The Impact of Synchronization in Parallel Stochastic Gradient Descent

Karl Bäckström(✉), Marina Papatriantafilou(✉), and Philippas Tsigas(✉)

Department of Computer Science and Engineering,
Chalmers University of Technology, Gothenburg, Sweden
{bakarl,ptrianta,tsigas}@chalmers.se

Abstract. In this paper, we discuss our and related work in the domain of efficient parallel optimization, using Stochastic Gradient Descent, for fast and stable convergence in prominent machine learning applications. We outline the results in the context of aspects and challenges regarding synchronization, consistency, staleness and parallel-aware adaptiveness, focusing on the impact on the overall convergence.

Keywords: Stochastic gradient descent · Lock-free · Machine Learning

1 Introduction

Among the most prominent methods used for common optimization problems in data analytics and Machine Learning (ML), especially for problems tackling large datasets using *Artificial Neural Networks* (ANN), is the widely used Stochastic Gradient Descent (SGD) optimization method, introduced by Augustin-Louis Cauchy back in 1847. By iteratively processing data, SGD enables Artificial Neural Network (ANN) training, Logistic Regression, Support Vector Machines, and other ML methods. Let us use ANNS as an example. ANNs build on the concept of biological neurons, where the model of a neuron is called a perceptron. Several perceptions can be used in connected layers, forming an ANN that can be trained on different ML tasks. A perceptron consists of a weight, a bias, and a non-linear activation function. The network will produce an output for a given input, and this output can be compared to an expected output through a loss function. From this point, the training process becomes a numerical optimization problem where the sets of weights and biases producing the lowest error are the target. On one hand large datasets generally allow for more complex tasks and better generalizing capabilities of the model; on the other hand they demand larger computational time. Parallelism is one of the major way for both speeding up model training and handling large datasets.

This work is supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP), Knut and Alice Wallenberg Foundation, the SSF proj. “FiC” nr. GMT14-0032 and the VR proj. with nr. 2021-05443.

© Springer Nature Switzerland AG 2022
R. Bapi et al. (Eds.): ICDCIT 2022, LNCS 13145, pp. 60–75, 2022.
https://doi.org/10.1007/978-3-030-94876-4_4

Many algorithms for ML are far from trivial to parallelize. Taking SGD as the primary example, but any other iterative algorithm as well, usually every iteration requires the computation of the previous iteration to be completed, and available to be used in the next. As a consequence, parallelization would impose either that threads work in parallel only during each individual iteration and synchronize at the end in a lock-step manner, or relax the semantics of the original algorithm. These two main approaches to parallel SGD came to be known as *synchronous* and *asynchronous* parallel SGD, respectively, with fundamentally different properties in scalability, convergence and applicability.

It is easy to realize that synchronous parallelization suffers limitations in scalability due to the fact that each iteration is only as fast as the slowest contributing thread. Hence, slow threads, i.e. stragglers, present particularly in heterogeneous computing environments, can significantly impact the convergence time. Asynchronous approaches alleviate this limitation, showing improved scalability in some applications. However, the reduced inter-thread coordination that asynchrony entails breaks the semantics of the original SGD algorithm, and hence introduces several questions, among the most important is how the convergence time of SGD is affected. Moreover, the degree of synchronization that is still required, such as when accessing shared variables, becomes a focal point. For example, degradation in convergence due to lock-free inconsistent access is a risk, depending on the application. This can be avoided with consistency-enforcing mechanisms, one option being locking, however it is unclear whether or not it is worth the computational overhead it introduces in practice.

In this paper, we survey our work together with other recent related results in the domain of efficient parallel optimization with SGD for fast and stable convergence in prominent machine learning applications. We explore aspects of synchronization, consistency, staleness and parallel-aware adaptiveness, focusing on the impact on the overall convergence.

2 Preliminaries

2.1 SGD and Machine Learning

Machine learning with SGD is at its core an optimization problem:

$$\underset{\theta}{\text{minimize}} \quad f_D(\theta) \tag{1}$$

for a non-negative function $f : \mathbb{R}^d \rightarrow \mathbb{R}^+$. In machine learning (ML) applications, $\theta_t \in \mathbb{R}^d$ typically represents an encoding of the *learned knowledge*, and f_D quantifies the performance error of the model θ_t on the dataset D at iteration t. Solutions may be found using Stochastic Gradient Descent (SGD), defined as repeating the following with data mini-batches $B \in D$ sampled randomly:

$$\theta_{t+1} = \theta_t - \eta \widetilde{\nabla} f_B(\theta_t) \tag{2}$$

where $\widetilde{\nabla} f_B(\theta_t)$ and an unbiased estimate of the true gradient. The choice of the initialization point θ_0 is chosen at random according to some distribution, which as one might expect may significantly impact the convergence [32].

The negative gradient of a function constitutes the direction of *steepest descent*, resulting in a trajectory corresponding to the slope of the target function. The iteration (2) is repeated until a solution θ^* of sufficient quality is found, i.e. $f_D(\theta^*) < \epsilon$, referred to as ϵ-convergence.

