Slow and Stale Gradients Can Win the Race

Sanghamitra Dutta, Jianyu Wang, and Gauri Joshi

***Abstract*—Distributed Stochastic Gradient Descent (SGD) when run in a synchronous manner, suffers from delays in *runtime* as it waits for the slowest workers (stragglers). Asynchronous methods can alleviate stragglers, but cause gradient staleness that can adversely affect the convergence *error*. In this work, we present a novel theoretical characterization of the speedup offered by asynchronous methods by analyzing the trade-off between the *error* in the trained model and the actual training *runtime* (wallclock time). The main novelty in our work is that our runtime analysis considers random straggling delays, which helps us design and compare distributed SGD algorithms that strike a balance between straggling and staleness. We also provide a new error convergence analysis of asynchronous SGD variants without bounded or exponential delay assumptions. Finally, based on our theoretical characterization of the error-runtime trade-off, we propose a method of gradually varying synchronicity in distributed SGD and demonstrate its performance on the CIFAR10 dataset.**

***Index Terms*—Asynchronous stochastic gradient descent, distributed machine learning, optimization, performance analysis, stragglers.**

I. Introduction

STOCHASTIC gradient descent (SGD) is the backbone of most state-of-the-art machine learning algorithms. Thus, improving the stability and convergence rate of SGD algorithms is critical for making machine learning algorithms fast and efficient. Classical SGD was designed to be run on a single computing node, and its error-convergence with respect to the number of iterations has been extensively analyzed and improved in optimization and learning theory literature. Due to the massive training data-sets and deep neural network architectures used today, running SGD at a single node can be prohibitively slow. This calls for distributed implementations of SGD, where gradient computation and aggregation is parallelized across multiple worker nodes. Although parallelism boosts the amount of data processed per iteration, it exposes SGD to unpredictable node slowdown and communication delays stemming from variability in the computing infrastructure. Thus, there is a critical need to make distributed SGD fast, and yet robust to system variability.

Manuscript received February 28, 2021; revised June 16, 2021; accepted August 4, 2021. Date of publication August 10, 2021; date of current version September 20, 2021. This work was supported in part by the IBM Faculty Award; in part by the National Science Foundation CRII Award under Grant CCF-1717314; and in part by the Qualcomm Innovation Fellowship. Some of the results have appeared in AISTATS 2018. This is an extended version with additional results, in particular, an adaptive synchronicity strategy called AdaSync. *(Corresponding author: Sanghamitra Dutta.)*

The authors are with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: sanghamd@andrew.cmu.edu; jianyuw1@andrew.cmu.edu; gaurij@andrew.cmu.edu).

This article has supplementary downloadable material available at https://doi.org/10.1109/JSAIT.2021.3103770, provided by the authors.

Digital Object Identifier 10.1109/JSAIT.2021.3103770

The convergence speed of distributed SGD depends on two factors: 1) the error in the trained model versus the number of iterations, and 2) the number of iterations completed per second. Traditional single-node SGD analysis focuses on optimizing the first factor, because the second factor is generally a constant when SGD is run on a single dedicated server. In distributed SGD, which is often run on shared cloud infrastructure, the second factor depends on several aspects such as the number of worker nodes, their gradient computation delays, and the protocol (synchronous or asynchronous) used to aggregate their gradients. Hence, in order to achieve the fastest convergence speed we need: 1) optimization techniques to maximize the error-convergence rate with respect to iterations, and 2) scheduling techniques to maximize the number of iterations completed per second. These directions are inter-dependent and need to be explored together rather than in isolation. While many works have advanced the first direction, the second is less explored from a theoretical point of view, and the juxtaposition of both is an unexplored problem. Our goal is to design SGD algorithms that easily lend themselves to distributed implementations, and are robust to fluctuations in computation and network delays as well as unpredictable node failures. This work improves the true convergence speed of distributed SGD with respect to wallclock time by jointly designing scheduling techniques to reduce per-iteration delay, and optimization algorithms to minimize error-versus-iterations.

A commonly used distributed SGD framework, which is first deployed at a large-scale in Google's DistBelief [1], is the parameter server framework, which consists of a central parameter server (PS) that is used to aggregate gradients computed by worker nodes as shown in Figure 1 (a). In synchronous SGD, the PS waits for all workers to push gradients before it updates the model parameters. Random delays in computation (referred to as straggling) are common in today's distributed systems as pointed out in the influential work of [2]. Waiting for slow and straggling workers can diminish the speedup offered by parallelizing the training. To alleviate the problem of stragglers, SGD can be run in an asynchronous manner, where the central parameters are updated without waiting for all workers. However, workers may return *stale* gradients that were evaluated at an older version of the model, and this can make the algorithm unstable. Synchronous SGD typically has better convergence error but has a higher wallclock runtime per iteration because it requires synchronization of straggling workers. On the other hand, asynchronous SGD has faster wallclock runtime per iteration but it also has higher convergence error due to the problem of gradient staleness.

Our goal is to achieve the lower envelope of the error-runtime trade-offs achieved by synchronous and asynchronous

2641-8770 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on May 31,2022 at 14:55:50 UTC from IEEE Xplore. Restrictions apply.

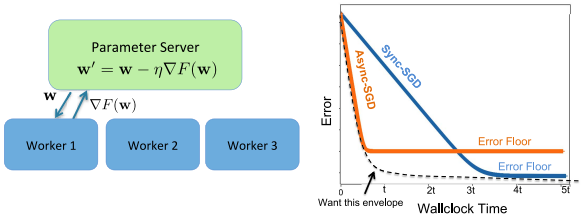


Fig. 1. (a) The parameter server framework (b) Synchronous SGD has lower error floor but higher runtime, while, asynchronous SGD converges faster but has a higher error floor. We want to achieve the lower envelope between the two curves which characterizes the best error-runtime trade-off.

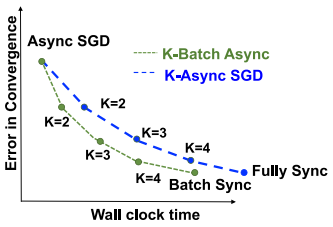


Fig. 2. Distributed SGD variants span the error-runtime trade-off between fully Sync-SGD and fully Async-SGD. Here K is the number of workers or mini-batches the PS waits for before updating the model parameters, as we elaborate in Section II.

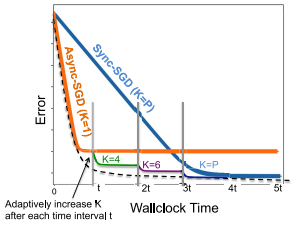
SGD (see Figure 1(b)), which characterizes the best error-runtime trade-off. Towards achieving this goal, in this work we present a systematic theoretical analysis of the trade-off between error and the actual runtime (instead of iterations), modeling wallclock runtimes as random variables with a general distribution. Based on our analysis, we propose AdaSync, which is a method of adaptively increasing the number of nodes whose gradients are aggregated synchronously by the central PS. Our theoretical results are also substantiated with experiments on CIFAR10 [3] dataset.

A. Main Contributions

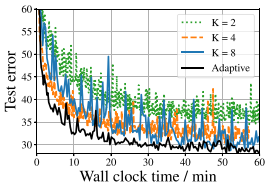
Existing machine learning algorithms mostly try to optimize the trade-off of error with the number of iterations, epochs or "work complexity" [4], [5], while assuming the time spent per iteration to be a constant. However, due to straggling and synchronization bottle-necks in the system, the same gradient computation task can often take different time to complete across different workers or iterations [2]. This work departs from the classic optimization theory view of analyzing error convergence with respect to the number of iterations and takes the novel approach of minimizing the error with respect to the wallclock time. By taking a joint runtime and error optimization approach, we provide the first comprehensive runtime-per-iteration comparison of SGD variants and design adaptive synchronous SGD algorithms that can achieve a super-linear runtime speed-up over naive synchronous SGD, while still preserving a low error floor. The main contributions of this paper are summarized below.

- *Straggler-Resilient Variants of Synchronous and Asynchronous SGD:* In order to a strike a balance between the two extremes: synchronous and asynchronous SGD, we propose partially synchronous SGD variants such as K-sync, K-batch-sync, K-async and K-batch-async SGD, where K is the number of workers (out of the total of P workers) that the parameter waits for when aggregating gradients. Although some of these distributed SGD variants have been proposed previously, to the best of our knowledge, this is the first work to provide a unified error convergence and runtime analysis of these variants.
- *Runtime Analysis of the Distributed SGD Variants:* We provide the first systematic analysis of the expected runtime per iteration of synchronous and asynchronous SGD and their variants. We do so by modelling the runtimes at each worker as random variables with an arbitrary general distribution. For commonly used delay distributions such as exponential, asynchronous SGD is $O(P \log P)$ times faster than synchronous SGD where P is the total number of workers.
- *More General Error Analysis of Asynchronous SGD Variants:* We propose a new error convergence analysis for asynchronous SGD and its variants for *strongly convex* objectives that can also be extended to provide relaxed guarantees for *non-convex* formulations. In this analysis we relax the bounded delay assumption in [6] and the bounded gradient assumption in [7]. We also remove the assumption of exponential computation time and the staleness process being independent of the parameter values [8] as we will elaborate in Section IV. Interestingly, our analysis also brings out the regimes where asynchronous SGD can be better or worse than synchronous SGD in terms of speed of convergence.
- *Insights from the Error-versus-wallclock Time Trade-off:* By combining our runtime and error analysis described above, we can theoretically characterize the error-versus-wallclock time trade-off for different SGD variants. Figure 2 illustrates the error at convergence (or error floor) versus the time to reach convergence of different SGD variants. Observe how the K-batch-async and K-async strategies can span different points on the trade-off as K varies. By choosing the right value of K we can achieve a desired error at convergence in minimum time. The theoretical results presented in this paper are corroborated by rigorous experiments on training deep neural networks for classification of the CIFAR10 dataset.
- *AdaSync strategy to Adapt Synchronocity during Training:* Instead of fixing K, we can achieve a win-win in the error-runtime trade-off by adapting K so as to gradually increasing the synchrony of the different SGD variants. We propose AdaSync, a method that uses the theoretical characterization of the error-runtime trade-off, to decide how to adapt K, as illustrated in Figure 3. This method is inspired from [9], [10] which adapts the communication frequency for a different class of SGD methods known as periodic averaging SGD. Interestingly, similar to [9], our proposed method does not require knowledge of the algorithm parameters such as Lipschitz constant, variance of the stochastic gradient etc. as one would otherwise require if they choose to

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on May 31,2022 at 14:55:50 UTC from IEEE Xplore. Restrictions apply.



