Thesis for The Degree of Licentiate of Engineering

Adaptiveness and Lock-free Synchronization in Parallel Stochastic Gradient Descent

Karl Bäckström

Division of Networks and Systems
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2021

Adaptiveness and Lock-free Synchronization in Parallel Stochastic Gradient Descent

Karl Bäckström

Copyright ©2021 Karl Bäckström
except where otherwise stated.
All rights reserved.

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

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2021.

"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 Hennessy, Stanford President (2006)

Abstract

The emergence of big data in recent years due to the vast societal digitalization and large-scale sensor deployment has entailed significant interest in machine learning methods to enable automatic data analytics. In a majority of the learning algorithms used in industrial as well as academic settings, the first-order iterative optimization procedure Stochastic Gradient Descent (SGD), is the backbone. However, SGD is often time-consuming, as it typically requires several passes through the entire dataset in order to converge to a solution of sufficient quality.

In order to cope with increasing data volumes, and to facilitate accelerated processing utilizing contemporary hardware, various parallel SGD variants have been proposed. In addition to traditional synchronous parallelization schemes, asynchronous ones have received particular interest in recent literature due to their improved ability to scale due to less coordination, and subsequently waiting time. However, asynchrony implies inherent challenges in understanding the execution of the algorithm and its convergence properties, due the presence of both stale and inconsistent views of the shared state.

In this work, we aim to increase the understanding of the convergence properties of SGD for practical applications under asynchronous parallelism, and develop tools and frameworks that facilitate improved convergence properties as well as further research and development. First, we focus on understanding the impact of staleness, and introduce models for capturing the dynamics of parallel execution of SGD. This enables (i) quantifying the statistical penalty on the convergence due to staleness and (ii) deriving an adaptation scheme, introducing a staleness-adaptive SGD variant *MindTheStep-AsyncSGD*, which provably reduces this penalty. Second, we aim at exploring the impact of synchronization mechanisms, in particular consistency-preserving ones, and the overall effect on the convergence properties. To this end, we propose *Leashed-SGD*, an extensible algorithmic framework supporting various synchronization mechanisms for different degrees of consistency, enabling in particular a lock-free and consistency-preserving implementation. In addition, the algorithmic construction of *Leashed-SGD* enables dynamic memory allocation, claiming memory only when necessary, which reduces the overall memory footprint. We perform an extensive empirical study, benchmarking the proposed methods, together with established baselines, focusing on the prominent application of Deep Learning for image classification on the benchmark datasets MNIST and CIFAR, showing significant improvements in converge time for *Leashed-SGD* and *MindTheStep-AsyncSGD*.

Keywords

Stochastic gradient descent, parallelism, machine learning

Acknowledgment

The most sincere gratitude is owed to my supervisors Marina Papatriantafilou, Philippas Tsigas and Vincenzo Gulisano. They have provided unconditional, ambitious 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, Lucas, Magnus, Masoud, Nasser, Olaf, Oliver, Parthasarathy, Romaric, Sólrún, Thomas, Valentin, 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 and insightful discussions which have provided invaluable advice.

An especially warm thanks is directed 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 all of my accomplishments in both education and career, and this thesis in particular. And, of course, to my dear Rita, for your unconditional support and remarkable patience.

I would like to acknowledge the Wallenberg AI, Autonomous Systems and Software Program (WASP) for providing financial support, and arranging several international study visits, which have provided much inspiration and joy.

List of Publications

Appended publications

This thesis is based on the following publications:

[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 (to appear, 10 pages).

Other publications

The following publications were published during my PhD studies, or are currently in submission/under revision. However, they are not appended to this thesis, due to contents overlapping that of appended publications or contents not related to the thesis.

[a] 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

[b] 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

[c] 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

Contents

Abstract **v**

Acknowledgement **vii**

List of Publications **ix**

1 Overview **1**
1.1 SGD and Artificial Neural Networks 3
1.1.1 Metrics of interest 6
1.2 Parallel SGD 7
1.2.1 Synchronous parallel SGD 9
1.2.2 Asynchronous parallel SGD 10
1.3 Problems and challenges 13
1.3.1 Scalability 13
1.3.2 Convergence under asynchrony 14
1.3.3 Benchmarking and evaluation 15
1.4 Thesis contributions 16
1.4.1 **Paper A**: Convergence of staleness-adaptive SGD . . . 16
1.4.2 **Paper B**: Framework for lock-freedom and consistency 17
1.5 Conclusions 18

2 PAPER A 21
2.1 Introduction 22
2.2 Preliminaries 24
2.2.1 Stochastic Gradient Descent 24
2.2.2 System Model and Asynchronous SGD 24
2.2.3 Momentum 25
2.3 On the scalability of Sync-PSGD 25
2.4 The proposed framework 27
2.4.1 The *MindTheStep-AsyncSGD* Framework 27
2.4.2 Tuning the impact of asynchrony 27
2.5 Convex convergence analysis 31
2.6 Experimental study 33
2.7 Related work 35
2.8 Conclusions and Future Work 37

A Appendix - Paper A **39**

3 PAPER B 45
3.1 Introduction 46
3.2 Preliminaries 48
3.2.1 SGD and DL 48
3.2.2 System Model 49
3.2.3 Synchronization methods and consistency 50
3.2.4 Problem overview 50
3.3 The *Leashed-SGD* framework 50
3.3.1 Introducing *ParameterVector* 51
3.3.2 Baselines outline 52
3.3.3 *Leashed-SGD*: Lock-free consistent *AsyncSGD* 52
3.4 Contention and staleness 55
3.4.1 Dynamics of *Leashed-SGD* 55
3.4.2 Persistence analysis 56
3.5 Evaluation 57
3.5.1 Implementation 57
3.5.2 Experiment setup 59
3.5.3 Experiment outcomes 64
3.5.4 Summary of outcomes 65
3.6 Related Work 65
3.7 Conclusions 66

B Appendix - Paper B **69**

Bibliography **75**

List of Figures

1.1 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 . 4

1.2 The input $\mathbf{o}^{(0)}$ to an ANN undergoes several transformations, parameterized by θ. 5

1.3 In *SyncSGD* the threads' individual gradients are aggregated by averaging, after which a global iteration is performed. *SyncSGD* essentially corresponds to parallelization on the gradient computation level. 9

1.4 *AsyncSGD* parallelizes the SGD iterations, allowing asynchronous read (R) and update (U) operations on the shared state. 10

2.1 CNN architecture; Four convolutional layers with 3×3 kernels, with intermediate MaxPool layers. 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. 33

2.2 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. 34

2.3 *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. 35

3.1 Convergence rate is the product of computational and statistical efficiency, sensitive to hyper-parameters tuning. We show the significant impact of lock-free synchronization on these factors and on reducing the dependency on tuning, enabling improved convergence. 49

3.2 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_t only through a read operation, then computes the update at a new memory location, becoming a candidate for $\theta_{t+\tau}$ 55
3.3 ϵ-convergence rate for MLP training under varying parallelism, measuring wall-clock time until reaching an error threshold. . . 58
3.4 Computational efficiency for MLP training under varying parallelism. 59
3.5 ϵ-convergence rate for MLP with $m = 16$ threads to high precision (*top*), $m = 34$ threads (*middle*) and maximum parallelism $m = 68$ threads (*bottom*). 60
3.6 MLP training progress over time with $m = 16$ threads (*top*), with $m = 34$ threads (*middle*) and maximum parallelism $m = 68$ threads (*bottom*). 61
3.7 Staleness distribution for MLP with $m = 16$ threads (*top*), $m = 34$ threads (*middle*) and maximum parallelism $m = 68$ threads (*bottom*). 62
3.8 ϵ-convergence rates to different precision for CNN training with $m = 16$ threads 63

B.1 Step size tuning (top), confirming the choice $\eta = 0.005$, and statistical efficiency (bottom), showing 50%-convergence 72
B.2 Gradient computation and parameter update times T_c, T_u (top, bottom, respectively) for MLP and CNN 73
B.3 Memory consumption measured continuously on second granularity for MLP (top) and CNN (bottom) 73

Chapter 1

Overview

"I propose to consider the question, 'Can machines think?' " are the opening words of Alan Turing in the article *Computing Machinery and Intelligence*, published 1950 in Mind [1]. Realizing swiftly the danger of posing such an, at the time, controversial question without hope to ever provide a reasonable definition of "think" (or even "machine" for that matter) Turing circumvents this issue simply by defining a game:

The *Imitation Game*, or the more widely known *Turing Test*, 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".

Undoubtedly, Turing´s prediction cannot be argued to have been fulfilled, although significant steps have been taken towards automation by the use of artificially intelligent machines. In recent years, Artificial Intelligence (AI) has proven very effective, often exceeding human performance, in many application domains, including natural language processing, image analysis and speech recognition [2]. For example, it is estimated that during 2020, 97% of mobile phone users are using AI-powered voice assistants.

In addition, vast digitalization of society in recent years has enabled wide deployment of sensing technology, which has made available huge amounts of information, sparking the research within Big Data. Subsequently, a primary focus of AI and Machine Learning (ML) technologies have been towards data-driven methods, meaning created with the help of, and used for analyzing, data. Among the most prominent is the widely used Stochastic Gradient Descent (SGD) optimization algorithm, the credit for which is usually attributed Augustin-Louis Cauchy who first proposed it in 1847. By iteratively processing data, SGD enables Artificial Neural Network (ANN) training, Logistic Regression, Support Vector Machines, and other ML methods. The emergence of Big Data research in recent years has not only affected the development in terms of software, but to a high degree also hardware which is equipped with a growing

number of processing cores in order to accommodate for growing demand on data analytics to scale [3].

The quote in the initial pages of this thesis, by the previous President of Stanford John 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 [4]. The new *Moore's law* is to double the number of processor per chip in each new generation of technology. Ever since, it has been the privilege of the software development community to solve the numerous problems arising in the field of parallel programming, in the strive to fully utilize the capability of the new standard of processors. The interest in the field has increased rapidly due to the significant performance gains that it entails, 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.

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

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

In this thesis, we contribute to the domain of efficient parallel optimization with SGD for fast and stable convergence in prominent machine learning applications. We target the aforementioned critical questions, exploring aspects of synchronization, consistency, staleness and parallel-aware adaptiveness, focus-

ing on the impact on the overall convergence. We introduce analytical models that capture features of interest in multi-thread dynamics, providing critical clues for the algorithm development. The theoretical work is complemented by empirical results, benchmarking the proposed methods compared to established baselines for common practical applications that are widely used in both academic and industrial settings.

In Table 1.1 we provide a comprehensive list of variable notation to be used throughout the overview chapter of this thesis.

1.1 SGD and Artificial Neural Networks

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

$$\underset{\theta}{\text{minimize}} \quad f_D(\theta) \tag{1.1}$$

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

$$\theta_{t+1} = \theta_t - \eta \widetilde{\nabla} f_B(\theta_t) \tag{1.2}$$

Table 1.1: List of symbols used in the overview chapter of this thesis

Topic	***Symbol***	***Meaning***
Parallelism	m	Number of threads
	Δ	Update to the shared state by a thread
	τ	Staleness, defined as the number of applied concurrent updates
	$\hat{\tau}$	Assumed maximum system staleness $\hat{\tau} = \max_t \tau_t$
	v_t	The *view* of a thread when applying an update, $v_t = \theta_{t-\tau_t}$
	m_C^*	Computational saturation point
	m_S^*	System saturation point
Optimization	f	Non-negative target function
	ϵ	Precision threshold for convergence
	θ_t	Vector of trainable parameters (weights and bias) at iteration t
	d	Equals $\|\theta\|$, i.e. optimization problem dimension
	D	Dataset used for training
	B	Mini-batch of data drawn randomly from D
	b	Mini-batch size $b = \|B\|$
	η	Step size
	$\widetilde{\nabla} f_B(\theta)$	Stochastic gradient of f, computed w.r.t. batch B and parameters θ
	μ	Momentum parameter
ANN	$y(x : \theta)$	Output computed based on input x and parameters θ
	$\hat{y}(x)$	Label for x
	$\mathbf{o}^{(l)}$	Output vector of layer l
	σ	Non-linear activation function

where $\widetilde{\nabla} f_B(\theta_t)$ and an unbiased estimate of the true gradient. The choice of the initialization point θ_0 is chosen at random according to some distribution, which as one might expect may significantly impact the convergence [5].

The negative gradient of a function constitutes the direction of *steepest descent*, resulting in a trajectory corresponding to the slope of the target function (see Figure 1.1). The iteration (1.2) is repeated until a solution θ^* of sufficient quality is found, i.e. $f_D(\theta^*) < \epsilon$, referred to as ϵ-convergence.

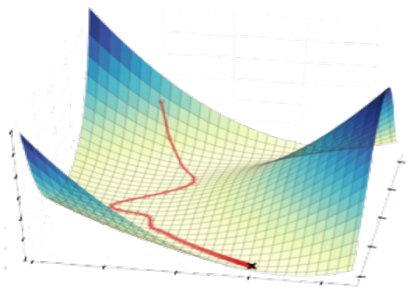

Figure 1.1: Stochastic Gradient Descent is a first-order iterative numerical optimization algorithm which repeatedly steps in the direction of steepest descent, following the slope of the target function

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

$$\theta_{t+1} \leftarrow \theta_t + \mu(\theta_t - \theta_{t-1}) - \eta \widetilde{\nabla} f_B(\theta_t) \tag{1.3}$$

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

Deep Learning (DL), i.e. deep Artificial Neural Network (ANN) training, constitutes one of the most important applications of SGD in present day. ANNs are computational structures inspired by the biological brain and consist of many fundamental units referred to as neurons. An ANN consists of several non-linear transformations, known as *layers*, processing the input data in consecutive steps (see Figure 1.2). Each layer is parameterized by a *weight*

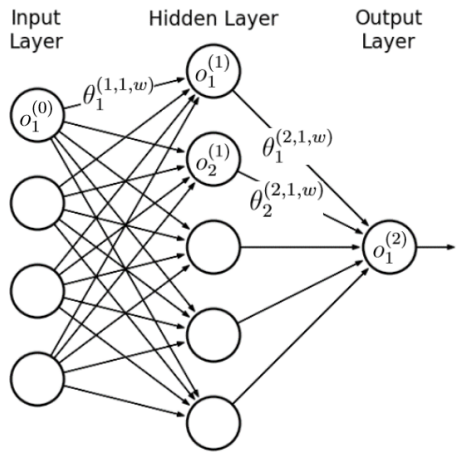

Figure 1.2: The input $\mathbf{o}^{(0)}$ to an ANN undergoes several transformations, parameterized by θ.

matrix and a *bias* vector, both constituting part of the parameter array θ, which is learned through the optimization process (1.2). 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 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:

$$o_n^{(l)} = \sigma\left(\sum_{i=0}^{|N_{l-1}|-1} \theta_i^{(l,n,w)} \cdot o_i^{(l-1)} + \theta^{(l,n,b)}\right)$$

  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:

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

  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 $\sigma_i(x) = e^{x_i} / \sum_{j=1}^{|x|} e^{x_j}$ (e being Euler's number) for each output neuron i, is often used for classification problems. Its output satisfies the requirements on a probability distribution function, and is consequently interpreted as such in this context, i.e. the estimated class distribution y of an input x. 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:

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

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 $f_D(\theta) = \frac{1}{|D|} \sum_{x \in D} L(\hat{y}, y(x : \theta))$. The *BackProp* algorithm is used for computing $\nabla_\theta f(\theta)$, 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.

1.1.1 Metrics of interest

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 [6] (and **Paper 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

The overall *convergence rate*, i.e. the wall-clock time until ϵ-convergence, is the most relevant in practice. As also pointed out in [6], the convergence rate is essentially the product:

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

Consequently, when proposing new algorithms (or altering existing ones) in this application domain that potentially change the *computational efficiency*, it is not sufficient to evaluate the invention by measuring only the *statistical efficiency*, i.e. counting the iterations until convergence. One must in general

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

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 we shall see in the following.