The original deterministic counterpart Gradient Descent (GD) to SGD simply lets $B = D$, i.e. considers the entire dataset in every iteration. The stochastic element of random data subsampling in SGD entails two major benefits, namely that (i) sampling and processing only small mini-batches enables significantly faster iterations and (ii) the algorithm is effective on also non-convex target functions, as opposed to GD. However, SGD introduces a new hyper-parameter, the batch size b, which introduces *stochasticity* or *noise* in the convergence. While a certain degree of noise is necessary for enabling convergence in non-convex settings, it can be fatal when too high, causing endless sporadic oscillation about the initialization point θ_0. In practice, b consequently requires careful tuning. An established method for reducing such oscillation, while maintaining the stochasticity as necessary, is *Momentum-SGD* (MSGD), defined as follows:

$$\theta_{t+1} \leftarrow \theta_t + \mu(\theta_t - \theta_{t-1}) - \eta \widetilde{\nabla} f_B(\theta_t) \tag{3}$$

for some momentum parameter $\mu \in [0, 1]$. Momentum can accelerate the convergence of SGD in many practical settings, especially so for target functions which are irregular and asymmetric in shape, forming narrow valleys. Such irregularities are in particular known to arise in *deep learning* (DL) applications.

2.2 Performance Metrics

The implementation of any algorithm affects its performance and usefulness in practice. Considering SGD, or any iterative optimization algorithm, the performance is influenced by many implementation aspects and system features. As described in [22] a useful decomposition of the performance is to consider the *statistical* and *computational* efficiency, defined as follows.

1. *statistical efficiency* measures the number of SGD iterations required until reaching a solution of sufficient quality, ϵ-convergence
2. *computational efficiency* measures the number of iterations per time unit

The overall *convergence rate*, i.e. the wall-clock time until ϵ-convergence, is the most relevant in practice, and is essentially the product [22]:

$$\text{convergence rate} = \text{statistical efficiency} \times \text{computational efficiency}$$

Consequently, when proposing new algorithms (or altering existing ones) in this application domain that potentially change the *computational efficiency*, it is not sufficient to evaluate the invention by measuring only the *statistical efficiency*,

i.e. counting the iterations until convergence. One must in general consider these metrics in conjunction, and measure the overall convergence rate. Ideally, they should also be measured separately, as this is the only way to truly understand from where potential improvements originate.

These metrics become particularly important in parallel algorithms for iterative optimization, since the parallelization method can have significant impact on the computational and statistical efficiency, as we shall see in the following.

3 Parallel SGD

While parallelism can improve computational efficiency, simply by managing to apply a greater number of updates in each unit of time, the impact on the statistical efficiency, and thereby the overall convergence rate, is unpredictable. Parallelization is consequently not trivial, and requires synchronization in every iteration (prior to applying an update) in order to not break the original sequential semantics of SGD. Alternatively, threads can execute the SGD algorithm, i.e. accessing and updating the shared state θ, asynchronously, although this might not conform to the sequential semantics.

These approaches correspond to two main directions of methods for parallel SGD, referred to as *synchronous* and *asynchronous*.

Most methods mentioned in this paper were originally introduced in the centralized shared-state context, either on a shared-memory parallel system or a distributed one with one node acting as a parameter server, which sequentializes updates. Most approaches can be naturally generalized to different computing infrastructure, and also to their decentralized counterpart. However, in this paper we retain the focus on asynchronous SGD in the context of shared-memory parallel systems, but keep in mind distributive and decentralizing generalizations, with occasional remarks on that topic.

3.1 Synchronous SGD

Synchronous SGD (*SyncSGD*) is a lock-step data-parallel version of SGD where threads or nodes access the shared θ_t at an iteration t, then compute gradients based on individual randomly sampled data-batches, see Fig. 1. The threads synchronize by averaging the resulting gradients before taking a global step according to (2) [37]. In the original version, *SyncSGD* is statistically equivalent to sequential SGD with larger mini-batch size [13], and can hence be considered a method for accelerated gradient computation. From this perspective, the *SyncSGD* approach does not break the semantics of the sequential SGD algorithm, and the vast empirical results and theoretical convergence guarantees in the literature entail predictable performance of *SyncSGD*. From a scalability perspective, since each SGD iteration is only as fast as the slowest contributing thread, the presence of slower threads, i.e. *stragglers*, becomes a bottleneck. A comprehensive overview of methods along this approach is provided in [8].

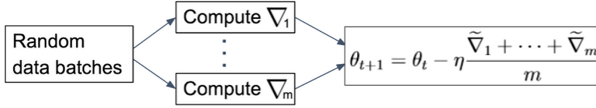

Fig. 1. In *SyncSGD* the threads' individual gradients are aggregated by averaging, after which a global iteration is performed. *SyncSGD* essentially corresponds to parallelization on the gradient computation level.

Stale-synchronous parallel (SSP) relaxes the strict synchronous semantics of *SyncSGD*, allowing faster threads to asynchronously compute a bounded number of SGD steps based on a local version of the state before synchronizing [14]. The method is useful in heterogeneous computing systems, where stragglers are kept in check. SSP has been proven useful for distributed DL applications, e.g. in [36] where a method for dynamically adjusting the *staleness* (the number of applied updates between the read vector and the one where the update is applied on) threshold is proposed, enabling improvements in computational efficiency.

From a progress perspective, note that the original *SyncSGD* as well as SSP provide weak progress guarantees, since in the presence of halting threads, the system as a whole will halt indefinitely in the synchronization step. This is partially addressed by *n-softsync* [35], a further relaxed variant of *SyncSGD* with partial synchronization, requiring only a fixed number n of threads to contribute a gradient at the synchronization point. Contrary to SSP, there is no bound on the maximum staleness. Introduced originally in the context of centralized distributed SGD with a parameter server [13,35], the recent work [18] implements similar semantics in a decentralized setting utilizing a `partial-allreduce` primitive which atomically applies the aggregated updates and redistributes the result.

