Adaptiveness, Asynchrony, and Resource Efficiency in Parallel Stochastic Gradient Descent

Karl Bäckström

Adaptiveness, Asynchrony, and Resource Efficiency in Parallel Stochastic Gradient Descent

KARL BÄCKSTRÖM

Department of Computer Science & Engineering
Division of Networks and Systems
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

"We can only see a short distance ahead, but we can see plenty there that needs to be done."
–Alan Turing, 1950

Abstract

Accelerated digitalization and sensor deployment in society in recent years poses critical challenges for associated data processing and analysis infrastructure to scale, and the field of big data, targeting methods for storing, processing, and revealing patterns in huge data sets, has surged. Artificial Intelligence (AI) models are used diligently in standard Big Data pipelines due to their tremendous success across various data analysis tasks, however exponential growth in Volume, Variety and Velocity of Big Data (known as its three V's) in recent years require associated complexity in the AI models that analyze it, as well as the Machine Learning (ML) processes required to train them. In order to cope, parallelism in ML is standard nowadays, with the aim to better utilize contemporary computing infrastructure, whether it being shared-memory multi-core CPUs, or vast connected networks of IoT devices engaging in Federated Learning (FL).

Stochastic Gradient Descent (SGD) serves as the backbone of many of the most popular ML methods, including in particular *Deep Learning*. However, SGD has inherently sequential semantics, and is not trivially parallelizable without imposing strict synchronization, with associated bottlenecks. *Asynchronous SGD* (*AsyncSGD*), which relaxes the original semantics, has gained significant interest in recent years due to promising results that show speedup in certain contexts. However, the relaxed semantics that asynchrony entails give rise to fundamental questions regarding *AsyncSGD*, relating particularly to its *stability* and *convergence rate* in practical applications.

This thesis explores vital knowledge gaps of *AsyncSGD*, and contributes in particular to: **Theoretical frameworks** – Formalization of several key notions related to the impact of asynchrony on the convergence, guiding future development of *AsyncSGD* implementations; **Analytical results** – Asymptotic convergence bounds under realistic assumptions. Moreover, several technical solutions are proposed, targeting in particular: **Stability** – Reducing the number of non-converging executions and the associated wasted energy; **Speedup** – Improving convergence time and reliability with instance-based adaptiveness; **Elasticity** – Resource-efficiency by avoiding over-parallelism, and thereby improving stability and saving computing resources. The proposed methods are evaluated on several standard DL benchmarking applications and compared to relevant baselines from previous literature. Key results include: (i) persistent speedup compared to baselines, (ii) increased stability and reduced risk for non-converging executions, (iii) reduction in the overall memory footprint (up to 17%), as well as the consumed computing resources (up to 67%).

In addition, along with this thesis, an open-source implementation is published, that connects high-level ML operations with asynchronous implementations with fine-grained memory operations, leveraging future research for efficient adaptation of *AsyncSGD* for practical applications.

Keywords

Acknowledgment

The most sincere gratitude is owed to my supervisors Marina Papatriantafilou and Philippas Tsigas for their support and guidance, without which this work would not have been possible.

For countless spontaneous fruitful discussions, and the best of company during much needed coffee breaks, I wish to thank my colleagues Amir, Aras, Bastian, Beshr, Babis, Carlo, Christos, Dimitri, Elad, Fazeleh, Georgia, Hannah, Iulia, Ivan, Jens, Joris, Kåre, Lucas, Magnus, Martin, Masoud, Nasser, Olaf, Oliver, Parthasarathy, Romaric, Sólrún, Thomas, Valentin, Vincenzo, and Wissam. A special thanks is directed to my friends Ebrahim and Mahmood who have provided continuous support throughout my work, and to whom I owe many inspiring discussions and invaluable advice.

Warmest thanks are directed to my family for their endless loving support. In particular to my grandfather, Karl-Johan "Boa" Bäckström, and to my loving mother Sofia – in addition to discussing ideas and tireless proof-reading, I have both of you to thank for my accomplishments in both education and career, and this thesis in particular. Also, to my brother Felix, my grandmother Christina, and my father Mats, for always bringing joy, even when times are tough. And, to my dearest Rita for your encouragement and remarkable patience.

List of Publications

Appended publications

The results of the thesis are also reported and disseminated in the following articles:

[A] **K. Bäckström**, M. Papatriantafilou, and P. Tsigas "MindTheStep-AsyncPSGD: Adaptive Asynchronous Parallel Stochastic Gradient Descent"
Proceedings of the 7th IEEE International Conference on Big Data, 2019, pp. 16-25.

[B] **K. Bäckström**, I. Walulya, M. Papatriantafilou, and P. Tsigas "Consistent Lock-free Parallel Stochastic Gradient Descent for Fast and Stable Convergence."
Proceedings of the 35th IEEE International Parallel & Distributed Processing Symposium, 2021, pp. 423-432.

[C] **K. Bäckström**, M. Papatriantafilou, and P. Tsigas "`ASAP.SGD`: Instance-based Adaptiveness to Staleness in Asynchronous SGD"
Proceedings of the 39th International Conference on Machine Learning, 2022, pp. 1261-1276.

[D] **K. Bäckström**, M. Papatriantafilou, and P. Tsigas "*Less is more:* Elastic Parallelism Control for Asynchronous SGD"
Submitted work under review.

Other publications

During time overlapping my my doctoral studies i have also contributed to the following articles. However, they are not appended to this thesis, due to contents overlapping that of appended publications or contents not related to the thesis.

[a] **K. Bäckström**, M. Papatriantafilou, P. Tsigas "The Impact of Synchronization in Parallel Stochastic Gradient Descent"
Proceedings of the 19th International Conference on Distributed Computing and Intelligent Technology, 2022

[b] E. Balouji, **K. Bäckström**, P. Hovila "A Deep Learning Approach to Earth Fault Classification and Source Localization"
Proceedings of the 10th IEEE PES Innovative Smart Grid Technologies Europe, 2020

[c] D. Parthasarathy, **K. Bäckström**, J. Henriksson, S. Einarsdóttir "Controlled time series generation for automotive software-in-the-loop testing using GANs"
Proceedings of the 3rd IEEE International Conference On Artificial Intelligence Testing, 2020

[d] E. Balouji, **K. Bäckström**, T. McKelvey, Ö. Salor "Deep Learning Based Harmonics and Interharmonics Pre-Detection Designed for Compensating Significantly Time-varying EAF Currents"
Proceedings of the 54th IEEE IAS Annual Meeting, 2019

[e] V. Gustafsson, H. Nilsson, **K. Bäckström**, M. Papatriantafilou, V. Gulisano "Mimir - Streaming Operators Classification with Artificial Neural Networks"
Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems, 2019

Personal Contribution

I am the main author of the appended publications, lead designer of algorithms and implementations, and responsible for analytical and empirical results, in collaboration with all other authors. In **Chapter B**, the implementation was done in collaboration with Ivan Walulya, who made important contributions, and I performed the majority of the work.

Contents

Chapter 1

Thesis Overview

1.1 Introduction

"I propose to consider the question, 'Can machines think?' " are the opening words of Alan M. Turing in the article *Computing Machinery and Intelligence*, published 1950 in Mind [1]. Realizing that such an, at the time, controversial question would undoubtedly be overwhelmed with criticism, if not utterly dismissed, especially without hope to ever provide a reasonable definition of "think" (or even "machine" for that matter) Turing swiftly circumvents this issue simply by defining a game:

The Imitation Game. Also known as the *Turing Test*, the Imitation Game determines whether a machine possesses human intelligence. In the test, a human evaluator queries, by text messages, two participants, one of which is a machine. If the evaluator cannot distinguish the human participant from the machine, it passes the test. In the article, Turing optimistically argues against common objections to the statement "machines can think" and predicts "... at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted".

Artificial Intelligence today. One cannot easily argue against Turing's prophecy, to a large extent, having been fulfilled at the moment of writing this thesis. A classic example of early advancement of Artificial Intelligence (AI) is the ability of computers to play chess. Already in the 1980s, state of art chess computers beat prominent human players, and in 1997 the IBM machine *Deep Blue* beat the, at the time, reigning human world champion Garry Kasparov in a six-game chess match [2]. Among many factors attributed to the success of *Deep Blue*, the ability to perform extensive tree searches, thanks to a *massively parallel* implementation, is considered one of the most important. More recently, the success of the AI model *AlphaGo* from *DeepMind* in beating the human champion at the substantially more complex game of *Go* in 2016 [3] marks a significant milestone for the evolution of AI. The trend in increasingly capable AI models, performing exceptionally well at a variety of tasks, keeps progressing; Only during the last year, at the time of writing, tremendous leaps within so-called *AI language models* (sometime loosely referred to as *chat bots*) have been achieved. Among the most successful are the *LaMDA* model [4] from *Google* and *ChatGPT* from *OpenAI*, which both show astounding capabilities of (i) conversing in human language, (ii) retrieving and explaining information, (iii) language translation, as well as (iv) computer code generation and debugging. In some circles, it has even been disputed whether or not these models have passed the *Turing Test*. The consequences of the above achievements cannot be underestimated, considering that 97% of mobile phone users are using AI-powered voice assistants. In general, AI shows its strengths in almost every industry nowadays, e.g., for autonomous vehicles, personalized medical care, and intelligent power management in smart grids [5], and the usage of AI exhibits super-linear growth across academic disciplines [6].

Digitalization and Big Data. The success of AI technology as means for automatic intelligent data processing has never been more needed, considering the vast digitalization and sensor deployment in society, which results in huge amounts of information to be analyzed. This has sparked the interest in the field of *Big Data*, defined as the collection of methods for storing, processing,

and analyzing extremely large data sets to reveal patterns, trends, and associations, especially relating to human behavior and interactions. Big Data is often associated with its *three V's*, by which it is typically characterized, namely its (i) Volume, (ii) Variety, and (iii) Velocity. As digitalization continues to progress rapidly, so do also the challenges associated with the three V's, including the need for increasingly complex AI models that can analyze it, and consequently also for efficient infrastructure and implementations of such models to scale.

Indeed, data is the new oil.   It is increasingly common to come across the comparison between the significance of data in modern society and the impact that oil has had during the last century. The two indeed resemble one another in many ways, some of the most prominent being (i) how value is extracted through careful refinement processes, (ii) utilization for higher efficiency in industrial processes, (iii) increase in quality of life through products for private use. Although the inadvertent environmental impact of oil is well-known nowadays, the strong resemblance of also this aspect to Big Data is scarcer in literature. Analogous to oil, the processing and consumption of also data has an increasingly substantial impact on the global environment; Increasing consumption of computational resources for data processing necessitate analogous energy consumption, sometimes disproportionately high to common belief [7,8] For instance, it has been estimated that training a single AI language model can emit close to 300,000kg of carbon dioxide equivalent, which is nearly five times the lifetime emissions of an average car, including its manufacture [9]. Consequently, along with rapidly growing complexity of AI models, super-linear growth in the environmental footprint of them exhibits similar trends [6]. This is significant, in particular when considering that since 2012, the amount of computing power used by state-of-art AI models has grown exponentially with a 3.4-month doubling time[1] [10], and expectations of associated increasing environmental impact cannot reasonably be excluded.

New challenges.   As opposed to oil, which is a finite resource, the global estimated volume of digital data exhibits exponential growth at an astounding rate (Figure 1.1), and this trend has no end in sight [11]. As a consequence, there is an associated critical demand for scalable computing infrastructure for AI to analyze it. In addition to scalability, such AI infrastructure must cope with several substantial technical difficulties, stemming from the fact that data is typically stored in isolated depots, and associated with strict data privacy concerns. In fact, where data privacy is most critical is often where advances in automated data analysis with AI is needed the most, e.g., competition-critical business operational data, power consumption patterns in smart grids, and sensitive medical data of private individuals. The associated technical solutions face additional heavy resistance from growing administrative measures for data privacy, including the General Data Protection Regulation (GDPR) of the European Union, as every privacy-concerned EU-citizen knows the tedious process of unticking a long list of data-sharing options. However, privacy concerns of data owners must be taken seriously, and any viable technical solution must adhere to them. The above challenges associated to scalability and security constitute major driving factors for finding methods that better utilize parallel and distributed computing infrastructure for AI, since it can

---

[1]In comparison, Moore's law exhibits a mere 2-year doubling period
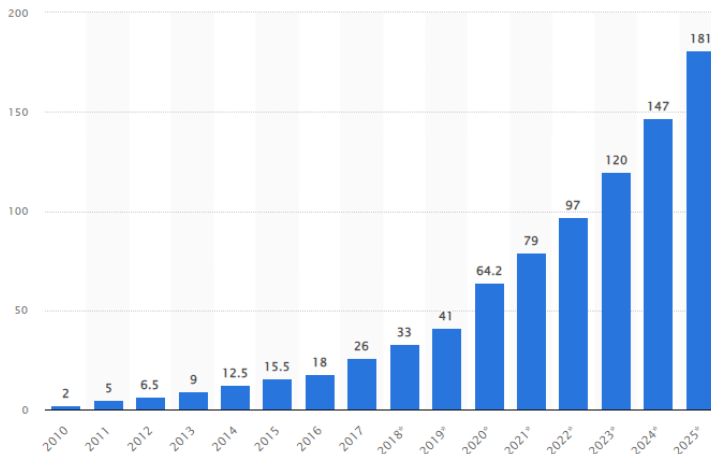
Figure 1.1: Global volume of created digital data (ZB) during the last decade, and forecasts until 2025 [11].

contribute to solving both. Such methods, showing promising potential for data privacy in particular, have started to appear within the recently popular *Federated Learning* (FL) paradigm, in which the training of AI models, i.e., the *Machine Learning* (ML), is done in parallel across distributed data depots where the data is located. The locally trained AI models, which encode the knowledge gained from the data associated to individual depots, are shared and aggregated with the effort to construct a global AI model. FL indeed eliminates the need for data sharing, however the paradigm gives rise to critical research questions regarding the implementations of the underlying ML algorithms that actually train the AI models, in particular their scalability, since they are not always trivially parallelizable.

ML – under the hood. The numerical iterative optimization algorithm Stochastic Gradient Descent (SGD), the credit for which is usually attributed Augustin-Louis Cauchy who first proposed it in 1847, is the backbone of the many successful modern ML algorithms. Through efficient processing of data in random batches, SGD enables training a variety of AI models, including Logistic Regression, Support Vector Machines, and the recently popular Artificial Neural Networks (ANN). However, SGD dates back to long before computers were used in practice, not to mention ones capable of parallel processing. SGD is in fact, like many other iterative algorithms, highly sequential in its nature. This is due to that the computation associated with each iteration is dependent on the outcome of the previous one. However, everything considered, i.e., rapidly growing (i) computational demands (the three V's), (ii) complexity of AI models, as well as (iii) data privacy concern and distribution of data, the community must work towards finding solutions to parallel and federated learning in order to fully utilize contemporary hardware, and increased connectivity and distributed computing resources.

> *"When we start talking about parallelism and ease of use of truly parallel computers, we're talking about a problem that's as hard as any that computer science has faced. … I would be panicked if I were in industry."*
>
> John L. Hennessy — Stanford President (2006)

Trends in systems and networks. The above quote, by the previous President of Stanford, John L. Hennessy, is from the beginning of this century, when the end of *Moore's law* was growing more apparent. The amount of processing power that a single integrated circuit could bring was approaching a limit, and improvements in performance were increasingly difficult and expensive to achieve. As a result, the microprocessor manufacturing industry shifted from the pattern of increasing the performance of conventional sequential processors, to focus instead on equipping processing chips with a higher number of cores [12]. In the meanwhile, the growth of the IoT paradigm, with vast networks of connected physical devices equipped with data-collecting sensors and processing ability, started accelerating at unprecedented rates [13]. The new "law" that emerged was consequently one of exponentially growing number of parallel computational workers, either as (i) processors in computing chips, or (ii) nodes in distributed computing networks. Ever since, it has been the privilege of the software development community to solve the numerous problems arising in the field of parallel and distributed programming, in the strive to fully utilize the new generation computing infrastructure. The interest in the field has increased rapidly due to the significant performance gains that it can entail, and while the hardware community continuously challenges with new computing infrastructure, so does also the software community challenge by constructing the parallel algorithms that master it.

Parallel SGD today. The sequential nature of SGD, and other iterative methods like it, leaves two options for parallelization, namely (i) that parallelism is allowed only during each individual iteration and synchronizing at the end in a lock-step manner, (ii) or relaxing the semantics of the original algorithm. These options are both viable approaches to parallelization of SGD and are known as *synchronous* and *asynchronous* SGD (*SyncSGD* and *AsyncSGD*), respectively. Among the two, *SyncSGD* is significantly more widely adopted — in fact, the principle of averaging workers' partial result after each iteration is used on many levels of applications, ranging from standard GPU-accelerated ML libraries on private desktop computers, to the standard *federated averaging* FL approach on high-end distributed cloud infrastructure. In addition to its simplicity in implementation, an important reason for the wide adoption of *SyncSGD* is that it is understandable; As pointed out already in [14], "*If all processors communicate to each other their partial results at each instance of time and perform computations synchronously, the distributed* [or parallel] *algorithm is mathematically equivalent to a single processor (serial) algorithm and its convergence may be studied by conventional means*". However, it is easy to realize that the synchronous parallelization approach suffers limitations in scalability. This is due to the fact that each iteration is only as fast as the slowest contributing worker. Hence, slow workers, i.e., *stragglers*, present particularly in heterogeneous computing environments, can significantly impact the convergence time. On the other hand, asynchronous approaches alleviate

this limitation, and show improved scalability in many practical applications. However, the reduced inter-worker coordination that asynchrony entails breaks the semantics of the original SGD algorithm and leads to several critical questions; Among the most important are (i) how the convergence rate is affected, and (ii) what impact asynchronous parallelization of SGD has on the consumption of computing resources. Moreover, the degree of synchronization that is still required, such as when accessing shared variables in shared-memory contexts, 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. Similarly to how the *massively parallel implementation* of the *Deep blue* chess computer was integral for it constituting a milestone in early AI development, also today, as introduced above, parallelism is crucial for modern ML deployments.

Despite the many challenges that are observable today, the parallel computing paradigm will surely continue to be crucial for the continued progress of AI in the years to come, just like new challenges will surely arise – the quote by A. Turing in the initial pages of this thesis remains, and will remain, true: *"We can only see a short distance ahead, but we can see plenty there that needs to be done." —Alan Turing, 1950*

Thesis objective. Results on asynchronous parallelism in iterative optimization for ML were not widely reported until the beginning of the last decade, when the interest in the topic started growing significantly along with the increased demand for scalable AI. Since then, *AsyncSGD* has shown promising potential to achieve scalable speedup in certain contexts, however several fundamental questions remain regarding the usefulness of the method in practice. This thesis identifies and explores relevant knowledge gaps of *AsyncSGD* in previous literature and proposes technical solutions that enable efficient utilization in relevant application areas. To achieve the above, the thesis focuses particularly on (i) formalization of key concepts associated to *AsyncSGD*, (ii) extending analytical convergence results under realistic theoretical assumptions, (iii) develop algorithmic adaptations that accommodate for the unique challenges associated to asynchrony, (iv) realistic empirical evaluation on benchmarks relevant for practical applications.

Thesis structure. The remainder of the **Overview** chapter covers preliminary theoretical foundation and metrics of interest related to SGD and its application in ML (Section 1.2), background of approaches to parallel SGD and current state-of-art (Section 1.3), a summary of relevant research questions (Section 1.4), and the main contributions of this thesis to address them (Section 1.5). The chapters that follow consist of the published articles associated with this dissertation, where adaptations in notation have been made for the purpose of unification across the chapters; A full list of the adopted nomenclature is available in Table 1.1.

Table 1.1: List of symbols used throughout this thesis.

	Symbol	Meaning
	L	Non-negative loss function
	ϵ	Precision threshold for convergence
	θ_i	AI model, i.e., vector of trainable parameters, at iteration i
	d	Equals $|\theta|$, i.e., optimization problem dimension
Optimization	D	Dataset used for training
	B_i	Mini-batch of data sampled randomly from D in iteration i
	b	Mini-batch size $b = |B|$
	η_i	Step size in iteration i
	η	Step size, constant throughout execution
	$\nabla\widetilde{L}(\theta), \nabla L_B(\theta)$	Stochastic gradient of L at θ, computed over a random mini-batch
	μ	Momentum parameter
	i	Iteration number
	t	Wall-clock time
	$\hat{m}$	System maximum parallelism level
Parallelism	Δ	Update to the shared state by a worker
	τ	Staleness, defined as the number of applied concurrent updates
	$\hat{\tau}$	Assumed maximum system staleness $\hat{\tau} = \max_i \tau_i$
	v_i	The $view$ of a worker when applying an update, $v_i = \theta_{i-\tau_i}$
	m_C^*	Computational saturation point
	m_S^*	System saturation point
	$y(x : \theta)$	Output computed based on input x and parameters θ
ANN	$\hat{y}(x)$	Label for x
	$\mathbf{o}^{(l)}$	Output vector of layer l
	σ	Non-linear activation function

1.2   SGD for machine learning

In this section, SGD is introduced, in particular in the context of ML applications. Examples of such applications relevant for this thesis are introduced as well, including different types of ANN layers. Moreover, metrics of interest are defined, including a discussion on a useful decomposition of them, particularly useful as KPIs when evaluating new implementations of parallel ML methods.

1.2.1   Stochastic gradient descent

SGD in ML.   SGD is a first-order iterative numerical optimization algorithm, which follows:

$$\theta_{i+1} = \theta_i - \eta\nabla\tilde{L}(\theta_i) \tag{1.1}$$

given an optimization problem

$$\underset{\theta\in\mathbb{R}^d}{\text{minimize}} \quad L_D(\theta) \tag{1.2}$$

where (i) D is a data set to be processed, (ii) $\theta \in \mathbb{R}^d$ is an AI model that encodes the learned knowledge from D for a specific task and (iii) $\tilde{L}$ are random
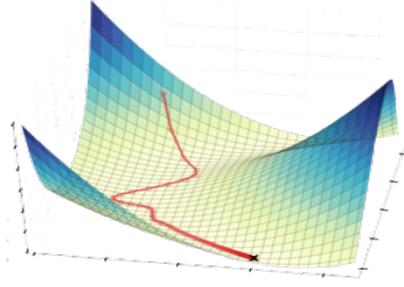
Figure 1.2: Stochastic Gradient Descent is a first-order iterative numerical optimization algorithm which repeatedly steps in the direction of steepest descent, following the slope of the target function.

observations of the target function $L\colon \mathbb{R}^d \to \mathbb{R}^+$ which quantifies the loss of θ on D, and (iv) η is the step size – a hyper-parameter that typically needs to be tuned to individual applications for SGD to be effective. The initialization point θ_0 is chosen at random according to some distribution, which as one might expect may significantly impact the convergence [15]. A random observation $\tilde{L}$ of L in iteration i is acquired by evaluating θ_i on a subset, or mini-batch, $B_i \subset D$, sampled uniformly at random, i.e., $\tilde{L}(\theta_i) = L_B(\theta_i)$.

The SGD iteration (1.1) repeatedly adjusts the model θ along the negative gradient of L, the direction of which constitutes the one of *steepest descent*. The convergence trajectory hence corresponds to the slope of the target function (see Figure 1.2). The iteration (1.1) is repeated until a solution θ^* of sufficient quality is found, typically $L_D(\theta^*) < \epsilon$, referred to as ϵ-convergence.

Benefits of SGD.   In the context of data processing, the original deterministic counterpart Gradient Descent (GD) to SGD sets $B = D$, i.e., considers the entire data set in every iteration. The stochastic element of SGD due to random data sub-sampling entails two major benefits: (i) sampling and processing only small mini-batches enables significantly faster iterations and (ii) the algorithm is effective on non-convex target functions, as opposed to GD. However, SGD introduces a new hyper-parameter, namely the batch size $b = |B_i|$, which relates to the level of sampling variance, i.e., the level of *stochasticity* or *noise* in the convergence trajectory.

SGD with momentum.   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. A widely established method for reducing such oscillation, while maintaining the stochasticity as necessary, is *Momentum-SGD* (MSGD), defined as follows:

$$\theta_{i+1} \leftarrow \theta_i + \mu(\theta_i - \theta_{i-1}) - \eta\nabla L_B(\theta_i) \qquad (1.3)$$

for some momentum parameter $\mu \in [0, 1]$. Momentum has become known to significantly accelerate the convergence of SGD in many practical settings. Especially for target functions which are irregular and asymmetric in its shape,
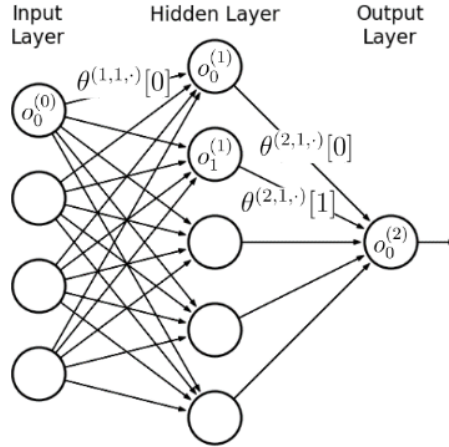
Figure 1.3: The input $\mathbf{o}^{(0)}$ to a multi-layer ANN undergoes several transformations which are parameterized by the components of θ, namely $\theta^{(l,n,w)}[j]$ denoting the jth component of the transformation weight associated with the nth neuron of the lth layer.

forming narrow valleys. Such irregularities are in particular known to arise in *deep learning* applications.

1.2.2 Artificial neural networks

Deep Learning   (DL), i.e., deep Artificial Neural Network (ANN) training, the training of which is enabled by SGD, is a major component of many recently successful AI models, including *DeepMind*'s *AlphaGo* [3], *Google*'s *LaMDA* [4], *ChatGPT* from *OpenAI*, among others. At the core, ANNs are computational structures inspired by the biological brain and consist of many fundamental units referred to as neurons. They consist of several layers of non-linear transformations that process the input data in consecutive steps (see Figure 1.3). Each layer is parameterized by a *weight* matrix and a *bias* vector, both constituting part of the parameter vector θ, which is learned through the optimization process (1.1). The input data, e.g., an image, to be analyzed by the ANN is provided as initialization to the first layer. After processing throughout the layers, this results in some output in the last layer, corresponding to e.g., the predicted class of the input image. Different topological properties, such as connectivity among neurons, give rise to a diverse set of neural architectures. Among the most commonly used are *Multi-Layer Perceptrons* (MLPs) and *Convolutional Neural Networks* (CNNs):

- **MLPs** consist of densely-connected layers, each applying a non-linear transformation of its input and passing the result on to the next layer:

$$o_n^{(l)} = \sigma \left(\sum_{j=0}^{|N_{l-1}|-1} \theta^{(l,n,w)}[j] \cdot o_j^{(l-1)} + \theta^{(l,n,b)} \right)$$

where $o_n^{(l)}$ is the output of neuron $n \in \{0, \ldots, N_l - 1\}$ in the l-th layer, σ is a non-linear activation function, typically the ReLU function $\sigma(x) = max(0, x)$, and $\theta^{(l,n,w)}, \theta^{(l,n,b)}$ contains the learnable weights and bias parameters of to the n-th neuron.

- **CNNs** consist of layers that convolve the input with learnable filters for feature detection:

$$o_{n,f}^{(l)} = \sigma \left(\sum_{j=0}^{k} \theta^{(l,f,w)}[j] \cdot o_{n+j}^{(l-1)} + \theta^{(l,f,b)} \right)$$

for a number of filters f, corresponding to a 1D convolution. This can be naturally extended to 2D, with filter matrices being convolved with the input in both axes. Convolutional layers are sparsely connected and reduce the number of weights to be learned. They are especially efficient for analysis of image (or other spatially dependent) data due to the translation-invariant property of feature detection with convolution.

Convolutional layers are often used in combination with *MaxPool* layers, which pick the maximum output of a number of consecutive neurons as the output of the layer. This is meant to leverage detection of relevant features, as well as significantly reduce its dimension. It will hence also decrease the total number of learnable weights. In the last, output, layer of an ANN, the *softmax* activation function $\sigma_j(x) = e^{x_j} / \sum_{k=1}^{|x|} e^{x_k}$ (e being Euler's number) for each output neuron j, is often used for classification problems. Its output satisfies the requirements of a probability distribution function, and is consequently interpreted as such in this context, i.e., the estimated class distribution y of an input x.

The training. Given the true class label $\hat{y}$ of the input x, the performance of an ANN for classification is quantified by some error function, e.g., the cross-entropy loss function:

$$L(\hat{y}, y(x : \theta)) = - \sum_{j}^{|out|} y(x : \theta)_j \log(\hat{y}_j)$$

where y is the output of the last layer, and naturally depends on the input x and the current state of θ. The *training* process of an ANN now constitutes of iteratively adjusting θ to minimize the error function $L_D(\theta) = \frac{1}{|D|} \sum_{x \in D} L(\hat{y}, y(x : \theta))$. The *BackProp* algorithm is used for computing $\nabla_\theta L(\theta)$, and SGD is then used for minimizing f, and training the ANN. In every iteration the input is selected at random, either as single data point or as a *mini-batch* over which the error is averaged.

1.2.3 Metrics of interest

Metric decomposition. The implementation of any algorithm affects its performance and usefulness in practice. When it comes to SGD, or any other iterative optimization algorithm for that matter, the performance is influenced by many aspects of the implementation as well as the system on which it is executed. As described in [16] (and **Chapter B** in this thesis)

a useful decomposition of the performance is to consider the *statistical* and *computational* efficiency, defined as follows:

(i) *Statistical efficiency* measures the number of SGD iterations required until reaching ϵ-convergence.

(ii) *Computational efficiency* measures the number of iterations per time unit.

Convergence.   The overall *convergence time*, i.e., the wall-clock time until ϵ-convergence (when $L(\theta^*) < \epsilon$), is the most relevant in practice. The *convergence rate*, in this context referring to the instantaneous rate of change of the loss function L, i.e., $\frac{\partial L}{\partial t}$ is closely related, where t denotes wall-clock time. As also pointed out in [16], the convergence rate is essentially the product:

convergence rate = statistical efficiency × computational efficiency

A similar conclusion is reached in **Chapter D** by more formal means, however reaching a more precise description, by using the chain rule as follows:

$$\frac{\partial L}{\partial t} = \frac{\partial L}{\partial i}\frac{\partial i}{\partial t} \tag{1.4}$$

where the right-hand side factors correspond exactly to the statistical and computational efficiency, respectively. When proposing new algorithms (or altering existing ones) in this application domain that potentially change the *computational efficiency*, it is advisable to not only evaluate the *statistical efficiency*, by 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.

The aforementioned metrics become particularly important in the context of parallel algorithms for iterative optimization, since depending on the method for parallelization, such algorithms often have significant impact on computational and statistical efficiency, as shall be seen in the following.

1.3   Parallel SGD

Motivation.   With rapidly growing demands for data analysis, there is an increasing interest in achieving the necessary scalability by utilizing parallel algorithms for SGD, capable of utilizing modern many-core processing infrastructure as well as large clusters of distributed computing networks, e.g., *Federated Learning*. While parallelism can improve computational efficiency, simply by applying a greater number of updates in each unit of time, the impact on the statistical efficiency, and thereby the overall convergence rate, is unpredictable. This follows from that the original SGD algorithm is inherently sequential, requiring the computation of each iteration to be completed in order to perform the computation of the next. Parallelization of SGD is consequently non-trivial, and requires synchronization in every iteration (prior to applying an update) in order to maintain the original sequential semantics of SGD. Alternatively, workers can execute the SGD algorithm, i.e., accessing and

updating the shared state θ, asynchronously, although this approach breaks the sequential semantics. These approaches correspond to two main directions of methods for parallel SGD, referred to as *synchronous* and *asynchronous*. In this context, methods with varying degree of asynchrony have been proposed within the spectrum that emerges, several of which can be considered particularly representative ones, due to their requirements on synchronization, staleness, and other consistency and progress guarantees.

Multi-dimensionality of parallelism. In addition to that methods for parallel SGD are placed along the synchrony-asynchrony spectrum, some address specifically *shared-memory* contexts, where workers are parallel processes in the same node, while others target implementations for *distributed networks* of collaborative workers nodes, such as in FL. Moreover, *centralized* vs. *decentralized* parallel SGD is another dimension of interest, however orthogonal to the previous two to a large extent. Although decentralization can be of some relevance in shared-memory contexts, in particular for improving consumption of computing resources and memory, it is mostly researched in the context of distributed computing, targeting e.g., reduced communication overhead and need for trusted third parties, particularly relevant in FL settings [17].

Establishing unified terminology. There are several representative methods for parallel SGD, which employ fundamentally different mechanisms for synchronization. There is however a *terminological inconsistency* in the literature of this domain, with opposing notions of *consistency* and *progress* guarantees. For instance, the properties *wait-freedom* and *inconsistency* have been attributed to parallel algorithms for SGD due to the presence of *asynchrony* in the update aggregation [18, 19], while in other contexts *lock-freedom* refers, somewhat more traditionally, to progress guarantees for concurrent operations on the shared state [20]. The inconsistency in notation stems from the fact that parallelization of an iterative algorithm, such as SGD, has at least two different types, or *dimensions*, of synchronization. The first relates to how updates (denoted by Δ) are aggregated (if at all) and is referred to as the *coarse-grained* synchronization dimension. The second relates to the *fine-grained* synchronization for operations on the shared state, such as reading or applying an update. Although it is theoretically imaginable that the *fine-grained* dimension be of interest in distributed settings, for instance in *distributed shared memory* settings, it has not been explored in practice. Instead, the above terminology relating to the *coarse-grained* dimension is exclusively used in distributed settings, while the *fine-grained* is used in shared-memory ones. Hence, different works typically explore one of these dimensions, or the other, and since both are explored within the scope of this thesis, a consistent notation is adopted in following, which distinguishes the two. In the coarse-grained (Δ) dimension, we have:

> Δ-*Progress* describes synchronization mechanisms regulating how updates are aggregated. Examples of such progress properties include computing a global iteration by averaging the update contributions from a certain number of workers, once-in-a-while synchronization, etc. An algorithm which does not employ such aggregation is referred to as *asynchronous* (Figure. 1.5), as opposed to *synchronous* (Figure. 1.4) ones which e.g., aggregate updates by averaging them in a lock-step manner.

Table 1.2: Adopted terminology regarding properties of concurrent operations.

	Meaning
Lock-freedom	System-wide throughput, allows starvation
Wait-freedom	System-wide throughput with starvation-freedom
Consistent	Read operations return a consistent snapshot

Table 1.3: Consistency and progress guarantees for different methods for parallel SGD.

		Synchronous		*Hybrid*	*Asynchronous*		
Δ	**Asynchronous**	$\times$	-	-	$\checkmark$	$\checkmark$	$\checkmark$
	Consistent	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\times$	$\checkmark$
θ	**Lock-free**	$\times$	$\times$	$\times$	$\times$	$\checkmark$	$\checkmark$
		SyncSGD [22]	Stale-synchronous [23]	n-softsync [24]	*AsyncSGD* [25]	HOGWILD! [20]	*Leashed-SGD* [**Chapter B**]

Δ-*Consistency* refers to conformity to a sequential execution. An algorithm that allows any degree of asynchrony is typically not consistent with a sequential execution, hence not Δ-consistent.

There is a strong dependency between Δ-*Progress* and Δ-*Consistency*, since stronger progress requires higher degree of asynchrony and staleness, which entails higher deviation from a sequential execution. The second dimension relates to shared-memory parallelism contexts, and is relevant for characterizing fine-grained synchronization for operations on the shared state θ. Here, the following notation is introduced, with the aim to conform also to standard notation (summarized in Table 1.2) in established literature on concurrent implementations of shared data objects [21]:

θ-*Progress* refers to progress guarantees with respect to operations on the shared state θ, in particular *read* and *update*. This includes in particular *lock-freedom*.

θ-*Consistency* refers to the *consistency model* for operations on the shared state θ, including in particular *consistent* read operations, as defined in Table 1.2.

For the remainder of this thesis, in order to distinguish between different concepts and conform to standard notation to the greatest extent possible, the terms *progress* and *consistency* shall be used in the latter sense, i.e., with respect to operations on the shared state θ. The coarse-grained dimension of synchronization of the updates (Δ) is primarily referred to as *asynchrony*. In table 1.3, an overview is provided of some of the representative methods in the literature, on the synchrony-asynchrony spectrum, which are relevant within the scope of this thesis, and more detailed descriptions of these are provided in the following (Section 1.3.1 and 1.3.2).
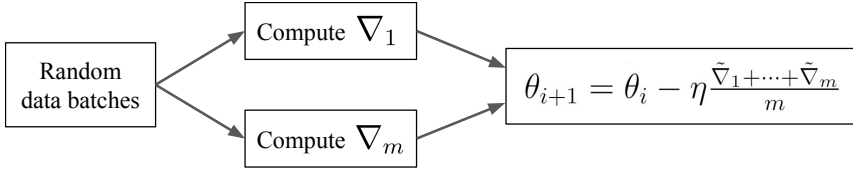
Figure 1.4: In *SyncSGD* the workers' individual updates are aggregated by averaging, after which a global iteration is performed. *SyncSGD* essentially corresponds to parallelization on the gradient computation level.

1.3.1   Synchronous parallel SGD

***SyncSGD* rundown.**   *Synchronous parallel SGD (SyncSGD)* is a straightforward lock-step-style parallel implementation of SGD, in which workers access the shared state θ, then compute updates based on individual randomly sampled data-batches and synchronize by averaging the resulting gradients before taking a global step according to (1.1) [22], see Fig. (1.4). Today, *SyncSGD* is widely adopted – in fact, the principle of averaging workers' partial result after each iteration is the core of parallelization on many levels of applications, ranging from standard GPU-accelerated ML libraries on private desktop computers, to the standard *federated averaging* FL approach [26] on high-end distributed cloud infrastructure. In the original version, *SyncSGD* is statistically equivalent to sequential SGD with larger mini-batch size, as observed empirically in [27], and shown by more analytical means in **Chapter A**, and can be considered a method for accelerated gradient computation. Hence, the original *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*. As pointed out already in [14], "*If all processors communicate to each other their partial results at each instance of time and perform computations synchronously, the distributed* [or parallel] *algorithm is mathematically equivalent to a single processor (serial) algorithm and its convergence may be studied by conventional means*". A comprehensive overview of methods along this approach is provided in [28]. However, *SyncSGD* has inherent limitations in scalability, since the presence of slower workers, i.e., *stragglers*, become bottlenecks due to the fact that each SGD iteration is only as fast as the slowest contributing worker. Some hybrid approaches, which allow a certain degree of asynchrony, aim to alleviate such bottlenecks, and some representative works are highlighted in the following.

Hybrid approaches.   *Stale-synchronous parallel* (SSP) SGD relaxes the strict synchronous semantics of *SyncSGD*, allowing faster workers to asynchronously compute a bounded number of SGD steps based on a local version of the state before synchronizing [23]. This method is particularly useful in heterogeneous computing systems, where stragglers are kept in check. SSP has been proven useful for distributed DL applications, e.g., in [29] where a method for dynamically adjusting the staleness 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 workers, the system as a whole will halt indefinitely in

the synchronization step. This is partially addressed by the *n-softsync* protocol, which is a further relaxed variant of *SyncSGD* with partial synchronization, requiring only a fixed number n of workers to contribute a gradient at the synchronization point. As opposed to SSP, there is no bound on the maximum staleness. Introduced originally in the context of centralized distributed SGD with a parameter server [24] [27], 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.

Limitations.   The computational scalability limitation of *SyncSGD* due to stragglers, in particular in heterogenous computing environments, can be alleviated to some extent through the aforementioned hybrid approaches. However, the observation made in **Chapter A**, as well as in [27], namely that increased synchronous parallelism in SGD is statistically equivalent to sequential SGD with larger batch sizes, constitutes yet another scalability limitation. This stems from the fact that the batch size, as highlighted in Section 1.2.1, corresponds to a critical element of SGD, namely its stochastic *variance*, which in appropriate magnitudes enable effectiveness on non-convex applications. However, when not tuned properly to the problem at hand, it can severely impact the quality of the models that SGD executions outputs, in particular in terms of generalizability [30], as well as the quality of the convergence trajectory in general [31] [32]. Hence, it is necessary to explore additional tiers of parallelism, that can enable extended scalability – and many promising solutions can be found among *asynchronous* methods.

1.3.2   Asynchronous parallel SGD
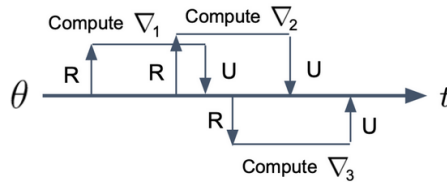
Figure 1.5: *AsyncSGD* parallelizes the SGD iterations, allowing asynchronous read (R) and update (U) operations on the shared state.

Highlights.   *Asynchronous parallel SGD* (*AsyncSGD*) removes the gradient averaging synchronization step, allowing workers to access and update the shared state asynchronously. Consequently, while an update is being computed by one worker, there can be concurrent updates applied by others. Hence, *AsyncSGD* follows:

$$\theta_{i+1} \leftarrow \theta_i - \eta \widetilde{\nabla} f(v_i) \tag{1.5}$$

where $v_i = \theta_{i-\tau_i}$ is a worker's *view* of θ and τ_i is the number of concurrent updates, which defines the *staleness*. Updates are consequently generally computed based on states which are older than the ones to which the updates are applied. The resulting overall impact on the convergence trajectory is

referred to as *asynchrony-induced noise* (AIN), and affects, together with the overall distribution of the staleness observations τ_i, the statistical efficiency. The notion of AIN is mentioned is several related previous works, however a formal definition has not been proposed.

Orthogonality. Note that, while *SyncSGD* adheres to the original sequential semantics of SGD and is consequently equivalent to accelerating the gradient computation, the same does not hold for *AsyncSGD*. Instead, *AsyncSGD* provides an additional layer of parallelism, which relaxes the sequential SGD semantics, with inherent consequences to computational and statistical efficiency. Since the two paradigms impact distinct levels of parallelism, they can be used in conjunction, in particular by allowing asynchronously parallel workers to individually engage in synchronous parallelization of the gradient computation.

Computational vs. statistical efficiency. *AsyncSGD* almost surely enables increased computational efficiency with higher parallelism, as discussed in Section 1.2.3, however, only up to a point where contention due to concurrent shared-memory access attempts becomes too severe, in accordance with Amdahl's law. We denote the corresponding number of workers by m_C^*, referred to as the *computational saturation point*, and at this point the system stagnates, and additional computing workers provide no additional speedup. In addition, the presence of staleness and AIN in *AsyncSGD*, and the resulting deviation from the original sequential semantics of SGD, results in decay in statistical efficiency, which grows as more workers are introduced to the system (see Section 1.2.3). Over-parallelization may thereby not only be redundant, but in fact harm the statistical efficiency, with potentially dire consequences on the overall convergence rate, or even non-converging or crashing executions. The resulting trade-off between computational and statistical efficiency (1.4) requires careful tuning of the level of parallelism (number of workers) m. The appropriate choice of m depends heavily on the properties of the optimization problem itself, as well as the choice of other hyper-parameters. In addition, previous literature exclusively assumes that the appropriate parallelism choice in *AsyncSGD* is *constant*, which is not necessarily the case.

Seminal works. The research direction of asynchronous iterative optimization is not new and sparked primarily due to the works by Bertsekas and Tsitsiklis, e.g., [33] and [34]. In those works, several highly relevant observations regarding asynchronous parallelization were highlighted, including for instance a characterization of different degrees of asynchrony, and predicts that it will imply reduced waiting, in particular under heterogeneity. In [34] in particular, the following is concluded: "very strong evidence suggesting that asynchronous iterations converge faster than their synchronous counterparts". Other predictions include usefulness of *roll-back compensation*, as well as *ring-based communication topology* in *AsyncSGD*, bearing strong resemblance to recently proposed approaches for delay compensation [35] and access pattern-efficiency in NUMA architectures [36], respectively. Such seminal works, in turn, build upon the early works by Jack L. Rosenfeld [37], as well as Dan Chazan and Willard Miranker [38], where asynchrony, initially referred to as *chaotic relaxation*, was first proposed as means to scale sequential iterative algorithms, at the time in the context of solving ordinary differential equations.

***AsyncSGD* and momentum.** More recently, Chaturapruek et al. [25] 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 [39] 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 workers, and assuming bounded staleness. Several works have
followed, aiming at understanding the impact of asynchrony on the convergence.
In [40] Mitliagkas et al. show that under certain stochastic staleness models,
asynchronous parallelism has an effect on convergence similar to momentum.
In [41] 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 [19] Alistarh
et al. introduces the concept of bounded divergence between the parameter
vector and the workers' view of it, proving convergence bounds for convex and
non-convex problems.

***AsyncSGD* and lock-freedom.**   HOGWILD! [20], introduced by Niu et
al., implements *AsyncSGD* with guarantees on *lock-freedom* (θ-progress) with
respect to the shared state θ. This is achieved in a straight-forward 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, 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 [42] [43]. De Sa
et. al [44] introduced a framework for analysis of HOGWILD!-style algorithms for
sparse problems. The analysis was extended in [45], showing that due to the lack
of θ-*consistency* 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. This 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. In [16]
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 comparison of different computing architectures.