1.2 Parallel SGD

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. While parallelism can improve computational efficiency, simply by managing to apply a greater number of updates in each unit of time, the impact on the statistical efficiency, and thereby the overall convergence rate, is unpredictable. This is mainly due to 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 is consequently not trivial, and requires synchronization in every iteration (prior to applying an update) in order to not break the original sequential semantics of SGD. Alternatively, threads can execute the SGD algorithm, i.e. accessing and updating the shared state θ, asynchronously, although this might not conform to the sequential semantics. These approaches correspond to two main directions of methods for parallel SGD, referred to as *synchronous* and *asynchronous*. These can be further categorized into several representative methods, depending on their requirements on synchronization, staleness, and other consistency and progress guarantees.

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

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 [7,8], while in other contexts lock-freedom refers to operations on the shared state [9]. These terms are used in specific contexts, and refer to different levels of algorithmic implementations of objects and methods related to SGD. 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),

which we refer 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. Different works typically explore one of these dimensions, or the other, and it is important to adopt a consistent notation which distinguishes the two.

To this end, we introduce the following notation to distinguish between the two dimensions. 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 threads, once-in-a-while synchronization, etc. An algorithm which does not employ such aggregation is referred to as *asynchronous* (Figure. 1.4), as opposed to *synchronous* (Figure. 1.3) ones which e.g. aggregate updates by averaging them in a lock-step manner.
>
> Δ-*Consistency* refers to conformity to a sequential execution. An algorithm that allows staleness is consequently not strictly consistent with a sequential execution.

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.

For the second dimension, which relates to the fine-grained synchronization for operations on the shared state θ, we adopt the following notation which aims to conform to standard notation (summarized in Table 1.2) in established literature on concurrent implementations of shared data objects [10]:

> 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, we shall use the terms *progress* and *consistency* 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*.

Table 1.2: Brief overview of the adopted convention on notation and terminology regarding properties of concurrent operations.

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

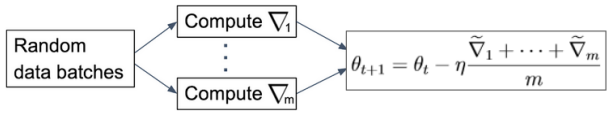

Figure 1.3: In *SyncSGD* the threads' individual gradients are aggregated by averaging, after which a global iteration is performed. *SyncSGD* essentially corresponds to parallelization on the gradient computation level.

Table 1.3 provides an overview of some of the representative methods available in the literature which are relevant within the scope of this thesis, and more detailed descriptions of these are provided in the following (Section 1.2.1 and 1.2.2).

1.2.1 Synchronous parallel SGD

Synchronous SGD (*SyncSGD*) is a straight-forward lock-step data-parallel version of SGD where threads or nodes access the shared state θ_t at an iteration t, then compute gradients based on individual randomly sampled data-batches, see Fig. (1.3). The threads synchronize by averaging the resulting gradients before taking a global step according to (1.2) [11]. In the original version, *SyncSGD* is statistically equivalent to sequential SGD with larger mini-batch size [15], as also shown in **Paper A**, and can hence be considered a method for accelerated gradient computation. From this perspective, the *SyncSGD* approach does not break the semantics of the sequential SGD algorithm, and the vast empirical results and theoretical convergence guarantees in the literature entail predictable performance of *SyncSGD*. From a scalability perspective, since each SGD iteration is only as fast as the slowest contributing thread, the presence of slower threads, i.e. *stragglers*, becomes a bottleneck. A comprehensive overview of methods along this approach is provided in [16].

Stale-synchronous parallel (SSP) relaxes the strict synchronous semantics of *SyncSGD*, allowing faster threads to asynchronously compute a bounded number of SGD steps based on a local version of the state before synchronizing [12]. 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 [17] 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

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

		Synchronous		***Hybrid***	***Asynchronous***		
Δ	**Asynchronous**	×	-	-	✓	✓	✓
θ	**Consistent**	✓	✓	✓	✓	×	✓
	Lock-free	×	×	×	×	✓	✓
		SyncSGD [11]	Stale-synchronous [12]	n-softsync [13]	*AsyncSGD* [14]	HOGWILD! [9]	*Leashed-SGD*[this thesis]

SSP provide weak progress guarantees, since in the presence of halting threads, the system as a whole will halt indefinitely in the synchronization step. This is partially addressed by *n-softsync*, which is a further relaxed variant of *SyncSGD* with partial synchronization, requiring only a fixed number n of threads to contribute a gradient at the synchronization point. 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 [13] [15], the recent work [7] implements similar semantics in a decentralized setting utilizing a `partial-allreduce` primitive which atomically applies the aggregated updates and redistributes the result.

1.2.2 Asynchronous parallel SGD

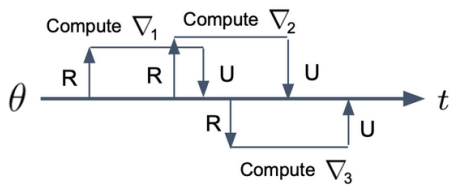

Figure 1.4: *AsyncSGD* parallelizes the SGD iterations, allowing asynchronous read (R) and update (U) operations on the shared state.

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

$$\theta_{t+1} \leftarrow \theta_t - \eta \widetilde{\nabla} f(v_t) \tag{1.4}$$

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

AsyncSGD surely enables increased computational efficiency with higher parallelism, up to a point where contention due to concurrent shared-memory access attempts becomes too severe. We denote the corresponding number of threads by m_C^*, and at this point the system stagnates and additional computing threads provide no additional speedup. In addition, the presence of staleness in *AsyncSGD* causes decay in statistical efficiency from the asynchrony-induced noise, which grows as more threads are introduced to the system. Over-parallelization may thereby not only be redundant, but in fact harm the statistical efficiency, with potentially dire consequences on the overall convergence rate. There is hence a trade-off between computational and statistical efficiency, which in practice requires careful tuning of the level of

parallelism (number of threads) m. The appropriate choice of m depends heavily on the properties of the optimization problem itself, as well as the choice of other hyper-parameters, e.g. the step size η and the batch size b.

In the following, we give a brief overview of the literature most relevant in the context of this thesis.

***AsyncSGD* and momentum.** The research direction of asynchronous iterative optimization is not new, and sparked due to the works by Bertsekas and Tsitsiklis [18] in 1989. More recently, Chaturapruek et al. [14] show that, under several analytical assumptions such as convexity (linear and logistic regression), the convergence of *AsyncSGD* is not significantly affected by asynchrony and that the noise introduced by staleness is asymptotically negligible compared to the noise from the stochastic gradients. In [19] Lian et al. show that these assumptions can be partially relaxed, and it is shown that convergence is possible for non-convex problems, however with a bounded number of threads, and assuming bounded staleness. Several works have followed, aiming at understanding the impact of asynchrony on the convergence. In [20] Mitliagkas et al. show that under certain stochastic staleness models, asynchronous parallelism has an effect on convergence similar to momentum. This work is extended in parts of **Paper A**, which introduces models which capture the dynamics of the system more accurately, leading to alternate conclusions, as well as means to improve the statistical efficiency by asynchrony-awareness. In [21] 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 [8] Alistarh et al. introduces the concept of bounded divergence between the parameter vector and the threads' view of it, proving convergence bounds for convex and non-convex problems.

***AsyncSGD* and lock-freedom.** Hogwild! [9], 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 [22] [23]. De Sa et. al [24] introduced a framework for analysis of Hogwild!-style algorithms for sparse problems. The analysis was extended in [25], 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. This is the main focus of **Paper B**, where we introduce a consistency-preserving lock-free implementation of *AsyncSGD* in practice for DL. In [6] 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 [26] the focus is the fundamental limitation of data parallelism in ML. They observe that the limitations are due to concurrent SGD parameter accesses, during ML training, usually diminishing or even negating the parallelization benefits provided by additional parallel compute resources. To alleviate this, they propose the use of static analysis for identification of data that do not cause dependencies, for parallelizing their access. They do this as part of a system that uses Julia, a script language that performs just-in-time compilation. Their approach is effective and works well for e.g. Matrix factorization SGD. For DNNs, as they explain, their work is not directly applicable, since in DNNs permitting "good" dependence violation is the common parallelization approach. Asynchronous SGD approaches for DNNs are scarce in the current literature. In the recent work [27], 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 [28] theoretical convergence analysis is presented for *SyncSGD* with *once-in-a-while* synchronization. They mention the analysis can guide in applying *SyncSGD* for DL, however the analysis requires strong convexity of the target function. [29] 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 [30] a topology for decentralized parallel SGD is proposed, using pair-wise averaging synchronization. In this thesis we leverage evaluation of our proposed methods for deep learning problems in particular, in order to make sure benchmarks are as useful as possible in practice.

Asynchrony-adaptive SGD. Delayed optimization in asynchronous first-order optimization algorithms was analyzed initially in [31], 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 [32] under assumptions of gradient sparsity and *read* and *write* operations having the same relative ordering. A similar approach was used in [13], however for synchronous SGD with the *softsync* protocol. In [13] 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 [33], 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 [33] is a uniform stochastic distribution, which implies a strict upper bound on the delays, making the system model partially asynchronous. **Paper A** extends this line of research, exploring further the idea of adapting updates based on staleness, and studies in particular analytical foundations to motivate how.

1.3 Problems and challenges

1.3.1 Scalability

Growing batch size. In *SyncSGD*, stragglers become a bottleneck, making every iteration only as fast as the slowest thread. This issue can however partially be reduced through relaxed semantics, such as SSP and the n-softsync protocol (See 1.2). Moreover, the convergence of *SyncSGD* under increasing parallelism is statistically equivalent to sequential SGD with a larger *mini-batch size* b [15], also shown in **Paper A**, which is a hyper-parameter that requires careful tuning depending on the problem. In particular, the convergence can be worsened if b is too large [34] [35]. As discussed in section 1.2.1, this indicates limited scalability, as over-parallization will impose large-batch properties, which in some cases worsens the convergence [15]. This motivates further exploration of asynchronous parallelism for scalability.

Staleness. *AsyncSGD* eliminates many scalability bottlenecks of *SyncSGD* due to reduced inter-thread coordination (stronger Δ-progress guarantees), however this also introduces other challenges related to asynchrony. As discussed in section 1.2.2, asynchronous access to and update of the shared state leads to staleness due to the fact that updates may occur by threads concurrently to the gradient computation. The updates that are applied are hence not necessarily, in fact rarely in practice, based on the latest shared state, as described by (1.4). For problems satisfying assumptions on convexity, smoothness and bounded gradients, staleness has little impact on the convergence of *AsyncSGD* [14]. However, for a wider class of problems staleness can have significant impact. In particular for problems not conforming to e.g. convexity assumptions, such as the recently relevant DL applications. Crucial steps toward understanding how convergence is affected in *AsyncSGD* due to staleness were taken by Mitliagkas et al. [20], explicitly quantifying the impact of concurrency, under a certain statistical staleness model. The results indicate that the influence of asynchrony has an effect similar to momentum in SGD, and a reduced step size. **Paper A** extends this analysis, proposing models better capturing the staleness dynamics, and showing that the momentum effect grows and the step size reduces monotonically as the parallelism is increased. This indicates a scalability limitation in convergence, which however can be partially alleviated by using a staleness-adaptive step size (**Paper A**).

Progress and consistency guarantees for Δ vs. θ. As previously mentioned, read and update operations on the shared state θ become focal in *AsyncSGD*, since they constitute the remaining synchronization steps in the otherwise asynchronous algorithm. There must be primitives in place to handle concurrent attempts to read and update by several threads, and these become bottlenecks for scalability at sufficiently high levels of parallelism. Traditionally, a separate thread or node acting as a parameter server is responsible for providing the latest parameter state to workers, as well as processing contributing gradients, sequentializing the updates [36]. To efficiently utilize multi-core systems, this was extended to shared-memory implementations [9, 19, 22]. The access to the shared state is then scheduled by the operating system, and regulated by some synchronization method, such as locking, to ensure consistency in case of concurrent read and update attempts. However, locks can

be relatively computationally expensive, in particular when the gradient computation step itself incurs little latency. In addition, the total time spent on waiting for locks grows as more threads are introduced to the system, potentially making it scalability bottleneck. By allowing completely uncoordinated component-wise atomic read and update operations, i.e. HOGWILD! [9], such contention is eliminated, allowing significant speedup for sparse optimization problems in particular. However, for other problems, as summarized in Table 1.3, HOGWILD! introduces inconsistency when read and update operations occur concurrently, with unpredictable impact on the convergence. There is currently a lack of methods providing a middle-ground solutions in the literature in the realm in between these two endpoints of the synchronization spectrum, i.e. the consistency-enforcing lock-based *AsyncSGD* and the lock-free inconsistency-prone HOGWILD!. This spectrum is explored further in **Paper B**, and *Leashed-SGD* is proposed as a middle-ground solution.

Memory consumption. An additional aspect of scalability to consider is memory consumption; standard *AsyncSGD* implementations in the literature require each thread to copy the entire shared state θ prior to its individual gradient computation. The result of the computation, i.e. the stochastic gradient, is of the same dimension d as θ, and is stored locally until applied to the shared state in an SGD iteration. The magnitude of d varies, however in DL applications it is often in the magnitude of hundreds of thousands, sometimes millions, which is why the memory consumption of the *AsyncSGD* implementation needs to be carefully considered. This aspect is discussed further in **Paper B**, and possible improvements are explored.

1.3.2 Convergence under asynchrony

Staleness. The staleness that arises in *AsyncSGD* due to parallelism significantly impacts the statistical efficiency of the convergence; it has been shown analytically that the number of SGD iterations to ϵ-convergence increases linearly in the maximum staleness [24, 25]. Hence, only if the gains in computational efficiency from parallelism are sufficiently great, will there be an overall improvement in wall-clock time until ϵ-convergence. In addition, inconsistent synchronization as in HOGWILD! potentially incurs further statistical penalty; the expected number of iterations required increases linearly in $\sqrt{d}$ [25]. Subsequently, there are challenges in understanding whether it is worth the computational overhead to ensure consistency for a given problem, and which synchronization primitives are appropriate to utilize.

Synchronization. As a consequence of Amdahl's law [37], when there is a synchronization overhead, the achievable speedup is bounded. In the context of *AsyncSGD*, this applies in particular for the computational efficiency, i.e. how many SGD updates can be applied in a given time unit. This implies that there is a *computational saturation point* m_C^* for which additional threads will not provide additional significant computational speedup. For this statement, as well as the ones to follow in this paragraph, empirical evidence is provided in **Paper B**. Moreover, due to the presence of staleness there is a degradation of statistical efficiency coupled to parallelism in *AsyncSGD* [21, 26]. Hence, as more threads are introduced to the system, more iterations are required until reaching ϵ-convergence. At some level of parallelism, which we refer to

as the *system saturation point* m_S^*, additional threads will no longer reduce the wall-clock time to ϵ-convergence, and might instead even increase it. It can be concluded that $m_S^* \leq m_C^*$ from a simple argument of contradiction, assuming that statistical efficiency degrades with higher parallelism. This assumption is in accordance with results in previous literature [24, 25], and explored further in **Paper A** and **Paper B**. There are substantial challenges in understanding the appropriate range of the number m of threads in order to (i) fully utilize the parallel computation ability of the system and (ii) avoid over-parallelization, potentially harming or completely obstructing convergence. Ideally an implementation of *AsyncSGD* feature resilience to tuning, providing reliable and fast convergence over a broad spectrum of parallelism, towards which **Paper B** takes significant steps.

