Advances in Asynchronous Parallel and Distributed Optimization

Mahmoud Assran, Arda Aytekin, Hamid Reza Feyzmahdavian, Mikael Johansson, and Michael G. Rabbat

***Abstract*—Motivated by large-scale optimization problems arising in the context of machine learning, there have been several advances in the study of asynchronous parallel and distributed optimization methods during the past decade. Asynchronous methods do not require all processors to maintain a consistent view of the optimization variables. Consequently, they generally can make more efficient use of computational resources than synchronous methods, and they are not sensitive to issues like stragglers (i.e., slow nodes) and unreliable communication links. Mathematical modeling of asynchronous methods involves proper accounting of information delays, which makes their analysis challenging. This article reviews recent developments in the design and analysis of asynchronous optimization methods, covering both centralized methods, where all processors update a master copy of the optimization variables, and decentralized methods, where each processor maintains a local copy of the variables. The analysis provides insights as to how the degree of asynchrony impacts convergence rates, especially in stochastic optimization methods.**

I. Introduction

Since the slowing of Moore's scaling law, parallel and distributed computing have become a primary means to solve large computational problems. Much of the work on parallel and distributed optimization during the past decade has been motivated by machine learning applications. The goal of fitting a predictive model to a dataset is formulated as an optimization problem that involves finding the model parameters that provide the best predictive performance. During the same time, advances in machine learning have been enabled by the availability of ever larger datasets and the ability to use larger models, resulting in optimization problems potentially involving billions of free parameters and billions of data samples [1]–[3].

There are two general scenarios where the use of parallel computing resources naturally arises. In one scenario, the data is available in one central location (e.g., a data center), and the aim is to use parallel computing to train a model faster than would be possible using serial methods. The ideal outcome is to find a parallel method that achieves *linear scaling*, where the time to achieve a solution of a particular quality decreases proportionally to the number of processors used; i.e., doubling the number of parallel processors reduces the compute time by half. However, unlike serial methods, parallel optimization methods generally require coordination or communication among multiple processors.

In the second scenario, which is receiving increasing interest, the data is widely distributed (e.g., residing on users' devices), and the goal is to train a model using all of the data without collecting it in one location. The motivation to process the data in a distributed way may be for privacy reasons, and it may also be too expensive (in terms of communication time and bandwidth) to communicate the data. Although this survey primarily focuses on the first scenario, many of the results and methods discussed can be readily applied in the second.

Parallel and distributed algorithms may be classified as synchronous or asynchronous. Synchronous algorithms require that the processors *serialize* after every update, so that every processor always has a consistent view of optimization variables. This generally makes serial algorithms easier to analyze, implement, and debug, and consequently synchronous algorithms have been more widely studied. However, synchronization may also lead to poor utilization of computational resources. If one processor is slower than the others, they must all wait *idling* at the serialization point until the slowest processor catches up, resulting in wasted compute cycles.

Asynchronous algorithms do not impose a global synchronization point at the end of each iteration. Consequently, each processor may have a different view of the optimization variable when performing local computations. Properly accounting for these inconsistencies complicates the analysis of asynchronous iterative algorithms. Also, because their execution is non-deterministic, asynchronous algorithms can be challenging to implement and debug. The appeal of asynchronous methods is that they indeed can make more efficient use of computational resources, resulting in faster wall-clock convergence.

A. Historical context

The dynamics of asynchronous iterations are much richer than their synchronous counterparts, and quantifying the impact of asynchrony on the convergence times of iterative algorithms is challenging. Some of the first results on the convergence of asynchronous iterations were derived by Chazan and Miranker [4] who were motivated by encouraging empirical results for parallel and asynchronous linear equation solvers [5]. Their theoretical results considered linear iterations under bounded asynchrony. Several authors have extended this work to nonlinear iterations involving maximum norm contractions (e.g., [6]) and for monotone iterations (e.g., [7]). Powerful convergence results for broad classes of asynchronous algorithms, including maximum norm contractions and monotone mappings, under different assumptions on communication delays and update rates were presented by Bertsekas [8], Tsitsiklis et al. [9], and in the celebrated book of Bertsekas and Tsitsiklis [10]. An important insight in this line of work is that asynchrony can be modelled as time-varying update rates and information delays with respect to a global ordering of events in the system.

The framework of [10] defines two models of asynchrony: *totally asynchronous* and *partially asynchronous algorithms*.

In totally asynchronous algorithms the information delays may grow arbitrarily large, and therefore the best one can expect is asymptotic convergence. In partially asynchronous algorithms, both inter-update times and information delays remain bounded and algorithms may converge to a target accuracy in finite time.

Although the framework for modeling asynchronous algorithms in [10] is both powerful and elegant, the most concrete results consider totally asynchronous iterations. In particular, pseudo-contractions in the block-maximum norm are shown to converge when executed in a totally asynchronous manner [10, Section 6.2]; however, in machine-learning applications, first-order methods rarely result in maximum-norm contractions. For example, for convex quadratic optimization problems, the gradient descent iterations are maximum norm contractions only if the Hessian is diagonally dominant. The asymptotic nature of these results also means that they do not characterize how the amount of asynchrony impacts convergence times.

In contrast, we will show below that many important optimization algorithms for machine-learning tolerate some level of asynchrony and that it is possible to quantify how asynchrony affects the number of iterations required to find a solution with a given target accuracy. Such results provide important insight into engineering trade-offs between the longer iteration times in synchronous systems and additional (but faster) iterations in asynchronous implementations. A challenge comes from the fact that many optimization methods used for deep learning are inherently stochastic; e.g., they use sampling to approximate the gradient when determining the direction in which to update the model parameters. The results available in [10] for partially asynchronous methods are difficult to apply in the stochastic setting, where randomized update orders and workloads affect update frequencies and computation times. To address these challenges, a new wave of research on analysis and design of asynchronous and distributed optimization algorithms has emerged, and significant theoretical and technological advances have recently been made.

B. This article

This article surveys advances in the field of asynchronous and parallel distributed optimization made in the past decade. Since the work we survey is mainly motivated by applications from machine learning, we review background material on this topic in Section II. Then, in Section III, we introduce necessary background on parallel and distributed computing systems.

The parallel and distributed optimization methods we survey can be divided into two categories. *Centralized* methods, which we discuss in Section IV, maintain one master copy of the optimization variables. In one iteration of these methods, a processor reads the master variables, performs a local computation, and then updates the master copy. However, between the time a processor reads the master variables and then updates them, other processors may have already performed updates. In extreme cases, the master variables may be updated simultaneously by multiple processors while they are being read by others.

Decentralized methods, discussed in Section V, form the second category. In decentralized methods there is no master copy; rather, each processor maintains a local copy of the optimization variables. Processors update their local copies and synchronize them with other processors directly. These methods are typically implemented on distributed-memory systems, although they may also be applicable in large shared-memory systems with non-uniform memory access, where the time it takes a given processor to access different locations in memory depends on the physical distance between the memory location and the processor.

Throughout this article we focus on first-order methods, which only make use of gradient information, since they are the most widely-used methods for training machine learning models today [11], [12]. We also focus on stochastic optimization methods, where gradient information is obtained by sampling a subset of data points. Newton-type methods, which make use of second-order derivatives and curvature, have not received wide adoption because they involve computing and storing the Hessian matrix or an approximation thereof, which requires significant computation and memory when the problem dimension is large, as is typical of many machine learning problems. Also, Newton-type methods tend to be more sensitive to noise and stochasticity. Nevertheless, recent work explores the viability of stochastic Newton-type methods in centralized [13] and distributed [14] settings.

When discussing both centralized and decentralized methods, we survey existing algorithms and their convergence guarantees, and we also discuss the main analysis techniques. We emphasize results for the partially asynchronous model, where information delays are bounded. In Section V we include a numerical example illustrating how asynchronous decentralized algorithms may be used for training deep neural-networks. We conclude in Section VI with a discussion of open problems and directions for future work.

C. Other Applications

Throughout the rest of this article, we will use example applications from machine learning to illustrate asynchronous optimization methods. We note in passing that asynchronous parallel and distributed optimization methods are also relevant to a variety of other applications, including the operation of distributed infrastructure systems such as power systems [15] and water distribution, the internet-of-things (IoT), smart grids, networks of autonomous vehicles, and wireless communication networks [16].

II. Background

A. Optimization

This article focuses on optimization methods to solve the problem

$$\underset{w\in\mathbb{R}^d}{\text{minimize}} \quad f(w)\,. \tag{1}$$

A point $w \in \mathbb{R}^d$ is called a local minimizer of the objective function f if $f(w) \leq f(w')$ for all other points w' in a neighborhood of w, and $w^\star$ is called a global minimizer if $f(w^\star) \leq f(w')$ for all $w' \in \mathbb{R}^d$.

Serial iterative optimization methods start from an initial point $w^{(0)}$ and generate a sequence $w^{(1)}, w^{(2)}, \dots$ by performing update steps. The performance of an optimization method is measured in terms of the number of steps required to reach a solution of a certain quality. Performance guarantees depend on the particular method, the assumed characteristics of the objective function f, and how it can be evaluated.

Update steps, going from $w^{(k)}$ to $w^{(k+1)}$, typically make use of the (negative) gradient $-\nabla f(w^{(k)})$ at $w^{(k)}$, which points in the direction that decreases the objective function f in a small neighborhood around $w^{(k)}$. The objective function is said to be *smooth* if the gradient changes gradually; formally, there is a constant $L > 0$ such that

$$\big\|\nabla f(w) - \nabla f(w')\big\| \leq L\big\|w - w'\big\|$$

for all points $w, w' \in \mathbb{R}^d$. The smaller L, the smoother the function f, and consequently first-order optimization methods can take larger steps while still ensuring convergence. A point $w \in \mathbb{R}^d$ is called a *stationary point* if $\nabla f(w) = 0$; in this case the gradient does not point in any direction, so w may be a (local or global) minimizer. Non-smooth problems are more challenging because even a small change in parameter space may result in a drastic change in the gradient direction.

An important class of optimization problems is related to the notion of convexity. Formally, f is convex if for any $\alpha \in (0, 1)$ it holds that

$$f(\alpha w + (1-\alpha)w') \leq \alpha f(w) + (1-\alpha) f(w').$$

When f is convex, every stationary point of f is a global minimizer of f, and so iterative methods that converge to stationary points are guaranteed to find a global minimizer. In general, there may be many points that minimize f. When $f(w) - \frac{\mu}{2}\|w\|_2^2$ is also convex, for some parameter $\mu > 0$, we say that f is *strongly convex*, and we refer to μ as the strong convexity parameter. Strongly convex functions have a unique global minimizer that is also the unique stationary point of f, and they are also easier to optimize than functions which are only convex. We refer the interested reader to [17] for details.

B. Machine Learning

Machine learning methods train a parameterized model to make predictions on a set of data with the goal that the model makes accurate predictions on never-before seen data. For example, in an image classification task, the model is shown an image and asked to predict which class, among a finite but possibly large set of classes, best describes the image content (e.g., dog, cat, human, ...). Similarly, in a document classification task, the model is given a text and asked to predict which class best describes its content (e.g., in which section of the newspaper the text appeared).

Training of such models is typically formulated as an optimization problem on the form (1) with

$$f(w) = \frac{1}{m}\sum_{i=1}^{m} f_i(w) + r(w) \tag{2}$$

and $f_i(w) = \ell(p(x_i; w), y_i)$. Here, $\{(x_i, y_i)\}_{i=1}^m$ denotes a collection of m training samples, each consisting of an input x_i and target y_i. The goal is to learn the parameters w of a model $p(x; w)$ so that $p(x_i; w)$ matches y_i well on the training data. The loss function ℓ measures how well a prediction $p(x_i; w)$ matches the target y_i. When $d > m$ or if the model class $\{p(x; w) : w \in \mathbb{R}^d\}$ is otherwise rich/expressive, one may use a regularization function $r(w)$ to avoid over-fitting the training data. The regularization function can also be used to impose other constraints on the model parameters w.

This framework can describe a variety of common machine learning settings. For example, for a binary classification problem where $y_i \in \{0, 1\}$, using the model $p(y_i = 1|x_i; w) = \big(1 + \exp(-\langle w, x_i\rangle)\big)^{-1}$, loss $\ell(p, y_i) = -y_i \log(p) - (1 - y_i)\log(1-p)$, and regularizer $r(w) = \frac{1}{2}\|w\|_2^2$ corresponds to the popular ℓ_2-regularized logistic regression method [18, Section 8.3]; the resulting optimization problem is smooth and strongly convex. Other methods, including support vector machines, least-squares regression, and sparsity-inducing ℓ_1-regularized methods can all be cast in this framework as convex optimization problems [19].

