DSAG: A Mixed Synchronous-Asynchronous Iterative Method for Straggler-Resilient Learning

Albin Severinson, Eirik Rosnes, *Senior Member, IEEE*, Salim El Rouayheb, *Senior Member, IEEE*, and Alexandre Graell i Amat, *Senior Member, IEEE*

***Abstract*— We consider straggler-resilient learning. In many previous works, e.g., in the coded computing literature, straggling is modeled as random delays that are independent and identically distributed between workers. However, in many practical scenarios, a given worker may straggle over an extended period of time. We propose a latency model that captures this behavior and is substantiated by traces collected on Microsoft Azure, Amazon Web Services (AWS), and a small local cluster. Building on this model, we propose DSAG, a mixed synchronous-asynchronous iterative optimization method, based on the stochastic average gradient (SAG) method, that combines timely and stale results. We also propose a dynamic load-balancing strategy to further reduce the impact of straggling workers. We evaluate DSAG for principal component analysis, cast as a finite-sum optimization problem, of a large genomics dataset, and for logistic regression on a cluster composed of 100 workers on AWS, and find that DSAG is up to about 50% faster than SAG, and more than twice as fast as coded computing methods, for the particular scenario that we consider.**

***Index Terms*— Coded computing, iterative optimization, load-balancing, principal component analysis (PCA), stochastic average gradient (SAG), straggler mitigation, variance reduction.**

I. Introduction

We are interested in reducing the latency of distributed iterative optimization methods for empirical risk minimization. In particular, we want to reduce the impact of straggling workers, i.e., workers experiencing delays, which can significantly slow down distributed algorithms. The straggler problem is a consequence of the design of modern large-scale compute clusters (sometimes referred to as *warehouse-scale computers*), which are built from a large number of commodity servers connected in a heterogeneous manner, and where many virtual machines may share the same physical host server, to maximize cost-efficiency [1], [2]. Examples include Microsoft Azure, Google Cloud, and Amazon Web Services (AWS).

Straggling is often assumed, e.g., in the coded computing literature [3], [4], [5], [6], to be caused by random delays that are independent and identically distributed (i.i.d.) between workers and iterations. However, from traces collected on Microsoft Azure and AWS, we find that stragglers tend to remain stragglers. Similar behavior is considered in [7] and [8], where latency probability distributions may differ between workers, and [9], where latency distributions may also change over time. As a result, data processed by stragglers may never factor in for stochastic methods that only rely on the results from the fastest subset of workers.

This work consists of three parts. First, we propose a latency model that accounts for differences in the mean and variance of the latency between different workers and over time. This model can be seen as an extension of the model considered in [9]. Further, for the proposed model we show how to efficiently estimate the latency of the w-th fastest worker out of a set of N workers, including for iterative computations, where a worker may remain unavailable over several subsequent iterations.

Second, based on this model, we propose DSAG, an iterative method for finite-sum optimization (machine learning problems are typically cast as finite-sum optimization problems) which adapts the stochastic average gradient (SAG) method [10] to distributed environments. The key idea of DSAG is to wait for the w fastest workers in each iteration—i.e., DSAG is a stochastic method—while simultaneously integrating stale results received from the $N - w$ stragglers as they are received over subsequent iterations. DSAG relies on the *variance reduction* technique of SAG to suppress the potentially high variance caused by this strategy and improve convergence. Finally, we propose a dynamic load-balancing strategy for reducing the variation in latency between workers, that is based on the model proposed in part one.

We validate the proposed model on Azure, AWS, and a small local cluster, and find that the model accurately

Manuscript received 12 June 2022; revised 3 October 2022; accepted 20 November 2022. Date of publication 7 December 2022; date of current version 16 February 2023. The work of Salim El Rouayheb was supported in part by the NSF under grant CNS-1801630. The work of Alexandre Graell i Amat was supported by the Swedish Research Council under grant 2020-03687. Also, the research presented in this article has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053 (https://www.ex3.simula.no/). The associate editor coordinating the review of this article and approving it for publication was J. Zhang. *(Corresponding author: Eirik Rosnes.)*

Albin Severinson was with Simula UiB, 5006 Bergen, Norway, and also with the Department of Informatics, University of Bergen, 5006 Bergen, Norway. He is now with G-Research, WC1E 7EA London, U.K. (e-mail: albin@severinson.org).

Eirik Rosnes is with Simula UiB, 5006 Bergen, Norway (e-mail: eirikrosnes@simula.no).

Salim El Rouayheb is with the Department of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ 08854 USA (e-mail: salim.elrouayheb@rutgers.edu).

Alexandre Graell i Amat is with the Department of Electrical Engineering, Chalmers University of Technology, 41296 Gothenburg, Sweden, and also with Simula UiB, 5006 Bergen, Norway (e-mail: alexandre.graell@chalmers.se).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCOMM.2022.3227286.

Digital Object Identifier 10.1109/TCOMM.2022.3227286

predicts latency across the three platforms. We evaluate the performance of DSAG by using it for principal component analysis (PCA), cast as an optimization problem, of a large genomics dataset, and for logistic regression. For both PCA and logistic regression, DSAG with load-balancing reduces latency significantly compared to SAG—for a scenario with 100 workers on AWS, DSAG is about 10% faster than SAG for PCA and up to 50% faster for logistic regression. Furthermore, it is more than twice as fast as coded computing methods.

We provide the source code of our implementation and the latency traces we have collected under a permissive license at [11].

A. Related Work

Recently, *coded computing* has been proposed to deal with stragglers [3]. The key idea is to add redundant computations (thus increasing the per-worker computational load), such that the result of the computation can be recovered from a subset of the workers, typically via a decoding operation. Coded computing methods have been proposed for, e.g., matrix-vector multiplication [3], [7], [8], [12], [13], matrix-matrix multiplication [14], [15], [16], [17], [18], [19], [20], [21], polynomial evaluation [22], and gradient computations [4], [9], [23], [24]. For example, the method in [4] increases the computational load per worker by a factor $(N-w)+1$ compared to gradient descent (GD) to tolerate any $N-w$ stragglers.

Another method to deal with stragglers is stochastic optimization, the simplest form of which is to ignore stragglers for GD. This is a stochastic gradient descent (SGD) method, sometimes referred to as ignoring stragglers SGD. SGD does not converge to the optimum unless the stepsize is reduced as the algorithm progresses. However, a smaller stepsize reduces the rate of convergence, and it is difficult to determine the correct rate at which to reduce the stepsize. Approximate coded computing methods combine ignoring stragglers SGD with redundancy, e.g., [5], [25], [26]. These methods improve the rate of convergence per iteration compared to ignoring stragglers SGD but typically do not converge to the optimum, and typically increase the computational load compared to GD by a factor 2 or 3.

The above methods treat iterations independently, ignoring the correlation between the results computed in subsequent iterations, which is often significant. The coded version of the power method proposed in [6] is an exception in that the previous iterate is used as side information during decoding. The process is related to *sketch-and-project* methods (see, e.g., [27], [28], [29]), i.e., iterative methods to approximate some quantity from low-rank sketches. In particular, the method in [6] can be seen as a special case of the one in [28]. A significant shortcoming of the method in [6] is that it requires a complex decoding process to be performed by the coordinator for each iteration.

The method in [28] is a variance-reduced stochastic method for first-order optimization. For each iteration, these methods use an estimate of the gradient to, e.g., perform a gradient step, i.e., they are stochastic. Variance-reduced methods converge to the optimum despite being stochastic by using information contained in previous iterates and/or gradients to ensure that the variance of this estimate tends to zero as the method progresses. Examples of variance-reduced methods include SAG [10], SAGA [30] (including a peer-to-peer version [31]), SARAH [32], SVRG [33], SEGA [28], and MARINA [34]. These works do not consider the straggler problem.

Exploiting stale gradients in combination with asynchronicity to alleviate the straggler problem has been explored in several previous works, see, e.g., [35], [36], and references therein, in the neighboring area of federated learning, and [16]. These methods are similar to the proposed DSAG, but do not employ variance reduction. For example, a mixed synchronous-asynchronous distributed version of SGD that is similar to ours has been proposed and analyzed in [16]. Like the method we propose, the method in [16] uses asynchronicity to reduce iteration latency. However, unlike our method, the method in [16] gradually increases the level of synchronicity, thus increasing iteration latency, to improve convergence, whereas our method relies on variance reduction. There has also been a significant amount of work on asynchronous optimization for shared-memory systems, e.g., [37], [38], and references therein. However, these works do not consider the straggler problem.

The load-balancing approach we suggest is designed specifically for DSAG, but is inspired by the large number of previous works on the topic; see, e.g., [39], [40], [41], [42], and references therein. These suggest approaches to balance either i) the complexity of the subtasks that make up a particular large computation (e.g., [39], [40]), or ii) incoming requests between instances of a distributed application, such as a web server (e.g., [41], [42]). The approach we suggest, like those of [41] and [42], but unlike [39] and [40], accounts for latency differences between servers and over time—as is the case in the cloud—but balances the complexity of subtasks, as in [39] and [40]. Furthermore, DSAG is designed with load-balancing in mind, and, as a result, unlike the approach of [39] and [40], does not require moving data between servers to perform load-balancing.

II. Preliminaries

Denote by $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ a data matrix, where $\mathbb{R}$ denotes the real numbers and n is the number of samples and d the dimension. Many learning problems (e.g., linear and logistic regression, PCA, matrix factorization, and training neural networks) can be cast as a finite-sum optimization problem of the form

$$\boldsymbol{V}^* = \underset{\boldsymbol{V} \in \mathcal{L}}{\arg\min} \left[F(\boldsymbol{V}, \boldsymbol{X}) \triangleq R(\boldsymbol{V}) + \sum_{i=1}^{n} f_i(\boldsymbol{V}, \boldsymbol{x}_i) \right], \quad (1)$$

where $\mathcal{L}$ is the solution space, f_i is the loss function with respect to the i-th sample (row of $\boldsymbol{X}$), which we denote by $\boldsymbol{x}_i$, and R is a regularizer, which serves to, e.g., bias $\boldsymbol{V}^*$ toward sparse solutions. For the remainder of this paper, we write $F(\boldsymbol{V})$ and $f_i(\boldsymbol{V})$, leaving the dependence of F and f_i on $\boldsymbol{X}$ and $\boldsymbol{x}_i$, respectively, implicit.

These problems are often solved (e.g., for the examples mentioned above) using so-called first-order iterative optimization methods, i.e., methods that iteratively update a solution based on the gradient of the loss function F, which we denote

by ∇F. One example of such a method is GD, the update rule of which is

$$\boldsymbol{V}^{(t+1)} = G\left(\boldsymbol{V}^{(t)} - \eta \nabla F\left(\boldsymbol{V}^{(t)}\right)\right), \tag{2}$$

where t is the iteration index, η the stepsize, and G a projection operator (possibly the identity operator).

In this work, we consider a distributed scenario in which the rows of $\boldsymbol{X}$ are stored over N worker nodes, such that each node stores an equal fraction of the rows. The workers are responsible for computing the subgradients $\nabla f_1, \ldots, \nabla f_n$ and the coordinator is responsible for aggregating those subgradients and performing a gradient step.

A. Variance-Reduced Optimization

Our work is based on SAG [10], which is a stochastic variance-reduced optimization method for solving problems of the form given in (1). Variance-reduced methods perform updates of the form[1]

$$\boldsymbol{V}^{(t+1)} = G\left(\boldsymbol{V}^{(t)} - \eta \nabla \hat{F}\left(\boldsymbol{V}^{(t)}\right)\right),$$

where $\nabla \hat{F}$ is an estimate of ∇F with the property that

$$\nabla \hat{F}\left(\boldsymbol{V}^{(t)}\right) \longrightarrow \nabla F\left(\boldsymbol{V}^{*}\right) \quad \text{as} \quad t \to \infty,$$

i.e., the gradient estimate tends to the gradient at the optimum as the method progresses—the variance of $\nabla \hat{F}$ is reduced. As a result, despite relying on an estimate of the gradient, variance-reduced methods converge to the optimum. For example, SGD with gradually decreasing step size is a variance-reduce method [43]. However, methods with constant step size can achieve a higher rate of convergence. Such methods instead rely on information contained in previous iterates and/or gradients. Examples include SAG [10], SAGA [30], SARAH [32], SVRG [33], SEGA [28], and MARINA [34].