1.3.3 Benchmarking and evaluation

Standardization. There are significant challenges in conducting empirical evaluations and comparisons which are useful and fair within the domain of parallel SGD, for several reasons: Firstly, there are several metrics of interest related to convergence of SGD, the measurements of which must be effectively aggregated as to show the overall performance. Traditionally, in ML the statistical efficiency is the metric most used, i.e. the number of SGD iterations until reaching sufficient performance, i.e. ϵ-convergence. However, when improvements in statistical efficiency is achieved by altering the underlying algorithm, this potentially alters the computational efficiency, i.e. the number of SGD iterations per time unit. In such cases, it is hence necessary that evaluations take this into consideration, and ideally provide measurements of the overall convergence rate, i.e. the wall-clock time until converging to a solution of sufficient quality. Secondly, the domain of shared-memory parallel SGD lacks established universal procedures for benchmarking, leaving the task of setting up an appropriate test environment to the individual authors. The domain contains a wide spectrum of questions, ranging from efficient communication protocols [38] in wide distributed DL networks to exploring the impact of progress guarantees and synchronization in shared data structures [9, 25]. 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 [39] takes important steps in providing such an environment, although it focuses primarily on higher-level distributed SGD. For instance, the framework provides a Python interface for development, which does not facilitate exploration of for instance efficient shared data structures for fine-grained synchronization and mechanisms for memory management.

Hyper-parameter dependencies. Another key issue in benchmarking parallel SGD for machine learning is the inherent dependency between parallelism and various hyper-parameters crucial for achieving convergence [15, 20], some of the most important being the step size η and the mini-batch size b. As mentioned above, it is known that higher parallelism in *SyncSGD* exhibits similar convergence properties as sequential SGD with a larger batch size. As more threads or nodes are introduced to the system, the scalability of *SyncSGD* can hence appear to be limited due to the statistical penalty from a too large

value of b. This can be avoided by choosing a sufficiently small initial b for each thread or node, which will then instead give the appearance of high scalability, but only until a certain level of parallelism [15]. It is hence of interest in such evaluations to provide empirical evidence from test scenarios that indicate the general ability of the proposed method to scale independently of hyper-parameter choices. Analogously, for *AsyncSGD*, there is delicate interplay between the step size η and the staleness distribution, stemming from the fact that stale updates correspond to gradients based on old views of the state, and are applied with a coarsity proportional to η [32, 40]. A smaller η implies less impact on the convergence per update, hence tends to tolerate updates with higher staleness, and subsequently higher levels of parallelism. This can give the appearance of good scalability, showing speedup for a larger number of threads. It is in this case also of interest to provide empirical results that indicate scalability independently of hyper-parameters, such as η, for instance by testing for several choices of η.

In summary, there are challenges in establishing standardized evaluation methodologies, making fair and useful comparisons between methods difficult. This is mainly due to the wide span of research questions in the application domain, and a collective strive towards standardized benchmarking platform for various methodological aspects is imperative. In addition, the dependence of the performance and scalability of parallel SGD algorithms on various hyper-parameters, such as step size η and batch size b, complicate empirical evaluations. Providing evaluations with exhaustive combinations of such hyper-parameters is not feasible in practice, however it is necessary that some evidence is provided that indicate that the hyper-parameters are not tuned with respect to the test results. In this thesis, we aim to address critical questions among the aforementioned challenges, as summarized in the following section.

1.4 Thesis contributions

1.4.1 Paper A: Convergence of staleness-adaptive SGD

The scalability limitations of traditional synchronous parallel SGD highlighted in section 1.3.1 motivates further exploration of asynchronous parallelization, i.e. *AsyncSGD* which has shown promising improvements in ability to scale for many applications. The degradation of statistical efficiency due to staleness is however a limiting factor, forcing the user to carefully tune the level of parallelism in order to maintain an actual overall speedup in convergence rate, as also highlighted in section 1.3.1. In order to address this issue, we first propose methods to statistically model the behaviour of staleness in *AsyncSGD*. The models, which are proposed based on reasoning of the dynamics of the algorithm and its dependency on scheduling, capture the staleness distribution in practice to a high degree of precision, and more accurately than models previously proposed in the literature.

Based on the proposed staleness models, we provide analytical results that quantify the side-effect of asynchrony on the statistical efficiency (Lemma 1, Chapter 2). Moreover, it enables derivation of a staleness-adaptive step size, referred to as *MindTheStep-AsyncSGD*, which provably reduces this side-effect (Theorem 4, Chapter 2), and in expectation can, depending on the

rate of adaptiveness, alter it into the more desired behaviour of momentum (Theorem 3, 5, Chapter 2). We prove also that the staleness-adaptive step size is efficiently computable (Theorem 2, Chapter 2), ensuring minimal additional synchronization overhead for maximal scalability capability, as described in section 1.3.2. We provide an empirical evaluation of the proposed staleness models and the adaptive step size for a relevant use case, namely DL for image classification. The empirical results show in particular

(i) significantly improved accuracy in modelling the staleness with our proposed models

(ii) reduced penalty from asynchrony-induced noise, leading to up to a $\times 1.5$ speedup in convergence compared to baseline (standard *AsyncSGD* with constant step size) under high parallelism.

1.4.2 Paper B: Framework for lock-freedom and consistency

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

Among the analytical results for *Leashed-SGD*, we have in particular guarantees on lock-freedom and atomicity (Lemma 2, Chapter 3), safety and exhaustiveness and bounds on the memory consumption (Lemma 3, Chapter 3). Moreover, we model the progression of the algorithm over time, finding in particular fixed points in the system useful for estimating potential contention (Theorem 7, Corollary 5, Chapter 3) and the effect of the built-in contention-regulating mechanism (Corollary 6, Chapter 3).

We conduct an extensive empirical study of *Leashed-SGD* for MLP and CNN training (see section 1.1) for image classification. The empirical study focuses on scalability, dependence on hyper-parameters, distribution of the staleness, and benchmarks the proposed framework compared to established baselines, namely lock-based *AsyncSGD* and HOGWILD!. We draw the following main conclusions 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 1.3.1. The baselines however require careful tuning of the number of threads in order to avoid tediously slow convergence and are more prone to completely failing or crashing executions.

(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 achieving convergence.

(iii) For MLP training we observe up to 27% reduced median running time for ϵ-convergence for *Leashed-SGD* compared to baselines, with similar memory footprint. For CNN training, we observe a $\times 4$ speedup for ϵ-convergence, with a memory footprint reduction with 17% on average.

For the empirical study, a modular and extensible C++ framework is developed with the purpose of facilitating development of shared-memory parallel SGD with varying synchronization mechanisms. Hence, we take steps towards addressing the challenges (highlighted in section 1.3.3) that the community faces regarding a general platform for further exploration of aspects of fine-grained synchronization in this domain.

1.5 Conclusions

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

While higher parallelism in *AsyncSGD* enables more iterations per second, its inherent staleness and asynchrony-induced noise leads to an deteriorating statistical efficiency, requiring a growing number of iterations to achieve sufficient convergence. Understanding and modelling the dynamics of the staleness enables explicitly quantifying its side-effect on the convergence, towards which important steps were taken in [20], however under simplifying assumptions. Under a more practical system model, this analysis was extended in **Paper A**, and used to show how adaptiveness to staleness reduces asynchrony-induced noise, and thereby improves convergence. In addition, it allows derivation of the proposed staleness-adaptive *MindTheStep-AsyncSGD* which provably reduces this side-effect. The analytical results are confirmed in practice in **Paper A**, showing increased statistical efficiency in ANN training for image classification.

Relaxed inter-thread synchronization, with weak consistency requirements as in HOGWILD! [9], enables a straight forward way for achieving lock-freedom in shared state operations. The significantly reduced computational overhead allows overall speedup for sparse problems, where inconsistency consequently has little impact [9] and asymptotic convergence bounds can be established. However, the inconsistency has implications on the statistical efficiency, as observed theoretically in [25] and confirmed in **Paper A** and **Paper B**. In **Paper B** an interface is introduced which provides abstractions of operations on the shared state θ, which are used in the proposed lock-free *Leashed-SGD*

implementation, which guarantees consistency. The lock-free nature of *Leashed-SGD* has a self-regulating effect which avoids congestion under high parallelism, which by reducing the overall staleness distribution enables fast and stable convergence in contexts where the baselines fail. In this context, the dynamic memory allocation featured in *Leashed-SGD* allows for significantly reduced memory footprint, which is critical in particular for DL applications where the problems dimension can be in the order of millions.

Natural steps for future work include extending the *Leashed-SGD* framework with ideas from *MindTheStep-AsyncSGD*, enabling staleness-adaptive updates, as well as adaptiveness to other aspects of asynchrony and consistency considerations. This is particularly interesting considering the ability of *Leashed-SGD* to naturally fundamentally alter the overall staleness distribution in the system.

Chapter 2

PAPER A

Adaptation of the article:

***MindTheStep-AsyncSGD*: Adaptive Asynchronous Parallel Stochastic Gradient Descent**

K. Bäckström, M. Papatriantafilou, P. Tsigas

Proceedings of the 7th IEEE International Conference on Big Data, 2019, pp. 16-25

.

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.

2.1 Introduction

The explosion of data volumes available for Machine Learning (ML) has posed tremendous scalability challenges for machine intelligence systems. Understanding the ability to parallelise, scale and guarantee convergence of basic ML methods under different synchronization and consistency scenarios have 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 $f : \mathbb{R}^d \to \mathbb{R}$ of a d-dimensional vector x using a first-order light-weight iterative optimization approach; i.e., given a randomly chosen starting point x_0, SGD repeatedly changes x 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 α_t defines how coarse the updates are:

$$x_{t+1} \leftarrow x_t - \alpha_t \nabla F(x_t) \tag{2.1}$$

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 [41].

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 x, 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 [12] [42] and efficient gradient communication [38] [43].

Motivation and summary of state-of-the-art

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

$$f_D(x) = \frac{1}{|D|} \sum_{d \in D} L(d; x) \tag{2.2}$$

where the parameter vector x, encodes previously gathered features from D. In this context, SGD typically selects mini-batches $B \subseteq D$ over which f_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 (2.2) is to utilize *data parallelism* [11], 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 x, 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 (*SyncPSGD*) due to its barrier-based nature. In its simple form, SyncPSGD has scalability issues due to the waiting time that is inherent in the aggregation when different workers compute with different speed. 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 x 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 x, which are *statistically inefficient*. However, gains in *computational efficiency* due to parallelism and asynchrony can compensate for this, reducing the overall 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 x 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 precense 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 2.3-2.6, we outline preliminaries and background. Following the results-sections, we provide an extensive discussion on related work, conclusions and future work.

2.2 Preliminaries

2.2.1 Stochastic Gradient Descent

We consider the optimization problem

$$\underset{x}{\text{minimize}} \quad f(x) \tag{2.3}$$

for a function $f : \mathbb{R}^d \to \mathbb{R}$. In this context, we focus on methods to address this minimization problem (2.3) using SGD, defined by (2.1) for some randomly chosen starting position x_0. We assume that the stochastic gradient ∇F is an unbiased estimator of ∇f, i.e. $\mathbf{E}[\nabla F(x) \mid x] = \nabla f(x)$ for all x. This assumption holds for several relevant applications, in particular for problems of the form (2.2), 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 2.5 we adopt some additional standard assumptions on smoothness and convexity which we will introduce in that section.

2.2.2 System Model and Asynchronous SGD

We consider a system with 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 m asynchronous workers aim at performing SGD updates according to (2.1). Since each worker W must get a state x_t prior to computing a gradient, there can be intermediate updates from other workers before gradient from W is applied. The number of such updates defines the *staleness* τ_t corresponding to the gradient $\nabla F(x_t)$.

Assuming that the read and update operations can be performed atomically (see details in Section 2.4), under the system model above, the SGD update (2.1) becomes

$$x_{t+1} \leftarrow x_t - \alpha_t \nabla F(v_t) \tag{2.4}$$

where $v_t = x_{t-\tau_t}$ is the thread's *view* of x.

We assume that the staleness values τ_t 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. $\mathbf{E}[\tau_t] = \bar{\tau}$ for all t. 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_t is by definition dependent on writing time of concurrent updates, which in turn are dependent on their respective staleness values. For the analysis in Section 2.5, 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.

2.2.3 Momentum

SGD is typically inefficient in *narrow valleys* when the target function in some neighbourhood increases more rapidly in one direction relative to another. Such neighbourhoods are frequent in target functions that arise in ML applications due to their inherent highly irregular and non-convex nature. Adding *momentum* (2.5) to SGD has been seen to significantly improve the convergence speed for such functions. SGD with momentum, defined in (2.5), takes all previous gradient samples into account with exponentially decaying magnitude in its parameter μ. As pointed out in [20], μ is often left out in parameter tuning, and in some instances even failed to be reported [44]. However, the optimal value of algorithmic parameters such as μ, just like α, depends the problem, underlying hardware, as well as the choice of other parameters. Tuning μ has been shown to significantly improve performance [5], especially under asynchrony [20].

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

$$x_{t+1} \leftarrow x_t + \mu(x_t - x_{t-1}) - \alpha_t \nabla F(x_t) \tag{2.5}$$

2.3 On the scalability of Sync-PSGD

Optimal convergence with *SyncSGD* requires, as observed empirically in [15], that the mini-batch size is reduced as the number of worker nodes increase. 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 [34] [35], 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 $f(x)$ of the form (2.2) 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 $F(x) = f_B(x)$ is an unbiased estimator of $f(x)$ since

$$\begin{aligned}\mathbf{E}[F(x)] = \mathbf{E}[f_B(x)] &= \frac{b}{|D|}\sum_i f_{B_i}(x) \\ &= \frac{b}{|D|}\sum_i \frac{1}{b}\sum_{d\in B_i} L(d;x) = \frac{1}{|D|}\sum_{d\in D} L(d;x) = f(x)\end{aligned}$$

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 $F(x)$ 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 $F(x)$. 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. In particular, it has been seen that the convergence can suffer if b is too large [34] [35].

In the following theorem we show that by increasing the number of worker nodes in SyncPSGD, 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 1 SyncSGD *with m workers, all using batch size b, is equivalent to a sequential execution of SGD with batch size $m \cdot b$, reffered to as effective batch size.*

The proof appears in Appendix A. The main idea is to compute the average of two workers, using batch size b, from which it is clear that the result is equivalent to an execution of sequential SGD with batch size $2b$. The result follows inductively.

Since the mini-batch size is clearly lower bounded, Theorem 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 $m = b^*$, each using mini-batch size $b = 1$.

2.4 The proposed framework

We outline *MindTheStep-AsyncSGD* for staleness-adaptive steps and analyze how to choose a suitable adaptive step size function under different staleness models. Proofs appear in Appendix A , while brief intuitive arguments are presented here instead.

2.4.1 The *MindTheStep-AsyncSGD* Framework

We consider a standard parameter-server type of algorithm [12] [36], with atomic read and write operations, ensuring that workers acquire consistent views of the state x. In a distributed system, the consistency can be realized through the communication protocol. In a multi-core system, where worker nodes are threads and x 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 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 1: *MindTheStep-AsyncSGD*

```
GLOBAL start point x_0, functions F(x) and α(τ)

Worker W;                                  Parameter server S;
(t, x) ← (0, x_0)                          (t', x) ← (0, x_0)
repeat                                     repeat
    compute g ← ∇F(x)                          receive (t, g) from a ready worker W
    send (t, g) to S                           τ ← t' − t
    receive (t, x) from S                      x ← x − α(τ)g
                                               t' ← t' + 1
                                               send (t', x) to W
```

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

2.4.2 Tuning the impact of asynchrony

As pointed out in [20], asynchrony and delays introduce *memory* in the behaviour of the algorithms. In particular, in [20, 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 1 *Let τ be distributed according to some PDF p such that $P[\tau = i] = p(i)$. Then, for an adaptive step size function $\alpha(\tau)$, we have*

$$\mathbf{E}[x_{t+1} - x_t] = \mathbf{E}[x_t - x_{t-1}] + \sum_{i=0}^{\infty} \big(p(i)\alpha(i) - p(i+1)\alpha(i+1)\big)\nabla f(x_{t-i-1}) - p(0)\alpha(0)\nabla f(x_t) \tag{2.6}$$

The proof of Lemma 1 follows the structure the one in [20], now taking into account the adaptive step size. The main takeaways from Lemma 1 are that, under asynchrony, (i) the gradient contribution diminishes as the number of workers increases[1]; (ii) there is a momentum-like term introduced with parameter $\mu = 1$ and (iii) the update depends on the series term:

$$\Sigma^{\nabla}_{p,\alpha} = \sum_{i=0}^{\infty} \big(p(i)\alpha(i) - p(i+1)\alpha(i+1)\big)\nabla f(x_{t-i-1}) \tag{2.7}$$

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 α. However, this would require step sizes proportional to $p(0)^{-1}$, which rapidly grows out of bounds as the number of workers increase. Since large α 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 $\Sigma^{\nabla}_{p,\alpha}$ is manifested in the convergence behaviour in the form of asynchrony-induced memory with a *momentum* effect; see Theorem 3 of [20], repeated here for self-containment:

Theorem 2 ([20]) *Let all τ_t be geometrically distributed with parameter p, i.e. $\mathbf{P}[\tau = k] = p(1-p)^k$. Then, for a constant α, the expected update (2.4) becomes*

$$\mathbf{E}[x_{t+1} - x_t] = (1-p)\mathbf{E}[x_t - x_{t-1}] - p\alpha\nabla f(x_t) \tag{2.8}$$

The statement of Theorem 2 is easily confirmed by substituting $p(i)$ in (2.7) with constant α with the geometric PDF, which yields $\Sigma^{\nabla}_{p,\alpha} = -p\mathbf{E}[x_t - x_{t-1}]$.