3.2 Asynchronous Parallel SGD

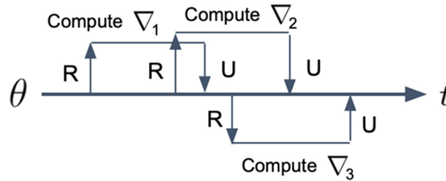

Fig. 2. *AsyncSGD* parallelizes the SGD iterations, allowing asynchronous read (R) and update (U) operations on the shared state.

Asynchronous parallel SGD (*AsyncSGD*) removes the gradient averaging synchronization step, allowing threads to access and update the shared state asynchronously. Consequently, while an update is being computed by one thread, there can be concurrent updates applied by other ones, i.e. *AsyncSGD* follows:

$$\theta_{t+1} \leftarrow \theta_t - \eta \widetilde{\nabla} f(v_t) \tag{4}$$

where $v_t = \theta_{t-\tau_t}$ is a thread's *view* of θ and τ_t is the number of concurrent updates, which defines the staleness. Updates are consequently generally computed based on states which are older than the ones on which the updates are applied (Fig. 2). The resulting impact on the convergence is referred to as *asynchrony-induced noise*, and affects, together with the overall distribution of the stalenesses τ_t, the statistical efficiency.

AsyncSGD enables increased computational efficiency with higher parallelism, up to a point where contention due to concurrent shared-memory access attempts becomes severe. We denote the corresponding number of threads by m_C^*; at this point the system stagnates and additional computing threads provide no additional speedup. In addition, the presence of staleness in *AsyncSGD* causes decay in statistical efficiency from the asynchrony-induced noise, which grows as more threads are introduced. Over-parallelization may thereby not only be redundant, but in fact harm the statistical efficiency, with potentially dire consequences on the overall convergence. There is hence a trade-off between computational and statistical efficiency, which in practice requires careful tuning of the level of parallelism (number of threads) m. The appropriate choice of m depends on the properties of the optimization problem itself, as well as the other hyper-parameters, e.g. the step size η and the batch size b.

***AsyncSGD* and Momentum.** The research direction of asynchronous iterative optimization is not new, and sparked due to the works by Bertsekas and Tsitsiklis [9] in 1989. More recently, Chaturapruek et al. [11] show that, under several analytical assumptions such as convexity (linear and logistic regression), the convergence of *AsyncSGD* is not significantly affected by asynchrony and that the noise introduced by staleness is asymptotically negligible compared to the noise from the stochastic gradients. In [19] Lian et al. show that these assumptions can be partially relaxed, and it is shown that convergence is possible for non-convex problems, however with a bounded number of threads, and assuming bounded staleness. Several works have followed, aiming at understanding the impact of asynchrony on the convergence. In [26] Mitliagkas et al. show that under certain stochastic staleness models, asynchronous parallelism has an effect on convergence similar to momentum. This work is extended in parts of [6], which introduces models which capture the dynamics of the system more accurately, leading to alternate conclusions, as well as means to improve the statistical efficiency by asynchrony-awareness. In [23] Mania et al. model the algorithmic effect of asynchrony in *AsyncSGD* by perturbing the stochastic iterates with bounded noise. Their framework yields convergence bounds which, as described in the paper, are not tight, and rely on strong convexity of the target function. In the recent [2] Alistarh et al. introduced the concept of bounded divergence between the parameter vector and the threads' view of it, proving convergence bounds for convex and non-convex problems.

***AsyncSGD* and Lock-Freedom.** Hogwild! [28], introduced by Niu et al., implements *AsyncSGD* with *lock-free* accesses to the shared state θ. This is achieved in a straightforward manner by allowing uncoordinated, component-wise atomic access to the shared state θ_t, as opposed to traditional consistency-preserving access implemented with locks. This significantly reduced the computational synchronization overhead, and was shown to achieve near-optimal convergence rates, however assuming sparse updates. *AsyncSGD* with sparse or component-wise updates has since been a popular target of study due to the performance benefits of lock-freedom [27,29]. De Sa et al. [12] introduced a framework for analysis of Hogwild!-style algorithms for sparse problems. The analysis was extended in [3], showing that due to the lack of θ-*consistency* (shared state consistency for shared state θ) of Hogwild! (i.e. read operation includes partial updates) the convergence bound increases with a magnitude of $\sqrt{d}$ when relaxing the sparsity assumption. This indicates in particular higher statistical penalty for high-dimensional problems and motivates development of algorithms which, while enjoying the computational benefits of lock-freedom, also ensure consistency, in particular for high-dimensional problems such as DL. This is the main focus of [10], where a consistency-preserving lock-free implementation of *AsyncSGD* for DL is introduced. In [22] a detailed study of parallel SGD focusing on Hogwild! and a new, GPU-implementation, is conducted, focusing on convex functions, with dense and sparse data sets and a comparison of different computing architectures.