***AsyncSGD* for DL.**   In [46] 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 comput-
ing 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 [47], Lopez et
al. proposes a semi-asynchronous SGD variant for DNN training, however

requiring a master thread or node 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 [48] theoretical convergence analysis is presented for *SyncSGD* with *once-in-a-while* synchronization. They mention that the analysis can guide in applying *SyncSGD* for DL, however the analysis requires strong convexity of the target function. [49] 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 [50] 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 [51], 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 [52] under assumptions of gradient sparsity and *read* and *write* operations having the same relative ordering. A similar approach was used in [24], however for synchronous SGD with the *softsync* protocol. In [24] 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 [53], 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 τ^{-1}. The staleness model in [53] is a uniform stochastic distribution, which implies a strict upper bound on the delays, making the system model partially asynchronous.

Asynchronous Federated Learning Recently, introducing an increased degree asynchrony in federated contexts has gained traction, primarily as a solution to the inherent heterogeneity of such contexts due to the emergence of scalability bottlenecks, analogous to the ones highlighted in Section 1.3.1. An extensive overview of such methods is provided in [54]. However, the existing approaches are partially asynchronous, in the spirit of the aforementioned *hybrid* approaches, and consequently cannot fully utilize the computational benefits of asynchronous computation. This is not surprising, considering the substantial knowledge gaps of the dynamics of *AsyncSGD* on practical applications in simpler shared-memory and distributed contexts, hence deployment in wide federated contexts naturally poses additional critical challenges.

1.4 Research problems and state of art

Limitations of synchrony. In *SyncSGD*, stragglers become bottlenecks, making every iteration only as fast as the slowest worker. This issue can however partially be reduced through relaxed semantics, such as SSP and the *n*-softsync protocol (See section 1.3). Moreover, the convergence of *SyncSGD*

under increasing parallelism is statistically equivalent to sequential SGD with a larger *mini-batch size b* [27], which is a hyper-parameter that requires careful tuning depending on the problem. In particular, the convergence can suffer if b is too large [31] [32]. Moreover, in distributed settings, such as in FL applications, synchronous distributed SGD rounds entails a significant communication overhead, which, in addition to the above statistical scalability limitation, is a contributing factor as to why such rounds typically employ only a limited number of federated worker nodes. As discussed also in section 1.3.1, the above considerations indicate limited scalability, as over-parallelization will impose large-batch properties, which in some cases worsens the convergence [27] of *SyncSGD*. This motivates further exploration of asynchronous parallelism as a complement to *SyncSGD* for improved scalability and overall efficient utilization of contemporary computing infrastructure.

Staleness and asynchrony-induced noise. *AsyncSGD* eliminates many scalability bottlenecks of *SyncSGD* due to reduced inter-worker coordination (stronger Δ-progress guarantees), however this also introduces other challenges related to asynchrony. As discussed in section 1.3.2, asynchronous access to, and update of, the shared state leads to staleness, i.e., that multiple updates occur concurrently to the gradient computation of some worker. The updates that are applied are hence not necessarily, in fact rarely in practice, based on the latest shared state, as captured in (1.5). The associated notion of AIN, i.e., the *asynchrony-induced noise* being the overall impact of asynchrony on the convergence trajectory of *AsyncSGD*, is mentioned in previous literature. However, no explicit definition of such has been proposed, let alone any method for measurement, leaving significant knowledge gaps in the dynamics of *Async-SGD*, and its deviation from its sequential counterpart. The impact of AIN on the statistical efficiency of the convergence is unpredictable, however some works indicate that the number of SGD iterations to ϵ-convergence increases linearly in the maximum staleness [44, 45]. 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. Crucial steps toward understanding how convergence is affected in *AsyncSGD* due to staleness are taken in e.g., [40], explicitly quantifying the impact of parallelism, 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.

Parallelism tuning. There is currently no well-established way to pre-determine appropriate parallelism ranges of a given ML problem prior to initiating and monitoring the execution, and such are typically found through costly extensive searches [47, 55]. Hence, without such searches, there are significant risks of (i) under-parallelism, with unnecessarily time-consuming executions, however more importantly (ii) over-parallelism, with the associated impact on the convergence trajectory and increased energy consumption. In fact, over-parallelism tends to imply higher staleness and AIN, with unstable, fluctuating loss values as a consequence, and even diverging and crashing executions in the worst case [51, 56]. Thus, using appropriate parallelism is important for ensuring high convergence quality, as well as for avoiding unnecessary consumption of computational resources, especially considering the general energy consumption due to modern deep learning research [7]. However, in practice only a small number of executions to reach a model of

sufficient quality can be tolerated, and exhaustive searches are excessively time consuming, and require significant computational resources. This implies a fatal idea-implementation gap, where on the one side *AsyncSGD* can provide tremendous acceleration of DL jobs, but it is agonizingly difficult to apply in practice. There is hence a need for *AsyncSGD* implementations which are efficient on wider ranges of parallelism, and otherwise automatically adjusts it as to increase the convergence rate, which will be discussed extensively in the chapters to follow.

Synchronization. As a consequence of Amdahl's law [57], when there is a synchronization overhead, the achievable speedup is bounded. In the context of *AsyncSGD*, this applies in particular for 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 workers 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 the subsequent chapters. Moreover, the degradation of statistical efficiency coupled to parallelism in *AsyncSGD* [41, 46], i.e., more iterations required to reach ϵ-convergence, implies that at some level of parallelism, which we refer to as the *system saturation point* m_S^*, additional workers 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 [44, 45], and explored further in subsequent chapters.

Progress and consistency guarantees for Δ vs. θ. As previously mentioned, read and update operations on θ become focal in *AsyncSGD*, since they constitute the remaining synchronization in the otherwise asynchronous algorithm. This is especially the case in shared-memory contexts, and there must be primitives in place to handle concurrent attempts to read and update by several workers, which unavoidably 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 [58]. To efficiently utilize multi-core systems, this was extended to shared-memory implementations [20, 39, 42]. 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 computationally expensive, in particular when the gradient computation step itself incurs little latency. In addition, the total time spent waiting for locks grows as more workers are introduced to the system, potentially posing a scalability bottleneck. By allowing completely uncoordinated component-wise atomic read and update operations, i.e., HOGWILD! [20], such contention is eliminated, allowing significant speedup for sparse and convex optimization problems in particular. However, for other problems, as summarized in Table 1.3, HOGWILD! introduces inconsistency when read and update operations occur concurrently. Such inconsistency, in addition to the potential impact of AIN, potentially incurs further statistical penalty; there are indications that the expected number of iterations required to converge increases linearly in $\sqrt{d}$ [45]. Consequently, there are challenges

in understanding whether it is worth the computational overhead to ensure consistency for a given problem. In addition, in such contexts it is not clear which synchronization primitives to utilize – which are scarce; there is a lack of 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!.

Benchmarking and evaluation.   There are 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 are 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 parallel SGD lacks established universal procedures for benchmarking, in particular for shared-memory contexts, 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 [59] in distributed contexts, such as FL, to exploring the impact of progress guarantees and synchronization in shared data structures [20, 45]. 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 [60] 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.

In this thesis, critical questions among the aforementioned challenges are addressed, as summarized in the following section.

1.5   Thesis contributions

This section highlights the key contributions of this thesis' associated publications, appearing in the subsequent chapters.

1.5.1   Convergence of staleness-adaptive SGD

The scalability limitations of traditional synchronous parallel SGD highlighted in section 1.4 motivate 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 section 1.4. In **Chapter A**, this is addressed through

statistical modelling of the behavior 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, analytical results that quantify the side-effect of asynchrony on the statistical efficiency are established (Lemma A.4.1). Moreover, the proposed models enable derivation of a staleness-adaptive step size, referred to as *MindTheStep-AsyncSGD*, which provably reduces this side-effect (Theorem A.4.5), and can in expectation, depending on the rate of adaptiveness, alter it into the more desired behavior of momentum (Theorem A.4.3, A.4.6). It is shown that the staleness-adaptive step size is efficiently computable (Theorem A.4.7), ensuring minimal additional synchronization overhead for maximal scalability capability, as described in section 1.4. An empirical evaluation is provided, benchmarking 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.

1.5.2   Framework for lock-freedom and consistency

AsyncSGD significantly reduces waiting compared to *SyncSGD*, as explained in the previous sections, and this holds particularly in shared-memory contexts. However, the remaining synchronization that is needed, in particular access to the shared state, becomes focal and constitutes a possible bottleneck. Motivated by analytical results in previous literature that indicate computational benefits of lock-freedom, however a statistical penalty from inconsistency and staleness, in **Chapter B** *Leashed-SGD* (lock-free consistent asynchronous shared-memory SGD) is proposed, 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. An analysis of the proposed framework is provided, particularly in terms of safety, memory consumption, and a model of the progression of parallel threads in the execution of SGD is proposed, which is used 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 have in particular guarantees on lock-freedom and atomicity, safety and exhaustiveness and bounds on memory consumption (Lemma B.3.1, B.3.2). Moreover, the progression of the algorithm over time is modelled, finding in particular fixed points in the system useful for estimating potential contention and the effect of the built-in contention-regulating mechanism (Theorem B.4.1, Corollary B.4.2 and B.4.3).

An extensive empirical study of *Leashed-SGD* with lock-free consistent methods for synchronization is conducted, evaluating its performance for MLP and CNN training for image classification. The empirical study focuses on scalability, dependence on hyper-parameters, distribution of the staleness; the study benchmarks the proposed method compared to established baselines, namely lock-based *AsyncSGD* and HOGWILD!. We draw the following main conclusions from the empirical study:

(i) *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 section 1.4. 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.

(ii) 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 convergence.

(iii) For MLP training, up to 27% reduced median running time for ϵ-convergence is observed for *Leashed-SGD* compared to baselines, with similar memory footprint. For CNN training, a $\times 4$ speedup for ϵ-convergence is observed, 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. Hence, critical steps are taken towards addressing the challenges (highlighted in section 1.4) that the community faces regarding a general platform for further exploration of aspects of fine-grained synchronization in this domain.

1.5.3   Instance-based step size adaptiveness

As highlighted above, there are significant potential benefits of adaptiveness to asynchrony, as shown in **Chapter A** and **B**, as well as concurrent works (see Section 1.3.2). Additionally, convergence stability is critically sensitive to parallelism degree and progress and consistency guarantees of the algorithmic implementation, and the mechanisms to ensure them, e.g., locking, and in **Chapter C**, these aspects are studied in conjunction, in order to understand how *AsyncSGD* can be utilized effectively in practice.

Moreover, the existing approaches to staleness-adaptiveness are either (i) heuristic or (ii) model-based, and common approaches typically use straightforward rules, such as dampening the step size linearly, or exponentially, based on the observed staleness. An inherent pitfall of such approaches is that the *overall* magnitude of the step size is altered, which by itself will impact the statistical efficiency, especially for applications which are sensitive to that parameter, including DL. However, this is not rare in the literature, e.g., in [24, 53, 56, 61], which exclusively diminish the overall step size. This is problematic for several reasons, e.g., (i) it reduces applicability to step size-sensitive applications, and (ii) it introduces ambiguity regarding the source of potential performance

improvements, reducing comparability between methods. **Chapter C** targets the above challenges, in particular by introducing the first *instance-based staleness-adaptive* step size function TAIL-τ. Also, a framework for <u>*a*daptiveness *to* <u>s</u>taleness in <u>a</u>synchronous <u>p</u>arallel <u>SGD</u></u> (`ASAP.SGD`) is established, capturing key properties of general staleness-adaptive step size functions. In detail:

(i) The analytical framework `ASAP.SGD` is established, which captures general staleness-adaptive step size function properties that are (i) necessary for maintaining overall step size magnitudes and ensuring method comparability, and (ii) desired for prioritizing gradient freshness, and hence is suitable to serve as a common platform for guiding the design of new asynchrony-aware step size functions.

(ii) Within `ASAP.SGD`, TAIL-τ is introduced – an instance-based dynamic staleness-adaptive step size function, which utilizes the overall observed *staleness distribution* as means to implicitly take underlying system parameters into account, to generate an execution-specific adaptation strategy.

(iii) The convergence properties of the proposed TAIL-τ step size are analyzed, as well as the wider collection of general ones within the `ASAP.SGD` framework, for convex and non-convex applications (Theorem C.5.6). In addition, novel convergence bounds are established, in particular for loss functions satisfying the Polyak-Lojasiewicz (PL) condition, a more realistic generalization of strong convexity, which applies to a wide set of relevant applications, including least squares, logistic regression, support vector machines [62] and certain deep ANNs [63] (Theorem C.5.8).

(iv) The TAIL-τ function is evaluated on several benchmark implementations and applications that capture several representative system features associated with synchronization, parallelism, execution-ordering properties. The results show that LeNet and MLP training with *AsyncSGD*, on MNIST, Fashion-MNIST, and CIFAR-10, TAIL-τ persistently achieves significantly faster convergence (60% speedup on average). The results show that this holds for three fundamentally different *AsyncSGD* implementations, namely (i) lock-based *AsyncSGD*, (ii) HOGWILD!, and (iii) the lock-free consistent *Leashed-SGD* implementation, introduced in **Chapter B**. The evaluation shows additionally that TAIL-τ drastically lowers the risk of non-converging executions, especially to higher precision.

1.5.4 Elastic parallelism control

As highlighted above, appropriate degree of parallelism is important for ensuring high convergence quality, as well as for avoiding unnecessary consumption of computational resources, especially considering the energy consumption due to modern deep learning research [7]. This is particularly important for *AsyncSGD*, since over-parallelism leads to higher staleness and AIN, with unstable, fluctuating loss trajectories, as well as non-converging executions, with the associated energy losses. Since exhaustive searches to find optimal parallelism is infeasible in practice, there is a new generation of robust, *instance-adaptive AsyncSGD* methods, that balance the computational vs. statistical trade-off, while retaining the computational benefits of asynchrony. Critical

steps to achieve such are taken in **Chapter D**, where the proposed *ElAsyncSGD* automatically balances the computational vs. statistical efficiency trade-off by adjusting the parallelism level. By doing so, *ElAsyncSGD* enables *single* executions with fast and stable convergence, and competes with, and most of the time beats, the *best* constant-parallelism baselines that have been tuned to optimal performance through exhaustive parallelism searches, at a significantly lower cost in computing resources. Hence, evidence is provided that the optimal parallelism is indeed not constant, but rather varies throughout the convergence trajectory. In more detail, the chapter makes the following contributions:

 (i) A formal definition of *asynchrony-induced noise* (AIN) is provided, denoted by ξ, which in particular quantifies the deviation of *AsyncSGD* from its sequential counterpart, however, does the same for general numerical iterative algorithms. In addition, a generic algorithmic extension of *AsyncSGD* for efficiently measuring the AIN in real-time in such algorithms is introduced, which is used to report its magnitudes for several relevant DL benchmarking problems in the empirical study.

 (ii) Novel convergence bounds are established, focusing particularly on the impact of ξ and the parallelism degree, on the asymptotic convergence of *AsyncSGD*. The convergence of *AsyncSGD* and *ElAsyncSGD* are established on general non-convex problems (Theorem D.5.6, D.5.8), as well as ones satisfying the Polyak-Lojasiewicz criterion, which applies for several relevant DL problems (Theorem D.5.10).

(iii) *ElAsyncSGD* is introduced as an extension of *AsyncSGD*, which dynamically regulates the parallelism level in real-time based on the instantaneous convergence rate of the execution instance, targeting a balance between computational and statistical efficiency, while striving for minimal consumption of computational resources.

(iv) An extensive evaluation of the proposed *ElAsyncSGD* is presented, benchmarking against standard *AsyncSGD* with *tuned best constant parallelism*, on several DL benchmarks, including LeNet and MLP training on MNIST, Fashion-MNIST and CIFAR-10. The evaluation reveals that the intelligent parallelism regulation of *ElAsyncSGD* entails drastic reductions in thread-seconds, and hence overall energy consumed by computational resources. In addition, *ElAsyncSGD* additionally exhibits more stable convergence trajectories, and converges to higher precision, thanks to improved computational vs. statistical efficiency balance.

1.6   Conclusions and new research directions

The paradigm of *synchronous parallel SGD* (*SyncSGD*) is standard nowadays, and its application span ranges from standard GPU-accelerated desktops to high-end distributed networks of computing nodes engaging in *Federated Learning*. *SyncSGD* is effective on various modern ML tasks, and is widely adopted due to the simplicity and understandability – it is in fact equivalent to traditional, sequential SGD. However, the strict scalability limitations of *SyncSGD*, that are highlighted in this thesis and related work, motivate *asynchronous parallelism*

(*AsyncSGD*). By relaxing the sequential SGD semantics, *AsyncSGD* provides new magnitudes of potential speedup on a variety of optimization problems, including modern ML tasks, and is in addition algorithmically orthogonal to *Sync-SGD*, and can hence be applied in conjunction to make use of the two optimally.

At present, there are several known challenges of *AsyncSGD*, stemming from the presence of *asynchrony-induced noise* (AIN) and *staleness* in its updates. This thesis targets those, in particular through (i) formalization of emerging notions, such as AIN, (ii) analysis of the impact of asynchrony on the convergence properties under realistic assumptions, (iii) extending *AsyncSGD* with real-time awareness to asynchrony-related phenomena in an instance-adaptive fashion, as well as (iv) extensive evaluation of such for practically relevant benchmarking applications. The above is performed with general applications in mind, ranging from distributed federated learning deployments to shared-memory contexts.

This thesis confirms that *AsyncSGD* significantly can accelerate relevant ML applications, in particular *Deep Learning* ones. On the other hand, *AsyncSGD* is sensitive to its parameters, especially the *parallelism level*; over-parallelism can entail slower executions, and even non-converging executions. The sensitivity can be alleviated greatly, and significant performance benefits achieved, through the *asynchrony-aware* adaptiveness, in particular instance-based, mechanisms that are proposed in this thesis. Moreover, in shared-memory contexts, it is established that the mechanisms for fine-grained synchronization of access to the shared state have a similar impact. Additionally, this thesis discovers that a *time-varying* parallelism level is superior to *the best constant* one, in terms of convergence stability, as well as consumption of computing resources – an elastic parallelism-regulating *AsyncSGD* extension is proposed, which achieves superior convergence trajectories, while saving significant computing resources.

Within the scope of this thesis, it is established that *asynchrony-awareness* and *elasticity* have tremendous benefits in *AsyncSGD* deployments, both contributing to improved statistical efficiency, and the latter particularly to efficiency in computing resources. With this in mind, and considering the aforementioned limitations of traditional synchronous approaches, a natural next step, and recommended future work, is to explore algorithmic extensions that improve *AsyncSGD* in conjunction – in particular the ones proposed in this thesis for resource-aware intelligent elasticity and asynchrony-awareness and adaptiveness. By doing so, vital lessons regarding their combined impact of such extensions may be learned, and crucial steps may be taken towards enabling increased utilization of *AsyncSGD* in practical deployments. Considering the rapidly growing needs for convergence speed, scalability, and resource efficiency of such deployments, every such step is vital, as *AsyncSGD* will surely play a growingly integral role in modern machine learning and AI systems.

Chapter A

Adaptation of the article:

MindTheStep-AsyncSGD: Adaptive Asynchronous Parallel Stochastic Gradient Descent

K. Bäckström, M. Papatriantafilou, P. Tsigas

.

Abstract

Stochastic Gradient Descent (SGD) is very useful in optimization problems with high-dimensional non-convex target functions, and hence constitutes an important component of several Machine Learning and Data Analytics methods. Recently there have been significant works on understanding the parallelism inherent to SGD, and its convergence properties. Asynchronous, parallel SGD (*AsyncSGD*) has received particular attention, due to observed performance benefits. On the other hand, asynchrony implies inherent challenges in understanding the execution of the algorithm and its convergence, stemming from the fact that the contribution of a thread might be based on an old (stale) view of the state. In this work we aim to deepen the understanding of *AsyncSGD* in order to increase the statistical efficiency in the presence of stale gradients. We propose new models for capturing the nature of the staleness distribution in a practical setting. Using the proposed models, we derive a staleness-adaptive SGD framework, *MindTheStep-AsyncSGD*, for adapting the step size in an online-fashion, which provably reduces the negative impact of asynchrony. Moreover, we provide general convergence time bounds for a wide class of staleness-adaptive step size strategies for convex target functions. We also provide a detailed empirical study, showing how our approach implies faster convergence for deep learning applications.

A.1   Introduction

The explosion of data volumes available for Machine Learning (ML) has posed tremendous scalability challenges for machine intelligence systems. Understanding the ability to parallelize, scale and guarantee convergence of basic ML methods under different synchronization and consistency scenarios has recently attracted a significant interest in the literature. The classic Stochastic Gradient Descent (SGD) algorithm is a significant target of research studying its convergence properties under parallelism.

In SGD, the goal is to minimize a function $L : \mathbb{R}^d \to \mathbb{R}$ of a d-dimensional vector θ using a first-order light-weight iterative optimization approach; i.e., given a randomly chosen starting point θ_0, SGD repeatedly changes θ in the negative direction of a stochastic gradient sample, which provably is the direction in which the target function is expected to decrease the most. The step size η_i defines how coarse the updates are:

$$\theta_{i+1} \leftarrow \theta_i - \eta_i \nabla L(\theta_i) \tag{A.1}$$

SGD is very useful in nonconvex optimization with high-dimensional target functions, and hence constitutes a major part in several ML and Data Analytics methods, such as regression, classification and clustering. In many applications, the target function is differentiable and the gradient can be efficiently computed, e.g., Artificial Neural Networks (ANNs) using Back Propagation [64].

To better utilize modern computing architectures, recent efforts propose *parallel* SGD methods, complemented with different approaches for analyzing the convergence. However, asynchrony poses challenges in understanding the algorithm due to *stale* views of the state of θ, which leads to reduced *statistical efficiency* in the SGD steps, requiring a larger number of iterations for achieving similar performance. In this work, we focus on increasing the statistical efficiency of the SGD steps and propose a staleness-adaptive framework *MindTheStep-AsyncSGD* that adapts parameters to significantly reduce the number of SGD steps required to reach sufficient performance. Our framework is compatible with recent orthogonal works focusing on computational efficiency, such as efficient parameter server architectures [23] [65] and efficient gradient communication [59] [66].

Motivation and summary of state-of-the-art.   Many established ML methods, such as ANN training and Regression, constitute of minimizing a function $L(\theta)$ that takes the form of a finite sum of error terms $L(d; \theta)$ parameterized by θ, evaluated at different data points x from a given set D of measurements:

$$L_D(\theta) = \frac{1}{|D|} \sum_{x \in D} L(x; \theta) \tag{A.2}$$

where the parameter vector θ, encodes previously gathered features from D. In this context, SGD typically selects mini-batches $B \subseteq D$ over which L_B is minimized and is known as *Mini-Batch* Gradient Descent (MBGD). This type of SGD reduces the computational load in each step and hence enables processing of large datasets more efficiently. Moreover, randomly selecting mini-batches induces stochastic variation in the algorithm, which makes it effective in non-convex problems as well.

A natural approach to distribute work for objective functions of the form (A.2) is to utilize *data parallelism* [22], where different workers (threads in a multicore system or nodes in a distributed one) run SGD over different subsets of D. This will result in differently learned parameter vectors θ, which are aggregated, commonly in a *shared parameter server* (thread or node). The aggregation typically computes the average of the workers contributions; this approach is referred to as *Synchronous* Parallel SGD (*SyncSGD*) due to its barrier-based nature. In its simple form, *SyncSGD* has scalability issues due to the waiting time that is inherent in the aggregation when different workers compute at different speeds. As more workers are introduced to the system, the waiting time will increase unbounded. Requiring only a fixed number of workers in the aggregation, known as λ-*softsync*, bounds this waiting time. The barrier-based nature of the synchronous approaches to parallel SGD enables a straightforward (yet expensive) linearization making the vast analysis of classical SGD applicable also to the parallel version. As a result, its convergence is well-understood also in the parallel case, which however suffers from the performance-degradation of the barrier mechanisms.

An alternative type of parallelization is *Asynchronous* Parallel SGD (*AsyncSGD*), in which workers *get* and *update* the shared variable θ independently of each other. There are inherent benefits in performance due to that *AsyncSGD* eliminates waiting time, however the lack of coordination implies that gradients can be computed based on *stale* (old) views of θ, which are *statistically inefficient*. However, gains in *computational efficiency* due to parallelism and asynchrony can compensate for this, reducing the wall-clock computation time.

Challenges. *AsyncSGD* shows performance benefits due to allowing workers to continue doing work independently of the progress of other workers. However, asynchrony comes with inherent challenges in understanding the execution of the algorithm and its convergence. In this work we address mainly (i) understanding the impact on the convergence and statistical efficiency of *stale* gradients computed based on old views of θ and (ii) how to adapt the step size in SGD to accommodate for the presence of asynchrony and delays in the system.

Contributions. With the above challenges in mind, in this work we aim to increase the understanding of *AsyncSGD* and the effect of stale gradients in order to increase the statistical efficiency of the SGD iterations. To achieve this, we find models suitable for capturing the nature of the staleness distribution in a practical setting. Under the proposed models, we derive a staleness-adaptive framework *MindTheStep-AsyncSGD* for adapting the step size in the presence of stale gradients. We prove analytically that our framework reduces the negative impact of asynchrony. In addition, we provide an empirical study which shows that our proposed method exhibits faster convergence by reducing the number of required SGD iterations compared to *AsyncSGD* with constant step size. In some more detail:

- We prove analytically scalability limitations of the standard *SyncSGD* approach that were observed empirically in other works.

- We propose a new distribution model for capturing the staleness in *AsyncSGD*, and show analytically how the optimal parameters can be chosen efficiently. We evaluate our proposed models by measuring the distance

to the real staleness distribution observed empirically in a deep learning application and compare the performance to models proposed in other works.

- Under the proposed distribution models, we derive efficiently computable staleness-adaptive step size functions which we show analytically can control the impact of asynchrony. We show how this enables *tuning* the implicit momentum to any desired value.

- We provide an empirical evaluation of *MindTheStep-AsyncSGD* using the staleness-adaptive step size function derived from our proposed model, where we observe a significant reduction in the number of SGD iterations required to reach sufficient performance.

Before the presentation of the results in Sections A.3-A.6, we outline preliminaries and background. Following the results-sections, we provide an extensive discussion on related work, conclusions and future work.

A.2   Preliminaries

A.2.1   Stochastic gradient descent

We consider the optimization problem

$$\underset{\theta}{\text{minimize}} \quad L(\theta) \tag{A.3}$$

for a function $L : \mathbb{R}^d \to \mathbb{R}$. In this context, we focus on methods to address this minimization problem (A.3) using SGD, defined by (A.1) for some randomly chosen starting position θ_0. We assume that the stochastic gradient $\nabla \tilde{L}$ is an unbiased estimator of ∇L, i.e., $\mathbf{E}[\nabla \tilde{L}(\theta) \mid \theta] = \nabla L(\theta)$ for all θ. This assumption holds for several relevant applications, in particular for problems of the form (A.2), including regression and ANN training. We assume that the stochastic gradient samples are i.i.d, which is reasonable since the sampling occurs independently by different threads. For the analysis in section A.5 we adopt some additional standard assumptions on smoothness and convexity which we will introduce in that section.

A.2.2   System model and asynchronous SGD

We consider a system with $\hat{m}$ workers (that can be threads in a multicore system or nodes in a distributed one), which repeatedly compute gradient contributions based on independently drawn data mini-batches from some given data set D. We also consider a *shared parameter server* (that can be a thread or a node respectively), which communicates with each of the workers independently, to give state information and get updates that it applies according to the algorithm it follows.

The $\hat{m}$ asynchronous workers aim at performing SGD updates according to (A.1). Since each worker m must get a state θ_i prior to computing a gradient, there can be intermediate updates from other workers before gradient from m is applied. The number of such updates defines the *staleness* τ_i corresponding to the gradient $\nabla L(\theta_i)$.

Assuming that the read and update operations can be performed atomically (see details in Section A.4), under the system model above, the SGD update (A.1) becomes

$$\theta_{i+1} \leftarrow \theta_i - \eta_i \nabla L(v_i) \tag{A.4}$$

where $v_i = \theta_{i-\tau_i}$ is the thread's *view* of θ.

We assume that the staleness values τ_i constitute a *stochastic process* which is influenced by the computation speed of individual threads as well as the scheduler. Unless explicitly specified, we make no particular assumptions on the scheduler or computational speed among threads, except that all delays follow the same distribution with the same expected delay, i.e., $\mathbf{E}[\tau_i] = \bar{\tau}$ for all i. We do not require the staleness to be globally upper bounded, only that updates are eventually applied, making our system model *fully asynchronous*.

While we assume above that gradient samples are pairwise independent, it is not reasonable to make the same assumption for the staleness. In fact, a staleness τ_i is by definition dependent on the writing time of concurrent updates, which in turn are dependent on their respective staleness values. For the analysis in Section A.5, we assume that *stochastic gradients* and *staleness* are uncorrelated, i.e., that the stochastic variation of the gradients does not influence the delays and vice versa. This is also a realistic assumption, since delays are due to computation time and scheduling and the gradient's stochastic variation is due to random draws from a dataset.

A.2.3   Momentum

SGD is typically inefficient in *narrow valleys* when the target function in some neighborhood increases more rapidly in one direction relative to another. Such neighborhoods are frequent in target functions that arise in ML applications due to their inherent highly irregular and non-convex nature. Adding *momentum* (A.5) to SGD has been seen to significantly improve the convergence speed for such functions. SGD with momentum, defined in (A.5), takes all previous gradient samples into account with exponentially decaying magnitude in its parameter μ. As pointed out in [40], μ is often left out in parameter tuning, and in some instances even failed to be reported [67]. However, the optimal value of algorithmic parameters such as μ, just like η, depends on the problem, underlying hardware, as well as the choice of other parameters. Tuning μ has been shown to significantly improve performance [15], especially under asynchrony [40].

For $\mu \in [0,1]$, SGD with momentum is defined by

$$\theta_{i+1} \leftarrow \theta_i + \mu(\theta_i - \theta_{i-1}) - \eta_i \nabla L(\theta_i) \tag{A.5}$$

A.3   On the scalability of *SyncSGD*

Optimal convergence with *SyncSGD* requires, as observed empirically in [27], that the mini-batch size is reduced as the number of worker nodes increases. We prove analytically this empirical observation. We show that, from an optimization perspective, the effect of more workers on the convergence is equivalent to using a larger mini-batch size, which we refer to as the *effective*

mini-batch size. For maintaining a desired effective mini-batch size, which is the case in many applications [31] [32], workers must hence use smaller batches prior to the aggregation. Since the mini-batch size clearly is lower bounded, there is an implied strict upper bound on the number of worker nodes that can leverage the parallelization, which provides a bound on the scalability of the synchronous approach.

In mini-batch GD for target functions $L(\theta)$ of the form (A.2) the stochasticity is due to randomly drawing mini-batches B of size b from a dataset D without replacement. For any mini-batch size b, we have that $\tilde{L}(\theta) = L_B(\theta)$ is an unbiased estimator of $L(\theta)$ since

$$\mathbf{E}[L(\theta)] = \mathbf{E}[L_B(\theta)] = \frac{b}{|D|} \sum_j L_{B_j}(\theta)$$

$$= \frac{b}{|D|} \sum_j \frac{1}{b} \sum_{x \in B_j} L(x; \theta) = \frac{1}{|D|} \sum_{x \in D} L(x; \theta) = L(\theta)$$

Hence, the SGD updates are in expectation representing the entire dataset D. Note that we assume $\bigcup B_i = D$. We have, however, that as the batch size b increases, the stochasticity of $\tilde{L}(\theta)$ diminishes. One can realize this by considering the extreme case $b = |D|$ for which the data sampling is deterministic. Hence, decreasing b induces larger variance for the distribution of $L(\theta)$. This enables SGD to avoid local minima and hence be effective also in non-convex optimization problems.

The optimal value of b is dependent on the problem and requires tuning; it has been seen that the convergence can suffer if b is too large [31] [32].

In the following theorem we show that by increasing the number of worker nodes in *SyncSGD*, from an optimization perspective, we get a behavior equivalent to a sequential execution of SGD with a larger mini-batch size, which we refer to as the *effective* mini-batch size.

Theorem A.3.1. SyncSGD *with $\hat{m}$ workers, all using batch size b, is equivalent to a sequential execution of SGD with batch size $\hat{m} \cdot b$, referred to as effective batch size.*

Proof. Consider the case with two worker nodes. Assuming that the batches are disjoint, which is likely for large datasets, each SGD step is of the form

$$\theta_{i+1} = \frac{\left(\theta_i - \eta \nabla L_{B_1}(\theta_i)\right) + \left(\theta_i - \eta \nabla L_{B_2}(\theta_i)\right)}{2}$$

$$= \theta_i - \frac{\eta}{2} \left(\nabla L_{B_1}(\theta_i) + \nabla L_{B_2}(\theta_i)\right)$$

For mini-batch GD, i.e., a target function of the form (A.2), and with mini-batch size b, the above formula becomes:

$$\theta_{i+1} = \theta_i - \frac{\eta}{2} \left(\nabla \frac{1}{b} \sum_{x \in B_1} L(x, \theta_i) + \nabla \frac{1}{b} \sum_{x \in B_2} L(x, \theta_i)\right)$$

From linearity of the gradient, we have

$$\theta_{i+1} = \theta_i - \eta \nabla \frac{1}{2b} \sum_{x \in B_1 \cup B_2} L(x, \theta_i) = \theta_i - \eta \nabla L_{B_1 \cup B_2}(\theta_i)$$

that corresponds to the SGD step with batch size $2b$. This inductively implies the theorem statement. □

Since the mini-batch size is clearly lower bounded, Theorem A.3.1 implies that for a sufficiently large number of worker nodes, the effective mini-batch size scales linearly in the number of workers nodes. In order to maintain reasonable mini-batch size with sufficient variation in the updates, this implies a strict upper bound on the number of workers nodes. Moreover, under the assumption that there is an optimal mini-batch size b^* for a given problem, which has been seen to be a common assumption, we have that the maximum number of workers possible in order to achieve optimal convergence is exactly $\hat{m} = b^*$, each using mini-batch size $b = 1$.

A.4 The proposed framework

We outline *MindTheStep-AsyncSGD* for staleness-adaptive steps and analyze how to choose a suitable adaptive step function under different staleness models.

A.4.1 The *MindTheStep-AsyncSGD* framework

We consider a standard parameter-server type of algorithm [23] [58], with atomic read and write operations, ensuring that workers acquire consistent views of the state θ. In a distributed system, consistency can be realized through the communication protocol. In a multi-core system, where worker nodes are threads and θ can be stored on shared memory, consistency can be realized with appropriate synchronization and producer-consumer data structures, with the extra benefit that they can pass pointers to the data (parameter arrays) instead of moving it. In Algorithm A.4.1 we show the pseudocode for *MindTheStep-AsyncSGD*, describing how standard *AsyncSGD* using a parameter server (thread or node) is extended with a staleness-adaptive step.

Algorithm A.1 *MindTheStep-AsyncSGD*

1: GLOBAL start point θ_0, functions $L(\theta)$ and $\eta(\tau)$

2: <u>Worker w</u>	**9:** <u>Parameter server S</u>
3: $(i, \theta) \leftarrow (0, \theta_0)$	**10:** $(i', \theta) \leftarrow (0, \theta_0)$
4: repeat	**11: repeat**
5: compute $g \leftarrow \nabla L(\theta)$	**12:** *receive* (i, g) from a ready worker w
6: *send* (i, g) to S	**13:** $\tau \leftarrow i' - i$
7: *receive* (i, θ) from S	**14:** $\theta \leftarrow \theta - \eta(\tau)g$
8: until break	**15:** $i' \leftarrow i' + 1$
	16: *send* (i', θ) to w
	17: until break

Note that *MindTheStep-AsyncSGD* as a framework essentially "modularizes" the role of η as a parameter that can configure and tune performance, with criteria and benefits that are analyzed in the next subsection.

A.4.2   Tuning the impact of asynchrony

As pointed out in [40], asynchrony and delays introduce *memory* in the behavior of the algorithms. In particular, in [40, Theorem 2], they quantify this and show its resemblance to momentum, however for a constant step size. The corresponding result for a stochastic staleness-adaptive step size is formulated here:

Lemma A.4.1. *Let τ be distributed according to some PDF p such that $P[\tau = i] = p(i)$. Then, for an adaptive step size function $\eta(\tau)$, we have*

$$\mathbf{E}[\theta_{i+1} - \theta_i] = \mathbf{E}[\theta_i - \theta_{i-1}] + \sum_{j=0}^{\infty} \big(p(j)\eta(j) - \tag{A.6}$$

$$p(j+1)\eta(j+1)\big)\nabla f(x_{i-j-1}) - p(0)\eta(0)\nabla L(\theta_i)$$

The proof of Lemma A.4.1 follows the structure of the one in [40], now taking into account the adaptive step size. The main takeaways from Lemma A.4.1 are that, under asynchrony, (i) the gradient contribution diminishes as the number of workers increases[2]; (ii) there is a momentum-like term introduced with parameter $\mu = 1$ and (iii) the update depends on the series term:

$$\Sigma_{p,\eta}^{\nabla} = \sum_{j=0}^{\infty} \big(p(j)\eta(j) - p(j+1)\eta\big)\nabla L(\theta_{i-j-1}) \tag{A.7}$$

which quantifies the potential impact of stale gradients depending on the distribution of τ.

The issue of diminishing gradient contributions as the number of workers increase can in theory be resolved by choosing a larger η. However, this would require step sizes proportional to $p(0)^{-1}$, which rapidly grows out of bounds as the number of workers increases. Since large η can significantly impact the statistical efficiency of the SGD steps in practice and in fact needs to be carefully tuned, this poses a scalability limitation.

This is where *MindTheStep-AsyncSGD* can help tune the impact of asynchrony, as we show in the following.

Momentum from geometric τ.   Assuming a geometrically distributed τ, the series $\Sigma_{p,\eta}^{\nabla}$ is manifested in the convergence behavior in the form of asynchrony-induced memory with a *momentum* effect; see Theorem 3 of [40], repeated here for self-containment:

Theorem A.4.2 ([40]). *Let all τ_i be geometrically distributed with parameter p, i.e., $\mathbf{P}[\tau = k] = p(1-p)^k$. Then, for a constant η, the expected update (A.4) becomes*

$$\mathbf{E}[\theta_{i+1} - \theta_i] = (1-p)\mathbf{E}[\theta_i - \theta_{i-1}] - p\eta\nabla L(\theta_i) \tag{A.8}$$

Theorem A.4.2 is easily confirmed by substituting $p(i)$ in (A.7) with constant η with the geometric PDF, which yields $\Sigma_{p,\eta}^{\nabla} = -p\mathbf{E}[\theta_i - \theta_{i-1}]$.

Eq. (A.8) resembles the definition of momentum, with expected implicit asynchrony-induced momentum of magnitude $\mu = 1 - p$. As the number of

---

[2]Here it is assumed that $p(0)$ tends to zero as the number of workers increases. This is easily realized for our proposed CMP τ model (A.12). For the geometric staleness model we confirm empirically in section A.6 that this assumption holds in practice, recall that $p(0) = p$.

workers grow and p tends to 0, Theorem A.4.2 suggests an implicit momentum that approaches 1. This would imply a scalability limitation since the parameter μ requires careful tuning.

Assuming a geometric staleness model, we show in the following how *MindTheStep-AsyncSGD* with a particular step size function resolves this issue:

Theorem A.4.3. *Let staleness $\tau \in Geom(p)$ and*

$$\eta_i = C^{-\tau_i} p^{-1} \eta \tag{A.9}$$

where C is a parameter to be chosen suitably. Then

$$\mathbf{E}[\theta_{i+1} - \theta_i] = \mu_{C,p} \mathbf{E}[\theta_i - \theta_{i-1}] - \eta \nabla L(\theta_i)$$

and the implicit asynchrony-induced momentum is

$$\mu_{C,p} = 2 - (1 - p)/C \tag{A.10}$$

Proof. We have from (A.4)

$$
\begin{aligned}
\theta_{i+1} - \theta_i &= -\eta_i \nabla L(v_i) \\
&= \theta_i - \theta_{i-1} - (\theta_i - \theta_{i-1}) - \eta_i \nabla L(v_i) \\
&= \theta_i - \theta_{i-1} + \eta_i \nabla L(v_{i-1}) - \eta_i \nabla L(v_i)
\end{aligned}
$$

Since the gradient and staleness processes are independent, we take first expectation conditioned on the staleness

$$\mathbf{E}[\theta_{i+1} - \theta_i \mid \tau_i, \tau_{i-1}] = \mathbf{E}[\theta_i - \theta_{i-1} \mid \tau_i, \tau_{i-1}] + \eta_i \nabla L(v_{i-1}) - \eta_i \nabla L(v_i)$$

Now, take expectation w.r.t. the stochastic staleness τ_i, τ_{i-1}

$$
\begin{aligned}
\mathbf{E}[\theta_{i+1} - \theta_i] &= \mathbf{E}[\theta_i - \theta_{i-1}] \\
&\quad + \mathbf{E}[\eta_i \nabla L(v_{i-1})] - \mathbf{E}[\eta_i \nabla L(v_i)] \\
&= \mathbf{E}[\theta_i - \theta_{i-1}] + \sum_{j=0}^{\infty} P[\tau = j] \frac{\eta \nabla L(\theta_{i-j-1})}{C^j p} \\
&\quad - \sum_{j=0}^{\infty} P[\tau = j] \frac{\eta \nabla L(\theta_{i-j})}{C^j p} \\
&= \mathbf{E}[\theta_i - \theta_{i-1}] + p \sum_{j=0}^{\infty} (1-p)^j \frac{\eta \nabla L(\theta_{i-j-1})}{C^j p} \\
&\quad - p \sum_{j=0}^{\infty} (1-p)^j \frac{\eta \nabla L(\theta_{i-j})}{C^j p} \\
&= \mathbf{E}[\theta_i - \theta_{i-1}] - \eta \nabla L(\theta_i) + \sum_{j=0}^{\infty} (1-p)^j \frac{\eta \nabla L(\theta_{i-j-1})}{C^j} \\
&\quad - \sum_{j=1}^{\infty} (1-p)^j \frac{\eta \nabla L(\theta_{i-j})}{C^j}
\end{aligned}
$$

$$= \mathbf{E}[\theta_i - \theta_{i-1}] - \eta \nabla L(\theta_i)$$

$$+ \sum_{j=0}^{\infty} \left(\frac{(1-p)^j}{C^j} - \frac{(1-p)^{j+1}}{C^{j+1}} \right) \eta \nabla L(\theta_{i-j-1})$$

$$= \mathbf{E}[\theta_i - \theta_{i-1}] - \eta \nabla L(\theta_i)$$

$$+ \sum_{j=0}^{\infty} \frac{(1-p)^j}{C^j} \left(1 - \frac{1-p}{C} \right) \eta \nabla L(\theta_{i-j-1})$$

$$= \mathbf{E}[\theta_i - \theta_{i-1}] - \eta \nabla L(\theta_i)$$

$$+ \left(1 - \frac{1-p}{C} \right) \sum_{j=0}^{\infty} \frac{p(1-p)^j}{C^j p^{j+1}} \eta \nabla L(\theta_{i-j-1})$$

$$= \mathbf{E}[\theta_i - \theta_{i-1}] - \eta \nabla L(\theta_i)$$

$$+ \left(1 - \frac{1-p}{C} \right) \mathbf{E}[\eta_i \nabla L(v_{i-1})]$$

$$= \mathbf{E}[\theta_i - \theta_{i-1}] - \eta \nabla L(\theta_i) + \left(1 - \frac{1-p}{C} \right) \mathbf{E}[\theta_i - \theta_{i-1}]$$

$$= \left(2 - \frac{1-p}{C} \right) \mathbf{E}[\theta_i - \theta_{i-1}] - \eta \nabla L(\theta_i)$$

$$\square$$

Note that the expected implicit momentum vanishes for $C = (1-p)/2$. More generally:

Corollary A.4.4. *Any desired momentum μ^* is, in expectation, implicitly induced by asynchrony by using the staleness-adaptive step size in (A.9) with*

$$C = (1-p)/(2 - \mu^*) \tag{A.11}$$

Applicability of geometric τ. Each gradient staleness is comprised of two parts, one of which is the staleness τ_C which counts the number of gradients applied from other workers concurrent with the gradient computation. The second part of the staleness, which we denote τ_S, counts, after the gradient computation of a worker finishes, the number of gradients from other workers which are applied first, which is decided by the order with which the workers are scheduled to apply their updates. The complete staleness of a gradient is $\tau = \tau_C + \tau_S$. Note that, if we assume a uniform fair stochastic scheduler, then τ_S is decided exactly by the number of Bernoulli trials until a specific gradient is chosen, hence $\tau_S \in \text{Geom}(\cdot)$. The geometric τ model is therefore applicable for problems where $\tau_C << \tau_S$, i.e., when the gradient computation time typically is smaller than the time it takes to apply a computed gradient (eq. A.4).

Now consider also relevant applications of SGD where the gradient computation time τ_C is far from negligible, e.g., the increasingly popular Deep Learning, which typically includes ANN training with BackProp [64] for gradient computation. The BackProp algorithm requires in the best-case multiple multiplications of matrices of dimension d, which by far dominates the SGD update step (A.4) which consists of exactly d floating point multiplications and additions. For such applications the geometric τ model is hence not sufficient;

we confirm this empirically in Section A.6. In the following, we propose a class of τ distributions which is more suitable.