Deep neural networks (DNNs) currently achieve state-of-the-art performance for the majority of machine learning tasks. The details of the DNN models $p(x; w)$ and loss functions ℓ are beyond the scope of this paper; the interested reader is referred to [20]. From the view of asynchronous distributed optimization, training DNNs also fits into the framework of (2), but the resulting optimization problem is typically not convex.

C. Optimization Methods for Large Scale Learning

An optimization problem with objective (2) can become "large-scale" in a few different ways: the number of training samples m can be large, the dimension of the training inputs x and/or targets y can become large, and the dimension d of the optimization variable can be large. To handle the large m setting, it is common to select a smaller subset of samples at each update. In some cases, to handle the large d setting one may also use coordinate descent methods that only update a subset of the coordinates at each update.

The iterative methods we consider all have the following general form: given an initial point $w^{(0)}$, repeat for $k \geq 0$,

$$w^{(k+1)} = w^{(k)} - \gamma^{(k)} s^{(k)}, \tag{3}$$

where $\gamma^{(k)} \in \mathbb{R}_+$ is a step-size, and $s^{(k)} \in \mathbb{R}^d$ is a search (or update) direction. The full gradient[1] of the objective f in (2) may be expensive to compute (e.g., if m or d is large), so we focus on algorithms that approximate the gradient $\nabla f(w^{(k)})$ by a quantity which is easier to compute and which can be efficiently computed in parallel.

D. Stochastic Gradient Descent

To simplify the discussion for now, suppose that there is no regularizer; i.e., $r(w) = 0$. The classical *stochastic gradient descent* (SGD) method [12], [21] considers random search directions $s^{(k)}$ which are equal to the gradient in expectation

[1]To simplify the discussion throughout, we assume the loss and regularization functions are continuously differentiable, and will talk about gradient methods. We make specific remarks about modifications to handle non-smooth functions below.

and have bounded second moment, i.e., $\mathbb{E}[s^{(k)} \mid w^{(k)}] = \nabla f(w^{(k)})$ and $\mathbb{E}[\|s^{(k)} - \nabla f(w^{(k)})\|^2 \mid w^{(k)}] \leq \sigma^2$, where $\sigma^2 < \infty$ is assumed to be given.

For optimization problems of the form (2), one natural way to obtain random search directions is to use

$$s^{(k)} = \nabla f_{i^{(k)}}(w^{(k)}), \tag{4}$$

where $i^{(k)}$ is drawn uniformly at random from $\{1, \ldots, m\}$. By only evaluating the gradient of a single component function, the computational cost per iteration is effectively reduced by a factor m.

The SGD method (3)–(4) is inherently serial: the gradient computations take place on a single processor which needs access to the whole dataset. The desire to have faster methods for training larger models on larger datasets has resulted in a strong interest in developing parallel optimization algorithms that are able to split the data and distribute the computation across multiple processors or multiple servers.

A common practical solution for parallelizing stochastic gradient methods and reducing the stochastic variance is to employ mini-batches. Mini-batch SGD method evaluates a subset $I^{(k)} \subseteq \{1, \ldots, m\}$ of b component gradients in parallel and uses the search direction

$$s^{(k)} = \frac{1}{b} \sum_{i \in I^{(k)}} \nabla f_i(w^{(k)}). \tag{5}$$

If we have $n \leq b$ processors working in parallel, the hope is that we can compute $s^{(k)}$ roughly n times faster than a single processor. In addition, by averaging b component gradients, we reduce the variance of the search direction by a factor of $1/b$ per iteration. These factors combine to allow parallel mini-batch SGD algorithms to achieve near-linear speedup in the number of compute nodes [22].

However, the parallel SGD method as described above is *synchronous*: the compute nodes all read the current decision vector $w^{(k)}$, evaluate the gradient of their assigned component function(s) at $w^{(k)}$, and then update the current iterate. Once all b gradient updates have been applied to the decision vector, the algorithm can proceed to the next iteration.

Below we will discuss a variety of ways in which methods of the form (3) can be made to run asynchronously. First, we review key aspects of parallel and distributed computing architectures, that will inform the subsequent discussion.

III. Architectures

Advances in computing hardware and communication infrastructures along with the emergence of virtualization and container technologies have enabled a multitude of options for affordable large-scale computing. High-performance computing (HPC) environments traditionally available only at supercomputing centers are now easily accessible as commoditized cloud services provided by many companies. Similarly, hardware accelerators such as tensor processing units (TPUs), general-purpose graphics processing units (GPUs) and multi-core central processing units (CPUs), together with generous access to high-bandwidth memory and storage have increased the compute capabilities of traditional servers by many orders of magnitude. Finally, enabling technologies such as the 5G make it possible to deploy and interconnect low-powered compute nodes to jointly collect and process data on the *edge*.

Obtaining the best possible performance from such compute resources relies on our ability to parallelize the computations across multiple computational units (cores, devices, clouds). Different compute resources can work concurrently using their local copies of the model parameters and local datasets to speed up the computation of (stochastic) gradients. However, simultaneously updating the shared model parameters in (3) using these local gradients results in undefined behaviour, and thus, has to be *serialized*. There are two different approaches to serializing simultaneous updates. *Synchronous* approaches mandate that *all* compute nodes arrive at a *barrier* to exchange their local updates before proceeding. This makes it possible for all the nodes to have the same view over the shared parameters at all times as if the algorithm is running *serially*. The main disadvantage of this approach is that the overall performance of the algorithm depends on the *slowest* compute node. *Asynchronous* approaches, on the other hand, let the compute nodes update the shared parameters at their own pace. This makes it possible for faster nodes to progress *without* needing to wait for the slower nodes, which results in better performance. However, the main challenge in this setting is to incorporate the stale updates coming from slower nodes to the shared model parameters.

We can categorize parallel computing architectures into two classes based on how the simultaneous updates are serialized. *Shared-memory architectures* span compute resources such as many-core CPUs and hardware accelerators, which have many compute nodes that share the same physical memory space (Figure 1, left). Even though the nodes share the same physical memory, non-uniform memory access (NUMA) designs and deep cache hierarchies in today's architectures invalidate the assumption that nodes have immediate access to a memory region (see, e.g., [23]–[28] that revisit algorithms and take this issue into account). In shared-memory architectures, all the serialization primitives are in the same physical space. Synchronous operations are usually implemented using *semaphores*, where nodes signal their presence and proceed with their next task only after every other node has also arrived. Asynchronous operations, on the other hand, are usually implemented in one of two ways: by mutual exclusion (*mutex*) locks or *atomic operations*. Mutex locks are generally used to protect the shared model parameters *as a whole* during simultaneous accesses, which results in *consistent* views over the parameters. Atomic operations, on the other hand, protect the *individual* elements of the shared parameters, and thus, result in *inconsistent* views over the parameters as a whole.

To better understand this, let us consider a scenario when two nodes are trying to update the shared parameter vector $w = [0, 0, 0]^\top$ while another one is trying to read it, all at the same time (see Figure 2). At the top, we observe the case when nodes acquire the lock, one by one (in a sequential order in the example) before attempting their task (updating or reading) and release the lock afterwards. Because the second node acquires the lock after the first node has finished updating, it reads the new parameter as $w_2 = [1, 0, 1]^\top$. At the bottom, we observe

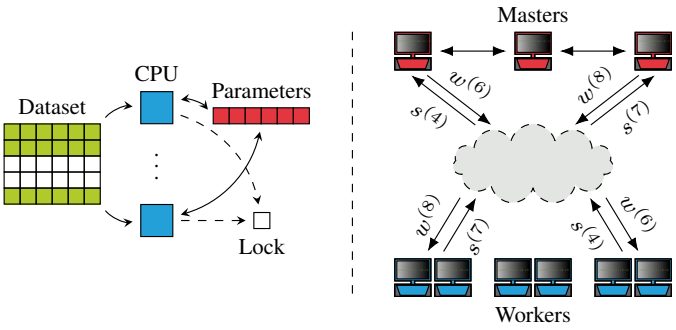


Figure 1. (Left) Shared-memory architecture where multiple CPUs read simultaneously from a dataset and update the parameters by obtaining a lock. (Right) Distributed-memory architecture where shared parameters are kept in centralized masters, and workers send their updates asynchronously.

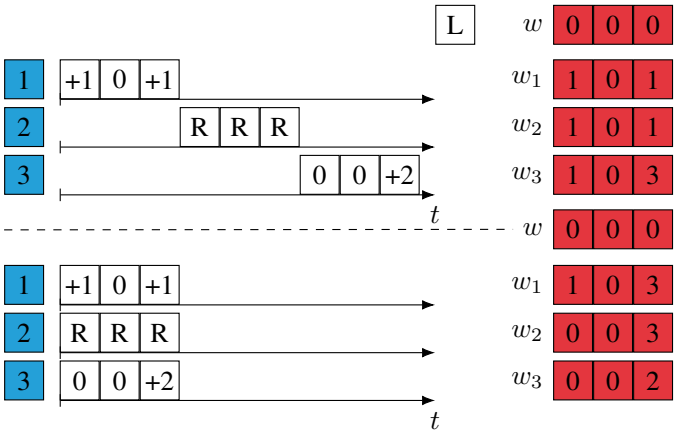


Figure 2. (Top) Serializing access to shared parameters using a mutex lock (L). (Bottom) Serializing access to individual elements using atomic operations. Different compute nodes' local updates are shown on the time axis for visualization purposes. Only the non-zero updates are applied, and thus, take time.

the case when individual coordinates of the parameter vector are updated/read using atomic operations. In this case, the first two nodes are accessing the first coordinate atomically at the same time (with a load before store ordering in the example) whereas the third node is updating the third coordinate. Later, the second node is reading the second coordinate while the first one is updating the third coordinate. As a result, the second node has an inconsistent view over the parameter vector (i.e., the vector, $w_2 = [0, 0, 3]^\top$, read by the second node would have never existed during the update sequence). It is worth noting that, in this example, using mutex locks takes more time (six units of time, discarding prefetching and caching effects) than using atomic operations (roughly three units of time). In fact, this observation is true when the updates are *sparse*. Atomic operations provide faster serialization (at the expense of inconsistent views) when fewer nodes are competing for the same blocks of coordinates whereas mutex locks provide more efficient serialization when the underlying operations are more expensive (dense updates).

Distributed-memory architectures cover networks of computers, IoT-enabled edge devices and modern computer setups in which CPUs and accelerators work together but have different physical memory spaces. Figure 1 (right) shows an example of a network of computers in a particular communication topology. In this setup, also known as the *parameter server* setup [29], [30], the communication is *centralized* around a set of nodes (called the *masters* or the *servers*) that keep the shared parameters and constitute the hubs of a star network. *Worker* nodes (or *clients*) pull the shared parameters from and send their updates to the central nodes.

When there are no masters in the setup, and the nodes are allowed to communicate with each other in a more general way, it becomes a *multi-agent* setup. In this setup, updating the shared parameter vector is *decentralized* among the participating *agents* while obeying their respective communication topologies. The main challenge in distributed-memory architectures, as opposed to shared-memory, is that access to data in another node's memory space requires some sort of message passing over network sockets. This, in turn, makes serialization and synchronization operations rather expensive. Synchronous operations use blocking communication primitives in the sense that all participating nodes have to wait for a message before proceeding. Depending on the communication constraints and topology, this can yield different communication complexities. For instance, in a ring topology in which only point-to-point communication is allowed, and nodes are queried in a round-robin fashion, we have $\mathcal{O}(n)$ communication complexity. On the other hand, if many-to-many or all-to-all communication is allowed, naïve implementations offer $\mathcal{O}\big(n^2\big)$ complexities whereas collective communication operations (such as broadcast and reduce) that use a butterfly-like communication pattern achieve the optimal $\mathcal{O}\big(\log(n)\big)$ [31]. Nevertheless, even the optimal synchronous operations still suffer from the *deadlock* (a situation in which all the nodes are waiting for an output of each other or of a dead/offline node) and *straggler* (node that is slow due to either computation or communication performance) problems, especially when messages are delivered slowly or may be lost altogether.

To alleviate these problems, asynchronous operations are preferred, although, this time, it becomes harder to design and analyze algorithms. In asynchronous operations, nodes can interleave communication and computation based on their own pace; yet, they have to deal with not only delayed information over the parameter vector, when there is one writer and multiple readers (e.g., one master node in a parameter server), but also inconsistent views over the vector when the vector is shared among multiple masters.