***AsyncSGD* for DL.** In [33] the focus is the fundamental limitation of data parallelism in ML. They observe that the limitations are due to concurrent SGD parameter accesses, during ML training, usually diminishing or even negating the parallelization benefits provided by additional parallel compute resources. To alleviate this, they propose the use of static analysis for identification of data that do not cause dependencies, for parallelizing their access. They do this as part of a system that uses Julia, a script language that performs just-in-time compilation. Their approach is effective and works well for e.g. Matrix factorization SGD. For DNNs, as they explain, their work is not directly applicable, since in DNNs permitting "good" dependence violation is the common parallelization approach. Asynchronous SGD approaches for DNNs are scarce in the current literature. In the recent work [21], Lopez et al. propose a semi-asynchronous SGD variant for DNN training, however requiring a master thread synchronizing the updates through gradient averaging, and relying on atomic updates of the entire parameter vector, resembling more a shared-memory implementation of parameter server. In [31] theoretical convergence analysis is presented for *SyncSGD* with *once-in-a-while* synchronization. They mention the analysis can guide in applying *SyncSGD* for DL, however the analysis requires strong convexity of the target function. [15] proposes a consensus-based SGD algorithm for distributed DL. They provide theoretical convergence guarantees, also in the non-convex case, however the empirical evaluation is limited to iteration counting as opposed to wall-clock time measurements, with mixed performance positioning relative

to the baselines. In [20] a topology for decentralized parallel SGD is proposed, using pair-wise averaging synchronization.

Asynchrony-Adaptive SGD. Delayed optimization in asynchronous first-order optimization algorithms was analyzed initially in [1], where Agarwal et al. introduce step sizes which diminish over the progression of SGD, depending on the maximum staleness allowed in the system, but not adaptive to the actual delays observed. Adaptiveness to delayed updates during execution was proposed and analyzed in [24] under assumptions of gradient sparsity and *read* and *write* operations having the same relative ordering. A similar approach was used in [35], however for synchronous SGD with the *softsync* protocol. In [35] statistical speedup is observed in some cases for a limited number of worker nodes, however by using *momentum SGD*, which is not the case in their theoretical analysis, and step size decaying schedules on top of the staleness-adaptive step size. In [30], AdaDelay is proposed, which addresses a particular constrained convex optimization problem, namely training a logistic classifier with projected gradient descent. It utilizes a network of worker nodes computing gradients in parallel which are aggregated at a central parameter server with a step size that is scaled proportionally to the inverse staleness, τ^{-1} (τ denotes staleness defined as the number of applied concurrent updates). The staleness model in [30] is a uniform stochastic distribution, which implies a strict upper bound on the delays, making the system model partially asynchronous. [6] extends this line of research, exploring further the idea of adapting updates based on staleness, and studies in particular analytical foundations to motivate how.

4 Problems and Challenges with Parallel SGD

4.1 Scalability

Growing Batch Size. In *SyncSGD*, stragglers become a bottleneck, making every iteration only as fast as the slowest thread. This issue can however partially be reduced through relaxed semantics, such as SSP and the n-softsync protocol (see Sect. 3). Moreover, the convergence of *SyncSGD* under increasing parallelism is statistically equivalent to sequential SGD with a larger *mini-batch size* b [13], also shown in [6], which is a hyper-parameter that requires careful tuning depending on the problem. In particular, the convergence can be slower if b is too large [16,25]. As discussed in Sect. 3.1, this indicates limited scalability, as over-parallization will impose large-batch properties, which in some cases slows down the convergence [13]. This motivates further exploration of asynchronous parallelism for scalability.

Staleness. *AsyncSGD* eliminates many scalability bottlenecks of *SyncSGD* due to reduced inter-thread coordination, however this also introduces other challenges related to asynchrony. As discussed in Sect. 3.2, asynchronous access to and update of the shared state leads to staleness due to the fact that updates may occur by threads concurrently to the gradient computation. The updates that are applied are in fact rarely in practice based on the latest shared state, as

described by (4). For problems satisfying assumptions on convexity, smoothness and bounded gradients, staleness has little impact on the convergence of *AsyncSGD* [11]. However, for a wider class of problems, staleness can have significant impact. In particular for problems not conforming to e.g. convexity assumptions, such as the recently relevant DL applications. Crucial steps toward understanding how convergence is affected in *AsyncSGD* due to staleness were taken by Mitliagkas et al. [26], explicitly quantifying the impact of concurrency, under a certain statistical staleness model. The results indicate that the influence of asynchrony has an effect similar to momentum in SGD, and a reduced step size. This analysis is extended in [6], proposing models better capturing the staleness dynamics, and showing that the momentum effect grows and the step size reduces monotonically as the parallelism is increased. This indicates a scalability limitation in convergence, which however can be partially alleviated by using a staleness-adaptive step size.