Eq. (2.8) resembles the definition of momentum, with expected implicit asynchrony-induced momentum of magnitude $\mu = 1 - p$. As the number of workers grow and p tends to 0, Theorem 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 theorem how *MindTheStep-AsyncSGD* with a particular step size function resolves this issue.

Theorem 3 *Let staleness $\tau \in Geom(p)$ and*

$$\alpha_t = C^{-\tau_t} p^{-1} \alpha \tag{2.9}$$

where α is a parameter to be chosen suitably. Then

$$\mathbf{E}[x_{t+1} - x_t] = \mu_{C,p}\mathbf{E}[x_t - x_{t-1}] - \alpha\nabla f(x_t)$$

[1]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 (2.12). For the geometric staleness model we confirm empirically in section 2.6 that this assumption holds in practice, recall that $p(0) = p$.

and the implicit asynchrony-induced momentum is

$$\mu_{C,p} = 2 - (1-p)/C \tag{2.10}$$

The result is confirmed by substituting $\alpha(\tau)$ in (2.7) with the step size (2.9). Note that the expected implicit momentum vanishes for $C = (1-p)/2$. More generally:

Corollary 1 *Any desired momentum μ^* is, in expectation, implicitly induced by asynchrony by using the staleness-adaptive step size in (2.9) with*

$$C = (1-p)/(2-\mu^*) \tag{2.11}$$

Applicability of geometric τ. Each gradient staleness is comprised by 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. 2.4).

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 [41] 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 (2.4) 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 2.6. 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 $\tau \in \text{CMP}(\lambda, \nu)$ if

$$P[\tau = i] = \frac{1}{Z(\lambda,\nu)} \frac{\lambda^i}{(i!)^\nu} \ , \ Z(\lambda,\nu) = \sum_{j=0}^{\infty} \frac{\lambda^i}{(j!)^\nu} \tag{2.12}$$

which reduces to the Poisson distribution in the special case $\nu = 1$, i.e if $\tau \in \text{CMP}(\lambda, 1)$ then $\tau \in \text{Poi}(\lambda)$. For the remainder of this section we aim to further investigate the behaviour 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 $\tau \in \text{CMP}(\lambda, \nu)$ then the mode of τ is $\lfloor \lambda^{1/\nu} \rfloor$, and we therefore hypothesize the following relation:

$$\lambda^{1/\nu} = m \tag{2.13}$$

For the special case $\nu = 1$, i.e. a Poisson τ model, (2.13) enables us to immediately choose an appropriate value for λ given the number of workers m. In general, (2.13) 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 α. In the following, we argue analytically about how to choose an adaptive step size function $\alpha(\tau)$ for reducing the negative impact of stale gradients. We will see how a certain τ-adaptive step size can bound the magnitude of $\Sigma^{\nabla}_{p,\alpha}$ (2.7), and even tune the implicit asynchrony-induced momentum to any desired value.

Theorem 4 *Assume $\tau \in CMP(\lambda, \nu)$, and let the adaptive step size function be defined as follows:*

$$\alpha(\tau) = C\lambda^{-\tau}(\tau!)^{\nu}\alpha \tag{2.14}$$

for any constant C. Then we have $\Sigma^{\nabla}_{p,\alpha} = 0$.

Theorem 4 shows how a simple and tunable τ-adaptive step size mitigates the $\Sigma^{\nabla}_{p,\alpha}$ quantity. The proof consists of confirming that each contribution of the sum $\Sigma^{\nabla}_{p,\alpha}$ (2.7) vanishes when applying the definition of the CMP distribution (2.12) and the adaptive step size (2.14).

However, from Lemma 1, we see that even though $\Sigma^{\nabla}_{p,\alpha}$ is mitigated by the adaptive step size (2.14), the SGD steps still have a fixed implicit momentum term of magnitude $\mu = 1$. We show in Theorem 5 how the implicit momentum can be tuned to any desired value through a particular choice of $\alpha(\tau)$.

Theorem 5 *Assume $\tau \in CMP(\lambda, \nu)$. Then, we have that $\Sigma^{\nabla}_{p,\alpha}$ in expectation takes the form of asynchrony-induced momentum of magnitude exactly K, i.e.*

$$\Sigma^{\nabla}_{p,\alpha} = K\mathbf{E}[x_t - x_{t-1}]$$

when using the adaptive step size function:

$$\alpha(\tau) = c(\tau)\lambda^{-\tau}(\tau!)^{\nu}\alpha \tag{2.15}$$

where

$$c(\tau) = 1 - \frac{K}{\alpha e^{\lambda}} \sum_{j=0}^{\tau-1} \frac{\lambda^j}{(j!)^{\nu}} \tag{2.16}$$

Theorem 5 shows how the series term $\Sigma^{\nabla}_{p,\alpha}$ can take the form of momentum of desired magnitude by using a particular τ-adaptive step size. The main idea of the proof is the observation that the contributions of $\Sigma^{\nabla}_{p,\alpha}$ are simplified by the particular choice of the adaptive factor $c(\tau)$ (2.15), and the result follows from the definition of expectation. 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 $\alpha(\tau)$ under the Poisson τ-model.

Corollary 2 *Assuming $\tau \in Pois(\lambda)$, the series term $\Sigma^{\nabla}_{p,\alpha}$ takes the form of implicit momentum of magnitude K when using the adaptive step size function:*

$$\alpha(\tau) = \left(1 - \frac{K}{\alpha}\frac{\Gamma(\tau,\lambda)}{\Gamma(\tau)}\right)\lambda^{-\tau}\tau!\alpha \tag{2.17}$$

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

Corollary 2 shows how the series $\Sigma^{\nabla}_{p,\alpha}$ is in expectation replaced by momentum of any desired magnitude. Assuming Poisson τ, the $\mathcal{O}(\tau)$ sum in (2.16) is replaced in (2.17) by the Gamma and Upper Incomplete Gamma function, for which there exist efficient ($\mathcal{O}(1)$) and accurate numerical approximation methods [45].

2.5 Convex convergence analysis

In this section we analyze the convergence time of *MindTheStep-AsyncSGD*-type algorithms for convex and smooth optimization problems. Proofs appear in the Appendix A , while brief intuitive arguments are presented instead.

Consider the optimization problem (2.3) where an acceptable solution x satisfies ϵ-convergence, defined as

$$\|x - x^*\|^2 \leq \epsilon \tag{2.18}$$

We assume that the problem is addressed using *MindTheStep-AsyncSGD* under the system model described in Section 2.2. Note that we consider a staleness-adaptive step size, hence $\alpha_t = \alpha(\tau_t)$ is stochastic.

For the analysis in this section, we consider strong convexity and smoothness, specified in *Assumption 1*. These analytical requirements are common in convergence analysis for convex problems [24] [33] [25] [46].

Assumption 1 *We assume that the objective function f is, in expectation with respect to the stochastic gradients, strongly convex with parameter c with L-Lipschitz continuous gradients and that the second momentum of the stochastic gradient is upper bounded.*

$$\mathbf{E}\left[(x-y)^T\big(\nabla f(x) - \nabla f(y)\big) \mid x, y\right] \geq c\|x-y\|^2 \tag{2.19}$$

$$\mathbf{E}\left[\|\nabla F(x) - \nabla F(y)\|\right] \mid x, y] \leq L\|x-y\| \tag{2.20}$$

$$\mathbf{E}\left[\|\nabla F(x)\|^2 \mid x\right] \leq M^2 \tag{2.21}$$

The assumption (2.19) is standard in convex optimization and ensures that gradient-based methods will converge to a global optimum. Lipschitz continuity (2.20) is a type of strong continuity which bounds the rate with which the gradients can vary. Due to that $\mathbf{E}[\nabla F(x^*)] = 0$, (2.21) can be interpreted as bounding the variance of the gradient norm around the optimum x^*.

In addition to our system model in Section 2.2, we make the following assumption on the staleness process:

Assumption 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*

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

Assumption 2 is justifiable considering that the staleness (i.e. scheduler's decisions) at time i should not be considered to be influenced by staleness values τ_t of gradients yet to be computed.

Under Assumptions 1 and 2 above, we give a general bound on the number of iterations sufficient for expected ϵ-convergence in the following theorem:

Theorem 6 *Consider the unconstrained convex optimization problem of (2.3). Under Assumptions 1 and 2, for any $\epsilon > 0$, there is a sufficiently large number T of asynchronous SGD updates of the form (2.4) such that*

$$T \leq \Bigg(2\big(c - LM\epsilon^{-1/2}\mathbf{E}\left[\tau\alpha\right]\big)\mathbf{E}\left[\alpha\right] - \epsilon^{-1}M^2\mathbf{E}\left[\alpha^2\right] \Bigg)^{-1} \ln\left(\|x_0 - x^*\|^2\epsilon^{-1}\right) \tag{2.22}$$

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

The main idea in the proof of Theorem 6 is to bound $||x_{t+1} - x^*||/||x_t - x^*||$, which quantifies the improvement of each SGD step. The statement then follows from a recursive argument.

Corollary 3 *Under the same conditions as in Theorem 6, there exists a choice of a step size α 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 α be*

$$\alpha = \theta\frac{c\epsilon M^{-1}}{M + 2L\sqrt{\epsilon}\bar{\tau}} \tag{2.23}$$

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

$$T \leq \frac{M + 2L\sqrt{\epsilon}\bar{\tau}}{\theta(2-\theta)c^2M^{-1}\epsilon}\ln(\epsilon^{-1}\|x_0 - x^*\|^2) \tag{2.24}$$

The results in Theorem 6 and Corollary 3 are related to the results presented in [24] and [25]. The main differences are that in our analysis we tighten the bound with a factor $(2-\theta)^{-1}$, 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 $\alpha(\tau)$.

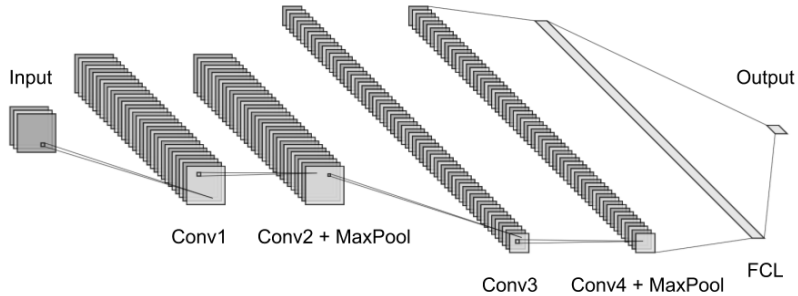

Figure 2.1: CNN architecture; Four convolutional layers with 3×3 kernels, with intermediate MaxPool layers. First two convolutions have 32 filters, the last two 64. The architecture has two fully connected layers, one with 256 neurons, and the output layer with 10 neurons.

Corollary 4 *Under the same conditions as Theorem 6, let $\alpha_t = \alpha(\tau_t)$ be a non-increasing function of τ_t. Then we have the following bound on the expected number of iterations until convergence:*

$$\begin{aligned} T \leq & \left(2c\mathbf{E}\left[\alpha\right] - \epsilon^{-1}M\left(M + 2L\sqrt{\epsilon}\bar{\tau}\right)\mathbf{E}\left[\alpha^2\right]\right)^{-1} \\ & \cdot \ln(\epsilon^{-1}\|x_0 - x^*\|^2) \end{aligned} \tag{2.25}$$

Corollary 4 describes a general convergence bound for any step size function $\alpha(\tau)$ which decays in τ. We see that such step size functions also achieve the asymptotic $\mathcal{O}(\bar{\tau}^{-1})$ bound, similar to the one for a constant α (2.24).

2.6 Experimental study

In this section we aim to evaluate the results derived in section 2.4 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. 2.1) on the common image classification benchmark dataset CIFAR10 [47]. 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 standard Python multiprocessor library as well as TensorFlow [44] for gradient computation.

CMP/Poisson τ. We evaluate the τ models (Poisson, CMP) proposed in section 2.4 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 [20] and the bounded uniform τ model [33].

The distribution parameters in Table 2.1 are found through an exhaustive search where we aim to minimize the Bhattacharyya distance to the τ

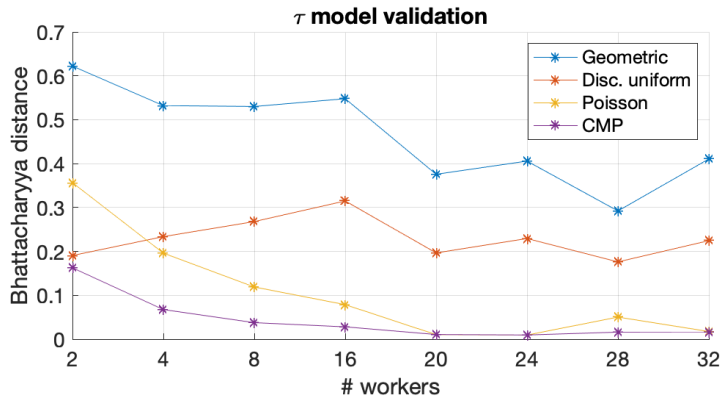

Figure 2.2: Bhattacharyya distance of different τ models compared to the observed distribution. The graph shows that the CMP τ model is the most accurate in all tests, with the Poisson τ model as a close second. The uniform and geometric τ models are persistently less accurate, and show poor scalability in comparison.

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 2.1: Optimal distribution parameters for different number of workers

distribution observed in practice. Note that: (i) For the Poisson τ model, as hypothesized in Section 2.4, the distribution parameter λ indeed corresponds well to the number of worker nodes. From Fig. 2.2 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 1, we confirm in Table 2.1 that $\mathbf{P}[\tau = 0]$, i.e. p, decays as m increases. (iii) We see in Fig. 2.2 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 (2.13) the other parameter λ is calculated. The result in Fig. 2.2 therefore validates the assumption (2.13).

Convergence with τ-adaptive α. 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 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 [12] [42] and efficient gradient communication and quantization [38] [43].

We compare standard *AsyncSGD* with constant step size $\alpha_c = 0.01$, $b = 128$ to *MindTheStep-AsyncSGD* with an adaptive step size function according to

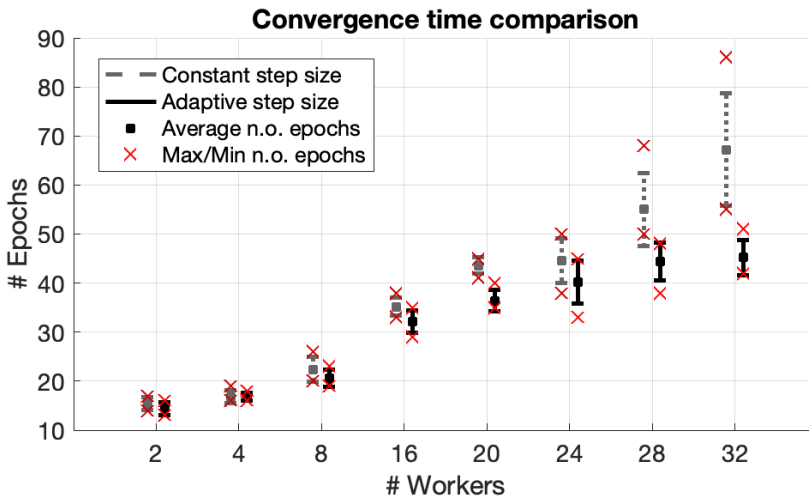

Figure 2.3: *AsyncSGD* vs. *MindTheStep-AsyncSGD* comparison. The plot shows the n.o. epochs required until sufficient performance (cross-entropy loss ≤ 0.05). The statistics are computed based on 5 runs, and the bar height corresponds to the standard deviation.