Consider the problem of estimating the sum

$$\nabla f\left(\boldsymbol{V}^{(t)}\right) \triangleq \frac{1}{n}\sum_{i=1}^{n} \nabla f_i\left(\boldsymbol{V}^{(t)}\right)$$

under the constraint of only computing one of the terms $\nabla f_1\left(\boldsymbol{V}^{(t)}\right), \ldots, \nabla f_n\left(\boldsymbol{V}^{(t)}\right)$ at a time. Given such an estimate, the overall gradient $\nabla F\left(\boldsymbol{V}^{(t)}\right)$ is then easily estimated, since we assume that $\nabla R\left(\boldsymbol{V}^{(t)}\right)$ is easy to compute. Further, after computing any of these terms, the iterate is updated, which causes the gradients to change in some unknown way—$\nabla f_i\left(\boldsymbol{V}^{(t+1)}\right) \neq \nabla f_i\left(\boldsymbol{V}^{(t)}\right)$ in general. SAG [10] and SAGA [30] use the following estimate:[2]

$$\nabla \hat{f}_{\alpha}^{(t)} \triangleq \alpha\left[\nabla f_{i^{(t)}}\left(\boldsymbol{V}^{(t)}\right) - \boldsymbol{Z}_{i^{(t)}}^{(t-1)}\right] + \frac{1}{n}\sum_{j=1}^{n} \boldsymbol{Z}_j^{(t-1)},$$

where α is a scalar parameter, $i^{(t)}$ is the index of the term computed in the t-th iteration, and $\boldsymbol{Z}_1^{(t)}, \ldots, \boldsymbol{Z}_n^{(t)}$ is a table of previously computed gradients, recursively obtained as

$$\boldsymbol{Z}_j^{(t)} = \begin{cases} \nabla f_{i^{(t)}}\left(\boldsymbol{V}^{(t)}\right) & \text{if } j = i^{(t)} \\ \boldsymbol{Z}_j^{(t-1)} & \text{otherwise.} \end{cases}$$

[1]Update rules other than GD, e.g., accelerated GD, may also be used.

[2]This explanation is due to [30].

The expectation and variance of $\nabla \hat{f}_{\alpha}^{(t)}$ are

$$\mathbb{E}\left[\nabla \hat{f}_{\alpha}^{(t)}\right] = \alpha\, \mathbb{E}\left[\nabla f\left(\boldsymbol{V}^{(t)}\right)\right] + (1-\alpha)\, \mathbb{E}\left[\frac{1}{n}\sum_{j=1}^{n} \boldsymbol{Z}_j^{(t-1)}\right]$$

and

$$\begin{aligned} &\operatorname{var}\left(\nabla \hat{f}_{\alpha}^{(t)}\right) \\ &\quad = \alpha^2\left[\operatorname{var}\left(\nabla f\left(\boldsymbol{V}^{(t)}\right)\right) + \operatorname{var}\left(\frac{1}{n}\sum_{j=1}^{n} \boldsymbol{Z}_j^{(t-1)}\right)\right. \\ &\qquad \left. - 2\operatorname{cov}\left(\nabla f\left(\boldsymbol{V}^{(t)}\right), \frac{1}{n}\sum_{j=1}^{n} \boldsymbol{Z}_j^{(t-1)}\right)\right], \end{aligned}$$

respectively, where $\operatorname{var}(\cdot)$ and $\operatorname{cov}(\cdot,\cdot)$ denote the variance and covariance of its argument(s), respectively. Hence, the variance is reduced if the covariance between $\nabla f\left(\boldsymbol{V}^{(t)}\right)$ and the average of the table entries is large enough, which is true if the gradients do not change too much between iterations (as a result these methods only converge if the step size is less than some limit). Note that by choosing $\alpha = 1$ the estimate is unbiased. In fact, SAGA uses exactly this estimate, i.e., $\nabla \hat{f}_{\mathsf{SAGA}}^{(t)} = \nabla \hat{f}_{1}^{(t)}$. However, by choosing a smaller α, we can further reduce the variance of the estimate at the expense of it becoming biased. Indeed, SAG uses $\alpha = 1/n$, i.e., $\nabla \hat{f}_{\mathsf{SAG}}^{(t)} = \nabla \hat{f}_{1/n}^{(t)}$—in this case the estimate is equal to the sum of the table entries.

We base our work on SAG since it combines low variance with a lookup-table based estimation strategy that is well-suited to dealing with stragglers. More recent variance-reduced algorithms, e.g., MARINA, that use an inner-outer loop strategy, achieve variance reduction by periodically processing a large fraction of the dataset. These strategies are more difficult to make straggler-resilient, since the outer loop then requires waiting for more workers to respond.

B. Experimental Setup

The results presented in this work are from experiments conducted on compute clusters hosted on Microsoft Azure (region `West Europe`), AWS (region `eu-north-1`), and the eX3 cluster.[3] For Azure, the nodes are of type `F2s_v2` and for AWS the nodes are of type `c5.xlarge`.[4] For AWS, we also provide traces for nodes of type `c5.xlarge` in region `us-east-1` and of type `t3.xlarge` in region `eu-north-1` [11]. The nodes used on eX3 are equipped with AMD EPYC 7302P processors and high-speed InfiniBand interconnects. We use the same type of node for the coordinator and the workers. Our implementation is written in the Julia programming language, and we use OpenMPI for communication—specifically, the `Isend` and `Irecv` nonblocking, point-to-point communication subroutines. For Azure and AWS, we use the CycleCloud and ParallelCluster systems, respectively, to create workers on-demand.

[3]See ex3.simula.no.

[4]Both `F2s_v2` and `c5.xlarge` nodes are based on Intel Xeon Platinum 8000 series processors. `F2s_v2` nodes have an expected network speed of 875 Mbps, whereas `c5.xlarge` nodes have a network speed of up to 10 Gpbs.

Throughout this paper, we consider a data matrix derived from the 1000 Genomes phase-3 dataset [44]. More precisely, we consider a binary representation of the data for each chromosome, where a nonzero entry in the (i, j)-th position indicates that the genome of the i-th subject differs from that of the reference genome in the j-th position. The matrix we use is the concatenation of such matrices computed for each chromosome. It is a sparse matrix of size 81271767×2504 with density about 5.360%. In Section VII, we also consider the HIGGS dataset, which consists of 11000000 samples with 28 features [45]. For all computations, each worker stores the subset of the dataset assigned to it in memory throughout the computation.

III. Modeling the Latency of Gradient Computations

In this section, we propose a model of the communication and computation latency of workers performing gradient computations in a distributed setting. Later, we use this model to predict the latency of the w-th fastest worker out of a set of workers. We first consider the latency of workers operating in *steady state* (Section III-A), after which we consider how the latency of a particular worker changes over time (Section III-B).

The model we propose is based on latency traces collected in clusters composed of up to 108 workers on AWS, Azure, and eX3, with varying per-worker computational load, which we denote by c, and b bytes communicated per iteration. Here, the computational load can be any quantity that captures the amount of work performed by each worker and iteration, such that a change in c results in a proportional change in the expected computation latency of a single worker. The number of bytes communicated and the computational load are equal for all workers, and we repeat the experiment at different days and times of the day.

In particular, for each worker, we record the latency associated with sending to the worker an iterate $\boldsymbol{V}$ and for the worker to respond with the result of the computation,

$$\boldsymbol{X}_{i:j}^{\mathsf{T}} \boldsymbol{X}_{i:j} \boldsymbol{V}, \tag{3}$$

for some integers $1 \leq i \leq j \leq n$, where $\boldsymbol{X}_{i:j}$ denotes the submatrix of $\boldsymbol{X}$ consisting of rows i through j. Hence, our results generalize to computations that rely on matrix multiplication, although the model is also easily adapted to other types of computations. In addition, we make available traces recorded for other computations and datasets, and we find consistent behavior across the computations and datasets considered [11].

We let c be the number of operations required to perform this computation, i.e., $c = 2\zeta dk(j-i+1)$, where d is the sample dimension, k is the number of columns of $\boldsymbol{V}$, and ζ is the density of the data matrix. For all recordings, we randomly permute the rows of the matrix to break up dense blocks, and we adjust the computational load by tuning the number of samples processed.

In Fig. 1, we plot the range of computational loads considered, together with the mean and variance of the computation latency recorded for 100 different workers for each computational load, when the number of bytes communicated per iteration is $b = 30048$. For reference, we also plot a line passing through the origin fitted to the data.

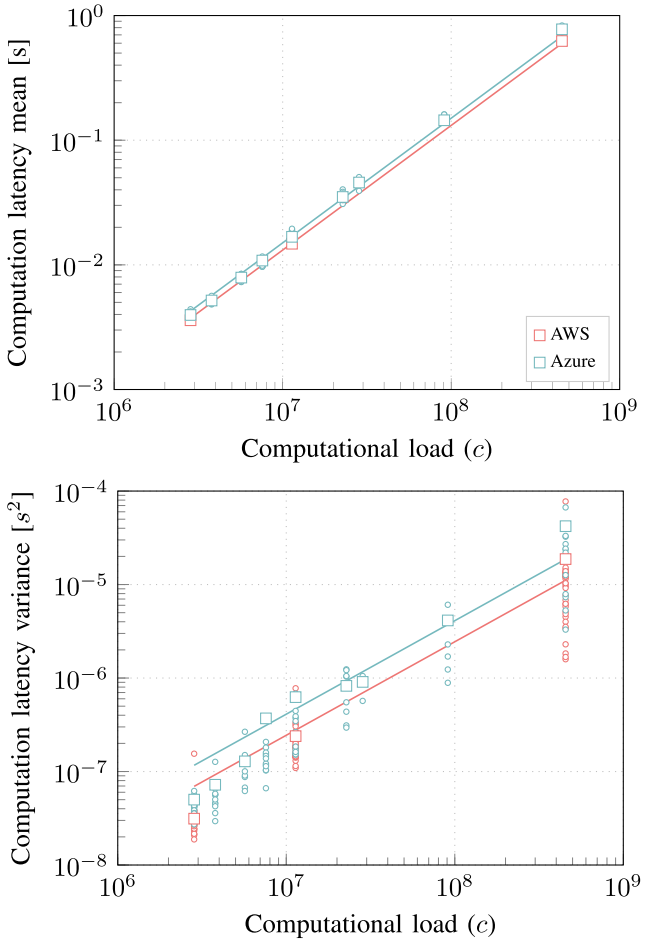


Fig. 1. Mean and variance of the computation latency recorded for 100 different workers as a function of the computational load. Circles correspond to workers, and the mean over all recordings for each computational load is marked by a square. For reference, we also show a line passing through the origin fitted to the data.

A. Steady-State Latency

We find that the latency distribution of workers may change significantly over time, but that these changes typically occur quickly and that the distribution remains approximately constant between changes. Here, we characterize the latency of individual workers while in steady state, i.e., for periods where the latency distribution remains approximately constant. Our results are based on traces collected from running many iterations of (3) in sequence over a set of workers. For each iteration, we wait for all workers to return their result before proceeding to the next iteration. For this section, we have deliberately chosen traces for which the latency distribution does not change significantly throughout the computation.