(a) Idea behind AdaSync.



(b) Adaptive K-async for CIFAR 10 classification.

Fig. 3. We propose AdaSync, a method of adaptively increasing K which is a measure of the synchronicity of the algorithm. AdaSync aggregates gradients from any K out the P total nodes. It helps achieve the best error-runtime trade-off by gradually increasing K.

simply minimize the error-runtime trade-off with respect to parameter K. Experimental results on CIFAR 10 (see Figure 3b) show that AdaSync not only helps achieve the same training loss much faster but also gives smaller test error than fixed-K strategies.

B. Related Works

Single Node SGD: Analysis of gradient descent dates back to classical works [11] in the optimization community. The problem of interest is the minimization of empirical risk:

$$\min_{\mathbf{w}} \left\{ F(\mathbf{w}) \stackrel{\text{def}}{=} \frac{1}{N} \sum_{n=1}^{N} f(\mathbf{w}, \xi_n) \right\}. \tag{1}$$

Here, ξ_n denotes the $n-$th data point and its label where $n = 1, 2, \ldots, N$, and $f(\mathbf{w}, \xi_n)$ denotes the composite loss function. Gradient descent is a way to iteratively minimize this objective function by updating the parameter $\mathbf{w}$ in the opposite direction of the gradient of $F(\mathbf{w})$ at every iteration, as given by:

$$\mathbf{w}_{j+1} = \mathbf{w}_j - \eta \nabla F(\mathbf{w}_j) = \mathbf{w}_j - \frac{\eta}{N} \sum_{n=1}^{N} \nabla f\big(\mathbf{w}_j, \xi_n\big).$$

The computation of $\sum_{n=1}^{N} \nabla f(\mathbf{w}_j, \xi_n)$ over the entire dataset is expensive. Thus, stochastic gradient descent [12] with mini-batching is generally used in practice, where the gradient is evaluated over small, randomly chosen subsets of the data. Smaller mini-batches result in higher variance of the gradients, which affects convergence and error floor [4], [13], [14]. Algorithms such as AdaGrad [15] and Adam [16] gradually vary learning rate to achieve a lower error floor. Another class of algorithms includes stochastic variation reduction techniques such as SVRG [17], SAGA [18] and their variants listed out in [19]. For a detailed survey of SGD variants, see [20].

Synchronous SGD and Stragglers: To process large datasets, SGD is parallelized across multiple workers with a central PS. Each worker processes one mini-batch, and the PS aggregates all the gradients. The convergence of synchronous SGD is same as mini-batch SGD, with a P-fold larger mini-batch, where P is the number of workers. However, the time per iteration grows with the number of workers, because straggling workers slow down randomly [2]. Thus, it is important to juxtapose the error reduction per iteration with the runtime per iteration to understand the true convergence speed of distributed SGD.

Asynchronous SGD and Staleness: A complementary approach to deal with the issue of straggling is to use asynchronous SGD. In asynchronous SGD, any worker can evaluate the gradient and update the central PS without waiting for the other workers, as explained in more detail in Section II. Asynchronous variants of existing SGD algorithms have also been proposed and implemented in systems [1], [21]–[24]. In general, analyzing the convergence of asynchronous SGD with the number of iterations is difficult in itself because of the randomness of gradient staleness. There are only a few pioneering works such as [6]–[8], [25]–[34] in this direction. In [25], a fully decentralized analysis was proposed that considers no central PS. In [7], a new asynchronous algorithm called Hogwild was proposed and analyzed under bounded gradient and bounded delay assumptions. This direction of research has been followed upon by several interesting works such as [6] which proposed novel theoretical analysis under bounded delay assumption for other asynchronous SGD variants. In [30]–[33], the framework of ARock was proposed for parallel coordinate descent and analyzed using Lyapunov functions, relaxing several existing assumptions such as bounded delay assumption and the independence of the delays and the index of the blocks being updated. In algorithms such as Hogwild, ARock etc. every worker only updates a part of the central parameter vector $\mathbf{w}$ at every iteration and are thus essentially different in spirit from conventional asynchronous SGD settings [6], [26] where every worker updates the entire $\mathbf{w}$. In an alternate direction of work [27], asynchrony is modeled as a perturbation. In this work, we present a new and simpler analysis of asynchronous SGD with number of iterations that relaxes some of the assumptions in previous literature, and helps us to characterize the error-runtime trade-off as well as easily derive adaptive update rule for gradually increasing synchrony.

Erasure Coded Computing. To deal with stragglers and speed up machine learning, system designers have proposed several straggler mitigation techniques such as [35] that try to detect and avoid stragglers. An alternate direction of work is to use redundancy techniques, e.g., replication or erasure codes, as proposed in [36]–[59] to deal with the stragglers. While recent works such as [41] propose erasure coding techniques to overcome straggling workers, the SGD variants considered in this paper such as K-sync and K-batch-sync SGD exploit the inherent redundancy in the data itself, and ignore the gradients returned by straggling workers. If the data is well-shuffled such that it can be assumed to be i.i.d. across workers, then for the same effective batch-size, ignoring straggling gradients will give equivalent error scaling as coded strategies, and at a lower computing cost. However, coding strategies may be useful in the non i.i.d. case, when the gradients supplied by each worker provide diverse information that is important to capture in the trained model. Other related works on gradient coding or approximate gradient coding include [48], [60]. Following our initial conference publication, other interesting related works include [61]–[67] that bridge distributed gradient descent, coding theory, and asynchrony.

The rest of the paper is organized as follows. Section II describes our problem formulation introducing the system

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on May 31,2022 at 14:55:50 UTC from IEEE Xplore. Restrictions apply.

model and assumptions. Section III provides the theoretical results on the analysis of true wallclock runtime per iteration for the different SGD variants and provides insights on quantifying the speedups that one variant provides over another. In Section IV, we discuss our analysis of error convergence with number of iterations where we also include our new convergence analysis for asynchronous and K-async SGD. Proofs and detailed discussions are presented in the Appendix in the supplementary material. In Section V, we combine the runtime analysis with the error analysis to derive novel error-runtime trade-offs and demonstrate how our analysis could inform predicting the trend of the trade-off for distributed systems with different runtime distributions. Finally, in Section VI, we introduce our proposed method AdaSync that gradually varies synchronicity (parameter K) to achieve the desirable error-runtime trade-off, followed by experimental results. We conclude with a brief discussion in Section VII.

II. Problem Formulation

Our objective is to minimize the risk function of the parameter vector $\mathbf{w}$ as mentioned in (1) given N training samples. Let S denote the total set of N training samples, i.e., a collection of some data points with their corresponding labels. We use the notation ξ to denote an index $\in S$ (either a single datapoint and its label or a single mini-batch of m samples of data and their labels).

A. System Model

We assume that there is a central parameter server (PS) with P parallel workers as shown in Section I. The workers fetch the current parameter vector $\mathbf{w}_j$ from the PS as and when instructed in the algorithm. Then they compute gradients using one mini-batch and push their gradients back to the PS as and when instructed in the algorithm. At each iteration, the PS aggregates the gradients computed by the workers and updates the parameter $\mathbf{w}$. Based on how these gradients are fetched and aggregated, we have different variants of synchronous or asynchronous SGD.

B. Variants of SGD

We now describe the SGD variants considered in this paper. We note that some of these variants have been proposed earlier under alternate names in different papers, as we will refer to during our descriptions. In this work, we give a unified runtime and error analysis to compare them with each other in terms of their true error-runtime trade-off, a problem that has not been considered in prior works. Please refer to Figure 4 and Figure 5 for a pictorial illustration of the SGD variants.

K-sync SGD: This is a generalized form of synchronous SGD, also suggested in [21], [68] to offer some resilience to straggling as the PS does not wait for all the workers to finish. The PS only waits for the first K out of P workers to push their gradients. Once it receives K gradients, it updates $\mathbf{w}_j$ and cancels the remaining workers. The updated $\mathbf{w}_{j+1}$ is sent to all

Fig. 4. For $K = 2$ and $P = 3$, we illustrate the K-sync and K-batch-sync SGD in comparison with fully synchronous SGD. Lightly shaded arrows indicate straggling gradient computations that are canceled.

Fig. 5. For $K = 2$ and $P = 3$, we illustrate the K-async and K-batch-async algorithms in comparison with fully asynchronous SGD.

P workers for the next iteration. The update rule is given by:

$$\mathbf{w}_{j+1} = \mathbf{w}_j - \frac{\eta}{K}\sum_{l=1}^{K} g\big(\mathbf{w}_j, \xi_{l,j}\big). \tag{2}$$

Here $l = 1, 2, \ldots, K$ denotes the index of the K workers that finish first, $\xi_{l,j}$ denotes the mini-batch of m samples used by the l-th worker at the j-th iteration and $g(\mathbf{w}_j, \xi_{l,j}) = \frac{1}{m}\sum_{\xi\in\xi_{l,j}} \nabla f(\mathbf{w}_j, \xi)$ denotes the average gradient of the loss function evaluated over the mini-batch $\xi_{l,j}$ of size m. For $K = P$, the algorithm is equivalent to a fully synchronous SGD with P workers.

K-batch-sync SGD: In K-batch-sync, all the P workers start computing gradients with the same $\mathbf{w}_j$. Whenever any worker finishes, it pushes its update to the PS and evaluates the gradient on the next mini-batch at the same $\mathbf{w}_j$. The PS updates using the first K mini-batches that finish and cancels the remaining workers. Theoretically, the update rule is still the same as (2) but here l now denotes the index of the mini-batch (out of the K mini-batches that finished first) instead of the worker. However K-batch-sync will offer advantages over K-sync in runtime per iteration as no worker is idle.