Progress and Consistency Guarantees. As previously mentioned, read and update operations on the shared state θ become focal in *AsyncSGD*, since they constitute the remaining synchronization steps in the otherwise asynchronous algorithm. There must be primitives in place to handle concurrent attempts to read and update by several threads, and these become bottlenecks for scalability at sufficiently high levels of parallelism. Traditionally, a separate thread or node acting as a parameter server is responsible for providing the latest parameter state to workers, as well as processing contributing gradients, sequentializing the updates [17]. To efficiently utilize multi-core systems, this was extended to shared-memory implementations [19,28,29]. The access to the shared state is then scheduled by the operating system, and regulated by some synchronization method, such as locking, to ensure consistency in case of concurrent read and update attempts. However, locks can be relatively computationally expensive, in particular when the gradient computation step itself incurs little latency. In addition, the total time spent on waiting for locks grows as more threads are introduced to the system, potentially making it scalability bottleneck. By allowing completely uncoordinated component-wise atomic read and update operations, i.e. Hogwild! [28], such contention is eliminated, allowing significant speedup for sparse optimization problems in particular. However, for other problems, Hogwild! introduces inconsistency when read and update operations occur concurrently, with unpredictable impact on the convergence. There is currently a lack of methods providing a middle-ground solutions in the literature in the realm in between these two endpoints of the synchronization spectrum, i.e. the consistency-enforcing lock-based *AsyncSGD* and the lock-free inconsistency-prone Hogwild!. This spectrum is explored further in [10], and *Leashed-SGD* is proposed as a middle-ground solution. *Leashed-SGD* ensures consistency in arbitrarily dense problems, while enjoying the benefits of lock-freedom, reducing computational synchronization bottlenecks (see Sect. 5.1).

Memory Consumption. An additional aspect of scalability to consider is memory consumption; standard *AsyncSGD* implementations in the literature require each thread to copy the entire shared state θ prior to its individual

gradient computation. The result of the computation, i.e. the stochastic gradient, is of the same dimension d as θ, and is stored locally until applied to the shared state in an SGD iteration. The magnitude of d varies, however in DL applications it is often in the magnitude of hundreds of thousands, sometimes millions, which is why the memory consumption of the *AsyncSGD* implementation needs to be carefully considered. This aspect is discussed further in [10], and possible improvements are explored.

4.2 Convergence Under Asynchrony

Staleness. The staleness that arises in *AsyncSGD* due to parallelism significantly impacts the statistical efficiency of the convergence; it has been shown analytically that the number of SGD iterations to ϵ-convergence increases linearly in the maximum staleness [3,12]. Hence, only if the gains in computational efficiency from parallelism are sufficiently great, will there be an overall improvement in wall-clock time until ϵ-convergence. In addition, inconsistent synchronization as in HOGWILD! potentially incurs further statistical penalty; the expected number of iterations required increases linearly in $\sqrt{d}$ [3]. Subsequently, there are challenges in understanding whether it is worth the computational overhead to ensure consistency for a given problem, and which synchronization primitives are appropriate to utilize.

Synchronization. As a consequence of Amdahl's law [5], when there is a synchronization overhead, the achievable speedup is bounded. In the context of *AsyncSGD*, this applies in particular for the computational efficiency, i.e. how many SGD updates can be applied in a given time unit. This implies that there is a *computational saturation point* m_C^* for which additional threads will not provide additional significant computational speedup. For this statement, as well as the ones to follow in this paragraph, empirical evidence is provided in [10]. Moreover, due to the presence of staleness there is a degradation of statistical efficiency coupled to parallelism in *AsyncSGD* [23,33]. Hence, as more threads are introduced to the system, more iterations are required until reaching ϵ-convergence. At some level of parallelism, which we refer to as the *system saturation point* m_S^*, additional threads will no longer reduce the wall-clock time to ϵ-convergence, and might instead even increase it. It can be concluded that $m_S^* \leq m_C^*$ from a simple argument of contradiction, assuming that statistical efficiency degrades with higher parallelism. This assumption is in accordance with results in previous literature [3,12], and explored further in [6,10] There are substantial challenges in understanding the appropriate range of the number m of threads in order to (i) fully utilize the parallel computation ability of the system and (ii) avoid over-parallelization, potentially harming or completely obstructing convergence. Ideally an implementation of *AsyncSGD* feature resilience to tuning, providing reliable and fast convergence over a broad spectrum of parallelism, towards which [10] takes significant steps (see Sect. 5.2).

4.3 Benchmarking and Evaluation

Standardization. There are significant challenges in conducting empirical evaluations and comparisons which are useful and fair within the domain of parallel SGD, for several reasons: Firstly, there are several metrics of interest related to convergence of SGD, the measurements of which must be effectively aggregated as to show the overall performance. Traditionally, in ML the statistical efficiency is the metric most used, i.e. the number of SGD iterations until reaching sufficient performance, i.e. ϵ-convergence. However, when improvements in statistical efficiency is achieved by altering the underlying algorithm, this potentially alters the computational efficiency, i.e. the number of SGD iterations per time unit. In such cases, it is hence necessary that evaluations take this into consideration, and ideally provide measurements of the overall convergence rate, i.e. the wall-clock time until converging to a solution of sufficient quality. Secondly, the domain of shared-memory parallel SGD lacks established universal procedures for benchmarking, leaving the task of setting up an appropriate test environment to the individual authors. The domain contains a wide spectrum of questions, ranging from efficient communication protocols [4] in wide distributed DL networks to exploring the impact of progress guarantees and synchronization in shared data structures [3,28]. This renders the task of designing a universal benchmarking platform for parallel SGD including such universal procedures immensely difficult, if not impossible. The Deep500 framework [7] takes important steps in providing such an environment, although it focuses primarily on higher-level distributed SGD. For instance, the framework provides a Python interface for development, which does not facilitate exploration of for instance efficient shared data structures for fine-grained synchronization and mechanisms for memory management.