IV. Centralized Asynchronous Algorithms

Recall that our goal is to minimize an objective function of the form (2), and let us again suppose there is no regularizer, so that

$$f(w) = \frac{1}{m} \sum_{i=1}^{m} f_i(w).$$

In this section, we discuss asynchronous iterative methods for minimizing f that involve updating a master copy of the optimization variables $w^{(k)}$.

The serial SGD method discussed in Section II-D performs updates of the form

$$w^{(k+1)} = w^{(k)} - \gamma^{(k)} s^{(k)}$$

with search direction $s^{(k)} = \nabla f_{i^{(k)}}(w^{(k)})$ where $i^{(k)}$ is sampled uniformly at random between 1 and m; i.e., the gradient of f is approximated as the gradient at one of the data points.

Although the search direction in SGD is an unbiased estimator of ∇f, its variance (and higher moments) are typically non-zero, which limits the achievable solution accuracy. Specifically, if f is μ-strongly convex and each f_i is L-smooth, then iterates of SGD for a fixed step-size $\gamma^{(k)} = \gamma \in \left(0, \frac{1}{L}\right)$ satisfy

$$\mathbf{E}[\|w^{(k)} - w^\star\|^2] \leq \left[1 - 2\gamma\mu(1-\gamma L)\right]^k \|w^{(0)} - w^\star\|^2 + \frac{\gamma\sigma^2}{\mu(1-\gamma L)},$$

where $w^\star = \arg\min_w f(w)$, and σ^2, defined as

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} \left\|\nabla f_i(w^\star)\right\|^2,$$

is the variance of the search direction at the optimum [32, Theorem 2.1]. The first term of the SGD theoretical upper bound decays to zero at a linear rate, while the second term is a constant and describes the residual error. This means that SGD with a fixed step-size γ converges linearly to a neighborhood of the optimum whose radius is proportional to σ^2 and γ. Decreasing γ reduces the residual error, but it also results in a slower convergence. For any desired accuracy $\varepsilon > 0$, letting

$$\gamma = \frac{\mu\varepsilon}{2\mu\varepsilon L + 2\sigma^2}$$

ensures that $\mathbf{E}[\|w^{(k)} - w^\star\|^2] \leq \varepsilon$ for all iterations

$$k \geq 2\left(Q + \frac{\sigma^2}{\mu^2\varepsilon}\right)\log(\varepsilon^0/\varepsilon),$$

where $\varepsilon^0 = \|w^{(0)} - w^\star\|^2$, and $Q = L/\mu$ is the condition number of the function f. This expression shows how a large value of σ^2 forces a small step-size γ to reach ε-accuracy, and therefore results in long convergence times.

It is common for the iteration complexity, and for optimal or allowable choices of algorithm parameters, to depend on the problem constants L and μ. In practice, these may not be directly known, and one may use an upper bound on L and a lower bound on μ instead. An upper bound on the gradient Lipschitz constant L can be easily obtained during the execution of an algorithm by tracking the ratio $\left\|\nabla f(x_{k+1}) - \nabla f(x_k)\right\| / \|x_{k+1} - x_k\|$. This leads to methods which resemble a backtracking line search, such as those described in [33], [34], to upper bound the allowable step sizes. Note, however, that exactly tracking the aforementioned ratio requires the evaluation of the full gradient, which might not be plausible in the asynchronous setting. In such scenarios, techniques such as that in [35] could be used to estimate upper bounds on the allowable step sizes *without* requiring the full gradient evaluation. Estimating a lower bound on μ is much more challenging. One approach for estimating a lower bound on μ is provided in [36]. However, when using an ℓ_2 regularizer of the form $r(w) = (\lambda/2)\|w\|_2^2$, then λ directly provides a lower bound on μ.

A. Asynchronous parallel stochastic gradient methods

Recall that the mini-batch SGD method, which uses the search direction (5), can leverage multiple processors to compute gradient terms in parallel, but it is inherently synchronous: processors read the current decision vector, compute a gradient of their assigned component functions, and update the iterate. Once all b gradient updates have been performed on the decision vector, the algorithm proceeds to the next iteration.

As suggested in [37], this process can be pipelined. In such a realization, a master node interacts with worker nodes in a round-robin fashion, collecting their most recent stochastic (mini-batch) gradient, updating the decision vector and returning the new iterate to the worker before proceeding to serve the next worker. In a system with n workers, each worker evaluates a stochastic gradient on a decision vector which is $\tau = n$ iterates old, but can use a local mini-batch which is $\mathcal{O}(n)$ times larger and still finish its work before the next chance to interact with the master node. In this algorithm, the decision vector is thus updated using the search direction

$$s^{(k)} = \frac{1}{b}\sum_{i \in I^{(k)}} \nabla f_i(w^{(k-\tau)}).$$

Interestingly, the constant delay τ introduces negligible penalty in the convergence rate of the algorithm [37].

The round-robin interaction can be problematic in practice if some workers struggle to finish their work in time. The master may then have to wait, accept a suboptimal search direction from the worker, or skip its update altogether. As shown in [38], however, the same performance can be attained by an asynchronous parallel mini-batch algorithm which avoids global synchronization and allows worker nodes to read and write back to the master at their own pace. In this case, the search direction used in the update rule of the algorithm is given by

$$s^{(k)} = \frac{1}{b}\sum_{i \in I^{(k)}} \nabla f_i(w^{(k-\tau^{(k)})}),$$

where $\tau^{(k)}$ is a time-varying delay capturing the staleness of the information used to compute the search direction for the k^{th} update. In [39], it was shown that if $\tau^{(k)}$ is bounded, so that $\tau^{(k)} \leq \tau_{\max}$ for all k, then the iteration complexity of the asynchronous parallel mini-batch algorithm for strongly convex smooth optimization is given by

$$\mathcal{O}\left(\left((\tau_{\max}+1)^2 Q + \frac{\sigma^2}{\mu^2\varepsilon}\right)\log(1/\varepsilon)\right).$$

In practice, $\tau_{\max}$ will depend on the number of parallel processors used for implementation of the algorithm. As long as $\tau_{\max}$ is of the order $1/\sqrt{\varepsilon}$, the iteration complexity of the asynchronous algorithm is asymptotically $\mathcal{O}((1/\varepsilon)\log(1/\varepsilon))$, which is exactly the iteration complexity achieved by serial SGD. This means that the delay becomes increasingly harmless as the

asynchronous algorithm progresses. Furthermore, as n workers are being run asynchronously and in parallel, updates may occur roughly n times as quickly, which means that a near-linear speedup in the number of workers can be expected.

When the decision vector dimension is very large, reading or writing the full vector takes considerable time. In a shared-memory system, it is then ineffective to lock the full vector during memory access. Instead, one typically only protects individual entries by using atomic read and write operations. This process is even more efficient if gradients are sparse and tend to have non-overlapping support. The probability that different workers simultaneously attempt to access the same elements of the decision vector is then low, and the n workers effectively run independently in parallel.

The first analysis of such a "lock-free" SGD algorithm, called HOGWILD!, appeared in [40]. To describe the convergence results, we need to introduce some additional notation. In particular, let E_i be the support of $\nabla f_i(x)$ and define

$$\Delta = \frac{\max\limits_{j=1,\ldots,d} |\{i : j \in E_i\}|}{m}.$$

The parameter $\Delta \in [\frac{1}{m}, 1]$ is a measure of the sparsity for the optimization problem. In a fully dense dataset, Δ is equal to 1 and in a completely sparse dataset, Δ is equal to $1/m$.

HOGWILD! lets each core run its own SGD iterations without any attempt to coordinate or synchronize with other cores, repeating the following steps:

- Use atomic read operations to copy the shared decision variable w into a local variable $\widehat{w}$.
- Sample a component function f_i and use $\widehat{w}$ to compute $s = \nabla f_i(\widehat{w})$.
- Use atomic write operations to update the current w in shared memory

$$[w]_e \leftarrow [w]_e - \gamma [s]_e, \qquad \text{for } e \in E_i.$$

  Note that for the write operation, only elements in the support of ∇f_i need to be updated.

During the execution of HOGWILD!, processors do not synchronize or follow an order between reads or writes. This implies that while one processor is evaluating its gradient, others may update the value of w stored in the shared memory. Therefore, the value $\widehat{w}$ at which the gradient is calculated by a processor may differ from the value of w to which the update is applied. Note also that a full vector $\widehat{w}$ read for a processor might not correspond to any state of w in the shared memory at any time point (cf. Figure 2, bottom).

It was shown in [41] that the iteration complexity of HOGWILD! for smooth strongly convex optimization is

$$\mathcal{O}\left(\left((\sqrt{\Delta}\tau_{\max}+1)Q + \frac{\sigma^2}{\mu^2\varepsilon}\right)\log(1/\varepsilon)\right),$$

where $\tau_{\max}$ is the maximum delay between reading and updating for cores. It follows that when $\tau_{\max} = \mathcal{O}\left(\frac{1}{\sqrt{\Delta}}\right)$, HOGWILD! converges at the same rate as the serial SGD and therefore enjoys near-linear speedup.

B. Variance reduction and incremental aggregation methods

A drawback with constant step-size SGD algorithms, including the parallel and asynchronous variants discussed above, is that the iterates $\{w^{(k)}\}$ will not converge to an optimizer $w^\star$, but will exhibit a residual error. This error, which arises due to the mismatch between the gradients of individual component functions and their average, can be eliminated using incremental aggregation methods. Such methods maintain an estimate of the full gradient $\nabla f(w)$ whose error has a diminishing variance.

Stochastic average gradient (SAG) is a randomized incremental aggregation method, which uses the search direction as the average of all component gradients evaluated at previous iterates. Specifically, at iteration k, SAG will have stored $\nabla f_i(w^{(d_i^k)})$ for all $i \in \{1,\ldots,m\}$, where d_i^k represents the latest iterate at which ∇f_i was evaluated. An index $j \in \{1,\ldots,m\}$ is then drawn uniformly at random and the search direction is set by

$$\begin{aligned} s^{(k)} &= \frac{1}{m}\left(\nabla f_j(w^{(k)}) - \nabla f_j(w^{(d_j^k)}) + \sum_{i=1}^m \nabla f_i(w^{(d_i^k)})\right) \\ &= \frac{1}{m}\left(\nabla f_j(w^k) + \sum_{i=1, i\neq j}^m \nabla f_i(w^{(d_i^k)})\right). \end{aligned} \tag{6}$$

Although this $s^{(k)}$ is not an unbiased estimator of $\nabla f(w^{(k)})$, SAG enjoys a linear rate of convergence to the true optimizer without any residual error [42]. Specifically, SAG with the constant step-size $\gamma = \frac{1}{16L}$ has the iteration complexity of

$$\mathcal{O}\left((Q+n)\log(1/\varepsilon)\right).$$

Inspired by SAG, several variance reduction methods were proposed including, to name a few, stochastic variance reduced gradient (SVRG) [43], stochastic average gradient with an unbiased estimator (SAGA) [44], and stochastic dual coordinate accent (SDCA) [45]. The SAG method is a randomized variant of the incremental aggregated gradient method (IAG) [46]. The search direction of IAG is identical to that of SAG (6), but the index j of the component function updated at every iteration is chosen cyclically rather than randomly. More precisely, the component functions are processed one-by-one using a deterministic cyclic order on the index set $\{1, 2, \ldots, m\}$, and hence, d_i^k admits the recursion

$$d_i^k = \begin{cases} k & \text{if } i = (k-1 \mod m) + 1, \\ d_i^{k-1} & \text{otherwise.} \end{cases}$$

The natural parallelization strategy for SAG and IAG is the same as for SGD: multiple workers draw independent indices uniformly at random from $\{1,\ldots,m\}$, compute $\nabla f_i(x)$ and return these to a master that modifies the search direction, updates the decision vector, and pushes the updated decision vectors to idle workers. More precisely, the search direction used in the update rule of the asynchronous IAG is given by

$$s^{(k)} = \frac{1}{m}\sum_{i=1}^m f_i(w^{(d_i^k)}).$$

Note that the values

$$\tau_i^k := k - d_i^k,$$

can be viewed as the delay encountered by the gradients of the component functions at k^{th} update.

In [47], it was shown that if $\tau_i^k \leq \tau_{\max}$ for all i and $k \in \mathbb{N}$, then asynchronous IAG with constant step-size

$$\gamma \in \left(0,\ \frac{8\mu}{25L(\tau_{\max}+1)(\mu+L)}\right)$$

requires

$$\mathcal{O}\left((\tau_{\max}+1)^2 Q^2 \log(1/\varepsilon)\right)$$

iterations to achieve an ε-optimal solution. Since the analysis in [47] considers deterministic guarantees on ε-optimality, it is natural that the convergence time bounds are more conservative than those of its stochastic counterparts.