(2.17) with $\alpha = \alpha_c$, $K = 1$, and $\lambda = m$. In addition, we bound the step size $\alpha(\tau) \leq 5 \cdot \alpha_c$ 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 α_c, speedup can always be achieved by using an adaptive step size strategy $\alpha(\tau)$ which overall increases the average step size. To ensure a fair comparison, the adaptive step size function $\alpha(\tau)$ is normalized so that

$$\mathbf{E}_\tau[\alpha(\tau)] = \alpha_c \tag{2.26}$$

where the expectation is taken over the real τ distribution observed in the system. Enforcing (2.26) ensures that any potential speedup is achieved due to how the step size function $\alpha(\tau)$ adaptively changes the impact of gradients depending on their staleness, and not because of the overall magnitude of the step size.

Fig. 2.3 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 $\times 1.7$.

2.7 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 [48], however not in the context of SGD. Apart

from this, there are several variation of SGD in the sequential case introducing adaptivness to aspects of the problem topology, such as Adagrad, Adadelta, RMSprop, Adam, AdaMax, and Nadam (cf. [49] and references therein).

In [20] 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 [50] 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 [5] that the magnitude of momentum can have significant impact on convergence, the result by Mitliagkas et al. would imply a harsh scalability limitation of AsyncPSGD. In this paper, we show that under the same τ model as in [20], *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 different 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 $\alpha(\tau)$ 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 [11] and [51], synchronous distributed SGD with data-parallelism was observed and proven to accelerate convergence. This was implemented on a larger scale by Dekel et al. [52] where the convergence rates were improved under stronger analytical assumptions. In [12] 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 [15] 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 2.3.

The study of numerical methods under parallelism is not new, and sparked due to the works by Bertsekas and Tsitsiklis [18] in 1989. Recent works [14] [19] show under various analytical assumptions that the convergence of Async-PSGD 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 [14] for convex problems (linear and logistic regression) for a small number of cores. In [19] 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 Async-PSGD in shared-memory, i.e. HOGWILD!, was proposed by Niu et al. [9] and was shown to achieve near-optimal convergence rates assuming sparse gradients. Properties of Async-PSGD with sparse updates have since been rigourously studied in recent literature due to the performance benefits of lock-freedom [22] [24]. The gradient sparsity assumption was relaxed in the recent work [25] 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 [31], 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 [32] under assumptions of gradient sparsity and *read* and *write* operations having the same relative ordering. A similar approach was used in [40], however for synchronous SGD with the *softsync* protocol. In [13] 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 [33] 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 [33] 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.

2.8 Conclusions and Future Work

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 to models proposed in other works.

We derive and analyse 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 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 the staleness, i.e. effect of schedulers and synchronization methods, for understanding the impact of asynchrony and for choosing appropriate adaptive strategies.

Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP), Knut and Alice Wallenberg Foundation, the SSF proj. “FiC” nr. GMT14-0032 and the Chalmers Energy AoA framework proj. INDEED.

Appendix A

Appendix - Paper A

Proof of Theorem 1 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

$$\begin{aligned} x_{t+1} &= \frac{\big(x_t - \alpha \nabla f_{B_1}(x_t)\big) + \big(x_t - \alpha \nabla f_{B_2}(x_t)\big)}{2} \\ &= x_t - \frac{\alpha}{2}\left(\nabla f_{B_1}(x_t) + \nabla f_{B_2}(x_t)\right) \end{aligned}$$

For mini-batch GD, i.e. a target function of the form (2.2), and with mini-batch size b, the above formula becomes:

$$x_{t+1} = w - \frac{\alpha}{2}\left(\nabla \frac{1}{b} \sum_{d \in B_1} L(d, x_t) + \nabla \frac{1}{b} \sum_{d \in B_2} L(d, x_t)\right)$$

From linearity of the gradient, we have

$$\begin{aligned} x_{t+1} &= w - \alpha \nabla \frac{1}{2b} \sum_{d \in B_1 \cup B_2} L(d, x_t) \\ &= w - \alpha \nabla f_{B_1 \cup B_2}(x_t) \end{aligned}$$

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

Proof of Theorem 3 We have from (2.4)

$$\begin{aligned} x_{t+1} - x_t &= -\alpha_t \nabla F(v_t) \\ &= x_t - x_{t-1} - (x_t - x_{t-1}) - \alpha_t \nabla F(v_t) \\ &= x_t - x_{t-1} + \alpha_t \nabla F(v_{t-1}) - \alpha_t \nabla F(v_t) \end{aligned}$$

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

$$\begin{aligned} \mathbf{E}[x_{t+1} - x_t \mid \tau_t, \tau_{t-1}] = \mathbf{E}[x_t - x_{t-1} \mid \tau_t, \tau_{t-1}] \\ + \alpha_t \nabla f(v_{t-1}) - \alpha_t \nabla f(v_t) \end{aligned}$$

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

$$
\begin{aligned}
\mathbf{E}[x_{t+1} - x_t] &= \mathbf{E}[x_t - x_{t-1}] \\
&\quad + \mathbf{E}[\alpha_t \nabla f(v_{t-1})] - \mathbf{E}[\alpha_t \nabla f(v_t)] \\
&= \mathbf{E}[x_t - x_{t-1}] + \sum_{i=0}^{\infty} P[\tau = i] \frac{\alpha \nabla f(x_{t-i-1})}{C^i p} \\
&\quad - \sum_{i=0}^{\infty} P[\tau = i] \frac{\alpha \nabla f(x_{t-i})}{C^i p} \\
&= \mathbf{E}[x_t - x_{t-1}] + p \sum_{i=0}^{\infty} (1-p)^i \frac{\alpha \nabla f(x_{t-i-1})}{C^i p} \\
&\quad - p \sum_{i=0}^{\infty} (1-p)^i \frac{\alpha \nabla f(x_{t-i})}{C^i p} \\
&= \mathbf{E}[x_t - x_{t-1}] - \alpha \nabla f(x_t) + \sum_{i=0}^{\infty} (1-p)^i \frac{\alpha \nabla f(x_{t-i-1})}{C^i} \\
&\quad - \sum_{i=1}^{\infty} (1-p)^i \frac{\alpha \nabla f(x_{t-i})}{C^i}
\end{aligned}
$$

$$
\begin{aligned}
&= \mathbf{E}[x_t - x_{t-1}] - \alpha \nabla f(x_t) \\
&\quad + \sum_{i=0}^{\infty} \left(\frac{(1-p)^i}{C^i} - \frac{(1-p)^{i+1}}{C^{i+1}} \right) \alpha \nabla f(x_{t-i-1}) \\
&= \mathbf{E}[x_t - x_{t-1}] - \alpha \nabla f(x_t) \\
&\quad + \sum_{i=0}^{\infty} \frac{(1-p)^i}{C^i} \left(1 - \frac{1-p}{C} \right) \alpha \nabla f(x_{t-i-1}) \\
&= \mathbf{E}[x_t - x_{t-1}] - \alpha \nabla f(x_t) \\
&\quad + \left(1 - \frac{1-p}{C} \right) \sum_{i=0}^{\infty} \frac{p(1-p)^i}{C^i p^{i+1}} \alpha \nabla f(x_{t-i-1}) \\
&= \mathbf{E}[x_t - x_{t-1}] - \alpha \nabla f(x_t) \\
&\quad + \left(1 - \frac{1-p}{C} \right) \mathbf{E}[\alpha_t \nabla f(v_{t-1})] \\
&= \mathbf{E}[x_t - x_{t-1}] - \alpha \nabla f(x_t) + \left(1 - \frac{1-p}{C} \right) \mathbf{E}[x_t - x_{t-1}] \\
&= \left(2 - \frac{1-p}{C} \right) \mathbf{E}[x_t - x_{t-1}] - \alpha \nabla f(x_t)
\end{aligned}
$$

■

Proof of Theorem 4 We have

$$
\Sigma_{p,\alpha}^{\nabla} = \sum_{i=0}^{\infty} \big(p(i)\alpha(i) - p(i+1)\alpha(i+1) \big) \nabla f(x_{t-i-1})
$$

Substituting $p(i)$ for the CMP PDF (2.12) gives

$$\Sigma^{\nabla}_{p,\alpha} = \frac{1}{Z(\lambda,\nu)} \sum_{i=0}^{\infty} \frac{\lambda^i}{(i!)^\nu} \left(\alpha(i) - \lambda \frac{\alpha(i+1)}{(i+1)^\nu} \right) \nabla f(x_{t-i-1}) \tag{A.1}$$

Now, applying the adaptive step size (2.14) gives

$$\begin{aligned}
\Sigma^{\nabla}_{p,\alpha} =& \frac{C}{Z(\lambda,\nu)} \sum_{i=0}^{\infty} \frac{\lambda^i}{(i!)^\nu} \alpha \Bigg(\lambda^{-i} (i!)^\nu - \\
& \frac{\lambda}{(i+1)^\nu} \lambda^{-(i+1)} ((i+1)!)^\nu \Bigg) \nabla f(x_{t-i-1}) \\
=& \frac{C}{Z(\lambda,\nu)} \sum_{i=0}^{\infty} \frac{\lambda^i}{(i!)^\nu} \alpha \left(\frac{(i!)^\nu}{\lambda^i} - \frac{(i!)^\nu}{\lambda^i} \right) \nabla f(x_{t-i-1}) = 0
\end{aligned}$$

■

Proof of Theorem 5 Let $\Psi(i) = \alpha(i) - \lambda \frac{\alpha(i+1)}{(i+1)^\nu}$, and hence

$$\Sigma^{\nabla}_{p,\alpha} = \frac{1}{Z(\lambda,\nu)} \sum_{i=0}^{\infty} \frac{\lambda^i}{(i!)^\nu} \Psi(i) \nabla f(x_{t-i-1})$$

Applying the adaptive step size (2.15) gives

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

Now,

$$\begin{aligned}
\Psi(i) = K \Leftrightarrow c(i) - c(i+1) &= \frac{K}{\alpha e^\lambda} \frac{\lambda^i}{i!^\nu} \\
\Leftrightarrow c(i) &= c(i-1) - \frac{K}{\alpha e^\lambda} \frac{\lambda^{i-1}}{(i-1)!^\nu} \\
= c(0) - \frac{K}{\alpha e^\lambda} \sum_{j=1}^{i} \frac{\lambda^{i-j}}{(i-j)!^\nu} &= c(0) - \frac{K}{\alpha e^\lambda} \sum_{j=1}^{i} \frac{\lambda^j}{(j)!^\nu}
\end{aligned}$$

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

$$\begin{aligned}
\Sigma^{\nabla}_{p,\alpha} &= K \sum_{i=0}^{\infty} \frac{1}{Z(\lambda,\nu)} \frac{\lambda^i}{(i!)^\nu} \nabla f(x_{t-i-1}) \\
&= K \mathbf{E} \left[\nabla f(v_{t-1}) \right] = K \mathbf{E} \left[x_t - x_{t-1} \right]
\end{aligned}$$

■

Proof of Corollary 2 Under the Poisson τ model, which is CMP with $\nu = 1$, (2.16) rewrites to

$$\begin{aligned}
c(i) &= 1 - \frac{K}{\alpha e^\lambda} \sum_{j=0}^{\tau-1} \frac{\lambda^j}{(j!)} = 1 - \frac{K}{\alpha} \frac{\Gamma(i,\lambda)}{(i-1)!} \\
&= 1 - \frac{K}{\alpha} \frac{\Gamma(i,\lambda)}{\Gamma(i)}
\end{aligned}$$

■

Proof of Theorem 6

$$
\begin{aligned}
&\|x_{t+1} - x^*\|^2 = \|x_t - \alpha_t \nabla F(v_t) - x^*\|^2 \\
&= \|x_t - x^*\|^2 + \alpha_t^2 \|\nabla F(v_t)\|^2 - 2\alpha_t (x_t - x^*)^T \nabla F(v_t) \\
&= \|x_t - x^*\|^2 + \alpha_t^2 \|\nabla F(v_t)\|^2 - 2\alpha_t (x_t - x^*)^T \nabla F(x_t) \\
&\quad + 2\alpha_t (x_t - x^*)^T \big(\nabla F(x_t) - \nabla F(v_t)\big)
\end{aligned}
$$

Under expectation, conditioned on the natural filtration $\mathbb{F}_t^X = \big((\tau_i)_{i=0}^t, \big(\nabla F(v_i)\big)_{i=0}^t\big)$ of the *past* of the process, we have

$$
\begin{aligned}
&\mathbf{E}\big[\|x_{t+1} - x^*\|^2 \mid \tau_t, \mathbb{F}_{t-1}^X\big] = \|x_t - x^*\|^2 \\
&\quad - 2\alpha_t \mathbf{E}\big[(x_t - x^*)^T \big(\nabla F(x_t) - \nabla F(x^*)\big) \big| \mathbb{F}_{t-1}^X\big] \\
&\quad + 2\alpha_t \mathbf{E}\big[(x_t - x^*)^T \big(\nabla F(x_t) - \nabla F(v_t)\big) \big| \mathbb{F}_{t-1}^X\big]
\end{aligned}
$$

Applying the assumptions (2.19)-(2.21) gives

$$
\begin{aligned}
&\mathbf{E}\big[\|x_{t+1} - x^*\|^2 \big| \tau_t, \mathbb{F}_{t-1}^X\big] \leq \|x_t - x^*\|^2 + M^2 \alpha_t^2 \\
&\quad - 2\alpha_t c \|x_t - x^*\|^2 + 2\alpha_t L \|x_t - x^*\| \|x_t - v_t\| \\
&= (1 - 2c\alpha_t)\|x_t - x^*\|^2 + M^2 \alpha_t^2 \\
&\quad + 2\alpha_t L \|x_t - x^*\| \|x_t - v_t\| \\
&= (1 - 2c\alpha_t)\|x_t - x^*\|^2 + M^2 \alpha_t^2 \\
&\quad + 2\alpha_t L \|x_t - x^*\| \sum_{i=1}^{\tau_t} x_{t-i+1} - x_{t-i}\| \\
&\leq (1 - 2c\alpha_t)\|x_t - x^*\|^2 + M^2 \alpha_t^2 \\
&\quad + 2\alpha_t L \sum_{i=1}^{\tau_t} \|x_t - x^*\| \alpha_{t-i} \|\nabla F(v_{t-i})\|
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathbf{E}\left[\|x_{t+1} - x^*\|^2 \big| \tau_t, (\nabla)_0^t\right] \\
&\quad \leq (1 - 2c\alpha_t)\mathbf{E}\big[\|x_t - x^*\|^2 \big| \tau_t, (\nabla)_0^t\big] + M^2 \alpha_t^2 \\
&\quad + 2L\alpha_t \sum_{i=1}^{\tau_t} \mathbf{E}\big[\alpha_{t-i} \|x_t - x^*\| \big| \tau_t, (\nabla)_0^t\big] \|\nabla F(v_{t-i})\|
\end{aligned}
$$