Conway-Maxwell-Poisson (CMP) τ. Considering applications with time-consuming gradient computation such as ANN training, we aim to find a suitable staleness model. Since now we consider (i) that $\tau_C >> \tau_S$ and (ii) that applying a computed gradient is relatively fast, we can consider the completion of gradient computations as rare arrival events. This opts for a variant of the Poisson distribution, such as the CMP distribution which in addition to Poisson has a parameter ν which controls the rate of decay. We have that $\tau \in \mathrm{CMP}(\lambda, \nu)$ if

$$P[\tau = i] = \frac{1}{Z(\lambda, \nu)} \frac{\lambda^i}{(i!)^\nu} \ , \ Z(\lambda, \nu) = \sum_{j=0}^{\infty} \frac{\lambda^i}{(j!)^\nu} \tag{A.12}$$

which reduces to the Poisson distribution in the special case $\nu = 1$, i.e if $\tau \in \mathrm{CMP}(\lambda, 1)$ then $\tau \in \mathrm{Poi}(\lambda)$. For the remainder of this section, we aim to further investigate the behavior of parallelism in SGD under the CMP and Poisson models and propose an adaptive step size strategy to reduce the negative impact and improve the statistical efficiency under asynchrony.

In a homogeneous system with m equally powerful worker nodes/threads, we expect that the most frequent staleness observation (the distribution mode) should relate to the number of workers. More precisely, since a sequential execution would always have $\tau = 0$, an appropriate choice of τ distribution should have the mode $m - 1$. For the CMP distribution, we have that if $\tau \in \mathrm{CMP}(\lambda, \nu)$ then the mode of τ is $\lfloor \lambda^{1/\nu} \rfloor$, and we therefore hypothesize the following relation:

$$\lambda^{1/\nu} = m \tag{A.13}$$

For the special case $\nu = 1$, i.e., a Poisson τ model, (A.13) enables us to immediately choose an appropriate value for λ given the number of workers m. In general, (A.13) simplifies the parameter search when fitting a CMP distribution model to a one-dimensional line search, which is in practice a significant complexity reduction.

τ-adaptive η. In the following, we argue analytically about how to choose an adaptive step size function η for reducing the negative impact of stale gradients. We will see how a certain τ-adaptive step size can bound the magnitude of $\Sigma_{p,\eta}^{\nabla}$ (A.7), and even tune the implicit asynchrony-induced momentum to any desired value.

Theorem A.4.5. *Assume $\tau \in CMP(\lambda, \nu)$, and let the adaptive step size function be defined as follows:*

$$\eta(\tau) = C\lambda^{-\tau}(\tau!)^\nu \eta \tag{A.14}$$

for any constant C. Then we have $\Sigma_{p,\eta}^{\nabla} = 0$.

Proof. We have

$$\Sigma_{p,\eta}^{\nabla} = \sum_{j=0}^{\infty} \big(p(j)\eta(j) - p(j+1)\eta(j+1)\big)\nabla L(\theta_{i-j-1})$$

Substituting $p(j)$ for the CMP PDF (A.12) gives

$$\Sigma_{p,\eta}^{\nabla} = \frac{1}{Z(\lambda,\nu)} \sum_{j=0}^{\infty} \frac{\lambda^j}{(j!)^{\nu}} \left(\eta(j) - \lambda \frac{\eta(j+1)}{(j+1)^{\nu}} \right) \nabla L(\theta_{i-j-1}) \tag{A.15}$$

Now, applying the adaptive step size (A.14) gives

$$\begin{aligned}
\Sigma_{p,\eta}^{\nabla} &= \frac{C}{Z(\lambda,\nu)} \sum_{j=0}^{\infty} \frac{\lambda^j}{(j!)^{\nu}} \eta \Big(\lambda^{-j}(j!)^{\nu} - \\
&\quad \frac{\lambda}{(j+1)^{\nu}} \lambda^{-(j+1)} ((j+1)!)^{\nu} \Big) \nabla L(\theta_{i-j-1}) \\
&= \frac{C}{Z(\lambda,\nu)} \sum_{j=0}^{\infty} \frac{\lambda^j}{(j!)^{\nu}} \eta \left(\frac{(j!)^{\nu}}{\lambda^j} - \frac{(j!)^{\nu}}{\lambda^j} \right) \nabla L(\theta_{i-j-1}) = 0
\end{aligned}$$

$\square$

Theorem A.4.5 shows how a simple and tunable τ-adaptive step size mitigates the $\Sigma_{p,\eta}^{\nabla}$ quantity.

However, from Lemma A.4.1, we see that even though $\Sigma_{p,\eta}^{\nabla}$ is mitigated by the adaptive step size (A.14), the SGD steps still have a fixed implicit momentum term of magnitude $\mu = 1$. We show in Theorem A.4.6 how the implicit momentum can be tuned to any desired value through a particular choice of $\eta(\tau)$.

Theorem A.4.6. *Assume $\tau \in CMP(\lambda,\nu)$. Then, $\Sigma_{p,\eta}^{\nabla}$, in expectation, takes the form of asynchrony-induced momentum of magnitude exactly K, i.e.*

$$\Sigma_{p,\eta}^{\nabla} = K\mathbf{E}[\theta_i - \theta_{i-1}]$$

when using the adaptive step size function:

$$\eta(\tau) = c(\tau) \lambda^{-\tau} (\tau!)^{\nu} \eta \tag{A.16}$$

where

$$c(\tau) = 1 - \frac{K}{\eta e^{\lambda}} \sum_{j=0}^{\tau-1} \frac{\lambda^j}{(j!)^{\nu}} \tag{A.17}$$

Proof. Let $\Psi(j) = \eta(j) - \lambda \frac{\eta(j+1)}{(j+1)^{\nu}}$, and hence

$$\Sigma_{p,\eta}^{\nabla} = \frac{1}{Z(\lambda,\nu)} \sum_{j=0}^{\infty} \frac{\lambda^j}{(j!)^{\nu}} \Psi(j) \nabla L(\theta_{i-j-1})$$

Applying the adaptive step size (A.16) gives

$$\Psi(j) = \frac{j!^{\nu}}{\lambda^j} e^{\lambda} \eta \left(c(j) - c(j+1) \right)$$

Now,

$$\Psi(j) = K \Leftrightarrow c(j) - c(j+1) = \frac{K}{\eta e^\lambda} \frac{\lambda^j}{j!^\nu}$$

$$\Leftrightarrow c(j) = c(j-1) - \frac{K}{\eta e^\lambda} \frac{\lambda^{j-1}}{(j-1)!^\nu}$$

$$= c(0) - \frac{K}{\eta e^\lambda} \sum_{k=1}^{j} \frac{\lambda^{j-k}}{(j-k)!^\nu} = c(0) - \frac{K}{\eta e^\lambda} \sum_{k=1}^{j} \frac{\lambda^k}{(k)!^\nu}$$

Since $\eta(0) = \eta$, we have $c(0) = 1$. Now we have

$$\Sigma_{p,\eta}^{\nabla} = K \sum_{j=0}^{\infty} \frac{1}{Z(\lambda, \nu)} \frac{\lambda^i}{(i!)^\nu} \nabla L(\theta_{i-j-1})$$

$$= K \mathbf{E} \left[\nabla L(v_{i-1}) \right] = K \mathbf{E} \left[\theta_i - \theta_{i-1} \right]$$

$\square$

Theorem A.4.6 shows how the series term $\Sigma_{p,\eta}^{\nabla}$ can take the form of momentum of desired magnitude by using a particular τ-adaptive step size. The $c(\tau)$ contains a sum that is $\mathcal{O}(\tau)$ in computation time. This indicates that such an adaptive step size function might not scale well, since τ is expected to be in the magnitude of m. In the following Corollary we show how this is resolved by the corresponding $\eta(\tau)$ under the Poisson τ-model.

Corollary A.4.7. *Assuming $\tau \in Pois(\lambda)$, the series term $\Sigma_{p,\eta}^{\nabla}$ takes the form of implicit momentum of magnitude K when using the adaptive step size:*

$$\eta(\tau) = \left(1 - \frac{K}{\eta} \frac{\Gamma(\tau, \lambda)}{\Gamma(\tau)} \right) \lambda^{-\tau} \tau! \eta \tag{A.18}$$

where $\Gamma(\cdot)$ and $\Gamma(\cdot, \cdot)$ are the Gamma and Upper Incomplete Gamma function, respectively.

Proof. Under the Poisson τ model, i.e., CMP with $\nu = 1$, (A.17) rewrites to:

$$c(i) = 1 - \frac{K}{\eta e^\lambda} \sum_{j=0}^{\tau-1} \frac{\lambda^j}{(j!)} = 1 - \frac{K}{\eta} \frac{\Gamma(i, \lambda)}{(i-1)!}$$

$$= 1 - \frac{K}{\eta} \frac{\Gamma(i, \lambda)}{\Gamma(i)}$$

$\square$

Corollary A.4.7 shows how the series $\Sigma_{p,\eta}^{\nabla}$ is in expectation replaced by momentum of any desired magnitude. Note that there exist efficient ($\mathcal{O}(1)$) and accurate numerical approximation methods for the Gamma and Upper Incomplete Gamma function [68].

A.5 Convex convergence analysis

In this section we analyze the convergence time of *MindTheStep-AsyncSGD*-type algorithms for convex and smooth optimization problems.

Consider the optimization problem (A.3) where an acceptable solution θ^* satisfies ϵ-convergence, defined as $\|\theta - \theta^*\|^2 \leq \epsilon$.

We assume that the problem is addressed using *MindTheStep-AsyncSGD* under the system model described in Section A.2. Note that we consider a staleness-adaptive step size, hence $\eta_i = \eta(\tau_i)$ is stochastic.

For the analysis in this section, we consider strong convexity and smoothness, specified in *Assumption A.5.1*. These analytical requirements are common in convergence analysis for convex problems [44] [53] [45] [69].

Assumption A.5.1. We assume that the objective function L is, in expectation with respect to the stochastic gradients, strongly convex with parameter $\mathcal{C}$ with $\mathcal{L}$-Lipschitz continuous gradients and that the second momentum of the stochastic gradient is upper bounded.

$$\mathbf{E}\left[(\theta^1 - \theta^2)^T\left(\nabla f(\theta^1) - \nabla f(\theta^2)\right) \mid \theta^1, \theta^2\right] \geq \mathcal{C}\|\theta^1 - \theta^2\|^2 \tag{A.19}$$

$$\mathbf{E}\left[\|\nabla F(\theta^1) - \nabla F(\theta^2)\|\| \mid \theta^1, \theta^2\right] \leq \mathcal{L}\|\theta^1 - \theta^2\| \tag{A.20}$$

$$\mathbf{E}\left[\|\nabla F(\theta)\|^2 \mid \theta\right] \leq \mathcal{M}^2 \tag{A.21}$$

The assumption (A.19) is standard in convex optimization and ensures that gradient-based methods will converge to a global optimum. Lipschitz continuity (A.20) is a type of strong continuity which bounds the rate with which the gradients can vary. Due to that $\mathbf{E}[\nabla L(\theta^*)] = 0$, (A.21) can be interpreted as bounding the variance of the gradient norm around the optimum θ^*.

In addition to our system model in Section A.2, we make the following assumption on the staleness process:

Assumption A.5.2. The staleness process (τ_i) is non-anticipative, i.e., mean-independent of the outcome of future states of the algorithm (e.g. future delays and gradients). In particular, we have:

$$\mathbf{E}[\tau_i \mid \tau_j] = \mathbf{E}[\tau_i] \text{ for all } i < j$$

Assumption A.5.2 is justifiable considering that the staleness (i.e., scheduler's decisions) at iteration i should not be considered to be influenced by staleness values τ_j of gradients yet to be computed.

Under Assumptions A.5.1 and A.5.2 above, we give a general bound on the number of iterations sufficient for expected ϵ-convergence in the following:

Theorem A.5.3. *Consider the unconstrained convex optimization problem of (A.3). Under Assumptions A.5.1 and A.5.2, for any $\epsilon > 0$, there is a sufficiently large number T of asynchronous SGD updates of the form (A.4) such that:*

$$T \leq \left(2\big(\mathcal{C} - \mathcal{L}\mathcal{M}\epsilon^{-1/2}\mathbf{E}\left[\tau\eta\right]\big)\mathbf{E}\left[\eta\right] - \right.$$
$$\left. \epsilon^{-1}\mathcal{M}^2\mathbf{E}\left[\eta^2\right]\right)^{-1}\ln\left(\|\theta_0 - \theta^*\|^2\epsilon^{-1}\right) \tag{A.22}$$

for which we have $\mathbf{E}[\|\theta_T - \theta^*\|^2] < \epsilon$

Proof.

$$\|\theta_{i+1} - \theta^*\|^2 = \|\theta_i - \eta_i \nabla L(v_i) - \theta^*\|^2$$
$$= \|\theta_i - \theta^*\|^2 + \eta_i^2 \|\nabla L(v_i)\|^2 - 2\eta_i(\theta_i - \theta^*)^T \nabla L(v_i)$$
$$= \|\theta_i - \theta^*\|^2 + \eta_i^2 \|\nabla L(v_i)\|^2 - 2\eta_i(\theta_i - \theta^*)^T \nabla L(\theta_i)$$
$$+ 2\eta_i(\theta_i - \theta^*)^T \big(\nabla L(\theta_i) - \nabla L(v_i)\big)$$

Under expectation, conditioned on the natural filtration $\mathbb{F}_i^X = \big((\tau_j)_{j=0}^i, \big(\nabla L(v_j)\big)_{j=0}^i\big)$ of the *past* of the process, we have

$$\mathbf{E}\big[\|\theta_{i+1} - \theta^*\|^2 \mid \tau_i, \mathbb{F}_{i-1}^X\big] = \|\theta_i - \theta^*\|^2$$
$$- 2\eta_i \mathbf{E}\big[(\theta_i - \theta^*)^T \big(\nabla L(\theta_i) - \nabla L(\theta^*)\big)\big|\mathbb{F}_{i-1}^X\big]$$
$$+ 2\eta_i \mathbf{E}\big[(\theta_i - \theta^*)^T \big(\nabla L(\theta_i) - \nabla L(v_i)\big)\big|\mathbb{F}_{i-1}^X\big]$$

Applying the assumptions (A.19)-(A.21) gives

$$\mathbf{E}\big[\|\theta_{i+1} - \theta^*\|^2\big|\tau_i, \mathbb{F}_{i-1}^X\big] \le \|\theta_i - \theta^*\|^2 + M^2\eta_i^2$$
$$- 2\eta_i c\|\theta_i - \theta^*\|^2 + 2\eta_i L\|\theta_i - \theta^*\|\|\theta_i - v_i\|$$
$$= (1 - 2c\eta_i)\|\theta_i - \theta^*\|^2 + M^2\eta_i^2$$
$$+ 2\eta_i L\|\theta_i - \theta^*\|\|\theta_i - v_i\|$$
$$= (1 - 2c\eta_i)\|\theta_i - \theta^*\|^2 + M^2\eta_i^2$$
$$+ 2\eta_i L\|\theta_i - \theta^*\| \sum_{j=1}^{\tau_i} \theta_{i-j+1} - \theta_{i-j}\|$$
$$\le (1 - 2c\eta_i)\|\theta_i - \theta^*\|^2 + M^2\eta_i^2$$
$$+ 2\eta_i L \sum_{j=1}^{\tau_i} \|\theta_i - \theta^*\|\eta_{i-j}\|\nabla L(v_{i-j})\|$$

The gradient process does not influence the expected delays, so we first consider the expectation conditioned on the gradient process $(\nabla)_0^i := \big(\nabla L(v_j)\big)_{j=0}^i$

$$\mathbf{E}\left[\|\theta_{i+1} - \theta^*\|^2\big|\tau_i, (\nabla)_0^i\right]$$
$$\le (1 - 2c\eta_i)\mathbf{E}\big[\|\theta_i - \theta^*\|^2\big|\tau_i, (\nabla)_0^i\big] + M^2\eta_i^2$$
$$+ 2L\eta_i \sum_{j=1}^{\tau_i} \mathbf{E}\big[\eta_{i-j}\|\theta_i - \theta^*\|\big|\tau_i, (\nabla)_0^i\big]\|\nabla L(v_{i-j})\|$$

From the non-anticipativity of the delay process we have

$$\mathbf{E}\left[\eta_{i-j}\|\theta_i - \theta^*\| \mid \tau_i, (\nabla)_0^i\right]$$
$$= \mathbf{E}\left[\mathbf{E}\left[\eta_{i-j}\|\theta_i - \theta^*\| \mid \theta_i\right] \mid \tau_i, (\nabla)_0^i\right]$$
$$= \mathbf{E}\left[\|\theta_i - \theta^*\|\mathbf{E}\left[\eta_{i-j} \mid \theta_i\right] \mid \tau_i, (\nabla)_0^i\right]$$
$$= \mathbf{E}\left[\eta_{i-j}\right]\mathbf{E}\left[\|\theta_i - \theta^*\| \mid (\nabla)_0^i\right]$$

Since the delays and gradients are identically distributed, we have $\mathbf{E}[\eta_i] = \mathbf{E}[\eta_j]$ for all i, j. Taking expectation conditioned on the last delay τ_i and applying Hölder's inequality gives

$$\mathbf{E}\left[\|\theta_{i+1} - \theta^*\|^2 \mid \tau_i\right] \leq (1 - 2c\eta_i)\mathbf{E}\left[\|\theta_i - \theta^*\|^2\right] + M^2\eta_i^2$$
$$+ 2L\tau_i\eta_i\mathbf{E}\left[\eta_i\right]\sqrt{\mathbf{E}\left[\|\theta_i - \theta^*\|^2\right]}\sqrt{\mathbf{E}\left[\|\nabla L(v_i)\|^2\right]}$$

and the full expectation satisfies

$$\mathbf{E}\left[\|\theta_{i+1} - \theta^*\|^2\right] \leq (1 - 2c\mathbf{E}\left[\eta_i\right])\mathbf{E}\left[\|\theta_i - \theta^*\|^2\right]$$
$$+ M^2\mathbf{E}\left[\eta_i^2\right] + 2LM\mathbf{E}\left[\tau_i\eta_i\right]\mathbf{E}\left[\eta_i\right]\sqrt{\mathbf{E}\left[\|\theta_i - \theta^*\|^2\right]}$$

As long as the process has not converged, i.e., $\mathbf{E}\left[\|\theta_i - \theta^*\|^2\right] > \epsilon$, we have

$$\mathbf{E}\left[\|\theta_{i+1} - \theta^*\|^2\right] \leq \mathbf{E}\left[\|\theta_i - \theta^*\|^2\right](1 - 2c\mathbf{E}\left[\eta_i\right]$$
$$+ \epsilon^{-1}M^2\mathbf{E}\left[\eta_i^2\right] + 2LM\epsilon^{1/2}\mathbf{E}\left[\tau_i\eta_i\right]\mathbf{E}\left[\eta_i\right])$$
$$=: \mathbf{E}\left[\|\theta_i - \theta^*\|^2\right](1 - \delta)$$
$$\Rightarrow \mathbf{E}\left[\|\theta_i - \theta^*\|^2\right] \leq \mathbf{E}\left[\|\theta_0 - \theta^*\|^2\right](1 - \delta)^T$$
$$\Rightarrow T \leq -\ln(1 - \delta)^{-1}\ln\frac{\mathbf{E}\left[\|\theta_0 - \theta^*\|^2\right]}{\mathbf{E}\left[\|\theta_i - \theta^*\|^2\right]}$$
$$< \delta^{-1}\ln\left(\mathbf{E}\left[\|\theta_0 - \theta^*\|^2\right]\epsilon^{-1}\right)$$

for any T such that $\mathbf{E}\left[\|\theta_i - \theta^*\|^2\right] > \epsilon$. Equivalently, expected convergence is implied by T exceeding the bound above, which concludes the proof. $\square$

Corollary A.5.4. *Under the same conditions as in Theorem A.5.3, there exists a choice of a step size η such that the convergence time T is in the magnitude of $\mathcal{O}\left(\mathbf{E}\left[\tau\right]\right)$ (remember $\mathbf{E}\left[\tau\right]$ is denoted by $\bar{\tau}$). In particular, letting η be*

$$\eta = \rho\frac{\mathcal{C}\epsilon\mathcal{M}^{-1}}{\mathcal{M} + 2\mathcal{L}\sqrt{\epsilon}\bar{\tau}} \tag{A.23}$$

for a tunable factor $\rho \in (0, 2)$, there exists a T such that

$$T \leq \frac{\mathcal{M} + 2\mathcal{L}\sqrt{\epsilon}\bar{\tau}}{\rho(2 - \rho)\mathcal{C}^2\mathcal{M}^{-1}\epsilon}\ln(\epsilon^{-1}\|\theta_0 - \theta^*\|^2) \tag{A.24}$$

Proof. Let $\rho = \frac{c\epsilon M^{-1}}{M + 2L\sqrt{\epsilon}\bar{\tau}}$. From Theorem A.5.3 we have the improvement factor

$$\delta = 2\left(c - LM\epsilon^{1/2}\mathbf{E}\left[\tau\eta\right]\right)\mathbf{E}\left[\eta\right] - \epsilon^{-1}M^2\mathbf{E}\left[\eta^2\right]$$
$$= 2c\eta - \epsilon^{-1}M\left(M + 2L\sqrt{\epsilon}\bar{\tau}\right)\eta^2$$
$$= c\rho^{-1}\eta(2\rho - \eta)$$

so $\delta > 0$ when $0 < \eta < 2\rho$, and the improvement is maximized for $\rho = 1$. Now, using the choice (A.23) of step size, we have

$$\delta = c\rho^{-1}\theta\rho(2\rho - \theta\rho)$$
$$= \theta(2 - \theta)c\rho$$

Substituting for ρ, the convergence bound of Theorem A.5.3 rewrites to (A.24) $\square$

The results in Theorem A.5.3 and Corollary A.5.4 are related to the results presented in [44] and [45]. The main differences are that in our analysis we tighten the bound with a factor $(2 - \rho)^{-1}$, expand the allowed step size interval, as well as relax the *maximum staleness* assumption and reduce the magnitude of the bound from linear in the *maximum* staleness $\mathcal{O}(\hat{\tau})$ to the *expected* $\mathcal{O}(\bar{\tau})$.

In the following corollary, we give a general bound assuming *any* non-increasing step size function $\eta(\tau)$.

Corollary A.5.5. *Under the same conditions as Theorem A.5.3, let $\eta_i = \eta(\tau_i)$ be a non-increasing function of τ_i. Then we have the following bound on the expected number of iterations until convergence:*

$$
\begin{aligned}
T \leq &\left(2\mathcal{C}\mathbf{E}\left[\eta\right] - \epsilon^{-1}\mathcal{M}\left(\mathcal{M} + 2\mathcal{L}\sqrt{\epsilon}\bar{\tau}\right)\mathbf{E}\left[\eta^2\right]\right)^{-1} \\
&\cdot \ln(\epsilon^{-1}\|\theta_0 - \theta^*\|^2)
\end{aligned}
\tag{A.25}
$$

Proof. Since η_i is a non-increasing function in τ_i we have:

$$
\begin{aligned}
\mathbf{E}\left[\tau_i\eta(\tau_i)\right] &= \mathbf{E}\left[\tau_i\eta(\tau_i)\right] - \mathbf{E}\left[\bar{\tau}\eta(\tau_i)\right] + \mathbf{E}\left[\tau_i\right]\mathbf{E}\left[\eta(\tau_i)\right] \\
&= \mathbf{E}\left[(\tau_i - \bar{\tau})(\eta(\tau_i) - \eta(\bar{\tau}))\right] + \mathbf{E}\left[\tau\right]\mathbf{E}\left[\eta\right] \\
&\leq \mathbf{E}\left[\tau\right]\mathbf{E}\left[\eta\right]
\end{aligned}
$$

Using this property, (A.22) rewrites to (A.25).

$\square$

Corollary A.5.5 describes a general convergence bound for any step size function $\eta(\tau)$ which decays in τ. We see that such step size functions also achieve the asymptotic $\mathcal{O}(\bar{\tau}^{-1})$ bound, as the one for a constant η (A.24).

A.6 Evaluation

In this section we evaluate the results derived in section A.4 in a practical setting. This is achieved by (i) measuring the accuracy and scalability of the proposed τ-models (ii) evaluating the convergence properties of *MindTheStep-AsyncSGD* with an adaptive step size function derived under the CMP/Poisson τ models.
Setup. We apply *MindTheStep-AsyncSGD* for training a 4-layer Convolutional Neural Network (CNN) architecture (see Fig. A.1) on the common image classification benchmark dataset CIFAR10 [70]. The performance of the CNN is measured as the *cross entropy* between the true and the predicted class distribution. The algorithm is evaluated on a setup with a 36-thread Intel Xeon CPU and 64GB memory. The implementation is in Python 2.7 and uses the Python multiprocessor library as well as TensorFlow [67] for gradient computation.
CMP/Poisson τ. We evaluate the τ models (Poisson, CMP) proposed in section A.4 by comparing with the τ distribution observed in practice for different number of workers. We compare our proposed τ models with distributions proposed in other works, namely the geometric τ model [40] and the bounded uniform τ model [53].

The distribution parameters in Table A.1 are found through an exhaustive search where we aim to minimize the Bhattacharyya distance to the τ distribution observed in practice. Note that: (i) For the Poisson τ model, as
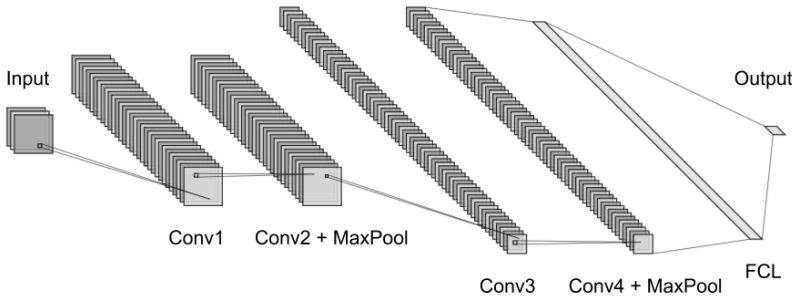
Figure A.1: CNN architecture; Four convolutional layers with 3×3 kernels, with intermediate MaxPool layers. The first two convolutions have 32 filters, the last two 64. The architecture has two fully connected layers, one with 256 neurons, and the output layer with 10 neurons.
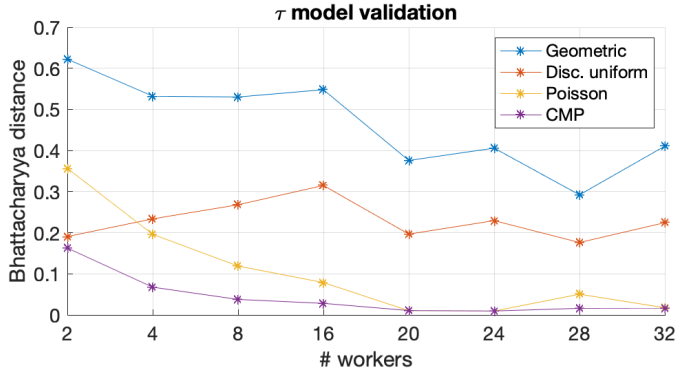

Figure A.2: Bhattacharyya distance of different τ models compared to the observed distribution. The graph shows that the CMP τ model is the most accurate in all tests, with the Poisson τ model as a close second. The uniform and geometric τ models are persistently less accurate and show poor scalability in comparison.

hypothesized in Section A.4, the distribution parameter λ indeed corresponds well to the number of worker nodes. From Fig. A.2 we see that the proposed CMP and Poisson τ models by far outperforms the geometrical and uniform τ models, in particular for larger number of workers. (ii) As mentioned in Footnote 2, we confirm in Table A.1 that $\mathbf{P}[\tau = 0]$, i.e., p, decays as the concurrency level increases. (iii) We see in Fig. A.2 that the CMP τ model outperforms the others in terms of accuracy and scalability. The CMP distribution parameter ν is found through a 1-d search and using the assumption (A.13) the other parameter λ is calculated. The result in Fig. A.2 therefore validates the assumption (A.13).

Convergence with τ-adaptive η.   We evaluate *MindTheStep-AsyncSGD* compared with standard *AsyncSGD* by measuring the number of *epochs* required until a certain error threshold is reached, epochs being the number of passes through the dataset. The number of SGD iterations in one epoch

τ model	2	4	8	16	20	24	28	32
p (Geom)	0.34	0.21	0.12	0.06	0.05	0.04	0.04	0.03
$\hat{\tau}$ (Unif)	2	5	11	22	31	37	48	48
λ (Pois)	2.0	4.0	8.0	16.0	19.7	23.8	26.5	32
ν (CMP)	6.28	5.26	4.18	3.48	0.93	0.95	0.39	0.87

Table A.1: Optimal distribution parameters for different number of workers.
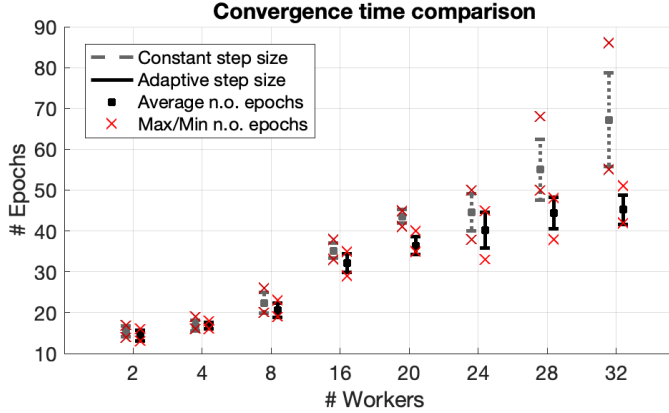

Figure A.3: *AsyncSGD* vs. *MindTheStep-AsyncSGD* comparison. The plot shows the n.o. epochs required until sufficient performance (cross-entropy loss ≤ 0.05). The statistics are computed based on 5 runs, and the bar height corresponds to the standard deviation.

is $\lceil |D|/b \rceil$ where $|D|$ is the size of the dataset and b the batch size. In our experiments we have $\lceil |D|/b \rceil = 469$. We consider performance in terms of *statistical efficiency*, i.e., the statistical benefit of each SGD step. In practice, the approach can be applied to any orthogonal work focusing on computational efficiency, such as efficient parameter server architectures [23] [65] and efficient gradient communication and quantization [59] [66].

We compare standard *AsyncSGD* with constant step size $\eta_c = 0.01$, $b = 128$ to *MindTheStep-AsyncSGD* with an adaptive step size function according to (A.18) with $\eta = \eta_c$, $K = 1$, and $\lambda = m$. In addition, we bound the step size $\eta(\tau) \leq 5 \cdot \eta_c$ to mitigate issues with numerical instability in the SGD algorithms, and (very infrequent) gradients with $\tau > 150$ are not applied.

In principle, given a sufficiently small η_c, speedup can always be achieved by using an adaptive step size strategy $\eta(\tau)$ which overall increases the average step size. To ensure a fair comparison, the adaptive step size function $\eta(\tau)$ is normalized so that:

$$\mathbf{E}_\tau[\eta(\tau)] = \eta_c \qquad (A.26)$$

where the expectation is taken over the real τ distribution observed in the system. Enforcing (A.26) ensures that any potential speedup is achieved due to how the step size function $\eta(\tau)$ adaptively changes the impact of gradients depending on their staleness, and not because of the overall magnitude of the step size.

Fig. A.3 shows how *MindTheStep-AsyncSGD* exhibits persistently faster

convergence for different number of workers. For many workers ($m = 28, 32$) *MindTheStep-AsyncSGD* requires significantly fewer epochs compared to standard *AsyncSGD* to achieve sufficient performance. Observe that for $m = 32$ the average speedup is $\times 1.5$ while the worst-case is $\times 1.7$.

A.7   Related work

Orthogonal to this work, there are numerous works dedicated to optimizing the effectiveness of SGD by utilizing data sparsity, topology of the search space, and other properties of the problems. One example is introducing momentum to the updates, originally proposed in [71], however not in the context of SGD. Apart from this, there are several variations of SGD in the sequential case introducing adaptiveness to aspects of the problem topology, such as Adagrad, Adadelta, RMSprop, Adam, AdaMax, and Nadam (cf. [72] and references therein).

In [40] Mitliagkas et al. show that under certain stochastic delay models, asynchrony has an effect on convergence similar to momentum, referred to as asynchrony-induced or implicit momentum, where more workers imply a larger magnitude of the effect. In [73] these similarities are investigated further, and it is shown that *AsyncSGD* and momentum shows different convergence rates in general and that *AsyncSGD* is in fact faster in expectation. Since it has been seen [15] that the magnitude of momentum can have significant impact on convergence, the result by Mitliagkas et al. would imply a harsh scalability limitation of *AsyncSGD*. In this paper, we show that under the same τ model as in [40], *MindTheStep-AsyncSGD* can in theory mitigate this issue, and even allow the expected asynchrony-induced momentum to be tuned implicitly by the rate of adaptation. In addition, in this work we propose a new class of τ distribution models and show how they better capture the real τ values observed in a deep learning application. From our proposed models we derive an adaptive step size function $\eta(\tau)$ which we show significantly reduces the number of SGD steps required for convergence.

Below we give a brief overview of works on synchronous distributed SGD. Under smoothness and convexity assumptions, in [22] and [74], synchronous distributed SGD with data-parallelism was observed and proven to accelerate convergence. This was implemented on a larger scale by Dekel et al. [75] where the convergence rates were improved under stronger analytical assumptions. In [23] the synchronization is relaxed using a Stale Synchronous Parameter Server with a tunable staleness threshold in order to reduce the waiting-time, which is shown to outperform synchronous SGD. In [27] Gupta et al. give a rigorous empirical investigation of practical trade-offs the number of workers, mini-batch size and staleness; the results provide useful insights in scalability limitations in synchronous methods with averaging. We address this issue in this paper from a theoretical standpoint and explain the results observed in practice. This is discussed in detail in Section A.3.

The study of numerical methods under parallelism is not new and sparked due to the works by Bertsekas and Tsitsiklis [33] in 1989. Recent works [25] [39] show under various analytical assumptions that the convergence of *AsyncSGD* is not significantly affected by asynchrony and that the noise introduced by delays is asymptotically negligible compared to the noise from the stochastic

gradients. This is confirmed in [25] for convex problems (linear and logistic regression) for a small number of cores. In [39] Lian et al. relax the theoretical assumptions and establish convergence rates for non-convex minimization problems, assuming bounded gradient delays and number of workers. Lock-free *AsyncSGD* in shared-memory, i.e. HOGWILD!, was proposed by Niu et al. [20] and was shown to achieve near-optimal convergence rates assuming sparse gradients. Properties of *AsyncSGD* with sparse updates have since been rigorously studied in recent literature due to the performance benefits of lock-freedom [42] [44]. The gradient sparsity assumption was relaxed in the recent work [45] which magnified the convergence time bound in the order of magnitude $\sim \sqrt{d}$, d being the problem dimensionality.

Delayed optimization in completely asynchronous first-order optimization algorithms was analyzed initially in [51], 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. In comparison, in this work we relax the maximum staleness restriction and derive a strategy for adapting the step size depending on the actual staleness values observed in the system in an online fashion. Adaptiveness to delayed updates during execution was proposed and analyzed in [52] under assumptions of gradient sparsity and *read* and *write* operations having the same relative ordering. A similar approach was used in [76], however for synchronous SGD with the *softsync* protocol. In [24] speedup in statistical efficiency 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.

The work closest to ours is AdaDelay [53] 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 τ^{-1}. The staleness model in [53] is a uniform stochastic distribution, which implies a strict upper bound on the delays, making the system partially asynchronous. In comparison, in this work we analyze the convergence of *MindTheStep-AsyncSGD* for non-convex optimization, relax the bounded gradient staleness assumption, as well as evaluate more delay models both theoretically and empirically. Moreover, we validate our findings experimentally by training a Deep Neural Network (DNN) classifier using real-world dataset, which constitutes a highly non-convex and high-dimensional optimization problem. In addition, we provide convergence analysis in the convex case for *MindTheStep-AsyncSGD*, where we show explicitly a probabilistic time bound for ϵ-convergence, for any step size function decaying in the staleness τ.

A.8   Conclusions

In this paper, we first analytically confirm scalability limitations of the standard *SyncSGD*, which were observed empirically in other works; we thus motivate the need to further investigate asynchronous approaches. We propose a new class of τ-distribution models, show analytically how the parameters can be efficiently chosen in a practical setting, and validate the models empirically, as well as compare them to models proposed in other works.

We derive and analyze adaptive step size strategies which reduce the impact of asynchrony and stale gradients, using our framework *MindTheStep-AsyncSGD*. We show that the proposed strategies enable turning asynchrony into implicit asynchrony-induced momentum of desired magnitude. We provide convergence bounds for a wide class of τ-adaptive step size strategies for convex target functions. We validate our findings empirically for a deep learning application and show that *MindTheStep-AsyncSGD* with our proposed step size strategy converges significantly faster compared to standard *AsyncSGD*.

The concept of staleness-adaptive *AsyncSGD* has been under-explored, despite the fact that, as shown here, it significantly improves scalability and helps maintain statistical efficiency. Continuing to investigate asynchrony-aware SGD is therefore of interest. Future research directions also include further studying the nature of staleness, i.e., effect of schedulers and synchronization methods, for understanding the impact of asynchrony and for choosing appropriate adaptive strategies.

Chapter B

Consistent Lock-free Parallel Stochastic Gradient Descent for Fast and Stable Convergence

K. Bäckström, I. Walulya, M. Papatriantafilou, P. Tsigas

Abstract

Stochastic gradient descent (SGD) is an essential element in Machine Learning (ML) algorithms. Asynchronous parallel shared-memory SGD (*AsyncSGD*), including synchronization-free algorithms, e.g., HOGWILD!, have received interest in certain contexts, due to reduced overhead compared to synchronous parallelization. Despite the fact that they induce staleness and inconsistency, they have shown speedup for problems satisfying smooth, strongly convex targets, and gradient sparsity. Recent works take important steps towards understanding the potential of parallel SGD for problems not conforming to these strong assumptions, in particular for *deep learning* (DL). There is however a gap in current literature in understanding when *AsyncSGD* algorithms are useful in practice, and in particular how mechanisms for synchronization and consistency play a role.

We contribute by answering questions in this gap by studying a spectrum of parallel algorithmic implementations of *AsyncSGD*, aiming to understand how shared-data synchronization influences the convergence properties in fundamental DL applications. We focus on the impact of consistency-preserving non-blocking synchronization in SGD convergence, and in sensitivity to hyper-parameter tuning. We propose *Leashed-SGD*, an extensible algorithmic framework of consistency-preserving implementations of *AsyncSGD*, employing lock-free synchronization, effectively balancing throughput and latency. *Leashed-SGD* features a natural contention-regulating mechanism, as well as dynamic memory management, allocating space only when needed. We argue analytically about the dynamics of the algorithms, memory consumption, the threads' progress over time, and the expected contention. The analysis further shows the contention-regulating mechanism that *Leashed-SGD* enables.

We provide a comprehensive empirical evaluation, validating the analytical claims, benchmarking the proposed *Leashed-SGD* framework, and comparing to baselines for two prominent *deep learning* (DL) applications: *multilayer perceptrons* (MLP) and *convolutional neural networks* (CNN). We observe the crucial impact of *contention*, *staleness* and *consistency* and show how, thanks to the aforementioned properties, *Leashed-SGD* provides significant improvements in stability as well as wall-clock time to convergence (from 20-80% up to $4\times$ improvements) compared to the standard lock-based *AsyncSGD* algorithm and HOGWILD!, while reducing the overall memory footprint.

B.1   Introduction

The interest in Machine Learning (ML) methods for data analytics has peaked in the last decade due to their tremendous impact across various applications. Parallel algorithms for ML, utilizing modern computing infrastructure, have gained particular interest, showing high scalability potential, necessary in accommodating significant growing data demands as well as data availability. Parallelization schemes for Stochastic Gradient Descent (SGD) have been of particular interest, since SGD serves as a backbone in many widely used ML algorithms and has proven effective on convex problems (e.g., linear, logistic regression, SVM), as well as non-convex (e.g., matrix completion, deep learning).

  The first-order iterative minimizer SGD follows the simple rule (B.1) of moving in the direction of the negative stochastic gradient $\widetilde{\nabla} f$ with a step size η, of a differentiable target function $f : \mathbb{R}^d \to \mathbb{R}$, quantifying the error of a ML model:

$$\theta_{t+i} = \theta_i - \eta \widetilde{\nabla} f(\theta_i) \tag{B.1}$$

where θ_i contains the *learned* parameters of the model at iteration i, typically encoding features of a given dataset. Iterations, calculating over *batches* of one or multiple data samples each, typically repeat until ϵ-*convergence*, i.e., reaching a sufficiently low error threshold ϵ. As in SGD each update relies on the outcome of the previous one, data parallelization is challenging. Still, several approaches have been proposed, distinguished into *synchronous* and *asynchronous* ones:
Synchronous SGD (*SyncSGD*) is a lock-step parallelization scheme where the gradient computation is delegated to threads/nodes, then aggregated by averaging before taking a global step according to eq. (B.1) [22]. In its original form, *SyncSGD* is statistically equivalent to sequential SGD with larger *data-batch* [27], as also established in **Chapter A**. This method is well-understood and widely used, e.g., in *federated learning* [26]. However, its scalability suffers as every step is limited by the slowest contributing thread. In addition, higher parallelism implies an impact on the convergence, inherent to *large-batch* training [31]. Semi-synchronous variants have shown improvements [18, 77], relaxing lock-step semantics and requiring only a subset of threads to synchronize, hence reducing waiting. In the recent [18] it was seen that requiring only a few, even just one, thread at synchronization, implies significant speedup due to less waiting and higher throughput, motivating further study of *asynchronous* parallel SGD.
Asynchronous SGD (*AsyncSGD*) on the other hand employs parallelism on SGD algorithm level, allowing threads to execute (B.1) on a shared vector θ with less coordination, and has shown superior speedup compared to *SyncSGD* in several applications [20, 78]. It was first introduced for distributed optimization with a parameter server sequentializing the updates. In this context it was proven that the algorithm converges for convex problems [51] despite the presence of noise due to stale updates. A relaxed variant, HOGWILD! [20], allowing completely uncoordinated component-wise reads and updates in θ, showed substantial speedup, however only on smooth convex problems with sparse gradients. This, besides *staleness*, also introduces *inconsistency* incurred by non-coordinated concurrent reads and writes on θ, penalizing the statistical efficiency. Only if parallelization gains counterbalance the latter penalty, will there be an actual improvement in the wall-clock time for convergence.

Challenges. There are substantial analytical results and empirical evidence that *AsyncSGD* [20, 25, 51, 79] provides speedup for problems satisfying varying assumptions on convexity, strong convexity, smoothness and sparsity assumptions, e.g., Logistic regression, Matrix completion, Graph cuts and SVM training. Recently, a target of study is parallelism in SGD for wider class of more unstructured problems, not conforming to strict analytical assumptions, such as Artificial Neural Network (ANN) training, or Deep Learning (DL) in general. Recent works [28, 80] explore aspects of data-parallelism in the context of distributed and parallel SGD for DL. However, for empirical results using abstraction libraries, such as TensorFlow and Keras, in Python implementations, with its inherent limitations in parallelism and performance, makes time measurements unreliable. As a consequence, the existing literature addresses the topic mostly from an analytical standpoint, and empirical convergence rates are almost exclusively measured in *statistical efficiency*, i.e., n.o. iterations, as opposed to actual *wall-clock time*. With new methods that potentially affect the *computational efficiency*, i.e., time per iteration, such results can be delusive, with unclear usefulness in practice. Moreover, such implementations have limited capability of fine-grained exploration of aspects of synchronization mechanisms and consistency, the critical impact of which on the convergence properties has been observed analytically; (i) It was shown in [44] that the number of iterations until convergence increases linearly in the magnitude of the maximum staleness and (ii) in [45] that inconsistency due to HOGWILD!-style updates further increases the same bound with a factor of $\sqrt{d}$, d being the size of θ. There is a need for further exploration of how synchronization, lock-freedom and consistency impact the *actual* wall-clock time to convergence, to facilitate work in development of standardized platforms for accelerated DL.

For DL applications, convergence of sufficient quality is challenging to achieve, requiring exhaustive neural architecture searches and careful tuning of many *hyper-parameters*. Unsuccessful such tuning typically results in models never converging to sufficient quality, or even executions which crash due to numerical instability in the SGD steps [81]. The step size η is among the most important hyper-parameters, while data-batch size, momentum, dropout, also play a significant role. Tuning is vital for the convergence and end performance and is a time-consuming process. On one hand, parallelism in SGD is crucial for speedup, but it introduces new hyper-parameters to tune, such as number of threads, staleness bound and aspects of synchronization protocol. In addition, *AsyncSGD* introduces noise due to staleness, further impacting convergence and potentially causing unsuccessful executions. There is hence a need for methods enabling speedup by parallelism tolerant to existing parameters and avoiding the overhead of tuning additional ones related to parallelism.

Focal point and contributions In summary, there are challenges in understanding the dynamics of asynchrony and consistency on the SGD convergence [46] in practice as outlined in Fig. B.1, in particular for applications as DL. Understanding better the tradeoff between computational and statistical efficiency is a core issue [16]. It is known that consistency helps in *AsyncSGD* [45]. However, whether it is worth the overhead to ensure consistency with locks or other synchronization means, to improve the overall convergence, is a research question attracting significant attention, as we describe here and in the related work section.
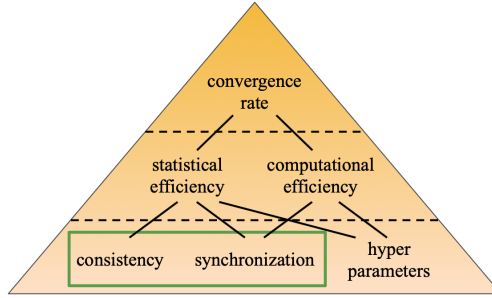
Figure B.1: Convergence rate is the product of computational and statistical efficiency, sensitive to hyper-parameter tuning. We show the significant impact of lock-free synchronization on these factors and on reducing the dependency on tuning, enabling improved convergence.

