CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Asynchronous Parallel Stochastic Gradient Descent

A study of the influence of synchronization methods and hyperparameters

Master's thesis in Computer science and engineering

Hampus Ek

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Asynchronous Parallel Stochastic Gradient Descent

A study of the influence of synchronization methods and hyperparameters

Hampus Ek

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Asynchronous Parallel Stochastic Gradient Descent
A study of the influence of synchronization methods and hyperparameters
Hampus Ek

Supervisor: Philippas Tsigas, Department of Computer Science and Engineering
Examiner: Marina Papatriantafilou, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Asynchronous Parallel Stochastic Gradient Descent
A study of the influence of synchronization methods and hyperparameters
Hampus Ek
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Artificial Neural Networks (ANN) can solve complex tasks and be found in applications such as language translation, object recognition, and more. The underlying optimization algorithm for training ANNs is often Stochastic Gradient Descent (SGD). SGD is a first-order numerical optimization algorithm that repeatedly takes a step in the negative gradient direction of a loss function.
Training ANNs generally require a large amount of data, and with an increasing amount of data, more complex tasks can be learned. However, more data increases the training time since several passes through the data are required. The training time can be reduced by parallelizing the training process. There are several ways to do this, and parallelizing the SGD iterations is one way. Parallelizing the SGD iterations can mainly be done in two different ways, synchronously or asynchronously. The asynchronous parallel SGD has shown performance benefits over the synchronous approach and has therefore gained increased attention in recent literature.
However, asynchrony introduces challenges with understanding the execution and convergence criteria of SGD. These challenges originate from the fact that asynchrony allows for gradient updates on stale (old) views of the state. Parallel SGD and asynchronous parallel SGD, in particular, can make hyperparameter tuning even more time-consuming and challenging compared to regular sequential SGD. Parallelization of SGD introduces an additional dimension, e.g., the level of parallelism, to the training phase. Increased parallelism also increases the risk of crashed executions.
This work aims to increase the understanding of the convergence properties of SGD under asynchronous parallelism. This is done by (i) analyzing how the memory model affects convergence under different levels of parallelism. (ii) What impact batch size and step size have on convergence under a varying level of parallelism is also analyzed. (iii) Moreover, an analysis of how the staleness distribution is affected by different batch sizes is made. Furthermore, (iv) backoff methods are tested to reduce contention for the lock-based algorithms and *Leashed-SGD*.
An alternative lock-based approach to regular mutex lock is proposed using a read-write lock. The read-write lock, mutex lock, and two lock-free algorithms, *Leashed-SGD* [1] and HOGWILD! [2], are compared in all of the dimensions stated in the previous paragraph (i-iv).
We can empirically see that the memory model used impacts convergence, where NUMA converges faster than UMA for both lock-based and lock-free AsyncSGD. Further, the level of parallelism has a high impact on what hyperparameters to use. The level of parallelism is also related to the number of crashed and diverged executions, where higher parallelism increases the risk of crashed executions.

Keywords: Stochastic gradient descent, parallelism, asynchrony, machine learning

Acknowledgements

First and foremost, I am incredibly grateful to my supervisor, Philippas Tsigas, for his advice, support, and guidance. I would also like to thank Marina Papatriantafilou and Karl Bäckström for their technical support and guidance. The weekly meetings with Philippas, Marina, and Karl have been invaluable for my work and a great source of inspiration. Finally, I would like to express my gratitude to my friends and family for their support and encouragement.

Hampus Ek, Gothenburg, June 2021

Contents

List of Figures **xi**

List of Tables **xv**

1 Introduction **1**
1.1 Statement of the problem . . . 2
1.2 Purpose of the study and hypotheses . . . 3
1.3 Limitations and Delimitations . . . 3

2 Theory and Background **5**
2.1 Stochastic Gradient Descent . . . 5
2.1.1 Metrics of interest . . . 7
2.1.2 Parameters and hyperparameters . . . 8
2.1.3 Artificial Neural Networks . . . 10
2.1.4 The training process . . . 11
2.2 Parallel Stochastic Gradient Descent . . . 11
2.2.1 Parallel computer architecture and parallelization . . . 12
2.2.2 Synchronous parallel SGD . . . 13
2.2.3 Asynchronous parallel SGD . . . 14
2.2.4 A note on hyperparameters and parallel SGD . . . 17
2.3 Problems and challenges . . . 18
2.3.1 Convergence and asynchrony . . . 18
2.3.2 Scalability . . . 18
2.3.3 Testing and Benchmarking . . . 19