Hyper-parameter Dependencies. Another key issue in benchmarking parallel SGD for machine learning is the inherent dependency between parallelism and various hyper-parameters crucial for achieving convergence [13,26], some of the most important being the step size η and the mini-batch size b. As mentioned above, it is known that higher parallelism in *SyncSGD* exhibits similar convergence properties as sequential SGD with a larger batch size. As more threads or nodes are introduced to the system, the scalability of *SyncSGD* can hence appear to be limited due to the statistical penalty from a too large value of b. This can be avoided by choosing a sufficiently small initial b for each thread or node, which will then instead give the appearance of high scalability, but only until a certain level of parallelism [13]. It is hence of interest in such evaluations to provide empirical evidence from test scenarios that indicate the general ability of the proposed method to scale independently of hyper-parameter choices. Analogously, for *AsyncSGD*, there is delicate interplay between the step size η and the staleness distribution, stemming from the fact that stale updates correspond to gradients based on old views of the state, and are applied with a coarsity proportional to η [24,34]. A smaller η implies less impact on the convergence per update, hence tends to tolerate updates with higher staleness, and subsequently higher levels of parallelism. This can give the appearance of good

scalability, showing speedup for a larger number of threads. It is in this case also of interest to provide empirical results that indicate scalability independently of hyper-parameters, such as η, for instance by testing for several choices of η.

In summary, there are challenges in establishing evaluation methodologies, making fair and useful comparisons between methods difficult. This is mainly due to the wide span of research questions in the domain. A collective strive towards standardized benchmarking platform for various methodological aspects is imperative. In addition, the dependence of the performance and scalability of parallel SGD algorithms on various hyper-parameters, such as step size η and batch size b, complicate empirical evaluations.

5 Our Work on Parallelizing SGD

5.1 Convergence of Staleness-Adaptive SGD

The scalability limitations of traditional synchronous parallel SGD highlighted in Sect. 4.1 motivates further exploration of asynchronous parallelization, i.e. *AsyncSGD* which has shown promising improvements in ability to scale for many applications. The degradation of statistical efficiency due to staleness is however a limiting factor, forcing the user to carefully tune the level of parallelism in order to maintain an actual overall speedup in convergence rate, as also highlighted in Sect. 4.1. In order to address this issue, we first propose methods to statistically model the behaviour of staleness in *AsyncSGD*. The models, which are proposed based on reasoning of the dynamics of the algorithm and its dependency on scheduling, capture the staleness distribution in practice to a high degree of precision, and more accurately than models previously proposed in the literature.

Based on the proposed staleness models, we provide analytical results that quantify the side-effect of asynchrony on the statistical efficiency. Moreover, our approach enables derivation of a staleness-adaptive step size, referred to as *MindTheStep-AsyncSGD*, which provably reduces this side-effect, and in expectation can, depending on the rate of adaptiveness, alter it into the more desired behaviour of momentum. We prove also that the staleness-adaptive step size is efficiently computable, ensuring minimal additional synchronization overhead for maximal scalability capability, as described in Sect. 4.2. We provide an empirical evaluation of the proposed staleness models and the adaptive step size for a relevant use case, namely DL for image classification. The empirical results show in particular: (i) significantly improved accuracy in modelling the staleness with our proposed models, (ii) reduced penalty from asynchrony-induced noise, leading to up to a $\times 1.5$ speedup in convergence compared to baseline (standard *AsyncSGD* with constant step size) under high parallelism.

5.2 A Framework for Lock-Freedom and Consistency

Asynchronous parallelization of SGD, i.e. *AsyncSGD*, significantly reduces waiting compared to *SyncSGD*, as explained in the previous sections. However, the

remaining synchronization that is needed, in particular access to the shared state, becomes focal and constitute a possible bottleneck. Motivated by analytical results in previous literature that indicate great computational benefits of lock-freedom, however a statistical penalty from inconsistency and staleness, we propose *Leashed-SGD* (lock-free consistent asynchronous shared-memory SGD), which is an extensible framework supporting algorithmic lock-free implementations of *AsyncSGD* and diverse mechanisms for consistency, and for regulating contention. It utilizes an efficient on-demand dynamic memory allocation and recycling mechanism, which reduces the overall memory footprint. We provide an analysis of the proposed framework in terms of safety, memory consumption, and model the progression of parallel threads in the execution of SGD, which we use for estimating contention over time and confirming the potential of the built-in contention regulation mechanism to reduce the overall staleness distribution.

Among the analytical results for *Leashed-SGD*, we provide guarantees on lock-freedom and atomicity, safety and exhaustiveness and bounds on the memory consumption. Moreover, we model the progression of the algorithm over time, finding in particular fixed points in the system useful for estimating potential contention and the effect of the built-in contention-regulating mechanism.

We conduct an extensive empirical study of *Leashed-SGD* for Multi-Layer Perceptron (MLP) and Convolutional Neural Network (CNN) training for image classification. The empirical study focuses on scalability, dependence on hyperparameters, distribution of the staleness, and benchmarks the proposed framework compared to established baselines, namely lock-based *AsyncSGD* and HOGWILD!. We draw the following main conclusions:

1. *Leashed-SGD* provides significantly higher tolerance towards the level of parallelism, with fast and stable convergence for a wide spectrum, taking significant steps towards addressing the scalability challenges highlighted in Sect. 4.1. The baselines however require careful tuning of the number of threads in order to avoid tediously slow convergence and are more prone to completely failing or crashing executions.
2. The lock-free nature of *Leashed-SGD* entails a self-regulating balancing effect between latency and throughput, leading to an overall reduced staleness distribution, which in many instances is crucial for achieving convergence.
3. For MLP training we observe up to 27% reduced median running time for ϵ-convergence for *Leashed-SGD* compared to baselines, with similar memory footprint. For CNN training, we observe a $\times 4$ speedup for ϵ-convergence, with a memory footprint reduction with 17% on average.