In Fig. 2, we plot the communication latency (circles) and computation latency (triangles) recorded for two workers over 100 iterations (out of a total of 1600) on Azure. Note that the average latency differs between the two workers; worker 2 is about 14% slower. We show the associated cumulative

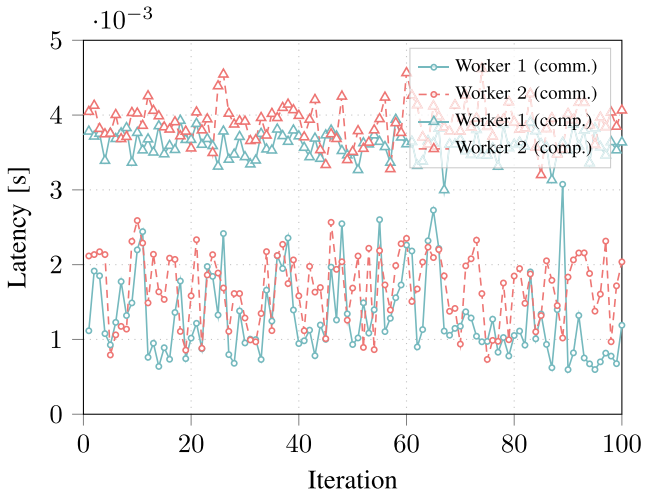


Fig. 2. Per-iteration latency of two workers on Azure, with $b = 30048$ bytes communicated per iteration (circles) and computational load $c = 2.841 \cdot 10^6$ (triangles).

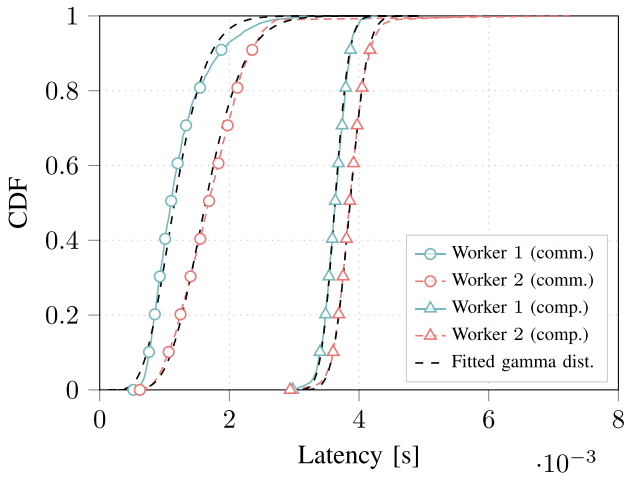


Fig. 3. Empirical CDF of the per-iteration latency of the two workers in Fig. 2. Worker 2 is, on average, 14% slower than worker 1. Black dashed lines indicate fitted gamma distributions.

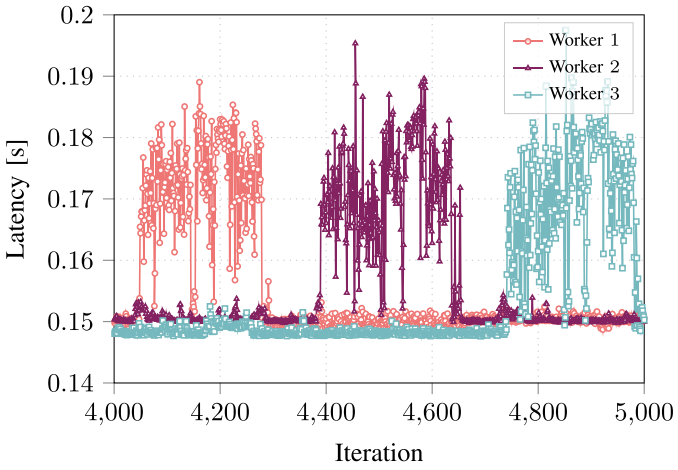


Fig. 4. Per-iteration computation latency of 3 workers (out of $N = 36$) on AWS, with computational load $c = 7.566 \cdot 10^7$. Workers typically experience bursts of high latency.

distribution functions (CDFs) in Fig. 3. Now, for a set of workers, we model the latency of the i-th worker by the random variable

$$X_i^{(b,c)} = Y_i^{(b)} + Z_i^{(c)},$$

where $Y_i^{(b)}$ and $Z_i^{(c)}$ are random variables associated with the communication and computation latency, respectively, of the worker, when the number of bytes communicated is b and the computational load is c.[5] We often omit the superscripts b and c.

We find that the communication and computation latency of workers on Azure and AWS is well-approximated by independent gamma-distributed random variables,[6] but that the parameters of these distributions typically differ between workers, i.e., probability distributions have to be fitted to the particular set of workers used for each computation, especially for systems like Azure CycleCloud and AWS ParallelCluster, which create new worker instances on-demand at the start of a computation. Failing to account for these differences can significantly reduce the accuracy of predictions made using the model; see Section IV-A and Fig. 5.

B. Variability Over Time

The latency distribution of any particular worker typically changes over time. In particular, as a consequence of the design of cloud computing systems, where multiple virtual machines share the same physical host machine, workers experience *bursts* of higher latency. For example, performing memory-intensive operations, such as matrix multiplication, can more than halve the bandwidth available to other threads on the same machine [46].[7] Further, computations managed by cluster schedulers, such as Borg or Kubernetes, are often only guaranteed a very low fraction of the CPU cycles of the server it is assigned to, but may opportunistically use any cycles not used by other computations [47], [48, Ch. 14.3], potentially resulting in large performance fluctuations.

In Fig. 4, we show an example of such high-latency bursts, with the average latency of each of 3 workers out of the $N = 36$ workers used for a particular computation on AWS increasing by about 12% for about one minute.[8] The entire computation lasts for about 30 minutes, and most of the 36 workers experience at least one such burst over this time. Further, at least one worker is currently experiencing a burst of high latency for about 40% of the iterations. This problem becomes more severe for a larger number of workers—for computations consisting of hundreds of workers, the probability that no worker is currently experiencing a latency burst is close to zero.

[5]The model proposed in [22], where the latency of each worker is assumed to take on one of two discrete values, is similar to ours in the sense that latency may differ between workers. However, for our model, latency takes on values according to a continuous probability distribution.

[6]In several previous works, latency is modeled by shifted exponential-distributed random variables. These models are related, since the sum of several exponential random variables is gamma-distributed. Hence, a possible interpretation is that the latency we record is the sum of the latency of several smaller computations, each of which has exponentially distributed latency.

[7]This is known as the *noisy neighbor* problem.

[8]Similar behavior was observed on AWS in [41] and [42].

IV. Predicting the Latency of Distributed Gradient Computations

Here, we show how to efficiently estimate the latency of the w-th fastest worker ($w \leq N$) of a set of workers, i.e., the w-th *order statistic* of the per-worker latency. Later, we use these predictions for dynamic load-balancing to minimize latency variations between workers (see Section VI). Throughout this section, we have deliberately chosen traces where workers are operating in steady state. When used for load-balancing, we account for bursts by dynamically updating the estimate of the latency distribution associated with each worker. We first consider the case where all workers are available at the start of each iteration (Section IV-A), before considering the more realistic case where workers may remain unavailable over several iterations (Section IV-B).

A. Order Statistics Latency

From the distributions of Y_i and Z_i for each worker, we can compute the distribution of the latency of the w-th fastest worker. However, the computational complexity of doing so analytically may be prohibitively high when the number of workers is large. Instead, we use Monte Carlo integration. The complexity of sampling from the latency of the w-th fastest worker is linear in the number of workers, since we can first sample from Y_i and Z_i for each worker and then find the w-th smallest value of the resulting list in linear time, e.g., using the Quickselect algorithm. Through this process we can estimate, e.g., the expected latency of the w-th fastest worker.

In Fig. 5, we plot the average latency of the w-th fastest worker out of $N = 72$ workers for a particular computation with $b = 30048$ and $c = 4.545 \cdot 10^8$ on Azure. We also plot predictions made using Monte Carlo integration as explained above, and, for reference, predictions made by the commonly adopted i.i.d. model, where the latency of each worker is modeled by a random variable with mean and variance equal to the global mean and variance computed across all workers.[9] The proposed model yields an accurate prediction of the empirical performance, while assuming that latency is i.i.d. between workers can significantly reduce accuracy.

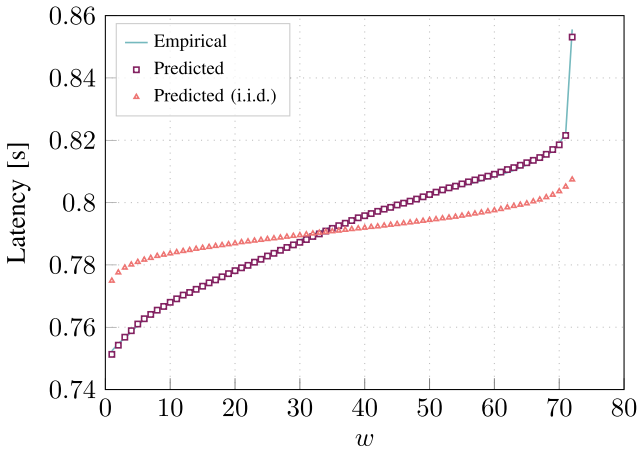


Fig. 5. Average latency of the w-th fastest worker out of $N = 72$ workers for a computation on Azure, and predicted latency, where the per-worker latency is modeled as either independent, but not necessarily identically distributed, or i.i.d., between workers. The i.i.d. assumption can significantly reduce accuracy.

B. Order Statistics Latency of Iterative Computations

In Section VI-A, we considered order statistics in cases where all workers are available at the start of each iteration. However, for straggler-resilient methods, we wish to proceed to the next iteration immediately after receiving results from the w fastest workers, without waiting for the remaining $N - w$ workers, which may remain unavailable over several subsequent iterations.

Here, we show how to estimate latency in this scenario. Denote by $T_w^{(t)}$ the time at which the t-th iteration of an iterative computation, for which the coordinator waits for the w-th fastest worker in each iteration, is completed (i.e., the latency of the t-th iteration is $T_w^{(t)} - T_w^{(t-1)}$). We wish to simulate the time series process $T_w^{(1)}, \ldots, T_w^{(\ell)}$, where ℓ is the number of iterations. We do so by using a two-state model, where workers are either idle or busy. First, each worker has a local first-in-last-out task queue of length 1. If the i-th worker is idle and there is a task in its queue, it immediately removes the task from the queue and becomes busy for a random amount of time, which is captured by the random variable X_i (recall that we can sample from X_i, see Section III-A), before becoming idle again. At the start of each iteration, the coordinator assigns a task to each worker, and once w of those tasks have been completed, the coordinator proceeds to the next iteration.

Using this model, we can efficiently simulate realizations of $T_w^{(1)}, \ldots, T_w^{(\ell)}$ by using a priority queue data structure (see, e.g., [49]) to map the index of each worker to the next time at which it will transition from busy to idle. This strategy is typically referred to as *event-driven* simulation. By performing such simulations we can estimate, e.g., the expected time required to perform ℓ iterations, in a manner that accounts for the fact that workers may remain unavailable over several iterations. We provide an implementation of such a simulator in [11].

In Fig. 6, we plot the cumulative latency over 100 iterations for two jobs, with $b = 30048$, $c = 7.575 \cdot 10^6$, and $N = 72$ on AWS, where, in one job, we wait for $w = 9$ workers (blue curves) and, in the other, for all $w = N = 72$ workers (red curves). We also plot the predictions made by the proposed model based on event-driven simulations, which accounts for the interaction between iterations, and the model described in Section IV-A, which does not. For $w = N = 72$, both models give accurate predictions. However, for $w = 9 < N$, the model of Section IV-A underestimates the overall latency, since it does not account for the case where workers remain unavailable over multiple iterations. The model based on event-driven simulations remains accurate.

V. DSAG

In this section, we consider learning in cloud computing systems. In particular, we want an optimization method that i) is able to make progress even when some workers fail to

[9]We model the latency distribution by a gamma distribution, which we find provides more accurate predictions than the more commonly used shifted exponential distribution.