In [41], a lock-free asynchronous version of SAGA, called ASAGA, was proposed. If the maximum delay bound in the ASAGA implementation satisfies $\tau_{\max} < m/10$, then the iteration complexity is given by

$$\mathcal{O}\left(\left((\sqrt{\Delta}\tau_{\max}+1)Q + m\right)\log(1/\varepsilon)\right).$$

Therefore, ASAGA obtains the same iteration complexity as SAG and SAGA when $\tau_{\max}$ satisfies $\tau_{\max} \leq \mathcal{O}(m)$ and

$$\tau_{\max} \leq \mathcal{O}\left(\frac{1}{\sqrt{\Delta}}\max\left\{1, \frac{m}{Q}\right\}\right).$$

This means that in the well-conditioned regime where $m > Q$, a linear speedup is theoretically possible for ASAGA even without sparsity. This is in contrast to some work on asynchronous incremental gradient methods which required sparsity to get a theoretical linear speedup over their sequential counterpart [48].

C. Asynchronous coordinate descent methods

Stochastic gradient methods handle datasets with many samples m by using a search direction that avoids evaluating the gradient of the loss at every sample. Similarly, *coordinate descent* methods address problems with large decision vector dimension d by avoiding to compute updates for every decision variable in every iteration.

Coordinate descent methods, which traditionally cycle through coordinates in a deterministic order, have a long history in optimization (see, e.g., [49]). The research was revitalized by Nesterov's elegant analysis of randomized coordinate descent methods [50]. At each iteration k, these methods draw a random coordinate $j(k)$ from $\{1, \ldots, d\}$ and perform the update

$$[w^{(k+1)}]_{j(k)} = [w^{(k)}]_{j(k)} - \gamma[\nabla f(w^{(k)})]_{j(k)} \tag{7}$$

Similarly to mini-batching in SGD, one is not restricted to picking a single coordinate to update in each iteration, but can sample random subsets (blocks) of coordinates [50].

Synchronous parallel coordinate descent methods have been suggested in, e.g., [51], [52]. In each iteration of these methods, a master node draws n (blocks of) coordinates and distributes the work to evaluate the associated partial gradients on n workers. The master waits for all workers to return before it updates the decision vector and continues with the next iteration. An asynchronous coordinate descent method for shared-memory architectures was proposed and analyzed in [53]. In essence, this method spawns n parallel coordinate descent processes. In each process, a worker thread performs an inconsistent read of $w^{(k)}$ from the shared memory, draws a random coordinate index and performs the update (7) using atomic writes. Linear convergence is proven under the assumption that the maximum overlap $\tau_{\max}$ (defined in the same way as for HOGWILD! above) is small enough. When the coupling between components is weak (in a precise sense defined in [53]), $\tau_{\max}$ can be of order $d^{1/4}$, while the maximal admissible $\tau_{\max}$ shrinks close to one when the coupling is strong. An extension of this asynchronous coordinate scheme to operator mappings is presented in [54].

Note that coordinate descent methods are naturally variance-reduced, since partial derivatives at the optimum are all zero, *i.e.*, $[\nabla f(w^\star)]_j = 0$ for all $j = 1, \ldots, d$. Thus, the value of variance-reduced coordinate descent methods may appear limited. However, the main drawback of many coordinate descent methods is that they cannot handle non-separable regularizers. Variance-reduced coordinate descent methods, on the other hand, allow us to solve optimization problems with arbitrary (not necessarily separable) regularizers [55], [56].

D. Proximal methods for convex and non-convex optimization

For ease of exposition, we have described stochastic gradient methods for smooth and strongly convex losses. However, many of the results extend directly to proximal gradient methods for optimization problems with convex and non-convex loss functions plus a possibly non-smooth regularization term. Specifically, the results in [39], [53] already consider composite optimization problems comprising a smooth finite-sum term and a non-smooth regularizer. An extension of ASAGA to such problems is described and analyzed in [57]. Convergence rate of asynchronous mini-batch algorithms and randomized coordinate descent methods for non-convex optimization are studied in [58]. Extensions of HOGWILD! to non-convex optimization problems are presented in [59] and [60]. In [61], a theoretical upper-bound on the convergence rate of IAG for non-convex composite optimization is derived.

E. Analysis techniques

As mentioned above, the framework of [10] for partially asynchronous algorithms (i.e., those with bounded delay), is not directly applicable to stochastic optimization methods described above. Instead, convergence guarantees have typically been established on a per-algorithm basis, often using complex and laborious induction proofs in which sources of conservatism are hard to isolate. A closer analysis of these proofs reveals that they rely on a few common principles. One such principle is to introduce a well-defined global ordering of events in the system, and model (bounded) asynchrony as (bounded) time-varying delays [10]. As discussed in [41], this ordering may be non-trivial in algorithms such as ASAGA and HOGWILD!. Another principle is to view iterates as perturbed versions of

ideal quantities [48]. A third principle is to reduce expressions for the evolution of the iterate suboptimality, which typically depends on many previous iterates, to standard forms that are well-understood. A number of such sequence results, derived specifically for asynchronous optimization algorithms, are introduced in [62], [63].

To be more concrete, let us consider asynchronous algorithms as gradient iterations with additive gradient errors, i.e.,

$$w^{(k+1)} = w^{(k)} - \gamma\big(\nabla f(w^{(k)}) + e^{(k)}\big).$$

Similar to known lines of convergence proofs for gradient and subgradient methods with errors, it follows directly by expanding the squared norm of the iterate error that

$$\begin{aligned}||w^{(k+1)} - w^\star||^2 = ||w^{(k)} - w^\star||^2 - 2\gamma\langle w^{(k)} - w^\star, \nabla f(w^{(k)})\rangle \\ + \gamma^2||\nabla f(w^{(k)})||^2 + E^{(k)},\end{aligned}$$

where the gradient errors are encapsulated by the last term

$$E^{(k)} = \gamma^2||e^{(k)}||^2 - 2\gamma\langle w^{(k)} - \gamma\nabla f(w^{(k)}) - w^\star, e^{(k)}\rangle.$$

Letting $V^{(k)} = ||w^{(k)} - w^\star||^2$, and using standard strong convexity and smoothness inequalities [17] allow us to derive

$$V^{(k+1)} \leq \left(1 - 2\gamma\frac{\mu L}{L+\mu}\right)V^{(k)} + E^{(k)},$$

for any step-size $\gamma \in \left(0, \frac{2}{L+\mu}\right]$. Note that when $E^{(k)} = 0$, taking $\gamma = \frac{2}{L+\mu}$ leads to

$$V^{(k+1)} \leq \left(\frac{Q-1}{Q+1}\right)^2 V^{(k)},$$

which guarantees linear convergence of the iterates to the optimum. This is the standard analysis of the gradient descent method. For the asynchronous case, the error term $E^{(k)}$ can often be bounded by terms involving only distances of the current and past iterates from $w^\star$, i.e.,

$$V^{(k+1)} \leq \left(1 - 2\gamma\frac{\mu L}{L+\mu}\right)V^{(k)} + \mathcal{H}\big(V^{(k)}, V^{(k-1)}, \ldots, V^{(0)}\big),$$

where the function $\mathcal{H}$ models the history dependence. For example, the analysis of IAG in [47] establishes the bound

$$\mathcal{H}\big(V^{(k)}, V^{(k-1)}, \ldots, V^{(0)}\big) = (6\gamma^2L^2\tau + 9\gamma^4L^4\tau^2)\max_{k-2\tau\leq s\leq k} V^{(s)}.$$

To analyze the effect of the asynchrony on IAG, it is then convenient to use the following sequence result.

Lemma 1 ([62])**.** *Let $\{V^{(k)}\}$ be a nonnegative sequence satisfying*

$$V^{(k+1)} \leq pV^{(k)} + q\max_{k-d\leq s\leq k} V^{(s)}, \quad k \in \mathbb{N},$$

for some positive integer d and non-negative scalars p and q such that $p + q \leq 1$. Then, we have

$$V^{(k)} \leq \rho^k V^{(0)}, \quad k \in \mathbb{N},$$

where $\rho = (p+q)^{\frac{1}{p+q}}$.

Using this result with

$$d = 2\tau, \quad p = 1 - 2\gamma\frac{\mu L}{L+\mu}, \quad q = 6\gamma^2L^2\tau + 9\gamma^4L^4\tau^2,$$

yields that the iterates generated by IAG with constant step-size

$$\gamma \in \left(0, \frac{8\mu}{25L(\tau+1)(\mu+L)}\right)$$

are globally linearly convergent [47].

For stochastic asynchronous algorithms, it is sometimes more natural to interpret the algorithmic effects of asynchrony as perturbing the stochastic iterates with bounded noise [48]. Consider the following iteration

$$w^{(k+1)} = w^{(k)} - \gamma g(w^{(k)} + \eta^{(k)}, \xi^{(k)}),$$

where $\eta^{(k)}$ is a stochastic error term, $\xi^{(k)}$ is a random variable independent of $w^{(k)}$, and g is an unbiased estimator of the true gradient of f at $w^{(k)}$:

$$\mathbb{E}_{\xi^{(k)}}\big[g(w^{(k)}, \xi^{(k)})\big] = \nabla f(w^{(k)}).$$

Let $\widehat{w}^{(k)} = w^{(k)} + \eta^{(k)}$. Then,

$$\begin{aligned}||w^{(k+1)} - w^\star||^2 = ||w^{(k)} - w^\star||^2 + \gamma^2||g(\widehat{w}^{(k)})||^2 \\ - 2\gamma\langle\widehat{w}^{(k)} - w^\star, g(\widehat{w}^{(k)})\rangle \\ + 2\gamma\langle\widehat{w}^{(k)} - w^{(k)}, g(\widehat{w}^{(k)})\rangle.\end{aligned}$$

Assume that $\widehat{w}^{(k)}$ and $\xi^{(k)}$ are independent. Then, taking expectation and using a standard strong convexity bound as well as a squared triangle inequality yields

$$\begin{aligned}V^{(k+1)} \leq \left(1 - \frac{\gamma\mu}{2}\right)V^{(k)} - 2\gamma X^{(k)} + \gamma^2\underbrace{\mathbb{E}[||g(\widehat{w}^{(k)})||^2]}_{R_0^{(k)}} \\ + \gamma\mu\underbrace{\mathbb{E}[||\widehat{w}^{(k)} - w^{(k)}||^2]}_{R_1^{(k)}} \\ + 2\gamma\underbrace{\mathbb{E}[\langle\widehat{w}^{(k)} - w^{(k)}, g(\widehat{w}^{(k)})\rangle]}_{R_2^{(k)}},\end{aligned}$$

where $V^{(k)} = \mathbb{E}[||w^{(k)} - w^\star||^2]$ and $X^{(k)} = \mathbb{E}[f(w^{(k)}) - f^\star]$. Note that $R_0^{(k)}$, $R_1^{(k)}$, and $R_2^{(k)}$ are error terms due to asynchrony: $R_0^{(k)}$ captures the delayed gradient decay with each iteration, $R_1^{(k)}$ represents the mismatch between the true iterate and its noisy (outdated) estimate, and $R_2^{(k)}$ measures the size of the projection of that mismatch on the gradient at each step. To derive the convergence rate, we bound these error terms using past values of $X^{(k)}$, i.e.,

$$\begin{aligned}V^{(k+1)} \leq \left(1 - \frac{\gamma\mu}{2}\right)V^{(k)} - 2\gamma X^{(k)} \\ + \mathcal{H}\big(X^{(k)}, X^{(k-1)}, \ldots, X^{(0)}\big).\end{aligned}$$

For example, HOGWILD! admits the history function

$$\begin{aligned}\mathcal{H}\big(X^{(k)}, X^{(k-1)}, \ldots, X^{(0)}\big) = 4\gamma^2LC_2\sum_{j=k-\tau}^{k-1} X^{(j)} \\ + 4\gamma^2LC_1X^{(k)},\end{aligned}$$

where $C_1 = 1 + \sqrt{\Delta}\tau$ and $C_2 = \sqrt{\Delta} + \gamma\mu C_1$ [41]. The following result is then convenient to apply.