For the empirical study, a modular and extensible C++ framework is developed with the purpose of facilitating development of shared-memory parallel SGD with varying synchronization mechanisms (https://github.com/dcs-chalmers/shared-memory-sgd). Hence, we take steps towards addressing the challenges (highlighted in Sect. 4.3) that the community faces regarding a general platform for further exploration of aspects of fine-grained synchronization in this domain.

6 Conclusions

There are significant challenges for asynchronous parallel SGD methods for machine learning to scale, due to (i) staleness and reduced update freshness and (ii) computational overhead from synchronization for shared-memory operations.

While higher parallelism in *AsyncSGD* enables more iterations per second, its inherent staleness and asynchrony-induced noise leads to an deteriorating statistical efficiency, requiring a growing number of iterations to achieve sufficient convergence. Understanding and modelling the dynamics of the staleness enables explicitly quantifying its side-effect on the convergence, towards which important steps were taken in [26], however under simplifying assumptions. Under a more practical system model, this analysis was extended in [6], and used to show how adaptiveness to staleness reduces asynchrony-induced noise, and thereby improves convergence. In addition, it allows derivation of the proposed staleness-adaptive *MindTheStep-AsyncSGD* which provably reduces this side-effect. The analytical results are confirmed in practice in [6], showing increased statistical efficiency in ANN training for image classification.

Relaxed inter-thread synchronization, with weak consistency requirements as in HOGWILD! [28], enables a straightforward way for achieving lock-freedom without consistency guarantees in shared state. The reduced computational overhead allows overall speedup for sparse problems, where inconsistency might have little impact [28] and asymptotic convergence bounds can be established. However, the inconsistency has implications on the statistical efficiency, as observed theoretically in [3] and confirmed in [6,10]. In [10] an interface is introduced, providing abstractions of operations on the shared state θ, utilized in the proposed lock-free *Leashed-SGD* framework, which includes an implementation that guarantees consistency and which is extensible to provide configurable consistency. The lock-free nature of *Leashed-SGD* has a self-regulating effect which avoids congestion under high parallelism, which by reducing the overall staleness distribution enables fast and stable convergence in contexts where the baselines fail. In this context, the dynamic memory allocation featured in *Leashed-SGD* allows for significantly reduced memory footprint, which is critical in particular for DL applications where the problems dimension can be in the order of millions.

References

1. Agarwal, A., Duchi, J.C.: Distributed delayed stochastic optimization. In: Advances in Neural Information Processing Systems, pp. 873–881 (2011)
2. Alistarh, D., Chatterjee, B., Kungurtsev, V.: Elastic consistency: a general consistency model for distributed stochastic gradient descent. arXiv preprint arXiv:2001.05918 (2020)
3. Alistarh, D., De Sa, C., Konstantinov, N.: The convergence of stochastic gradient descent in asynchronous shared memory. In: ACM Symposium on Principles of Distributed Computing, PODC 2018, pp. 169–178. ACM, New York (2018). https://doi.org/10.1145/3212734.3212763
4. Alistarh, D., Grubic, D., Li, J., Tomioka, R., Vojnovic, M.: QSGD: communication-efficient SGD via gradient quantization and encoding. In: Advances in Neural Information Processing Systems, pp. 1709–1720 (2017)

5. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, pp. 483–485 (1967)
6. Bäckström, K., Papatriantafilou, M., Tsigas, P.: MindTheStep-AsyncPSGD: adaptive asynchronous parallel stochastic gradient descent. In: 2019 IEEE International Conference on Big Data (Big Data), pp. 16–25. IEEE (2019)
7. Ben-Nun, T., Besta, M., Huber, S., Ziogas, A.N., Peter, D., Hoefler, T.: A modular benchmarking infrastructure for high-performance and reproducible deep learning. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 66–77. IEEE (2019)
8. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: an in-depth concurrency analysis. ACM Comput. Surv. (CSUR) **52**(4), 1–43 (2019)
9. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and Distributed Computation: Numerical Methods, vol. 23. Prentice Hall, Upper Saddle River (1989)
10. Bäckström, K., Walulya, I., Papatriantafilou, M., Tsigas, P.: Consistent lock-free parallel stochastic gradient descent for fast and stable convergence. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 423–432 (2021). https://doi.org/10.1109/IPDPS49936.2021.00051
11. Chaturapruek, S., Duchi, J.C., Ré, C.: Asynchronous stochastic convex optimization: the noise is in the noise and SGD don't care. In: Advances in Neural Information Processing Systems, pp. 1531–1539 (2015)
12. De Sa, C.M., Zhang, C., Olukotun, K., Ré, C., Ré, C.: Taming the wild: a unified analysis of Hogwild-style algorithms. In: Advances in Neural Information Processing Systems, vol. 28, pp. 2674–2682. Curran Associates, Inc. (2015). http://papers.nips.cc/paper/5717-taming-the-wild-a-unified-analysis-of-hogwild-style-algorithms.pdf
13. Gupta, S., Zhang, W., Wang, F.: Model accuracy and runtime tradeoff in distributed deep learning: a systematic study. In: 2016 IEEE 16th International Conference on Data Mining (ICDM), pp. 171–180. IEEE (2016)
14. Ho, Q., et al.: More effective distributed ml via a stale synchronous parallel parameter server. In: Advances in Neural Information Processing Systems, pp. 1223–1231 (2013)
15. Jiang, Z., Balu, A., Hegde, C., Sarkar, S.: Collaborative deep learning in fixed topology networks. In: Advances in Neural Information Processing Systems, pp. 5904–5914 (2017)
16. Keskar, N.S., Mudigere, D., Nocedal, J., Smelyanskiy, M., Tang, P.T.P.: On large-batch training for deep learning: generalization gap and sharp minima. arXiv:1609.04836 (2016)
17. Li, M., et al.: Scaling distributed machine learning with the parameter server. In: 11th Symposium on Operating Systems Design and Implementation, pp. 583–598 (2014)
18. Li, S., Ben-Nun, T., Girolamo, S.D., Alistarh, D., Hoefler, T.: Taming unbalanced training workloads in deep learning with partial collective operations. In: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 45–61 (2020)
19. Lian, X., Huang, Y., Li, Y., Liu, J.: Asynchronous parallel stochastic gradient for nonconvex optimization. In: Advances in Neural Information Processing Systems, pp. 2737–2745 (2015)
20. Lian, X., Zhang, W., Zhang, C., Liu, J.: Asynchronous decentralized parallel stochastic gradient descent. In: International Conference on Machine Learning, pp. 3043–3052. PMLR (2018)