K-async SGD: This is a generalized version of asynchronous SGD, also suggested in [21]. In K-async SGD, all the P workers compute their respective gradients on a single mini-batch. The PS waits for the first K out of P that finish first, but it does not cancel the remaining workers. As a result, for every update the gradients returned by each worker might be computed at a stale or older value of the parameter $\mathbf{w}$. The update rule is thus given by:

$$\mathbf{w}_{j+1} = \mathbf{w}_j - \frac{\eta}{K}\sum_{l=1}^{K} g\big(\mathbf{w}_{\tau(l,j)}, \xi_{l,j}\big). \tag{3}$$

Here $l = 1, 2, \ldots, K$ denotes the index of the K workers that contribute to the update at the corresponding iteration, $\xi_{l,j}$ is one mini-batch of m samples used by the l-th worker at the j-th iteration and $\tau(l, j)$ denotes the iteration index when the l-th worker last read from the central PS where $\tau(l, j) \leq j$. Also, $g(\mathbf{w}_{\tau(l,j)}, \xi_{l,j}) = \frac{1}{m}\sum_{\xi\in\xi_{l,j}} \nabla f(\mathbf{w}_{\tau(l,j)}, \xi_{l,j})$ is the average gradient of the loss function evaluated over the mini-batch $\xi_{l,j}$ based on the stale value of the parameter $\mathbf{w}_{\tau(l,j)}$. For $K = 1$,

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on May 31,2022 at 14:55:50 UTC from IEEE Xplore. Restrictions apply.

TABLE I
IMPORTANT NOTATIONS

CONSTANTS	RANDOM VARIABLES
J: Total Iterations	T: Runtime per iteration
P: Total number of workers (processors/nodes)	X_i: Runtime of a worker for a mini-batch
K: Number of workers or mini-batches to wait for	$X_{K:P}$: K-th order statistic out of P iid runtimes
m: Mini-batch size	$\mathbf{w}_j$: Model value at iteration j

the algorithm is equivalent to fully asynchronous SGD, and the update rule can be simplified as:

$$\mathbf{w}_{j+1} = \mathbf{w}_j - \eta g\big(\mathbf{w}_{\tau(j)}, \xi_j\big). \tag{4}$$

Here ξ_j denotes the set of samples used by the worker that updates at the j-th iteration such that $|\xi_j| = m$ and $\tau(j)$ denotes the iteration index when that particular worker last read from the central PS. Note that $\tau(j) \leq j$.

K-batch-async SGD: Observe in Figure 5 that K-async also suffers from some workers being idle while others are still working on their gradients until any K finish. In K-batch-async (proposed in [6]), the PS waits for K mini-batches before updating itself but irrespective of which worker they come from. So wherever any worker finishes, it pushes its gradient to the PS, fetches current parameter at PS and starts computing gradient on the next mini-batch based on the current value of the PS. Surprisingly, the update rule is again similar to (3) theoretically except that now l denotes the indices of the K mini-batches that finish first instead of the workers and $\mathbf{w}_{\tau(l,j)}$ denotes the version of the parameter when the worker computing the $l-$th mini-batch last read from the PS. While the error convergence of K-batch-async is similar to K-async, it reduces the runtime per iteration as no worker is idle.

C. Performance Metrics and Goal

Our metrics of interest are:

Definition 1 (Expected Runtime per iteration): The expected runtime per iteration is the expected time (average time) taken to perform each iteration, i.e., the expected time between two consecutive updates of the parameter $\mathbf{w}$ at the central PS.

Definition 2 (Expected Error after J iterations): The expected error after J iterations is defined as $\mathbb{E}\big[F(\mathbf{w}_J) - F^*\big]$, i.e., the expected gap of the risk function from its optimal value.

Goal: Our goal is to determine the trade-off between the expected error (measures the accuracy of the algorithm) and the expected runtime **after a total of J iterations** for the different SGD variants. Based on our characterization, we would like to derive a method of gradually varying synchronicity (parameter K) in the SGD variants to achieve a desirable error-runtime trade-off.

In Table I, we provide a list of some of the important notations used in this paper.

TABLE II
EXPECTED RUNTIME FOR THE DIFFERENT SGD VARIANTS

SGD Variant	Expected Runtime per iteration $\mathbb{E}[T]$
K-sync	$\mathbb{E}[T] = \mathbb{E}[X_{K:P}]$ (all distributions)
K-batch-sync	$\mathbb{E}[T] \leq K\mathbb{E}[X_{1:P}]$ (new-longer-than-used)
K-async	$\mathbb{E}[T] \leq \mathbb{E}[X_{K:P}]$ (new-longer-than-used)
K-batch-async	$\mathbb{E}[T] = \frac{K}{P}\mathbb{E}[X]$ (all distributions)

III. RUNTIME ANALYSIS: INSIGHTS ON QUANTIFYING SPEEDUP

Our runtime analysis provides useful insights in quantifying the speedup offered by different SGD variants. We first state our key modeling assumptions in Section III-A, followed by our main theoretical results on quantifying speedup in Section III-B. Next, we include our detailed runtime analysis for the four SGD variants considered in this paper in Section III-C, some of which are useful in the proofs of the main results on speedups. For a summary of the expected runtime of the different SGD variants, we refer to Table II.

A. Modeling Assumptions

The time taken by a worker to compute gradient on one mini-batch is denoted by random variable X, whose realization X_i for $i = 1, 2, \ldots, P$ are i.i.d. across mini-batches and workers.

The assumption that the computational times are identically distributed across workers makes the analysis tractable. Since the X_i's are random variables, the actual computational times (values of X_i) can still be quite different across workers. It would be an interesting future work to extend the analysis to the case of non-identical distributions across workers. If the distributions are known to be non-identical, i.e., the workers are known to be heterogeneous, one might also consider alternate techniques of allocating the tasks that somehow leverages this heterogeneity, e.g., more mini-batches to the faster workers, which can also be explored as future work.

B. Main Results on Quantifying Speedups

Our first result Theorem (1) analytically characterizes the speedup offered by asynchronous SGD for *any general distribution on the wallclock time of each worker*.

Theorem 1: Let the wallclock time of each worker to process a single mini-batch be i.i.d. random variables $X_1, X_2, \ldots, X_P \sim F_X$. Then the ratio of the expected runtimes per iteration for synchronous and asynchronous SGD is

$$\frac{\mathbb{E}\big[T_{Sync}\big]}{\mathbb{E}\big[T_{Async}\big]} = P\frac{\mathbb{E}[X_{P:P}]}{\mathbb{E}[X]} \tag{5}$$

where $X_{P:P}$ is the P^{th} order statistic of P i.i.d. random variables $X_1, X_2, \ldots, X_P$.

Proof of Theorem 1: Fully synchronous SGD is actually K-sync SGD with $K = P$, i.e., waiting for all the P workers to finish. On the other hand, fully asynchronous SGD is actually K-batch-async with $K = 1$. By taking the ratio of the expected runtimes per iteration for K-sync SGD (see Lemma 1 in Section III-C) with $K = P$ and K-batch-async (see

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on May 31,2022 at 14:55:50 UTC from IEEE Xplore. Restrictions apply.

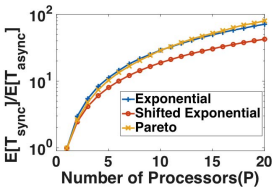


Fig. 6. Simulation demonstrating the speedup using asynchronous over synchronous SGD for three different types of distributions, namely, $Exp(1)$, $1 + Exp(1)$ and $Pareto(2, 1)$. The speedup is shown in log-scale, *i.e.*, $\log \frac{\mathbb{E}[T_{Sync}]}{\mathbb{E}[T_{Async}]}$ is plotted with total number of nodes P.

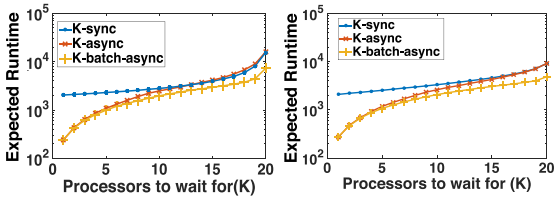


Fig. 7. Simulation of expected runtime for 2000 iterations for K-sync, K-async and K-batch-async SGD: (Left) Pareto distribution $Pareto(2, 1)$ and (Right) Shifted exponential distribution $1 + Exp(1)$.

Lemma 5 in Section III-C) with $K = 1$, we get the result in Theorem 1. ∎

In the following corollary, we highlight this speedup for the special case of exponential computation time.

Corollary 1: Let the wallclock time of each worker to process a single mini-batch be i.i.d. exponential random variables $X_1, X_2, \ldots, X_P \sim Exp(\mu)$. Then the ratio of the expected runtimes per iteration for synchronous and asynchronous SGD is $\Theta(P \log P)$.

Thus, the speedup scales with P and can diverge to infinity for large P. We illustrate the speedup for different distributions in Figure 6. It might be noted that a similar speedup as Corollary 1 has also been obtained in a recent work [31] under exponential assumptions.

Proof of corollary 1: The expectation of the maximum of P i.i.d. $X_i \sim Exp(\mu)$ is $\mathbb{E}[X_{P:P}] = \sum_{i=1}^{P} \frac{1}{i\mu} \approx \frac{\log P}{\mu}$ [69]. This can be substituted in Theorem 1 to get corollary 1. ∎

The next result illustrates the advantages offered by K-batch-sync and K-batch-async over their corresponding counterparts K-sync and K-async respectively.

Theorem 2: Let the wallclock time of each worker to process a single mini-batch be i.i.d. exponential random variables $X_1, X_2, \ldots, X_P \sim Exp(\mu)$. Then the ratio of the expected runtimes per iteration for K-async (or sync) SGD and K-batch-async (or sync) SGD is

$$\frac{\mathbb{E}\left[T_{K-async}\right]}{\mathbb{E}\left[T_{K-batch-async}\right]} = \frac{P\mathbb{E}[X_{K:P}]}{K\mathbb{E}[X]} \overset{1<K<P}{\approx} \frac{P \log \frac{P}{P-K}}{K}$$

where $X_{K:P}$ is the K^{th} order statistic of i.i.d. random variables $X_1, X_2, \ldots, X_P$.