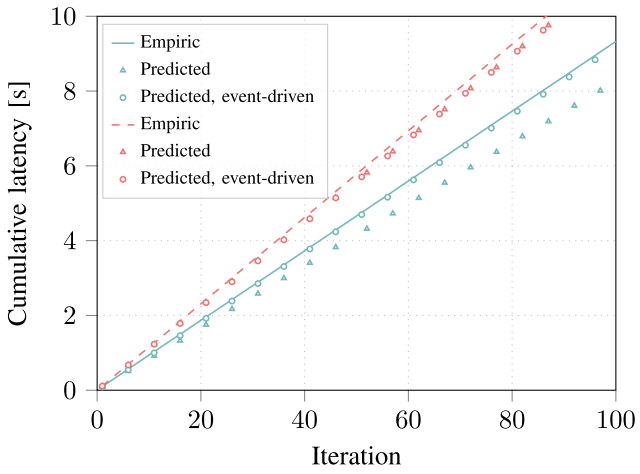


Fig. 6. Cumulative latency over 100 iterations. Blue curves correspond to $w = 9$ and red curves to $w = 72$. Each iteration ends once w workers have completed their task. For $w < N$, we need to account for the case where workers remain unavailable over several iterations, which we do using the model based on event-driven simulations.

respond, ii) has fast initial convergence, similar to SGD, which is achieved by performing many fast, but inexact, iterations, iii) eventually converges to the optimum, iv) allows for dynamic load-balancing, and v) has low update complexity. GD and SAG fail points i) and iv), SGD fails point iii), and coded computing methods fail either point ii) or iii), and, in most cases, points iv) and v).

To address i)–v), we introduce DSAG, which adapts SAG (as described in Section II-A) to distributed environments (hence the "D") with heterogenous and straggling workers. As with SAG, the key idea is to cache stale subgradients. However, unlike SAG, DSAG utilizes subgradients computed in previous iterations that arrive late. Further, DSAG allows for load-balancing by dynamically changing the number of data partitions (and hence the number of samples that make up each partition). DSAG meets all of the above criteria.

DSAG works as follows. Denote by

$$\boldsymbol{Y}_{i:j}^{(t)} \triangleq \sum_{k=i}^{j} \nabla f_k \left(\boldsymbol{V}^{(t)} \right)$$

the subgradient computed from samples i through j, where $j \geq i$. The coordinator maintains a set of such subgradients, denoted by $\mathcal{Y}$, which we refer to as the gradient cache. Upon receiving a subgradient $\boldsymbol{Y}_{i:j}^{(t)}$ from a worker, the coordinator first selects the subset of overlapping subgradients

$$\mathcal{Y}' \triangleq \left\{ \boldsymbol{Y}_{i':j'}^{(t')} \in \mathcal{Y} \ : \ i \leq i' \leq j \text{ or } i \leq j' \leq j \right\}.$$

If any such subgradient is more recent than the received subgradient (i.e., if $t' \geq t$ for some $\boldsymbol{Y}_{i':j'}^{(t')} \in \mathcal{Y}'$), the process is aborted and the received subgradient discarded. Otherwise, the overlapping subgradients are discarded in favour of the received subgradient, i.e.,

$$\mathcal{Y} \leftarrow (\mathcal{Y} \setminus \mathcal{Y}') \cup \left\{ \boldsymbol{Y}_{i:j}^{(t)} \right\}.$$

This process allows for changing partition boundaries at runtime, e.g., due to load-balancing, and can be implemented efficiently by storing the elements of $\mathcal{Y}$ as nodes in a tree data structure.[10] Denote by

$$\boldsymbol{H} \triangleq \sum_{\boldsymbol{Y} \in \mathcal{Y}} \boldsymbol{Y}$$

the sum of the elements of $\mathcal{Y}$. The coordinator maintains this sum by assigning

$$\boldsymbol{H} \leftarrow \boldsymbol{H} + \boldsymbol{Y}_{i:j}^{(t)} - \sum_{\boldsymbol{Y} \in \mathcal{Y}'} \boldsymbol{Y}$$

whenever a subgradient $\boldsymbol{Y}_{i:j}^{(t)}$ is inserted into $\mathcal{Y}$. Finally, $\boldsymbol{H}$ is used in place of the exact gradient ∇F to update $\boldsymbol{V}^{(t)}$, i.e.,

$$\boldsymbol{V}^{(t+1)} = G\left(\boldsymbol{V}^{(t)} - \eta \left(\frac{1}{\xi} \boldsymbol{H} + \nabla R\left(\boldsymbol{V}^{(t)} \right) \right) \right),$$

where ξ is the fraction of samples covered by the elements of $\mathcal{Y}$. Scaling the gradient in this way improves the rate of convergence for the iterations before the coordinator has received subgradients covering all samples.[11]

We remark that if there exists $\boldsymbol{Y}_{i':j'}^{(t')}$ in $\mathcal{Y}$ such that $i' = i$ and $j' = j$, the existing element can be updated in-place. In this case, and if the received subgradient is computed from the most recent iterate, the update process degrades to that of SAG.

A. Distributed Implementation

Here, we describe our distributed implementation of DSAG. In particular, we wish to maintain predictable and low latency in the presence of stragglers. For SAG or SGD, this can be achieved by only waiting for a subset of workers to return in each iteration, and ignoring any results computed by straggling workers. However, since the same workers are likely to be stragglers for extended periods of time, the subgradients received from the fastest subset of workers by the coordinator will not be selected uniformly at random, unless all workers store the entire dataset or the coordinator waits for all workers. This can significantly reduce the rate of convergence, since parts of the dataset may never factor into the learning process (see Section VII and Fig. 8).

DSAG addresses this shortcoming by utilizing stale results and through dynamic load-balancing. In particular, at the t-th iteration, the coordinator waits until it has received subgradients computed from $\boldsymbol{V}^{(t)}$ from at least w workers. During this time, the coordinator may also have received subgradients from previous iterations, which the coordinator stores if they are less stale than the currently stored subgradients it would replace. In this way, DSAG ensures that data from all workers eventually factor into the learning process. Note that samples stored by straggling workers still factor into the learning process less often than those stored by faster workers. This problem could be alleviated by storing the dataset with some redundancy and dynamically controlling which data each worker processes, similar to, e.g., the strategy proposed in [24]. Further, we allow for a small margin, such that after

[10] When using a tree data structure, the complexity of deleting and inserting subgradients is in $\mathcal{O}(\log |\mathcal{Y}|)$.

[11] A similar scaling is used by SAG.

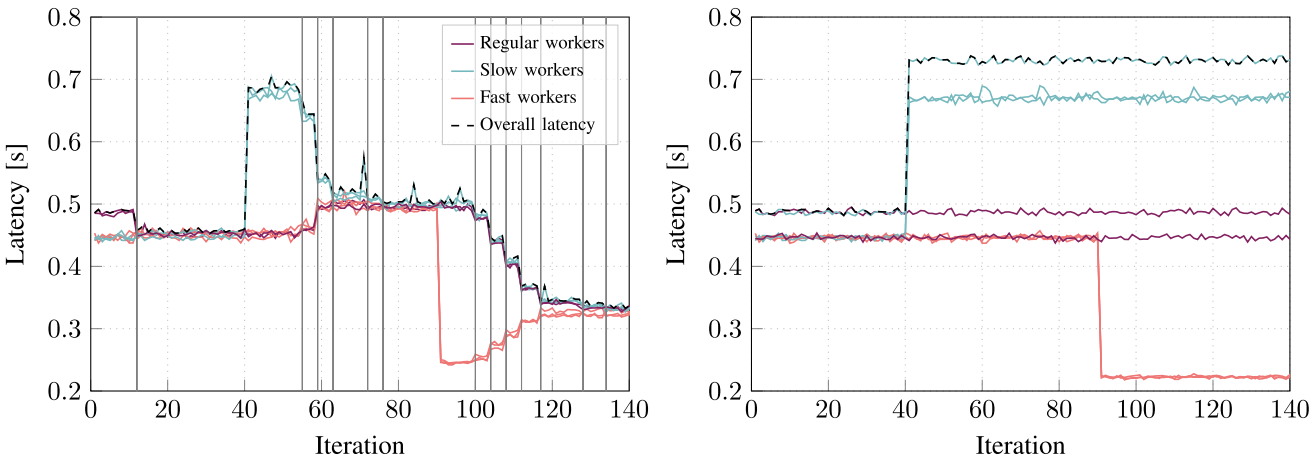


Fig. 7. Per-worker latency for $N = 8$ workers with (left) and without (right) load-balancing, when waiting for all workers (i.e., $w = N$). We artificially slow down 3 randomly selected workers (blue lines) after 40 iterations, and speed up another set of 3 randomly selected workers (red lines) after 90 iterations. Note that there is some natural variation in addition. The load-balancer automatically re-balances the workload in the iterations marked with gray lines. For the final 20 iterations, the overall latency of the unbalanced system is more than twice that of the system with load-balancing.

receiving the w-th fresh subgradient, the coordinator waits for 2% longer—collecting any subgradients received during this time—before updating the iterate. We do this since we find that several gradients may arrive approximately simultaneously from the point of view of the application acting as coordinator, perhaps because the operating system has buffered several incoming gradients before forwarding them to the application. Performance is not overly sensitive to the exact value chosen. The benefit is especially pronounced when combined with load-balancing. We explain the load-balancing strategy that we propose in Section VI.

B. Convergence of DSAG

DSAG builds upon the SAG method, for which the error $F\left(\boldsymbol{V}^{(t)}\right) - F\left(\boldsymbol{V}^{*}\right)$, where $\boldsymbol{V}^{*}$ is the optimum, decreases with $\mathcal{O}(1/t)$ and $\mathcal{O}(\rho^t)$, for some $\rho < 1$, for convex and strongly convex problems, respectively [10]. We do not have convergence proofs for DSAG—the analysis of asynchronous optimization methods is notoriously challenging, and the analysis of SAG is already complex—but we make a few remarks to relate the behavior of DSAG to that of SAG.

SAG updates one subgradient, selected uniformly at random over all partitions, at each iteration, and does not make use of stale subgradients. DSAG differs by updating one or more subgradients per iteration, and in that some of the updated subgradients may have been computed from a previous iterate, provided they are less stale than the replaced subgradients. Hence, the subgradients utilized by DSAG are at least as fresh as those used by SAG. Second, DSAG, unlike SAG, may discard cached subgradients if it receives a subgradient that is not aligned with an already cached subgradient (SAG does not support changing the partition boundaries at runtime).

Hence, we conjecture that the rate of convergence of DSAG is at least as good as that of SAG for iterations when no subgradients are discarded, and that it is worse than that of SAG in iterations where cached subgradients have been discarded, and until the discarded entries have been repopulated. We present empirical results that support this conjecture, see Section VII.

VI. Load-Balancing

Recall that computing speed typically differs between workers and may change over time (see Section III). Unless these differences are accounted for, fast workers typically spend a significant amount of time waiting for slower ones, and some workers may never be among the w fastest ones. Here, we propose a strategy to dynamically adjust the size of the data partitions stored by each worker to alleviate this issue. The process consists of three steps:

1) Latency profiling to estimate the probability distribution of Z_i and Y_i for each worker based on recorded latency (see Section VI-A).
2) Optimizing the number of samples processed by each worker, which we control by tuning the number of subpartitions the data stored by the worker is divided into. The optimization is based on the latency model of Section IV-B to predict the impact of each change (see Section VI-B).
3) Re-partitioning the local dataset for any workers for which the number of subpartitions has changed (see Section VI-C).

All three steps are performed asynchronously in parallel and are running continuously in the background. In particular, whenever the optimizer finishes, it is restarted to include any new latency recordings.[12] We show how load-balancing affects latency in Fig. 7, and we describe the three steps in detail next.

A. Latency Profiler

The latency profiler is responsible for estimating the mean and variance of the communication and computation latency of each worker, and for providing these to the optimizer. It takes as its input latency recorded both by the coordinator and the workers themselves. In particular, for each worker, the coordinator records the time between sending an iterate to the worker and receiving a response. Meanwhile, the workers record the time between starting to process a received iterate

[12]The load-balancer proposed in [41] takes a similar approach, but is designed for web services.

and having a response ready, and include this recording in their responses.

