leq\frac{\sigma L^2}{M\sqrt{T}}+\frac{MF^2}{\sigma\sqrt{T}}+\frac{\sigma L^2}{MT}(s+1)^2M^2+\frac{2\sigma L^2}{M\sqrt{T}}(s+1)M$$

We define $\nu=(s+1)M$ and set $\sigma=\frac{F}{L\sqrt{2\nu}}$.

$$R[W]\leq FL\sqrt{\frac{2\nu}{T}}[\frac{1}{2\nu M}+M+\frac{\nu}{2M\sqrt{T}}+\frac{1}{M}]$$
$$\leq FL(M+3)\sqrt{\frac{2\nu}{T}}$$

If $\sigma=\frac{MF}{L\sqrt{2\nu}}$, we can further lower the upper bound:

$$R[W]\leq\frac{FL}{\sqrt{2\nu T}}+FL\sqrt{\frac{2\nu}{T}}+\frac{FL\nu\sqrt{\nu}}{\sqrt{2T}}+FL\sqrt{\frac{2\nu}{T}}$$
$$=FL\sqrt{\frac{2\nu}{T}}[2+\frac{1}{2\nu}+\frac{\nu}{2\sqrt{T}}]\leq 3FL\sqrt{\frac{2\nu}{T}}\ \text{(when T is large enough)}$$

□

B. PROOF OF DYNSGD'S CONVERGENCE BOUND

Similar to Lemma 1 and Lemma 2, we first provide the formal statement of the "noisy" parameter and the "restrained" parameter of DYNSGD.

Lemma 4. (DYNSGD) We set a fixed staleness $s>0$ for the SSP protocol. Adopting DYNSGD, the noisy parameter $\widetilde{w}_c^m$ can be decomposed into:

$$\widetilde{w}_c^m=w_0+\sum_{c'=1}^{c-s-1}\sum_{m'=1}^{M}\frac{1}{\mu_{c'}^{m'}}u_{c'}^{m'}+\sum_{c'=c-s}^{c-1}\frac{1}{\mu_{c'}^{m}}u_{c'}^{m}+\sum_{(c',m')\in H}\frac{1}{\mu_{c'}^{m'}}u_{c'}^{m'}\tag{11}$$

The "restrained" parameter is:

$$w_t=w_0+\sum_{t'=0}^{t}\frac{1}{\mu_{t'}}u_{t'},\ where\ u_t:=u_{t/M}^{t\%M},\mu_t:=\mu_{t/M}^{t\%M}\tag{12}$$

Lemma 5. (DYNSGD) Adopting DYNSGD under the SSP protocol, the difference between w_t and $\widetilde{w}_t$ is:

$$\widetilde{w}_t=w_t-\sum_{i\in D_t}\frac{1}{\mu_i}u_i+\sum_{i\in E_t}\frac{1}{\mu_i}u_i\tag{13}$$

Similar to Lemma 3, we give the following lemma to help bounding the instantaneous regret.

Lemma 6. (DYNSGD) For all $t>0$, if $w\in\mathbb{R}^n$, the following equation holds:

$$\langle\widetilde{w}_t-w^*,\widetilde{g}_t\rangle=\frac{\eta_t}{2\mu_t}\|\widetilde{g}_t\|^2+\frac{\mu_t}{\eta_t}[D(w^*\|w_t)-D(w^*\|w_{t+1})]$$
$$+\sum_{i\in D_t}\frac{\eta_i}{\mu_i}\langle\widetilde{g}_i,\widetilde{g}_t\rangle-\sum_{i\in E_t}\frac{\eta_i}{\mu_i}\langle\widetilde{g}_i,\widetilde{g}_t\rangle$$

Proof. According to the definition of $D(w\|w')$,

$$D(w^*\|w_{t+1})-D(w^*\|w_t)=\frac{1}{2}\|w^*-w_t+\frac{1}{\mu_t}\eta_t\widetilde{g}_t\|^2-\frac{1}{2}\|w^*-w_t\|^2$$
$$=\frac{\eta_t^2}{2\mu_t^2}\|\widetilde{g}_t\|^2-\frac{\eta_t}{\mu_t}\langle\widetilde{w}_t-w^*,\widetilde{g}_t\rangle-\frac{\eta_t}{\mu_t}\langle w_t-\widetilde{w}_t,\widetilde{g}_t\rangle$$