Proof of Theorem 2: For the exponential X_i, equality holds in (11) in Lemma 3, as we justify in Appendix B-C1, in the supplementary material. The expectation can be derived as $\mathbb{E}[X_{K:P}] = \sum_{i=P-K+1}^{P} \frac{1}{i\mu} \approx \frac{\log (P/P-K)}{\mu}$ for $1 < K < P$. For exponential X_i, the expected runtime per iteration for K-batch-async is given by $\mathbb{E}[T] = \frac{K\mathbb{E}[X]}{P} = \frac{K}{\mu P}$ from Lemma 5. ∎

We have $\frac{\mathbb{E}\left[T_{K-async}\right]}{\mathbb{E}\left[T_{K-batch-async}\right]} = \frac{P\mathbb{E}[X_{K:P}]}{K\mathbb{E}[X]} = 1$ (for $K = 1$) since $\mathbb{E}[X_{1:P}] = \frac{1}{P\mu}$ (also notice that 1-async and 1-batch-async are equivalent). When $K = P$, we have $\frac{\mathbb{E}\left[T_{K-async}\right]}{\mathbb{E}\left[T_{K-batch-async}\right]} = \frac{P\mathbb{E}[X_{K:P}]}{K\mathbb{E}[X]} \approx \log P$ since $\mathbb{E}[X_{P:P}] = \sum_{i=1}^{P} \frac{1}{i\mu} \approx \frac{\log P}{\mu}$.

Theorem 2 shows that as $\frac{K}{P}$ increases, the speedup using K-batch-async increases and can be upto $\log P$ times higher. For non-exponential distributions, we simulate the behavior of expected runtime in Figure 7 for K-sync, K-async and K-batch-async respectively for Pareto and Shifted Exponential.

C. Runtime Analysis for Four Different SGD Variants

Here, we rigorously analyze the theoretical wallclock runtime of the four different SGD variants. These results have also been used in providing insights on the speedup offered by different asynchronous and batch variants in Theorem 1 and Theorem 2.

1) Runtime of K-Sync SGD:

Lemma 1 (Runtime of K-sync SGD): The expected runtime per iteration for K-sync SGD is,

$$\mathbb{E}[T] = \mathbb{E}[X_{K:P}] \tag{6}$$

where $X_{K:P}$ is the K^{th} order statistic of P i.i.d. random variables $X_1, X_2, \ldots, X_P$.

Proof of Lemma 1: When all the workers start together, and we wait for the first K out of P i.i.d. random variables to finish, the expected computation time for that iteration is $\mathbb{E}[X_{K:P}]$, where $X_{K:P}$ denotes the K-th statistic of P i.i.d. random variables $X_1, X_2, \ldots, X_P$. ∎

Thus, for a total of J iterations, the expected runtime is given by $J\mathbb{E}[X_{K:P}]$.

Remark 1: For $X_i \sim Exp(\mu)$, the expected runtime per iteration is given by,

$$\mathbb{E}[T] = \frac{1}{\mu} \sum_{i=P-K+1}^{P} \frac{1}{i} \approx \frac{1}{\mu}\left(\frac{\log \frac{P}{P-K}}{\mu}\right)$$

where the last step uses an approximation from [69]. For justification, the reader is referred to Appendix B-A, in the supplementary material.

2) Runtime of K-Batch-Sync SGD: The expected runtime of K-batch-sync SGD is not analytically tractable in general, but we obtain an upper bound on it for a class of distributions called the "new-longer-than-used" distributions, as defined below.

Definition 3 (New-longer-than-used): A random variable U is said to have a new-longer-than-used distribution if the following holds for all $t, u \geq 0$:

$$\Pr(U > u + t | U > t) \leq \Pr(U > u). \tag{7}$$

To understand the intuition behind this notion, let a random variable U denote the computational time taken to perform a task. Suppose that the task has been running for t units of time but has not finished, and the scheduler needs to decide whether to keep the task running or abort it and launch a new

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on May 31,2022 at 14:55:50 UTC from IEEE Xplore. Restrictions apply.

copy. If U has a new-longer-than-used distribution, then a new copy is expected to take longer than waiting for the already running task to finish. Most of the continuous distributions we encounter like normal, shifted-exponential, gamma, beta are new-longer-than-used. Alternately, the hyper-exponential distribution is new-shorter-than-used and it satisfies

$$\Pr(U > u + t | U > t) \geq \Pr(U > u) \quad \forall t, u \geq 0. \tag{8}$$

For the exponential distribution, the inequality (7) holds with equality due to the memoryless property, and thus it can be thought of as both new-longer-than-used and new-shorter-than-used.

Lemma 2 (Runtime of K-batch-sync SGD): Suppose that each X_i has a new-longer-than-used distribution. Then, the expected runtime per iteration for K-batch-sync is upper-bounded as

$$\mathbb{E}[T] \leq K\mathbb{E}[X_{1:P}] \tag{9}$$

where $X_{1:P}$ is the minimum of P i.i.d. random variables $X_1, X_2, \ldots, X_P$.

The proof is provided in Appendix B-B, in the supplementary material. For the special case of $X_i \sim Exp(\mu)$, the runtime per iteration is distributed as $Erlang(K, P\mu)$ (see Appendix B-B, in the supplementary material). Thus, for K-batch-sync SGD,

$$\mathbb{E}[T] = \frac{K}{P\mu}, \tag{10}$$

which is what we obtain when (9) holds with equality.

3) Runtime of K-Async SGD: The expected runtime per iteration of K-async SGD is also not analytically tractable for non-exponential X_i, but we obtain an upper bound on it for "new-longer-than-used" distributions.

Lemma 3 (Runtime of K-async SGD): Suppose that each X_i has a new-longer-than-used distribution. Then, the expected runtime per iteration for K-async is upper-bounded as

$$\mathbb{E}[T] \leq \mathbb{E}[X_{K:P}] \tag{11}$$

where $X_{K:P}$ is the K^{th} order statistic of P i.i.d. random variables $X_1, X_2, \ldots, X_P$.

The proof of this lemma is provided in Appendix B-C, in the supplementary material.

Remark 2: Recall that the runtime of K-sync SGD is $\mathbb{E}[X_{K:P}]$. Therefore, Lemma 3 essentially implies that for new-longer-than-used distributions, the runtime of K-async SGD is upper-bounded by the runtime of K-sync SGD. For the special case of exponential runtimes, (7) holds with equality, and the expected runtime of both K-sync SGD and K-async SGD match theoretically. However, for other classes of distributions where (7) holds with strict inequality, the upper bound of Lemma 3 also holds with strict inequality. In general, intuitively the "more" is the new-longer-than-used property, the lower is the runtime of K-async SGD as compared to K-sync SGD. We show this by explicitly deriving an alternate upper bound for the shifted-exponential distribution (a special case of the new-longer-than-used distributions) that is lower than the expected runtime of K-sync SGD when the shift of the distribution is large.

Lemma 4 (Alternate Upper Bound for Shifted Exponential): Let $P = nK$ where n is an integer greater than 1, and each X_i follow a shifted exponential distribution with shift Δ, i.e., $X_i \sim \Delta + Exp(\mu)$. Then, the expected runtime for any n consecutive iterations of K-async SGD is upper-bounded as

$$\mathbb{E}[T_1 + T_2 + \ldots + T_n] \leq \Delta + \sum_{i=0}^{n-1} \mathbb{E}\big[\widetilde{X}_{K:(P-iK)}\big], \tag{12}$$

where $\widetilde{X} \sim Exp(\mu)$ and $\widetilde{X}_{K:(P-iK)}$ denotes the K^{th} order statistic out of $P - iK$ i.i.d. random variables.

The proof of this lemma is also provided in Appendix B-C, in the supplementary material. Based on this lemma, the upper-bound on per-iteration runtime can be approximated as:

$$\begin{aligned}
&\frac{1}{n}\left(\Delta + \sum_{i=0}^{n-1} \mathbb{E}\big[\widetilde{X}_{K:(P-iK)}\big]\right) \\
&\approx \frac{\Delta}{n} + \frac{1}{n\mu}\left(\log\frac{P}{P-K} + \log\frac{P-K}{P-2K}\right. \\
&\qquad\qquad \left. + \ \ldots + \log\frac{P-(n-2)K}{P-(n-1)K}\right) + \frac{1}{n\mu}\log(K) \\
&\approx \frac{\Delta}{n} + \frac{\log P}{n\mu} = \frac{K\Delta}{P} + \frac{K\log P}{P\mu}.
\end{aligned} \tag{13}$$

In comparison, the runtime per iteration for K-sync SGD is $(\Delta + \frac{\log P/(P-K)}{\mu})$. Thus, a high value of Δ implies that the runtime of K-async SGD is strictly lower than K-sync SGD.

4) Runtime of K-Batch-Async SGD: We derive an expression that holds for any distribution on X_i.

Lemma 5 (Runtime of K-batch-async SGD): The expected runtime per iteration for K-batch-async SGD in the limit of large number of iterations is given by:

$$\mathbb{E}[T] = \frac{K\mathbb{E}[X]}{P}. \tag{14}$$

Unlike the results for the synchronous variants, this result on average runtime per iteration holds only in the limit of large number of iterations. To prove the result we use ideas from renewal theory. For a brief background on renewal theory, the reader is referred to Appendix B-D, in the supplementary material.