For each worker, we take the latency recorded by the worker as a sample of the computation latency, and the difference between the latency recorded by the worker and coordinator as a sample of the communication latency, i.e., for the i-th worker, as realizations of Z_i and Y_i, respectively. Hence, we record the round-trip communication latency, which includes the time required for data to be sent over the wire and any queuing at either end.

Next, for each worker, the profiler computes the sample mean and variance over a moving time window, i.e., samples older than a given deadline (in seconds) are discarded before processing. Choosing a window size involves making a trade-off—a larger window size makes statistics computed over it less noisy, but increases the time needed for the profiler to adapt to changes.[13] Note that load-balancing provides no benefit if the parameters of the underlying latency distributions change more quickly than they can be estimated. We denote by $e_{\mathsf{Y},i}$ and $v_{\mathsf{Y},i}$ the mean and variance of the communication latency of the i-th worker, and by $e_{\mathsf{Z},i}$ and $v_{\mathsf{Z},i}$ the mean and variance of the computation latency, computed as described above. For each worker, whenever new latency recordings are available, the mean and variance of its communication and computation latency are re-computed and sent to the optimizer, which uses them to fit probability distributions.[14]

B. Optimizer

For each worker, we tune its workload by changing the number of samples processed per iteration. Specifically, we optimize the number of subpartitions the data it stores locally is divided into. The optimizer takes as its input the most recent statistics computed by the profiler and a vector $\boldsymbol{p} = [p_1, \ldots, p_N]$ containing the current number of subpartitions for each worker, and returns an updated vector $\boldsymbol{p}' = [p'_1, \ldots, p'_N]$. For any solution $\boldsymbol{p}$, we impose a constraint on the expected overall per-iteration *contribution*, which we define as

$$h(\boldsymbol{p}) \triangleq \sum_{i=1}^{N} h_i(\boldsymbol{p}), \quad \text{with} \quad h_i(\boldsymbol{p}) \triangleq \frac{u_i(\boldsymbol{p}) n_i}{p_i n},$$

where n_i is the number of samples stored by the i-th worker and $u_i(\boldsymbol{p})$ the fraction of iterations that the i-th worker delivers a fresh result in. Hence, $h_i(\boldsymbol{p})$ is a measure of the extent to which the i-th worker contributes to the learning process. Note that u_i is a nonlinear function of $\boldsymbol{p}$, i.e., it depends on the workload of the entire set of workers. The goal of the optimizer is to minimize latency variation between workers within this constraint. More formally, its goal is to solve

$$\begin{aligned} \underset{\boldsymbol{p}'}{\arg\min} \quad & \frac{\max\left\{e'_{\mathsf{X},1}, \ldots, e'_{\mathsf{X},N}\right\}}{\min\left\{e'_{\mathsf{X},1}, \ldots, e'_{\mathsf{X},N}\right\}} \\ \text{s.t.} \quad & h(\boldsymbol{p}') \geq h_{\mathsf{min}}, \end{aligned} \tag{4}$$

where h_{min} is the constraint, and $e'_{\mathsf{X},i}$ is the expected overall latency of the i-th worker if its local dataset is split into p'_i subpartitions. Throughout the optimization process, we use the approximations

$$e'_{\mathsf{Z},i} \triangleq e_{\mathsf{Z},i} \frac{p_i}{p'_i}, \quad v'_{\mathsf{Z},i} \triangleq v_{\mathsf{Z},i} \frac{p_i^2}{p_i'^2}, \quad \text{and} \quad e'_{\mathsf{X},i} \triangleq e_{\mathsf{Y},i} + e'_{\mathsf{Z},i},$$

where p_i is the current number of subpartitions of the i-th worker. Hence, we linearize the mean and variance of the computation latency around the value of p_i for which it was recorded.[15]

It is difficult to compute u_i, and thus h, analytically, but u_i can be estimated via event-driven simulations as explained in Section IV-B.[16] However, this requires that the optimizer i) is robust against noise in the estimates of u_i, and ii) evaluates u_i a small enough number of times to be computationally fast enough to provide useful solutions in time. We find that traditional optimization techniques that, e.g., rely on gradients, fail the first criteria, while *meta-heuristic* techniques (e.g., evolutionary algorithms) fail the second. Hence, we propose an optimizer that solves (4) by making small changes to $\boldsymbol{p}$ in an iterative fashion.

At a high level, the optimizer attempts to increase the contribution of workers that are always among the w fastest by giving them more work, without increasing the overall latency. This increases the overall per-iteration contribution, thus giving the optimizer leeway to reduce the overall iteration latency by reducing the workload of the slowest workers. The proposed algorithm is given in Algorithm 1. Since h is estimated via simulations, we evaluate the constraint with a 1% tolerance. Finally, we set the constraint to be $h_{\mathsf{min}} = h(\boldsymbol{p}_0)$, where $\boldsymbol{p}_0$ is the baseline number of subpartitions for each worker used at the start of the first iteration. This is to ensure that load-balancing does not reduce the rate of convergence.

C. Re-Partitioning

Whenever the optimizer produces an updated number of subpartitions for a particular worker, the update is included with the next iterate sent to the worker, which re-partitions its local dataset. However, re-partitioning carries a cost, since it invalidates subgradients cached by the coordinator. Here, we show how to minimize the number and impact of such *cache evictions* resulting from re-partitioning. First, we partition the data matrix such that the i-th worker stores locally the submatrix

$$\boldsymbol{X}^{(i)} \triangleq \boldsymbol{X}_{p_{\mathsf{start}}(n,N,i):p_{\mathsf{stop}}(n,N,i)},$$

where

$$p_{\mathsf{start}}(n,p,i) = \left\lfloor \frac{(i-1)\,n}{p} \right\rfloor + 1 \quad \text{and} \quad p_{\mathsf{stop}}(n,p,i) = \left\lfloor \frac{in}{p} \right\rfloor,$$

with $1 \leq p \leq n$ and $1 \leq i \leq p$. Next, for each worker, we subpartition the data it stores locally, such that, in each iteration, the i-th worker processes the matrix

[13] We use a window size of 10 seconds, which we find is a good trade-off for the applications we consider.

[14] The shape and scale parameter of a gamma-distributed random variable with mean e and variance v is e^2/v and v/e, respectively.

[15] This linearization is motivated by Fig. 1. If latency has been measured for several different values of p_i, we use a weighted average over the values of p_i for which we have recordings.

[16] With our implementation, for $N = 100$ workers and $w = 50$, simulating 100 iterations of the learning process takes about 1.5 milliseconds.

Algorithm 1 Load-Balancer

procedure OPTIMIZE($\boldsymbol{p}$)
  $\boldsymbol{p}' \leftarrow \boldsymbol{p}$
  $i \leftarrow \arg\max \left[e'_{\mathsf{X},1}, \ldots, e'_{\mathsf{X},N}\right]$ ▷ Slowest worker
  for $j = 1, \ldots, N$ **do**
    $p'_j \leftarrow \left\lfloor \frac{e_{\mathsf{Z},j} p_j}{e_{\mathsf{Y},i} + e_{\mathsf{Z},i} - e_{\mathsf{Y},j}} \right\rfloor$ ▷ Equalize total latency
  end for
  while $h(\boldsymbol{p}') < h_{\mathsf{min}}$ **do**
    $i \leftarrow \arg\min \left[e'_{\mathsf{X},1}, \ldots, e'_{\mathsf{X},N}\right]$ ▷ Fastest worker
    $p'_i \leftarrow \lfloor 0.99 \cdot p'_i \rfloor$ ▷ Increase workload
  end while
  while $h(\boldsymbol{p}') \geq 0.99 \cdot h_{\mathsf{min}}$ **do**
    $i \leftarrow \arg\max \left[e'_{\mathsf{X},1}, \ldots, e'_{\mathsf{X},N}\right]$ ▷ Slowest worker
    $p'_i \leftarrow \lceil 1.01 \cdot p'_i \rceil$ ▷ Decrease workload
  end while
  return $\boldsymbol{p}'$
end procedure
loop ▷ Optimizer main loop
  Collect updated latency statistics from the profiler
  $\boldsymbol{p} \leftarrow$ OPTIMIZE($\boldsymbol{p}$)
  Distribute the updated vector $\boldsymbol{p}$
end loop

$\boldsymbol{X}^{(i)}_{p_{\mathsf{start}}(n_i,p_i,k_i):p_{\mathsf{stop}}(n_i,p_i,k_i)}$, for some index k_i. Hence, we may tune the workload of a worker by sending it a new value p_i, which changes the number of samples processed per iteration. The following example shows how doing so leads to cache evictions.

Example 1 (Re-partitioning): Consider a scenario with 2 workers, $n_1 = n_2 = 10$ (i.e., $n = 20$), and $p_1 = p_2 = 2$, such that the partitions on the first worker are $\boldsymbol{X}_{1:5}$ and $\boldsymbol{X}_{6:10}$, and $\boldsymbol{X}_{11:15}$ and $\boldsymbol{X}_{16:20}$ on the second. Now, say that we let $p_1 \leftarrow 3$, such that the partitions on the first worker are $\boldsymbol{X}_{1:3}$, $\boldsymbol{X}_{4:6}$, and $\boldsymbol{X}_{7:10}$. Prior to this change, the coordinator stores gradients corresponding to partitions $\boldsymbol{X}_{1:5}$ and $\boldsymbol{X}_{6:10}$. Now, if in the next iteration the worker sends to the coordinator the subgradient computed over $\boldsymbol{X}_{4:6}$, both of the existing entries need to be evicted before inserting the new subgradient, leading to a lower rate of convergence until the missing cache entries have been populated.

We find that cache evictions due to re-partitioning can significantly reduce the rate of convergence, since the gradient used by DSAG no longer covers all samples of the dataset. We use two strategies to reduce the severity of this issue. First, we refrain from distributing an update $\boldsymbol{p}'$ to the workers until doing so would improve the objective function (4) by more than some threshold (e.g., 10%). Second, we process subpartitions in order to minimize the number of iterations for which evicted cache entries remain empty. More formally, the i-th worker stores a counter k_i that it increments in a cyclic fashion each time it receives an iterate, i.e.,[17]

$$k_i \leftarrow \mathrm{mod}\,(k_i, p_i) + 1. \tag{5}$$

[17]Note that, when $w < N$, workers, unlike the coordinator, are unaware of the current iteration index since they may have remained unavailable for an arbitrary amount of time.

Algorithm 2 Partition Alignment

$k_i \leftarrow \mathrm{mod}\,(k_i, p_i) + 1$
$k'_i \leftarrow p_{\mathsf{trans}}(n_i, p_i, p'_i, k_i)$
while $p_{\mathsf{start}}(n_i, p'_i, k'_i) \neq p_{\mathsf{start}}(n_i, p_i, k_i)$ **do**
  $k'_i \leftarrow k'_i - 1$
  $k_i \leftarrow p_{\mathsf{trans}}(n, p'_i, p_i, k'_i)$
end while
$p_i \leftarrow p'_i$
$k_i \leftarrow k'_i$

Next, it computes the gradient with respect to the k_i-th of its locally stored partitions. We show the benefit of this approach with the following example.