Lemma 2 ([63]). *Assume the non-negative sequences $\{V^{(k)}\}$ and $\{X^{(k)}\}$ satisfy*

$$V^{(k+1)} \leq aV^{(k)} - bX^{(k)} + c\sum_{j=k-\tau}^{k} X^{(j)}, \tag{8}$$

where $a \in (0,1)$, and b and c are nonnegative real numbers. If $a < 1$ and

$$\frac{c}{1-a}\frac{1-a^{\tau+1}}{a^{\tau}} \leq b,$$

then $V^{(k)} \leq a^k V^{(0)}$ for all $k \in \mathbb{N}_0$.

Using this result with $a = 1 - \gamma\mu/2$, $b = 2\gamma$, and $c = 4\gamma^2 LC_2$, immediately yields the convergence rate of HOGWILD! stated in Section IV-A above.

A similar analysis can be made for IAG, HOGWILD!, and many other algorithms, also in the absence of strong convexity. These results require slightly different sequence results. It is also possible to derive convergence rate results for iterations with unbounded delays. In fact, [64] presents the convergence rate results for asynchronous max-norm contractions for both the totally and partially asynchronous models.

There are methods that use an aggregate of iterates instead of the current iterate in their update rules, such as variance reduction methods MISO [65] and FINITO [66], incremental gradient methods [67], and delay-tolerant gradient methods [68]. To be more specific, let $y_i^{(k)}$ be the copy of the decision variable w used in the most recent computation of ∇f_i available at iteration k. The variable $y_i^{(k)}$ is updated as

$$y_i^{(k)} = \begin{cases} w^{(k)} & \text{if } i = i(k), \\ y_i^{(k-1)} & \text{otherwise,} \end{cases}$$

where $i(k)$ is the index of the component function chosen uniformly at random at step k. Then, the update of MISO and FINITO is given by

$$w^{(k+1)} = \frac{1}{m}\sum_{i=1}^{m} y_i^{(k)} - \gamma\sum_{i=1}^{m}\nabla f_i(y_i^{(k)}).$$

Since the update rule of these algorithms cannot be written as

$$w^{(k+1)} = w^{(k)} - \gamma s^{(k)},$$

their convergence does not follow directly from the arguments above. Nevertheless, we note that the proof in [68, Equation 8] hinges on the same argument as Lemma 1.

V. DECENTRALIZED ALGORITHMS

Decentralized algorithms (also known as "consensus," "gossip," or "multi-agent" algorithms) are an alternative to parameter-server algorithms in the distributed memory setting. As the name suggests, in decentralized methods, there is no authoritative copy of the parameters; rather, each worker maintains and updates a local working copy of the optimization variables. In contrast to centralized methods, decentralized methods do not have a single bottleneck or point-of-failure, and thus may potentially scale to larger problems.

Consider a system with n workers, and let $w_i^{(k)}$ denote the copy at worker i after k iterations. A simple synchronous decentralized method starts with all nodes at the same initial point $w_i^{(0)} = w^{(0)}$ and repeats updates of the form

$$w_i^{(k+1)} = \sum_{j=1}^{n} P_{i,j}^{(k)} w_j^{(k)} - \gamma_i^{(k)} s_i^{(k)}, \tag{9}$$

where $P_{i,j}^{(k)}$ is a scalar between 0 and 1 quantifying the influence of worker j on worker i, and $s_i^{(k)}$ is the search direction computed at worker i.

If $P_{i,j}^{(k)} = 1/n$ for all $i, j \in [n]$ and $k \geq 0$, then the variables at every worker are identical (i.e., exactly equal to their average) after every update. Furthermore, if the directions $s_j^{(k)}$ are independent stochastic gradients computed using b gradient samples, then the method is equivalent to SGD with mini-batch size nb. Implementing this update with $P_{i,j}^{(k)} = 1/n$ requires coordination among all workers. This can be accomplished in a communication-efficient way using the ALLREDUCE primitive mentioned in Section III; we refer to such a method as ALLREDUCE SGD (AR-SGD) [69], [70]. In a system where $P_{i,j}^{(k)} = 1/n$ for all $i, j \in [n]$ and $k \geq 0$, each node could viewed as being a worker *and* a central authority for the purpose of contrasting with the methods discussed in Section IV.

In general, the updates at each worker need not depend on the values from all other workers. For more general values of $P_{i,j}^{(k)}$, worker i only needs to receive messages from worker j if $P_{i,j}^{(k)} > 0$. When there are n workers, the entries $P_{i,j}^{(k)}$ can be collectively viewed as an $n \times n$ matrix $P^{(k)}$. Equivalently, one can form a communication graph with one vertex for each worker and with an edge set $\mathcal{E}^{(k)}$ containing a (directed) edge from j to i if $P_{i,j}^{(k)} \neq 0$.

A natural way to generalize the notion of averaging while reducing the communication overhead is to make use of matrices $P^{(k)}$ (equivalently, communication graphs) that are sparse, and that correspond to a diffusion or random walk. In an asynchronous implementation, it is possible that messages may be delayed, and so it is not practical to assume that $P^{(k)}$ is symmetric and doubly-stochastic,[2] since this would impose that if i receives a message from j then j also receives one from i to perform an update. Such a method is referred to as *push-pull* since it requires two-way exchange of information. Doubly-stochastic methods guarantee that workers converge to the network-wide average at a geometric rate, but they are inherently synchronous.

Methods using row-stochastic $P^{(k)}$ are referred to as *pull-based* methods; they only involve a one-way exchange of information, and the weights $P_{i,j}^{(k)}$ are determined and applied by the receiver i. Row-stochastic methods guarantee that the vectors $w_i^{(k)}$ at each worker converge to a consensus at a geometric rate. However, the consensus values are not necessarily an unbiased estimate of the network-wide average;

[2] A matrix P is row-stochastic (respectively, column-stochastic) if each row (respectively, column) of P sums to 1, and all entries are non-negative. P is doubly-stochastic if it is both row- and column-stochastic.

this bias in the consensus values can prevent the iterates in (9) from converging to a minimizer of (2).

Methods using column-stochastic $P^{(k)}$ are referred to as *push-based* methods; they only involve a one-way exchange of information, and the weights $P_{i,j}^{(k)}$ are determined and applied by the sender j. Column-stochastic methods guarantee that the vectors $w_i^{(k)}$ at each worker converge at a geometric rate, but not necessarily to a consensus. Rather, the limit value depends on the message-passing topology (through the stationary distribution, if $P^{(k)} = P$ is constant over time and seen as the transition matrix of a Markov chain). However, unlike row-stochastic methods, this discrepancy is easy to correct in column-stochastic methods.

The PUSH-SUM algorithm [71] (also called *ratio consensus*) is a column-stochastic method in which each worker tracks one additional parameter, referred to as the push-sum weight, that can be used to compensate for discrepancies due to the message-passing topology. The push-sum weight, which we denote by $\phi_i^{(k)}$, is initialized to 1 at every worker. Whenever a worker communicates its parameters, it also communicates the push-sum weight. Any imbalance built up in the parameters also appears in the push-sum weight. Therefore, by rescaling the parameters by the push-sum weight, workers running the PUSH-SUM algorithm are guaranteed to converge to a consensus on the network-wide average at a geometric rate. Decentralized optimization methods built on PUSH-SUM have the form,

$$\begin{aligned} w_i^{(k+1)} &= \sum_{j=1}^{n} P_{i,j}^{(k)} w_j^{(k)} - \gamma_i^{(k)} s_i^{(k)}, \\ \phi_i^{(k+1)} &= \sum_{j=1}^{n} P_{i,j}^{(k)} \phi_j^{(k)}, \\ z_i^{(k+1)} &= \frac{w_i^{(k+1)}}{\phi_i^{(k+1)}}. \end{aligned} \tag{10}$$

The (synchronous) Stochastic Gradient-Push (SGP) algorithm [72] is an analog of stochastic gradient descent for decentralized optimization. Specifically, SGP uses updates (10) where the search direction $s_j^{(k)}$ is a stochastic mini-batch gradient evaluated by worker j at the rescaled point $z_j^{(k)}$ on a subset of the data $I_j^{(k)}$,

$$s_j^{(k)} = \sum_{m \in I_j^{(k)}} \nabla \ell \left(p(x_m; z_j^{(k)}),\ y_m \right).$$

When the functions f_i are strongly convex, each worker j running SGP with a diminishing step-size is guaranteed to converge to a minimizer of f [72]:

$$f(\hat{z}_j^{(K)}) - f(w^\star) \leq \mathcal{O}\left(\frac{\log K}{K} \right),$$

where $\hat{z}_j^{(K)}$ is a weighted average of the sequence $\{z_j^{(k)}\}_{k=0}^{K}$ produced at worker j.

The SGP algorithm is synchronous since each worker i blocks to send and receive messages from other workers j for which $P_{i,j}^{(k)} > 0$ before proceeding to the next iteration. Since *pull-based* methods and *push-based* methods only involve a one-way exchange of information, they are readily amenable to asynchronous implementations. In the next section we will describe some specific asynchronous decentralized optimization methods along with their known convergence guarantees.

A. Asynchronous Decentralized Methods

The Overlap Stochastic Gradient-Push (OSGP) algorithm [73] builds on SGP by allowing for message delays. OSGP uses the same search direction as SGP, but reduces the communication and synchronization overhead by overlapping communication of parameters between workers with multiple stochastic gradient updates. Let $\tau_{i,j}^{(k)}$ denote the delay experienced by a message sent from worker j and received by worker i at iteration k (i.e., the message was transmitted at time $k - \tau_{i,j}^{(k)}$). By convention, we take $\tau_{i,i}^{(k)} = 0$ for all k and $i \in [n]$. Let $\mathcal{M}_i^{(k)}$ denote the set such that $\tau_{i,j}^{(k)} \in \mathcal{M}^{(k)}$ implies that worker i received a message from j at iteration k with delay $\tau_{i,j}^{(k)}$. The updates in OSGP can be written in terms of these delayed indices as

$$\begin{aligned} w_i^{(k+1)} &= \sum_{\tau_{i,j}^{(k)} \in \mathcal{M}_i^{(k)}} P_{i,j}^{(k-\tau_{i,j}^{(k)})} w_j^{(k-\tau_{i,j}^{(k)})} - \gamma_i^{(k)} s_i^{(k)}, \\ \phi_i^{(k+1)} &= \sum_{\tau_{i,j}^{(k)} \in \mathcal{M}_i^{(k)}} P_{i,j}^{(k-\tau_{i,j}^{(k)})} \phi_j^{(k-\tau_{i,j}^{(k)})}. \end{aligned} \tag{11}$$

The rescaling update for $z_i^{(k)}$ is identical to the one in SGP. Note that, although OSGP handles message delays, it does not deal with computation delays (i.e., heterogeneous update rates amongst workers). Specifically, each OSGP worker i must perform the updates in (11) at every iteration.

The Asynchronous Gradient-Push algorithm (AGP) proposed in [74] and analyzed in [75] is an analog of gradient descent for asynchronous decentralized optimization, and deals with both message and computation delays. This algorithm is similar to SGP, but removes all synchronization points. Let $\delta_i^{(k)} \in \{0, 1\}$ denote a binary indicator that is equal to 1 if worker i completes an update at iteration k, and is equal to 0 otherwise. The global iteration counter k (used only to describe the algorithm) increments whenever any worker (or subset of workers) completes a gradient-based update. If worker i completes an update at iteration k (i.e., $\delta_i^{(k)} = 1$), then the AGP update is identical to that in (11). If worker i does *not* complete an update at iteration k (i.e., $\delta_i^{(k)} = 0$), then its iterates remain unchanged

$$w_i^{(k+1)} = w_i^{(k)}, \quad \phi_i^{(k+1)} = \phi_i^{(k)}, \quad z_i^{(k+1)} = z_i^{(k)}. \tag{12}$$

In contrast to OSGP, note that the AGP workers do not necessarily update their parameters at every iteration.

Suppose the message delays and the time between an AGP worker's successive updates are bounded. When the functions f_i are strongly convex and L-smooth, workers running AGP up to a global iteration K minimize a re-weighted version of (2), defined as [75]

$$\underset{w \in \mathbb{R}^d}{\text{minimize}} \quad \sum_{i=1}^{n} \overline{p}_i^{(K)} f_i(w), \tag{13}$$

where the re-weighting values $\overline{p}_i^{(K)} > 0$ are given by

$$p_i^{(K)} := \sum_{k=0}^{K-1} \gamma_i^{(k)} \delta_i^{(k)}, \quad \text{and} \quad \overline{p}_i^{(K)} := \frac{p_i^{(K)}}{\sum_{i=1}^n p_i^{(K)}}. \tag{14}$$

In particular, letting $w_K^\star$ denote the minimizer of (13), it can be shown that [75, Theorems 4 & 5]

$$\frac{1}{K}\sum_{k=0}^{K} \left\| \frac{1}{n}\sum_{i=1}^n z_i^{(k)} - w_K^\star \right\|^2 \leq \mathcal{O}\left(\frac{1}{\sqrt{K}}\right).$$

If all workers use the same constant step-size and perform a similar number of gradient-based updates by the end of training, then $\overline{p}_i^{(K)} \approx 1/n$, and the workers converge to the unbiased minimizer of the objective in (2). On the other hand, workers that perform more updates than their peers bias the solution towards their local objective.