We study asynchronous SGD in a practical setting for DL. In a system-level environment, we explore aspects of synchronization, lock-freedom and consistency, and their impact on the overall convergence. In more detail, we make the following contributions:

- We propose *Leashed-SGD* (*lock-free* consistent asynchronous shared-memory SGD), an extensible algorithmic framework for lock-free implementations of *AsyncSGD*, allowing diverse mechanisms for consistency and for regulating contention, with efficient dynamic memory allocation and recycling.

- We analyze the proposed framework *Leashed-SGD* in terms of safety, memory consumption and we introduce a model for estimating thread progression and balance in the *Leashed-SGD* execution, estimating contention over time and the impact of the contention-regulation mechanism.

- We perform a comprehensive empirical study of the impact of synchronization, lock-freedom, and consistency on the convergence in asynchronous shared-memory parallel SGD. We extensively evaluate *Leashed-SGD*, the standard lock-based *AsyncSGD* and its synchronization-free counterpart HOGWILD! on two DL applications, namely *Multi-Layer Perceptrons* (MLP) and *Convolutional Neural Networks* (CNN) for image classification on the MNIST dataset. We study the dynamics of contention, staleness and consistency under varying parallelism levels, confirming also the analytical observations, focusing on the *wall-clock time* to convergence.

- We introduce a C++ framework supporting implementation of shared-memory parallel SGD with different mechanisms for synchronization and consistency. A key component is the PARAMETERVECTOR data structure, providing an abstraction of common operations on high-dimensional model parameters in ANN training, providing a modularization facilitating further exploration of aspects of parallelism.

The chapter is structured as follows: In section B.2 we outline preliminaries and key notions for describing *Leashed-SGD*, as well as its contention and staleness dynamics in sections B.3 and B.4. The comprehensive empirical study

is presented in B.5, followed by further discussion of related work in section B.6, after which we conclude in section B.7.

B.2 Preliminaries

Here we give a brief background, along with a more refined description, for the questions and the metrics in focus.

Optimization problem. We target the optimization problem of (1.2), in the same context as in Section 1.2, where the aim is to find θ, representing parameters of an ANN (see Section 1.2.2), that minimize a loss function $L\colon \mathbb{R}^d \to \mathbb{R}^+$. The collective set of ANN parameters θ is referred to as the *parameter vector* and defines the associated AI model, and the loss function L quantifies the error of θ. The metrics of interest are (i) statistical and (ii) computational efficiency, as well as the (iii) overall convergence rate, as defined in 1.2.3.

System model. We consider a system with m concurrent asynchronous threads, with access to shared memory through atomic operations to read, write and read-modify-write, e.g., CompareAndSwap (CAS), FetchAndAdd (FAA) [21] on single-word locations. Each thread A computes SGD updates (B.1) according to a pre-defined algorithm, in the context outlined in the previous paragraphs. Since A must read the current state θ_i prior to computing the corresponding stochastic gradient $\nabla L(\theta_i)$, before A's updates take place, there can be intermediate, referred to as *concurrent updates*, from other threads, The number of such updates, between A's read of the θ_i vector and A's update to apply its calculated gradient $\nabla L(\theta_i)$, defines the *staleness* τ of the latter update. When there is lack of synchronization, as in HOGWILD!, a total order of the updates is not imposed, and the definition of the staleness of an update is not straightforward; we adopt a definition similar to [45]. We refer to Section (B.3) for details on how the staleness is calculated for the different algorithms, and thereby the total order of the updates. Under the system model above, we have that the asynchronous SGD updates according to (B.1) instead will follow

$$\theta_{i+1} \leftarrow \theta_i - \eta \nabla L(v_i) \tag{B.2}$$

where $v_i = \theta_{i-\tau_i}$ is the thread's *view* of θ.

Synchronization methods and consistency. For consistency on concurrently accessed data, different methods for thread synchronization exist, the most traditional one being *locks* for mutually exclusive access. *Non-blocking synchronization* avoids the use of locks. [21]. A common choice is *lock-free synchronization*, ensuring that in the presence of concurrent object accesses, some are able to complete in a bounded number of steps, thus guaranteeing system progress. Such synchronization mechanisms usually implement a *retry loop* involving CAS or equivalent, in which a thread might need to repeat, in case another thread has succeeded.

Besides *progress* guarantees, to argue about concurrent data accesses, we consider *data consistency*. The most common is *atomicity* (aka *linearizability*, with non-blocking synchronization), and it implies that concurrent object operations act as if they are executed in sequence, affecting state and returning values according to the object's sequential specification [21].

Problem overview.   In the following, we focus on exploring the effectiveness of asynchronous parallel algorithms for SGD, for training Deep Neural Networks (DNNs). We study the computational and statistical efficiency for different applications, and the overall time to ϵ-convergence. We explore in particular the effect of different synchronization mechanisms on consistency, contention and staleness, and the resulting impact on the convergence and memory consumption.

B.3   The *Leashed-SGD* framework

In the following we define *Leashed-SGD* along with the proposed PARAME-TERVECTOR data structure's common interface, containing the values of the parameter vector, as well as metadata used for memory recycling. We also express *AsyncSGD* and HOGWILD! using this interface; both are well established versions of parallel SGD implementations [20, 51]. Modified versions, optimized for specific applications, have been proposed, e.g., in [82], however not in the context of DL.   In the following, we use them as general baselines, representative of the classes of *consistent* asynchronous SGD algorithms and the *synchronization-free, inconsistent* HOGWILD!-style ones.

B.3.1   Introducing PARAMETERVECTOR object

Considering (B.1), each worker in parallel SGD reads the shared data object θ, computes a gradient and updates the former. We propose a set of core components for this type of data structure, PARAMETERVECTOR, providing possibilities to get parameter values and submit updates. An instantiation of PARAME-TERVECTOR can be local or shared among threads. For concurrent access to it, its implementation can provide certain consistency and progress guarantees (cf. section B.2). Hence studying shared memory data-parallel SGD implementations with synchronization in focus, is to study implications of the properties of the algorithmic implementations of the parameter vector seen as shared object.

Algorithm B.1 PARAMETERVECTOR core components

Float(d) θ ▷ Array of dimension d
Int $i \leftarrow 0$ ▷ Sequence number of the most recent update of θ
Int n_rdrs $\leftarrow 0$
Bool stale_flag $\leftarrow false$, deleted $\leftarrow false$

procedure RAND_INIT()
   $\theta \leftarrow \mathcal{N}(0, 0.01)$
procedure SAFE_DELETE()
   if stale_flag $\wedge\, n\_rdrs = 0 \wedge$ CAS($deleted, false, true$) **then**
     delete θ
procedure START_READING()
   n_rdrs.fetch_add(1)
procedure STOP_READING()
   n_rdrs.fetch_add(-1)
   $self$.SAFE_DELETE()
procedure UPDATE(δ, η)
   i.fetch_add(1)
   for $d' = 0, \ldots, d-1$ **do**
     $\theta[d'] \leftarrow \theta[d'] - \eta \cdot \delta[d']$

Algorithm B.1 describes the core components for the algorithmic implementation of PARAMETERVECTOR. A main one is the array θ of dimension

d (typically a very large number in DL applications, e.g., in the well-known AlexNet [83] CNN architecture there are 62,378,344 parameters). A read of the parameters can be accomplished by getting a pointer to θ, while function UPDATE(, p)erforms the addition (B.2) on θ. Notice that algorithm B.1 does not provide specific synchronization for protecting reads of updates, which is instead left to the algorithmic implementation's *"front-end"* to specify, depending on the demands of consistency. It provides however additional methods and metadata for keeping track of accesses and for recycling memory, as explained further in this section. While there is some resemblance with a multi-word register [84,85], two significant issues here are (i) the nature of the update, which is a bulk Read-Modify-Write operation and (ii) the very large value of d, posing challenges both from the memory and from the timing (retry loop size) perspectives.

Algorithm B.2 *AsyncSGD*

 1: GLOBAL PARAMETERVECTOR PARAM
 2: GLOBAL Float η ▷ Step size
 3: GLOBAL Lock MTX   ▷ For accessing shared parameters
 4: Initialization
 5: PARAM ← new PARAMETERVECTOR
 6: PARAM.RAND_INIT()   ▷ Randomly initialize parameters
 7: Each thread
 8: local_grad ← new PARAMETERVECTOR   ▷ Local gradient memory
 9: local_param ← new PARAMETERVECTOR
10: **repeat**
11:    MTX.lock()
12:    local_param.θ = copy(PARAM.θ)
13:    MTX.unlock()
14:    local_grad.θ ← comp_rand_grad(local_param.θ)
15:    MTX.lock()
16:    PARAM.UPDATE(local_grad.θ, η)
17:    MTX.unlock()
18: **until** convergence

B.3.2 Baselines expressed using PARAMETERVECTOR

Algorithm B.2 shows the lock-based *AsyncSGD*, one of the baselines, achieving consistency in the reads and the updates of the parameters through locking. This introduces an overhead, influencing the thread interleaving, with unclear implications on staleness and statistical efficiency. This is further explored in Section B.5. There is one shared variable of type PARAMETERVECTOR, PARAM, and two local ones to each thread, one with a copy of the latest state of the shared parameter vector (*local_param*) and one for storing the gradient (*local_grad*). HOGWILD!'s algorithmic implementation is similar to Algorithm B.2, except that the locks are removed, since no synchronization happens among the threads accessing the parameter vector. Certain overhead is thus eliminated, however at the cost of inconsistency in the parameter updates. For problems with sparse gradients the lack of synchronization will not significantly impact the convergence, since the UPDATE($\cdot, \cdot$) operation will only influence a few of the d components in θ. For DL applications though, its influence is not well understood.

B.3.3   *Leashed-SGD*: lock-free consistent *AsyncSGD*

Algorithm B.3 *Leashed-SGD*

1: GLOBAL PARAMETERVECTOR ** P ▷ Address to latest pointer (cf. Fig. B.2)
2: GLOBAL Float η ▷ Step size
3: GLOBAL Int T_p ▷ Persistence threshold
4: **function** LATEST_POINTER()
5:    **repeat**
6:        latest_param ← *P ▷ Fetch latest pointer
7:        latest_param.START_READING()   ▷ Prevent it from being recycled
8:        **if** ¬ latest_param.stale_flag **then**
9:            **return** latest_param
10:       **else**
11:           latest_param.STOP_READING()   ▷ Avoid returning stale vector, let it be
12:    **until** break

13: <u>Initialization</u>
14: init_pv ← new PARAMETERVECTOR()   ▷ Pointer to initial parameters
15: init_pv.RAND_INIT()   ▷ Randomly initialize parameters
16: P ← &init_pv   ▷ Address of initial pointer
17: <u>Thread w</u>
18: *local_grad* ← new PARAMETERVECTOR   ▷ Local gradient memory
19: **repeat**
20:    latest_param ← LATEST_POINTER()
21:    local_grad ← comp_grad(latest_param)   ▷ Allocate new memory
                                                        and compute gradient
22:    latest_param.STOP_READING()
23:    new_param ← new PARAMETERVECTOR()   ▷ New parameters
24:    Int num_tries ← 0   ▷ Prepare for *LAU-SPC*
25:    **repeat**
26:        latest_param ← LATEST_POINTER()
27:        new_param.θ ← copy(latest_param.θ)
28:        new_param.i ← copy(latest_param.i)
29:        latest_param.STOP_READING()
30:        new_param.UPDATE(local_grad.θ, η)
31:        succ ← CAS(P, latest_param, new_param)
32:        **if** succ **then**
33:            latest_param.stale_flag ← true
34:            latest_param.SAFE_DELETE()
35:        **else**
36:            num_tries ← num_tries + 1
37:            **if** num_tries > T_p **then**
38:                **delete** new_param
39:                **break**
40:    **until** succ
41: **until** convergence

The key points and arguments supporting *Leashed-SGD*, which is shown in pseudocode in Algorithm B.3, using PARAMETERVECTOR core components from Algorithm B.1, are as follows:

P1. *Local calculation and sharing of new parameter values:* Each thread manages its update locally *new_param* and attempts to publish the result in a single atomic CAS operation (line 31), switching a global pointer P to point to its new instance (Fig. B.2). As a successful CAS replaces the previous "global" vector, copies of parameter vectors that become global are *totally ordered* on their sequence number, i. A vector that has been replaced using the aforementioned CAS, is labeled as *stale* through a boolean flag (*stale_flag* in PARAMETERVECTOR) that is one of the data structure's fields.

P2. *Memory recycling:* Since a new PARAMETERVECTOR is needed for each such update, a simple yet efficient *recycling* mechanism of *stale and unusable* ones ensures that the memory used is bounded. Besides the label for marking a PARAMETERVECTOR instance as stale (ensuring no new readers, making it a candidate for recycling), the field n_rdrs, indicates whether the PARAME-

TERVECTOR should persist due to active readers.

P3. *Lock-free atomic reads of the shared vector:* To access the global PARAM-ETERVECTOR threads acquire a *pointer* to the most recent by accessing P. Through that pointer, the thread can access and use the θ and metadata of that PARAMETERVECTOR, in particular for calculating the gradient without copying. While a PARAMETERVECTOR V is in use, $V.n\_rdrs$ is non-zero (it is atomically increment-able and decrement-able in the START_READING() and STOP_READING() functions). Note that the update of the global pointer P, and the marking of the previous global vector as stale, are two operations. Hence, for a thread to acquire the latest PARAMETERVECTOR in a concurrency-safe manner, this must be done in a retry loop, in LATEST_POINTER(). Due to this fact and how the global pointers are updated, a read preceded by another read will not return parameter values older than its preceding read returned.

P4. *Conditions for safe recycling:* For reclaiming the memory of a PARAM-ETERVECTOR V, the $V.stale\_flag$ must be *true* and $V.n\_rdrs$ must be zero. The first condition ensures that the PARAMETERVECTOR instance is not the most recently published, and its address is no longer available to any thread (Algorithm B.3, line 31), ensuring no additional future accesses. The second condition ensures that no thread is currently accessing V, with the exception when a thread just acquired a pointer that just became stale, which subsequently will repeat after the staleness check that follows in line 8. Note that stale instances of PARAMETERVECTOR will be reclaimed by the last thread to access it, when calling STOP_READING().

P5. *Lock-free atomic updates of the shared vector:* The publish is attempted through a CAS invoked in a retry loop, and if it fails, another thread must have succeeded. Update attempts are repeated until CAS succeeds, or until a persistence bound T_p decided by the user has been exceeded. The loop thus implies lock-free progress guarantees. For $T_p = 0$ it implies similar semantics as the LoadLinked/StoreConditional primitive, hence its name *LoadAndUpdate-StorePersistenceConditional* (*LAU-SPC*). Note that bounded T_p essentially implies bounded retries. As formulated in (B.2), due to asynchrony, the gradients can be applied on a different PARAMETERVECTOR instance than the one that was used to compute the gradient. Hence, after finishing the gradient computation, threads acquire the pointer to the most recent published PARAM-ETERVECTOR instance a second time (Figure B.2), on which the update will be applied. The result is then a candidate for publishing, the success of which is decided as described above, implying update atomicity.

Based on the previous paragraphs, (in particular on points P1, P3 and P5, respectively points P2 and P4) we have:

Lemma B.3.1. *Reads and updates of θ in Leashed-SGD (i.e., LATEST_POINTER() and the LAU-SPC loop) satisfy lock-freedom and atomicity.*

Lemma B.3.2. *The memory recycling in Leashed-SGD (i) is safe, i.e., will not reclaim memory which can be used by any thread for reading or updating and (ii) bounds the memory to max $3m$ PARAMETERVECTOR instances simultaneously.*

Proof sketch. The first claim (i) follows from the definition of the safe_delete operation of the PARAMETERVECTOR, ensuring that the memory of an instance PC_i is reclaimed only if $stale\_flag = true$ (P points to a newer instance, ensuring no new readers of PC_i), $n\_readers = 0$ (no readers currently) and
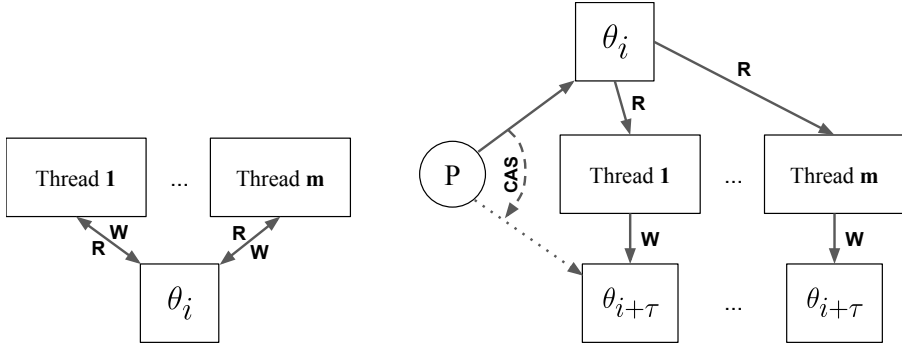
Figure B.2: Illustration of data access in *AsyncSGD* and Hogwild! (*left*) and *Leashed-SGD* (*right*). For *AsyncSGD* the read and write operations are protected through mutual exclusion. For *Leashed-SGD*, each thread accesses θ_i only through a read operation, then computes the update at a new memory location, becoming a candidate for $\theta_{i+\tau}$.

that the memory has not already been reclaimed. The second claim (ii) is realized by the fact that the memory recycling mechanism is exhaustive, i.e., PARAMETERVECTOR instances that will not be used further by any thread will eventually be reclaimed through the delete operation in line 10 of Algorithm 1. The reason is the following: each thread that finishes its use of a PARAMETERVECTOR instance will call the stop_reading operation, which in turn calls safe_delete, which reclaims the memory if safe, according to the above, i.e., it holds that the instance is currently not in use and will not be in the future. If that is not the case, then each thread that is currently using the instance will eventually invoke the safe_delete operation, the last of which will perform the reclamation. Now, from Algorithm 3 it is clear that in the worst case each thread has a unique *latest_param* on which it is an active reader, and an additional two PARAMETERVECTOR (*new_param* and *local_grad*), giving in total $3m$. □

A note on memory consumption. Note that *AsyncSGD* and Hogwild! need $2m + 1$ instances of PARAMETERVECTOR constantly. In *Leashed-SGD* threads compute gradients based on a published PARAMETERVECTOR instance, which will never be altered by any thread. After the gradient computation is finished, additional memory is allocated for new parameters. This mechanism enables an overall reduced memory footprint, in particular when gradient computation is time consuming. This is confirmed empirically in section B.5.

B.4 Contention and staleness

In the following we analyze the dynamics of the proposed *Leashed-SGD*, the effect of the *persistence bound*, and its impact on the contention and staleness.

B.4.1 Dynamics of *Leashed-SGD*

We analyze the dynamics of the threads, their progression under concurrent execution of *Leashed-SGD*. The model is similar to a G/G/1 queue, but with arrival and departure rates λ_i, μ_i varying over time, depending on the current state of the system.

For a single thread executing the gradient computation, the rate of arrival to the *LAU-SPC* (retry) loop is $\lambda^{(1)} = 1/T_c$, where T_c is the gradient computation time. For an m-thread fully concurrent execution, the arrival rate scales proportionally to the number of threads currently outside the *LAU-SPC* loop, hence $\lambda^{(m)} = (m-n)\lambda^{(1)}$ where n denotes the number of threads in the retry loop. Similarly, for the departure rate from the *LAU-SPC* loop we have $\mu^{(1)} = 1/T_u$ where T_u is the execution time the PARAMETERVECTOR UPDATE$(\cdot, \cdot)$. In summary:

$$\lambda_i^{(m)} = \frac{m - n_i}{T_c}, \quad \mu_i = \frac{n_i}{T_u} \tag{B.3}$$

We then describe the dynamics of how threads enter and leave the *LAU-SPC* retry loop of *Leashed-SGD* as follows:

$$n_{i+1} = n_i + \frac{m - n_i}{T_c} - \frac{n_i}{T_u} \tag{B.4}$$

where n_i is the number of threads executing the retry loop at time i. Note that the system (B.4) has a fixed point $n^* = (T_c/T_u + 1)^{-1}m$ at which the number of threads in the retry loop will stay constant. Note that n^* rewrites to $n^*/m = T_u/(T_u + T_c)$, i.e., that thread balance at the fixed point depends solely on the relative size of the update time T_u, highlighting the importance of the ratio T_u/T_c. In section B.5 we show closer measurements of T_c, T_u for different applications.

In the following, we study how n_i progresses for *Leashed-SGD*, stability and convergence about the fixed point.

Theorem B.4.1. *Assume we have an m-thread system where threads arrive to and depart from the Leashed-SGD LAU-SPC loop with the rates in (B.3). Then, we have that the number n_i of threads in the retry-loop in iteration i is given by*

$$n_i = \frac{1 - (1 - T_c^{-1} - T_u^{-1})^i}{1 + T_c/T_u}m + (1 - T_c^{-1} - T_u^{-1})^i n_o \tag{B.5}$$

where T_c, T_u denotes the time for gradient computation and update, and n_0 is the initial number of threads in LAU-SPC.

Proof. From (B.4), we have

$$
\begin{aligned}
n_i &= n_{i-1} + \frac{m - n_{i-1}}{T_c} - \frac{n_{i-1}}{T_u} \\
&= (1 - 1/T_c - 1/T_u)n_{i-1} + m/T_c \\
&= \ldots \\
&= \frac{m}{T_c} \sum_{j=0}^{i-1} (1 - 1/T_c - 1/T_u)^j + (1 - 1/T_c - 1/T_u)^i n_0 \\
&= \frac{m}{T_c} \frac{1 - (i - 1/T_c - 1/T_u)^i}{1/T_c + 1/T_u} + (1 - 1/T_c - 1/T_u)^i n_0
\end{aligned}
$$

$\square$

Corollary B.4.2. *The fixed point n^* is stable, and the system will converge towards $\lim_{i \to \infty} n_i = n^*$ for any initial n_0.*

The result is confirmed by taking $i \to \infty$ in (B.5).

The above results enable understanding of the dynamics of how threads progress throughout the execution, in particular that they converge to a balance between gradient computation and the *LAU-SPC*, to be used in the following.

B.4.2 Persistence analysis

The persistence bound implies a threshold on the maximum number of failed CAS attempts in *Leashed-SGD*, before threads compute a new gradient. This implies an increase, denoted by $\gamma > 0$, in departure rate from the *LAU-SPC* retry loop, proportional to the number of threads currently in the retry loop as follows:

$$\mu_i = \frac{n_i}{T_u}(1 + \gamma) \tag{B.6}$$

Corollary B.4.3. *Under the same conditions as in Theorem B.4.1, but using the departure rate (B.6), the fixed point moves to*

$$n_\gamma^* = \left(\frac{T_c}{T_u}(1 + \gamma) + 1\right)^{-1} m \tag{B.7}$$

Note that (i) $n_\gamma^* < n^*$ and (ii) n_γ^* vanishes as γ grows, showing the contention-regulating capability through a persistence bound, i.e., an increased γ.

As pointed out in **Chapter A**, the complete staleness τ_i of an update $\nabla L(v_i)$ according to (B.2) is comprised of two parts: $\tau_i = \tau_i^c + \tau_i^s$ where τ_i^c counts the number of published updates concurrent to the computation of $\nabla L(v_i)$, and τ_i^s counts the ones that compete with the update in focus and are scheduled before it; in particular here, the latter counts the competing updates in the *LAU-SPC* loop that succeed before that update. Considering now the estimation $\mathbf{E}[\tau_i^s] \approx n_\gamma^*$, it follows that the persistence mechanism described above for reducing contention effectively regulates the additional staleness component due to scheduling of ready gradients.

E.g., consider $T_p = 0$: for each published update there was no failed CAS, hence no other update was published after the corresponding gradient was used. Then $\tau_i^s = 0$, which is the maximum staleness reduction possible here. In section $B.5$ we study this empirically, showing it holds in practice and is effective for regulating contention and tune the staleness.

B.5 Evaluation

We present the results from our extended empirical study, benchmarking the methods in Section B.3, studying influence of consistency and associated synchronization, on the metrics described in Section B.2: convergence rate, statistical and computational efficiency, and memory consumption. The algorithms included are sequential SGD (SEQ), Lock-based *AsyncSGD* (ASYNC), Hogwild! (HOG), and *Leashed-SGD* with persistence $\infty, 1, 0$ (LSH_ps∞, LSH_ps1, LSH_ps0).

Implementation. The algorithms and the framework are implemented with C++, with OpenMP [86] for shared-memory parallel computations, and Eigen [87] for numerical. The framework extends the MiniDNN [88] C++ library for DL. For implementing the PARAMETERVECTOR and *Leashed-SGD*, a substantial refactoring was accomplished, extracting all learnable parameters into a collective data structure, the PARAMETERVECTOR. This abstraction forms an interface between SGD algorithm constructions and DL operations, enabling implementation of consistency of different degrees through various synchronization methods. The proposed framework *Leashed-SGD* is general and can be widely utilized for parallelization of SGD for various optimization problems, in particular ones of high dimension. For the empirical evaluation the framework is implemented in conjunction with ANN operations, facilitating further research exploring algorithms for parallel SGD for DL with various synchronization mechanisms.

Experiment setup. We evaluate the methods of Section B.3 for two DL applications, namely MLP and CNN training on the MNIST benchmarking dataset [89]. The proposed method, however, facilitates generic implementations of SGD, and is applicable over a broad spectrum of optimization problems. We choose to focus the evaluation around benchmarking on DL problems in order to evaluate on relevant applications, as well as to challenge the proposed method, keeping in mind the non-convex and highly irregular nature of the target functions such problems constitute. Moreover, it is in this domain where better understanding of how to support the processing infrastructure is the most needed. MNIST contains 60,000 images of hand-written digits $\in \{0, \ldots 9\}$, each belonging to one of ten classes, sampled in mini-batches of 512. The details of the MLP and CNN architectures are shown in Table B.1 and B.2 for MLP and CNN, respectively. The size of the parameter vector θ are $d = 134,794$ and $d = 27,354$ for MLP and CNN, respectively.

Layer	Layer details		
#	*Type*	*# Neurons*	*Act. fcn.*
1-3	Dense	128	ReLU
4	Dense	10	Softmax

Table B.1: MLP Architecture, $d = 134,794$.

Layer	Layer details				
#	*Type*	*# Filters*	*# Neurons*	*Kernel*	*Act. fcn.*
1	Conv[a]	4	-	(3,3)	ReLU
2	Pool[b]	-	-	(2,2)	ReLU
3	Conv[a]	8	-	(3,3)	ReLU
4	Pool[b]	-	-	(2,2)	ReLU
5	Dense	-	128	-	ReLU
6	Dense	-	10	-	Softmax

[a] *Convolutional layer* [b] *MaxPool layer*

Table B.2: CNN Architecture, $d = 27,354$.

The experiments are conducted on a 2.10 GHz Intel(R) Xeon(R) E5-2695 system with 36 cores on two sockets (18 cores per socket, each supporting two hyper-threads), 64GB memory, running Ubuntu 16.04.

Box plots in the figures contain statistics (1st and 3rd quantiles, minimum and maximum) from 11 independent executions of each setting; outliers are indicated with the symbol $+$. Where executions fail to reach the required precision ϵ, the measurement is not included as basis for the box. Such execution instances, and those that fail due to numerical instability from staleness, are indicated as 'Diverge' and 'Crash', respectively. This information is highlighted because failing DL training executions due to noise from staleness or hyper-parameter choices is a common problem in practice [81]. It is vital that training succeeds, and that the execution time thereby is not wasted. The threshold ϵ is specified in terms of percentage of the target function at initialization $L(\theta_0) \approx 2.3$.

Experiment outcomes. The steps of our experiment methodology, summarized in Table B.3, are as follows:

S1. *Convergence and hyper-parameter selection:* We benchmark the convergence of the algorithms considered under a wide spectrum of parallelism, and for varying step size η. In this step the executions are halted at $\epsilon = 50\%$ in order to acquire an overview of the general scalability and relative performance among the evaluated methods. The results are presented in Fig. B.3-B.4, showing a complete picture of the convergence rate and computational efficiency under varying parallelism, the metric of interest being the wall-clock time required until reaching ϵ-convergence. The baselines are at their best with $m = 16$ threads and $\eta = 0.005$, which we choose as a yardstick for further tests to ensure a fair comparison, and to stress-test *Leashed-SGD*. The results of the step size test appear in Figure B.11, showing higher capability of the proposed *Leashed-SGD* to converge for larger η.

S2. *High-precision convergence for MLP:* Using the setting selected according to the above, we benchmark the algorithms for reaching high precision ($\epsilon = 2.5\%$). We pay attention to the staleness τ distribution, to gain understanding based also on the results of section B.4. Using $m = 16$, $\eta = 0.005$, we benchmark *Leashed-SGD* and baselines to high-precision 2.5%-convergence, measuring the wall-clock time (Fig. B.5, top). *Leashed-SGD* shows competitive performance, with faster convergence and smaller fluctuations. In particular, LSH_ps∞ reaches $\epsilon = 2.5\%$ error within 65s median (compared to baselines' 89s and 80s). As hypothesized in section B.4, Fig. B.7 confirms that the staleness distribution is significantly reduced by the persistence bound.

S3. *Convergence rates for CNN:* We study the convergence for the CNN application, benchmarking time to convergence for increasing precision ϵ, studying the staleness and convergence over time. The proposed *Leashed-SGD*

Table B.3: Summary of experiments.

|	Experiment overview					
Step	*Architecture*	*Description*	*N.o. threads m*	*Precision ϵ*	*Step size η*	*Outcome*
S1	MLP*	Hyper-parameter selection	1-68	50%	$0.01-0.09$	Fig. B.3, B.4
S2	MLP*	High-precision convergence	16	$50\%, 10\%, 5\%, 2.5\%$	0.05	Fig. B.5-B.7
S3	CNN*	Convergence rate	16	$75\%, 50\%, 25\%, 10\%$	0.05	Fig. B.8
S4	MLP*	High parallelism	24, 34, 68	$75\%, 50\%, 25\%, 10\%$	0.05	Fig. B.5-B.7
S5	MLP*, CNN*	Memory consumption	16, 24, 34	any	0.05	presented in text
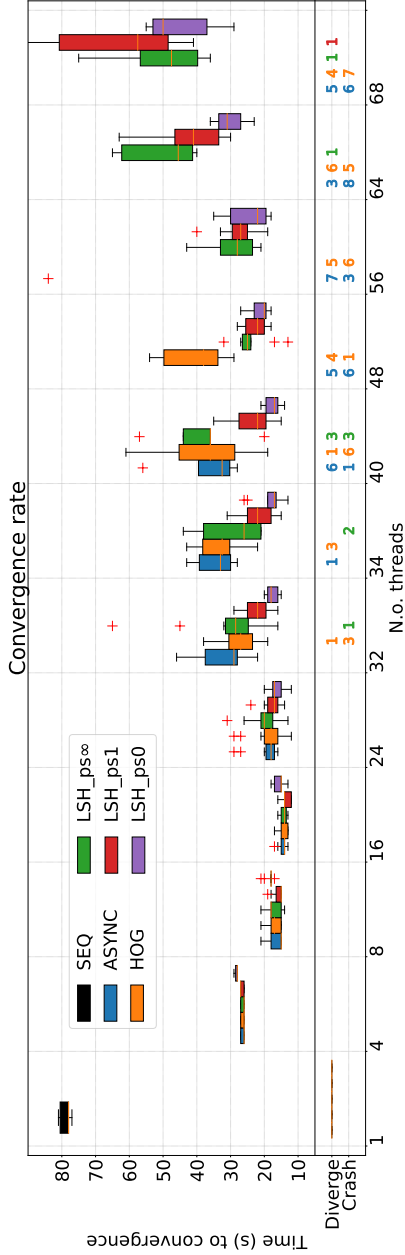
Figure B.3: ϵ-convergence rate for MLP training under varying parallelism, measuring wall-clock time until reaching an error threshold ($\epsilon = 50\%$ of initial error, providing a comparison of the general scalability). The optimum for the baselines ($m = 16$) is used in subsequent tests for a fair comparison. Under increased parallelism ($m > 16$), the convergence rate for the baselines (ASYNC, HOG) deteriorates, with many unstable executions, never achieving ϵ-convergence due to the instability from increased staleness. The proposed framework (LSH_psX, persistence bound X) on the other hand provides stable and fast convergence for up to 56 threads, with minimal penalty from staleness.
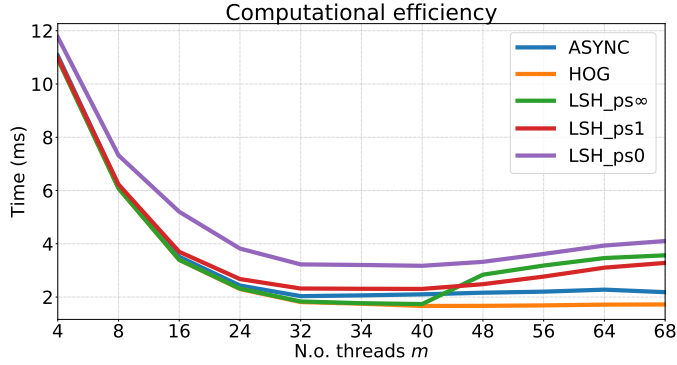
Figure B.4:   Computational efficiency, i.e., wall-clock computation time per SGD iteration. Computation time remains constant for the baselines under higher parallelism, although the many executions fail completely to converge, and would be wasted time in practice. The self-regulative property of *Leashed-SGD* on the other hand increases the computation time moderately under high parallelism, balancing latency and throughput under contention, and can hence achieve stable convergence in far more instances.

shows fewer diverging executions, with significant improvements in time to high precision convergence with up to $4\times$ speedup relative to the baselines *AsyncSGD* (Fig. B.8). The distribution of the wall-clock time to compute and apply gradients, respectively, are shown in Figure B.9. Despite having a lower dimensionality, the gradient computation time T_c is higher for CNN. This is due to the topological nature of the convolutional layer, where filters stride along the input image pixel by pixel. This requires in practice a large number of smaller matrix multiplications, as opposed to MLP which instead consists of few but significantly larger ones. However, the time to apply one gradient T_u is smaller in the CNN application, since the θ vector is smaller.   Since the dimension d of the PARAMETERVECTOR is significantly smaller for the CNN ($d = 27,354$) compared to the MLP ($d = 134,794$), the time T_u to apply an update is smaller, but due to the topological nature CNNs, the gradient computation time T_c is relatively high. The detailed measurements appear in Figure B.9; they are in the order of magnitude $T_c = 40$ms and $T_u = 0.6$ms for MLP, and $T_c = 110$ms and $T_u = 0.3$ms for MLP. This results in lower contention in the *LAU-SPC*. As a consequence, the contention-regulating effect of the *Leashed-SGD* algorithms does not kick in, hence showing similar staleness distribution as the baselines. The proposed *Leashed-SGD* nevertheless shows significant improvement in the convergence rate.

S4. *Higher parallelization for MLP:* We stress-test the methods, with $m = 24$, $m = 34$ (max. solo-core parallelism) and $m = 68$ (max. hyper-threading). The results appear in Fig. B.5-B.7, showing *Leashed-SGD* provides significantly improved convergence and stability, with improved staleness.

S5. *Memory consumption:* We perform a fine-grained continuous measurement of the memory consumption of all algorithms considered, for MLP and CNN training. For the CNN application, due to its sparse nature, the gradient
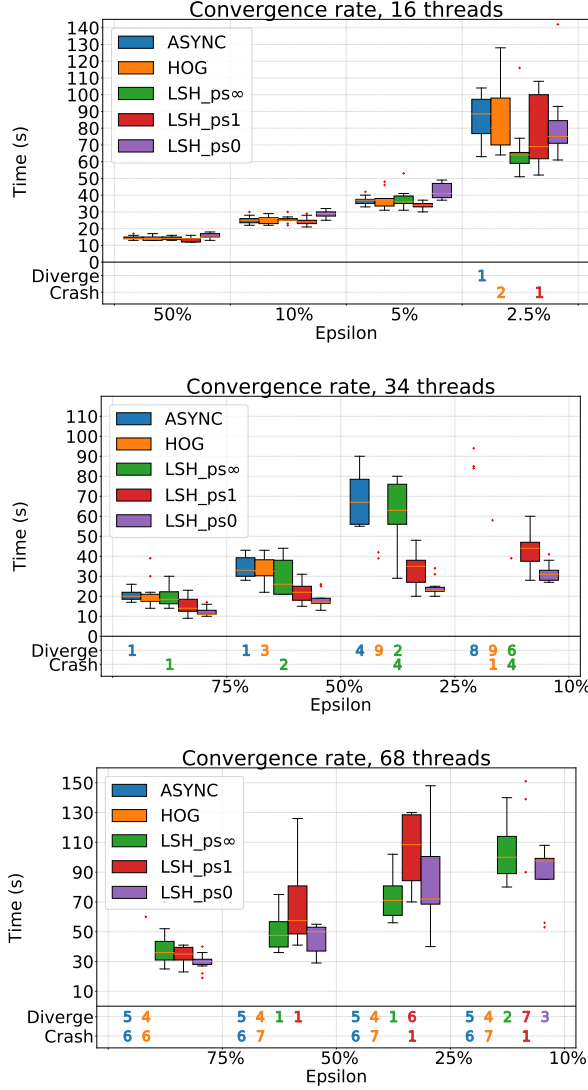
Figure B.5: ϵ-convergence rate for MLP with $m = 16$ threads to high precision (*top*), $m = 34$ threads (*middle*) and maximum parallelism $m = 68$ threads (*bottom*). The baselines (ASYNC, HOG) show an overall slower convergence and higher number of executions that fail before reaching the high precision (e.g., $\epsilon = 10\%$), especially under maximum parallelism $m = 68$, where no baseline execution managed to reach $\epsilon = 50\%$ of the error at initialization.
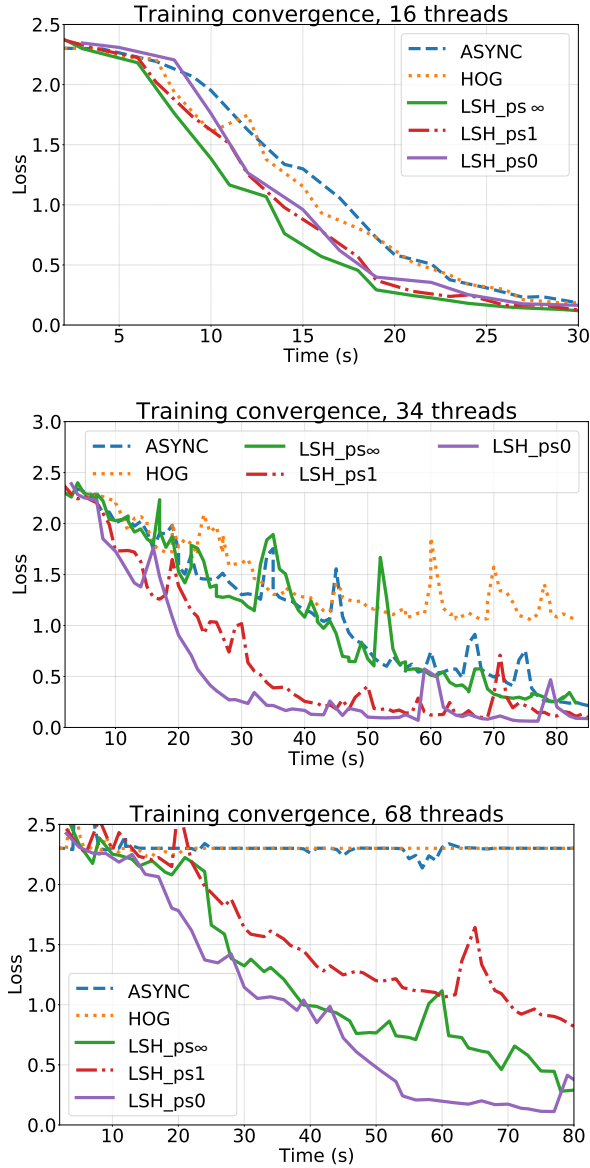
Figure B.6: MLP training progress over time with $m = 16$ threads (*top*), with $m = 34$ threads (*middle*) and maximum parallelism $m = 68$ threads (*bottom*). The proposed framework (LSH_psX, persistence bound X) converges significantly faster relative to baselines (ASYNC, HOG). Under maximum parallelism, the baselines completely fail to converge and oscillate around the initialization point.
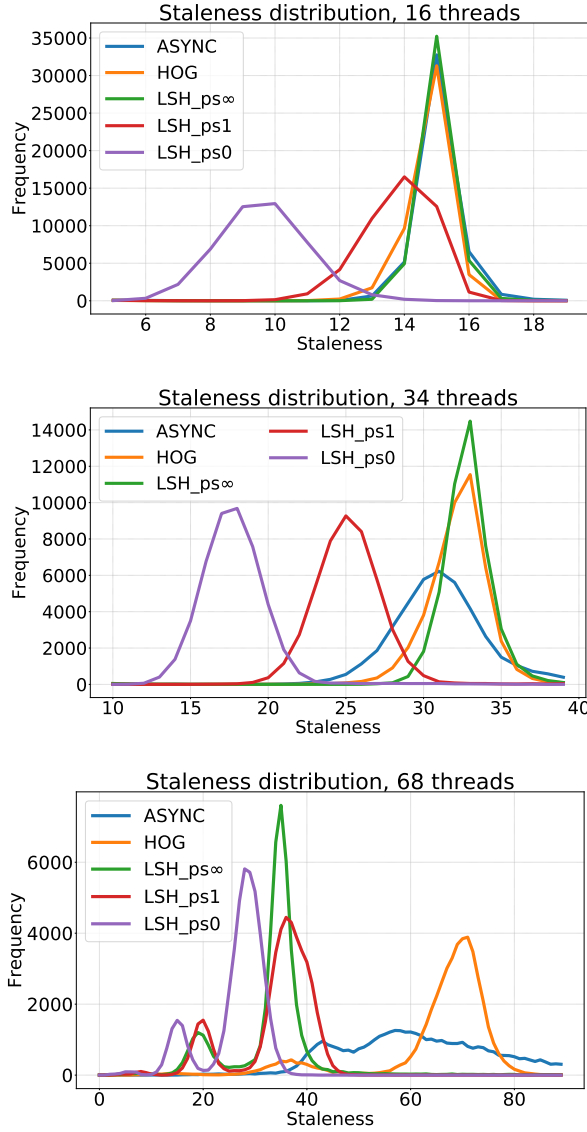
Figure B.7: Staleness distribution over time for MLP with $m = 16$ threads (*top*), $m = 34$ threads (*middle*) and maximum parallelism $m = 68$ threads (*bottom*). The effect from the contention-regulating persistence bound ($ps \in \{0, 1, \infty\}$) is clear, and effectively reduces the overall staleness distribution. Under maximum parallelism $m = 68$ the ability of the proposed framework (LSH) to self-regulate the balance between latency and throughput becomes clear, with overall lower staleness as well as naturally appearing clusters of threads with higher update rate. The baselines show overall higher staleness distributions, as well as high irregularity for ASYNC due to contention about the locks.
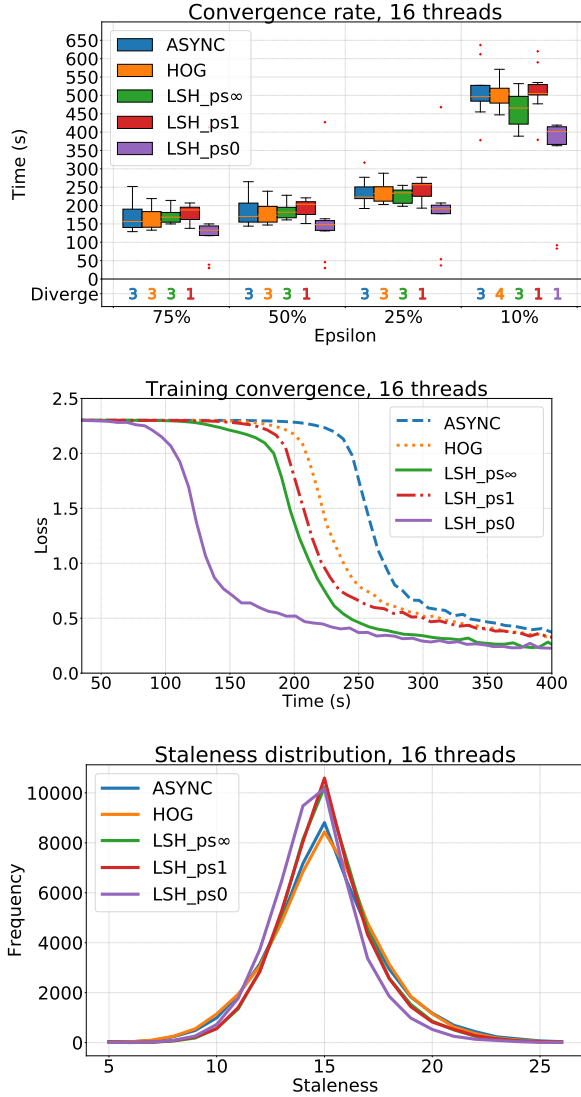
Figure B.8: ϵ-convergence rates to different precision for CNN training with $m = 16$ threads; LSH_ps0 shows 400s median 10%-convergence time, compared to ~ 500s for the baselines, with two executions showing remarkable 10%-convergence time of below 100s, i.e., a $4\times$ speedup relative to the best baseline convergence rate of 375s (*top*) training progress over time (*middle*) and staleness distribution (*bottom*). The proposed framework (LSH) consistently shows improved convergence rate, as well as solution of lower error.
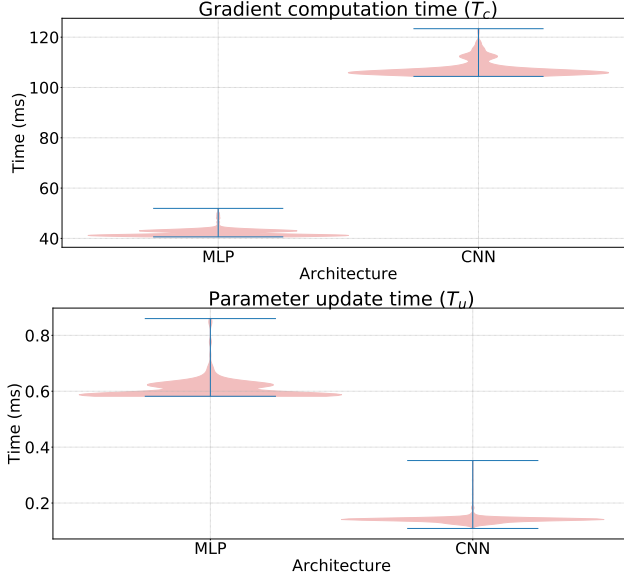
Figure B.9: Gradient computation and parameter update times T_c, T_u (top, bottom, respectively) for MLP and CNN.
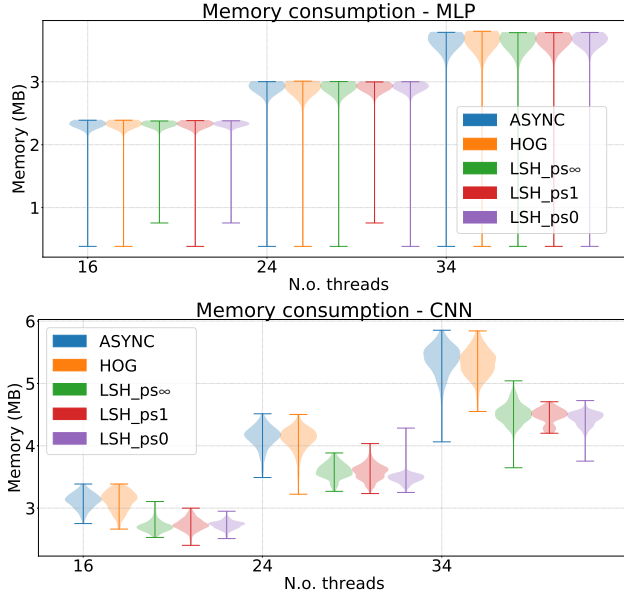

Figure B.10: Memory consumption measured continuously on second granularity for MLP (top) and CNN (bottom).
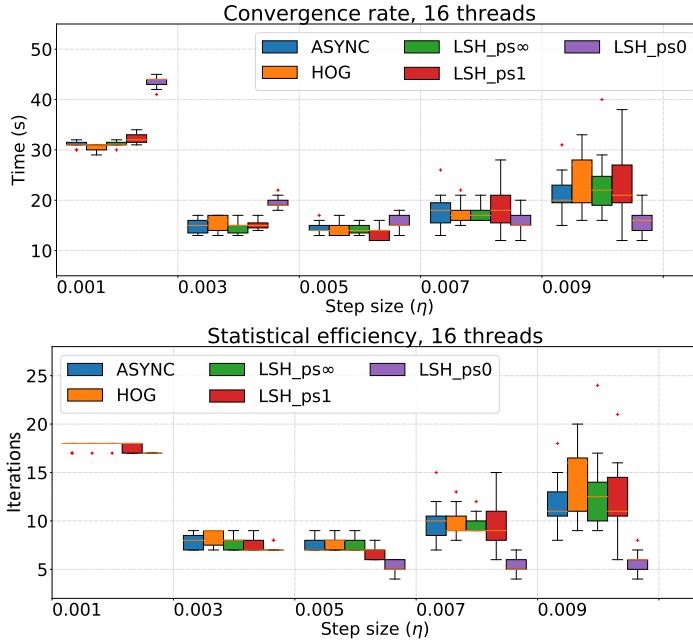
Figure B.11: Step size tuning (top), confirming the choice $\eta = 0.005$, and statistical efficiency (bottom), showing 50%-convergence.

computation vs. update application time ratio T_c/T_u is high, which enables *Leashed-SGD* to reduce the memory consumption by 17% on average thanks to dynamic allocation of PARAMETERVECTOR and efficient memory recycling (Figure B.10). The measurements were acquired using the UNIX `ps` command, collected with second granularity.