From the non-anticipativity of the delay process we have

$$
\begin{aligned}
&\mathbf{E}\left[\alpha_{t-i} \|x_t - x^*\| \mid \tau_t, (\nabla)_0^t\right] \\
&= \mathbf{E}\left[\mathbf{E}\left[\alpha_{t-i} \|x_t - x^*\| \mid x_t\right] \mid \tau_t, (\nabla)_0^t\right] \\
&= \mathbf{E}\left[\|x_t - x^*\| \mathbf{E}\left[\alpha_{t-i} \mid x_t\right] \mid \tau_t, (\nabla)_0^t\right] \\
&= \mathbf{E}\left[\alpha_{t-i}\right] \mathbf{E}\left[\|x_t - x^*\| \mid (\nabla)_0^t\right]
\end{aligned}
$$

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

$$\begin{aligned}\mathbf{E}\left[\|x_{t+1} - x^*\|^2 \mid \tau_t\right] &\leq (1 - 2c\alpha_t)\mathbf{E}\left[\|x_t - x^*\|^2\right] + M^2\alpha_t^2 \\ &+ 2L\tau_t\alpha_t\mathbf{E}\left[\alpha_t\right]\sqrt{\mathbf{E}\left[\|x_t - x^*\|^2\right]}\sqrt{\mathbf{E}\left[\|\nabla F(v_t)\|^2\right]}\end{aligned}$$

and the full expectation satisfies

$$\begin{aligned}\mathbf{E}\left[\|x_{t+1} - x^*\|^2\right] &\leq (1 - 2c\mathbf{E}\left[\alpha_t\right])\mathbf{E}\left[\|x_t - x^*\|^2\right] \\ &+ M^2\mathbf{E}\left[\alpha_t^2\right] + 2LM\mathbf{E}\left[\tau_t\alpha_t\right]\mathbf{E}\left[\alpha_t\right]\sqrt{\mathbf{E}\left[\|x_t - x^*\|^2\right]}\end{aligned}$$

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

$$\begin{aligned}\mathbf{E}\left[\|x_{t+1} - x^*\|^2\right] &\leq \mathbf{E}\left[\|x_t - x^*\|^2\right](1 - 2c\mathbf{E}\left[\alpha_t\right] \\ &+ \epsilon^{-1}M^2\mathbf{E}\left[\alpha_t^2\right] + 2LM\epsilon^{1/2}\mathbf{E}\left[\tau_t\alpha_t\right]\mathbf{E}\left[\alpha_t\right]) \\ &=: \mathbf{E}\left[\|x_t - x^*\|^2\right](1 - \delta) \\ &\Rightarrow \mathbf{E}\left[\|x_T - x^*\|^2\right] \leq \mathbf{E}\left[\|x_0 - x^*\|^2\right](1 - \delta)^T \\ &\Rightarrow T \leq -\ln(1 - \delta)^{-1}\ln\frac{\mathbf{E}\left[\|x_0 - x^*\|^2\right]}{\mathbf{E}\left[\|x_T - x^*\|^2\right]} \\ &< \delta^{-1}\ln\left(\mathbf{E}\left[\|x_0 - x^*\|^2\right]\epsilon^{-1}\right)\end{aligned}$$

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

Proof of Corollary 3 Let $\rho = \frac{c\epsilon M^{-1}}{M + 2L\sqrt{\epsilon}\bar{\tau}}$. From Theorem 6 we have the improvement factor

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

so $\delta > 0$ when $0 < \alpha < 2\rho$, and the improvement is maximized for $\theta = 1$. Now, using the choice (2.23) of step size, we have

$$\begin{aligned}\delta &= c\rho^{-1}\theta\rho(2\rho - \theta\rho) \\ &= \theta(2 - \theta)c\rho\end{aligned}$$

Substituting for ρ, the convergence bound of Theorem 6 rewrites to (2.24) ■

Proof of Corollary 4 Since α_t is a non-increasing function in τ_t we have

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

Using this property, (2.22) rewrites to (2.25) ■

Chapter 3

PAPER B

Adaptation of the article:

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

---

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

Proceedings of the 35th IEEE International Parallel & Distributed Processing Symposium, 2021 (to appear).

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 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 with 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 DL applications: Multi-Layer 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× improvements) compared to the standard lock-based *AsyncSGD* algorithm and HOGWILD!, while reducing the overall memory footprint.

3.1 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 for 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 (3.1) 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:

$$\theta_{t+1} = \theta_t - \eta \widetilde{\nabla} f(\theta_t) \tag{3.1}$$

where θ_t contains the *learned* parameters of the model at iteration t, typically encoding features of a given data-set. 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. (3.1) [11]. In its original form, *SyncSGD* is statistically equivalent to sequential SGD with larger *data-batch* [15] [53]. This method is well-understood and widely used, e.g. in *federated learning* [54]. 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 [34]. Semi-synchronous variants have shown improvements [7, 55], relaxing lock-step semantics and requiring only a subset of threads to synchronize, hence reducing waiting. In the recent [7] 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 (3.1) on a shared vector θ with less coordination, and has shown superior speedup compared to *SyncSGD* in several applications [9, 56]. 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 [31] despite the presence of noise due to stale updates. A relaxed variant, Hogwild! [9], 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* [9,14,31,57] 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 [16, 58] 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 address 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 [24] that the number of iterations until convergence increases linearly in the magnitude of the maximum staleness and (ii) in [25] 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 impacts 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 [59]. 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 [26] in practice as outlined in Fig. 3.1, in particular for applications as DL. Understanding better the tradeoff between computational and statistical efficiency is a core issue [6]. It is known that consistency helps in *AsyncSGD* [25]. 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.

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 on-demand 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 paper is structured as follows: In section 3.2 we outline preliminaries and key notions for describing *Leashed-SGD*, as well as its contention and staleness dynamics in sections 3.3 and 3.4. The comprehensive empirical study is presented in 3.5, followed by further discussion of related work in section 3.6, after which we conclude in section 3.7.

3.2 Preliminaries

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

3.2.1 SGD and DL

Artificial Neural Networks (ANNs) are computational structures of simple units known as *neurons*, inspired by the biological brain. Neurons are arranged into

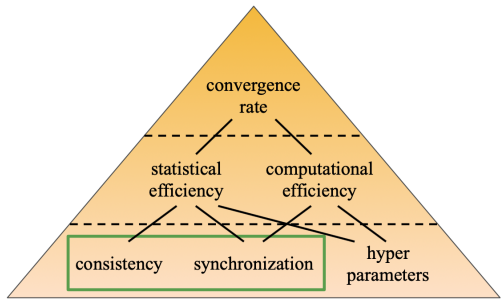

Figure 3.1: Convergence rate is the product of computational and statistical efficiency, sensitive to hyper-parameters tuning. We show the significant impact of lock-free synchronization on these factors and on reducing the dependency on tuning, enabling improved convergence.

layers, each performing a non-linear transformation of the output from the previous layer, parameterized by a set of learnable weights. The input layer is initialized as the input to be analyzed, e.g. an image to be classified. The output layer gives the final output, e.g. the class of an image. Different types of layer arrangements give rise to a diverse class of ANN architectures, with different applications. Among the most prominently used are Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs) [60], where MLPs consist of layers *densely* connected through a weight matrix, and CNNs of sparsely connected performing filter convolutions, used in conjunction with *MaxPool* downsampling layers. More information on MPLs and CNNs appears in Appendix B.

The aforementioned weights and filters consist of parameters, learned through the training process. We refer to the collection of all such parameters belonging to an ANN, flattened into a 1D array, as the *parameter vector*, denoted as θ_t, at iteration t of SGD. This abstraction is used in subsequent sections when arguing regarding consistency and progress. Non-linear activation functions are applied after each layer, where common choices are the $ReLU$ function $\sigma(x) = max(0, x)$ for all layers except the last, where instead the *softmax* activation function $\sigma_i(x) = e^{x_i} / \sum_{j=1}^{|x|} e^{x_j}$, for each output neuron i, is used in order to acquire a predicted probability distribution. With this, an error measure $f(\theta)$ can be defined, the minimization of which constitutes the training process.

The *metrics* of interest are (i) *statistical efficiency*, i.e. the number of SGD iterations required until reaching an error threshold $f(\theta^*) < \epsilon$, i.e. ϵ-convergence (ii) *computational efficiency* measuring the wall-clock time per iteration and, most importantly (iii) the *overall convergence rate*, i.e. the wall-clock time until ϵ-convergence, of most relevance in practice.

3.2.2 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) [10] on single-word locations. Each thread A computes SGD updates (3.1) according to a predefined algorithm, in the context outlined in the previous paragraphs. Since A

must read the current state θ_t prior to computing the corresponding stochastic gradient $\nabla f(\theta_t)$, 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_t vector and A's update to apply its calculated gradient $\nabla f(\theta_t)$, 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 [25]. We refer to Section (3.3) 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 (3.1) instead will follow

$$\theta_{t+1} \leftarrow \theta_t - \eta \nabla f(v_t) \tag{3.2}$$

where $v_t = \theta_{t-\tau_t}$ is the thread's *view* of θ.

3.2.3 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. [10]. 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 [10].

3.2.4 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.

3.3 The *Leashed-SGD* framework

In the following we define *Leashed-SGD* along with the proposed *ParameterVector* data structure's common interface, containing the values of the parameter vector, as well as metadata used for memory recycling. We also express *Async-SGD* and HOGWILD! using this interface; both are well established versions of parallel SGD implementations [9, 31]. Modified versions, optimized for specific applications, have been proposed, e.g. in [61], 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.

3.3.1 Introducing *ParameterVector*

Considering (3.1), 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 *ParameterVector* can be local or shared among threads. For concurrent accesses to it, its implementation can provide certain consistency and progress guarantees (cf. section 3.2). 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 2: *ParameterVector* core components

```
1  Float[d] theta // vector of dimension d
2  Int t ← 0                    // sequence number of the most recent update of theta
3  Int n_rdrs ← 0
4  Bool stale_flag ← false, deleted ← false
5  Function rand_init():
6      theta ← N(0, 0.01)
7  Function safe_delete():
8      if stale_flag ∧ n_rdrs = 0 ∧ CAS(deleted, false, true) then
9          delete theta
10 Function start_reading():
11     param.n_rdrs.fetch_add(1)
12 Function stop_reading():
13     n_rdrs.fetch_add(−1)
14     self.safe_delete()
15 Function update(δ, η):
16     t.fetch_add(1)
17     for i = 0, ..., d − 1 do
18         theta[i] ← theta[i] − η · δ[i]
```

Algorithm 2 describes the core components for the algorithmic implementation of *ParameterVector*. A main one is the array *theta* of dimension d (typically a very large number in DL applications, e.g. in the well-known AlexNet [62] CNN architecture there are 62,378,344 parameters). A read of the parameters can be accomplished by getting a pointer to *theta*, while function `update()` performs the addition (3.2) on *theta*. Notice that algorithm 2 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 [63,64], 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 3: *AsyncSGD*

```
1  GLOBAL ParamVector PARAM
2  GLOBAL Float η                                          // step size
3  GLOBAL Lock mtx                          // for accessing shared parameters
4  Initialization;
5  PARAM ← new ParamVector()
6  PARAM.rand_init()                         // randomly initialize parameters
7  Each thread;
8  local_grad ← new ParamVector()                      // local gradient memory
9  local_param ← new ParamVector()
10 repeat
11     mtx.lock()
12     local_param.theta = copy(PARAM.theta)
13     mtx.unlock()
14     local_grad.theta ← comp_grad(local_param.theta)
15     mtx.lock()
16     PARAM.update(local_grad.theta, η)
17     mtx.unlock()
18 until convergence;
```

3.3.2 Baselines outline

Algorithm 3 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 3.5. 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 3, 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. The algorithm outline is available in Appendix B. For problems with sparse gradients the lack of synchronization will not significantly impact the convergence, since the `update()` operation will only influence a few of the d components in *theta*. For DL applications though, its influence is not well understood.

3.3.3 *Leashed-SGD*: Lock-free consistent *AsyncSGD*

The key points and arguments supporting *Leashed-SGD*, which is shown in pseudocode in Algorithm 4, using *ParameterVector* core components from Algorithm 2, 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. 3.2). As a successful CAS replaces the previous "global" vector, copies of parameter vectors that become global are *totally ordered* on their sequence number, t. 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

Algorithm 4: *Leashed-SGD*

```
1  GLOBAL ParamVector ** P                    // address to latest pointer (cf. Fig. 3.2)
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
                 recycled and repeat to get a fresher one
12     until break;
13 Initialization;
14 init_pv ← new ParamVector()                          // pointer to initial parameters
15 init_pv.rand_init()                                // randomly initialize parameters
16 P ← &init_pc                                                  // address of initial pointer
17 Thread i;
18 local_grad ← new ParamVector()                                // local gradient memory
19 repeat
20     latest_param ← latest_pointer()
21     local_grad.theta ← comp_grad(latest_param.theta)
22     latest_param.stop_reading()
23     new_param ← new ParamVector()                                        // new parameters
24     Int num_tries ← 0                                   // prepare for the LAU-SPC loop
25     repeat
26         latest_param ← latest_pointer()
27         new_param.theta = copy(latest_param.theta)
28         new_param.t = latest_param.t
29         latest_param.stop_reading()
30         new_param.update(local_grad.theta, 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;
```

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 *ParameterVector* should persist due to active readers.

P3. *Lock-free atomic reads of the shared vector:* To access the global *ParameterVector* threads acquire a *pointer* to the most recent by accessing P. Through that pointer, the thread can access and use the *theta* 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 *ParameterVector* 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 4, 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 *LoadAndUpdateStorePersistenceConditional* (*LAU-SPC*). Note that bounded T_p essentially implies bounded retries. As formulated in (3.2), 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 *ParameterVector* instance a second time (Figure 3.2), 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 2 *Reads and updates of the θ vector by Leashed-SGD,* `latest_pointer()` *function and* LAU-SPC *loop, satisfy lock-freedom and atomicity.*

Lemma 3 *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.*

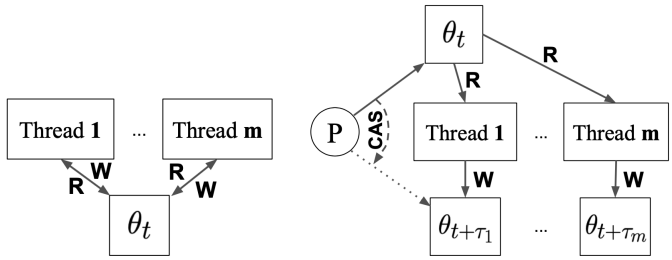

Figure 3.2: Illustration of data access in *AsyncSGD* and HOGWILD! (*left*) and *Leashed-SGD* (*right*). For *AsyncSGD* the read and write operations are protected through mutual exclusion. For *Leashed-SGD*, each thread accesses θ_t only through a read operation, then computes the update at a new memory location, becoming a candidate for $\theta_{t+\tau}$

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 3.5.

3.4 Contention and staleness

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

3.4.1 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_t, μ_t varying over time, depending on the current state of the system.

Table 3.1: 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. 3.3, 3.4
S2	MLP*	High-precision convergence	16	$50\%, 10\%, 5\%, 2.5\%$	0.05	Fig. 3.5-3.7
S3	CNN*	Convergence rate	16	$75\%, 50\%, 25\%, 10\%$	0.05	Fig. 3.8
S4	MLP*	High parallelism	24, 34, 68	$75\%, 50\%, 25\%, 10\%$	0.05	Fig. 3.5-3.7
S5	MLP*, CNN*	Memory consumption	16, 24, 34	any	0.05	presented in text*

*Details appear in Appendix B

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()`. In summary:

$$\lambda_t^{(m)} = \frac{m - n_t}{T_c}, \quad \mu_t = \frac{n_t}{T_u} \tag{3.3}$$

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

$$n_{t+1} = n_t + \frac{m - n_t}{T_c} - \frac{n_t}{T_u} \tag{3.4}$$

where n_t is the number of threads executing the retry loop at time t. Note that the system (3.4) 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 3.5 we show closer measurements of T_c, T_u for different applications.