This convergence theory also suggests an approach for correcting the bias: slower workers can use larger step-sizes in order to compensate for their slower update rates. To do this workers need an idea of how many updates they have performed locally relative to the total number of updates performed by all workers. This can also be estimated in a decentralized way by communicating one additional scalar variable, and when the functions f_i are convex and smooth, it can be shown that [76]

$$\max_{k \leq K} f(z_j^{(k)}) - f_i(w^\star) \leq \mathcal{O}\left(\frac{\log K}{\sqrt{K}}\right).$$

In [77] a similar method is analyzed in the context of stochastic gradients.

Further improvements are obtained in [78] and [79] by incorporating robust PUSH-SUM, which tolerates dropped messages by using additional memory at each worker, and by incorporating gradient-tracking schemes, which lead to faster iteration-wise convergence but also involve twice the communication overhead per iteration, and hence may not be practical for machine learning problems with high-dimensional models.

We note that decentralized asynchronous methods have also been proposed based on applying coordinate descent methods to a dual formulation [80], [81]. However, while these methods allow for randomized update order, they are not asynchronous in the sense considered in this paper, of allowing for communication and computation delays.

In the next subsection, we will describe general proof techniques for analyzing asynchronous decentralized optimization algorithms under bounded message *and* computation delays. Following that discussion, we will summarize empirical assessments of these methods in the literature, and conclude by describing practical challenges and open problems in this budding research area.

B. Analysis

Similar to centralized methods, analysis techniques for decentralized methods also exploit the idea of using a well-defined order of events in the system. However, decentralized methods require fundamentally different analysis techniques than centralized methods. To simplify the discussion, suppose

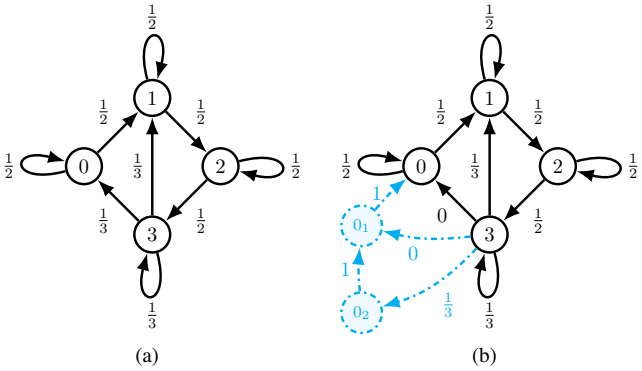


Figure 3. (a) Example of a delay-free 4-worker network. Edges are labeled with column-stochastic weights $P_{i,j}$. (b) The same network augmented with virtual nodes and edges (dashed blue lines). For readability, this figure only shows the virtual workers/edges used to model delays for messages transmitted to worker 0; in the analysis, virtual nodes and edges are added for every worker. This particular example illustrates a message from 3 to 0 with a delay of $\tau_{\max} = 2$.

that the averaging weights $P_{i,j}^{(k)}$ are static (i.e., workers always choose the same averaging weights to communicate with their neighbours). Equations such as (9) and (10) describe synchronous decentralized optimization algorithms from an individual worker's perspective. However, it is typically easier to study these methods from a global perspective by collectively viewing the entries $P_{i,j}$ as an $n \times n$ matrix P and viewing the variables $w_i^{(k)}$ and $s_i^{(k)}$ as the rows of $n \times d$ matrices $W^{(k)}$ and $S^{(k)}$ respectively. Then equation (9) can be re-written in matrix-vector form as

$$W^{(k+1)} = PW^{(k)} - \Gamma^{(k)} S^{(k)}, \tag{15}$$

where $\Gamma^{(k)}$ is an $n \times n$ diagonal matrix with the step-size $\gamma_i^{(k)}$ on the i^{th} diagonal. Similarly, (10) can be re-written as

$$\begin{aligned} W^{(k+1)} &= PW^{(k)} - \Gamma^{(k)} S^{(k)} \\ \phi^{(k+1)} &= P\phi^{(k)} \\ Z^{(k+1)} &= \text{diag}(\phi^{(k+1)})^{-1} W^{(k+1)}, \end{aligned} \tag{16}$$

where $\phi^{(k+1)}$ is an $n \times 1$ vector containing the push-sum weights, and $\text{diag}(\phi^{(k+1)})$ is a diagonal matrix with the push-sum weight $\phi_i^{(k+1)}$ on the i^{th} diagonal.

Broadly, there are three main steps involved in proving convergence of workers' parameters under the assumption of bounded message and computation delays: (i) mathematically modelling delays, (ii) proving convergence of the optimization iterates to a consensus sequence under the delay model, (iii) proving convergence of the consensus sequence to a minimizer.

(i) Modelling delays: Recall that one can form a communication graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with one vertex for each worker and with an edge set $\mathcal{E}$ containing a (directed) edge from j to i if $P_{i,j} \neq 0$. In order to model message delays in analysis, the reference graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is augmented with virtual nodes and edges that store information that has been transmitted but not yet received. Because the message delays are bounded, the number of virtual nodes and edges needed is finite. Figure 3 shows an example of such a graph augmentation for the delays along one edge. Note that the message delays under this bounded

delay model can still vary across edges, and can vary from one iteration to the next. Graph augmentation techniques have also been used to study distributed averaging and agreement algorithms with communication delays [82]–[85]; additional work is needed to properly account for computation delays in asynchronous distributed optimization. In short, one can model asynchronous delay-prone message passing over the graph $\mathcal{G}(\mathcal{V},\mathcal{E})$ as synchronous *time-varying* message passing over the time-varying augmented graph $\widetilde{\mathcal{G}}(\widetilde{\mathcal{V}},\widetilde{\mathcal{E}}^{(k)})$. One can also incorporate heterogeneous update rates into the analysis by multiplying the diagonal step-size matrix $\Gamma^{(k)}$ in equations (15) and (16) with a diagonal binary indicator matrix $\Delta^{(k)}$, with i^{th} diagonal $\delta_i^{(k)}$ indicating whether worker i completed an update at global iteration k. We emphasize that graph augmentation techniques are only used for the purpose of analysis, to model decentralized systems with delays.

The augmented version of equation (15) is

$$\widetilde{W}^{(k+1)} = \widetilde{P}^{(k)}\widetilde{W}^{(k)} - \widetilde{\Delta}^{(k)}\widetilde{\Gamma}^{(k)}\widetilde{S}^{(k)}, \tag{17}$$

where the augmented matrix $\widetilde{W}^{(k)}$ has dimensions $(\tau_{\max}+1)n \times d$; i.e., one row containing the parameters at each worker and each virtual node at iteration k. We emphasize that this modeling is only used for analysis and is not needed to implement asynchronous decentralized methods. The rows of the $(\tau_{\max}+1)n \times d$ matrix $\widetilde{S}^{(k)}$ that correspond to virtual nodes are always equal to 0. Equation (16) can be described similarly with respect to this enlarged state-space.

Note that the averaging matrix $\widetilde{P}^{(k)}$ in (17) is time-varying for two reasons: first, only those workers that are active at iteration k perform an update, and second, message delays can vary across iterations. Overall, this model reduces the time-varying and delay-prone dynamics of a decentralized asynchronous algorithm to the evolution of an augmented synchronous system.

(ii) Convergence to a consensus sequence: Note that (17) can be viewed as the evolution of a perturbed Markov chain with state-transition matrix $\widetilde{P}^{(k)}$ and perturbations $\widetilde{\Delta}^{(k)}\widetilde{\Gamma}^{(k)}\widetilde{S}^{(k)}$. Using standard tools from the Markov chain literature [86], [87], one can characterize the convergence rate of the iterates $w_i^{(k)}$ to a consensus sequence $\overline{w}^{(k)}$ using the joint spectral properties of the matrices $\widetilde{P}^{(k)}$. The resulting bounds are often of the form

$$\left\|w_i^{(k)} - \overline{w}^{(k)}\right\| \leq C\rho^k \left\|w_i^0\right\| + C\sum_{\ell=0}^{k} \rho_i^k \left\|\delta_i^{(k-\ell)}\gamma_i^{(k-\ell)}s_i^{(k-\ell)}\right\|, \tag{18}$$

for all $i \in [n]$, where $\rho \in (0,1)$ and $C \in (0,\infty)$ are constants that depend on the delays and graph connectivity, and $\{\overline{w}^{(k)}\}$ is the consensus sequence. It follows from (18) that if the perturbations $(\delta_i^{(k)}\gamma_i^{(k)}s_i^{(k)})$ tend to 0, then all workers converge to the consensus sequence $\overline{w}^{(k)}$, even in the presence of arbitrary, uniformly bounded message and computation delays. There are several different approaches in the literature for bounding ρ; see [75], [76], [78] for details. When the weight matrices $P^{(k)}$ are column-stochastic or doubly-stochastic, the consensus sequence $\overline{w}^{(k)}$ is typically defined as the network-wide average of the parameters at iteration k (i.e., $1/n \sum_{i=1}^n w_i^{(k)}$).

To prove that the iterates $w_i^{(k)}$ converge to the consensus sequence using (18), one must show that the perturbations (gradient-based updates) $(\delta_i^{(k)}\gamma_i^{(k)}s_i^{(k)})$ tend to 0. When using a diminishing step-size (i.e., $\gamma_i^{(k)} \to 0$), this is a trivial result (as long as the search directions remain bounded). When using a constant step-size, the conditions for consensus, namely $(\delta_i^{(k)}\gamma_i^{(k)}s_i^{(k)})$ converging to 0 for all $i \in [n]$, and the conditions for optimality, namely $\overline{w}^{(k)}$ converging to a minimizer, are often tightly interdependent. Gradient-tracking methods using a constant step-size, such as ASY-SONATA, have this interdependence, and typically show consensus and optimality simultaneously using (18) and the *small-gain theorem* [88]. In brief, the *small-gain-theorem* says that if, for all positive integers K, there exists $\lambda \in (0,1)$, finite constants $C_1, C_2 \geq 0$, and gains $G_1, G_2 \geq 0$ with $G_1G_2 < 1$, such that

$$\sup_{k\leq K} \frac{\left\|s_i^{(k)}\right\|}{\lambda^k} \leq \sup_{k\leq K} G_1 \frac{\left\|w_i^{(k)} - \overline{w}^{(k)}\right\|}{\lambda^k} + C_1,$$

and

$$\sup_{k\leq K} \frac{\left\|w_i^{(k)} - \overline{w}^{(k)}\right\|}{\lambda^k} \leq \sup_{k\leq K} G_2 \frac{\left\|s_i^{(k)}\right\|}{\lambda^k} + C_2,$$

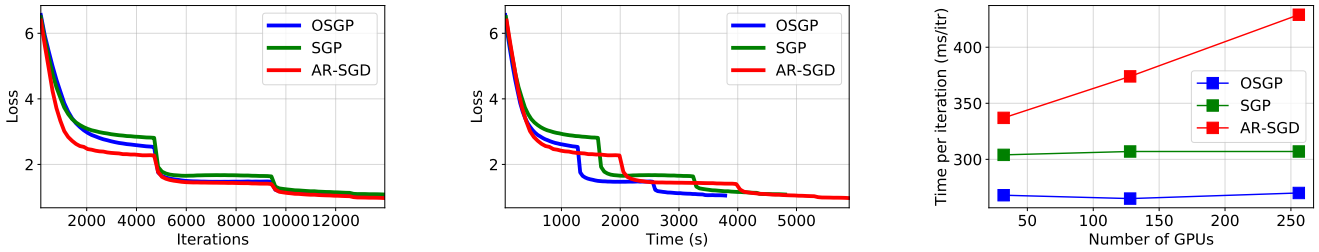
then both $\left\|w_i^{(k)} - \overline{w}^{(k)}\right\|$ and $\left\|s_i^{(k)}\right\|$ converge to 0 at a linear rate characterized by the sequence $\{\lambda^k\}$. It is relatively straightforward to generalize the small-gain-theorem to characterize other convergence rates as well; e.g., proving sublinear convergence by replacing the sequence $\{\lambda^k\}$ in the denominators with a sublinearly convergent sequence $\{r_k\}$.