Proof of Lemma 5: For the i-th worker, let $\{N_i(t),\ t > 0\}$ be the number of times the i-th worker pushes its gradient to the PS over in time t. The time between two pushes is an independent realization of X_i. Thus, the inter-arrival times $X_i^{(1)}, X_i^{(2)}, \ldots$ are i.i.d. with mean inter-arrival time $\mathbb{E}[X_i]$. Using the elementary renewal theorem [70, Ch. 5] we have,

$$\lim_{t\to\infty} \frac{\mathbb{E}[N_i(t)]}{t} = \frac{1}{\mathbb{E}[X_i]}. \tag{15}$$

Thus, the rate of gradient pushes by the i-th worker is $1/\mathbb{E}[X_i]$. As there are P workers, we have a superposition of P renewal processes and thus the average rate of gradient pushes to the PS is

$$\lim_{t\to\infty} \sum_{i=1}^{P} \frac{\mathbb{E}[N_i(t)]}{t} = \sum_{i=1}^{P} \frac{1}{\mathbb{E}[X_i]} = \frac{P}{\mathbb{E}[X]}. \tag{16}$$

Every K pushes are one iteration. Thus, the expected runtime per iteration or effectively the expected time for K pushes is given by $\mathbb{E}[T] = \frac{K\mathbb{E}[X]}{P}$. ■

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on May 31,2022 at 14:55:50 UTC from IEEE Xplore. Restrictions apply.

IV. Error Analysis: New Convergence Analysis for Asynchronous SGD

In this section, we discuss our analysis of error convergence with the number of iterations. We first state some assumptions on the objective function in Section IV-A, followed by the main theoretical results on the error analysis including our novel analysis for asynchronous SGD and its variants in Section IV-B.

A. Assumptions on the Objective Function

Closely following [4], we make these assumptions:

1) $F(\mathbf{w})$ is an $L-$ smooth function. Thus,

$$||\nabla F(\mathbf{w}) - \nabla F(\widetilde{\mathbf{w}})||_2 \leq L||\mathbf{w} - \widetilde{\mathbf{w}}||_2 \quad \forall\ \mathbf{w}, \widetilde{\mathbf{w}}. \tag{17}$$

2) $F(\mathbf{w})$ is strongly convex with parameter c. Thus,

$$2c\big(F(\mathbf{w}) - F^*\big) \leq ||\nabla F(\mathbf{w})||_2^2 \quad \forall\ \mathbf{w}. \tag{18}$$

Refer to Appendix A, in the supplementary material for discussion on strong convexity. Our results also extend to non-convex objectives, as discussed in Theorem 4.

3) The stochastic gradient is an unbiased estimate of the true gradient:

$$\mathbb{E}_{\xi_j|\mathbf{w}_k}\big[g(\mathbf{w}_k, \xi_j)\big] = \nabla F(\mathbf{w}_k) \quad \forall\ k \leq j. \tag{19}$$

Observe that this is slightly different from the common assumption that says $\mathbb{E}_{\xi_j}\big[g(\mathbf{w}, \xi_j)\big] = \nabla F(\mathbf{w})$ for all $\mathbf{w}$. Observe that all $\mathbf{w}_j$ for $j > k$ is actually not independent of the data ξ_j. We thus make the assumption more rigorous by conditioning on $\mathbf{w}_k$ for $k \leq j$. Our requirement $k \leq j$ means that $\mathbf{w}_k$ is the value of the parameter at the PS before the data ξ_j was accessed and can thus be assumed to be independent of the data ξ_j.

4) Inspired from [4], we also assume that the variance of the stochastic update given $\mathbf{w}_k$ at iteration k before the data point was accessed is also bounded as follows:

$$\mathbb{E}_{\xi_j|\mathbf{w}_k}\Big[||g\big(\mathbf{w}_k, \xi_j\big) - \nabla F(\mathbf{w}_k)||_2^2\Big] \leq \frac{\sigma^2}{m} + \frac{M_G}{m}||\nabla F(\mathbf{w}_k)||_2^2 \ \forall\ k \leq j. \tag{20}$$

5) In the analysis of K-async and K-batch-async SGD, we replace some assumptions in existing literature that we discuss in Section IV-B, and instead use an alternate staleness bound that allows for large, but rare delays. We assume that for some $\gamma \leq 1$,

$$\mathbb{E}\Big[||\nabla F(\mathbf{w}_j) - \nabla F\big(\mathbf{w}_{\tau(l,j)}\big)||_2^2\Big] \leq \gamma\mathbb{E}\Big[||\nabla F(\mathbf{w}_j)||_2^2\Big].$$

B. Main Theoretical Results

In this work, we provide a novel convergence analysis of K-async SGD for fixed η, relaxing the following assumptions in existing literature.

- In several prior works such as [8], [31], [40], [42], it is often assumed, for the ease of analysis, that runtimes are exponentially distributed. In this paper, we extend our analysis for any general service time X_i.
- In [8], it is also assumed that the staleness process is independent of $\mathbf{w}$. While this assumption simplifies the analysis greatly, it is not true in practice. For instance, for a two worker case, the parameter $\mathbf{w}_2$ after 2 iterations depends on whether the update from $\mathbf{w}_1$ to $\mathbf{w}_2$ was based on a stale gradient at $\mathbf{w}_0$ or the current gradient at $\mathbf{w}_1$, depending on which worker finished first. In this work, we remove this assumption.
- Instead of the bounded delay assumption in [6], we use a general staleness bound

$$\mathbb{E}\Big[||\nabla F(\mathbf{w}_j) - \nabla F(\mathbf{w}_{\tau(l,j)})||_2^2\Big] \leq \gamma\mathbb{E}\Big[||\nabla F(\mathbf{w}_j)||_2^2\Big]$$

which allows for large, but rare delays.
- In [7], the norm of the gradient is assumed to be bounded. However, if we assume that $||\nabla F(\mathbf{w})||_2^2 \leq M$ for some constant M, then using (18) we obtain $||\mathbf{w} - \mathbf{w}^*||_2^2 \leq \frac{2}{c}(F(\mathbf{w}) - F^*) \leq \frac{M}{c^2}$ implying that $\mathbf{w}$ itself is bounded which is a very strong and restrictive assumption, that we relax in this result.

1) Strongly Convex Loss Function:

Theorem 3: Suppose the objective $F(\mathbf{w})$ is c-strongly convex and the learning rate $\eta \leq \frac{1}{2L(\frac{M_G}{Km} + \frac{1}{K})}$. Also assume that for some $\gamma \leq 1$, we have $\mathbb{E}\big[||\nabla F(\mathbf{w}_j) - \nabla F(\mathbf{w}_{\tau(l,j)})||_2^2\big] \leq \gamma\mathbb{E}\big[||\nabla F(\mathbf{w}_j)||_2^2\big]$. Then, the error of K-async SGD after J iterations is,

$$\mathbb{E}[F(\mathbf{w}_J)] - F^* \leq \frac{\eta L\sigma^2}{2c\gamma' Km} + (1 - \eta c\gamma')^J\left(F(\mathbf{w}_0) - F^* - \frac{\eta L\sigma^2}{2c\gamma' Km}\right) \tag{21}$$

where $\gamma' = 1 - \gamma + \frac{p_0}{2}$ and p_0 is a lower bound on the conditional probability that $\tau(l, j) = j$, given all the past delays and parameters.

Here, γ is a measure of staleness of the gradients returned by workers; smaller γ indicates less staleness. The full proof is provided in Appendix C, in the supplementary material. We first prove the result for $K = 1$ in Appendix C-B, in the supplementary material for ease of understanding, and then provide the more general proof for any K in Appendix C-C, in the supplementary material.

Lemma 6 below provides bounds on p_0 for different classes of distributions.

Lemma 6 (Bounds on p_0): Let us denote the conditional probability of $\tau(l, j) = j$ given all the past delays and parameters as $p_0^{(j)}$. Define $p_0 = \inf_j p_0^{(j)}$. Then the following holds.

- For exponential computation times, $p_0^{(j)} = \frac{1}{P}$ for all j (invariant of j) and $p_0 = \frac{1}{P}$.
- For new-longer-than-used (See Definition 3) computation times, $p_0^{(j)} \leq \frac{1}{P}$ and thus $p_0 \leq \frac{1}{P}$.
- For new-shorter-than-used computation times, $p_0^{(j)} \geq \frac{1}{P}$ and thus $p_0 \geq \frac{1}{P}$.

The proof is provided in Appendix C-A, in the supplementary material.

Remark 3: For K-batch-async, the update rule is same as K-async except that the index l denotes the index of the mini-batch. Thus, the error analysis will be similar.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on May 31,2022 at 14:55:50 UTC from IEEE Xplore. Restrictions apply.

Now let us compare with K-sync SGD. We observe that the analysis of K-sync SGD is same as serial SGD with mini-batch size Km. Thus,

Lemma 7 (Error of K-sync [4]): Suppose that the objective $F(\mathbf{w})$ is c-strongly convex and learning rate $\eta \leq \frac{1}{2L(\frac{M_G}{Km}+1)}$. Then, the error after J iterations of K-sync SGD is

$$\mathbb{E}\big[F(\mathbf{w}_J) - F^*\big] \leq \frac{\eta L \sigma^2}{2c(Km)} + (1-\eta c)^J \left(F(\mathbf{w}_0) - F^* - \frac{\eta L \sigma^2}{2c(Km)} \right). \quad (22)$$

Can stale gradients win the race? For the same η, observe that the error given by Theorem 3 decays at the rate $(1 - \eta c(1-\gamma+\frac{p_0}{2}))$ for K-async or K-batch-async SGD while for K-sync, the decay rate with number of iterations is $(1-\eta c)$. Thus, depending on the values of γ and p_0, the decay rate of K-async or K-batch-async SGD can be faster or slower than K-sync SGD. The decay rate of K-async or K-batch-async SGD is faster if $\frac{p_0}{2} > \gamma$. As an example, one might consider an exponential or new-shorter-than-used service time where $p_0 \geq \frac{1}{P}$ and γ can be made smaller by increasing K. It might be noted that asynchronous SGD can still be faster than synchronous SGD with respect to wallclock time even if its decay rate with respect to number of iterations is lower as every iteration is much faster in asynchronous SGD (Roughly $P \log P$ times faster for exponential runtimes).