Summary of outcomes.   *Leashed-SGD* shows overall an improved convergence rate, stable under varying parallelism and hyper-parameters, and significantly fewer executions that fail to achieve ϵ-convergence. In presence of contention, the lock-free nature enables *Leashed-SGD* to self-regulate the balance between throughput and latency and converge in settings where the baselines fail completely. Even the case that with $T_p = \infty$, i.e., without starvation-freedom, we see persistent improvements relative to the baselines, demonstrating in this demanding context too, a useful property, namely that lock-freedom balances between system-wide throughput and thread-associated latency [90–92].

B.6   Related work

The study of numerical methods under parallelism sparked due to the works by Bertsekas and Tsitsiklis [33]. Distributed and parallel asynchronous SGD has since been an attractive target of study, e.g. [25, 39, 42, 79], among which HOGWILD! [20]. In the recent [19] the concept of bounded divergence between the parameter vector and the threads' view of it is introduced, proving convergence bounds for convex and non-convex problems. De Sa et. al [44] introduced a framework for analysis of HOGWILD!-style algorithms. This was

extended in [45], showing the bound increases with a magnitude of $\sqrt{d}$ due to inconsistency, implying higher statistical penalty for high-dimensional problems. This strongly motivates studying algorithms which, while enjoying the computational benefits of lock-freedom, also ensure consistency. To our knowledge, this has not been done prior to the present work.

In [41] the algorithmic effect of asynchrony in $AsyncSGD$ is modelled by perturbing the iterates with noise. Their framework yields convergence bounds, but as described in the paper, are not tight, and rely on strong convexity.

In [16], with motivation related to ours, 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 comparison of different computing architectures. In contrast, we propose a framework of consistency-preserving algorithmic implementations of $AsyncSGD$ together with HOGWILD!, that covers the associated design space of $AsyncSGD$ algorithms, and we focus on MLP and CNN, which are inherently more difficult to parallelize.

In [46], as in this work, the focus is the fundamental limitation of data parallelism in ML. They, too, point out that the limitations are due to concurrent SGD parameter accesses, usually diminishing or even negating the parallelization benefits. 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, that we consider in this paper, as they explain, their work is not directly applicable, since in DNNs permitting "good" dependence violation is the common parallelization approach.

There are works introducing adaptiveness to staleness [24, 52, 53] and in particular in **Chapter A** for a deep learning application. This research direction is orthogonal to this work and can be applied in conjunction with the algorithms and synchronization mechanisms considered here.

Asynchronous SGD approaches for DNNs are scarce in the current literature. In the recent work [47], 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 [48] theoretical convergence analysis is presented for $SyncSGD$ with $once\text{-}in\text{-}a\text{-}while$ synchronization. They mention the analysis can guide in applying $SyncSGD$ for DL, however the analysis requires strong convexity. [49] 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 [50] a topology for decentralized parallel SGD is proposed, using pair-wise averaging synchronization. In the recent [18] a partial all-reduce relaxation of $SyncSGD$ is proposed, showing improved convergence rates in practice when synchronizing only subsets of the threads at a time, due to higher throughput, complemented with convergence analysis for convex and non-convex problems. In particular, the empirical evaluation shows only requiring one thread (i.e., $AsyncSGD$) gives competitive performance due to the $wait\text{-}freedom$ that follows from the lack of synchronization.

B.7   Conclusions

We propose the extensible generic algorithmic framework *Leashed-SGD* for asynchronous lock-free parallel SGD, together with PARAMETERVECTOR, a data type providing an abstraction of common operations on high-dimensional model parameters in ANN training, facilitating modular further exploration of aspects of parallelism and consistency.

We analyze safety and progress guarantees of the proposed *Leashed-SGD*, as well as bounds on the memory consumption, execution dynamics, and contention regulation. Aiming at understanding the influence of synchronization methods for consistency of shared data in parallel SGD, we provide a comprehensive empirical study of *Leashed-SGD* and established baselines, benchmarking on two prominent *deep learning* (DL) applications, namely MLP and CNN for image classification. The benchmarks are chosen in order to challenge the proposed model against the baselines and provide new useful insights in the applicability of *AsyncSGD* in practice.

We observe that the baselines, i.e., standard implementations of *AsyncSGD*, are sensitive to hyper-parameter choices and are prone to unstable executions due to noise from staleness. The proposed framework *Leashed-SGD* outperforms the baselines where they perform the best, and provides a balanced behavior, implying stable and timely convergence for a far wider spectrum of parallelism.

The methods are implemented in an extensible C++ framework, interfacing DL operations with parallel SGD algorithms, facilitating further research of algorithms for parallel SGD for DL with various synchronization mechanisms.

Chapter C

Adaptation of the article:

`ASAP.SGD`: Instance-based Adaptiveness to Staleness in Asynchronous SGD

K. Bäckström, M. Papatriantafilou, P. Tsigas

Abstract

Concurrent algorithmic implementations of Stochastic Gradient Descent (SGD) give rise to critical questions for compute-intensive Machine Learning (ML). Asynchrony implies speedup in some contexts, and challenges in others, as stale updates may lead to slower, or non-converging executions. While previous works showed asynchrony-adaptiveness can improve stability and speedup by reducing the step size for stale updates according to static rules, there is no one-size-fits-all adaptation rule, since the optimal strategy depends on several factors. We introduce (i) `ASAP.SGD`, an analytical framework capturing necessary and desired properties of staleness-adaptive step size functions and (ii) TAIL-τ, a method for utilizing key properties of the *execution instance*, generating a tailored strategy that not only dampens the impact of stale updates, but also leverages fresh ones. We recover convergence bounds for adaptiveness functions satisfying the `ASAP.SGD` conditions, for general, convex and non-convex problems, and establish novel bounds for ones satisfying the Polyak-Lojasiewicz property. We evaluate TAIL-τ with representative *AsyncSGD* concurrent algorithms, for Deep Learning problems, showing TAIL-τ is a vital complement to *AsyncSGD*, with (i) persistent speedup in wall-clock convergence time in the parallelism spectrum, (ii) considerably lower risk of non-convergence, as well as (iii) precision levels for which original SGD implementations fail.

C.1   Introduction

The ascending interest in concurrent SGD is due to the explosion of data volumes, requiring scalable systems to process them in ML and Artificial Neural Network (ANN) applications. However, parallelization of the inherently sequential SGD process is non-trivial since each iteration requires the computation of the previous one. Besides understanding the dynamics of such executions, achieving resource-efficiency is a known significant target, since it can imply significant improvements in energy efficiency.

Traditional synchronous SGD (*SyncSGD*) conforms to the sequential SGD semantics by employing iteration-level parallelism, and lock-step-style synchronization. In practice, *SyncSGD* accelerates updates by data-parallel concurrent gradient computation, e.g., by GPU-acceleration, or aggregating gradient contributions in distributed settings. *SyncSGD* improves computational efficiency up to a point, but suffers limitations, as each iteration is as slow as the slowest contributing worker. However, from an optimization standpoint *SyncSGD* is analogous to sequential SGD, with well-understood convergence properties.

In contrast, asynchronous concurrent SGD (*AsyncSGD*) introduces a higher-level parallelism by relaxing the sequential SGD semantics, allowing asynchronous reads/updates on the shared ML model θ. Consequently, *AsyncSGD* provides computational benefits, however at the price of *asynchrony-induced noise* due to the *staleness* τ that arises when updates are not applied to the same states based on which they were computed, but instead on ones that have been updated τ times in-between. It was within *convex optimization*, targeting primarily regression problems [22, 93], where it was shown that the *asynchrony-induced noise* had small impact on the quality of the updates, and that the computational benefits of reduced synchronization provided speedup for certain problems. A relevant example is HOGWILD! which, with only component-wise atomic access to θ (i.e., relaxed consistency by not ensuring atomicity for the complete vector) showed almost-linear speedup for strongly convex and sparse problems [20, 43, 45]. Asymptotic bounds for *AsyncSGD* were similarly established under strong convexity and smoothness assumptions, and for non-convex problems, such as matrix completion [44].

However, these analytical confinements make the concluding outcomes non-applicable for a wider class of applications, including Deep Learning (DL), characterized by inherent non-convexity. This is confirmed in recent works [46, 47], as well as in **Chapter B**, showing challenges in achieving stable and high-quality convergence with *AsyncSGD* for ANN training, due to staleness. The importance of progress and consistency guarantees of *AsyncSGD* is emphasized in **Chapter B**, where the introduced lock-free and consistent *Leashed-SGD* shows major improvements in convergence stability (reduced risk of non-convergence) compared to lock-based *AsyncSGD* and HOGWILD!.

Recent works show that *asynchrony-awareness* can reduce negative effects of staleness by dampening the step size of stale updates [24, 53, 56, 61]. In particular, staleness-adaptiveness has proven to reduce the statistical penalty of asynchrony in *AsyncSGD*, improving its stability and convergence rate.

Challenges. In summary, stable convergence of *AsyncSGD* is critically sensitive to (i) parallelism degree, (ii) asynchrony-awareness, and (iii) progress and consistency guarantees of the algorithmic implementation, and the mecha-

nisms to ensure them, e.g., locking. These factors have been studied mostly in isolation, and there is an imminent need to evaluate them in conjunction, to understand how *AsyncSGD* can be utilized effectively in practice.

Moreover, existing staleness-adaptive methods either (i) statically scale the overall system step size at initialization or (ii) use a pre-defined heuristic or staleness model to regulate the step size based on the observed staleness. An inherent pitfall of adapting the step size during execution is that the *overall* magnitude might be altered, which by itself will impact the efficiency. E.g., previous works employ adaptiveness strategies that almost exclusively *diminish* the overall step size [24, 53, 56, 61]. This is problematic for several reasons, e.g., (i) it may be fatal for applications sensitive to the choice of the step size (read: Deep Learning), leading to non-converging executions, and (ii) it introduces ambiguity regarding the source of potential performance improvements, reducing comparability between methods. Besides, these approaches take no consideration of the effect of underlying system parameters, such as hardware, scheduling, synchronization and consistency properties. These aspects, just like the number of workers, and other hyper-parameters, significantly influence the convergence rate in general [16], and the *staleness distribution* in particular, as shown in **Chapter B**. and can even result in *multi-modal* ones as we show here. Hence, there are inherent challenges in designing adaptation schemes capable of incorporating the effects of all of these critical aspects in conjunction.

Contributions. We introduce the *instance-based asynchrony-awareness* paradigm, with the execution-dynamic TAIL-τ staleness-adaptive step size function. We also establish a framework for <u>a</u>daptiveness to <u>s</u>taleness in <u>a</u>synchronous <u>p</u>arallel <u>SGD</u> (`ASAP.SGD`), capturing key properties of such functions in general. In detail:

- `ASAP.SGD` captures general staleness-adaptive step size function properties, (i) necessary for maintaining overall step size magnitudes and ensuring method comparability, and (ii) desired for prioritizing gradient freshness.

- Within `ASAP.SGD`, we introduce TAIL-τ, a dynamic staleness-adaptive step size function, that (i) utilizes the observed *staleness distribution* as means to implicitly take underlying system parameters into consideration, and (ii) generates an execution-specific adaptation strategy, in the spirit of instance-based optimization [94]

- We recover asymptotic convergence bounds for TAIL-τ in particular, and general ones within the `ASAP.SGD` framework, for convex and non-convex applications. We establish novel bounds for functions satisfying the Polyak-Lojasiewicz (PL) condition, which characterizes the shape of certain non-convex targets, and holds for several relevant ML loss functions, including least squares, logistic regression, support vector machines and deep ANNs.

- We implement TAIL-τ, extending the parallel *AsyncSGD* implementation framework of **Chapter B**, to promote further exploration of general staleness-adaptiveness within `ASAP.SGD`. The results show that TAIL-τ is a vital complement for *fast* and *stable* convergence for any *AsyncSGD* implementation, across the parallelism spectrum, due to its dynamic instance-based generation of tailored step size strategies. In particular, the evaluation, capturing several representative system features associated with synchronization, parallelism,

execution-ordering properties, shows that for image classification training with LeNet and MLP, on MNIST, Fashion-MNIST, and CIFAR-10, TAIL-τ achieves significantly faster convergence persistently (e.g., 60% speedup, on average, for LeNet training on MNIST), for three fundamentally different *AsyncSGD* implementations, and drastically lowers the risk of non-converging executions, especially to higher precision.

C.2   Background and related work

Adaptive parallel SGD.   (cf. also Table C.1) Staleness/asynchrony awareness was first studied for smooth and convex problems in [51], introducing a step size reduction based on *worst-case* staleness. Adaptiveness to *observed* staleness was studied in [52] assuming convexity, sparse gradients and certain ordering of *reads* and *updates* across threads and evaluated for logistic regression. [24], with a $1/\tau$ staleness compensation scheme in *semi-synchronous* distributed settings, show empirically speedup for ANN training with limited parallelism. For *partial asynchrony*, [96] introduced an adaptive scheme for regulating synchronization frequency, showing convergence bounds in non-convex Polyak-Lojasiewicz functions. In contrast, our work establishes convergence bounds for *fully asynchronous* SGD with unbounded staleness, using staleness-adaptive step size strategies, for general non-convex functions, as well as ones satisfying the Polyak-Lojasiewicz condition.

AdaDelay [53] proposed $\mathcal{O}(1/\sqrt{\tau})$ staleness-adaptive step sizes for smooth, convex problems, showing scalability improvements. Their analysis was based on a uniformly distributed staleness model, which **Chapter A** establishes to be a simplifying assumption; the latter also introduced the *MindTheStep-AsyncSGD* framework, proposing $\mathcal{O}(C^{-\tau})$ and $\mathcal{O}(C^{-\tau}/\tau!)$ schemes based on a Poisson-based staleness model, showing improved convergence rates for practical DL. [95] introduced a regret-based delay-adaptive approach for convex, smooth settings, while in a Federated Learning (FL) context, [56] adopted an exponential dampening approach ($\mathcal{O}(C^{-\beta\tau})$), explored initially in [97] (see **Chapter A**), where the rate of decay is based on the s-th percentile of the staleness distribution. The approach of [56] showed practical benefits for online ML applications at the edge.   In [61], a $\mathcal{O}(i^{-\tau})$ staleness-adaptive scheme is proposed, analyzed under quasi- and star-convex functions, showing improved convergence for projected GD with artificial noise, under simulated staleness. The aforementioned works are however mostly *static* in their strategy, i.e., are either model-based or based on a heuristic (e.g., $1/\tau$). Here we study closely the staleness distribution and explore how to utilize this information to *generate the strategy itself*, which (see Section C.6) entails significant improvements in convergence and robustness.

Non-convex asynchronous SGD.   The literature on standard, non-adaptive, convex *AsyncSGD* is vast, and a useful overview is in [28]. Here we highlight recent relevant *AsyncSGD* results for practical non-convex applications, including DL. In [55], linear convergence was established under the Polyak-Lojasiewicz condition, however assuming bounded staleness; the empirical evaluation focused on logistic regression, a convex problem. [46] proposed a static method for ensuring data-disjoint concurrent accesses, showing promising scalability for the non-convex problem of matrix factorization with SGD; the method

Table C.1: Staleness-adaptive *AsyncSGD*: literature highlights and new contributions.

	Online	Strategy	Mean-pres.	Prio-pres.	Convergence guarantees	Evaluation
[51]	×	-	×	-	Convex	LR[1]
[52]	✓	-	×	-	Convex	LR[1]
[24]	✓	$1/\tau$ (static)	×	✓	Non-convex	ANN
AdaDelay [53], [95]	✓	$\mathcal{O}(1/\sqrt{\tau})$ (static)	×	✓	Convex	LR[1]
MindTheStep-AsyncSGD (**Chapter A**)	✓	Model-based (static)	✓	-	Convex	ANN
[61]	✓	$\mathcal{O}(i^{-\tau})$ (static)	×	✓	Convex+[2]	LR+[2]
FLeet [56]	✓	$\mathcal{O}(e^{-\beta\tau})$ (semi-dynamic)	×	✓	-	ANN
ASAP.SGD	✓	Dynamic	✓	✓	Non-convex, PL	ANN

[1] Logistic regression [2] includes star- and quasi convex, and the Rosenbrock test function.

is however, as they state, not applicable to DL in general. [47] proposed a semi-asynchronous SGD approach, showing speedup for DL on CPU and GPU architectures, requiring a synchronizing master thread which averages updates [98] proposed a byzantine-tolerant asynchronous SGD framework, using a parameter server ensuring quality and relevance of updates. Similarly to ours, their work covers Polyak-Lojasiewicz problems, however to the best of our knowledge, our work is the first to do so for instance-based asynchrony-aware algorithmic implementations of *AsyncSGD*.

Optimization problem. We consider the optimization problem

$$\underset{\theta \in \mathbb{R}^d}{\text{minimize}} \quad L_D(\theta) \tag{C.1}$$

where (i) D is the data set to be processed, (ii) $\theta \in \mathbb{R}^d$ is the ML model that encodes the learned knowledge of D and (iii) the target function $L \colon \mathbb{R}^d \to \mathbb{R}^+$ quantifies the loss (error) of θ over D. Given some randomly chosen initial θ_0, the first-order iterative optimizer SGD repeats the following:

$$\theta_{i+1} = \theta_i - \eta_i \nabla \tilde{L}(\theta_i) \tag{C.2}$$

where $\theta_i \in \mathbb{R}^d$ is the state of the model θ, and η_i is the step size, in iteration i. We assume that $\tilde{L} = L_B$ where $B \subset D$ is a uniformly sampled mini-batch of data, and that $\tilde{L}$ is an unbiased estimator of L_D, i.e., $\mathbf{E}\left[\tilde{L}(\theta)\right] = L_D(\theta) \, \forall \theta \in \mathbb{R}^d$.

We assume that mini-batch samples, and hence the stochastic gradients $\nabla \tilde{L}$, are mutually statistically independent. The loss function $L_D \colon \mathbb{R}^d \to \mathbb{R}^+, \theta \mapsto L_D(\theta)$ quantifies the performance of an ANN model, parameterized by θ. SGD is repeated until θ satisfies ϵ-convergence, defined as $\|L(\theta) - L(\theta^*)\| < \epsilon$, θ^* being a global minimum of L.

C.3   System model and problem analysis

Algorithm C.1 Staleness-adaptive shared-memory *AsyncSGD*

GLOBAL loss function L, iteration counter i, max. n.o. iterations T, shared state θ, step size function $\eta(\tau)$

1: Let $(i, \theta) \leftarrow (0, \text{rand\_init}())$ ▷ Randomly initialize θ
2: <u>Each thread:</u>
3: **while** $i < T$ **do**
4:     $(i_{local}, \theta_{local}) \leftarrow \text{copy}(i, \theta)$
5:     $\nabla_{local} \leftarrow \nabla \tilde{L}(\theta_{local})$ ▷ Compute local random gradient
6:     $(i', \theta') \leftarrow \text{copy}(i, \theta)$ ▷ Acquire latest state
7:     $\tau \leftarrow i' - i_{local}$ ▷ Calculate staleness
8:     $(i, \theta) \leftarrow (i' + 1, \theta' - \eta(\tau)\nabla_{local})$

We consider a system with m concurrent asynchronous threads or processes. Threads follow the SGD rule of (1.1) asynchronously, within the boundaries of the algorithm which implements it, being responsible for potential guarantees on consistency and progress. An outline of a standard shared-memory parallel *AsyncSGD* implementation is provided in Algorithm C.1, showing also how a staleness-adaptive step size is introduced. In direct shared memory communication, threads have atomic access to single-word locations for read

and modify operations; in arbitrary contexts, steps 4 and 8 in Algorithm C.1 can be executed through the help of a parameter server, through requests to read and update θ. Due to asynchronous reads and updates of θ there can be intermediate updates, hence, the progression of θ follows:

$$\theta_{i+1} = \theta_i - \eta_i \nabla \tilde{L}(v_i) \tag{C.3}$$

where $v_i = \theta_{i-\tau_i}$ is the view of the updating thread in iteration i, and τ_i is the staleness, defined as the number of *intermediate* updates. When atomicity is not guaranteed, e.g., HOGWILD!, a total ordering of the updates is not naturally imposed and has to be defined. Here, we assume a total ordering as in [45] and define the staleness thereafter. By *staleness distribution* we refer to the distribution of all observed staleness values throughout a particular execution of *AsyncSGD*. We consider staleness (τ_i) to be a stochastic process, the elements of which, unlike the stochastic gradients, are not necessarily mutually independent. However, we assume $\mathbf{E}[\tau_i] = \bar{\tau}\ \forall i$ and that the execution is *non-anticipative* in the sense that states are mean-independent of future instances of the staleness, in particular:

$$\mathbf{E}[\tau_i \mid \tau_{i'}] = \mathbf{E}[\tau_i]\ \forall i < i' \tag{C.4}$$

since the expected staleness is not influenced by future staleness. Note that this implies $\mathbf{E}[\tau_i \tau_{i'}] = \mathbf{E}[\tau_i]\mathbf{E}[\tau_{i'}]$. Similarly, we assume that the staleness is mean-independent of θ, since the statistical properties of the convergence progress are not expected to influence the delays of individual threads, which are related to hardware and scheduling.

The step η_i of iteration i will, unless stated otherwise, is considered a function of the staleness τ_i in the same iteration; the details appear in the subsequent section. In the analysis section, we will generally use the notation $\mathbf{E}[x] = \bar{x}$.

Dampening is not sufficient.   The primary focus of previous approaches has been to dampen the step size of stale updates (cf. Table C.1). However, the overall staleness distribution changes with higher parallelism [53] (see also **Chapter B**), in particular for m threads $\mathbf{E}[\tau] \approx m - 1$ holds in practice, as discussed in **Chapter A**. An adaptive step size which merely diminishes stale updates, will consequently tend to very small values as m increases. This introduces a scalability issue, especially for non-convex problems, such as DL, which are step size sensitive, requiring updates of sufficiently coarse granularity to retain the level of stochasticity necessary for convergence. In fact, the commonly adopted $\mathcal{O}(\tau^{-1})$ scaling, as well as the FLeet $\mathcal{O}(e^{-\beta\tau})$ exponential dampening, were evaluated in our study, and compared with both constant step size *AsyncSGD*, as well as the staleness-adaptive function TAIL-τ proposed here. The results show convergence rates *orders of magnitude* slower than just the corresponding non-adaptive variant, and especially when compared to the proposed TAIL-τ function (see Section C.6).

C.4   Method

Here we present the `ASAP.SGD` framework, the staleness-adaptive TAIL-τ step size function, and their properties.

C.4.1   The `ASAP.SGD` framework

We start by formalizing the concept of a staleness-adaptive step size, and its connection to the overall base step size.

Definition C.4.1. A staleness-adaptive step size function $\eta\colon \mathbb{N} \to \mathbb{R}^+$, $\tau \mapsto \eta(\tau : \eta^0)$, given some base step size $\eta^0 \in \mathbb{R}^+$, maps the stochastic staleness τ_i of the update at time i onto a step size $\eta(\tau : \eta^0)$ to be used for that update.

The base step size η_0 is typically the "best known" step size for the problem at hand for standard sequential SGD. A staleness-adaptive function then alters this step size online, based on observed staleness. Definition C.4.1 implies in particular that the step size, as a function of the staleness, is consequently also considered stochastic. The step size η_i at iteration i is however influenced only by the staleness τ_i.

A challenge with staleness-adaptive step sizes is that they may alter the overall magnitude of the updates, which is undesirable since (i) it induces deviation from the expected step size magnitude, with unpredictable impact on the convergence, and (ii) makes it inherently different to compare the convergence impact of different strategies. We introduce the *mean-preservation* property to address this.

Definition C.4.2 (Mean-preservation)**.** A staleness-adaptive step size function η is referred to as *mean-preserving* if

$$\mathbf{E}\big[\eta(\tau : \eta^0)\big] = \eta^0 \tag{C.5}$$

Definition C.4.2 ensures that the average step size used throughout an execution of *AsyncSGD* is exactly η^0. Performance benefits due to a *mean-preserving* adaptive step size can hence be assured to be due to the *adaptation* strategy, as opposed to using a step size of an overall different magnitude. In the following, in the context of mean-preserving step sizes, we use notation η^0 and $\bar{\eta}$ interchangeably.

Lastly, we introduce the *priority-preservation* property:

Definition C.4.3 (Priority-preservation)**.** A staleness-adaptive step size function η is referred to as *priority-preserving* if η is *non-increasing* with respect to τ, i.e.

$$\eta(\tau + 1 : \eta^0) \leq \eta(\tau : \eta^0) \quad \forall \tau \tag{C.6}$$

Definition C.4.3 implies not only that *stale* updates are lower prioritized, but also that *fresh* updates can be *emphasized*. With the above in mind, we now define the `ASAP.SGD` framework for staleness-adaptive step size functions:

Definition C.4.4 (`ASAP.SGD`)**.** A function η is an `ASAP.SGD` step size function, iff η is *staleness-adaptive*, *mean-preserving* and *priority-preserving*.

C.4.2   The TAIL-τ function

Next, within the `ASAP.SGD` framework we define the instance-based staleness-adaptive TAIL-τ step size function, which considers the overall *staleness distribution*, to dynamically compute an execution-tailored adaptation strategy.
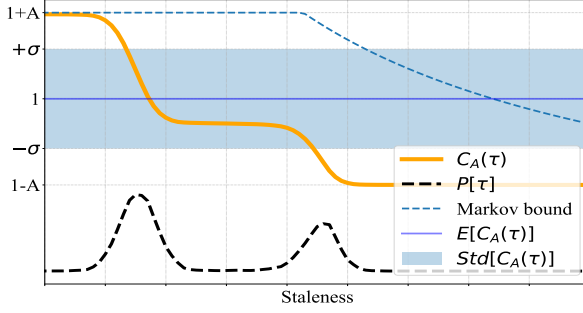
Figure C.1: The scaling factor $C_A(\tau)$ of TAIL-τ, relative a staleness distribution, with properties from Lemma C.4.7. Fresh updates (low staleness) are emphasized, stragglers are dampened.

Definition C.4.5. A TAIL-τ function is a staleness-adaptive step size function η which is of the form

$$\eta(\tau : \eta^0) = C_A(\tau) \cdot \eta^0 \qquad (C.7)$$

where the scaling factor C_A is given by

$$C_A(\tau) = 1 + A \cdot (1 - 2F_{\bar{\tau}}(\tau))$$

Here, $F_{\bar{\tau}}(\tau) = \mathbf{P}[\bar{\tau} \leq \tau]$ is the cumulative distribution function (CDF) of the stochastic staleness.

The amplitude parameter A of the TAIL-τ function specifies the maximum deviation of η from the base step size η^0. The scaling factor $C_A(\tau)$ of the TAIL-τ function (C.7) utilizes the CDF of τ, implicitly taking the system staleness distribution into consideration for generating a dynamic adaptive step size function, tailored to the specific execution. This is visualized in Figure C.1, showing the response of $C_A(\tau)$ to different ranges of stochastic staleness. E.g., it enables tailored treatment of *multi-modal staleness distributions*, which may emerge under hyperthreading or congestion for accessing shared resources. The name TAIL-τ relates to the $1 - F_{\bar{\tau}}(\tau) = \mathbf{P}[\bar{\tau} > \tau]$ component, which is known as the *tail distribution* function.

Practical note. The TAIL-τ function is straight-forward to apply to any implementation of *AsyncSGD*, since it requires only measuring the distribution of the staleness and computing the corresponding CDF $F_{\bar{\tau}}(\tau) = \mathbf{P}[\bar{\tau} < \tau]$, to be used in the step size (C.7). TAIL-τ introduces negligible overhead, as measuring τ is a small, constant-time operation, independently of e.g., architecture size.

Next, we verify that TAIL-τ satisfies properties of the `ASAP.SGD` framework.

Theorem C.4.6. *A* TAIL-τ *function η according to (C.7) is an* `ASAP.SGD` *step size function, according to Definition C.4.4*

The proof of Theorem C.4.6 is the same as of C.4.8 and appears below.

In the following lemma we show several core properties of the TAIL-τ function, to be used in subsequent analysis.

Lemma C.4.7. *For a* TAIL-τ *adaptive step size* $\eta(\tau : \eta^0) = C_A(\tau) \cdot \eta^0$, *we have*

$$\max_\tau \eta(\tau : \eta^0) = (1 + A)\eta^0 \tag{i}$$

$$\min_\tau \eta(\tau : \eta^0) = (1 - A)\eta^0 \tag{ii}$$

$$Var[\eta(\tau)] = (A\bar{\eta})^2/3 \tag{iii}$$

$$C_A(\tau) \leq 1 + (2(m-1)/\tau - 1)\, A \tag{iv}$$

Proof. (i)-(ii) follow directly from Definition C.4.5.

(iii): $Var[\eta(\tau)] = \mathbf{E}\big[\eta(\tau)^2\big] - \bar{\eta}^2$

$$= \sum_{\tau=1}^{\infty} (1 + A(1 - 2F(\tau))\bar{\eta})^2\, p_{\tilde{\tau}}(\tau) - \bar{\eta}^2$$

$$= \left(\frac{1}{6A}\, (1 + A(1 - 2F(\tau)))^3 \,\Big|_{\tau=\infty}^0 - 1 \right) \bar{\eta}^2$$

$$= \left(\frac{1}{6A}\, \big((1 + A)^3 - (1 - A)^3\big) - 1 \right) \bar{\eta}^2 = \frac{1}{3}(A\bar{\eta})^2$$

(iv): $C_A(\tau) = 1 + A(1 - 2F(\tau))$

$$= 1 + A\left(2\left(1 - F(\tau)\right) - 1\right)$$

Markov's inequality now gives

$$C_A(\tau) \leq 1 + A \left(2\frac{\mathbf{E}[\tau]}{\tau} - 1 \right)$$

Now, assuming $\mathbf{E}[\tau] \approx m - 1$, m being the number of threads, concludes the proof. Based on empirical studies, it has been observed that such an assumed condition is reasonable, generally holding in practice (see [24], **Chapter A**). $\square$

Lemma C.4.7 provides useful criteria for deciding the parameters η^0 and A in practice, for ensuring that $\eta(\tau : \eta^0)$ stays within desirable magnitudes suitable for the given problem.

TAIL-τ can be extended in the same spirit as Definition C.4.5, to allow a higher degree of flexibility in design choices, still satisfying the properties of Lemma C.4.7, and enjoying the convergence guarantees established in Section C.5. In particular, the degree with which the CDF of τ influences the scaling factor C_A, and how much variations of τ values are reflected in the step size, can be customized as follows:

Theorem C.4.8. *Let a staleness-adaptive step size function* η *be*

$$\eta(\tau : \eta^0) = C_{s,\phi}(\tau) \cdot \eta^0$$

where the scaling factor is given by

$$C_{s,\phi}(\tau) = 1 + A \cdot (1 - 2\,\phi(F_{\tilde{\tau}}(\tau)))$$

for some amplitude factor $A \in [0, 1]$ *and some non-decreasing function* $\phi \colon [0, 1] \to [0, 1]$. *If* ϕ *satisfies*

$$\int_0^1 \phi(x)\, dx = \frac{1}{2}, \quad \phi(0) = 0, \quad \phi(1) = 1 \tag{C.8}$$

then η *is (i) mean-preserving and (ii) priority-preserving.*

The choice of the function ϕ now allows customizing the rate with which η adapts to different ranges of staleness.

Proof. (i) Mean-preservation follows from

$$\mathbf{E}[C_{s,\phi}(\tau)] = \sum_{\tau=1}^{\infty} \left(1 + A \cdot (1 - 2\ \phi(F_{\tilde{\tau}}(\tau)))\right) p_{\tilde{\tau}}(\tau)$$

$$= 1 + A \cdot \left(1 - 2\sum_{\tau=1}^{\infty} \phi(F_{\tilde{\tau}}(\tau)) p_{\tilde{\tau}}(\tau)\right)$$

Let f extend p so that $f_{\tilde{\tau}}(\rho) = p_{\tilde{\tau}}(\tau)$ for $\rho \in (\tau - 1, \tau)$. Then we have

$$\mathbf{E}[C_{s,\phi}(\tau)] = 1 + A \cdot \left(1 - 2\int_{\tau=0}^{\infty} \phi(F_{\tilde{\tau}}(\tau)) f_{\tilde{\tau}}(\tau)\, d\tau\right)$$

$$= 1 + A \cdot \left(1 - 2\int_0^1 \phi(F_{\tilde{\tau}}(\tau))\, dF_{\tilde{\tau}}(\tau)\right) = 1$$

(ii) Priority-preservation follows directly from that $F_{\tilde{\tau}}(\tau)$ is a CDF, hence non-decreasing, and the assumptions on A and ϕ. $\qquad\square$

C.5   Convergence analysis

In this section we establish asymptotic convergence bounds of the method proposed in the previous section, considering:

Assumption C.5.1. Expected Lipschitz-continuous gradients

$$\mathbf{E}\big[\|\nabla L(\theta^1) - \nabla L(\theta^2)\|\big] \leq \mathcal{L}\mathbf{E}\big[\|\theta^1 - \theta^2\|\big] \ \forall \theta^1, \theta^2 \qquad (C.9)$$

Assumption C.5.2. Expected bounded gradient moment

$$\mathbf{E}\big[\|\nabla L(\theta)\|^2\big] \leq M^2 \ \forall \theta \qquad (C.10)$$

These provide the problem additional structure, hold for a wide set of loss functions in practice, and are widely adopted in the literature [24, 45, 51].

The following lemma shows the expected progression of the SGD iterates for arbitrary staleness-adaptive step sizes.

Lemma C.5.3. *Consider the optimization problem of (1.2), and follow the SGD step (C.3). Let $\eta_i = \eta(\tau_i; \bar{\eta})$ be a staleness-adaptive step size function (according to Definition C.4.1). Then we have the following expected iterative progression:*

$$\mathbf{E}[L(\theta_{i+1}) - L(\theta_i)]$$
$$\leq \mathcal{L}M^2 \left(\mathbf{E}[\eta^2]/2 + \mathbf{E}[\tau\eta]\mathbf{E}[\eta]\right) - \mathbf{E}[\eta]\mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big]$$

Proof. From assumption C.5.1 we have in particular

$$L(\theta_{i+1}) - L(\theta_i) - \langle \nabla L(\theta_i), \theta_{i+1} - \theta_i \rangle \leq \frac{\mathcal{L}}{2}\|\theta_{i+1} - \theta_i\|^2$$

From the SGD step we have

$$L(\theta_{i+1}) - L(\theta_i) - \eta_i \left\langle \nabla L(\theta_i), -\nabla \tilde{L}(\theta_i) + \left(\nabla \tilde{L}(\theta_i) - \nabla \tilde{L}(v_i) \right) \right\rangle$$

$$\leq \frac{\mathcal{L}}{2} \eta_i^2 \|\nabla \tilde{L}(v_i)\|^2$$

$$\Rightarrow L(\theta_{i+1}) - L(\theta_i) + \eta_i \left\langle \nabla L(\theta_i), \nabla \tilde{L}(\theta_i) \right\rangle - \eta_i \|\nabla L(\theta_i)\| \|\nabla \tilde{L}(\theta_i) - \nabla \tilde{L}(v_i)\|$$

$$\leq \frac{\mathcal{L}}{2} \eta_i^2 \|\nabla \tilde{L}(v_i)\|^2$$

We have by assumption C.5.1, and the triangle inequality

$$\|\nabla \tilde{L}(\theta_i) - \nabla \tilde{L}(v_i)\| \leq \mathcal{L} \|\theta_i - v_i\|$$

$$= \mathcal{L} \left\| \sum_{j=1}^{\tau_i} \theta_{i-j+1} - \theta_{i-j} \right\| \leq \mathcal{L} \sum_{j=1}^{\tau_i} \eta_{i-j} \|\nabla \tilde{L}(\theta_{i-j})\|$$

$$\Rightarrow L(\theta_{i+1}) - L(\theta_i) + \eta_i \left\langle \nabla L(\theta_i), \nabla \tilde{L}(\theta_i) \right\rangle - \eta_i \mathcal{L} \|\nabla L(\theta_i)\| \sum_{j=1}^{\tau_i} \eta_{i-j} \|\nabla \tilde{L}(\theta_{i-j})\|$$

$$\leq \frac{\mathcal{L}}{2} \eta_i^2 \|\nabla \tilde{L}(v_i)\|^2$$

Take expectation conditioned on the last staleness τ_i. From mean-independence, we have

$$\mathbf{E}[L(\theta_{i+1}) - L(\theta_i) \mid \tau_i] + \eta_i \mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big]$$

$$- \eta_i \mathcal{L} \sum_{j=1}^{\tau_i} \mathbf{E}[\eta_{i-j}] \mathbf{E}\Big[\|\nabla L(\theta_i)\| \|\nabla \tilde{L}(\theta_{i-j})\|\Big]$$

$$\leq \frac{\mathcal{L}}{2} \eta_i^2 \mathbf{E}\Big[\|\nabla \tilde{L}(v_i)\|^2\Big] \leq \frac{\mathcal{L}}{2} M^2 \eta_i^2$$

Applying Hölder's inequality, and that $\mathbf{E}[\tau_i] = \mathbf{E}[\tau] \ \forall t$

$$\mathbf{E}[L(\theta_{i+1}) - L(\theta_i) \mid \tau_i] + \eta_i \mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big]$$

$$- \eta_i \mathbf{E}[\eta] \mathcal{L} \sum_{j=1}^{\tau_i} \sqrt{\mathbf{E}[\|\nabla L(\theta_i)\|^2] \mathbf{E}\Big[\|\nabla \tilde{L}(\theta_{i-j})\|^2\Big]}$$

$$\leq \frac{\mathcal{L}}{2} \eta_i^2 \mathbf{E}\Big[\|\nabla \tilde{L}(v_i)\|^2\Big] \leq \frac{\mathcal{L}}{2} M^2 \eta_i^2$$

$$\Rightarrow \mathbf{E}[L(\theta_{i+1}) - L(\theta_i) \mid \tau_i] + \eta_i \mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big] - \tau_i \eta_i \mathbf{E}[\eta] \mathcal{L} M^2 \leq \frac{\mathcal{L}}{2} M^2 \eta_i^2$$

Now, take full expectation

$$\mathbf{E}[L(\theta_{i+1}) - L(\theta_i)] + \mathbf{E}[\eta] \mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big] - \mathbf{E}[\tau\eta] \mathbf{E}[\eta] \mathcal{L} M^2 \leq \frac{\mathcal{L}}{2} M^2 \mathbf{E}\big[\eta^2\big]$$

which concludes the proof. $\square$

The following lemma establishes the expected iterative progression of *Async-SGD* with `ASAP.SGD` step sizes:

Corollary C.5.4. *Under the same conditions as Lemma C.5.3, let in addition* $\eta(\tau_i; \bar{\eta})$ *be mean- and priority-preserving, hence an* `ASAP.SGD` *step size. Then we have:*

$$\mathbf{E}[L(\theta_{i+1}) - L(\theta_i)] \leq \mathcal{L}M^2 \left(\mathbf{E}[\eta^2]/2 + \bar{\eta}^2\bar{\tau}\right) - \bar{\eta}\mathbf{E}[\|\nabla L(\theta_i)\|^2]$$

Corollary C.5.4 follows directly from mean-preservation, and from that $\mathbf{E}[\tau\eta] \leq \mathbf{E}[\tau]\mathbf{E}[\eta]$ by priority-preservation. Corollary C.5.4 serves as a common starting point for convergence analysis of a wide range of `ASAP.SGD` step size functions. Using the specifics of the step size function, lemma C.5.3 can be used to derive explicit convergence bounds, as we demonstrate in the following for TAIL-τ.

Corollary C.5.5. *Assume the conditions of Lemma C.5.3 and let* $\eta(\tau_i; \bar{\eta}) = C_A(\tau) \cdot \bar{\eta}$ *be a* TAIL-τ *step size function. Then:*

$$\mathbf{E}[L(\theta_{i+1}) - L(\theta_i)] \leq \mathcal{L}M^2\bar{\eta}^2 \left(A^2/6 + \bar{\tau}\right) - \bar{\eta}\mathbf{E}[\|\nabla L(\theta_i)\|^2]$$

Corollary C.5.5 follows from Corollary C.5.4, utilizing the TAIL-τ properties from Lemma C.4.7, and shows the iterative improvement for TAIL-τ, to be used in subsequent results.

In the following theorem we establish expected time until convergence to an approximate critical point of SGD for general non-convex, smooth functions, using TAIL-τ.