Example 2 (Continuation of Example 1): Immediately after re-partitioning, the coordinator stores subgradients computed over partitions $\boldsymbol{X}_{1:5}$ and $\boldsymbol{X}_{6:10}$ (we omit partitions stored by the second worker). To minimize cache evictions, over the following 3 iterations, the first worker sends to the coordinator:

1) *The gradient over $\boldsymbol{X}_{1:3}$, evicting the gradient over $\boldsymbol{X}_{1:5}$, resulting in a cache with the gradients over $\boldsymbol{X}_{1:3}$ and $\boldsymbol{X}_{6:10}$, leaving the gradient over $\boldsymbol{X}_{4:5}$ missing.*
2) *The gradient over $\boldsymbol{X}_{4:6}$, evicting the gradient over $\boldsymbol{X}_{6:10}$, resulting in a cache with the gradients over $\boldsymbol{X}_{1:3}$ and $\boldsymbol{X}_{4:6}$, leaving the gradient over $\boldsymbol{X}_{7:10}$ missing.*
3) *The gradient over $\boldsymbol{X}_{7:10}$, resulting in a cache with the gradients over $\boldsymbol{X}_{1:3}$, $\boldsymbol{X}_{4:6}$, and $\boldsymbol{X}_{7:10}$, leaving no missing entries.*

In this case, the gradients over $\boldsymbol{X}_{4:5}$ and $\boldsymbol{X}_{7:10}$ are missing from the cache for 1 iteration each. If instead the worker had started by sending the gradient over $\boldsymbol{X}_{4:6}$, either the gradient over $\boldsymbol{X}_{1:3}$ or $\boldsymbol{X}_{7:10}$ would have been missing for 2 iterations, and the other for 1 iteration, resulting in a lower rate of convergence.

This approach is most effective if the first sample of the partition processed immediately after a re-partitioning is aligned with the first sample of a partition already in the cache, since otherwise the evicted entries are not re-populated until after a full pass over the data (this happens if the first worker in Example 2 starts by processing $\boldsymbol{X}_{4:6}$ after re-partitioning). Hence, when changing the number of subpartitions of the i-th worker from p_i to p'_i, instead of using (5), we update k_i using Algorithm 2, which relies on the function

$$p_{\mathsf{trans}}(n_i, p_i, p'_i, k_i) = \left\lceil p_{\mathsf{start}}(n_i, p_i, k_i) \frac{p'_i}{n_i} \right\rceil,$$

that returns the index of the partition containing sample $p_{\mathsf{start}}(n_i, p_i, k_i)$ when the number of partitions is p'_i. We illustrate Algortihm 2 with Example 3.

Example 3 (Continuation of Example 2): Say that, prior to re-partitioning, the first worker processed partition $\boldsymbol{X}_{1:5}$, so that $k_1 = 1$, and that we are changing the number of subpartitions from $p_1 = 2$ to $p'_1 = 3$. In this case, the $n_1 = 10$ samples stored by the first worker are subpartitioned as follows,

$$p_1 = 2: \quad [1, \quad 2, 3\,, \quad 4, 5], \quad [6\,, \, 7, \quad 8, 9, \quad 10]$$
$$p'_1 = 3: \quad [1, \quad 2, 3], \quad [4, 5, \quad 6], [\,7, \quad 8, 9, \quad 10]$$

where the indices are the row indices of $\boldsymbol{X}$, and brackets in the first and second line indicate partition boundaries before and after re-partitioning, respectively. Now, Algorithm 2 finds a partition out of $p'_1 = 3$ partitions such that its first sample is equal to that of some partition out of $p_1 = 2$. It proceeds as follows. First, let $k_1 \leftarrow \mathrm{mod}\,(1,2) + 1 = 2$ (Line 1), and $k'_1 \leftarrow p_{\mathsf{trans}}(10,2,3,k_1) = 2$ (Line 2). Since the k_1-th and k'_1-th partitions are not aligned (Lines 3)—$p_{\mathsf{start}}(10,3,k'_1) = 4 \neq 6 = p_{\mathsf{start}}(10,2,k_1)$—we let $k'_1 \leftarrow k'_1 - 1 = 1$ and $k_1 \leftarrow p_{\mathsf{trans}}(10,3,2,k'_1) = 1$ (Lines 4 and 5). Now the partitions are aligned (Line 3)—$p_{\mathsf{start}}(10,2,k_1) = 1 = p_{\mathsf{start}}(10,3,k'_1)$—and the worker assigns $p_1 \leftarrow p'_1$ and $k_1 \leftarrow k'_1$ (Lines 7 and 8).

Note that Algorithm 2 always terminates, since the first partition always starts at the first sample stored by the worker, i.e., $k_i = k'_i = 1$ results in the partitions being aligned regardless of the values of p_i and p'_i. However, $k_i = k'_i = 1$ may not be the only solution. For example, if $n_i = 10$, $p_i = 2$, and $p'_i = 4$, then $k_i = 2$ and $k'_i = 3$ also results in aligned partitions—$p_{\mathsf{start}}(10,4,3) = 6 = p_{\mathsf{start}}(10,2,2)$. Hence, Algorithm 2 improves timeliness, since always setting $k_i = k'_i = 1$ after re-partitioning could result in the first few subpartitions being processed much more frequently than the others.

VII. Experimental Results

Here, we evaluate the performance of DSAG for PCA and logistic regression, and compare it to that of GD, SGD, SAG, and coded computing methods, on eX3 and AWS (see Section II-B for details). We also evaluate the impact of load-balancing on performance for DSAG, SAG, and SGD. For PCA, the loss function is given by

$$R(\boldsymbol{V}) = \frac{1}{2}\left\|\boldsymbol{V}\right\|_{\mathsf{F}}^2 \quad \text{and} \quad f_i(\boldsymbol{V}) = \frac{1}{2}\left\|\boldsymbol{x}_i - \boldsymbol{x}_i\boldsymbol{V}\boldsymbol{V}^{\mathsf{T}}\right\|^2, \quad (6)$$

where the columns of $\boldsymbol{V}$ make up the computed principal components, $\|\cdot\|$ denotes the Euclidean norm, and $\|\cdot\|_{\mathsf{F}}$ denotes the Frobenius norm, and $\boldsymbol{V}$ is updated according to (2). For PCA, $G(\cdot)$ in (2) is the Gram-Schmidt operator, i.e., $G(\cdot)$ takes an input matrix and applies the Gram-Schmidt orthogonalization procedure to its columns such that the columns of the resulting matrix form an orthonormal basis with the same span as the columns of the input matrix. For logistic regression, $\boldsymbol{V}$ is a vector, and the loss is the *L2-regularized* classification error, i.e.,

$$R(\boldsymbol{V}) = \frac{\lambda}{2}\|\boldsymbol{V}\|^2 \quad \text{and} \quad f_i(\boldsymbol{V}) = \frac{\log\left[1 + \exp\left(-b_i\left\langle \boldsymbol{x}_i, \boldsymbol{V}\right\rangle\right)\right]}{n},$$

where $b_1, \ldots, b_n$ are the classification labels, with $b_i \in \{-1, +1\}$, $\langle\cdot,\cdot\rangle$ denotes vector inner product, λ is the regularization coefficient, and in this case $G(\cdot)$ is the identity operator. For PCA, we use a matrix derived from the 1000 Genomes phase-3 dataset [44], and for logistic regression we use the HIGGS dataset [45] (see Section II-B). For PCA, we compute the top 3 principle components, and for logistic regression, as in [10], we normalize all features to have zero mean and unit variance, add an intercept equal to 1, and set the regularization coefficient to 1 divided by the number of samples, i.e., $\lambda = 1/11000000$. We use 100 and 10 subpartitions for PCA and logistic regression, respectively.

We measure performance as the latency to solve either PCA or logistic regression to within some precision of the optimum, and, for all scenarios, we plot the suboptimality gap, i.e., the difference between the explained variance (for PCA) or classification error (for logistic regression) of the computed solution and that of the optimum, as a function of time. The results shown are averages over 5 experiments conducted on the respective computing systems. For GD and coded computing, we use a stepsize of $\eta = 1.0$ for both PCA and logistic regression, whereas for DSAG, SAG, and SGD, we use a stepsize of $\eta = 0.9$ for PCA and $\eta = 0.25$ for logistic regression (we need to reduce the stepsize relative to GD for the stochastic methods to ensure convergence). We remark that GD applied to solving the optimization problem in (1) with the loss function in (6) with $\eta = 1.0$ is equivalent to the *power method* for PCA, i.e., the power method is a special case of GD.

A. Coded Computing

Coded computing methods with *code rate* r (a quantity between 0 and 1) make it possible to either recover the gradient exactly (e.g., [4]) or an approximation thereof (e.g., [5], [6], [25], [26]) from intermediate results computed by a subset of the workers, at the expense of increasing the computational load of each worker by a factor $1/r$ relative to GD. The gradient is recovered via a decoding operation (that typically reduces to solving a system of linear equations), the complexity of which usually increases superlinearly with the number of workers. Ideally, the gradient can be recovered exactly from the results computed by any set of $\lceil rN \rceil$ workers—codes with this property are referred to as *maximum distance separable* (MDS) codes—but increasing the number of results required can allow for reducing the decoding complexity [12].

To compare against the wide range of coded computing methods, we use an idealized estimate derived from the GD results. In particular, we assume that the code is MDS, but that the decoding complexity is zero. More specifically, we set the latency per iteration equal to that of the $\lceil rN \rceil$-th fastest worker after scaling the computational latency recorded for GD of all workers by $1/r$, and the rate of convergence equal to that of GD. Hence, both the latency and rate of convergence of the estimate are bounds on what is achievable with coded computing. Further, for PCA, this bound includes coded computing methods for matrix multiplication (e.g., [3], [6], [12], [13]), since GD is equivalent to the power method in this instance.

B. Artificial Scenario

While we are primarily interested in cloud computing systems, for the sake of reproducibility, we first present results recorded for $N = 49$ workers on eX3, which is much more homogenous than the cloud, where we introduce variability in a controlled manner. In particular, we artificially increase the computational latency of the i-th worker by a factor $(i/N) \cdot 0.4$ by introducing delays at the worker nodes.[18] Further, we remove this artificial latency for workers 40 through 49 after one second has passed from the start of the learning

[18]This level of variability is comparable to what we have observed for instances of type `F2s_v2` on Azure.