The maximum allowable learning rate for synchronous SGD is $\max\{\frac{1}{c}, \frac{1}{2L(\frac{M_G}{Pm}+1)}\}$ which can be much higher than that for asynchronous SGD, i.e., $\max\{\frac{1}{c(1-\gamma+\frac{p_0}{2})}, \frac{1}{2L(\frac{M_G}{m}+1)}\}$. Similarly the error floor for synchronous is $\frac{\eta L \sigma^2}{2cPm}$ as compared to asynchronous whose error floor is $\frac{\eta L \sigma^2}{2c(1-\gamma+\frac{p_0}{2})m}$.

2) Extension to Non-Convex Loss Functions: The analysis can be extended to provide weaker guarantees for non-convex objectives. Let $\gamma' = 1 - \gamma + \frac{p_0}{2}$. For non-convex objectives, we have the following result.

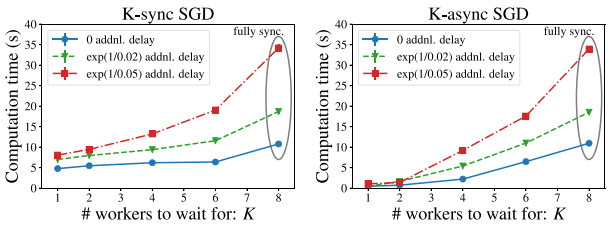
Theorem 4: For non-convex objective function $F(\cdot)$, where $F^* = \min_{\mathbf{w}} F(\mathbf{w})$, we have the following ergodic convergence result for K-async SGD:

$$\frac{1}{J}\sum_{j=0}^{J-1} \mathbb{E}\Big[||\nabla F(\mathbf{w}_j)||_2^2\Big] \leq \frac{2(F(\mathbf{w}_0)-F^*)}{J\eta\gamma'} + \frac{L\eta\sigma^2}{Km\gamma'} \quad (23)$$

The proof is provided in Appendix C-D, in the supplementary material. As before, the same analysis also holds for K-batch-async SGD. For K-sync and K-batch-sync, we can also obtain a similar result, substituting $\gamma' = 1$ in (23) (see [4]). Next, we combine our runtime analysis with the error analysis to characterize the error-runtime trade-off.

V. Experiments and Insights on the Error-Runtime Trade-off

We can combine our expressions for runtime per iteration with the error convergence per iteration to derive the error-runtime trade-off. In Figure 1(b), we compare the theoretical trade-offs between synchronous ($K = P$ in Lemma 7 and Lemma 1) and asynchronous SGD ($K = 1$ in Theorem 3 and Lemma 3) under the strongly-convex assumption. Asynchronous SGD converges very quickly, but to a higher floor. On the other hand, synchronous SGD converges slowly with respect to time, but reaches a much lower error-floor. To validate the trend observed in theory, we conduct experiments on training neural networks to perform image classification on CIFAR 10 dataset. These experiments give insights on how choosing the right K helps us strike the best error-runtime trade-off, depending upon the distribution of the gradient computation delays.



(a) Runtime-versus-K. (b) Runtime-versus-K.

Fig. 8. Runtime per iteration changes along with the parameter K for CIFAR10 dataset: K-async SGD always has lower runtime than K-sync SGD as one might expect from our theoretical analysis.

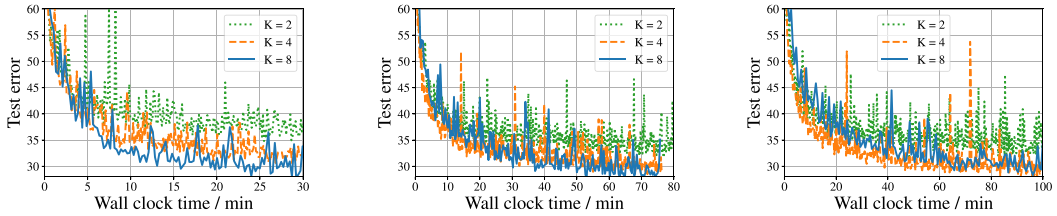
A. Experimental Setting

The algorithms are implemented in Pytorch (v1.0) using multiple nodes. Ray (v0.7) is used for supporting the distributed execution. We use the CIFAR-10 [3] dataset. This dataset consists of $60,000$ 32×32 color images in 10 classes. There are $50,000$ training images and $10,000$ validation images. The neural network used to classify this dataset has two convolutional layers and three fully connected layers. Experiments were conducted on a local cluster with 8 worker machines, each of which has an NVIDIA TitanX GPU. Machines are connected via a 40 Gbps (5000 MB/s) Ethernet interface. Mini-batch size per worker is 32 and learning rate is $\eta = 0.12$.

B. Speedup in Runtime

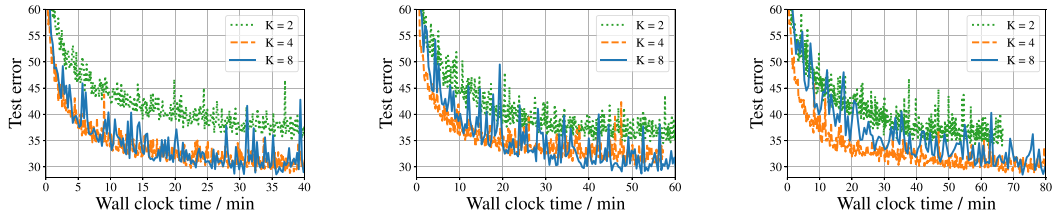
In Figure 8(a) and Figure 8(b) we compare the average runtime per epoch of K-sync and K-async SGD for different values of K. When $K = P = 8$, both the variants become identical to fully synchronous SGD. As we decrease K from $P = 8$ to 1, the computation time drastically reduces since we do not have to wait for straggling nodes. K-async SGD gives a larger delay reduction than K-sync SGD because we do not cancel partially completed gradient computation tasks. Each plot shows three cases: 1) with no artificial delays added to induce straggling, 2) with an additional exponential delay with mean 0.02sec, and 3) with an additional exponential delay with mean 0.05sec. The purpose of these curves is to demonstrate how variability in gradient computation time affects the runtime per iteration. Higher variability means that the system is more susceptible to straggling workers. Thus, as the delay variability increases (as we add a higher mean exponential delay per worker), setting a smaller K gives sharper delay reduction as compared to the $K = 8$ (fully synchronous SGD) case.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on May 31,2022 at 14:55:50 UTC from IEEE Xplore. Restrictions apply.



(a) No artificial delay. (b) Delay with mean 0.02s. (c) Delay with mean 0.05s.

Fig. 9. Test error (%) of **K-sync SGD** on CIFAR-10 with 8 worker nodes. We now demonstrate the error-runtime trade-off for the case with no artificial delay, and then also plot the trade-off as we add an exponential delay on each worker. As the mean of the additional delay increases, using an intermediate value of K achieves a better error-runtime trade-off. The trend of training loss is similar (see Figure 12 in Appendix E, in the supplementary material).



(a) No artificial delay. (b) Delay with mean 0.02s. (c) Delay with mean 0.05s.

Fig. 10. Test error (%) of **K-async SGD** on CIFAR-10 with 8 worker nodes. We demonstrate the error-runtime trade-off for the case with no artificial delay, and then also plot the trade-off as we add an exponential delay on each worker. As the mean of the additional delay increases, using an intermediate value of K achieves a better error-runtime trade-off. The trend of training loss is again similar (see Figure 13 in Appendix E, in the supplementary material).

C. Accuracy-Runtime Trade-off in K-Sync and K-Async

In this subsection, we examine the accuracy-runtime trade-offs for K-sync and K-async SGD variants for CIFAR10 dataset. We plot the test error against wallclock runtime for three cases: (a) No artificial delay; (b) Exponential delay with mean 0.02sec; and (c) Exponential delay with mean 0.05sec. In Figure 9 and Figure 10, we show the test errors for K-sync SGD and K-async SGD. For brevity, we present test error plots here and include the training loss plots in Appendix E, in the supplementary material. The training loss follows the same trend as test error.

As predicted from our theoretical analysis, increasing K improves the final error floor for all the SGD variants. But increasing K also increases the runtime per iteration. Hence, when the error is plotted against wallclock run-time, we begin to observe interesting trends in the error-runtime trade-offs – the highest K does not always achieve the best error-runtime trade-off. An intermediate value of K often achieves a better trade-off than fully-asynchronous or fully-synchronous SGD. For example, in Figure 9a, since there is little delay variability in the gradient computation time, $K = 8$ (fully synchronous SGD) is the best choice of K, but as the delay variability increases in Figure 9b and Figure 9c, $K = 4$ becomes the case that gives the fastest error-versus-wallclock time convergence. A similar trend can be observed in Figure 10 for K-async SGD.

Remark 4: The problem of choosing K is similar in spirit to that of selecting the best mini-batch size in standard synchronous SGD. The main difference is that we consider the error-runtime trade-off instead of the error-iterations trade-off when making the choice.

So far we have considered partially synchronous/asynchronous SGD variants with a fixed value of K. In Section VI we propose the AdaSync strategy that gradually adapts K during the training process in order to get the best error-runtime trade-off.

VI. AdaSync: Adaptive Synchronicity for Improving Error-Runtime Trade-Off

As we observed both theoretically and empirically above, asynchronous SGD converges faster (with respect to wallclock time) but has a higher error floor. On the other hand, synchronous SGD converges slower (with respect to wallclock time) but achieves a lower error floor. In this section, our goal is to try to achieve the best of both worlds, i.e.,, attain the most desirable error-runtime trade-off by gradually varying the level of synchronicity (parameter K) for the different SGD variants.

Let us partition the training time into intervals of time t each, such that, after every slot of time t we vary K. The number of iterations performed within time t is assumed to be approximately $N(t) \approx t/\mathbb{E}[T]$ where $\mathbb{E}[T]$ is the expected runtime per-iteration for the chosen SGD variant.[1] From Theorem 4, we can write a (heuristic) upper bound on the average of $\mathbb{E}\left[||\nabla F(\mathbf{w})||_2^2\right]$ within each time interval t as follows: Upper Bound as a function of: $K = u(K) = \frac{2(F(\mathbf{w}_{start})-F^*)\mathbb{E}[T]}{t\eta\gamma'} + \frac{L\eta\sigma^2}{Km\gamma'}$, where $\mathbf{w}_{start}$ denotes the value of the model $\mathbf{w}$ at the beginning of that time interval.[2] Our goal is to minimize $u(K)$ with respect to K for each time interval.