Theorem C.5.6. *Assume* $L(\theta_0) - L(\theta^*) < \delta$, *and let* $\eta(\tau_i; \bar{\eta}) = C_A(\tau) \cdot \bar{\eta}$ *be a* TAIL-τ *step size function. Then we have a* $\mathcal{O}(1/\sqrt{T})$ *convergence bound, to an approximate critical point, after* T *AsyncSGD iterations. Specifically, if*

$$\bar{\eta} = \sqrt{\delta/\mathcal{L}M^2T(A^2 + \bar{\tau})} \tag{C.11}$$

then $\min_t \mathbf{E}[\|\nabla L(\theta_i)\|^2] \leq 2\sqrt{\mathcal{L}M^2(A^2 + \bar{\tau})\delta/T}$

Proof. From Corollary C.5.5, we have

$$\mathbf{E}[\|\nabla L(\theta_i)\|^2] \leq \frac{\mathbf{E}[L(\theta_i) - L(\theta_{i+1})]}{\bar{\eta}} + \mathcal{L}M^2\bar{\eta}\left(\frac{A^2}{6} + \bar{\tau}\right)$$

$$\Rightarrow \frac{1}{T}\sum_{t=0}^{T-1} \mathbf{E}[\|\nabla L(\theta_i)\|^2] \leq \frac{\delta}{\bar{\eta}T} + \mathcal{L}M^2\bar{\eta}\left(\frac{A^2}{6} + \bar{\tau}\right)$$

which implies in particular that $\min_i \mathbf{E}[\|\nabla L(\theta_i)\|^2]$ satisfies the above upper bound as well. The bound now rewrites as in the statement by substituting $\bar{\eta}$ for (C.11), which concludes the proof. $\square$

Theorem C.5.6 shows that TAIL-τ recovers the standard bound $\mathcal{O}(1/\sqrt{T})$ of SGD on smooth non-convex problems [99]. However, this is rather pessimistic as it applies to general non-convex functions. We consider the following to provide more structure to the problem:

Assumption C.5.7. Polyak-Lojasiewicz (PL) condition. A function L is referred to as μ-PL if, for some $\mu > 0$:

$$\mathbf{E}\big[\|\nabla L(\theta)\|^2\big] \geq \mu \, \mathbf{E}[L(\theta) - L(\theta^*)] \; \forall \theta \tag{C.12}$$

PL is a geometric condition characterizing the shape of non-convex functions. It can be regarded as a generalization of strong convexity, however without requirements on e.g., uniqueness of minimizers. Several ML loss functions satisfy the condition, including least squares, logistic regression, support vector machines [62] and certain types of deep ANNs [63].

Next, we establish asymptotic convergence of *AsyncSGD* with TAIL-τ step size function for smooth, μ-PL functions.

Theorem C.5.8. *Let L be μ-PL, and $L(\theta_0) - L(\theta^*) < \delta$. Further, let $\eta(\tau_i; \bar{\eta}) = C_A(\tau) \cdot \bar{\eta}$ be a* TAIL-τ *step size function. Then we have expected ϵ-convergence in $T = \mathcal{O}\big(\frac{1}{\epsilon} \log\big(\frac{2\delta}{\epsilon}\big)\big)$ iterations. More precisely, let*

$$\bar{\eta} = \frac{\mu\epsilon}{\mathcal{L}M^2(A^2/3 + 2\bar{\tau})} \tag{C.13}$$

Then we have $\mathbf{E}[L(\theta_T) - L(\theta^*)] < \epsilon$ for

$$T > \frac{\mathcal{L}M^2(A^2/3 + 2\bar{\tau})}{\mu^2\epsilon} \log\left(\frac{2\delta}{\epsilon}\right)$$

Proof. With Corollary C.5.5 as a starting point, we have by assumption C.5.7

$$\mathbf{E}[L(\theta_{i+1}) - L(\theta_i)] + \bar{\eta}\mu\mathbf{E}[L(\theta_i) - L(\theta^*)] \leq \mathcal{L}M^2\bar{\eta}^2\left(\frac{A^2}{6} + \bar{\tau}\right)$$

$$\Rightarrow \mathbf{E}[L(\theta_T) - L(\theta^*)] \leq (1 - \bar{\eta}\mu)\mathbf{E}[L(\theta_{T-1}) - L(\theta^*)] + \mathcal{L}M^2\bar{\eta}^2\left(\frac{A^2}{6} + \bar{\tau}\right)$$

$$= (1 - \bar{\eta}\mu)^T \delta + \mathcal{L}M^2\bar{\eta}^2 \sum_{i=0}^{T-1}(1 - \bar{\eta}\mu)^i\left(\frac{A^2}{6} + \bar{\tau}\right)$$

$$\leq (1 - \bar{\eta}\mu)^T \delta + \frac{\mathcal{L}M^2\bar{\eta}}{\mu}\left(\frac{A^2}{6} + \bar{\tau}\right) < \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon$$

$\square$

The reason for the $O(A^2)$ term in Theorems C.5.6 and C.5.8 (negligible compared to the $O(\tau)$ additive term) is due to the analytical estimation of the variance of TAIL-τ (Lemma C.4.7, (iii)), which must be considered (Cor. C.5.5), specifically when expanding the $E[\eta^2]$ term. As we make no assumptions on PDF(τ), the bounds reflect the expected worst-case convergence over all possible τ distributions. Additional information on the τ distribution can (using Corollary C.5.4 as a starting point) derive tighter algorithm-specific bounds.

For the sake of self-containment, we establish fundamental convex convergence bounds for (i) arbitrary staleness-adaptive step size functions, (ii) ones satisfying the `ASAP.SGD` properties, as well as (iii) the TAIL-τ function. For this, we will require also strong convexity:

Assumption C.5.9. L is strongly convex with parameter $\mathcal{C}$

$$\mathbf{E}\left[(\theta^1 - \theta^2)^T \left(\nabla L(\theta^1) - \nabla L(\theta^2)\right)\right] \geq \mathcal{C}\|\theta^1 - \theta^2\|^2 \ \forall \theta^1, \theta^2$$

Theorem C.5.10. *Consider the unconstrained optimization problem of (1.2). Under Assumptions C.5.1, C.5.2, and C.5.9, for any precision $\epsilon > 0$, and for any staleness-adaptive step size function η (Definition C.4.1), there is a number T of* AsyncSGD *iterations, of the form (C.3) such that $\mathbf{E}[\|\theta_T - \theta^*\|^2] < \epsilon$, where T is bounded by:*

$$T \leq \frac{\ln\left(\|\theta_0 - \theta^*\|^2 \epsilon^{-1}\right)}{2\left(\mathcal{C} - \mathcal{L}M\epsilon^{-1/2}\mathbf{E}[\tau\eta]\right)\bar{\eta} - \epsilon^{-1}M^2\mathbf{E}[\eta^2]}$$

Corollary C.5.11. *Under the same conditions as Theorem C.5.10, let η_i be* ASAP.SGD *adaptive step size function. Then we have the following bound on the expected number of iterations until expected convergence:*

$$T \leq \frac{\ln\left(\|\theta_0 - \theta^*\|^2 \epsilon^{-1}\right)}{2\mathcal{C}\bar{\eta} - \epsilon^{-1}M\left(M + 2\mathcal{L}\sqrt{\epsilon}\bar{\tau}\right)\mathbf{E}[\eta^2]} \tag{C.14}$$

Corollary C.5.12. *Under the same conditions as Theorem C.5.10, let additionally η_i be* TAIL-τ *step size function. Then we have the following bound on the expected number of iterations until expected convergence:*

$$T \leq \frac{\ln\left(\|\theta_0 - \theta^*\|^2 \epsilon^{-1}\right)}{2\mathcal{C}\bar{\eta} - \epsilon^{-1}M\left(M + 2\mathcal{L}\sqrt{\epsilon}\bar{\tau}\right)\left(1 + \frac{A^2}{3}\right)\bar{\eta}^2} \tag{C.15}$$

The proofs for Theorem C.5.10, and Corollary C.5.11, C.5.12 build on and extend the results in **Chapter A** and follow from the properties of ASAP.SGD and TAIL-τ, specified in Section C.4.

C.6 Evaluation

We complement the analysis with benchmarking the TAIL-τ function, for implementations of *AsyncSGD* representative of a variety of system-execution properties relating with scheduling and ordering. We evaluate TAIL-τ, comparing to standard constant step size executions, for relevant DL benchmark applications, namely training the LeNet [100] architecture, as well as a 3-layer MLP, for image recognition for image recognition on both MNIST and Fashion MNIST. The evaluation focuses on convergence rates, primarily wall-clock time to ϵ-convergence (which is the most relevant in practice), as well as number of successful executions, for various precision levels ϵ. We include a broad spectrum of parallelism, giving a detailed picture of the capability of the methods to scale in practice. We also study the staleness distributions, the adaptive response of the TAIL-τ function, and its impact on the convergence.

Implementation.   We evaluate the TAIL-τ for three *AsyncSGD* algorithms, representing fundamentally different synchronization mechanisms and guarantees on progress and consistency, and in this way capturing a variety of scheduling and ordering system-execution properties: (i) lock-based *AsyncSGD* [22,51,93] (ii) lock-free, but inconsistent, HOGWILD! [20,44,45] and (iii) the

lock-free and consistent *Leashed-SGD* algorithm, introduced in **Chapter B**, denoted ASYNC, HOG and LSH, respectively. Executions using our TAIL-τ function are indicated by the suffix _TAIL. The implementation extends the open *Shared-Memory-SGD* [101] C++ library, connecting ANN operations to low-level implementations of parallel SGD, and is free to use for research purposes.

Experiment setup. We tackle the problem of ANN training for image classification on the datasets MNIST [89] of hand-written digits, CIFAR-10 [70] of everyday objects, and Fashion-MNIST [102] of clothing article images. All datasets contain 60k images, each belonging to one of ten classes $\in \{0, \ldots, 9\}$. For this, we train a LeNet CNN architecture, as well as a 3-layer MLP, with 128 neurons per layer (denoted MLP in the following), for 100 epochs. We use standard settings and hyper-parameters; For MNIST and Fashion-MNIST training we use a base step size of $\eta_0 = 1\mathrm{e}{-4}$ and mini-batch size 128, while for CIFAR-10 we use $\eta_0 = 5\mathrm{e}{-3}$ and a mini-batch size of 16. The *multi-class cross-entropy* loss function is used in all experiments. For *Leashed-SGD*, we use the default setting of an infinite persistence bound. We use a TAIL-τ step size function (as in Definition C.4.5), that adapts to each unique execution, based on the measured staleness distribution, with an adaptation amplitude of $A = 1$, due to its role in emphasizing fresh updates and dampening stragglers. The experiments are conducted on a 2.10 GHz Intel(R) Xeon(R) E5-2695 two-socket 36-core (18 cores per socket, each supporting two hyper-threads), 64GB non-uniform memory access (NUMA), Ubuntu 16.04 system. Plots show averaged values from 5 executions for each setting, unless otherwise stated. ϵ-convergence is achieved when $L(\theta) < \epsilon$, expressed as % of the initial loss $L(\theta_0)$. The number of executions that fail to reach ϵ-convergence is indicated by ∞ at the top, if such executions occurred. This is important, since such executions result in models of insufficient accuracy, and thereby are wasted work.

Convergence speedup and scalability. We measure convergence time to first 50% of the initial error (i.e., 50%-convergence), and then to higher precision (5% for MNIST, and 15% for Fashion-MNIST to enable clearer comparison, since baselines rarely converge to this level of precision) across the parallelism spectrum. The results (Figure C.2) show that the TAIL-τ step size function yields persistent and substantial speedup in convergence time (12% in the worst-case, 100% in the best), for all combinations of datasets, architectures and *AsyncSGD* implementations (see Table C.2 for details). Training plots, showing loss progression over time, are shown in Figure C.3, demonstrating the convergence speed being *orders of magnitudes* faster than standard *AsyncSGD*. Similar speedup is observed for training on the CIFAR-10 dataset, shown in Figure C.5. For higher-precision convergence, non-converging executions are frequent among the standard *AsyncSGD* algorithms, especially under higher parallelism. Their ability to converge varies, demonstrating the impact of underlying synchronization and progress guarantees. We observe in particular that HOGWILD! achieves the fastest overall convergence, while *Leashed-SGD* provides higher reliability, i.e., lower risk of non-convergence. However, independently of the properties of the *AsyncSGD* implementation, we observe, in addition to persistently faster convergence, also that the TAIL-τ step size ensures a significantly lower risk of non-convergence, hence higher reliability (see Table C.2). Staleness-based dampening according to the commonly adopted $\times \tau^{-1}$
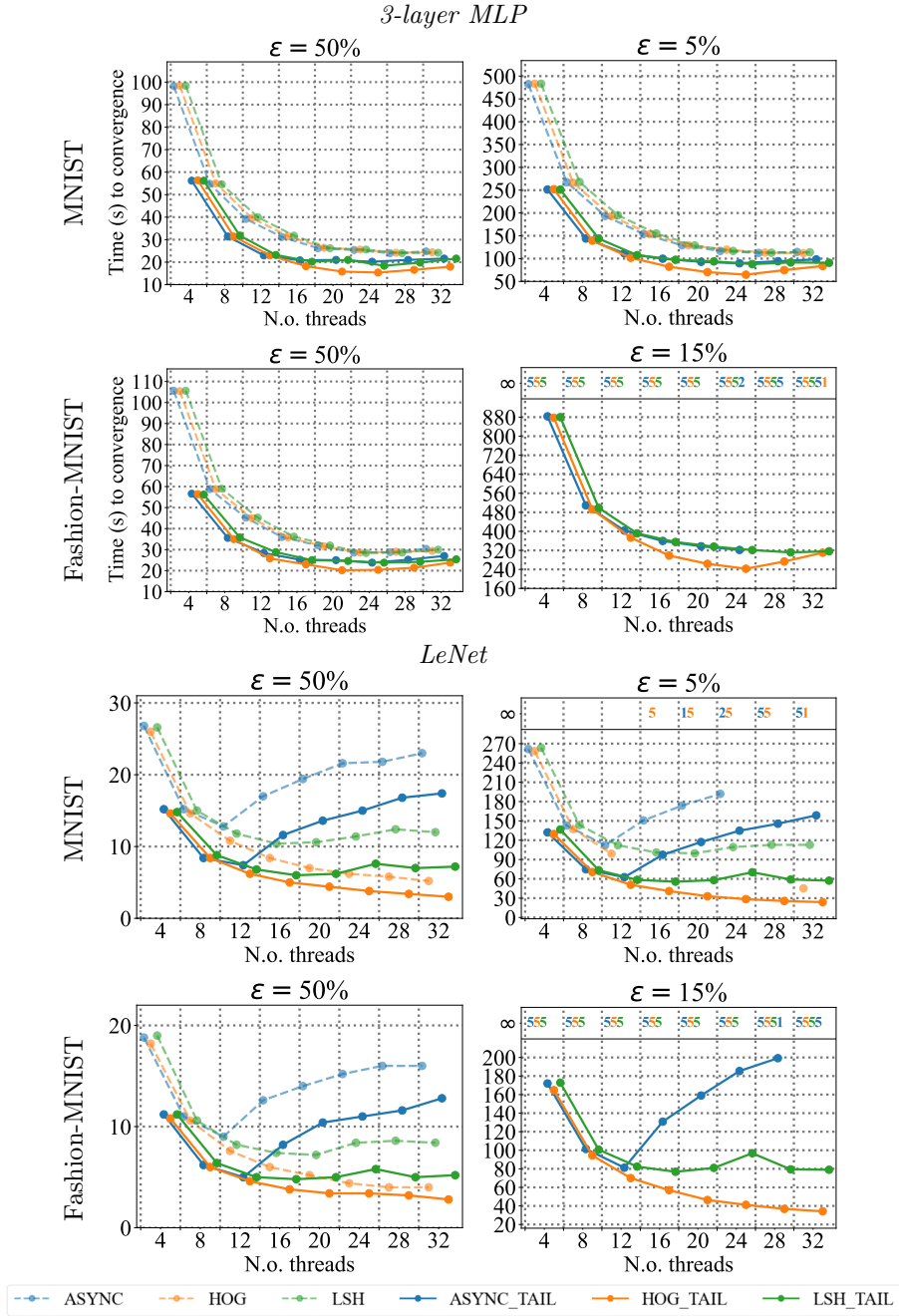
Figure C.2:   Convergence rates for LeNet (*left*) and a 3-layer MLP (*right*) for MNIST and Fashion-MNIST recognition training with *AsyncSGD*, with HOGWILD! (HOG), *Leashed-SGD* (LSH), and traditional lock-based (ASYNC) implementations. Executions using the instance-based TAIL-τ staleness-adaptive step size are indicated with the suffix _TAIL.
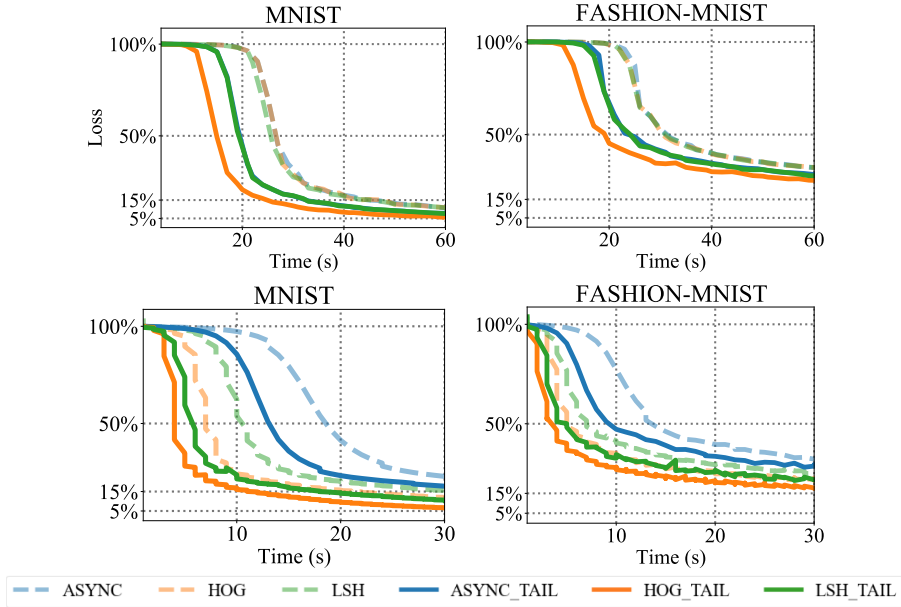
Figure C.3: Loss over time for Hogwild! (HOG), *Leashed-SGD* (LSH), and traditional lock-based (ASYNC) *AsyncSGD* with parallelism $m = 20$, with and without the TAIL-τ staleness-adaptive step size (suffix _TAIL) for LeNet (*left*) and a 3-layer MLP (*right*).

scheme, as well as the FLeet [3] exponential dampening [56], are evaluated for MLP and LeNet training on MNIST, compared to standard *AsyncSGD* with constant step size. The results (Figure C.6, C.7) show, as conjectured in Section C.3, convergence of significantly slower rate compared to a constant step size, which in addition (as opposed to TAIL-τ, as well as constant step size) decays with higher parallelism due to the reduced overall step size magnitude. The speedup of TAIL-τ compared to constant step size is shown in Figure C.2 and C.3. Figure C.11, C.6-C.7 show convergence rates and training plots under varying parallelism for the $\times\tau^{-1}$ scheme which, with some variation, appears often in previous works [24, 53, 61], compared to standard *AsyncSGD* with constant step size. For both LeNet and MLP training for MNIST recognition, we observe significant challenges with achieving convergence rates within the same order of magnitude as traditional, constant step size, *AsyncSGD*. This, in contrast with TAIL-τ which, as shown in Section C.6, provides persistent, and significant, speedup in convergence rates. The straight-forward $\times\tau^{-1}$ step size suffers significant challenges in achieving convergence, especially under higher parallelism, as explained in Section C.3.

Instance-based adaptiveness. Figure C.4 shows the staleness distribution of the *AsyncSGD* algorithms for LeNet and MLP, along with the corresponding

---

[3]As in the original paper, we use the dampening factor $\Lambda(\tau) = e^{-\beta\tau}$, where β satisfies $(\tau_{\text{thres}}/2 + 1)^{-1} = e^{-\beta\tau_{\text{thres}}/2}$, where τ_{thres} is the beginning of the staleness distribution *tail*, which we consider to be $\tau_{\text{thres}} = m + 1$ here, based on observation. The similarity-based boosting option was not considered here.
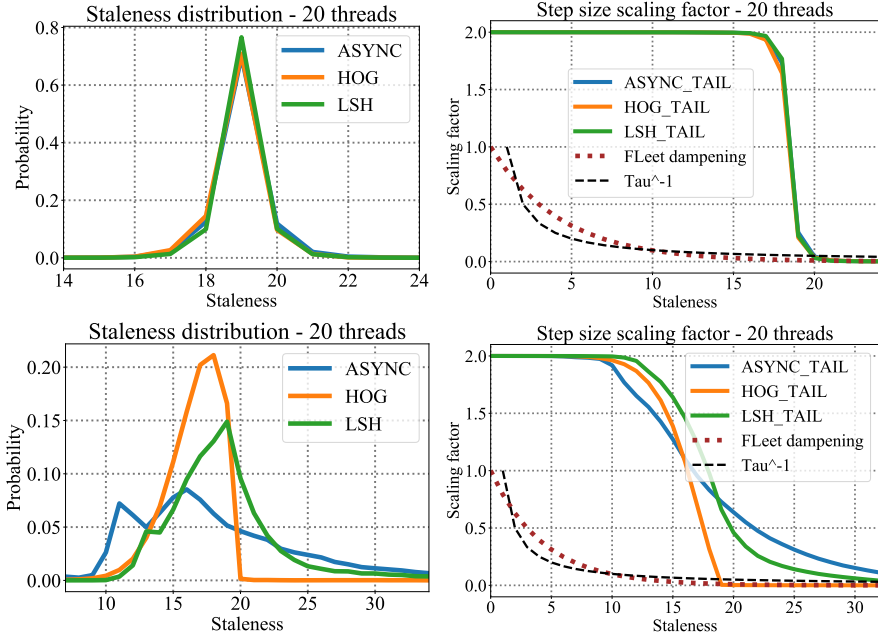
Figure C.4: Staleness distribution (*left*), and the TAIL-τ scaling factor (*right*) for LeNet (*top*) and a 3-layer MLP (*bottom*), for MNIST and Fashion-MNIST.
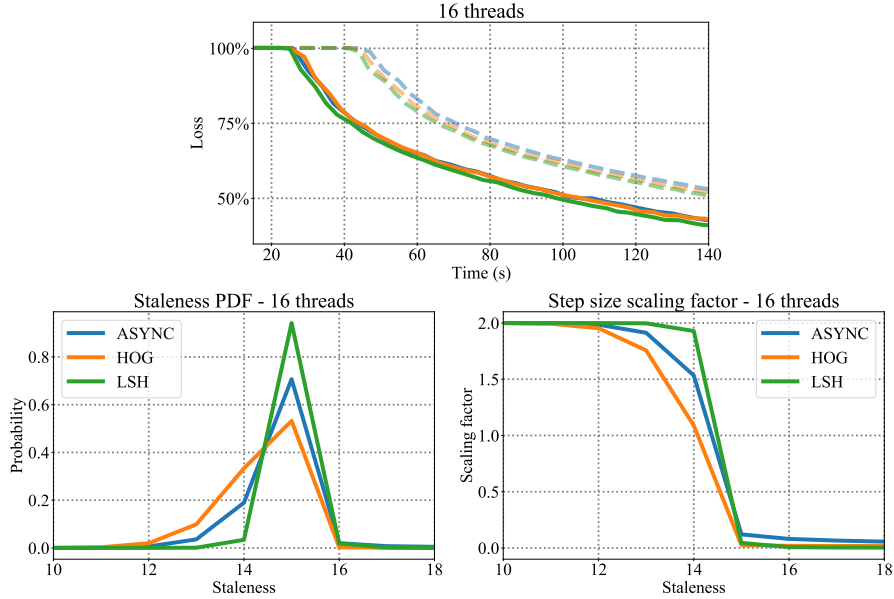

Figure C.5: Loss over time (*top*), staleness distribution (*bottom left*), and the TAIL-τ scaling factor (*bottom right*) for LeNet training on CIFAR-10.
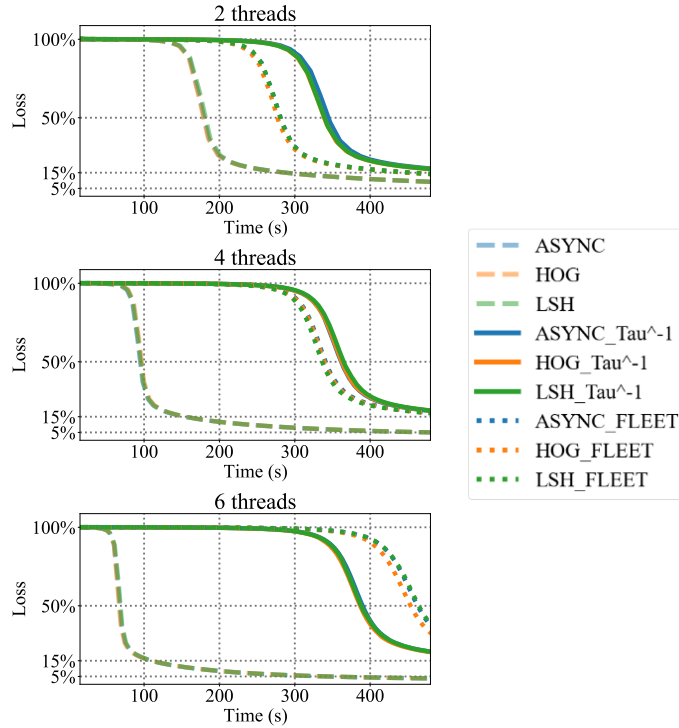
Figure C.6:   LeNet training with *AsyncSGD* on MNIST with Hogwild! (HOG), *Leashed-SGD* (LSH), and traditional lock-based (ASYNC) implementations, comparing executions using the FLeet exponential dampening approach (suffix: _FLEET) and the $\times \tau^{-1}$ staleness-adaptive scheme (suffix: _Tau^{-}1) against standard, constant step size.

TAIL-τ step size scaling factors, generated dynamically based on the staleness distribution of the particular execution. We observe, as expected from Section C.4, that the staleness-adaptive step size strategies, generated by TAIL-τ, emphasize fresh updates and diminishe the impact of stale ones, taking into consideration the underlying execution-specific staleness distribution. This, as shown above, results in increased stability, i.e., lowered risk of non-convergence, as well as significant increase in convergence rates.   Note that considering standard SGD eliminates side-effects of 'add-ons', enabling clearer comparisons. Other step-size-altering methods (e.g., ADAM or schedules) can be used in conjunction, by applying them to η_0 (Def. C.4.4) and is a relevant target for future studies.

Higher parallelism.   Figure C.8 provides an overview of the scalability of all algorithms evaluated here for MNIST and Fashion-MNIST, respectively, over a large parallelism spectrum. We observe speedup until around 32 threads, at which the system saturates and does not benefit from higher parallelism. The different algorithms perform differently at different parallelism levels. In particular, under hyper-threading ($m > 36$), the *AsyncSGD* implementations suffer a slower convergence due to increased computational overhead and asynchrony-induced noise. However, in all instances, TAIL-τ provides significant speedup, independently of the algorithm, dataset, and parallelism level. Figure C.9 and
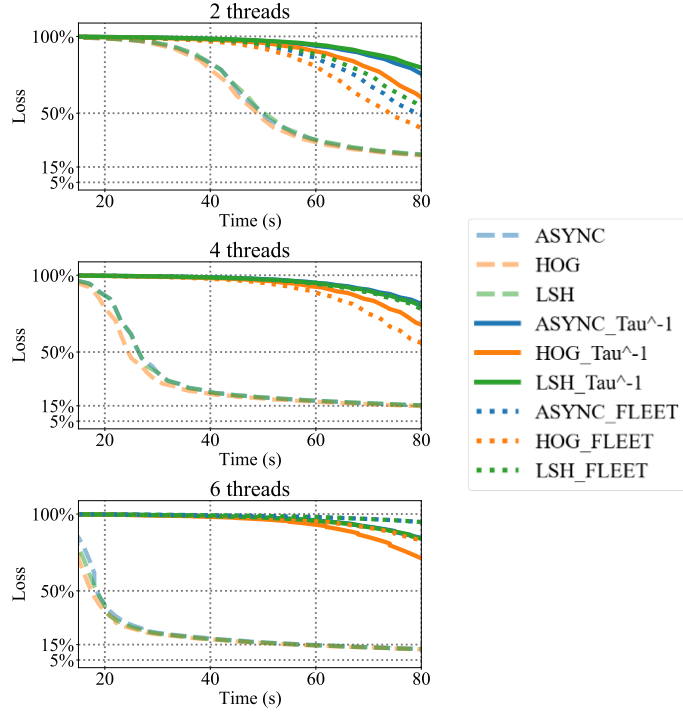
Figure C.7: MLP training with *AsyncSGD* on MNIST with Hogwild! (HOG), *Leashed-SGD* (LSH), and lock-based (ASYNC) implementations, comparing executions using the FLeet exponential dampening approach (suffix: _FLEET) and the $\times\tau^{-1}$ staleness-adaptive scheme (suffix: _Tau^{-}1) against constant step size.

C.10 show staleness distributions of the considered *AsyncSGD* algorithms for LeNet and MLP training, respectively, together with the corresponding scaling factors C_A of the staleness-adaptive TAIL-τ step size functions. The *AsyncSGD* implementations have fundamentally different staleness distributions, due to the underlying algorithmic mechanisms for progress and consistency. Moreover, we observe multi-modality in some executions, in particular for lock-based *AsyncSGD* due congestion about the locks, and under hyper-threaded parallelism ($m > 36$). The emergence of multi-modal staleness distributions in the presence of hyper-threading is due to that threads are mapped to the same core, and need to share that computing resource. This results in a subset of threads that share computing cores pair-wise, and another consisting of threads with exclusive access to such cores. The threads of the former subset will naturally compute at about half the speed as the threads of the latter portion, resulting particularly in a bimodal staleness distribution with modes at around (i) the number of cores and (ii) the maximum number of hyper-threads, as can be observed in Figure C.10.

The execution-specific staleness distributions are utilized by TAIL-τ to generate an instance-based adaptiveness strategy, accommodating for algorithmic differences and underlying system aspects, enabling the improvements in convergence rates and stability that are observed in Section C.6.
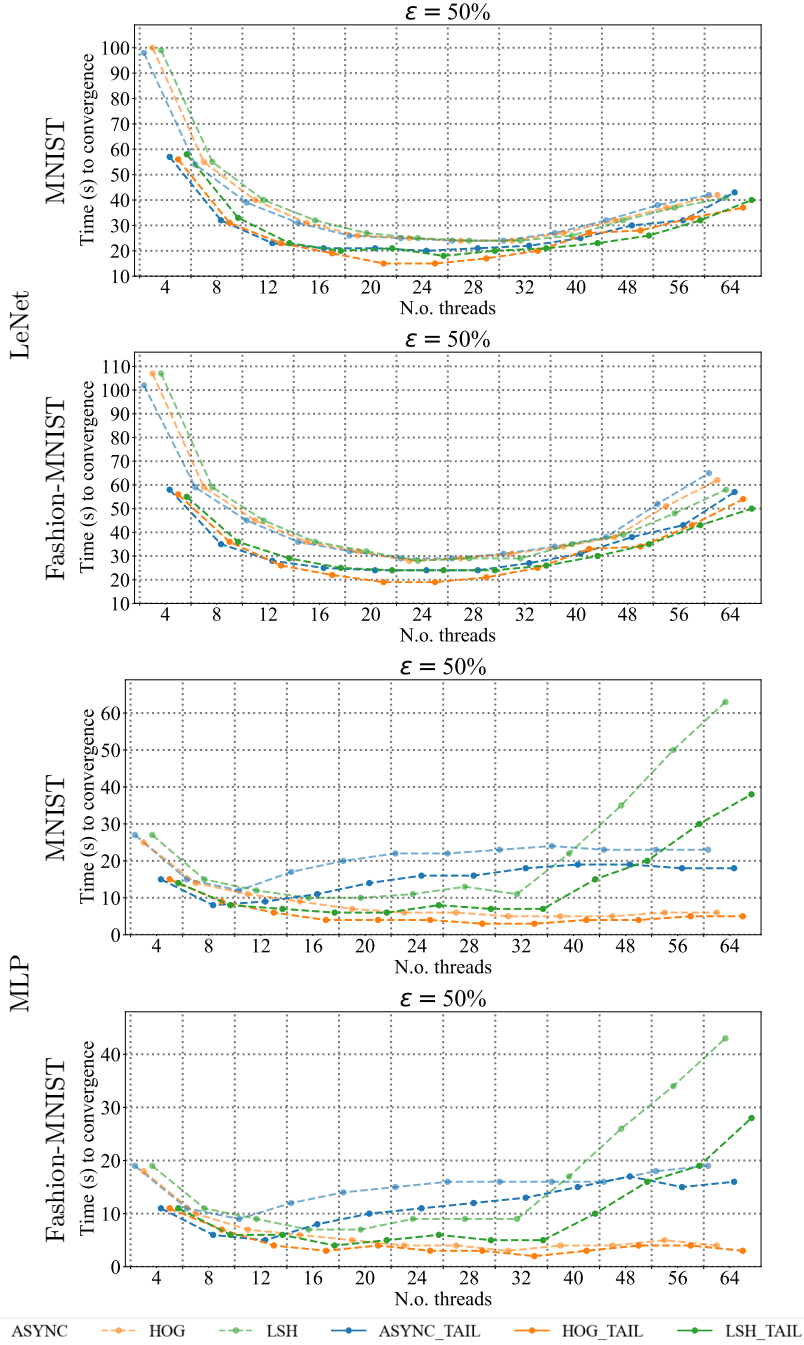
Figure C.8:   *AsyncSGD* convergence rates over a wide parallelism spectrum, including hyper-threading ($m > 36$), for HOGWILD! (HOG), *Leashed-SGD* (LSH), and lock-based (ASYNC), comparing executions using the staleness-adaptive TAIL-τ (suffix: _TAIL) against standard, constant step size.

Figure C.9: Staleness distributions for the considered *AsyncSGD* algorithms (*left*) and the corresponding scaling factors C_A of the generated staleness-adaptive TAIL-τ step sizes, as in Definition C.4.1 (*right*) for LeNet training on MNIST and Fashion-MNIST.

System-related insights. The emerging τ distribution of an AsyncSGD execution is influenced by many underlying factors, including (i) the compute infrastructure (UMA/NUMA, hyper-threading), (ii) consistency guarantees, and the associated synchronization mechanisms (e.g., loose consistency as in HOGWILD!, lock-based, or *Leashed-SGD* etc.), (iii) gradient computation vs application time, (iv) number of threads. Several works are dedicated to studying this [28], explored also in **Chapter B**, however the nature of this dependency is an open problem. Although this is not the main scope here, we see that implicitly adapting to the aforementioned aspects, through the PDF(τ) signature, yields significant practical benefits, as we show with TAIL-τ.
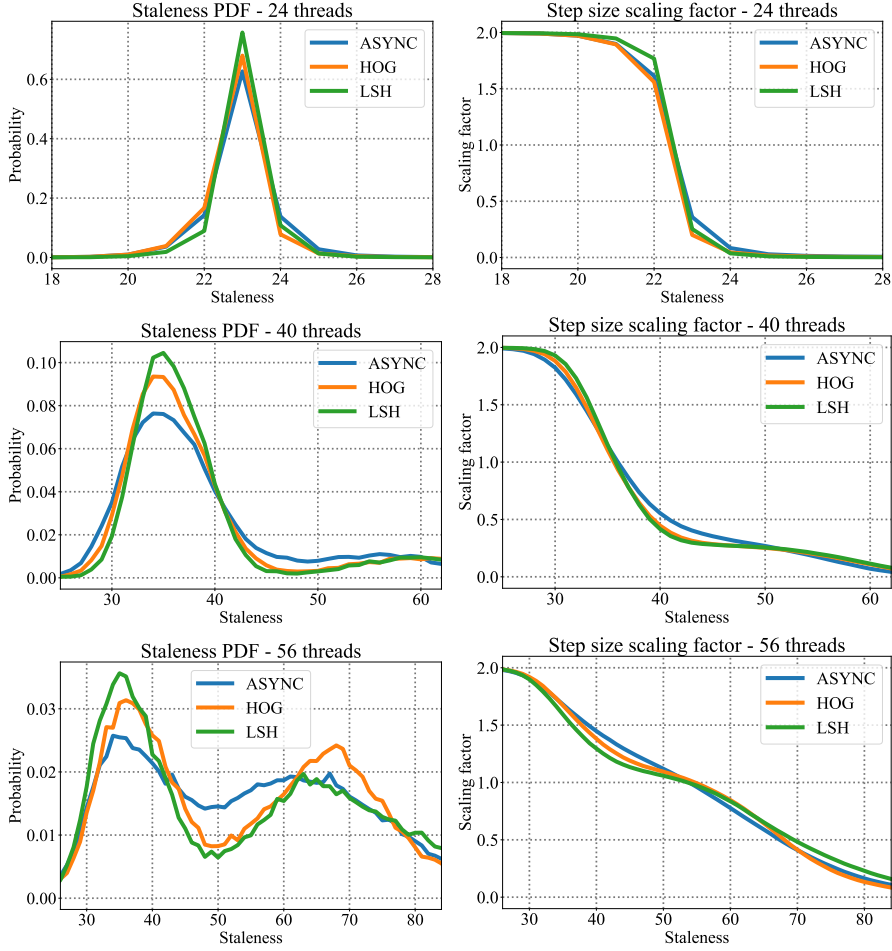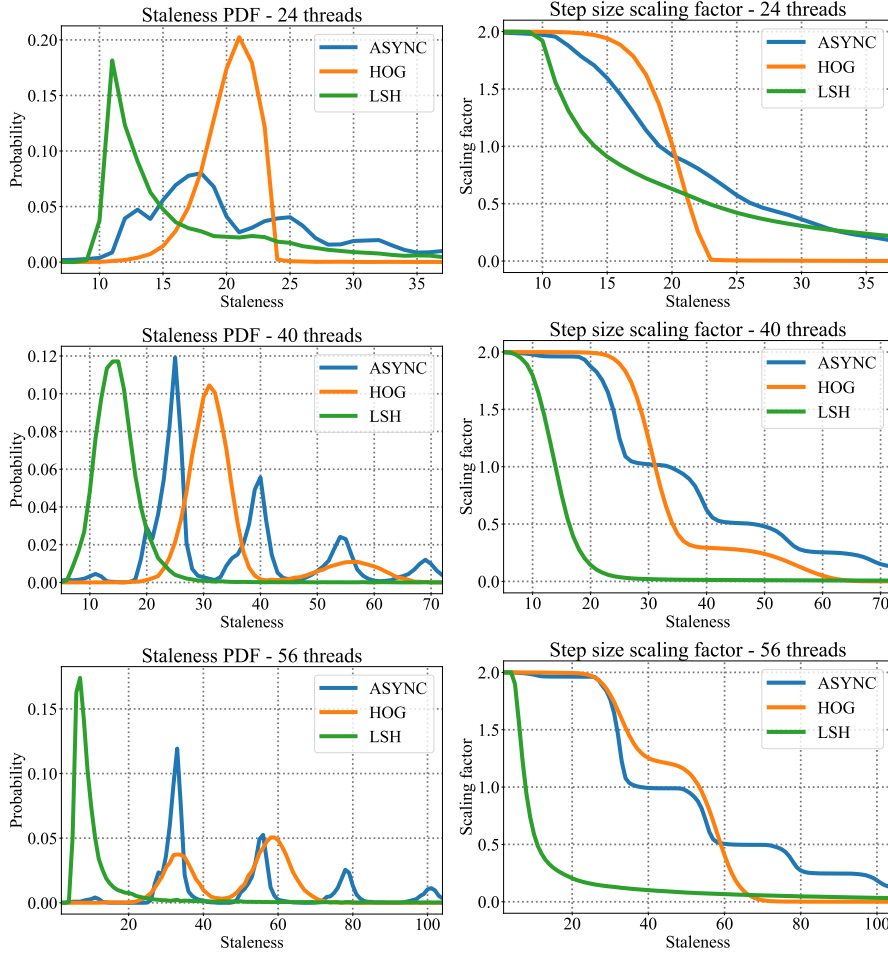
Figure C.10:   Staleness distributions for the considered *AsyncSGD* algorithms (*left*) and the corresponding scaling factors C_A of the generated staleness-adaptive TAIL-τ step sizes, as in Definition C.4.1 (*right*) for MLP training on MNIST and Fashion-MNIST.

Table C.2: Results overview - speedup across the parallelism spectrum achieved by the TAIL-τ step size, relative a standard constant one.

Speedup		*50%-convergence*				*5% (15%)-convergence*			
Dataset	*Architecture*	min	max	avg	success*	min	max	avg	success*
MNIST	LeNet	1.12	1.75	1.51	1.0	1.16	1.92	1.6	1.0
	MLP	1.30	2.0	1.66	1.0	1.42	1.99	1.82	3.53
Fashion-MNIST	LeNet	1.13	1.90	1.48	1.0	∞	∞	∞	∞
	MLP	1.25	1.8	1.56	1.0	∞	∞	∞	∞
CIFAR-10	LeNet	1.03	1.45	1.29	1.0	-	-	-	-

*Ratio between n.o. executions that reached the desired precision for TAIL-τ executions vs. standard *AsyncSGD*
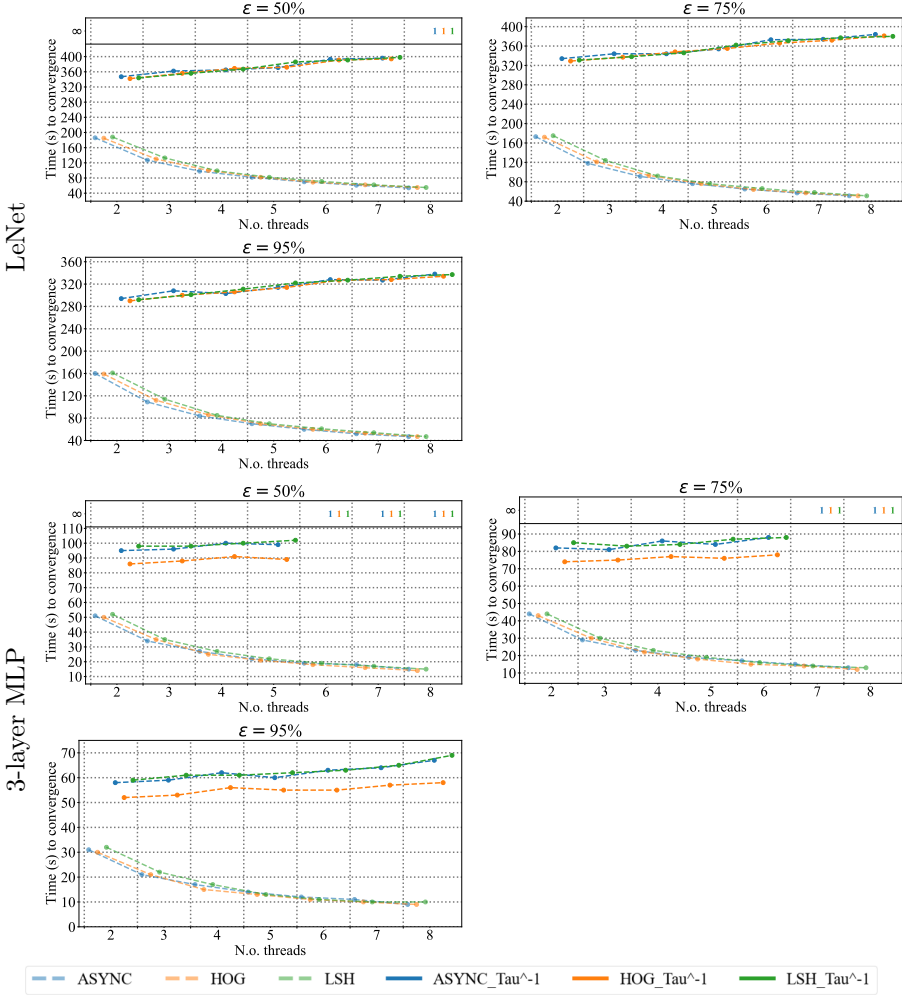
Figure C.11: Convergence rates on MNIST for LeNet and a 3-layer MLP with *AsyncSGD*, with HOGWILD! (HOG), *Leashed-SGD* (LSH), and traditional lock-based (ASYNC) implementations, comparing executions using the $\times \tau^{-1}$ staleness-adaptive scheme (suffix: _Tau∧-1) against standard, constant step size.

C.7   Conclusion

We introduce `ASAP.SGD` - a framework for capturing essential properties of general staleness-adaptive step size functions for $AsyncSGD$, providing structure to the domain of staleness-adaptiveness, and can guide the design of new adaptive step size strategies. Within `ASAP.SGD`, we introduce the first instance-based dynamic step size function, TAIL-τ, which generates a tailored adaptiveness strategy for each unique execution. We analyze general `ASAP.SGD` functions for $AsyncSGD$, as well as TAIL-τ in particular, and recover convergence bounds for both convex and non-convex problems, as well as establish new bounds for ones satisfying the Polyak-Lojasiewicz condition.

We implement TAIL-τ, extending existing $AsyncSGD$ implementations, to provide a platform for further research in the domain. The evaluation covers three implementations of $AsyncSGD$, with fundamentally different algorithmic properties, for training LeNet and an MLP for image recognition on MNIST and Fashion-MNIST. The results show that TAIL-τ is a vital component for $AsyncSGD$ practical deployments, due to its ability to, based on the properties of the unique execution, generate an adaptiveness strategy tailored to the specific execution, yielding persistent speedup across the entire parallelism spectrum, and tremendous increase in reliability to converge, in particular to high precision. Efficiency is important not only for timeliness, but also for resource utilization, especially when considering highly energy-consuming ANN training [103].