Applying Lemma 5 yields:

$$\langle w_t-\widetilde{w}_t,\widetilde{g}_t\rangle=-\sum_{i\in D_t}\frac{\eta_i}{\mu_i}\langle\widetilde{g}_i,\widetilde{g}_t\rangle+\sum_{i\in E_t}\frac{\eta_i}{\mu_i}\langle\widetilde{g}_i,\widetilde{g}_t\rangle$$

Then, we can obtain:

$$\frac{\mu_t}{\eta_t}[D(w^*\|w_{t+1})-D(w^*\|w_t)]$$
$$=\frac{\eta_t}{2\mu_t}\|\widetilde{g}_t\|^2-\langle\widetilde{w}_t-w^*,\widetilde{g}_t\rangle+\sum_{i\in D_t}\frac{\eta_i}{\mu_i}\langle\widetilde{g}_i,\widetilde{g}_t\rangle-\sum_{i\in E_t}\frac{\eta_i}{\mu_i}\langle\widetilde{g}_i,\widetilde{g}_t\rangle$$

Move $\langle\widetilde{w}_t-w^*,\widetilde{g}_t\rangle$ to the left side:

$$\langle\widetilde{w}_t-w^*,\widetilde{g}_t\rangle=\frac{\eta_t}{2\mu_t}\|\widetilde{g}_t\|^2+\frac{\mu_t}{\eta_t}[D(w^*\|w_t)-D(w^*\|w_{t+1})]$$
$$+\sum_{i\in D_t}\frac{\eta_i}{\mu_i}\langle\widetilde{g}_i,\widetilde{g}_t\rangle-\sum_{i\in E_t}\frac{\eta_i}{\mu_i}\langle\widetilde{g}_i,\widetilde{g}_t\rangle$$

□

We can prove Theorem 2 via employing Lemma 6.

Proof. We get the upper bound of $R[W]$ referring to the proof of Theorem 1:

$$R[W]\leq\frac{1}{T}\sum_{t=1}^{T}\langle\widetilde{g}_t,\widetilde{w}_t-w^*\rangle$$
$$=\frac{1}{T}\sum_{t=1}^{T}\frac{\eta_t}{2\mu_t}\|\widetilde{g}_t\|^2+\frac{1}{T}\sum_{t=1}^{T}\frac{\mu_t}{\eta_t}[D(w^*\|w_t)-D(w^*\|w_{t+1})]$$
$$+\frac{1}{T}\sum_{t=1}^{T}[\sum_{i\in D_t}\frac{\eta_i}{\mu_i}\langle\widetilde{g}_i,\widetilde{g}_t\rangle-\sum_{i\in E_t}\frac{\eta_i}{\mu_i}\langle\widetilde{g}_i,\widetilde{g}_t\rangle]$$

The upper bound of the expectation of $R[W]$ with $\mathbb{E}[\mu_t]=\mu$:

$$\mathbb{E}[R[W]]\leq\frac{1}{T}\sum_{t=1}^{T}\frac{\eta_t}{2\mathbb{E}[\mu_t]}\|\widetilde{g}_t\|^2+\frac{1}{T}\sum_{t=1}^{T}\frac{\mathbb{E}[\mu_t]}{\eta_t}[D(w^*\|w_t)$$
$$-D(w^*\|w_{t+1})]+\frac{1}{T}\sum_{t=1}^{T}[\sum_{i\in D_t}\frac{\eta_i}{\mathbb{E}[\mu_i]}\langle\widetilde{g}_i,\widetilde{g}_t\rangle-\sum_{i\in E_t}\frac{\eta_i}{\mathbb{E}[\mu_i]}\langle\widetilde{g}_i,\widetilde{g}_t\rangle]$$
$$=\frac{1}{\mu T}\sum_{t=1}^{T}\frac{1}{2}\eta_t\|\widetilde{g}_t\|^2+\frac{\mu}{T}\sum_{t=1}^{T}\frac{1}{\eta_t}[D(w^*\|w_t)-D(w^*\|w_{t+1})]$$
$$+\frac{1}{\mu T}\sum_{t=1}^{T}[\sum_{i\in D_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle-\sum_{i\in E_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle]$$
$$=\frac{1}{\mu T}\sum_{t=1}^{T}\frac{1}{2}\eta_t\|\widetilde{g}_t\|^2+\frac{1}{\mu T}\sum_{t=1}^{T}[\sum_{i\in D_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle-\sum_{i\in E_t}\eta_i\langle\widetilde{g}_i,\widetilde{g}_t\rangle]$$
$$+\frac{\mu}{T}[\frac{D(w^*\|w_1)}{\eta_1}-\frac{D(w^*\|w_{T+1})}{\eta_T}+\sum_{t=2}^{T}[D(w^*\|w_t)(\frac{1}{\eta_t}-\frac{1}{\eta_{t-1}})]]$$