Observe that,

$$\frac{du(K)}{dK} = \frac{2(F(\mathbf{w}_{start}))}{t\eta\gamma'}\frac{d\mathbb{E}[T]}{dK} - \frac{L\eta\sigma^2}{K^2m\gamma'}. \tag{24}$$

[1]Note that, for exponential inter-arrival times, this approximation holds in the limit of large t.

[2]Note that γ' can be set as 1 for synchronous variants. Though, this does not matter much here as the minimizing K does not depend on it.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on May 31,2022 at 14:55:50 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 AdaSync for K-Async SGD (Can Be Adapted to Other Variants)

1: Start with $K = K_0$ (typically $K_0 = 1$),
2: Iteration counter $j = 0$
3: $\mathbf{w}_{start} = \mathbf{w}_0$ (Initial value of model).
4: **While** Wallclock Time $\leq$ Total Time Budget **do:**
5: Perform an iteration of K-async SGD: $\mathbf{w}_{j+1} = \mathbf{w}_j - \frac{\eta}{K}\sum_{l=1}^{K} g(\mathbf{w}_{\tau(l,j)}, \xi_{l,j})$.
6: Update iteration counter: $j = j + 1$
7: **If** (Wallclock Time % $t = 0$) and ($K < P$) **do:**
8: Update $\mathbf{w}_{start}$ as follows: $\mathbf{w}_{start} = \mathbf{w}_j$ (value of model at the start of this time interval)
9: Update K as follows: $K = K_0\sqrt{\frac{F(\mathbf{w}_0)}{F(\mathbf{w}_{start})}}$

Setting $\frac{du(K)}{dK}$ to 0 therefore provides a rough heuristic on how to choose parameter K for each time interval, as long as, $\frac{d^2u(K)}{dK^2}$ is positive. We derive the rule for adaptively varying K for each of the SGD variants in Appendix D, in the supplementary material. Here, we include the method for one variant K-async to demonstrate the key idea.

For general distributions, the runtime of K-async is upper bounded by that of K-sync (see Lemma 3). For exponential distributions, the two become equal and an algorithm similar to K-sync works. Here, we examine the interesting case of shifted-exponential distribution. We approximate $\mathbb{E}[T] \approx \frac{K\Delta}{P} + \frac{K\log P}{P\mu}$ (see Lemma 4). This leads to

$$K^2 = \frac{L\eta\sigma^2 t\eta P\mu}{2m(F(\mathbf{w}_{start}))(\Delta\mu + \log P)}. \tag{25}$$

To actually solve for this equation, we would need the values of the Lipschitz constant, variance of the gradient etc. which are not always available. We sidestep this issue by proposing a heuristic here that relies on the ratio of the parameter K at different instants.

Observe that, the larger is $F(\mathbf{w}_{start})$, the smaller is the value of K required to minimize $u(K)$. We assume that $F(\mathbf{w}_{start})$ is maximum at the beginning of training, i.e., when $\mathbf{w}_{start} = \mathbf{w}_0$. Hence we start with the smallest initial K, e.g., $K_0 = 1$. Thus, we could start with a small K_0 and after each time interval t, we can update K by solving for

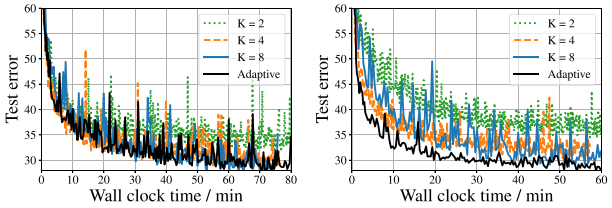
$$K^2 = K_0^2\frac{F(\mathbf{w}_0)}{F(\mathbf{w}_{start})}. \tag{26}$$

We can also verify that the second derivative is positive, i.e., $\frac{d^2u(K)}{dK^2} = \frac{2L\eta\sigma^2}{K^3m} > 0$.

The detailed algorithm for AdaSync for K-async SGD in described in Algorithm 1.

For K-sync SGD under exponential assumption, the update rule for K is derived by solving for K in the quadratic equation: $\frac{K^2}{P-K} = \frac{K_0^2}{P-K_0}\frac{F(\mathbf{w}_0)}{F(\mathbf{w}_{start})}$, as discussed in Appendix D, in the supplementary material. For the two other variants of distributed SGD, the adaptive update rule for K is $K = K_0\sqrt{\frac{F(\mathbf{w}_0)}{F(\mathbf{w}_{start})}}$, under certain assumptions on the runtime distribution, as also discussed in Appendix D, in the supplementary material.

We evaluate the effectiveness of AdaSync for both K-sync SGD and K-async SGD algorithms. An exponential delay with mean 0.02s is added to each worker node independently. We fix K for every $t = 60$ seconds (about 10 epochs). The initial values of K are fine-tuned and set to 2 and 4 for K-sync SGD and K-async SGD, respectively. As shown in Figure 11, the adaptive strategy achieves the fastest convergence in terms of error-versus-time. The adaptive K-async algorithm can even achieve a better error-runtime trade-off than the $K = 8$ case (i.e., fully synchronous case).



(a) Adaptive K-sync. SGD. (b) Adaptive K-async. SGD.

Fig. 11. Test error of **AdaSync SGD** on CIFAR-10 with 8 worker nodes. We add an exponential delay with mean 0.02s on each worker. The value of K is changed after every 60 seconds.

VII. Concluding Remarks

This work introduces a novel analysis of error-runtime trade-off of distributed SGD, accounting for both error reduction per iteration as well as the wallclock runtime in a delay-prone computing environment. Furthermore, we also give a new analysis of asynchronous SGD by relaxing some commonly made assumptions in existing literature. Lastly, we also propose a novel strategy called AdaSync that adaptively increases synchronicity during distributed machine learning to achieve the best error-runtime trade-off. Our results provide valuable insights into distributed machine learning that could inform choice of workers and preferred method of parallelization for a particular distributed SGD algorithm in a chosen distributed computing environment.

As future work, we plan to explore methods of gradually increasing synchrony in other distributed optimization frameworks, e.g., federated learning, decentralized SGD, elastic averaging etc., that is closely related to [9], [10], [71]–[78]. Our proposed techniques can also inform hyperparameter tuning. Given some knowledge of the computing environment, our technique could allow one to simulate the expected error-runtime trade-off in advance, and possibly choose training parameters such as parameter K, mini-batch size m etc. It is also an interesting future direction to extend our current analysis for non iid scenarios, i.e., when the runtime or the dataset of different workers are not independent and identically distributed.

Acknowledgment

The authors thank Soumyadip Ghosh, Parijat Dube, Pulkit Grover, Souptik Sen, Li Zhang, Wei Zhang, and Priya Nagpurkar for helpful discussions.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on May 31,2022 at 14:55:50 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] J. Dean *et al.*, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2012, pp. 1223–1231.

[2] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[3] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Dept. Comput. Sci., Univ. Toronto, Toronto, ON, Canada, Rep. TR-2009, 2009.

[4] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," 2016. [Online]. Available: arXiv:1606.04838.

[5] H. B. Curry, "The method of steepest descent for non-linear minimization problems," *Quart. Appl. Math.*, vol. 2, no. 3, pp. 258–261, 1944.

[6] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous parallel stochastic gradient for nonconvex optimization," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2015, pp. 2737–2745.

[7] B. Recht, C. Re, S. J. Wright, and F. Niu, "HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2011, pp. 693–701.

[8] I. Mitliagkas, C. Zhang, S. Hadjis, and C. Re, "Asynchrony begets momentum, with an application to deep learning," in *Proc. IEEE 54th Annu. Allerton Conf. Commun. Control Comput.*, Monticello, IL, USA, 2016, pp. 997–1004.

[9] J. Wang and G. Joshi, "Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD," 2018. [Online]. Available: arXiv:1810.08313.

[10] J. Wang and G. Joshi, "Cooperative SGD: A unified framework for the design and analysis of communication-efficient SGD algorithms," 2018. [Online]. Available: arXiv:1808.07576.

[11] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge Univ. Press, 2004.

[12] H. Robbins and S. Monro, "A stochastic approximation method," *Ann. Math. Stat.*, vol. 22, no. 3, pp. 400–407, 1951.

[13] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao, "Optimal distributed online prediction using mini-batches," *J. Mach. Learn. Res.*, vol. 13, no. 1, pp. 165–202, 2012.

[14] M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient mini-batch training for stochastic optimization," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, 2014, pp. 661–670.

[15] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 2, pp. 2121–2159, Jul. 2011.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2015, pp. 1–15.

[17] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2013, pp. 315–323.

[18] N. L. Roux, M. Schmidt, and F. R. Bach, "A stochastic gradient method with an exponential convergence rate for finite training sets," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2012, pp. 2663–2671.

[19] L. M. Nguyen, J. Liu, K. Scheinberg, and M. Takáč, "SARAH: A novel method for machine learning problems using stochastic recursive gradient," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 2613–2621.

[20] S. Ruder, "An overview of gradient descent optimization algorithms," 2016. [Online]. Available: arXiv:1609.04747.

[21] S. Gupta, W. Zhang, and F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study," in *Proc. Int. Conf. Data Min.*, 2016, pp. 171–180.

[22] J. Cipar *et al.*, "Solving the straggler problem with bounded staleness," in *Proc. 14th USENIX Conf. Hot Topics Oper. Syst.*, 2013, p. 22.

[23] H. Cui *et al.*, "Exploiting bounded staleness to speed up big data analytics," in *Proc. USENIX Annu. Techn. Conf. (ATC)*, 2014, pp. 37–48.

[24] Q. Ho *et al.*, "More effective distributed ML via a stale synchronous parallel parameter server," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2013, pp. 1223–1231.