Chapter D

Adaptation of the article:

Less is more: Elastic Parallelism Control for Asynchronous SGD

K. Bäckström, M. Papatriantafilou, P. Tsigas

Abstract

Parallel algorithms for Stochastic Gradient Descent (SGD) have gained significant interest due to their speed-up capabilities. In particular, Asynchronous SGD ($AsyncSGD$) has become a standard part in most modern deep learning (DL) frameworks and associated applications. However, contrary to traditional Synchronous parallelization, $AsyncSGD$ breaks the original SGD semantics, since the applied gradients are based on a stale state of the model parameters. This gives rise to a well-known phenomenon, referred to as *asynchrony-induced noise* (AIN). While $AsyncSGD$ is unmatched in computational efficiency, the presence of staleness and AIN impedes the statistical convergence properties of the execution. The trade-off that emerges requires careful tuning of the parallelism level m; under-parallelism implies unnecessarily slow executions, and over-parallelism causes non-convergence, oscillations, and crashes, all the while consuming excessive computational resources at a higher energy consumption.

We show that the optimal parallelism is not constant but varies throughout the execution. We present $ElAsyncSGD$, an elastic extension of $AsyncSGD$, which frees and deploys workers dynamically during the execution to balance the trade-off between computational and statistical efficiency. In addition, we introduce an explicit definition of AIN, and an efficient algorithmic implementation to measure it in real-time. We establish novel convergence bounds for non-convex problems, showing in particular the influence of AIN and m on the convergence of $AsyncSGD$. The extensive evaluation of $ElAsyncSGD$ in the paper, on several relevant DL benchmark datasets, architectures and synchronization methods, shows improvements in convergence speed and stability, as well as crucial savings in computational resources, with reductions between 30-67%. The latter is particularly useful, given the increasing awareness of the need to utilize efficiently resources and energy on tasks of this type.

D.1   Introduction

The interest in concurrent algorithms for Stochastic Gradient Descent (SGD) is at an all-time high, due to their critical role in accelerating Machine Learning (ML), particularly Deep Learning (DL), applications.

SGD is a first-order iterative optimization algorithm, which follows:

$$\theta_{i+1} = \theta_i - \eta \nabla \tilde{L}(\theta_i) \tag{D.1}$$

given an optimization problem

$$\underset{\theta \in \mathbb{R}^d}{\text{minimize}} \quad L_D(\theta) \tag{D.2}$$

where (i) D is the data set to be processed, (ii) $\theta \in \mathbb{R}^d$ is the ML model that encodes the learned knowledge of D and (iii) the target function $L \colon \mathbb{R}^d \to \mathbb{R}^+$ quantifies the loss of θ on D, and (iv) η is the step size; θ_0 is chosen randomly, and (D.1) is repeated until a convergence criterion is met.

Parallelization of the inherently sequential SGD algorithm is non-trivial, and in order to adhere to the sequential semantics, parallelism can be allowed only within individual iterations, with strict synchronization before the next. Traditional Synchronous SGD (*SyncSGD*), which employs strict gradient-averaging synchronization after every iteration, falls into this category, and consequently maintains the broad set of convergence results that are known for sequential SGD. Modern ML deployments that benefit from this fact range from GPU-accelerated DL jobs on nowadays standard desktop computers, to wide networks of computing devices that jointly solve complex data analytics problems using distributed data, e.g., *Federated Learning* (FL). However, *SyncSGD* suffers two fatal drawbacks, namely that (i) straggling workers cause global slow-down, and (ii) gradient averaging multiplies the effective mini-batch size, both contributing to limited scalability.

Asynchronous SGD (*AsyncSGD*) relieves both aforementioned drawbacks of *SyncSGD* by relaxing the sequential SGD semantics, allowing threads to individually follow (D.1), coordinating only access to the shared state θ. The lack of synchronization drastically reduces waiting, which can provide significant speedup. However, gradients are no longer necessarily applied to the same state as they were computed based on, and instead the execution follows:

$$\theta_{i+1} = \theta_i - \eta \nabla \tilde{L}(v_i) \tag{D.3}$$

where $v_i = \theta_{i-\tau_i}$ is the view of the updating worker in iteration i, and τ_i is the staleness, i.e., the number of *intermediate* iterations from the moment of *reading* the state θ until the corresponding update is applied. Higher staleness in an *AsyncSGD* execution implies larger deviation from the sequential semantics, the magnitude of which is referred to as *asynchrony-induced noise* (AIN). AIN has been discussed in connection to *AsyncSGD*, however current literature on the topic lacks formal definitions of the same, as well as methods for measuring it, and consequently actual experimental measurements are lacking.

Motivation and Challenges.   While the impact of the parallelism degree is well-known for *SyncSGD*, it is unpredictable and problem-specific for its asynchronous counterpart *AsyncSGD*. For *AsyncSGD*, it has been established that

over-parallelism leads to higher staleness and AIN, with unstable, fluctuating loss values as a consequence, and even diverging and crashing executions in the worst case [51, 56], also observed in **Chapter B**. Thus, using appropriate parallelism is important for ensuring high convergence quality, as well as for avoiding unnecessary consumption of computational resources, especially considering the energy consumption due to modern deep learning research [7]. Nonetheless, the computational benefits of asynchrony cannot be dismissed, and must be weighed against its impact on the statistical convergence properties, i.e., the computational vs. statistical efficiency trade-off.

Given any ML job, it is notoriously difficult to know the appropriate asynchronous parallelism degree in advance, and finding it generally requires costly exhaustive searches [47, 55] (see also **Chapter B**). However, in practice only a small number of executions to reach a model of sufficient quality can be tolerated, and exhaustive searches [4] are excessively time consuming, and require significant computational resources. This implies a fatal idea-implementation gap, where on the one side *AsyncSGD* can provide tremendous acceleration of DL jobs, but it is agonizingly difficult to apply in practice. Moreover, computational resources necessitate analogous energy consumption, sometimes disproportionately high to common belief, cf e.g. [7, 8]. There is hence a need for more robust, *instance-adaptive AsyncSGD* methods, that balance the computational vs. statistical trade-off, while retaining the computational benefits of asynchrony and utilizing computational resources in smart ways. Such methods should enable *single* executions with fast and stable convergence, automatically balancing the computational vs. statistical efficiency trade-off by regulating the parallelism level. To this end, the notion of AIN must be formally defined, so that it can be measured, and its impact on the convergence of *AsyncSGD* understood.

Contributions. This work takes crucial steps toward understanding and utilizing dynamic parallelism in *AsyncSGD* in practical applications, targeting resource efficiency and implied gains.

- We provide a formal definition of *asynchrony-induced noise* (AIN), denoted by ξ, which applies for asynchronous semantic relaxation in numerical iterative algorithms in general, and *AsyncSGD* in particular. In addition, we introduce a generic algorithmic extension of *AsyncSGD* for efficiently measuring the AIN in real-time in such algorithms, which we use to report its magnitudes for several relevant DL benchmarking problems.

- We establish several novel convergence bounds, focusing particularly on the impact of ξ and the parallelism degree, on the statistical performance of *AsyncSGD*. We analyze the convergence of *AsyncSGD* and *ElAsyncSGD* on general non-convex problems, as well as problems satisfying the Polyak-Lojasiewicz criterion, which applies for several relevant DL problems.

- We introduce *ElAsyncSGD*, an extension of *AsyncSGD*, which dynamically deploys and frees workers in real-time based on the execution instance, continually balancing the computational vs. statistical efficiency trade-off for optimal convergence speed, while striving for minimal consumption of

---

[4]Models of sufficient quality are typically found under sub-optimal parallelism level before such exhaustive searches complete, however after a large number of unfruitful trials.

computational resources. We formally argue for the ability of *ElAsyncSGD* to track and actualize the time-varying approximate optimal parallelism m^*.

- We present an extensive evaluation of the proposed *ElAsyncSGD*, benchmarking against standard *AsyncSGD* with *tuned optimal constant parallelism*, on several DL benchmarks, including LeNet and MLP training on MNIST, Fashion-MNIST and CIFAR-10. The evaluation reveals that the intelligent parallelism regulation of *ElAsyncSGD* entails drastic reduction of thread-seconds, and hence overall energy consumed by computational resources (reductions ranging between $30 - 67\%$). Due to improved balance between computational vs. statistical efficiency, *ElAsyncSGD* additionally exhibits more stable convergence trajectories, and converges to higher precision.

D.2 Literature review

The study of numerical methods under parallelism sparked due to the works by Bertsekas and Tsitsiklis [33]. Distributed and parallel asynchronous SGD has since been an attractive target of study, e.g. [25, 39, 42], among which Hogwild! [20] and *Leashed-SGD* of **Chapter B**.

The literature on parallelism and *AsyncSGD* in particular is vast, and a useful extensive overview is provided in [28].While it is emphasized in [7, 8] that resource utilization is a crucial factor in the deployment of associated applications and that we "*should prioritize computationally efficient hardware and algorithms*", to the best of our knowledge the literature on parallelization falls short in addressing the problem form the point of view of adaptiveness in computational resources.

Other approaches that target elasticity in *AsyncSGD* are present in the literature, however not targeting resource adaptation; here we give an overview of the most relevant works addressing asynchrony-awareness for improved convergence.

In [104] the framework ADAM is proposed, which combines semi-synchronous and model partitioning across nodes for boosting computational efficiency, which results in overall lower consumption of computational resources. The trade-off between computational and statistical efficiency was highlighted in [47], where a semi-asynchronous SGD approach was proposed, showing speedup for DL on CPU and GPU architectures. The same trade-off is described in [105], where a system that dynamically adapts momentum and mini-batch size parameters is proposed, based on the dynamics of asynchronous multi-worker DL, targeting overall reduced training time.

There are several works dedicated to staleness-adaptiveness, originating from [51], where a theoretical maximum staleness was used to decide the overall step size. Subsequently, several works utilize the actual observed staleness during execution to adapt the step size. In [52], convergence bounds for staleness-adaptive step sizes were established, assuming convexity, sparse gradients, and relative ordering of *read* and *update* operations. AdaDelay [53] proposed a $\mathcal{O}(1/\sqrt{\tau})$ staleness-adaptive step size for smooth, convex problems, showing improved scalability. **Chapter A** proposes a $\mathcal{O}(C^{-\tau})$ and $\mathcal{O}(C^{-\tau}/\tau!)$ schemes based on a staleness model, and in **Chapter C** a generic instance-based framework `ASAP.SGD` was proposed for generating execution-tailored staleness-adaptive step size functions. In [96], an adaptive scheme for regulating

synchronization frequency was proposed, in order to reduce communication overhead. In [106], the notion of elastic training is attributed to dynamically adjusting the step size and mini-batch size throughout the execution, in order to tune the momentum parameter.

In [41] the algorithmic effect of asynchrony in *AsyncSGD* is modelled by perturbing the stochastic iterates with bounded noise. Their framework yields convergence bounds, but as described in the paper, are not tight, and rely on strong convexity. It is useful to point out that the asynchrony-induced-noise (defined and denoted by ξ in our work) may also have positive effects in convergence of non-convex problems; however the understanding of the boundaries is limited; here we provide insights to this question, through the analysis of the correlation of ξ, the convergence rate and the degree of parallelism. In [107], an analysis framework, elastic consistency, is introduced and utilizes deviation between the true state and the perceived (in the spirit of asynchrony-induced noise) for establishing convergence of *AsyncSGD*.

While the above works utilize asynchrony-awareness for analysis and regulation of various system parameters, none of them propose elasticity in the sense of dynamically deploying and releasing computational resources. To the extent of the authors' knowledge, the proposed work is the first to do so, i.e., introduce resource elasticity for *AsyncSGD* by dynamically regulating the number of workers in order to continually and optimally balance the computational vs. statistical efficiency trade-off.

D.3   Preliminaries

Optimization problem.   We consider the unconstrained optimization problem of (D.2), and the *AsyncSGD* optimizer of (D.3). We assume that $\tilde{L} = L_B$ where $B \subset D$ is a uniformly sampled mini-batch of data, and that $\tilde{L}$ is an unbiased estimator of L_D, i.e., $\mathbf{E}\left[\tilde{L}(\theta)\right] = L_D(\theta) \ \forall \theta \in \mathbb{R}^d$. We assume that mini-batch samples, and hence the stochastic gradients $\nabla \tilde{L}$, are mutually statistically independent. The loss function $L_D \colon \mathbb{R}^d \to \mathbb{R}^+$, $\theta \mapsto L_D(\theta)$ quantifies the performance of an ANN model, parameterized by θ. SGD is repeated until θ satisfies ϵ-convergence, defined as $\|L(\theta) - L(\theta^*)\| < \epsilon$, θ^* being a global minimum of L, or until a pre-defined time limit elapses.

System model.   We consider a system with maximally $\hat{m}$ asynchronous workers.   We generally refer to workers as parallel, as in a shared-memory context, however the methods apply to general distributed contexts as well, such as *asynchronous federated learning*.   The workers follow the SGD rule of (D.1) asynchronously, hence the system progress follows (D.3). The instantaneous n.o. workers, i.e., the parallelism degree, is generally denoted by m. Access to the shared model parameters θ is regulated by the implementing algorithm, e.g., standard lock-based *AsyncSGD*, lock-free inconsistent HOGWILD!, or consistent lock-free *Leashed-SGD* of **Chapter B**. By *staleness distribution* we refer to the distribution of all observed staleness values throughout a particular execution of *AsyncSGD*. In the case of HOGWILD!, where updates are non-atomic, iterations have no natural total order, which is required to calculate staleness. For this, we define a total ordering of updates similarly as in [45]; details appear in Section D.4.3. The staleness (τ_i) is considered to be a stochastic process, where
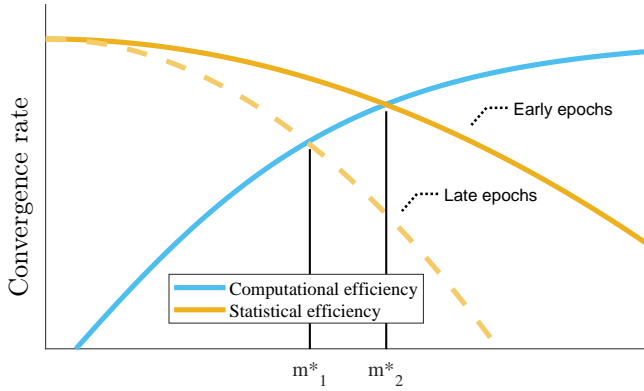
Figure D.1: Computational vs statistical efficiency trade-off.

instances are not necessarily mutually independent. In the case of constant parallelism, we assume that (τ_i) are identically distributed. We generally consider the executions to constitute *non-anticipative* processes, since future states do not influence the past, implying in particular that past states are mean-independent from them.

D.4 Method

In the following, we introduce an elastic extension to *AsyncSGD*, namely the *ElAsyncSGD*, as well as an efficient algorithm for measuring ξ, and argue for their correctness.

D.4.1 Problem analysis

The goal of iterative optimization for solving (D.2) is to optimize the overall convergence rate $\frac{\partial L}{\partial t}$, where t denotes wall-clock time. In particular, the convergence rate is a product of the *computational* and *statistical* efficiency of the implementing algorithm:

$$\frac{\partial L}{\partial t} = \frac{\partial L}{\partial i}\frac{\partial i}{\partial t} \tag{D.4}$$

according to the chain rule, where i denotes the iteration number. In the case of a minimization problem, we are naturally also interested in minimizing $\frac{\partial L}{\partial t}$. Now, higher parallelism generally ensures higher computational efficiency $\frac{\partial i}{\partial t}$, however it also induces noise due to staleness, with worse statistical efficiency $\frac{\partial L}{\partial i}$ as a consequence. Hence, a trade-off emerges, which requires choosing an appropriate parallelism level that balances the two factors.

Note that $\frac{\partial i}{\partial t}$ depends on the parallelism degree m, but is not dependent on the state θ, hence we write $\frac{\partial i}{\partial t}(m)$. The same does not hold for $\frac{\partial L}{\partial i}$, why we write $\frac{\partial L}{\partial i}(\theta : m)$. In fact, $\frac{\partial L}{\partial i}(\theta : m)$ is typically heavily influenced by the current state θ, i.e., the reduction in loss for each iteration varies depending on the current stage of the convergence. Since this is likely to disrupt the computational vs.

statistical efficiency balance throughout an *AsyncSGD* execution (D.4), it is likely that *a constant-parallelism approach is not optimal*.

Instead, we hereby hypothesize that there is an optimal, potentially non-constant, parallelism level trajectory m_i^*, that minimizes (D.4) for each iteration i throughout an *AsyncSGD* execution, as illustrated in Figure D.1.

$$m_i^* = \arg\min_m \frac{\partial L}{\partial i}(\theta : m)\frac{\partial i}{\partial t}(m)$$

With the above in mind, we pose the following:

Hypothesis D.4.1. There is a time-varying optimal parallelism m_i^* that outperforms the optimal constant parallelism, in overall better convergence rate (D.4), or in terms of computational resources consumption, or both.

We hence propose *ElAsyncSGD*, an elastic extension of *AsyncSGD*, that is designed to maintain a close-to-optimal balance between computational and statistical efficiency, dynamically, throughout the entire *AsyncSGD* execution. In particular, we want to enable speedup through asynchronous parallelism, avoiding over-parallelization with its associated *wasted computational resources* and instability, as well as avoiding under-parallelization with unnecessarily slow convergence. We are particularly interested in how an elastic parallelism strategy (i) balances the trade-off (D.4), (ii) affects the overall asynchrony-induced noise, and (iii) affects the overall computing resources used during an *AsyncSGD* execution.

D.4.2 Defining asynchrony-induced noise

In the following, we formally define the *asynchrony-induced noise* (AIN):

Definition D.4.2. The *asynchrony-induced noise* (AIN), denoted by ξ, constitutes a d-dimensional time series (ξ_i), where for each iteration i, ξ_i is the component-wise difference between the update $\nabla L(v_i)$ that is applied by a worker w and the correct update $\nabla L(\theta_i)$ according to the sequential semantics, scaled by the step size:

$$\xi_i = \eta \left(\nabla L(v_i) - \nabla L(\theta_i) \right) \tag{D.5}$$

where $v_i = \theta_{t-\tau_i}$ is w's *view* of the state used to compute the applied update.

Note that the AIN (ξ_i) constitutes a time series, where we generally do not assume mutual independence. The above definition of AIN is a natural one, in particular since it has the following property:

$$\theta_{i+1} = \theta_i - \eta \nabla L(v_i)$$
$$= \theta_i - \eta \nabla L(\theta_i) + \xi_i$$

From the above, we see that at iteration i, ξ_i is exactly the deviation between the update that is applied in an *AsyncSGD* execution and the update that should have been applied according to the sequential semantics. In the following, we frequently mention AIN in terms of its magnitude, i.e., $\|\xi\|$, partly as means to reduce its dimension (which is considerable in DL problems), and in particular due to the importance of the term for analyzing the convergence of *AsyncSGD*.
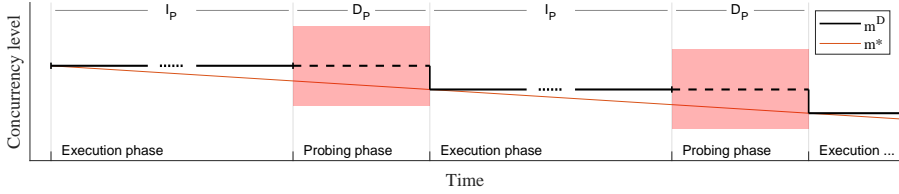
Figure D.2: Probing-based elastic parallelism control.

The significance of $\|\xi_i\|$ on the analytical convergence properties of *Async-SGD* in non-convex problems will be established in Section D.5, an efficient algorithm for its measurement is presented in Section D.4.3, and its overall magnitude serves as a metric for quantifying the influence of asynchrony on the convergence in practical DL applications (Section D.6).

D.4.3 *ElAsyncSGD*

The proposed *ElAsyncSGD*, an elastic extension of *AsyncSGD*, dynamically frees and deploys workers based on the time-varying dynamics of the optimization problem at hand. *ElAsyncSGD* finds an estimated optimal parallelism degree by actively probing relevant parallelism levels, and keeping track of their respective convergence rates, at different points in time. Specifically, *ElAsync-SGD* will regularly, with a certain frequency, probe parallelism levels adjacent to the current one, by executing *AsyncSGD* at those parallelism levels for a certain amount of sampling time (Figure D.2). Note that, while mentioning primarily parallel settings here, the methods disclosed in the following apply also to general asynchronous *distributed* and *federated learning* settings, with for instance a central parameter server which communicates with worker nodes through message passing. The relevant parameters of *ElAsyncSGD* are the following:

- I_P - Inter-probing interval, n.o. iterations between probes

- D_P - Probing duration, n.o. iterations per parallelism level during probing

- W_P - Parallelism-level exploration (also called *m-window size* in the algorithm); it defines maximum deviation from the previously best-known parallelism level.

In order to ensure reliable online measurements of the instantaneous convergence rate we make sure each worker performs several read-compute-update rounds during each measurement time window. To this end, the parameters I_P and D_P are used in multiples of the instantaneous parallelism level m.

Next, we show an outline of the main steps of *ElAsyncSGD*. Following the algorithm outline, the remainder of this section describes in detail the algorithmic structure of *ElAsyncSGD*:

[a] Parallelism level is initialized at some m_0^E

[b] First probing phase; adjust parallelism with best estimated convergence rate.

(a) Estimate convergence rate for adjacent parallelism levels $m \in \{m^E + 1, m^E - 1\}$

(b) Choose next m^E as parallelism with best known estimated convergence rate. If an improved m^E is found, repeat.

[c] Repeat until ϵ-convergence is achieved, or time elapses.

(a) Execution phase. Execute I_P iterations of *AsyncSGD* according to (D.3) with m^E workers. Terminate if ϵ-convergence is achieved, or time elapses.

(b) Probing phase.

i. Estimate convergence rate for parallelism levels $m \in \{m^E - W_P, m^E + W_P\}$ by executing D_P *AsyncSGD* iterations at each level.

ii. Choose next m^E with best estimated convergence rate, among $\{m^E - W_P, m^E + W_P\}$.

Algorithmic structure. The core of *ElAsyncSGD* appears in Algorithm D.4, which implements the REGULATOR procedure in particular, responsible for measuring ξ and regulating the parallelism level. The algorithm runs alongside a pool of workers, who execute an implementation of *AsyncSGD*. Algorithm D.2 implements traditional lock-based *AsyncSGD* and HOGWILD!, with the adaptations needed to utilize *ElAsyncSGD* in conjunction. For convenience, the *AsyncSGD* implementations use the PARAMETERVECTOR structure of **Chapter B** for managing θ. The *ElAsyncSGD* adaptations constitute communication of workers' states to the REGULATOR, which is structured through the dedicated interface REGULATORINTERFACE (Algorithm D.1). The REGULATOR relies on the PROBESTATE data structure (Algorithm D.3) for organizing measurements during probing phases, to be used for taking control actions for adjustments in parallelism level.

The probing and parallelism control is the responsibility of the dedicated REGULATOR procedure. In particular, the REGULATOR sets the elastic parallelism level m^E, which is accessed by asynchronous workers to check whether they should be active. Considering workers being enumerated by their IDs, any worker w checks this every iteration simply by evaluating $w < m^E$, and proceeds to be active whenever that condition is true. This behavior is denoted by the **whenever** keyword in Algorithm D.2.

Measuring AIN. Computing ξ_i, defined in (D.5), of iteration i under asynchronous parallelism is not straight-forward since the correct (according to the sequential semantics) update $\nabla f(\theta_i)$ is not available at update-time, i.e., when $\nabla f(v_i)$ is applied.

The solution presented here is based on the observation that the correct update $\nabla f(\theta_i)$ of iteration i (as opposed to the actually applied $\nabla f(v_i)$) will be computed, and applied eventually in the future, by any worker w that reads the state θ_i in iteration i. By making $\nabla f(\theta_i)$ available to w by the time w has finished computing $\nabla f(\theta_i)$, ξ_i can be computed at that time. This observation is summarized in the following:

Statement D.4.3. The update $\nabla f(v_i)$ of any iteration i, applied by some worker w, is the semantically correct update of some earlier iteration, namely $i - \tau_i$. By making $\nabla f(\theta_i)$ available to w, $\xi_{i-\tau_i}$ can consequently be computed.

This is solved in *ElAsyncSGD* by workers reporting, through the dedicated Regulatorinterface, whenever they are *viewers* of a state θ_i (which is then their individual latest *view* of θ) as well as when they finish computing updates (see Algorithm D.2). In addition to the pointer to the actual update, workers also report their unique worker IDs, as well as the current iteration number (see Table D.1). This information is summarized by Regulatorinterface (Algorithm D.1) and communicated, through a FIFO queue, to the Regulator (Algorithm D.4), which performs the computations for estimating ξ and regulating parallelism.

Parameter choice. Note that higher values of the probing duration D_P mean longer sampling time, hence higher measurement reliability. Similarly, larger parallelism exploration window size W_P implies increased visibility, and potentially broader basis for the parallelism choice. However, too large of either D_P or W_P may be misleading since the instantaneously optimal m^* changes over time. In practice, an inter-probing interval of approximately $I_P = 10 \cdot D_P$ is a useful rule of thumb that has been found empirically to work across various optimization problems, as will be seen in Section D.6.

D.4.4 Approximating m^*

In the following, we outline the main arguments for the estimation of m^* by the proposed *ElAsyncSGD*, which rely on the following:

Assumption D.4.4. m^* drifts maximally W_P within an execution and probing cycle, i.e., $I_P + D_P(2W_P + 1)$ iterations.

Assumption D.4.4 entails particularly that it is possible to choose the parameters I_P, D_P, W_P such that it holds. In practice, as we shall see in Section D.6, this is realistic, as straight-forward parameter choices satisfy this for several DL problems.

Note that *ElAsyncSGD* essentially follows an iterative active binary search principle, regularly probing the convergence rate $\frac{\partial L}{\partial t}$ at adjacent parallelism levels. Further considering $\frac{\partial L}{\partial t}(m)$, as a function of parallelism, having an optimum m^*, and monotonically decreasing and increasing, respectively, on either side of m^*, we outline the following arguments for that *ElAsyncSGD* actualizes the time-varying optimal m^* through its regulated elastic parallelism m^E:

[a] The first probing phase will converge to m^*. This follows from the assumptions on $\frac{\partial L}{\partial t}(m)$, and from that the first probe loop does not terminate until $m_i^E = m_i^*$ is observed.

[b] At the end of each probing phase, the algorithm will have caught up to new $m^E = m^*$. This follows recursively from that $m^E = m^*$ at the start of the phase, and by applying the argument from Assumption D.4.4 inductively.

Now, the above two points inductively entail in particular that the actual, by *ElAsyncSGD*, parallelism m^E will, as often as probes occur, assume the optimal time-varying m^*.

Algorithm D.1 REGULATORINTERFACE

GLOBAL FIFOQueue Q
views($\hat{m}$) $\leftarrow [false, \ldots, false]$

procedure MARK_VIEWER(w)
    views[w] $\leftarrow true$

procedure SUBMIT_GRAD(w, ∇, i) $\triangleright$ Worker w is **the writer** at iteration i
    v_i $\leftarrow \{j$ for views[j]$= true\}$ $\triangleright$ All viewers threads of the current θ
    views $\leftarrow [false, \ldots, false]$ $\triangleright$ Reset the *views* array
    $job \leftarrow \{$
        iteration: i
        write_thread: w
        update: ∇
        viewer_threads: **v**_i
    $\}$
    Q.enqueue(job)

Table D.1: Structure of tuples arriving at the regulator.

	Description
i	Current iteration number
w_i	ID of updating worker
δ_i	Pointer to update that was applied by w_i
v_i	Set of workers that *viewed* the current state

Algorithm D.2 Adapted *Lock-based AsyncSGD* and HOGWILD!

GLOBAL REGULATORINTERFACE REG_INTERF
GLOBAL PARAMETERVECTOR PARAM $\triangleright$ Shared model parameters
GLOBAL Float η $\triangleright$ Step size
GLOBAL Lock MTX $\triangleright$ For synchronizing access to θ

<u>Initialization</u>
PARAM.RAND_INIT() $\triangleright$ Randomly initialize parameters
<u>Each worker w</u>
whenever $w \leq m^E$ $\triangleright$ m^E is the elastic n.o. active workers
    if *Lock-based* **then** MTX.lock()

    local_param $\leftarrow$ copy(PARAM.theta) $\triangleright$ Allocate new memory and copy θ
    REG_INTERF.REPORT_VIEWER(w) $\triangleright$ For the current i, worker w is **one of the viewers** at iteration i, i.e., $w \in \mathbf{v}_i$

    if *Lock-based* **then** MTX.unlock()

    local_grad $\leftarrow$ comp_rand_grad(local_param) $\triangleright$ Allocate new memory and compute gradient

    if *Lock-based* **then** MTX.lock()

    REG_INTERF.SUBMIT_GRAD(w, local_grad, PARAM.i)
    PARAM.UPDATE(local_grad, η)
    delete local_param
    if *Lock-based* **then** MTX.unlock()

Algorithm D.3 PROBESTATE

---

conv_rate($\hat{m}$) ← [$null, \dots, null$]
first_probe ← $true$
pre_probe_m, probe_start_t, probe_start_loss ← $null$

procedure START_MEASURE(m, δ)
    if $m \neq$ null **then** pre_probe_m ← m^E
                                              ▷ Set only once every probing phase
    probe_start_t ← System.$now()$
    probe_start_loss ← δ_i.loss()

function STOP_MEASURE(δ)
    conv_rate[m^E-1] ← (δ.loss() - probe_start_loss) / (System.$now()$ - probe_start_t)
    if first_probe **then**
        window_center ← arg min (conv_rate) + 1
    else
        window_center ← pre_probe_m
    next_m^E ← $null$
    for m_probe = -W_P; m_probe ≤ W_P; m_probe += 1 **do**
        if conv_rate[window_center + m_probe - 1] == -1 **then**
            next_m^E ← window_center + m_probe
            break
    if next_$m^E = null$ **then**   ▷ Probing phase finished
        first_probe ← $false$
        done ← $true$
        next_m^E ← arg min (conv_rate) + 1
        conv_rate← [$null, \dots, null$]   ▷ Reset probe state
        pre_probe_m, probe_start_t, probe_start_loss ← $null$
    else   ▷ Probing window not fully explored yet
        probe_start_t ← System.$now()$
        probe_finish_at ← i+next_$m^E \cdot D_P$
    return next_m^E, done

Algorithm D.4 *ElAsyncSGD* - main Regulator procedure

GLOBAL FIFOQueue Q
ProbeState probe_state
frozen_grads(M) $\leftarrow$ [nullptr, ..., nullptr]
next_probe_start $\leftarrow 0$
probe_finish_at $\leftarrow$ null
repeat
    Q.deque($\{i,\ w\_i,\ \delta\_i,\ v\_i\}$)

    <u>Parallelism regulation</u>

    if $i = $ next_probe_start **then**
        probe_state.start_measure(m^E, δ_i)
        probe_finish_at $\leftarrow i + m^E \cdot D_P$
    else if $i = $ probe_finish_at **then**
        m^E, final $\leftarrow$ probe_state.stop_measure(δ_i)
        if final **then** $\triangleright$ This probing phase is completed
            next_probe_start $\leftarrow i + m^E \cdot D_E$
        else $\triangleright$ Probing phase continues
            probe_state.start_measure(null, δ_i)
            probe_finish_at $\leftarrow i + m^E \cdot D_P$

    <u>Measure ξ</u>
    if $v_i = \varnothing$ **then** $\triangleright$ No thread will eventually use δ_i for computing ξ
        delete δ_i
    else
        for viewer_i $\in v\_i$ **do** $\triangleright$ Leave pointer to δ_i for all reading threads r_t
            frozen_grads[viewer_i] $\leftarrow \delta\_i$
    if frozen_grads[w_i] $= $ nullptr **then** $\triangleright$ w_i was a reader $\in \mathbf{r}_{i-\tau_i}$. Check if some $\delta_{i-\tau_i}$ has been left for w_i
        continue
    $\delta\_(i - \tau_i) \leftarrow$ frozen_grads[w_i]
    for $j = 0, \ldots, m-1$ **do** $\triangleright$ w_i is the first reader of $\theta_{i-\tau_i}$ to finish computing an update. Clear the pointer to $\delta_{i-\tau_i}$ for all in $r_{i-\tau_i}$
        if frozen_grads[j] $= \delta_{i-\tau_i}$ **then**
            frozen_grads[j] $\leftarrow$ nullptr
    $\xi\_(i - \tau_i) \leftarrow \|\delta\_i - \delta\_(i - \tau_i)\|$
    delete $\delta\_(i - \tau_i)$ $\triangleright$ Delete update and reclaim memory
until convergence

D.5 Analysis

In the following, we establish asymptotic convergence for *ElAsyncSGD*, and general *AsyncSGD*, executions, studying especially the impact of the overall magnitude of AIN ($\|\xi\|$).

We make the following assumptions on the loss function L:

Assumption D.5.1. The loss function L has Lipschitz-continuous gradients; there exists a constant $\mathcal{L}$ such that:

$$\mathbf{E}\big[\|\nabla L(\theta^1) - \nabla L(\theta^2)\|\big] \le \mathcal{L}\mathbf{E}\big[\|\theta^1 - \theta^2\|\big] \ \forall \theta^1, \theta^2 \tag{D.6}$$

Assumption D.5.2. The loss function L has bounded expected gradient moment; there exists a constant $\mathcal{M}$ such that:

$$\mathbf{E}\big[\|\nabla L(\theta)\|^2\big] \le \mathcal{M}^2 \ \forall \theta \tag{D.7}$$

Assumptions D.5.1 and D.5.2 provide additional structure, hold for a wide set of practical applications, and are widely adopted in the literature [24,45,51].

First, we establish the linear dependency between the AIN and overall execution staleness in the following:

Lemma D.5.3. *The expected asynchrony-induced noise $\mathbf{E}[\|\xi\|]$ in iteration i is linearly upper-bounded by the average staleness $\mathbf{E}[\tau]$*

$$\mathbf{E}[\|\xi_i\|] \le \eta\mathcal{L}\mathcal{M}\mathbf{E}[\tau_i]$$

Proof. From the definition of ξ and Assumption D.5.1:

$$\|\xi_i\| = \|\nabla L(v_i) - \nabla L(\theta_i)\|$$
$$\le \mathcal{L}\|\sum_{j=i}^{\tau_i} \theta_{i-j+1} - \theta_{i-j}\| \le \eta\mathcal{L}\sum_{j=1}^{\tau_i} \|\nabla L(\theta_{i-j})\|$$

Considering an *AsyncSGD* execution as non-anticipative, since the future does not influence the past, implies historic states are mean-independent of future states. In particular:

$$\mathbf{E}[\|\xi_i\| \mid \tau_i] \le \eta\mathcal{L}\sum_{j=1}^{\tau_i} \mathbf{E}[\|\nabla L(\theta_{i-j})\|]$$
$$\le \eta\mathcal{L}\mathcal{M}\tau_i$$

by Assumption D.5.2. Now, taking the full expectation, the above rewrites to the Lemma statement. $\qquad\square$

Corollary D.5.4. *The asynchrony-induced noise $\|\xi\|$ in iteration i is linearly upper-bounded by the parallelism level m_i*

$$\mathbf{E}[\|\xi_i\|] \le \eta\mathcal{L}\mathcal{M}m_i$$

Corollary D.5.4 follows directly from Lemma D.5.3, since $\mathbf{E}[\tau_i] \approx m_i$ is generally known to hold [24] (see also **Chapter A** and **Chapter C**).

Next, we establish expected statistical progress per iteration:

Lemma D.5.5. *Consider the optimization problem of (D.2) and follow the SGD step (D.3). Then we have the following expected iterative progression:*

$$\mathbf{E}\left[\frac{\partial L}{\partial i}\right] \leq -\eta \mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big] +$$

$$\eta \mathcal{M}\left(\mathbf{E}[\|\xi_i\|] + \frac{1}{2}\mathcal{L}\mathcal{M}\eta\right)$$

Proof. From assumption D.5.1 we have in particular

$$\mathbf{E}\left[\frac{\partial L}{\partial i}\right] \leq \frac{\mathcal{L}}{2}\mathbf{E}\big[\|\theta_{i+1} - \theta_i\|^2\big] + \mathbf{E}[\langle \nabla L(\theta_i), \theta_{i+1} - \theta_i\rangle]$$

From the SGD step we have

$$\mathbf{E}\left[\frac{\partial L}{\partial i}\right] \leq -\eta \mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big]$$

$$+ \eta \mathbf{E}[\|\nabla L(\theta_i)\|\|\xi_i\|] + \frac{\mathcal{L}}{2}\mathcal{M}^2\eta^2$$

Assumption D.5.2, and historic states' mean-independence of future states, now gives:

$$\mathbf{E}\left[\frac{\partial L}{\partial i}\right] \leq -\eta \mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big]$$

$$+ \eta \mathbf{E}[\|\nabla L(\theta_i)\|]\mathbf{E}[\|\xi_i\|] + \frac{\mathcal{L}}{2}\mathcal{M}^2\eta^2$$

$$\leq -\eta \mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big] + \eta \mathcal{M}\left(\mathbf{E}[\|\xi_i\|] + \frac{1}{2}\mathcal{L}\mathcal{M}\eta\right)$$

$\square$

With Lemma D.5.5 as a starting point, next we derive explicit convergence bounds for *AsyncSGD* executions in general, and *ElAsyncSGD* in particular, focusing on the influence of $\|\xi\|$, as well as the ranges of $\|\xi\|$ and η for which convergence is guaranteed:

Theorem D.5.6. *Assume $L(\theta_0) - L(\theta^*) < \delta$. Then, for general non-convex target functions L, expected convergence to within ϵ of stationary point is guaranteed after*

$$I > \frac{L(\theta_0) - L(\theta^*)}{\eta(\epsilon + \frac{1}{2}\mathcal{L}\mathcal{M}^2\eta - \mathcal{M}\mathbf{E}[\|\xi\|])}$$

iterations, for step sizes η within

$$0 < \eta < \frac{2}{\mathcal{L}\mathcal{M}}\left(\frac{\epsilon}{\mathcal{M}} - \mathbf{E}[\|\xi\|]\right)$$

iff the asynchrony-induced noise is sufficiently low:

$$\mathbf{E}[\|\xi\|] < \frac{\epsilon}{\mathcal{M}} \tag{D.8}$$

Proof. From Lemma D.5.5, we have

$$\mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big] \leq \frac{L(\theta_i) - L(\theta_{i+1})}{\eta}$$

$$+ \mathcal{M}\left(\mathbf{E}[\|\xi_i\|] + \frac{1}{2}\mathcal{L}\mathcal{M}\eta\right)$$

$$\Rightarrow \frac{1}{I}\sum_{i=0}^{I-1}\mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big] \leq \frac{\delta}{\eta I} + \mathcal{M}\left(\mathbf{E}[\|\xi\|] + \frac{1}{2}\mathcal{L}\mathcal{M}\eta\right)$$

which implies in particular that the same bound applies for $\min_i \mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big]$ as well. Now, the right-hand side of the inequality is $\leq \epsilon$ iff

$$I \geq \frac{\delta}{\eta}\left(\epsilon - \mathcal{M}(\mathbf{E}[\|\xi\|] + \frac{1}{2}\mathcal{L}\mathcal{M}\eta)\right)^{-1}$$

from which the restrictions on η and $\mathbf{E}[\|\xi\|]$, and the theorem statement, follow. $\qquad\square$

Corollary D.5.7. *The bound of Theorem D.5.6 is tightest for*

$$\eta^* = \frac{\epsilon - \mathcal{M}\mathbf{E}[\|\xi\|]}{\mathcal{M}^2\mathcal{L}}$$

for which expected convergence to within ϵ of stationary point is achieved after

$$I > \frac{2\mathcal{M}^2\mathcal{L}}{(\epsilon - \mathcal{M}\mathbf{E}[\|\xi\|])^2}\left(L(\theta_0) - L(\theta^*)\right)$$

AsyncSGD *iterations.*

Note that (D.8) ensures positive denominators in the convergence bounds of Theorem D.5.6 and Corollary D.5.7, which implies that, in its allowed range, higher precision (smaller ϵ) requires overall smaller $\mathbf{E}[\|\xi\|]$, and that larger magnitude of $\mathbf{E}[\|\xi\|]$ implies longer time to ϵ-convergence.

Next, we show explicitly the influence of the overall parallelism degree on the convergence of *AsyncSGD*:

Theorem D.5.8. *Let $L(\theta_0) - L(\theta^*) < \delta$. Then:*

$$\min_i \mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big] \leq \frac{\delta}{\eta I} + \eta\mathcal{M}^2\mathcal{L}(\bar{m} + \frac{1}{2})$$

where $\bar{m} = \frac{1}{I}\sum_i m_i$ is the overall average parallelism level.

Proof. From Lemma D.5.5 and Corollary D.5.4, we adopt a proof approach similar to the one of Theorem D.5.6, and yield:

$$\mathbf{E}\left[\frac{\partial L}{\partial i}\right] \leq -\eta\mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big] + \eta^2\mathcal{M}^2\mathcal{L}(m_i + \frac{1}{2})$$

$$\Rightarrow \frac{1}{I}\sum_{i=0}^{I-1}\mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big] \leq \frac{\delta}{\eta I} + \eta\mathcal{M}^2\mathcal{L}(\bar{m} + \frac{1}{2})$$

why the bound holds also for $\min_i \mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big]$, from which the theorem statement follows. $\qquad\square$

Next, we study *AsyncSGD* convergence under the following:

Assumption D.5.9. Polyak-Lojasiewicz (PL) condition. A function L is referred to as μ-PL if, for some $\mu > 0$:

$$\mathbf{E}\big[\|\nabla L(\theta)\|^2\big] \geq \mu\,\mathbf{E}[L(\theta) - L(\theta^*)] \;\forall\theta \tag{D.9}$$

The PL condition is a generalization of convexity, without the requirement of optimum uniqueness, and geometrically characterizes a relevant class of ML loss functions. Examples of such are least squares, logistic regression, support vector machines [62] and certain types of deep ANNs [63]. The convergence of *AsyncSGD*, and in particular *ElAsyncSGD*, is guaranteed PL loss functions in the following:

Theorem D.5.10. *For target functions L satisfying the μ-PL condition of Assumption D.5.9, let*

$$0 < \eta \leq \frac{\mu\epsilon - 2\mathcal{M}^2\mathbf{E}[\|\xi\|]}{2\mathcal{M}^2\mathcal{L}}$$

Then we have expected ϵ-convergence after

$$I \geq \frac{2\mathcal{M}^2\mathcal{L}}{\mu(\mu\epsilon - 2\mathcal{M}^2\mathbf{E}[\|\xi\|])} \log\left(\frac{2(L(\theta_i) - L(\theta^*))}{\epsilon}\right)$$

iterations.

Proof. With Lemma D.5.5 as a starting point, and applying Assumption D.5.9, we have

$$\mathbf{E}\left[\frac{\partial L}{\partial i}\right] \leq -\eta\mathbf{E}\big[\|\nabla L(\theta_i)\|^2\big] + \eta\mathcal{M}\left(\mathbf{E}[\|\xi_i\|] + \frac{1}{2}\mathcal{L}\mathcal{M}\eta\right)$$

$$\Rightarrow \mathbf{E}[L(\theta_{i+1} - L(\theta^*)] \leq (1 - \eta\mu)\mathbf{E}[L(\theta_i) - L(\theta^*)]$$
$$+ \eta\mathcal{M}\left(\mathbf{E}[\|\xi_i\|] + \frac{1}{2}\mathcal{L}\mathcal{M}\eta\right)$$
$$\Rightarrow \mathbf{E}[L(\theta_I) - L(\theta^*)] \leq (1 - \eta\mu)^I\delta$$
$$+ \frac{1}{\mu}\mathcal{M}^2\left(\mathbf{E}[\|\xi_i\|] + \frac{1}{2}\mathcal{L}\mathcal{M}\eta\right) \leq \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon$$

$$\square$$

D.6 Evaluation

We evaluate the proposed *ElAsyncSGD* for LeNet and MLP training on several DL benchmark datasets, namely MNIST [89], Fashion-MNIST [102], and CIFAR-10 [70]. We compare the performance of *ElAsyncSGD* to standard static constant parallelism *AsyncSGD* baselines, using both the standard lock-based consistent implementation, as well as the lock-free inconsistent HOGWILD!. The evaluation focuses on overall convergence properties of the algorithms considered, final loss value achieved, convergence stability, savings in computational resources, as well as magnitudes of the AIN.

Selecting challenging baselines.   In order to ensure a relevant comparison, and to maximally challenge the proposed *ElAsyncSGD*, we perform exhaustive scalability tests in order to select the best static parallelism level for the *AsyncSGD* baselines, for all settings (Figure D.9), denoted by m^S. The constant-parallelism *AsyncSGD* baselines execute until 50%-convergence is achieved for each setting (combination of DL architecture, benchmark dataset, and *AsyncSGD* algorithm), over a wide parallelism spectrum, after which the parallelism level that gave the fastest convergence is chosen as baseline for that particular setting.

Baselines in practice.   Note that the above tuning process, although it gives the best-performing static-parallelism baselines, is utterly infeasible in practice. In any practical context, the goal is to find a model θ of sufficient quality (ϵ-convergence) utilizing as little time and computing resources as possible. Hence, such settings tolerate maximally a few executions of *AsyncSGD*, where unnecessarily slow, or unstable, parallelism levels are eliminated, and the problem is considered solved, however at (most likely) a sub-optimal parallelism level.   In contrast, *ElAsyncSGD* is a *single-execution* solution, which automatically regulates the parallelism for optimal convergence speed, as well as avoiding the instability associated with over-parallelism.   This implies in particular that *ElAsyncSGD* is drastically more applicable in practice compared to state-of-art.