3 Methods **21**
3.1 Algorithms and fine-grained synchronization . . . 22
3.2 Memory model and parallel SGD . . . 25
3.3 Convergence and hyperparameter selection . . . 25
3.4 Staleness and hyperparameter selection . . . 25
3.5 Backoff . . . 26

4 Empirical study **27**
4.1 Experiment setup . . . 27
4.2 Memory model and convergence . . . 29
4.3 Convergence analysis . . . 32
4.4 Staleness distribution and batch size . . . 42

4.5 Convergence comparison 44
4.6 Backoff 47
4.7 Discussion of Experiments 48

5 Conclusions and future work **51**

A Appendix A **I**
A.1 Convergence analysis I

List of Figures

2.1 Illustration of a Stochastic Gradient Descent iterative optimization algorithm for a two-dimensional target function. SGD repeatedly takes a step in the negative gradient direction of the target function . 6

2.2 Performance and Accuracy of SGD with respect to mini-batch size for a fixed number of steps (illustration), figure adopted from [3] . . . 9

2.3 Example of a feedforword DNN architecture with N hidden layers and n input-layers. The output of each layer is the input of the next layer as illustrated 10

2.4 Example of a CNN architecture 11

2.5 Example of a UMA(left) and NUMA(right) architecture 13

2.6 Data access for AsyncSGD and HOGWILD! (left) and *Leashed-SGD* (right). AsyncSGD uses locks to ensure mutual exclusion to the shared state, HOGWILD! uses component-wise atomic read and write of the shared state. In Leashed-SGD each thread only access θ_t through a read operation. The update are then calculated locally and stored at a new memory location that becomes a candidate for $\theta_{t+\tau}$, figure adopted form [1]. 17

4.1 MNIST handwritten digits example 27

4.2 Thread scheduling comparison for 7 threads. NUMA refers to 4 threads scheduled on socket 0 and 3 threads scheduled on socket 1, and UMA refers to 7 threads scheduled to socket 0. Time to reach 2 %-convergence (top). Average epoch time (e.g. Computational efficiency) (bottom left) and number of epoch to reach 2 %-convergence (e.g. statistical efficiency) (bottom right). Step size $\eta = 0.005$ and batch size $B = 64$. 29

4.3 Thread scheduling comparison for 8 threads. NUMA refers to 4 threads scheduled on socket 0 and 4 threads scheduled on socket 1 and UMA refers to 8 threads scheduled to socket 0. Average epoch time (e.g. Computational efficiency) (left) and number of epoch to reach 2 %-convergence (e.g. statistical efficiency) (right). Step size $\eta = 0.005$ and batch size $B = 64$. 30

4.4 Staleness distribution for eight and seven threads with UMA and NUMA memory model. t1-t8 indicate each individual thread and *Total* are the total staleness. Blue represent threads scheduled on socket 0 and orange threads scheduled on socket 1. Batch size, $B = 64$ and step size, $\eta = 0.005$. 31

4.5 Convergence time to reach 5 % of initial loss for AsyncSGD with mutex lock, 2 and 4 threads for UMA and NUMA. Crashed executions indicate number of crashed executions. Based on ten independent runs. 33

4.6 Convergence time to reach 5 % of initial loss (left) for AsyncSGD with mutex lock, 8, 16 and 32 threads, UMA and NUMA for 8 threads and OS scheduler for 16 and 32 threads. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. 34

4.7 Convergence time (left) to reach 5 % of initial loss for AsyncSGD with mutex lock, 2 and 4 threads for UMA and NUMA. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. 35

4.8 Convergence time to reach 5 % of initial loss (left) for AsyncSGD with rw-lock, 8, 16 and 32 threads, UMA and NUMA for 8 threads and OS scheduler for 16 and 32 threads. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. 36