We have bounded these three terms in Appendix A, thus:

$$\mathbb{E}[R[W]]\leq\frac{\sigma L^2}{\mu\sqrt{T}}+\frac{\mu F^2}{\sigma\sqrt{T}}+\frac{\sigma L^2}{\mu T}(s+1)^2M^2+\frac{2\sigma L^2}{\mu\sqrt{T}}(s+1)M$$

We define $\nu=(s+1)M$ and set $\sigma=\frac{F}{L\sqrt{2\nu}}$.

$$\mathbb{E}[R[W]]\leq FL\sqrt{\frac{2\nu}{T}}[\frac{1}{2\nu\mu}+\mu+\frac{\nu}{2\mu\sqrt{T}}+\frac{1}{\mu}]\leq(\mu+3)FL\sqrt{\frac{2\nu}{T}}$$

□

C. REVISION EXAMPLE OF DYNSGD

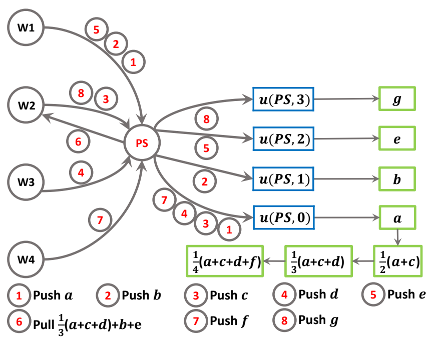


Figure 14: A revision example of DynSGD

Example. We use the example in Figure 14 to illustrate the revision procedure of DynSGD. Supposing that all the workers start from clock 0, the processing procedure is as follows:

1. The PS receives local updates a and b from $W1$, then stores them in $u(PS,0)$ and $u(PS,1)$, respectively. $V(W1)$ becomes two. $S(0)$ and $S(1)$ become two as well.
2. Local updates c and d are pushed to the PS. Since $V(W2)$=$V(W3)$=0 and $S(0)$=2, we first revise $u(PS,0)$ to $\frac{a+c}{2}$ and then to $\frac{a+c+d}{3}$. $S(0)$ becomes four.
3. Update e of $W1$ is stored in $u(PS,2)$ as $V(W1)$=2. And $S(2)$ is changed to two.
4. After finishing clock 0, $W2$ pulls the global parameter. The PS returns $\frac{a+c+d}{3}$+b+e, and $V(W2)$ is modified to three since there are already three versions of global updates on the PS.
5. $W4$ pushes its first update f. Since $V(4)$=0 and $S(0)$=4, we revise $u(PS,0)$ to $\frac{1}{4}(a$+c+d+$f)$.
6. $W2$ finishes clock 1 and generates the local update g. Since $V(W2)$=3, we use g to revise $u(PS,3)$.

D. IMPLEMENTATION DETAILS OF THE PROTOTYPE

Our designed prototype system, namely Angel, follows the classical data-parallel parameter server architecture [17, 41], as shown in Figure 15. There are three roles in our prototype, i.e., the master, the parameter server, and the worker.