Metrics of interest.   The relevant metrics relate to *loss*, *AIN*, and *computing resources*. More specifically:

   Loss — We measure and report ϵ-convergence rates, final loss values, and loss-over-time plots which show the overall convergence trajectory stability.

   AIN — We report overall AIN magnitudes throughout executions, meaning $\|\xi\|$ as in Definition D.4.2.

   Computing resources — Quantified by *worker-seconds*, i.e., total work time of workers, defined in detail below.

Computing resources consumption.   Analogous to the notion of man-hours spent in manual work, the concept of *worker-seconds* is used here to quantify the computing resources consumption by the proposed *ElAsyncSGD* and the best static parallelism *AsyncSGD* baseline, denoted by R^E and R^S, respectively. The worker-seconds of *ElAsyncSGD* is computed by integrating the parallelism level:

$$R^E(t) = \int_{t'=0}^{t} m_{t'} \ \mathrm{d}t' \tag{D.10}$$

The computing resources consumption of the static parallelism *AsyncSGD* baseline rewrites to:

$$R^S(t) = t \cdot m^S \tag{D.11}$$

Now we define the *excess* computing resources as the savings achieved by *ElAsyncSGD* relative to the static baseline.

$$R^{E-S}(t) = R^E(t) - R^S(t)$$

Whenever $R^{E-S} < 0$ the cumulative computing resources of the proposed *ElAsyncSGD* are lower than the best static parallelism *AsyncSGD* baseline, and we are interested in the t for which this is the case. It is however theoretically possible, although rare in practice, that $R^{E-S}(t)$ oscillates and has many zero-crossings. We therefore consider, and compute, a *break-even* time:

$$t_{\text{break-even}} = \min\{t \mid t' > t \Rightarrow R^{E-S}(t') < 0\}$$

which hence signifies the time at which *ElAsyncSGD* has consumed equal computing resources as the baseline and will have resulted in savings of such at every later point in time.

Implementation. We implement *ElAsyncSGD* for lock-based *AsyncSGD* and HOGWILD!, extending the open *Shared-Memory-SGD* [101] C++ library, connecting ANN operations to low-level implementations of parallel SGD.

Experiment setup. We benchmark *ElAsyncSGD* on DL benchmarks, particularly for image classification on MNIST [89] of hand-written digits, Fashion-MNIST [102] of clothing article images, and CIFAR-10 [70] of everyday objects. All datasets contain 60k images, each belonging to one of ten classes $\in \{0, \dots, 9\}$. For this, we train a LeNet CNN architecture, as well as a 4-layer MLP, with 128 neurons per layer (denoted MLP in the following), for a specific time period relevant to the particular problem setting. We use standard settings and hyper-parameters; for MLP training we use $\eta = 5e{-}3$ and mini-batch size 256, for LeNet $\eta = 1e{-}3$, and for CIFAR-10 $\eta = 1e{-}2$ and a mini-batch size of 8. The *multi-class cross-entropy* loss function is used in all experiments. The experiments are conducted on a 2.10 GHz Intel(R) Xeon(R) E5-2695 two-socket 36-core (18 cores per socket, each supporting two hyper-threads), 64GB non-uniform memory access (NUMA), Ubuntu 16.04 system. Note that, while the above describes a shared-memory context with parallel threads, the method, and its analysis in Section D.5 in particular, hold for a system of distributed worker nodes as well. The *ElAsyncSGD* parameters used are $m_0^E = \hat{m}/2$, and $I_P = 100 \cdot m^E, D_P = 10 \cdot m^E, W_P = 1$. Plots show averaged values from 5 executions for each setting, unless otherwise stated. ϵ-convergence is achieved when $L(\theta) < \epsilon$, expressed as % of the initial loss $L(\theta_0)$.

Outcome. For each of the aforementioned benchmarks, the best static parallelism *AsyncSGD* baseline, tuned according to above, and *ElAsyncSGD*, are evaluated and compared. Measurements appear in Figures D.3-D.7, where loss trajectory, parallelism level, AIN, and break-even points are visualized. Corresponding measurements for standard sequential SGD are included for reference. The overall performance differences between *ElAsyncSGD* and the baselines for all tests are summarized in Table D.2. There is a clear correlation between the parallelism level and the AIN, as defined in Section D.4.3 (Definition D.4.2), observable in all tests (Figure D.3-D.7). In addition, spikes in the AIN coincide with setbacks in the convergence progress, i.e., spikes in the loss trajectory. Such setbacks are especially apparent in the static parallelism baselines, which indicates how active parallelism control for keeping the AIN at reasonable levels may be beneficial for the convergence rate, which is indeed confirmed here. Figure D.8 shows how the staleness distributions change to overall lower magnitudes when utilizing the proposed *ElAsyncSGD* extension. Overall, *ElAsyncSGD* exhibits more stable convergence, with significantly
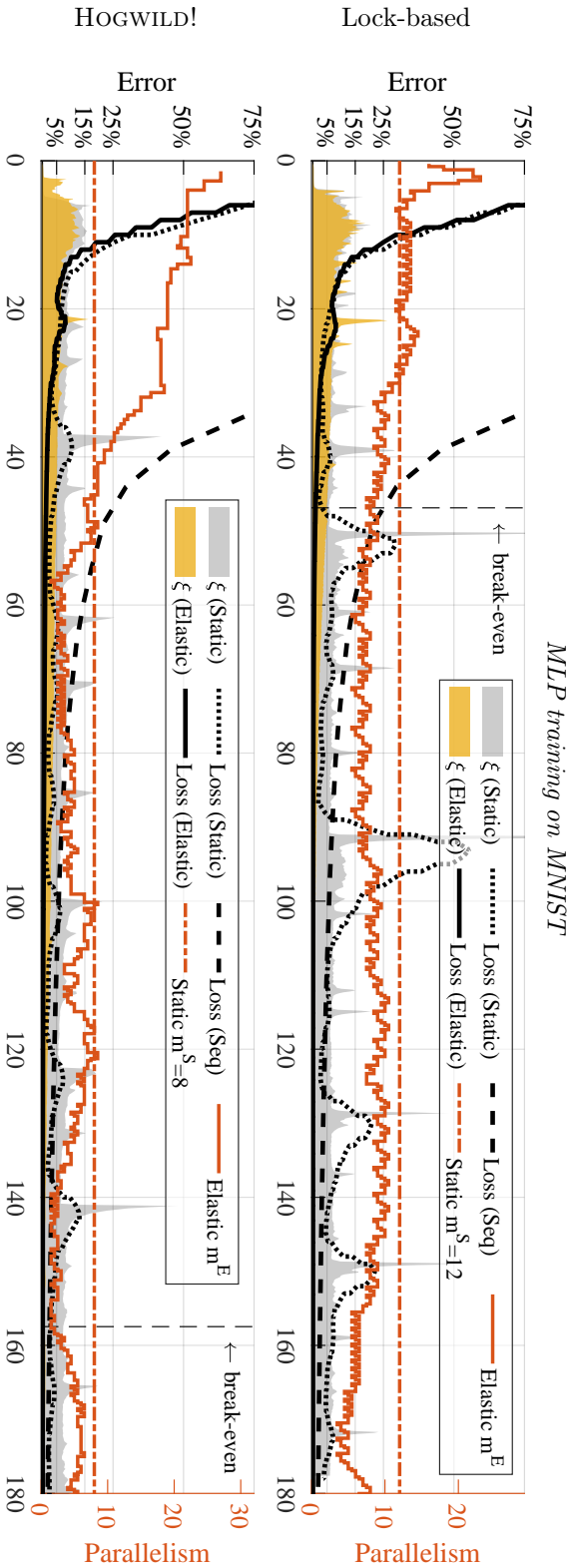
Figure D.3: Loss over time for MLP training on MNIST, comparing *ElAsyncSGD* (Elastic) with best constant parallelism baselines ($m = 12$ for *Lock-based*, $m = 8$ for HOGWILD!).
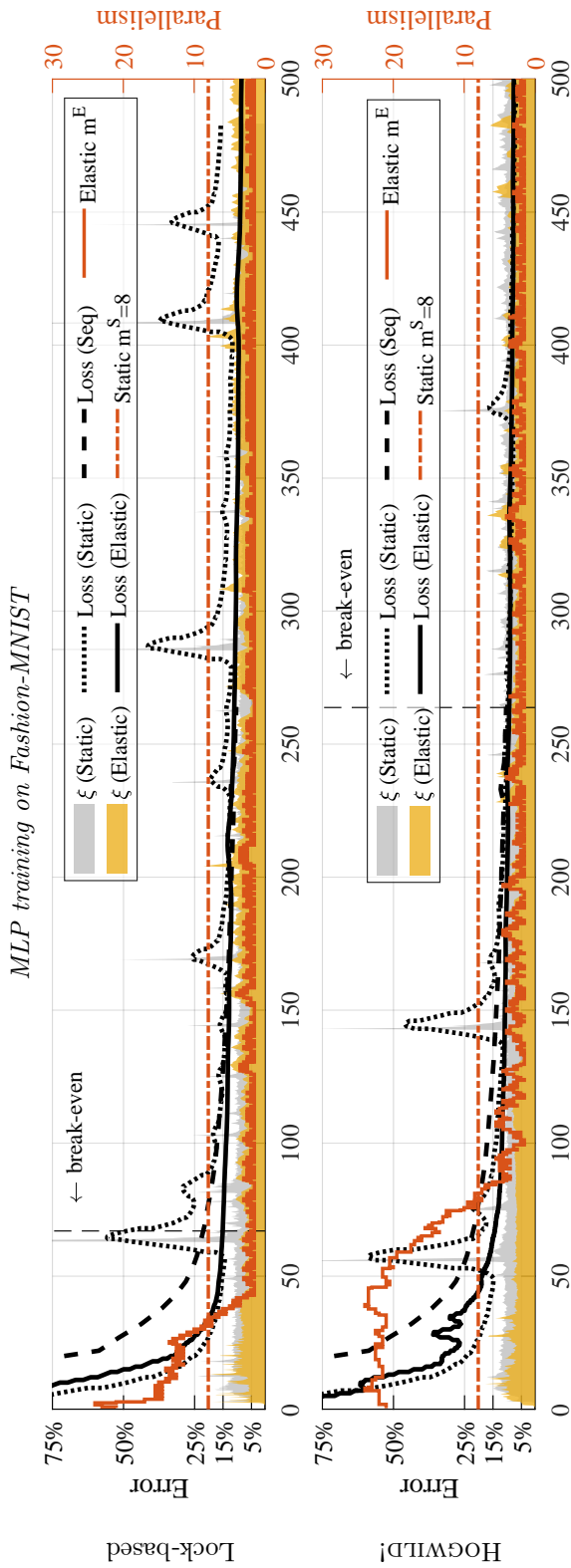
Figure D.4: Loss over time for MLP training on Fashion-MNIST, comparing *ElAsyncSGD* (Elastic) with best constant parallelism baselines ($m = 8$ for *Lock-based*, $m = 8$ for HOGWILD!).
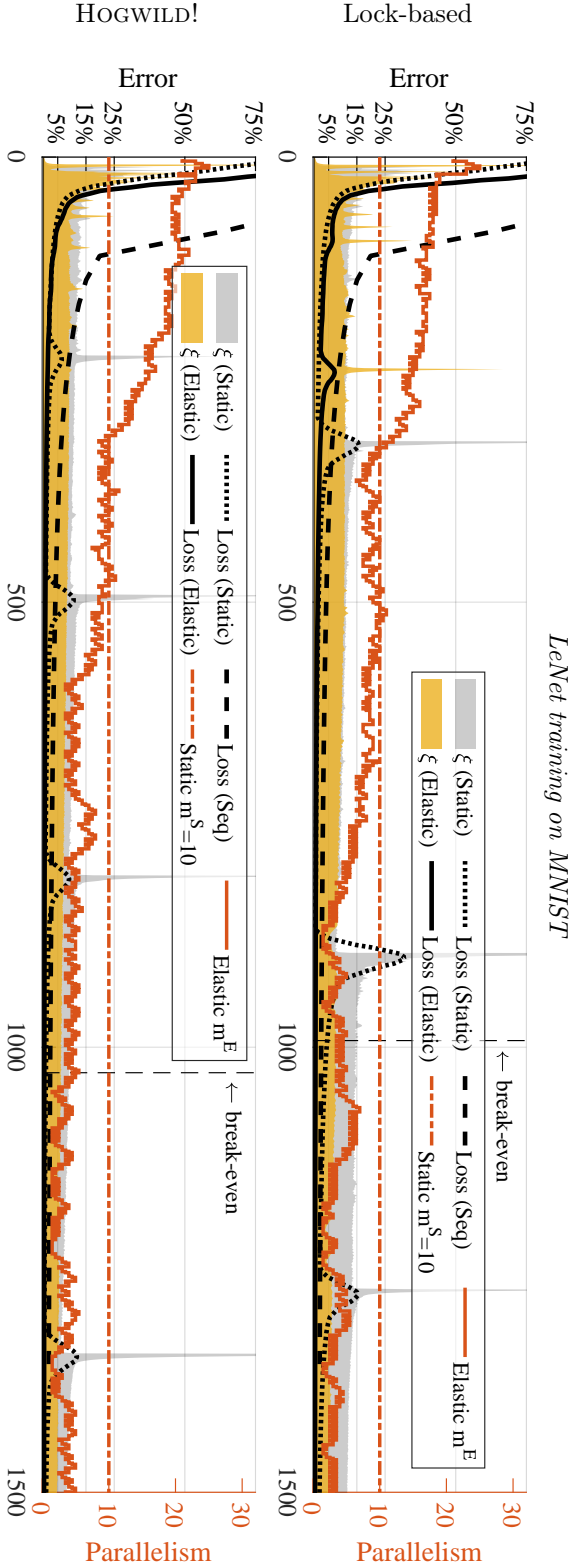
Figure D.5: Loss over time for LeNet training on MNIST, comparing *ElAsyncSGD* (Elastic) with best constant parallelism baselines ($m = 10$ for *Lock-based* and HOGWILD!).
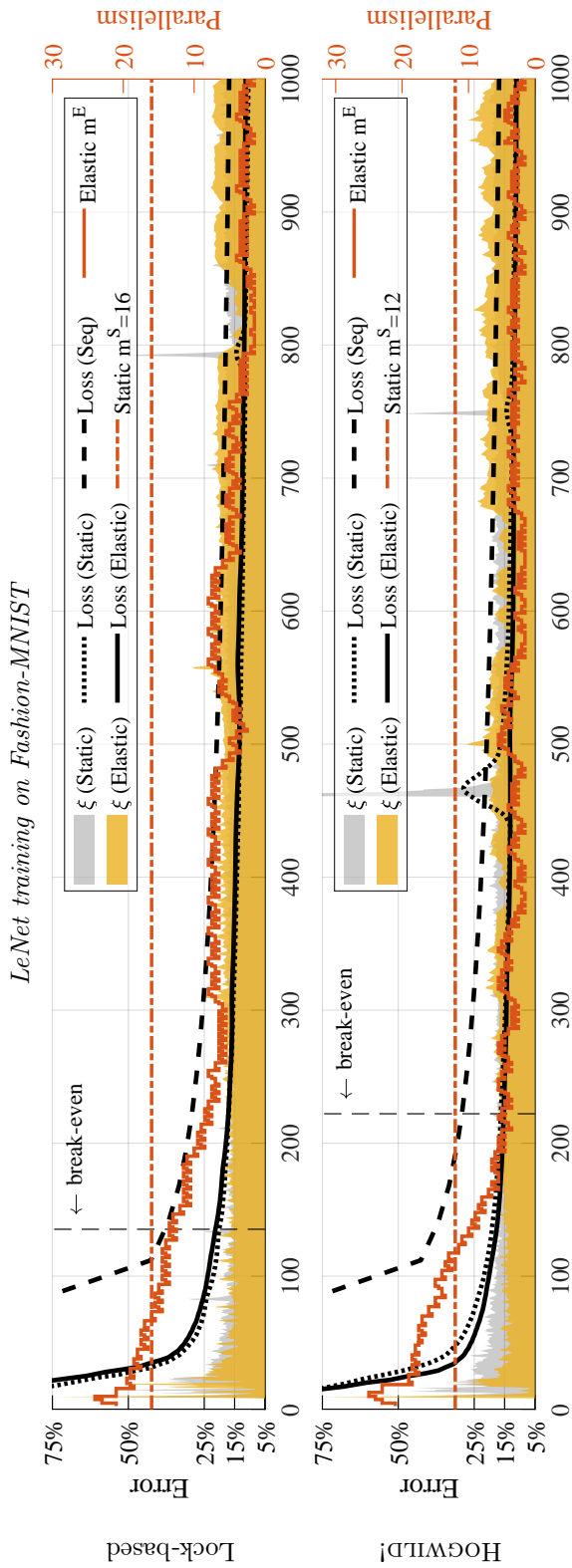
Figure D.6: Loss over time for LeNet training on Fashion-MNIST, comparing *ElAsyncSGD* (Elastic) with best constant parallelism baselines ($m = 16$ for *Lock-based*, $m = 12$ for HOGWILD!).
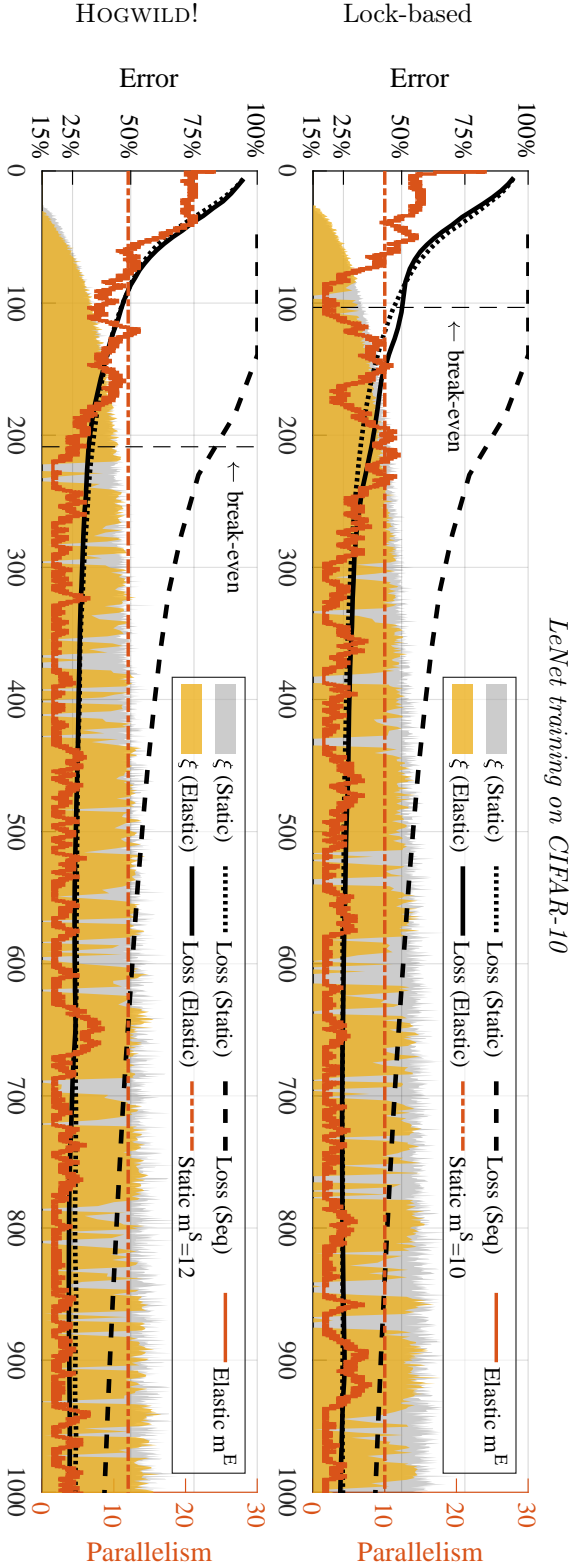
Figure D.7: Loss over time for LeNet training on Fashion–MNIST, comparing *ElAsyncSGD* (Elastic) with the best constant parallelism baselines ($m = 12$ for *Lock-based* and HOGWILD!).

Table D.2: Results overview. Best constant parallelism baseline in parenthesis, best value in bold.

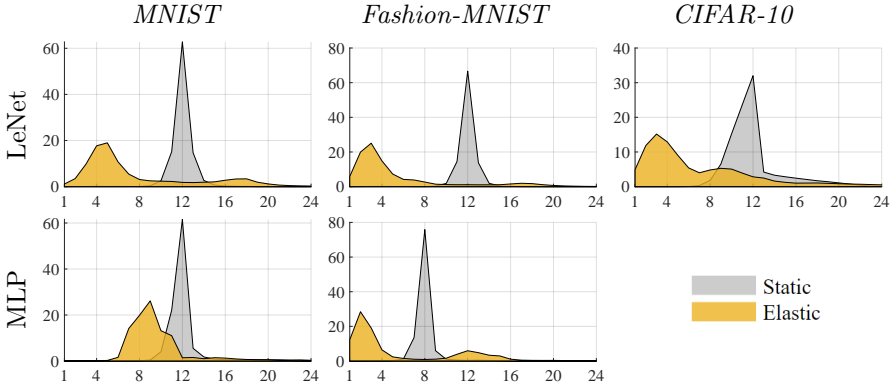| Setting			ε-convergence rate		Metrics		Computational resources	
Architecture	Dataset	Algorithm	ε = 50%	ε = 15%	Final loss	Avg ξ	Worker-seconds	Reduction
MLP	MNIST	Lock-based	9s (9s)	**13s** (14s)	**1.3e-04** (0.08)	**1.06** (2.82)	**1561s** (2160s)	**27.7%**
		HOGWILD!	9s (**8s**)	14s (**12s**)	1.1e-03 (0.02)	**1.37** (2.80)	**992s** (1440s)	**31%**
	Fashion-MNIST	Lock-based	14s (**10s**)	64s (**57s**)	**0.12** (0.37)	**3.06** (3.39)	**1313s** (4000s)	**67%**
		HOGWILD!	14s (**10s**)	64s (92s)	0.17 (**0.16**)	**2.63** (3.55)	**1628** (4000)	**59%**
LeNet	MNIST	Lock-based	29s (**18s**)	43s (**35s**)	**4.2e-03** (0.04)	**3.29** (4.19)	**10653s** (15000s)	**29%**
		HOGWILD!	26s (**19s**)	36s (**30s**)	**3.6e-03** (0.02)	**2.82** (3.48)	**10517s** (15000s)	**30%**
	Fashion-MNIST	Lock-based	32s (**29s**)	(**265s**) (355s)	**0.24** (0.26)	**5.65** (5.96)	**6723s** (16000s)	**58%**
		HOGWILD!	**25s** (29s)	239s (**300s**)	**0.25** (0.26)	**5.58** (5.88)	**4674** (12000s)	**61%**
				ε = 25%				
	CIFAR-10	Lock-based	104s (**93s**)	**626s** (∞)	**0.57** (0.60)	**12.83** (15.94)	**4344s** (10000s)	**57%**
		HOGWILD!	**86s** (93s)	**780s** (∞)	**0.56** (0.59)	**12.68** (15.26)	**4284s** (12000s)	**64%**

Figure D.8: Staleness distributions of standard constant-parallelism *AsyncSGD* (Static) and the proposed *ElAsyncSGD* (Elastic).

less oscillations. Despite the occasional deficit in 50%- and 15%-convergence, *ElAsyncSGD* persistently achieves a low final loss value, and overall lower $\|\xi\|$ across the board, due to its elastic parallelism control. Moreover, the overall computational resource consumption of *ElAsyncSGD* is drastically reduced (ranging from $30\% - 67\%$), compared to the best constant parallelism *AsyncSGD* baselines (see Figure D.3-D.7).

Discussion.   It can be observed that *ElAsyncSGD* typically accelerates in early stages of the execution, where the convergence rate is evidently generally less susceptible to AIN. The acceleration is particularly substantial for Hogwild! executions, compared to *Lock-based*. This is likely due to that *ElAsyncSGD* detects an increasing impact of lock-related bottlenecks on the convergence rate with higher parallelism, the influence of which Hogwild! lacks. As a consequence, the computational break-even time occurs later in the Hogwild! executions.   In addition, note that since *ElAsyncSGD* persistently achieves lower loss values, with more stable convergence trajectories, at a lower consumption of computing resources, we conclude that Hypothesis D.4.1 holds in the context of the evaluated applications.   Note that the aforementioned results are achieved immediately by *ElAsyncSGD*, as opposed to the baselines, which have been carefully tuned in order to maximally challenge the proposed *ElAsyncSGD*.
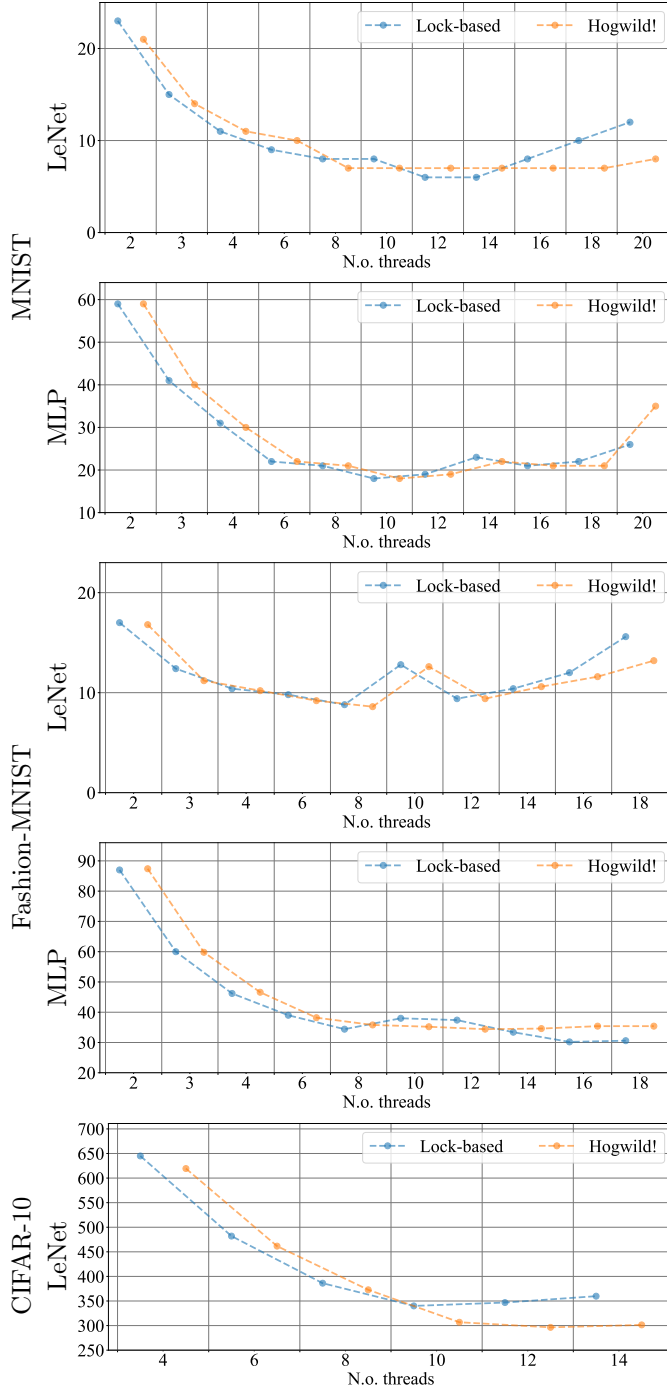
Figure D.9: Complete scalability evaluation of *AsyncSGD* for training MLP and LeNet on MNIST, Fashion-MNIST, and CIFAR10 to 50%-convergence, with the purpose of finding the optimal constant parallelism levels, used as baselines for benchmarking *ElAsyncSGD*.

D.7   Conclusions

We introduce *ElAsyncSGD*, an elastic extension of *AsyncSGD*, which dynamically regulates the parallelism level based on the execution instance, in order to balance the computational vs. statistical efficiency trade-off, as well as reduce the computational resources consumption. We formalize the notion of *asynchrony-induced noise* (AIN), we provide an efficient method to measure it accurately in real-time and establish analytically its crucial impact on the statistical convergence properties of *AsyncSGD* on general non-convex, as well as Polyak-Lojasiewicz, target functions. The proposed *ElAsyncSGD* exhibits far more stable, less volatile, convergence properties, compared to even the optimally tuned constant-parallelism static *AsyncSGD* baselines.   *ElAsyncSGD* entails drastic reduction in computational resources consumption, where the reduction in worker-seconds ranges between $30 - 67\%$, with the associated energy savings. Moreover, due to improved computational vs. statistical efficiency balance, *ElAsyncSGD* ensures overall lower staleness distributions, overall lower magnitudes of asynchrony-induced noise, and hence achieves more stable convergence, and converges to higher precision.

Bibliography

[1] A. M. Turing, "Computing machinery and intelligence," in *Parsing the turing test*. Springer, 2009, pp. 23–65.

[2] M. Campbell, A. Hoane, and F. hsiung Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, 2002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370201001291

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[4] A. D. Cohen, A. Roberts, and A. Molina, "Lamda: Language models for dialog applications," in *arXiv*, 2022.

[5] Z. Jan, F. Ahamed, W. Mayer, N. Patel, G. Grossmann, M. Stumptner, and A. Kuusk, "Artificial intelligence for industry 4.0: Systematic review of applications, challenges, and opportunities," *Expert Systems with Applications*, p. 119456, 2022.

[6] C.-J. Wu, R. Raghavendra, U. Gupta, B. Acun, N. Ardalani, K. Maeng, G. Chang, F. Aga, J. Huang, C. Bai *et al.*, "Sustainable ai: Environmental implications, challenges and opportunities," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 795–813, 2022.

[7] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," *CoRR*, vol. abs/1906.02243, 2019. [Online]. Available: http://arxiv.org/abs/1906.02243

[8] ——, "Energy and policy considerations for modern deep learning research," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 09, pp. 13 693–13 696, Apr. 2020. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/7123

[9] K. Hao, "Training a single AI model can emit as much carbon as five cars in their lifetimes," June 2019, [Online; posted 6-June-2019]. [Online]. Available: https://www.technologyreview.com/2019/06/06/239031/training-a-single-ai-model-can-emit-as-much-carbon-as-five-cars-in-their-lifetimes/

[10] D. Amodei, Dario; Hernandez, "AI and compute," May 2018, [Online; posted 16-May-2018]. [Online]. Available: https://openai.com/research/ai-and-compute

[11] "Total data volume worldwide 2010-2025," https://www.statista.com/statistics/871513/worldwide-data-created/, accessed: 2023-03-31.

[12] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. D. Kubiatowicz, E. A. Lee, N. Morgan, G. Necula, D. A. Patterson *et al.*, "The parallel computing laboratory at uc berkeley: A research agenda based on the berkeley view," *EECS Department, University of California, Berkeley, Tech. Rep*, 2008.

[13] J.-M. Frangos, "The internet of things will power the fourth industrial revolution. here's how," June 2017, [Online; posted 24-June-2017]. [Online]. Available: https://www.weforum.org/agenda/2017/06/internet-of-things-will-power-the-fourth-industrial-revolution/

[14] J. Tsitsiklis, D. Bertsekas, and M. Athans, "Distributed asynchronous deterministic and stochastic gradient optimization algorithms," *IEEE transactions on automatic control*, vol. 31, no. 9, pp. 803–812, 1986.

[15] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Int'l Conf. on Machine Learning*, 2013, pp. 1139–1147.

[16] Y. Ma, F. Rusu, and M. Torres, "Stochastic gradient descent on modern hardware: Multi-core cpu or gpu? synchronous or asynchronous?" in *2019 IEEE Int'l Parallel and Distributed Proc. Symp. (IPDPS)*. IEEE, 2019, pp. 1063–1072.

[17] R. Gupta and T. Alam, "Survey on federated-learning approaches in distributed environment," *Wireless Personal Communications*, vol. 125, no. 2, pp. 1631–1652, 2022.

[18] S. Li, T. Ben-Nun, S. D. Girolamo, D. Alistarh, and T. Hoefler, "Taming unbalanced training workloads in deep learning with partial collective operations," in *25th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2020, pp. 45–61.

[19] D. Alistarh, B. Chatterjee, and V. Kungurtsev, "Elastic consistency: A general consistency model for distributed stochastic gradient descent," *arXiv preprint arXiv:2001.05918*, 2020.

[20] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Inf. Proc. Systems (NIPS) 24*. Curran Associates, Inc., 2011, pp. 693–701.

[21] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[22] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural inf. proc. systems*, 2010, pp. 2595–2603.

[23] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Advances in neural information processing systems*, 2013, pp. 1223–1231.

[24] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-sgd for distributed deep learning," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16.   AAAI Press, 2016, p. 2350–2356.

[25] S. Chaturapruek, J. C. Duchi, and C. Ré, "Asynchronous stochastic convex optimization: the noise is in the noise and SGD don't care," in *Advances in Neural Inf. Proc. Systems*, 2015, pp. 1531–1539.

[26] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*.   PMLR, 2017, pp. 1273–1282.

[27] S. Gupta, W. Zhang, and F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*.   IEEE, 2016, pp. 171–180.

[28] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[29] X. Zhao, A. An, J. Liu, and B. X. Chen, "Dynamic stale synchronous parallel distributed training for deep learning," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, July 2019, pp. 1507–1517.

[30] F. He, T. Liu, and D. Tao, "Control batch size and learning rate to generalize well: Theoretical and empirical evidence," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[31] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv:1609.04836*, 2016.

[32] D. Mishkin, N. Sergievskiy, and J. Matas, "Systematic evaluation of convolution neural network advances on the imagenet," *Computer Vision and Image Understanding*, vol. 161, pp. 11–19, 2017.

[33] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and distributed computation: numerical methods*.   Prentice Hall, 1989, vol. 23.

[34] ——, "Some aspects of parallel and distributed iterative algorithms—a survey," *Automatica*, vol. 27, no. 1, pp. 3–21, 1991.

[35] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu, "Asynchronous stochastic gradient descent with delay compensation," in *International Conference on Machine Learning*.   PMLR, 2017, pp. 4120–4129.

[36] H. Zhang, C.-J. Hsieh, and V. Akella, "Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*.   IEEE, 2016, pp. 629–638.

[37] J. L. Rosenfeld, "A case study in programming for parallel-processors," *Communications of the ACM*, vol. 12, no. 12, pp. 645–655, 1969.

[38] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear algebra and its applications*, vol. 2, no. 2, pp. 199–222, 1969.

[39] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous parallel stochastic gradient for nonconvex optimization," in *Advances in Neural Inf. Proc. Systems*, 2015, pp. 2737–2745.

[40] I. Mitliagkas, C. Zhang, S. Hadjis, and C. Ré, "Asynchrony begets momentum, with an application to deep learning," in *54th Annual Allerton Conf. on Communication, Control, and Computing*. IEEE, 2016, pp. 997–1004.

[41] 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, 2017.

[42] S. Sallinen, N. Satish, M. Smelyanskiy, S. S. Sury, and C. Ré, "High performance parallel stochastic gradient descent in shared memory," in *IEEE Int'l Parallel and Distr. Proc. Symp.* IEEE, 2016, pp. 873–882.

[43] L. M. Nguyen, P. H. Nguyen, M. van Dijk, P. Richtárik, K. Scheinberg, and M. Takáč, "SGD and hogwild! convergence without the bounded gradients assumption," *arXiv preprint arXiv:1802.03801*, 2018.

[44] C. M. De Sa, C. Zhang, K. Olukotun, C. Ré, and C. Ré, "Taming the wild: A unified analysis of hogwild-style algorithms," in *Advances in Neural Inf. Proc. Systems 28*. Curran Associates, Inc., 2015, pp. 2674–2682. [Online]. Available: http://papers.nips.cc/paper/5717-taming-the-wild-a-unified-analysis-of-hogwild-style-algorithms.pdf

[45] D. Alistarh, C. De Sa, and N. Konstantinov, "The convergence of stochastic gradient descent in asynchronous shared memory," in *ACM Symp. on Principles of Distributed Computing*, ser. PODC '18. New York, NY, USA: ACM, 2018, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/3212734.3212763

[46] J. Wei, G. A. Gibson, P. B. Gibbons, and E. P. Xing, "Automating dependence-aware parallelization of machine learning training on distributed shared memory," in *14th EuroSys Conf. 2019*, 2019, pp. 1–17.

[47] F. Lopez, E. Chow, S. Tomov, and J. Dongarra, "Asynchronous SGD for DNN training on shared-memory parallel architectures," in *Int'l Parallel and Distr. Proc. Symp. Workshops (IPDPSW)*. IEEE, 2020, pp. 1–4.

[48] S. U. Stich, "Local SGD converges fast and communicates little," in *Int'l Conf. on Learning Representations (ICLR)*, 2019.

[49] Z. Jiang, A. Balu, C. Hegde, and S. Sarkar, "Collaborative deep learning in fixed topology networks," in *Advances in Neural Inf. Proc. Systems*, 2017, pp. 5904–5914.

[50] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous decentralized parallel stochastic gradient descent," in *Int'l Conf. on Machine Learning*. PMLR, 2018, pp. 3043–3052.

[51] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *Advances in Neural Inf. Proc. Systems*, 2011, pp. 873–881.

[52] B. McMahan and M. Streeter, "Delay-tolerant algorithms for asynchronous distributed online learning," in *Advances in Neural Inf. Proc. Systems 27*. Curran Associates, Inc., 2014, pp. 2915–2923. [Online]. Available: http://papers.nips.cc/paper/ 5242-delay-tolerant-algorithms-for-asynchronous-distributed-online-learning. pdf

[53] S. Sra, A. W. Yu, M. Li, and A. J. Smola, "Adadelay: Delay adaptive distributed stochastic convex optimization," *arXiv preprint arXiv:1508.05003*, 2015.

[54] C. Xu, Y. Qu, Y. Xiang, and L. Gao, "Asynchronous federated learning on heterogeneous devices: A survey," *arXiv preprint arXiv:2109.04269*, 2021.

[55] K. Yazdani and M. Hale, "Asynchronous parallel nonconvex optimization under the polyak-łojasiewicz condition," *IEEE Control Systems Letters*, 2021.

[56] G. Damaskinos, R. Guerraoui, A.-M. Kermarrec, V. Nitu, R. Patra, and F. Taiani, "Fleet: Online federated learning via staleness awareness and performance prediction," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 163–177.

[57] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[58] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th Symp. on Operating Systems Design and Implementation*, 2014, pp. 583–598.

[59] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding," in *Advances in Neural Inf. Proc. Systems*, 2017, pp. 1709–1720.

[60] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler, "A modular benchmarking infrastructure for high-performance and reproducible deep learning," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 66–77.

[61] Z. Ren, Z. Zhou, L. Qiu, A. Deshpande, and J. Kalagnanam, "Delay-adaptive distributed stochastic optimization," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 5503–5510.

[62] H. Karimi, J. Nutini, and M. Schmidt, "Linear convergence of gradient and proximal-gradient methods under the polyak-łojasiewicz condition," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2016, pp. 795–811.

[63] Z. Charles and D. Papailiopoulos, "Stability and generalization of learning algorithms that converge to global optima," in *International Conference on Machine Learning*. PMLR, 2018, pp. 745–754.

[64] P. J. Werbos, "Applications of advances in nonlinear sensitivity analysis," in *System modeling and optimization*. Springer, 1982, pp. 762–770.

[65] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Proc. of the 11th European Conf. on Computer Systems*. ACM, 2016, p. 4.

[66] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," in *Advances in neural information processing systems*, 2017, pp. 1509–1519.

[67] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[68] P. Greengard and V. Rokhlin, "An algorithm for the evaluation of the incomplete gamma function," *Advances in Computational Mathematics*, vol. 45, no. 1, pp. 23–49, 2019.

[69] D. Alistarh, T. Hoefler, M. Johansson, N. Konstantinov, S. Khirirat, and C. Renggli, "The convergence of sparsified gradient methods," in *Advances in Neural Information Processing Systems*, 2018, pp. 5977–5987.

[70] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[71] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.

[72] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv:1609.04747*, 2016.

[73] T. Liu, S. Li, J. Shi, E. Zhou, and T. Zhao, "Towards understanding acceleration tradeoff between momentum and asynchrony in nonconvex stochastic optimization," in *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., 2018, pp. 3686–3696.

[74] A. Agarwal, M. J. Wainwright, and J. C. Duchi, "Distributed dual averaging in networks," in *Advances in Neural Inf. Proc. Systems*, 2010, pp. 550–558.

[75] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao, "Optimal distributed online prediction," in *28th Int'l Conf. on Machine Learning (ICML-11)*, 2011, pp. 713–720.

[76] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-SGD for distributed deep learning," *arXiv preprint arXiv:1511.05950*, 2015.

[77] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On model parallelization and scheduling strategies for distributed machine learning," in *Advances in Neural Inf. Proc. Sys.*, 2014, pp. 2834–2842.

[78] Y. Ma, F. Rusu, and M. Torres, "Stochastic gradient descent on highly-parallel architectures," *arXiv preprint arXiv:1802.08800*, 2018.

[79] J. C. Duchi, S. Chaturapruek, and C. Ré, "Asynchronous stochastic convex optimization," *arXiv preprint arXiv:1508.00882*, 2015.

[80] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar, "Slow and stale gradients can win the race: Error-runtime trade-offs in distributed SGD," *arXiv preprint arXiv:1803.01113*, 2018.

[81] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, "An empirical study of common challenges in developing deep learning applications," in *30th Int'l Symp. on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 104–115.

[82] H. Zhang, C.-J. Hsieh, and V. Akella, "Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent," in *16th Int'l Conf. on Data Mining (ICDM)*. IEEE, 2016, pp. 629–638.

[83] A. Krizhevsky, I. Sutskever, and G. Hinton, "2012 alexnet," *Adv. Neural Inf. Process. Syst.*, pp. 1–9, 2012.

[84] A. Larsson, A. Gidenstam, P. H. Ha, M. Papatriantafilou, and P. Tsigas, "Multi-word atomic read/write registers on multiprocessor systems," in *European Symp. on Algorithms*. Springer, 2004, pp. 736–748.

[85] M. Ianni, A. Pellegrini, and F. Quaglia, "Anonymous readers counting: A wait-free multi-word atomic register algorithm for scalable data sharing on multi-core machines," *Trans. on Parallel and Distributed Systems*, vol. 30, no. 2, pp. 286–299, 2018.

[86] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[87] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[88] Y. Qiu, "Minidnn," https://github.com/yixuan/MiniDNN/, 2020.

[89] Y. LeCun and C. Cortes, "MNIST handwritten digit database," *Dat repository*, 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[90] D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, "A study of the behavior of synchronization methods in commonly used languages and systems," in *27th Int'l Symp. on Parallel and Distributed Proc.* IEEE, 2013, pp. 1309–1320.

[91] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proc. 24th on Operating Systems Principles*, 2013, pp. 33–48.

[92] V. Gulisano, Y. Nikolakopoulos, D. Cederman, M. Papatriantafilou, and P. Tsigas, "Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types," *ACM Trans. on Parallel Computing (TOPC)*, vol. 4, no. 2, pp. 1–28, 2017.

[93] M. Zinkevich, J. Langford, and A. Smola, "Slow learners are fast," *Advances in neural information processing systems*, vol. 22, 2009.

[94] T. Kraska, "Towards instance-optimized data systems, keynote," *2021 International Conference on Very Large Data Bases*, 2021.

[95] R. Z. Aviv, I. Hakimi, A. Schuster, and K. Y. Levy, "Asynchronous distributed learning: Adapting to gradient delays without prior knowledge," in *International Conference on Machine Learning.* PMLR, 2021, pp. 436–445.

[96] F. Haddadpour, M. M. Kamani, M. Mahdavi, and V. Cadambe, "Local sgd with periodic averaging: Tighter analysis and adaptive synchronization," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[97] K. Bäckström, M. Papatriantafilou, and P. Tsigas, "Mindthestep-asyncpsgd: Adaptive asynchronous parallel stochastic gradient descent," in *2019 IEEE International Conference on Big Data (Big Data).* IEEE, 2019, pp. 16–25.

[98] C. Xie, S. Koyejo, and I. Gupta, "Zeno++: Robust fully asynchronous sgd," in *International Conference on Machine Learning.* PMLR, 2020, pp. 10 495–10 503.

[99] S. Ghadimi and G. Lan, "Stochastic first-and zeroth-order methods for nonconvex stochastic programming," *SIAM Journal on Optimization*, vol. 23, no. 4, pp. 2341–2368, 2013.

[100] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[101] K. Bäckström, "shared-memory-sgd," https://github.com/dcs-chalmers/shared-memory-sgd, 2021.

[102] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[103] M. Sorbaro, Q. Liu, M. Bortone, and S. Sheik, "Optimizing the energy consumption of spiking neural networks for neuromorphic applications," *Frontiers in neuroscience*, p. 662, 2020.

[104] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, 2014, pp. 571–582.

[105] S. Hadjis, C. Zhang, I. Mitliagkas, D. Iter, and C. Ré, "Omnivore: An optimizer for multi-device deep learning on cpus and gpus," *arXiv preprint arXiv:1606.04487*, 2016.

[106] H. Lin, H. Zhang, Y. Ma, T. He, Z. Zhang, S. Zha, and M. Li, "Dynamic mini-batch sgd for elastic distributed training: learning in the limbo of resources," *arXiv preprint arXiv:1904.12043*, 2019.

[107] G. Nadiradze, I. Markov, B. Chatterjee, V. Kungurtsev, and D. Alistarh, "Elastic consistency: A practical consistency model for distributed stochastic gradient descent," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 10, 2021, pp. 9037–9045.