(iii) Convergence of consensus sequence to a minimizer: We will now describe a general approach for proving convergence of the consensus sequence in a way that provides some intuition into the convergence behaviour of asynchronous decentralized optimization methods.

Due to the presence of the binary indicator $\delta_i^{(k)}$ in the gradient-based updates in (17), one cannot guarantee a contraction with respect to the global objective at sufficiently large iterations k. Intuitively, some workers may take gradient steps that move the parameters away from the global minimizer. The key observation is that, while each iteration may not produce a descent direction, the sum of the gradient-based updates $\sum_k \sum_{i=1}^n \delta_i^{(k)}\gamma_i^{(k)}s_i^{(k)}$ over sufficiently many consecutive iterations may point in a descent direction when the computation delays are bounded. For example, for AGP it can be shown that this cumulative gradient vector points in a descent direction with respect to the re-weighted minimizer defined in (13); see [75, Lemmas 2 and 3].

Typically, after obtaining a contraction result over a finite-time horizon, standard tools from the optimization literature can be applied to obtain convergence of the consensus sequence.

Validity of the bounded delay assumption. All analysis techniques presented in this article assume arbitrary, uniformly bounded (but possibly time-varying) message and computation delays. In practice, we can control the upper bound on the delays by using tools described in Section III. If a worker has not received a message from its neighbours in over τ iterations, it blocks and waits to receive a message.



(a) Training loss vs. iterations, using 32 workers (256 GPUs and 1280 CPUs)

(b) Training loss vs. time, using 32 workers (256 GPUs and 1280 CPUs)

(c) Time-per-iteration (ms/itr) vs. number of workers

Figure 4. Training a ResNet-50 convolutional neural network on the ImageNet classification task over a network of servers. Each worker is an entire server consisting of 40 CPUs and 8 GPUs. Workers are interconnected by a 10 Gbps ethernet network. (a)/(b) The *average* training loss versus the number of iterations/wall-clock time when training with 32 workers. Shaded around each line is the max. and min. error achieved by any given worker in that iteration. (c) Observing how each worker's time per iteration scales as we increase the number of workers (proportional to the number of GPUs).

C. Non-convexity

While the discussion in this section has largely focused on convex objectives, much of the analysis techniques can be extended to non-convex objectives with minor alterations. Specifically, steps (i) and (ii) in the analysis (modelling delays and proving convergence of the optimization iterates to a consensus sequence), remain unchanged. Step (iii), proving convergence of the consensus sequence to a minimizer, is the only part that requires adjustment to account for non-convex objectives. In step (iii), one must obtain the contraction result over a finite-time horizon without making use of the (sub)gradient inequality; instead, it is common to make use of a Taylor-series expansion to express the relationship between the change in the objective error after taking an optimization step, and the expected descent provided by the stochastic search-direction over a finite-time horizon.

As an example, the analysis of the OSGP method for smooth non-convex objectives with stochastic gradients in [73] uses this general proof sketch. Suppose the message delays are uniformly bounded (i.e., there exists a $\tau_{\max} > 0$ such that $\tau_{i,j}^{(k)} \leq \tau_{\max}$ for all k and $i, j \in [n]$). If OSGP is run for K iterations and all workers use a constant step-size $\gamma := \sqrt{n/K}$, then each worker is guaranteed to converge to a stationary point of (2) when the objectives f_i are non-convex and L-smooth [73]

$$\frac{1}{n}\frac{1}{K}\sum_{k=0}^{K}\sum_{i=1}^{n}\left\|\nabla f_i(z_i^{(k)})\right\|^2 \leq \mathcal{O}\left(\frac{1}{\sqrt{nK}}\right). \qquad (19)$$

Remarkably, OSGP converges to a stationary point of smooth non-convex functions with the same order-wise iteration complexity as centralized SGD. Since the analysis for OSGP handles general (strongly-connected) digraphs, and arbitrary (but bounded) time-varying message delays, equation (19) also provides a bound on the convergence rate for SGP (synchronous delay-free setting), and ALLREDUCE SGD (synchronous delay-free all-to-all setting).

D. Example: Training a deep neural network

Next we provide an illustrative example of how some of the algorithms mentioned in Section V-A can be used to speed up training of a deep convolutional neural network model on an image classification task. Each worker is a server consisting of 40 CPU cores and 8 GPUs. The servers are interconnected via a 10 Gbps Ethernet network. All methods are implemented using PyTorch [89] version 0.4.1 using the `tcp` distributed backend. We train a ResNet-50 [90] model containing roughly 25 million optimizable parameters to classify images in the ImageNet dataset [91], made up of over 1 million images and 1000 different image classes.

We compare OSGP, SGP, and ALLREDUCE SGD. Recall that ALLREDUCE SGD is a synchronous method that exactly synchronizes all workers after every update, mathematically equivalent to taking $P_{i,j}^{(k)} = 1/n$ for all $i, j \in [n]$. SGP is a synchronous decentralized method, and OSGP is an asynchronous decentralized method whose analysis allows for delayed messages. Both OSGP and SGP use a time-varying communication graph sequence $P_{i,j}^{(k)}$ with 1 out-neighbour per node; i.e., after each update, a worker transmits a message to just one other worker. See [73] for additional details about the experimental setup.

Figure 4 (a) shows the loss, $f(w)$, as a function the number of iterations when training with 32 workers (256 GPUs and 1280 CPU cores). Figure 4 (b) shows the same loss as a function of the wall-clock time. Although OSGP and SGP converge at slower iteration-wise rates than ALLREDUCE SGD, Figure 4 (b) illustrates that OSGP has significantly reduced training time by mitigating the synchronization and communication overhead. Figure 4 (c) shows the time per iteration as a function of the number of workers in the system, for $n = 4$, 16, and 32 (i.e., 32, 128, and 256 GPUs). The time-per-iteration for both OSGP and SGP remains relatively constant since the communication overhead is always fixed. The time-per-iteration of ALLREDUCE SGD increases since the synchronization and communication costs increase with the number of workers. In short, the asynchronous decentralized method OSGP optimizes $f(w)$ faster than the synchronous methods (in terms of wall-clock time), and exhibits better scaling.

VI. CONCLUSIONS AND DIRECTIONS

While asynchronous decentralized optimization algorithms have shown promising results on large-scale machine learning benchmarks [73], [92]–[94], there is still much work to be done

in understanding the convergence properties of these methods. Section V-A outlines some of the known convergence results, but the constants in these rates are typically large and do not accurately reflect the scaling behaviour of the algorithms. For instance, it is not clear how well these constants reflect the true dependency of the convergence rates on the number of workers, the communication graph, or the delays due to asynchrony. Some recent works such as [95], [96] try to provide a closer look at the constants in the convergence rates of synchronous decentralized optimization algorithms.

Another challenge involves incorporating non-linear gradient-based updates. Optimization methods using non-linear momentum-based updates are commonly used in deep learning [97], [98]. If the gradient-based update is non-linear, then it may not be possible to guarantee that the consensus sequence converges to a minimizer of (2). This issue applies to both synchronous and asynchronous decentralized optimization algorithms. In practice, one typically observes drastic degradation in performance, relative to centralized methods, when naïvely applying decentralized optimization algorithms with non-linear gradient-based updates to large-scale deep learning tasks. One recent work [94] tackles this issue in the synchronous case by periodically incorporating global synchronization between agents. Incorporating non-linear gradient-based updates into both synchronous and asynchronous multi-agent optimization algorithms is still an open problem.

Our discussion focused on analysis under a partially asynchronous delay model, where information delays are only assumed to be bounded, and thus did not cover other work which assumes that delays follow more specific probabilistic models [99], [100]. Such assumptions may lead to tighter bounds when they accurately reflect the salient properties of the underlying system, but verifying the validity of such models is more challenging in practice. In contrast, bounded information delays can be enforced algorithmically, albeit potentially at the cost of some idling.

Lastly, although the inconsistent read perspective is a convenient abstraction for analysis of parallel optimization algorithms in shared memory, it is not an accurate description of the behavior of current multi-core systems with non-uniform memory access. In these systems, frequent concurrent operations on the same elements in shared memory create contention and reduce the efficiency of cache hierarchies. Instead, emerging high-performance algorithms for multi-processors, such as [24], [101] bear striking resemblance with the decentralized methods described in this paper: cores operate on local (inconsistent) copies of the decision vector and coordinate to guarantee global convergence. We believe that there is a significant scope for designing new algorithms tailored to the specifics of NUMA architetures instead of adapting algorithms designed with simpler hardware abstractions.

REFERENCES

[1] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multi-task learners," Feb. 2019, Open AI tech. report.

[2] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized BERT pretraining approach," Jul. 2019, arXiv: 1907.11692.

[3] D. Mahajan, R. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. van der Maaten, "Exploring the limits of weakly supervised pretraining," in *European Conference on Computer Vision (ECCV)*. Springer, 2018, pp. 185–201.

[4] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and its Applications*, vol. 2, no. 2, pp. 199–222, Apr. 1969.

[5] J. L. Rosenfeld, "A case study on programming for parallel processors."

[6] G. M. Baudet, "Asynchronous iterative methods for multiprocessors," *Journal of the ACM*, vol. 25, no. 2, pp. 226–244, Apr. 1978.

[7] D. P. Bertsekas and D. E. Baz, "Distributed asynchronous relaxation methods for convex network flow problems," *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 74–85, Jan. 1987.

[8] D. P. Bertsekas, "Distributed asynchronous computation of fixed points," *Mathematical Programming*, vol. 27, no. 1, pp. 107–120, Sep. 1983.

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

[10] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation*. Prentice Hall, 1989, vol. 23.

[11] L. Bottou and O. Bousquet, "The tradeoffs of large scale learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2007, pp. 161–168.

[12] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *SIAM Review*, vol. 60, no. 2, pp. 223–311, Jan. 2018.

[13] D. Kovalev, K. Mishchenko, and P. Richtárik, "Stochastic Newton and cubic Newton methods with simple local linear-quadratic rates," *arXiv preprint arXiv:1912.01597*, 2019.

[14] S. Soori, K. Mishchenko, A. Mokhtari, M. M. Dehnavi, and M. Gurbuzbalaban, "DAve-QN: A distributed averaged quasi-Newton method with local superlinear convergence rate," in *International Conference on Artificial Intelligence and Statistics*, 2020, pp. 1965–1976.

[15] P. Ramanan, M. Yildirim, E. Chow, and N. Gebraeel, "An asynchronous, decentralized solution framework for the large scale unit commitment problem," *Transactions on Power Systems*, vol. 34, no. 5, pp. 3677–3686, Sep. 2019.

[16] G. Scutari, D. P. Palomar, and S. Barbarossa, "Asynchronous iterative water-filling for gaussian frequency-selective interference channels," *Transactions on Information Theory*, vol. 54, no. 7, pp. 2868–2878, Jul. 2008.

[17] Y. Nesterov, *Introductory Lectures on Convex Optimization*. Springer, 2004.

[18] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[19] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning*. Cambridge University Press, 2014.

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, Nov. 2016. [Online]. Available: https://www.deeplearningbook.org

[21] H. Robbins and S. Monro, "A stochastic approximation method," *Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951.

[22] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao, "Optimal distributed online prediction using mini-batches," *Journal of Machine Learning Research*, vol. 13, pp. 165–202, Jan. 2012.

[23] S. Chen, J. Fang, D. Chen, C. Xu, and Z. Wang, "Adaptive optimization of sparse matrix-vector multiplication on emerging many-core architectures," in *International Conference on High Performance Computing and Communications (HPCC)*. IEEE, Jun. 2018.

[24] N. Ioannou, C. Mendler-Dünner, and T. Parnell, "SySCD: A system-aware parallel coordinate descent algorithm," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019, pp. 590–600.

[25] U. Thakker, G. Dasika, J. Beu, and M. Mattina, "Measuring scheduling efficiency of RNNs for NLP applications," Mar. 2019, international Workshop on Performance Analysis of Machine Learning Systems (Fastpath).

[26] G. E. Blleloch and Y. Gu, "Improved parallel cache-oblivious algorithms for dynamic programming and linear algebra," Aug. 2019, arXiv: 1809.09330.

[27] O. Kislal, M. T. Kandemir, and J. Kotra, "Cache-aware approximate computing for decision tree learning," in *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2016.