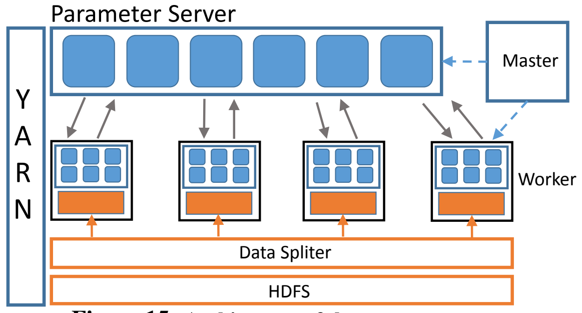


Figure 15: Architecture of the prototype system

Implementation details. We list the major implementation details of the system as follows:

1. The system is programmed in Java and deployed on YARN [40], the next generation MapReduce framework of Hadoop. We use Netty framework to accomplish the message passing.
2. A master is established to govern all the workers and parameter servers through sending periodical heartbeat signals. We also use the master to serve the partition synchronization mechanism as described in Section 6.
3. The PS equally partitions a global parameter by row and by column over parameter servers. We employ range-hash scheme to partition parameter, as described in Section 6.
4. We use HDFS to store the training data, and design a data splitter module to automatically partition training data before running machine learning tasks.
5. We design a data storage module and provide an easy-to-use data reading API for workers, with *memory*, *disk* and *memory-and-disk* storage levels.
6. We provide a failure recovery mechanism for the master and the PS by periodical check point. When encountering failures, master and parameter server can recover from the last check point, while worker restarts and pulls the latest parameter from the PS.
7. We implement some optimisations like other systems, such as KKT filter, parameter pre-fetching, range push and pull [14, 28].

Job Execution. We have implemented several ready-to-run algorithms, such as LR, SVM, KMeans, and LDA. We also offer well-designed interface for users to implement new algorithms by rewriting the logic of worker process. Our prototype executes a submitted job as follows:

1. Before submitting a job, the user should set configurations and hyperparameters, such as M (# workers), P (# parameter servers), s (the staleness threshold), η (the learning rate), and the dimension of model parameter.
2. After a job is submitted, YARN allocates resources for master, parameter servers, and workers.
3. The PS partitions the global parameter and notifies the master of the partitioning results.
4. The data splitter module partitions the training data and notifies every worker of its data shard.
5. Each worker pulls its data shard from HDFS and creates the data storage module.
6. Each worker pulls the global parameter from the PS and starts training according to the designed distributed SGD.

Improvements. Compared with other open-source systems with the PS architecture, such as Bösen [41], TensorFlow [4], and Parameter Server [28], our prototype offers several improvements to enhance the usability and reliability in large-scale industrial environments. First, our deployment, management, and maintenance are naturally integrated with open-source software providing friendly system interfaces and high portability. Second, we provide user-friendly data-loading and data-reading interfaces, while there is no feasible implementation in [28] and [41]. Third, the system offers well-designed failure recovery mechanism that the master and the PS can recover from check points, while [41] and [4] must restart the whole system. Finally, we implement multi-version control for the PS, and also design partition synchronization scheme. In fact, our prototype has been used for many industrial applications in Tencent Inc. for months.

E. EFFECT OF PARTITION SYNCHRONIZATION

As described in Section 6, we design a partition synchronization approach to prevent the desynchrony between different partitions. In order to assess the effect of this method, we use the URL dataset to train LR with DynSGD on Cluster-2.

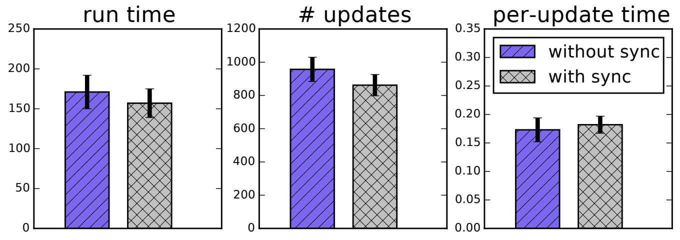


Figure 16: Effect of partition synchronization (DynSGD, LR, URL, s=3, M=30, 10%batch), run time and per-update time are in seconds

As Figure 16 shows, applying the proposed synchronization scheme increases the statistical efficiency by 11%, meaning that fewer updates are needed to converge. Although the partition synchronization induces extra communication with the master, it decreases the total run time by 8.9%.