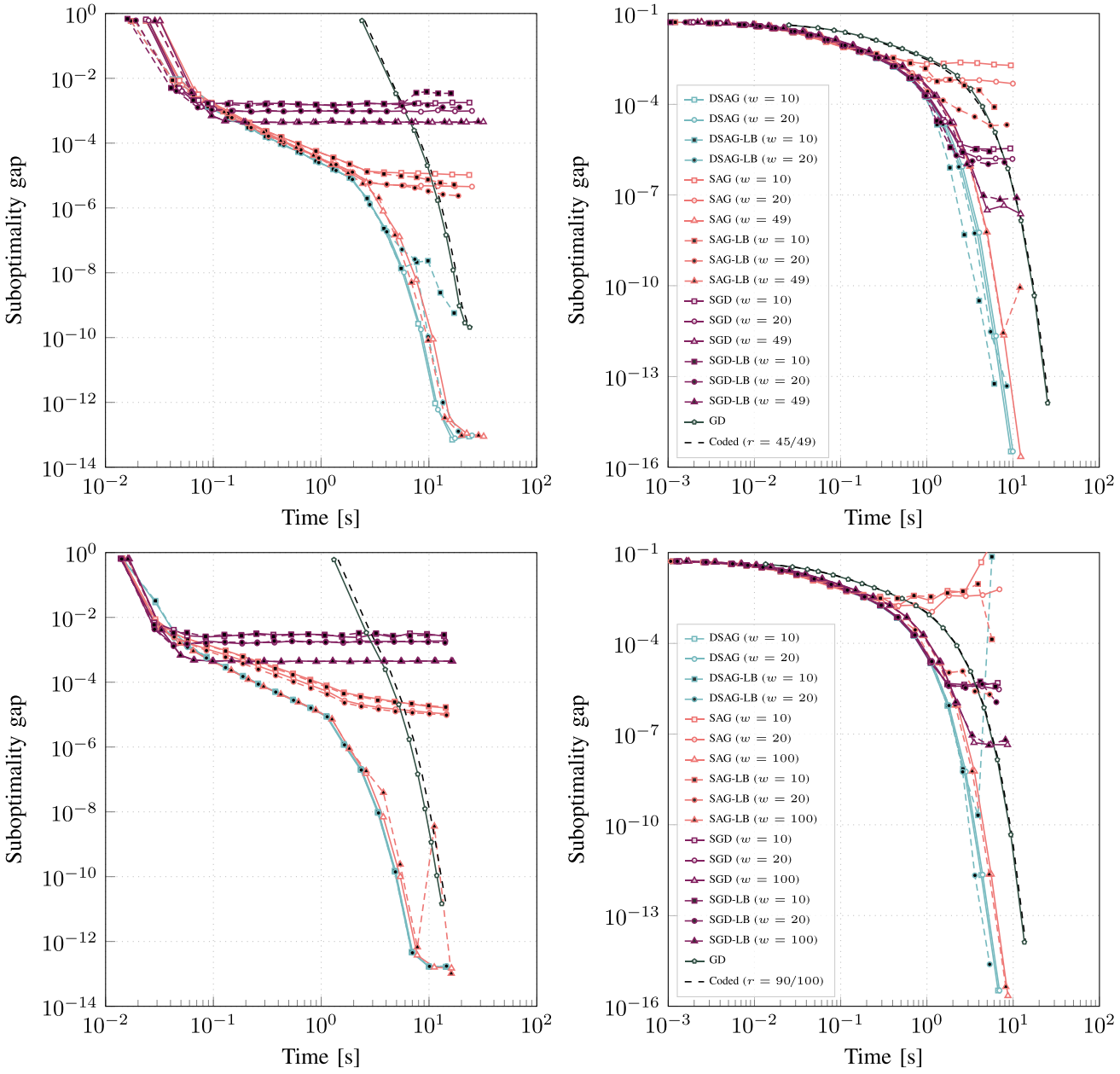


Fig. 8. Convergence of PCA (left column) and logistic regression (right column) for $N = 49$ workers on eX3 (top row) and $N = 100$ workers on AWS (bottom row). The dataset is split evenly over the workers and is initially subdivided into 100 subpartitions for PCA and 10 subpartitions for logistic regression. Stochastic optimization methods with $w < N$ effectively reduce the impact on latency of straggling workers, but only DSAG ensures convergence to the optimum. Load-balancing can improve latency further in some instances. The results shown are averages over 5 experiments.

process to simulate those workers coming out of a high-latency burst.

In Fig. 8 (top row), we show convergence of PCA (left) and logistic regression (right) in this scenario. First, for both PCA and logistic regression, at least one of the stochastic methods (DSAG, SAG, and SGD) is more than twice as fast as GD for any suboptimality gap—performing many fast, but inexact iterations, is often preferable to performing fewer more accurate iterations. However, for SAG, when $w < N$, and SGD, there is a point beyond which convergence effectively stops. For SGD, the high variance of its gradient estimate prevents it from converging—SGD is not a variance-reduced method[19]—although larger w increases precision since it causes a larger fraction of the dataset to be factored in. For SAG, which is variance-reduced, convergence stops as a result of not factoring in samples stored by workers that are straggling over many subsequent iterations (see Section V-A). For $w = N$, SAG converges to the optimum since all workers participate in each iteration, at the expense of increased latency, i.e., there is a trade-off between straggler-resiliency and convergence.

DSAG extends SAG by incorporating stale results, and, as a result, converges to the optimum even when $w < N$, allowing it to achieve both low latency and high precision in the presence of stragglers. Selecting w optimally is challenging, since it depends on the variance of the underlying dataset. Hence, we rely on experiments for choosing w. In this instance, DSAG with $w = 10$ is the fastest of all methods considered for

[19]A popular variance reduction technique for SGD is to gradually decrease the stepsize, but doing so reduces the rate of convergence.

both PCA and logistic regression, except for when solving PCA to within a precision of about 10^{-3}, in which case SGD is faster. In particular, DSAG with $w = 10$ achieves a rate of convergence comparable to that of SAG with $w = N$, but reduces latency by an amount that is proportional to the amount of latency variability. For example, for PCA, DSAG with $w = 10$ is between about 20% (for a suboptimality gap of 10^{-4}) and 30% (for a suboptimality gap of 10^{-8} or lower) faster than SAG with $w = N$, and, for logistic regression, DSAG with $w = 10$ is about 30% faster than SAG when the suboptimality gap is 10^{-4} or lower. Finally, for both PCA and logistic regression, the straggler-resiliency afforded by coding is canceled out by the higher computational load. Here, we consider a code rate $r = 45/49$, which we find yields lower latency compared to the lower rates typically used in coded computing (e.g., in [4], [5], [6], [25], and [26]).

Next, we evaluate the proposed load-balancer, which we apply to DSAG, SAG, and SGD—we refer to the corresponding load-balanced methods as DSAG-LB, SAG-LB, and SGD-LB, respectively. For SAG-LB, to allow for dynamically re-sizing the data partitions, we use the DSAG update rule (see Section V), except that stale results are discarded, instead of that in [10]. There are two important caveats. First, it takes about 7 and 0.5 seconds for the load-balancer to produce a first solution for PCA and logistic regression, respectively, before which it has no effect (it is slower for PCA due to the larger number of subpartitions). Second, load-balancing can reduce precision when the suboptimality gap is low due to cache invalidation (see Example 1).[20] This problem is especially severe when the number of subpartitions is large relative to the total number of iterations (as is the case for the PCA problem we consider) since a larger fraction of the overall optimization time is spent before the cache is re-populated. As a result, load-balancing does not result in a speedup for PCA. However, for DSAG with $w = 10$ applied to logistic regression, load-balancing results in about 30% to 40% lower latency when the suboptimality gap is between 10^{-6} and 10^{-12}. Interestingly, the primary mechanism by which load-balancing reduces latency is by increasing the average number of workers that respond within the 2% latency tolerance (see Section V-A), which allows it to reduce the workload for all workers without reducing the expected overall contribution (see Section VI-B). Further, load-balancing improves the precision of SAG with $w < N$ since the probability of each worker participating becomes more uniform.

C. Performance on AWS

Here, we consider performance on a cluster composed of $N = 100$ workers on AWS. To ensure that the results are representative, we use a fresh set of virtual machine instances for each set of experiments. While the results on AWS are similar to those on eX3, there are a few important differences. First, communication latency is about an order of magnitude higher on AWS compared to eX3, whereas computation latency is about 10% to 30% higher, depending on the scenario (when accounting for the fact that the per-worker computational load is about half that of eX3). We show the approximate latency range for the stochastic methods without load-balancing in Table I. As a result, the performance advantage of the stochastic methods compared to GD and coded computing is reduced somewhat, although they are still about twice as fast.

[20]This problem could be alleviated by disabling load-balancing when close to convergence.

TABLE I
APPROXIMATE LATENCY OF STOCHASTIC METHODS

	Comm. latency [s]	Comp. latency [s]
eX3 PCA	$2 \cdot 10^{-5}$ to $6 \cdot 10^{-5}$	$2.2 \cdot 10^{-2}$ to $3.1 \cdot 10^{-2}$
AWS PCA	$1.5 \cdot 10^{-4}$ to $1 \cdot 10^{-3}$	$1.3 \cdot 10^{-2}$ to $1.6 \cdot 10^{-2}$
eX3 Log. r.	$0.2 \cdot 10^{-5}$ to $3 \cdot 10^{-5}$	$1.8 \cdot 10^{-3}$ to $2.5 \cdot 10^{-3}$
AWS Log. r.	$1 \cdot 10^{-4}$ to $6 \cdot 10^{-4}$	$1.1 \cdot 10^{-3}$ to $1.3 \cdot 10^{-3}$

Second, latency is noisier on AWS, with workers experiencing unpredictable high-latency bursts, which may affect both communication and computation latency. Further, the noise makes up a larger fraction of the overall latency for lower average latency. As a result, the straggler problem is more severe for logistic regression than for PCA, for which each iteration is much slower (see Table I). In particular, for PCA, DSAG with $w = 10$ is only up to about 10% faster than SAG with $w = N$ (for a suboptimality gap below 10^{-6}), whereas for logistic regression DSAG with $w = 10$ is about 30% faster when the suboptimality gap is 10^{-4} or lower.

Finally, the level of static variation in latency between workers is smaller on AWS than on eX3 (which we modeled after Azure). Hence, the advantage of load-balancing is smaller—about 10% to 15% for DSAG-LB with $w = 20$ compared to DSAG with $w = 10$ (which is fastest when not load-balancing), for logistic regression, and up to about 50% faster than SAG with $w = N$.

VIII. CONCLUSION

Recently, there has been significant interest in coded computing, which is often motivated by the straggler problem in distributed machine learning and data analytics. However, we find that there are applications for which coded computing reduces performance compared to GD, even when not accounting for the decoding latency, which may be substantial. One issue is that coded computing methods are often designed under the assumption that latency is i.i.d. between workers, which is typically not the case. Further, there are fundamental differences between the distributed computing problem and the communication problem that erasure correcting codes were designed to address. In particular, we find that, for iterative methods, missing information can be substituted by stale information received over previous iterations, with only a marginal reduction to the rate of convergence. In this way, variance-reduced stochastic optimization methods can achieve straggler-resiliency without increasing computational complexity, as is the case for coded computing.

In this work, we have proposed DSAG, which alleviates the straggler problem by only waiting for the fastest subset of workers, while integrating the results computed by stragglers in an asynchronous manner. DSAG is based on the SAG

method and uses a variance reduction strategy to improve convergence. Further, we have proposed a load-balancing strategy that is able to counter some of the latency variability that exists in distributed computing systems, without moving data between workers. For both PCA and logistic regression, we have shown that DSAG can reduce latency significantly—by up to 50% for logistic regression on AWS, compared to SAG—through a combination of load-balancing and only waiting for the fastest subset of workers.

REFERENCES

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

[2] L. A. Barroso, U. Hölzle, and P. Ranganathan, "WSC hardware building blocks," in *The Datacenter as a Computer: Designing Warehouse-Scale Machines*, 3rd ed., M. Martonosi, Ed. San Rafael, CA, USA: Morgan & Claypool, 2018, ch. 3.

[3] 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.

[4] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, Sydney, NSW, Australia, Aug. 2017, pp. 3368–3376.

[5] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, "Straggler mitigation in distributed optimization through data encoding," in *Proc. Neural Inf. Process. Syst. (NIPS)*, Long Beach, CA, USA, Dec. 2017, pp. 5434–5442.

[6] Y. Yang, P. Grover, and S. Kar, "Coding for a single sparse inverse problem," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Vail, CO, USA, Jun. 2018, pp. 1575–1579.

[7] Y. Sun, J. Zhao, S. Zhou, and D. Gündüz, "Heterogeneous coded computation across heterogeneous workers," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Waikoloa, HI, USA, Dec. 2019, pp. 1–6.

[8] Y. Sun, F. Zhang, J. Zhao, S. Zhou, Z. Niu, and D. Gündüz, "Coded computation across shared heterogeneous workers with communication delay," *IEEE Trans. Signal Process.*, vol. 70, pp. 3371–3385, 2022.

[9] B. Buyukates, E. Ozfatura, S. Ulukus, and D. Gündüz, "Gradient coding with dynamic clustering for straggler-tolerant distributed learning," *IEEE Trans. Commun.*, early access, Apr. 12, 2022, doi: 10.1109/TCOMM.2022.3166902.

[10] M. Schmidt, N. Le Roux, and F. Bach, "Minimizing finite sums with the stochastic average gradient," *Math. Program.*, vol. 162, nos. 1–2, pp. 83–112, Mar. 2017.

[11] A. Severinson. (2021). *DSAG Source Code and Latency Traces*. [Online]. Available: http://www.github.com/severinson/CodedComputing.jl

[12] A. Severinson, A. Graell i Amat, and E. Rosnes, "Block-diagonal and LT codes for distributed computing with straggling servers," *IEEE Trans. Commun.*, vol. 67, no. 3, pp. 1739–1753, Mar. 2019.

[13] A. Severinson, A. Graell i Amat, E. Rosnes, F. Lázaro, and G. Liva, "A droplet approach based on Raptor codes for distributed computing with straggling servers," in *Proc. 10th Int. Symp. Turbo Codes Iterative Inf. Process. (ISTC)*, Hong Kong, Dec. 2018, pp. 1–5.