[28] E. Wszola, C. Mendler-Dünner, M. Jaggi, and M. Püschel, "On linear learning with manycore processors," in *International Conference on High Performance Computing (HiPC)*. IEEE, Dec. 2019, to appear.

[29] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large

scale distributed deep networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2012, pp. 1223–1231.

[30] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. Andersen, and A. J. Smola, "Parameter Server for distributed machine learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, Dec. 2013, big Learning Workshop.

[31] R. Rabenseifner, "Optimization of collective reduction operations," in *International Conference on Computational Science*, vol. LNCS 3036. Springer, 2004, pp. 1–9.

[32] D. Needell, R. Ward, and N. Srebro, "Stochastic gradient descent, weighted sampling, and the randomized kaczmarz algorithm," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2014, pp. 1017–1025.

[33] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM Journal on Imaging Sciences*, vol. 2, no. 1, pp. 183–202, 2009.

[34] M. Schmidt, R. Babanezhad, M. O. Ahmed, A. Defazio, A. Clifton, and A. Sarkar, "Non-uniform stochastic average gradient method for training conditional random fields," in *Artificial Intelligence and Statistics (AISTATS)*, 2015, pp. 819–828.

[35] F. Hanzely, P. Richtárik, and L. Xiao, "Accelerated Bregman proximal gradient methods for relatively smooth convex optimization," Apr. 2020, arXiv: 1808.03045.

[36] Y. Nesterov, "Gradient methods for minimizing composite functions," *Mathematical Programming*, vol. 140, no. 1, pp. 125–161, 2013.

[37] M. Zinkevich, J. Langford, and A. J. Smola, "Slow learners are fast," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2009, pp. 2331–2339.

[38] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2011, pp. 873–881.

[39] H. R. Feyzmahdavian, A. Aytekin, and M. Johansson, "An asynchronous mini-batch algorithm for regularized stochastic optimization," *Transactions on Automatic Control*, vol. 61, no. 12, pp. 3740–3754, Dec. 2016.

[40] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2011, pp. 693–701.

[41] R. Leblond, F. Pedregosa, and S. Lacoste-Julien, "ASAGA: Asynchronous Parallel SAGA," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, vol. 54. PMLR, Apr. 2017, pp. 46–54.

[42] N. Le 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 (NeurIPS)*, 2012, pp. 2663–2671.

[43] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2013, pp. 315–323.

[44] A. Defazio, F. Bach, and S. Lacoste-Julien, "SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2014, pp. 1646–1654.

[45] S. Shalev-Shwartz and T. Zhang, "Accelerated mini-batch stochastic dual coordinate ascent," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2013, pp. 378–385.

[46] D. Blatt, A. O. Hero, and H. Gauchman, "A convergent incremental gradient method with a constant step size," *SIAM Journal on Optimization*, vol. 18, no. 1, pp. 29–51, Jan. 2007.

[47] M. Gürbüzbalaban, A. Ozdaglar, and P. A. Parrilo, "On the convergence rate of incremental aggregated gradient algorithms," *SIAM Journal on Optimization*, vol. 27, no. 2, pp. 1035–1048, Jan. 2017.

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

[49] Z.-Q. Luo and P. Tseng, "On the convergence of the coordinate descent method for convex differentiable minimization," *Journal of Optimization Theory and Applications*, vol. 72, no. 1, pp. 7–35, Jan. 1992.

[50] Y. Nesterov, "Efficiency of coordinate descent methods on huge-scale optimization problems," *SIAM Journal on Optimization*, vol. 22, no. 2, pp. 341–362, Jan. 2012.

[51] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin, "Parallel coordinate descent for l1-regularized loss minimization," in *International Conference on Machine Learning (ICML)*, 2011, pp. 321–328.

[52] P. Richtárik and M. Takáč, "Parallel coordinate descent methods for big data optimization," *Mathematical Programming*, vol. 156, no. 1-2, pp. 433–484, Apr. 2015.

[53] J. Liu and S. J. Wright, "Asynchronous stochastic coordinate descent: Parallelism and convergence properties," *SIAM Journal on Optimization*, vol. 25, no. 1, pp. 351–376, Jan. 2015.

[54] Z. Peng, Y. Xu, M. Yan, and W. Yin, "ARock: An algorithmic framework for asynchronous parallel coordinate updates," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. A2851–A2879, Jan. 2016.

[55] F. Hanzely, K. Mishchenko, and P. Richtárik, "Sega: Variance reduction via gradient sketching," pp. 2082–2093, 2018.

[56] F. Hanzely, D. Kovalev, and P. Richtárik, "Variance reduced coordinate descent with acceleration: New method with a surprising application to finite-sum problems," *arXiv preprint arXiv:2002.04670*, 2020.

[57] F. Pedregosa, R. Leblond, and S. Lacoste-Julien, "Breaking the nonsmooth barrier: A scalable parallel method for composite optimization," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017, pp. 56–65.

[58] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous parallel stochastic gradient for nonconvex optimization," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2015, pp. 2737–2745.

[59] C. M. De Sa, C. Zhang, K. Olukotun, and C. Ré, "Taming the wild: A unified analysis of Hogwild-style algorithms," pp. 2674–2682, 2015.

[60] L. Nguyen, P. H. Nguyen, M. Dijk, P. Richtárik, K. Scheinberg, and M. Takac, "SGD and Hogwild! convergence without the bounded gradients assumption," in *International Conference on Machine Learning (ICML)*, vol. 80, 2018, pp. 3750–3758.

[61] P. Tseng and S. Yun, "Incrementally updated gradient methods for constrained and regularized optimization," *Journal of Optimization Theory and Applications*, vol. 160, no. 3, pp. 832–853, 2014.

[62] H. R. Feyzmahdavian, A. Aytekin, and M. Johansson, "A delayed proximal gradient method with linear convergence rate," in *International Workshop on Machine Learning for Signal Processing (MLSP)*. IEEE, Sep. 2014.

[63] A. Aytekin, H. R. Feyzmahdavian, and M. Johansson, "Analysis and implementation of an asynchronous optimization algorithm for the parameter server," Oct. 2016, arXiv: 1610.05507.

[64] H. R. Feyzmahdavian and M. Johansson, "On the convergence rates of asynchronous iterations," in *Conference on Decision and Control (CDC)*. IEEE, Dec. 2014.

[65] J. Mairal, "Incremental majorization-minimization optimization with application to large-scale machine learning," *SIAM Journal on Optimization*, vol. 25, no. 2, pp. 829–855, 2015.

[66] A. Defazio, J. Domke *et al.*, "Finito: A faster, permutable incremental gradient method for big data problems," pp. 1125–1133, 2014.

[67] A. Mokhtari, M. Gurbuzbalaban, and A. Ribeiro, "Surpassing gradient descent provably: A cyclic incremental method with linear convergence rate," *SIAM Journal on Optimization*, vol. 28, no. 2, pp. 1420–1447, 2018.

[68] K. Mishchenko, F. Iutzeler, J. Malick, and M.-R. Amini, "A delay-tolerant proximal-gradient algorithm for distributed learning," *International Conference on Machine Learning*, pp. 3587–3595, 2018.

[69] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford, "A reliable effective terascale linear learning system," *Journal of Machine Learning Research*, vol. 15, pp. 1111–1133, 2014.

[70] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," in *International Conference on Learning Representations (ICLR)*, 2016, workshop Track.

[71] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-based computation of aggregate information," in *Symposium on Foundations of Computer Science*. IEEE Computer Society, 2003.

[72] A. Nedić and A. Olshevsky, "Stochastic gradient-push for strongly convex functions on time-varying directed graphs," *Transactions on Automatic Control*, vol. 61, no. 12, pp. 3936–3947, Dec. 2016.

[73] M. Assran, N. Loizou, N. Ballas, and M. Rabbat, "Stochastic gradient push for distributed deep learning," in *International Conference on Machine Learning (ICML)*, vol. 97, 2019, pp. 344–353.

[74] M. Assran and M. Rabbat, "An empirical comparison of multi-agent optimization algorithms," in *Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, Nov. 2017.

[75] ——, "Asynchronous subgradient-push," Mar. 2018, arXiv: 1803.08950.

[76] J. Zhang and K. You, "AsySPA: An exact asynchronous algorithm for convex optimization over digraphs," *IEEE Transactions on Automatic Control*, 2019.

[77] A. Olshevsky, I. C. Paschalidis, and A. Spiridonoff, "Robust asynchronous stochastic gradient-push: asymptotically optimal and network-

independent performance for strongly convex functions," Nov. 2018, arXiv: 1811.03982.

[78] Y. Tian, Y. Sun, and G. Scutari, "ASY-SONATA: Achieving linear convergence in distributed asynchronous multiagent optimization," in *Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, Oct. 2018.

[79] ——, "Asynchronous decentralized successive convex approximation," *arXiv preprint arXiv:1909.10144*, 2019.

[80] P. Bianchi, W. Hachem, and F. Iutzeler, "A coordinate descent primal-dual algorithm and application to distributed asynchronous optimization," *IEEE Trans. Automatic Control*, vol. 61, no. 10, pp. 2947–2957.

[81] J. Lu, H. Feyzmahdavian, and M. Johansson, "Dual coordinate descent algorithms for multi-agent optimization," in *European Control Conference (ECC)*, Jul. 2015, pp. 715–720.

[82] M. Cao, A. S. Morse, and B. D. O. Anderson, "Reaching a consensus in a dynamically changing environment: A graphical approach," *SIAM Journal on Control and Optimization*, vol. 47, no. 2, pp. 575–600, Jan. 2008.

[83] K. I. Tsianos and M. G. Rabbat, "Distributed consensus and optimization under communication delays," in *Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, Sep. 2011.

[84] C. N. Hadjicostis and T. Charalambous, "Average consensus in the presence of delays in directed graph topologies," *Transactions on Automatic Control*, vol. 59, no. 3, pp. 763–768, Mar. 2014.

[85] T. Charalambous, Y. Yuan, T. Yang, W. Pan, C. N. Hadjicostis, and M. Johansson, "Distributed finite-time average consensus in digraphs in the presence of time delays," *Transactions on Control of Network Systems*, vol. 2, no. 4, pp. 370–381, Dec. 2015.

[86] J. Wolfowitz, "Products of indecomposable, aperiodic, stochastic matrices," *Proceedings of the American Mathematical Society*, vol. 14, no. 5, pp. 733–737, May 1963.

[87] E. Seneta, *Non-negative Matrices and Markov Chains*. Springer, 1981.

[88] A. Nedić, A. Olshevsky, and W. Shi, "Achieving geometric convergence for distributed optimization over time-varying graphs," *SIAM Journal on Optimization*, vol. 27, no. 4, pp. 2597–2633, Jan. 2017.

[89] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NeurIPS 2017 Workshop on Automatic Differentiation*, 2017.

[90] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Jun. 2016.

[91] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, Apr. 2015.

[92] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu, "Can decentralized algorithms outperform centralized algorithms? A case study for decentralized parallel stochastic gradient descent," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017, pp. 5330–5340.

[93] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous decentralized parallel stochastic gradient descent," in *International Conference on Machine Learning (ICML)*, vol. 80, 2018, pp. 3043–3052.

[94] J. Wang, V. Tantia, N. Ballas, and M. Rabbat, "SlowMo: Improving communication-efficient distributed SGD with slow momentum," Oct. 2019, arXiv: 1910.00643.

[95] A. Nedić, A. Olshevsky, and M. G. Rabbat, "Network topology and communication-computation tradeoffs in decentralized optimization," *Proceedings of the IEEE*, vol. 106, no. 5, pp. 953–976, May 2018.

[96] A. Olshevsky, I. C. Paschalidis, and S. Pu, "Asymptotic network independence in distributed optimization for machine learning," Jun. 2019, arXiv: 1906.12345.

[97] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," Dec. 2014, arXiv: 1412.6980.

[98] G. Hinton, N. Srivastava, and K. Swersky, "Lecture notes in neural networks for machine learning," Feb. 2014.

[99] L. Cannelli, F. Facchinei, V. Kungurtsev, and G. Scutari, "Asynchronous parallel algorithms for nonconvex optimization," *Mathematical Programming*, Jun. 2019.

[100] F. Mansoori and E. Wei, "Superlinearly convergent asynchronous distributed network newton method," in *Annual Conference on Decision and Control (CDC)*. IEEE, Dec. 2017.

[101] H. Zhang, C.-J. Hsieh, and V. Akella, "HogWild++: A new mechanism for decentralized asynchronous stochastic gradient descent," in *International Conference on Data Mining (ICDM)*. IEEE, Dec. 2016.