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

Theorem 7 *Assume we have an m-thread system where threads arrive to and depart from the Leashed-SGD* LAU-SPC *loop with the rates in (3.3). Then, we have that the number n_t of threads in the retry-loop at time t is given by*

$$n_t = \frac{1 - (1 - T_c^{-1} - T_u^{-1})^t}{1 + T_c/T_u} m + (1 - T_c^{-1} - T_u^{-1})^t n_o \tag{3.5}$$

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.

The proof appears in Appendix B.

Corollary 5 *The fixed point n^* is stable, and the system will converge towards $\lim_{t\to\infty} n_t = n^*$ for any initial n_0.*

The result is confirmed by taking $t \to \infty$ in (3.5).

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*, which will be used in the following.

3.4.2 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:

$$\mu_t = \frac{n_t}{T_u}(1 + \gamma) \tag{3.6}$$

Corollary 6 *Under the same conditions as in Theorem 7, but using the departure rate (3.6), the fixed point moves to*

$$n^*_\gamma = \big(\frac{T_c}{T_u}(1+\gamma)+1\big)^{-1} m \tag{3.7}$$

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 [53], the complete staleness τ_t of an update $\nabla f(v_t)$ according to (3.2) is comprised of two parts: $\tau_t = \tau^c_t + \tau^s_t$ where τ^c_t counts the number of published updates concurrent to the computation of $\nabla f(v_t)$, and τ^s_t 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 $\mathbf{E}[\tau^s_t] \approx n^*_\gamma$, 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 $\tau^s_t = 0$, which is the maximum staleness reduction possible here. In section 3.5 we study this empirically, showing it holds in practice and is effective for regulating contention and tune the staleness.

3.5 Evaluation

We present the results from our extended empirical study, benchmarking the methods in Section 3.3, studying influence of consistency and associated synchronization, on the metrics described in Section 3.2: 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).

3.5.1 Implementation

The algorithms and the framework are implemented with C++, with OpenMP [65] for shared-memory parallel computations, and Eigen [66] for numerical. The framework extends the MiniDNN [67] 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* application-specific and apply as parallelization of SGD for any optimization problem, in particular of high dimension. For the empirical evaluation an extensible framework is implemented in conjunction with ANN operations, facilitating further research exploring algorithms for parallel SGD for DL with various synchronization mechanisms.

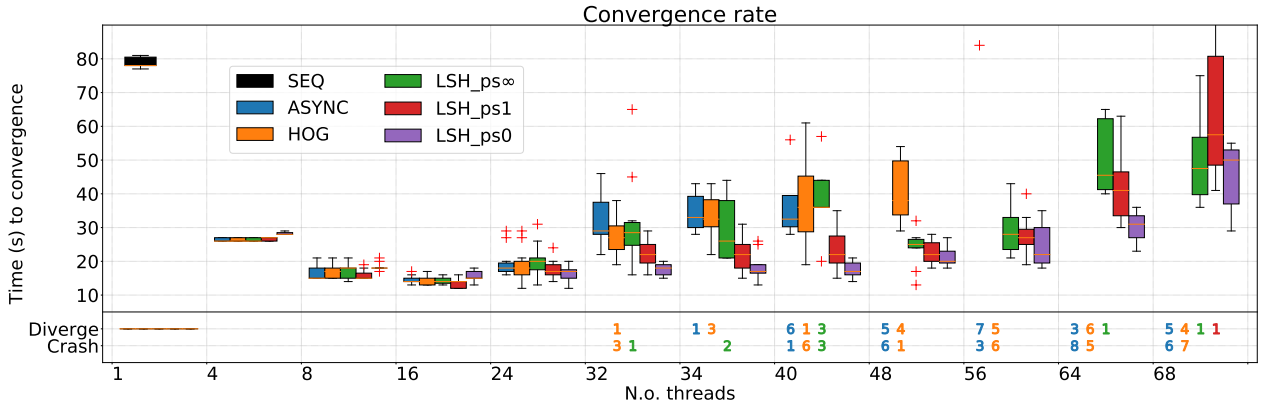

Figure 3.3: ϵ-convergence rate for MLP training under varying parallelism, measuring wall-clock time until reaching an error threshold ($\epsilon = 50\%$ of initial error, providing a comparison of the general scalability). The optimum for the baselines ($m = 16$) is used in subsequent tests for a fair 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.

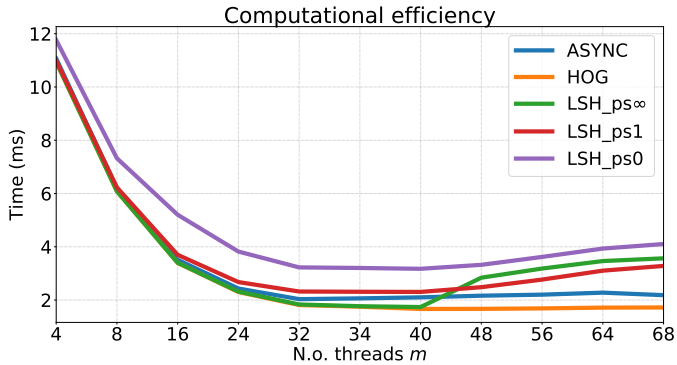

Figure 3.4: Computational efficiency, i.e. wall-clock computation time per SGD iteration. Computation time remains constant for the baselines under higher parallelism, although the many executions fail completely to converge, and would be wasted time in practice. The self-regulative property of *Leashed-SGD* 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.

3.5.2 Experiment setup

We evaluate the methods of Section 3.3 for two DL applications, namely MLP and CNN training on the MNIST benchmarking dataset [68]. 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 available in Appendix B. The size of the parameter vector θ are $d = 134,794$ and $d = 27,354$ for MLP and CNN, respectively. 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 [59]. It is vital that training succeeds, and that the execution time thereby is not wasted. The threshold ϵ is specified

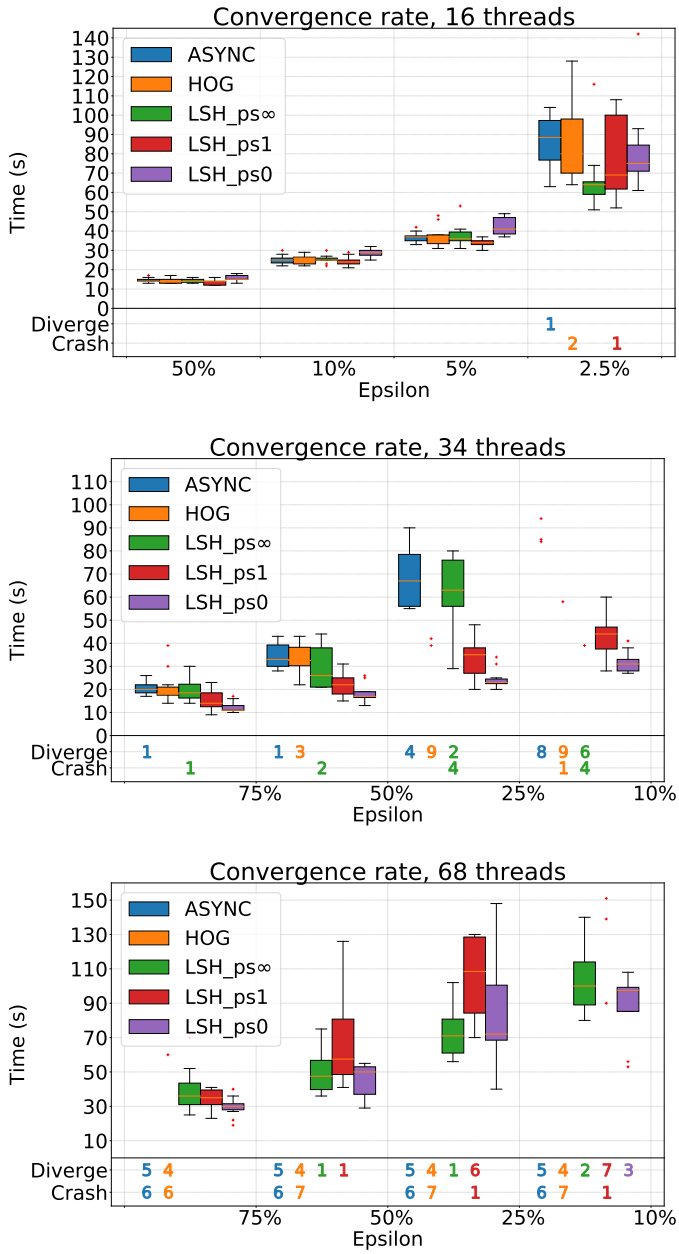

Figure 3.5: ϵ-convergence rate for MLP with $m = 16$ threads to high precision (*top*), $m = 34$ threads (*middle*) and maximum parallelism $m = 68$ threads (*bottom*). The baselines (ASYNC, HOG) show an overall slower convergence and higher number of executions that fail before reaching the high precision (e.g. $\epsilon = 10\%$), especially under maximum parallelism $m = 68$, where no baseline execution managed to reach $\epsilon = 50\%$ of the error at initialization.

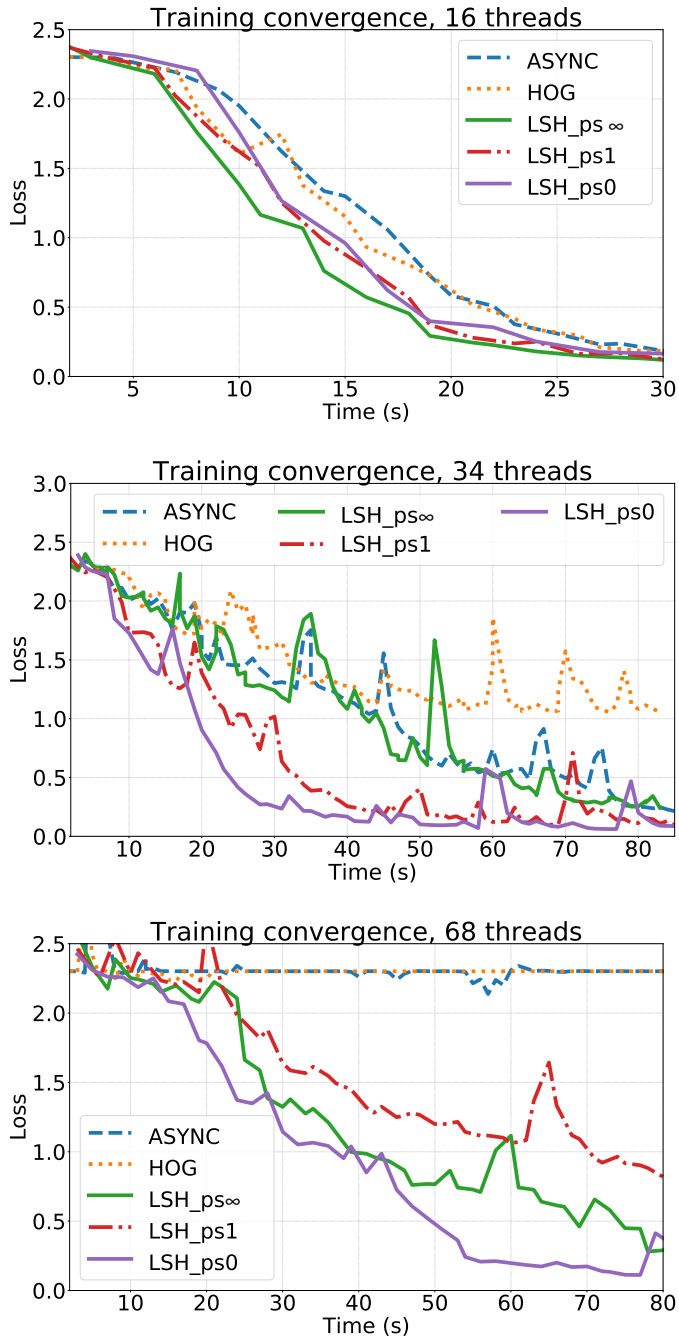

Figure 3.6: MLP training progress over time with $m = 16$ threads (*top*), with $m = 34$ threads (*middle*) and maximum parallelism $m = 68$ threads (*bottom*). The proposed framework (LSH_psX, persistence bound X) converges significantly faster relative to baselines (ASYNC, HOG). Under maximum parallelism, the baselines completely fail to converge and oscillate around the initialization point.

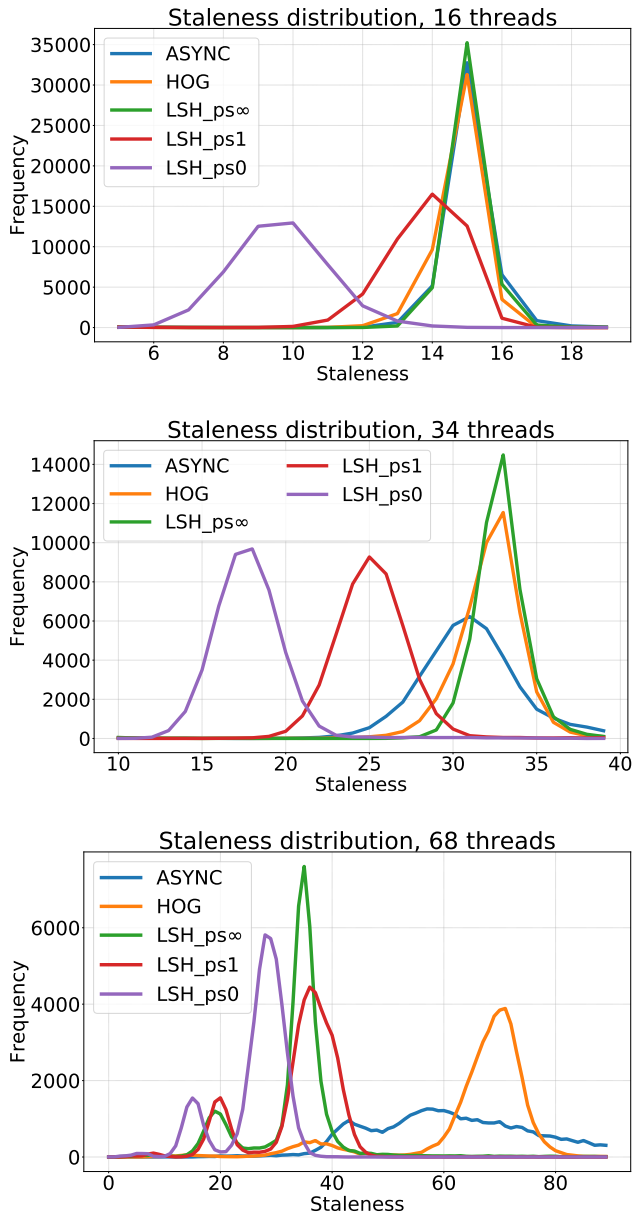

Figure 3.7: Staleness distribution over time for MLP with $m = 16$ threads (*top*), $m = 34$ threads (*middle*) and maximum parallelism $m = 68$ threads (*bottom*). The effect from the contention-regulating persistence bound ($ps \in \{0, 1, \infty\}$) is clear, and effectively reduces the overall staleness 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.