[25] J. Tsitsiklis, D. Bertsekas, and M. Athans, "Distributed asynchronous deterministic and stochastic gradient optimization algorithms," *IEEE Trans. Autom. Control*, vol. 31, no. 9, pp. 803–812, Sep. 1986.

[26] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2011, pp. 873–881.

[27] H. Mania, X. Pan, D. Papailiopoulos, B. Recht, K. Ramchandran, and M. I. Jordan, "Perturbed iterate analysis for asynchronous stochastic optimization," *SIAM J. Optim.*, vol. 27, no. 4, pp. 2202–2229, 2017.

[28] S. Chaturapruek, J. C. Duchi, and C. Ré, "Asynchronous stochastic convex optimization: The noise is in the noise and SGD don't care," in *Advances in Neural Information Processing Systems*. Red Hook, NY USA: Curran, 2015, pp. 1531–1539.

[29] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-SGD for distributed deep learning," in *Proc. 25th Int. Joint Conf. Artif. Intell.*, 2016, pp. 2350–2356.

[30] Z. Peng, Y. Xu, M. Yan, and W. Yin, "ARock: An algorithmic framework for asynchronous parallel coordinate updates," *SIAM J. Sci. Comput.*, vol. 38, no. 5, pp. A2851–A2879, 2016.

[31] R. Hannah and W. Yin, "More iterations per second, same quality–why asynchronous algorithms may drastically outperform traditional ones," 2017. [Online]. Available: arXiv:1708.05136.

[32] R. Hannah and W. Yin, "On unbounded delays in asynchronous parallel fixed-point algorithms," *J. Sci. Comput.*, vol. 76, pp. 299–326, Dec. 2017.

[33] T. Sun, R. Hannah, and W. Yin, "Asynchronous coordinate descent under more realistic assumptions," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2017, pp. 6183–6191.

[34] R. Leblond, F. Pedregosa, and S. Lacoste-Julien, "ASAGA: Asynchronous parallel saga," in *Proc. Int. Conf. Artif. Intell. Stat. (AISTATS)*, 2017, pp. 46–54.

[35] A. Harlap *et al.*, "Addressing the straggler problem for iterative convergent parallel ML," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, 2016, pp. 98–111.

[36] G. Joshi, Y. Liu, and E. Soljanin, "On the delay-storage trade-off in content download from coded distributed storage systems," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 5, pp. 989–997, May 2014.

[37] D. Wang, G. Joshi, and G. Wornell, "Using straggler replication to reduce latency in large-scale parallel computing," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 3, pp. 7–11, 2015.

[38] G. Joshi, E. Soljanin, and G. Wornell, "Queues with redundancy: Latency-cost analysis," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 2, pp. 54–56, 2015.

[39] G. Joshi, E. Soljanin, and G. Wornell, "Efficient redundancy techniques for latency reduction in cloud systems," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 2, no. 12, pp. 1–30, May 2017.

[40] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Trans. Inf. Theory*, vol. 64, no. 3, pp. 1514–1529, Mar. 2018.

[41] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 3368–3376.

[42] S. Dutta, V. R. Cadambe, and P. Grover, "Short-dot: Computing large linear transforms distributedly using coded short dot products," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2016, pp. 2100–2108.

[43] W. Halbawi, N. Azizan-Ruhi, F. Salehi, and B. Hassibi, "Improving distributed gradient descent using Reed-Solomon codes," 2017. [Online]. Available: arXiv:1706.05436.

[44] Y. Yang, P. Grover, and S. Kar, "Coded distributed computing for inverse problems," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2017, pp. 709–719.

[45] Y. Yang, P. Grover, and S. Kar, "Fault-tolerant distributed logistic regression using unreliable components," in *Proc. IEEE 54th Annu. Allerton Commun. Control Comput. (Allerton)*, Monticello, IL, USA, 2016, pp. 940–947.

[46] C. Karakus, Y. Sun, and S. Diggavi, "Encoded distributed optimization," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Aachen, Germany, 2017, pp. 2890–2894.

[47] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, "Straggler mitigation in distributed optimization through data encoding," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2017, pp. 5440–5448.

[48] Z. Charles, D. Papailiopoulos, and J. Ellenberg, "Approximate gradient coding via sparse random graphs," 2017. [Online]. Available: arXiv:1711.06771.

[49] S. Li, S. Supittayapornpong, M. A. Maddah-Ali, and S. Avestimehr, "Coded TeraSort," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, Lake Buena Vista, FL, USA, 2017, pp. 389–398.

[50] M. Fahim, H. Jeong, F. Haddadpour, S. Dutta, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," in *Proc. IEEE Commun. Control Comput. (Allerton)*, Monticello, IL, USA, 2017, pp. 1264–1270.

[51] M. Ye and E. Abbe, "Communication-computation efficient gradient coding," 2018. [Online]. Available: arXiv:1802.03475.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on May 31,2022 at 14:55:50 UTC from IEEE Xplore. Restrictions apply.

[52] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr, "A fundamental tradeoff between computation and communication in distributed computing," *IEEE Trans. Inf. Theory*, vol. 64, no. 1, pp. 109–128, Jan. 2018.

[53] V. Cadambe and P. Grover, "Codes for distributed computing: A tutorial," *IEEE Inf. Theory Soc. Newslett.*, vol. 67, no. 4, pp. 3–15, Dec. 2017.

[54] S. Dutta, Z. Bai, H. Jeong, T. M. Low, and P. Grover, "A unified coded deep neural network training strategy based on generalized PolyDot codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Vail, CO, USA, 2018, pp. 1585–1589.

[55] A. Mallick, M. Chaudhari, U. Sheth, G. Palanikumar, and G. Joshi, "Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication," 2018. [Online]. Available: arXiv:1804.10331.

[56] S. Dutta, V. Cadambe, and P. Grover, "Coded convolution for parallel and distributed computing within a deadline," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Aachen, Germany, 2017, pp. 2403–2407.

[57] U. Sheth *et al.*, "An application of storage-optimal matdot codes for coded matrix multiplication: Fast k-nearest neighbors estimation," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Seattle, WA, USA, 2018, pp. 1113–1120.

[58] A. Badita, P. Parag, and V. Aggarwal, "Sequential addition of coded subtasks for straggler mitigation," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Toronto, ON, Canada, 2020, pp. 746–755.

[59] M. F. Aktas, P. Peng, and E. Soljanin, "Straggler mitigation by delayed Relaunch of tasks," *SIGMETRICS Perform. Eval. Rev.*, vol. 45, no. 3, pp. 224–231, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3199524.3199564

[60] S. Wang, J. Liu, and N. Shroff, "Fundamental limits of approximate gradient coding," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 3, pp. 1–22, 2019.

[61] E. Ozfatura, D. Gündüz, and S. Ulukus, "Speeding up distributed gradient descent by utilizing non-persistent stragglers," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Paris, France, 2019, pp. 2729–2733.

[62] H. Al-Lawati, N. Ferdinand, and S. C. Draper, "Anytime minibatch with stale gradients," in *Proc. 53rd Annu. Conf. Inf. Sci. Syst. (CISS)*, Baltimore, MD, USA, 2019, pp. 1–5.

[63] R. K. Maity, A. S. Rawa, and A. Mazumdar, "Robust gradient descent via moment encoding and LDPC codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Paris, France, 2019, pp. 2734–2738.

[64] A. Reisizadeh, H. Taheri, A. Mokhtari, H. Hassani, and R. Pedarsani, "Robust and communication-efficient collaborative learning," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, Red Hook, NY, USA, 2019, pp. 8386–8397.

[65] M. M. Amiri and D. Gündüz, "Computation scheduling for distributed machine learning with straggling workers," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, Brighton, U.K., 2019, pp. 8177–8181.

[66] F. Haddadpour, M. M. Kamani, M. Mahdavi, and V. Cadambe, "Trading redundancy for communication: Speeding up distributed SGD for non-convex optimization," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 2545–2554.

[67] J. Regatti, G. Tendolkar, Y. Zhou, A. Gupta, and Y. Liang, "Distributed SGD generalizes well under asynchrony," in *Proc. 57th Annu. Allerton Conf. Commun. Control Comput. (Allerton)*, Monticello, IL, USA, 2019, pp. 863–870.

[68] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," 2016. [Online]. Available: http://arxiv.org/abs/1604.00981.

[69] S. M. Ross, *A First Course in Probability*. Boston, MA, USA: Pearson, 2019.

[70] R. G. Gallager, *Stochastic Processes: Theory for Applications*, 1st ed. Cambridge, U.K.: Cambridge Univ. Press, 2013.

[71] J. Zhang, C. De Sa, I. Mitliagkas, and C. Ré, "Parallel SGD: When does averaging help?" 2016. [Online]. Available: arXiv:1606.07365.

[72] D. Yin, A. Pananjady, M. Lam, D. Papailiopoulos, K. Ramchandran, and P. Bartlett, "Gradient diversity: A key ingredient for scalable distributed learning," 2017. [Online]. Available: arXiv:1706.05699.

[73] F. Zhou and G. Cong, "On the convergence properties of a K-step averaging stochastic gradient descent algorithm for nonconvex optimization," 2017. [Online]. Available: arXiv:1708.01012.

[74] S. Zhang, A. E. Choromanska, and Y. LeCun, "Deep learning with elastic averaging SGD," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2015, pp. 685–693.

[75] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Y. Arcas, "Communication-efficient learning of deep networks from decentralized data," 2016. [Online]. Available: arXiv:1602.05629.

[76] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," 2016. [Online]. Available: arXiv:1610.05492.

[77] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," 2019. [Online]. Available: arXiv:1908.07873.

[78] J. Wang, A. K. Sahu, Z. Yang, G. Joshi, and S. Kar, "MATCHA: Speeding up decentralized SGD via matching decomposition sampling," 2019. [Online]. Available: arXiv:1905.09435.

[79] D. M. Kreps, *A Course in Microeconomic Theory*, Princeton, NJ, USA: Princeton Univ. Press, vol. 41, 1990.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on May 31,2022 at 14:55:50 UTC from IEEE Xplore. Restrictions apply.