[14] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial codes: An optimal design for high-dimensional coded matrix multiplication," in *Proc. Neural Inf. Process. Syst. (NIPS)*, Long Beach, CA, USA, Dec. 2017, pp. 4403–4413.

[15] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Vail, CO, USA, Jun. 2018, pp. 2022–2026.

[16] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar, "Slow and stale gradients can win the race: Error-runtime trade-offs in distributed SGD," in *Proc. Int. Conf. Artif. Intell. Statist. (AISTATS)*, Canary Islands, Spain, Apr. 2018, pp. 803–812.

[17] K. Lee, C. Suh, and K. Ramchandran, "High-dimensional coded matrix multiplication," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Aachen, Germany, Jun. 2017, pp. 2418–2422.

[18] M. Fahim and V. R. Cadambe, "Numerically stable polynomially coded computing," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Paris, France, Jul. 2019, pp. 3017–3021.

[19] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," *IEEE Trans. Inf. Theory*, vol. 66, no. 1, pp. 278–301, Jan. 2020.

[20] V. Gupta, S. Wang, T. Courtade, and K. Ramchandran, "OverSketch: Approximate matrix multiplication for the cloud," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Seattle, WA, USA, Dec. 2018, pp. 298–304.

[21] T. Jahani-Nezhad and M. A. Maddah-Ali, "CodedSketch: Coded distributed computation of approximated matrix multiplication," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Paris, France, Jul. 2019, pp. 2489–2493.

[22] C.-S. Yang, R. Pedarsani, and A. S. Avestimehr, "Timely-throughput optimal coded computing over cloud networks," in *Proc. 20th ACM Int. Symp. Mobile Ad Hoc Netw. Comput. (MobiHoc)*, Catania, Italy, Jul. 2019, pp. 301–310.

[23] M. Ye and E. Abbe, "Communication-computation efficient gradient coding," in *Proc. 35th Int. Conf. Mach. Learn. (ICML)*, Stockholm, Sweden, Jul. 2018, pp. 5610–5619.

[24] E. Ozfatura, B. Buyukates, D. Gündüz, and S. Ulukus, "Age-based coded computation for bias reduction in distributed learning," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Taipei, Taiwan, Dec. 2020, pp. 1–6.

[25] R. Bitar, M. Wootters, and S. El Rouayheb, "Stochastic gradient coding for straggler mitigation in distributed learning," *IEEE J. Sel. Areas Inf. Theory*, vol. 1, no. 1, pp. 277–291, May 2020.

[26] H. Wang, Z. Charles, and D. Papailiopoulos, "ErasureHead: Distributed gradient descent without delays using approximate gradient coding," Jan. 2019, *arXiv:1901.09671*.

[27] P. Richtárik and M. Takáč, "Stochastic reformulations of linear systems: Algorithms and convergence theory," *SIAM J. Matrix Anal. Appl.*, vol. 41, no. 2, pp. 487–524, Jan. 2020.

[28] F. Hanzely, K. Mishchenko, and P. Richtárik, "SEGA: Variance reduction via gradient sketching," in *Proc. Neural Inf. Process. Syst. (NeurIPS)*, Montréal, QC, Canada, Dec. 2018, pp. 2082–2093.

[29] R. M. Gower and P. Richtárik, "Randomized iterative methods for linear systems," *SIAM J. Matrix Anal. Appl.*, vol. 36, no. 4, pp. 1660–1690, Jan. 2015.

[30] A. Defazio, F. Bach, and S. Lacoste-Julien, "SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives," in *Proc. Neural Inf. Process. Syst. (NIPS)*, Montréal, QC, Canada, Dec. 2014, pp. 1646–1654.

[31] C. Calauzènes and N. Le Roux, "Distributed SAGA: Maintaining linear convergence rate with limited communication," May 2017, *arXiv:1705.10405*.

[32] L. M. Nguyen, J. Liu, K. Scheinberg, and M. Takáč, "SARAH: A novel method for machine learning problems using stochastic recursive gradient," in *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, Sydney, NSW, Australia, Jul. 2017, pp. 2613–2621.

[33] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Proc. Neural Inf. Process. Syst. (NIPS)*, Lake Tahoe, NV, USA, Dec. 2013, pp. 315–323.

[34] E. Gorbunov, K. Burlachenko, Z. Li, and P. Richtárik, "MARINA: Faster non-convex distributed learning with compression," in *Proc. 38th Int. Conf. Mach. Learn. (ICML)*, Jul. 2021, pp. 3788–3798.

[35] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous parallel stochastic gradient for nonconvex optimization," in *Proc. Neural Inf. Process. Syst. (NIPS)*, Montréal, QC, Canada, Dec. 2015, pp. 2737–2745.

[36] S. Gupta, W. Zhang, and F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study," in *Proc. IEEE Int. Conf. Data Mining (ICDM)*, Barcelona, Spain, Dec. 2016, pp. 171–180.

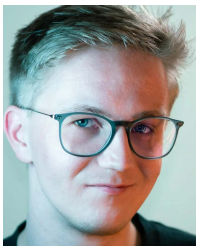
[37] F. Niu, B. Recht, C. Ré, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," in *Proc. Neural Inf. Process. Syst. (NIPS)*, Granada, Spain, Dec. 2011, pp. 693–701.

[38] X. Pan et al., "Cyclades: Conflict-free asynchronous machine learning," in *Proc. Neural Inf. Process. Syst. (NIPS)*, Barcelona, Spain, Dec. 2016, pp. 2568–2576.

[39] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. Heaphy, and L. A. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Long Beach, CA, USA, Mar. 2007, pp. 1–11.

[40] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Salt Lake City, UT, USA, Nov. 2016, pp. 678–689.

[41] S. A. Javadi and A. Gandhi, "DIAL: Reducing tail latencies for cloud applications via dynamic interference-aware load balancing," in *Proc. IEEE Int. Conf. Autonomic Comput. (ICAC)*, Columbus, OH, USA, Jul. 2017, pp. 135–144.
[42] A. K. Maji, S. Mitra, and S. Bagchi, "ICE: An integrated configuration engine for interference mitigation in cloud services," in *Proc. IEEE Int. Conf. Autonomic Comput. (ICAC)*, Grenoble, France, Jul. 2015, pp. 91–100.
[43] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro, "Robust stochastic approximation approach to stochastic programming," *SIAM J. Optim.*, vol. 19, no. 4, pp. 1574–1609, Jan. 2009.
[44] A. Auton et al., "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, pp. 68–74, Oct. 2015.
[45] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature Commun.*, vol. 5, pp. 4308:1–4308:9, Jul. 2014.
[46] J. Langguth, X. Cai, and M. Sourouri, "Memory bandwidth contention: Communication vs computation tradeoffs in supercomputers with multicore architectures," in *Proc. IEEE Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Singapore, Dec. 2018, pp. 497–506.
[47] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. 10th Eur. Conf. Comput. Syst. (EuroSys)*, Bordeaux, France, Apr. 2015, pp. 1–17.
[48] M. Lukša, *Kubernetes in Action*. Shelter Island, NY, USA: Manning, 2017.
[49] P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and implementation of an efficient priority queue," *Math. Syst. Theory*, vol. 10, no. 1, pp. 99–127, Dec. 1976.



Albin Severinson received the M.Sc. degree in communications engineering from Chalmers University of Technology, Gothenburg, Sweden, in 2017, and the Ph.D. degree from the Department of Informatics, University of Bergen, Bergen, Norway, under the supervision of Dr. Eirik Rosnes and Prof. Alexandre Graell i Amat. He is currently with the quantitative research and technology firm G-Research, London, U.K. He successfully defended his thesis in 2022.

Eirik Rosnes (Senior Member, IEEE) was born in Stavanger, Norway, in 1975. He received the Cand. Scient. degree in physics and the Dr. Scient. degree in informatics from the University of Bergen, Bergen, Norway, in 1999 and 2003, respectively.

From September 2001 to March 2002 and from March 2005 to September 2005, he was a Visiting Scholar with the Center for Magnetic Recording Research, University of California at San Diego, La Jolla, CA, USA. From 2003 to 2011, he was with the Department of Informatics, University of Bergen, first as a Post-Doctoral Researcher and then as a Senior Researcher. From October 2011 to August 2013, he was a Senior Engineer with Ceragon Networks, and since August 2013, he has been with both Simula Research Laboratory/Simula UiB as a Senior Researcher (from 2020 as a Chief Research Scientist) and a Section Leader, and the University of Bergen (until August 2017) as an Adjunct Associate Professor. His research interests include communication theory and information theory, classical error-control coding, codes on graphs, codes for distributed storage systems, and coding for privacy and security. He was a Technical Program Co-Chair of the 2009 International Workshop on Coding and Cryptography, Ullensvang, Norway, and served as an Area Editor for the AEU *International Journal of Electronics and Communications* from October 2014 to January 2020. He is currently serving as an Associate Editor for the IEEE TRANSACTIONS ON COMMUNICATIONS.

Salim El Rouayheb (Senior Member, IEEE) received the Diploma degree in electrical engineering from the Faculty of Engineering, Lebanese University, Roumieh, Lebanon, in 2002, the M.Sc. degree from the American University of Beirut, Lebanon, in 2004, and the Ph.D. degree in electrical engineering from Texas A&M University, College Station, TX, USA, in 2009. He was a Post-Doctoral Research Fellow with UC Berkeley from 2010 to 2011, and a Research Scholar with Princeton University from 2012 to 2013. He was an Assistant Professor with the Department of Electrical and Computer Engineering, Illinois Institute of Technology, from 2013 to 2017. He is currently an Associate Professor with the Department of Electrical and Computer Engineering, Rutgers University, New Brunswick, NJ, USA. His research interests include information theory and coding theory with applications to reliability, security, and privacy in distributed systems. He was a recipient of the NSF Career Award.

Alexandre Graell i Amat (Senior Member, IEEE) received the M.Sc. degree in electrical engineering from the Politecnico di Torino, Turin, Italy, in 2000, the M.Sc. degree in telecommunications engineering from the Universitat Politècnica de Catalunya, Barcelona, Catalonia, Spain, in 2001, and the Ph.D. degree in electrical engineering from the Politecnico di Torino, in 2004.

From 2001 to 2002, he was a Visiting Scholar with the University of California at San Diego, La Jolla, CA, USA. From 2001 to 2004, he held a part-time appointment at the STMicroelectronics Data Storage Division, Milan, Italy, as a Consultant on coding for magnetic recording channels. From 2002 to 2003, he held a Visiting Appointment at Universitat Pompeu Fabra, Barcelona, and the Telecommunications Technological Center of Catalonia, Barcelona. From 2004 to 2005, he was a Visiting Professor with Universitat Pompeu Fabra, Barcelona. From 2006 to 2010, he was with the Department of Electronics, IMT Atlantique (formerly ENST Bretagne), Brest, France. Since 2019, he has been an Adjunct Research Scientist at Simula UiB, Bergen, Norway. He is currently a Professor with the Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, Sweden. His research interests include coding theory and its application to distributed learning and computing, storage, privacy and security, and communications. He received the Marie Skłodowska-Curie Fellowship from the European Commission and the Juan de la Cierva Fellowship from the Spanish Ministry of Education and Science. He received the IEEE Communications Society 2010 Europe, Middle East, and Africa Region Outstanding Young Researcher Award. He was a General Co-Chair of the 7th International Symposium on Turbo Codes and Iterative Information Processing, Sweden, in 2012, and a TPC Co-Chair of the 11th International Symposium on Topics in Coding, Canada, in 2021. He was an Associate Editor of the IEEE COMMUNICATIONS LETTERS from 2011 to 2013. He was an Associate Editor and an Editor-at-Large of the IEEE TRANSACTIONS ON COMMUNICATIONS from 2011 to 2016 and from 2017 to 2020, respectively. He is also an Area Editor of the IEEE TRANSACTIONS ON COMMUNICATIONS.