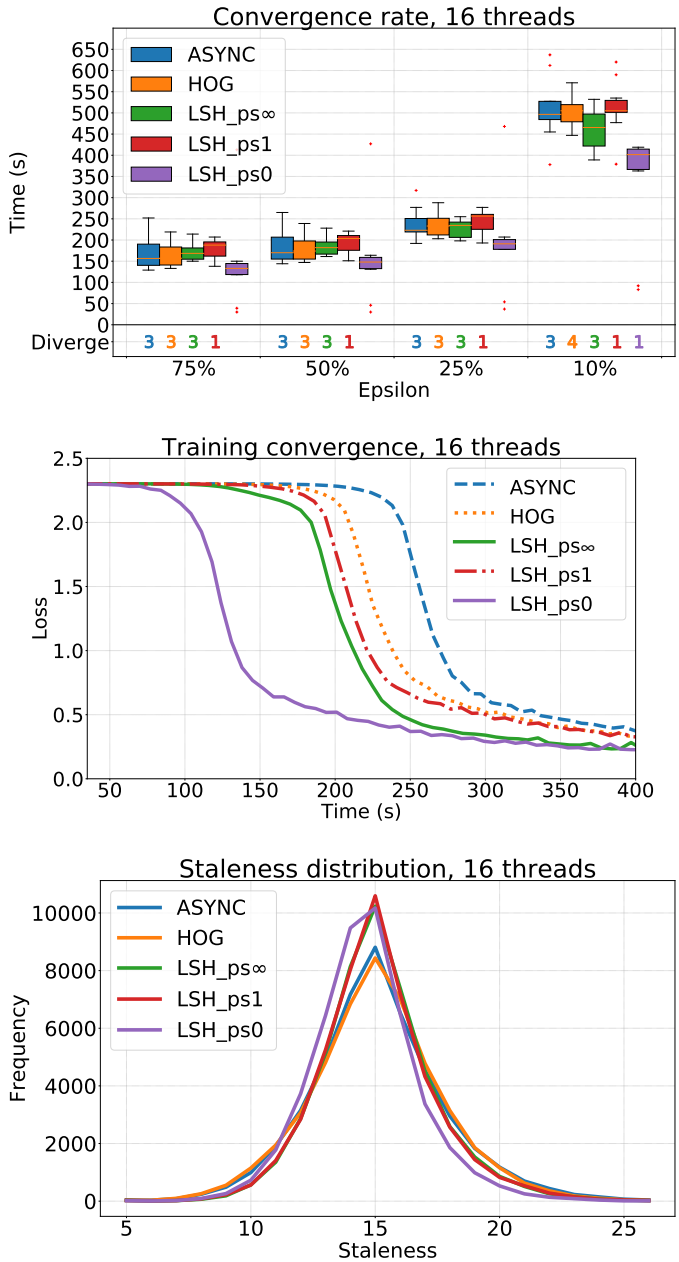

Figure 3.8: ϵ-convergence rates to different precision for CNN training with $m = 16$ threads; LSH_ps0 shows 400s median 10%-convergence time, compared to $\sim$ 500s for the baselines, with two executions showing remarkable 10%-convergence time of below 100s, i.e. a $4\times$ speedup relative to the 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.

in terms of percentage of the target function at initialization $f(\theta_0) \approx 2.3$.

3.5.3 Experiment outcomes

The steps of our experiment methodology, summarized in Table 3.1, 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. 3.3-3.4, 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 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 appears in Appendix B, 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 and their convergence rate 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 3.4.
Using $m = 16$, $\eta = 0.005$, we benchmark *Leashed-SGD* and baselines to high-precision 2.5%-convergence, measuring the wall-clock time (Fig. 3.5, 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 hypothesised in section 3.4, Fig. 3.7 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* shows fewer diverging executions, with significant improvements in time to high precision convergence with up to 4× speedup relative to the baselines *AsyncSGD* (Fig. 3.8). Measurements of memory consumption and computation times (T_c, T_u) appear in Appendix B. Due to the sparse nature of the CNN topology, the gradient computation vs. update application time ratio T_c/T_u is high, leading to a significantly reduced memory footprint (with 17% on average) of *Leashed-SGD*.
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. 3.5-3.7, 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, *Leashed-SGD* reduces the memory consumption by 17% on average thanks to dynamic allocation of *ParameterVector* and efficient memory recycling. The detailed plots appear in Appendix B.

3.5.4 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 [69–71].

3.6 Related Work

The study of numerical methods under parallelism sparked due to the works by Bertsekas and Tsitsiklis [18]. Distributed and parallel asynchronous SGD has since been an attractive target of study, e.g. [14, 19, 22, 57], among which HOGWILD! [9]. In the recent [8] 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 [24] introduced a framework for analysis of HOGWILD!-style algorithms. This was extended in [25], 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 [21] 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.

In [6], 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. Here we propose an extensible 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 parallelise.

In [26], 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 parallelisation benefits. To alleviate this, they propose the use of static analysis for identification of data that do not cause dependencies, for parallelising 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 [13, 32, 33] and in

particular in [53] 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 [27], 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 [28] theoretical convergence analysis is presented for *SyncSGD* with *once-in-a-while* synchronization. They mention the analysis can guide in applying *SyncSGD* for DL, however the analysis requires strong convexity. [29] 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 [30] a topology for decentralized parallel SGD is proposed, using pair-wise averaging synchronization. In the recent [7] 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-freedom* that follows from the lack of synchronization.

3.7 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 provides new useful insights in the applicability of *AsyncSGD* in practice.

We observe that the baselines, i.e. standard implementations of *AsyncSGD*, are very 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 behaviour, 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

exploring algorithms for parallel SGD for DL with various synchronization mechanisms.

Appendix B

Paper B

On MLPs and CNNs

MLPs consist of several stacked densely-connected layers of neurons, each applying a non-linear transformation of the input and passing the result to the next layer:

$$o_n^{(l)} = \sigma\left(\sum_{i=0}^{|N_{l-1}|-1} \theta_i^{(l,n,w)} \cdot o_i^{(l-1)} + \theta^{(l,n,b)}\right)$$

where $o_n^{(l)}$ is the output of neuron $n \in \{0, \ldots, N_l - 1\}$ in the lth 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 nth neuron.
CNNs consist of *convolutional* layers, convolving the input with learnable filters for feature detection:

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

for a number of filters f, corresponding to a 1D convolution, but can be naturally extended to 2D. Convolutional layers are sparsely connected, reducing the number of weights to be trained, and are especially efficient for analysis of image/spatial data due to the translation-invariant property of feature detection with convolution. Convolutional layers are often used in combination with *MaxPool* layers, which map the output of a number of consecutive neurons onto their maximum. This significantly reduces dimension of the signal and the learnable weights.

We refer to the collection of all parameters $\theta^{(l,n,w/b)}$, $\theta^{(l,f,w/b)}$ belonging to an ANN flattened into a 1D array as the *parameter vector*, denoted as θ_t, at iteration t of SGD. This abstraction is used in subsequent sections when arguing regarding consistency and progress.

In the output layer of an ANN, the *softmax* activation function $\sigma_i(x) = e^{x_i} / \sum_{j=1}^{|x|} e^{x_j}$, for each output neuron i, is often used for classification problems,

outputting an estimated class distribution y of an input x. Given the true class/label $\hat{y}$, the ANN performance is quantified by the cross-entropy loss function:

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

where y contains the outputs from the last layer, and depends on the input x and the current state of θ. The *training* process for ANNs then constitutes of iteratively adjusting θ to minimize the error function $f(\theta) = L(\hat{y}, y(x:\theta))$. The *BackProp* algorithm is used for computing $\nabla_\theta f(\theta)$, 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 *batch* considered in conjunction.

Analysis - complementary material

Proof sketch - Lemma 2: The first claim (i) follows from the definition of the safe_delete operation of the *ParameterVector*, ensuring that the memory of an instance PC_t is reclaimed only if $stale_flag = true$ (P points to a newer instance, ensuring no new readers of PC_t), $n_readers = 0$ (no readers currently) and 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 the threads that are currently using the instance will each 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 active reader, and an additional two *ParameterVector* (*new_param* and *local_grad*), giving in total $3m$. ■

Proof of Theorem 3 From (4), we have

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

■

Algorithm 5: HOGWILD! expressed using the *ParameterVector* interface

```
1  GLOBAL ParamVector PARAM
2  GLOBAL Float η                                          // step size
3  Initialization;
4  PARAM ← new ParamVector()
5  PARAM.rand_init()                          // randomly initialize parameters
6  Each thread;
7  local_grad ← new ParamVector()                  // local gradient memory
8  local_param ← new ParamVector()
9  repeat
10     local_param.theta = copy(PARAM.theta)
11     local_grad.theta ← comp_grad(local_param.theta)              // gradient
12     PARAM.update(local_grad.theta, η)
13 until convergence;
```

Evaluation - complementary material

The details of the ANN architectures implemented in the evaluation (Section 5) are shown in Table B.1 and B.2 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, $d = 27,354$

Convergence and hyper-parameter selection

Figure B.1 shows the convergence rate for different values of step size η. The baselines *AsyncSGD* and HOGWILD! show the best performance for $\eta = 0.005$, which is hence used in the subsequent test stages.

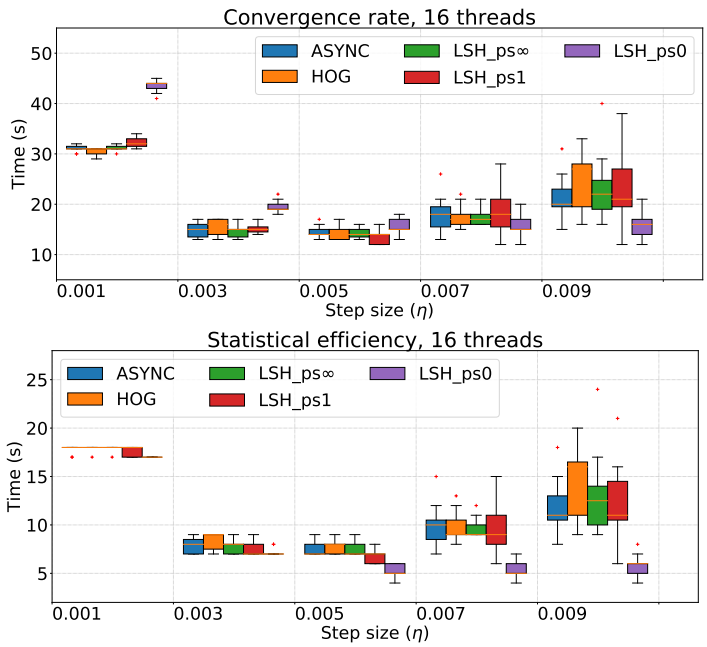

Figure B.1: Step size tuning (top), confirming the choice $\eta = 0.005$, and statistical efficiency (bottom), showing 50%-convergence

Gradient computation and update time - T_c, T_u

The distribution of the wall-clock time to compute and apply gradients, respectively, are shown in Figure B.2. Despite having a lower dimensionality, the gadient computation time T_c is higher for CNN. This is due to the topological nature of the convolutional layer, where filters are strided 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 the ; 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.

Memory consumption

Figure B.3 shows the distribution of the memory consumption of the different algorithms for MLP and CNN training. The measurements were acquired using the UNIX `ps` command, collected with second granularity.

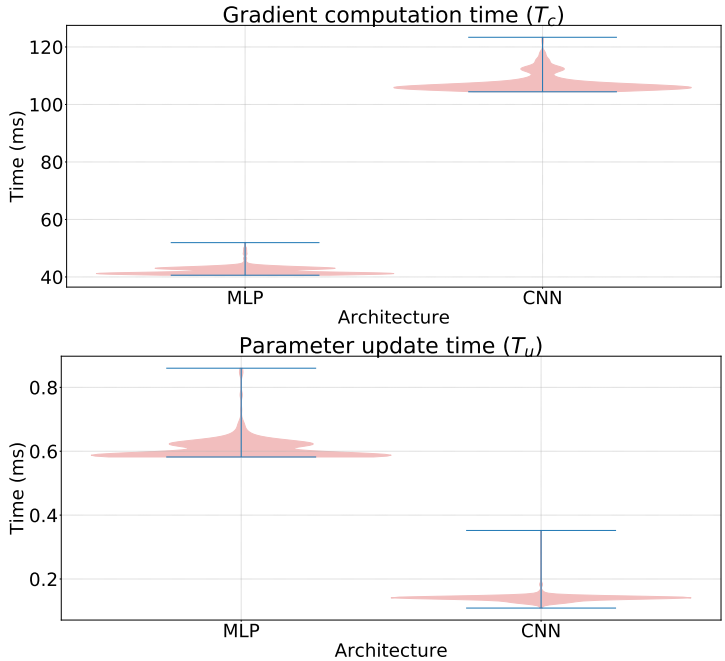

Figure B.2: Gradient computation and parameter update times T_c, T_u (top, bottom, respectively) for MLP and CNN

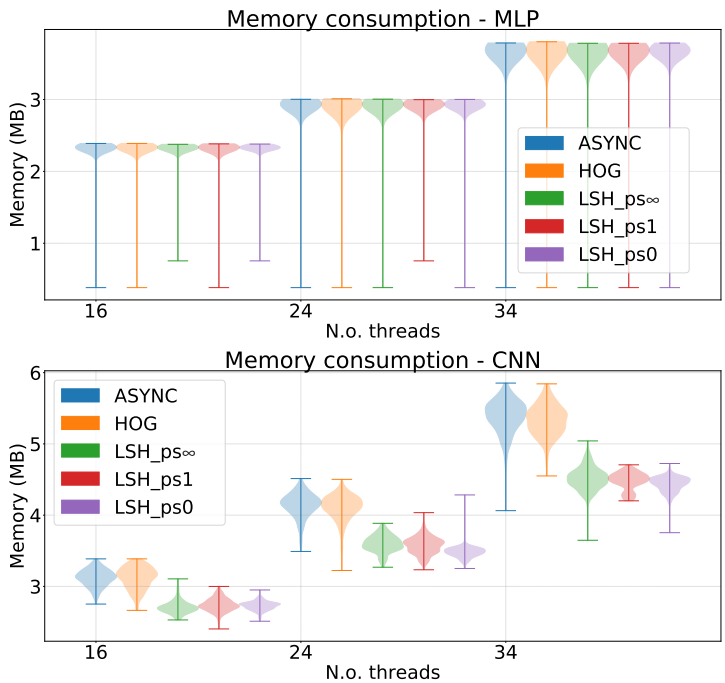

Figure B.3: Memory consumption measured continuously on second granularity for MLP (top) and CNN (bottom)

Bibliography

[1] A. M. Turing, "Computing machinery and intelligence," in *Parsing the turing test*. Springer, 2009, pp. 23–65.

[2] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018.

[3] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Information sciences*, vol. 275, pp. 314–347, 2014.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] D. Alistarh, B. Chatterjee, and V. Kungurtsev, "Elastic consistency: A general consistency model for distributed stochastic gradient descent," *arXiv preprint arXiv:2001.05918*, 2020.

[9] 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.

[10] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and distributed computation: numerical methods*. Prentice Hall, 1989, vol. 23.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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

[25] 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

[26] 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.

[27] 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.

[28] S. U. Stich, "Local SGD converges fast and communicates little," in *Int'l Conf. on Learning Representations (ICLR)*, 2019.

[29] 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.

[30] 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.

[31] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *Advances in Neural Inf. Proc. Systems*, 2011, pp. 873–881.

[32] 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

[33] S. Sra, A. W. Yu, M. Li, and A. J. Smola, "Adadelay: Delay adaptive distributed stochastic convex optimization," *arXiv preprint arXiv:1508.05003*, 2015.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-SGD for distributed deep learning," *arXiv preprint arXiv:1511.05950*, 2015.

[41] P. J. Werbos, "Applications of advances in nonlinear sensitivity analysis," in *System modeling and optimization*. Springer, 1982, pp. 762–770.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[48] 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.

[49] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv:1609.04747*, 2016.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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*, 2017, pp. 1273–1282.

[55] 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.

[56] Y. Ma, F. Rusu, and M. Torres, "Stochastic gradient descent on highly-parallel architectures," *arXiv preprint arXiv:1802.08800*, 2018.

[57] J. C. Duchi, S. Chaturapruek, and C. Ré, "Asynchronous stochastic convex optimization," *arXiv preprint arXiv:1508.00882*, 2015.

[58] 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.

[59] 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.

[60] A. Baldominos, Y. Saez, and P. Isasi, "A survey of handwritten character recognition with mnist and emnist," *Applied Sciences*, vol. 9, no. 15, p. 3169, 2019.

[61] 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.

[62] A. Krizhevsky, I. Sutskever, and G. Hinton, "2012 alexnet," *Adv. Neural Inf. Process. Syst.*, pp. 1–9, 2012.

[63] 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.

[64] 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.

[65] 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.

[66] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[67] Y. Qiu, "Minidnn," https://github.com/yixuan/MiniDNN/, 2020.

[68] Y. LeCun and C. Cortes, "MNIST handwritten digit database," *Dat repository*, 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[69] 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.

[70] 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.

[71] 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.