21. Lopez, F., Chow, E., Tomov, S., Dongarra, J.: Asynchronous SGD for DNN training on shared-memory parallel architectures. In: International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1–4. IEEE (2020)
22. Ma, Y., Rusu, F., Torres, M.: Stochastic gradient descent on modern hardware: multi-core CPU or GPU? Synchronous or asynchronous? In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 1063–1072. IEEE (2019)
23. Mania, H., Pan, X., Papailiopoulos, D., Recht, B., Ramchandran, K., Jordan, M.I.: Perturbed iterate analysis for asynchronous stochastic optimization. SIAM J. Optim. **27**(4), 2202–2229 (2017)
24. McMahan, B., Streeter, M.: Delay-tolerant algorithms for asynchronous distributed online learning. In: Advances in Neural Information Processing Systems, vol. 27, pp. 2915–2923. Curran Associates, Inc. (2014). http://papers.nips.cc/paper/5242-delay-tolerant-algorithms-for-asynchronous-distributed-online-learning.pdf
25. Mishkin, D., Sergievskiy, N., Matas, J.: Systematic evaluation of convolution neural network advances on the ImageNet. Comput. Vis. Image Underst. **161**, 11–19 (2017)
26. Mitliagkas, I., Zhang, C., Hadjis, S., Ré, C.: Asynchrony begets momentum, with an application to deep learning. In: 54th Annual Allerton Conference on Communication, Control, and Computing, pp. 997–1004. IEEE (2016)
27. Nguyen, L.M., Nguyen, P.H., van Dijk, M., Richtárik, P., Scheinberg, K., Takáč, M.: SGD and Hogwild! convergence without the bounded gradients assumption. arXiv preprint arXiv:1802.03801 (2018)
28. Recht, B., Re, C., Wright, S., Niu, F.: Hogwild: a lock-free approach to parallelizing stochastic gradient descent. In: Advances in Neural Information Processing Systems (NIPS), vol. 24, pp. 693–701. Curran Associates, Inc. (2011)
29. Sallinen, S., Satish, N., Smelyanskiy, M., Sury, S.S., Ré, C.: High performance parallel stochastic gradient descent in shared memory. In: IEEE International Parallel and Distributed Processing Symposium, pp. 873–882. IEEE (2016)
30. Sra, S., Yu, A.W., Li, M., Smola, A.J.: AdaDelay: delay adaptive distributed stochastic convex optimization. arXiv preprint arXiv:1508.05003 (2015)
31. Stich, S.U.: Local SGD converges fast and communicates little. In: International Conference on Learning Representations (ICLR) (2019)
32. Sutskever, I., Martens, J., Dahl, G., Hinton, G.: On the importance of initialization and momentum in deep learning. In: International Conference on Machine Learning, pp. 1139–1147 (2013)
33. Wei, J., Gibson, G.A., Gibbons, P.B., Xing, E.P.: Automating dependence-aware parallelization of machine learning training on distributed shared memory. In: 14th EuroSys Conference 2019, pp. 1–17 (2019)
34. Zhang, W., Gupta, S., Lian, X., Liu, J.: Staleness-aware Async-SGD for distributed deep learning. arXiv preprint arXiv:1511.05950 (2015)
35. Zhang, W., Gupta, S., Lian, X., Liu, J.: Staleness-aware Async-SGD for distributed deep learning. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, pp. 2350–2356. AAAI Press (2016)
36. Zhao, X., An, A., Liu, J., Chen, B.X.: Dynamic stale synchronous parallel distributed training for deep learning. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 1507–1517, July 2019. https://doi.org/10.1109/ICDCS.2019.00150
37. Zinkevich, M., Weimer, M., Li, L., Smola, A.J.: Parallelized stochastic gradient descent. In: Advances in Neural Information Processing Systems, pp. 2595–2603 (2010)