4.9 Convergence time (left) to reach 5 % of initial loss for *Leashed-SGD* with persistence bound $ps = \infty$, 2 and 4 threads for UMA and NUMA. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. 37

4.10 Convergence time to reach 5 % of initial loss (left) for *Leashed-SGD* with persistence bound $ps = \infty$, 8, 16 and 32 threads, UMA and NUMA for 8 threads and OS scheduler for 16 and 32 threads. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. 38

4.11 Convergence time (left) to reach 5 % of initial loss for HOGWILD!, 2 and 4 threads for UMA and NUMA. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. 39

4.12 Convergence time to reach 5 % of initial loss (left) for HOGWILD!, 8, 16 and 32 threads, UMA and NUMA for 8 threads and OS scheduler for 16 and 32 threads. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. 40
4.13 Staleness distribution for AsyncSGD with mutex lock, based on the average of 10 independent executions. 42
4.14 Staleness distribution for AsyncSGD with RW-lock lock, based on the average of 10 independent executions. 42
4.15 Staleness distribution for Lsashed-SGD [1], with persistence bound $ps = \infty$, based on the average of 10 independent executions. 43
4.16 Staleness distribution for Lsashed-SGD [1], with persistence bound ps=1, based on the average of 10 independent executions. 43
4.17 Staleness distribution for Lsashed-SGD [1], with persistence bound ps=0, based on the average of 10 independent executions. 44
4.18 Staleness distribution for HOGWILD! [2], based on the average of 10 independent executions. 44
4.19 Convergence rate for MLP with $\epsilon = \{10\%, 5\%, 2\%\}$ of the initial error, maximum level of parallelism m=8 and minimum level of parallelism m=2. UMA (left), NUMA (right), step size (η) and batch size (B) are indicated in the legend and in Table 4.3. Diverge indicate the number of times respective algorithm failed to reach ϵ-convergence within 60 seconds and crash indicate the number of times the execution crashed. PS indicate the persistence bound for *Leashed-SGD*. Based on ten independent runs of each setting. 45
4.20 Convergence rate for MLP with $\epsilon = \{10\%, 5\%, 2\%\}$ of the initial error, for 16 and 32 threads. Step size (η) and batch size (B) are indicated in the legend and in Table 4.3. Diverge indicate the number of times respective algorithm failed to reach ϵ-convergence within 60 seconds and crash indicate the number of times the execution crashed. PS indicate the persistence bound for *Leashed-SGD*. Based on ten independent runs of each setting. 46
4.21 Average epoch time (e.g. computational efficiency, left) and number of epochs to reach 2 %-convergence (e.g. statistical efficiency, right) for eight threads. Diverge indicate the number of times respective algorithm failed to reach ϵ-convergence within 60 seconds and crash indicate the number of times the execution crashed. PS indicate the persistence bound for *Leashed-SGD*. Based on ten independent runs, settings presented in Table 4.3 46
4.22 Convergence rate with linear, exponential, random and no backoff, seven (left) and eight (right) threads. ϵ =2 %, settings used are presented in Table 4.3. 47
4.23 Convergence rate of AsyncSGD (mutex- and rw-lock) and *Leashed-SGD* with linear, exponential, random and no backoff, 16 (left) and 32 (right) threads. ϵ =2 %, settings used are presented in Table 4.3. . 48

A.1 Heat map of Leashed-SGD with persistence bound 1, time to reach 5 %-convergence (left). Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. For 2 and 4 threads. II
A.2 Heat map of Leashed-SGD with persistence bound 1, time to reach 5 %-convergence (left). Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. For 8, 16 and 32 threads. III
A.3 Heat map of Leashed-SGD with persistence bound 0, time to reach 5 %-convergence (left). Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. For 2 and 4 threads. IV
A.4 Heat map of Leashed-SGD with persistence bound 0, time to reach 5 %-convergence (left). Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. For 8, 16 and 32 threads. V

List of Tables

2.1 Notations used 5
2.2 Overview of the most important metrics 7
2.3 Some of the most commonly used updating rules, table adopted from [3] 9
2.4 A brief overview of notation and terminology used regarding concurrent operations. 12

3.1 Overview of the dimensions analyzed 21
3.2 Progress guarantees and consistency for the algorithms used in testing, table adopted from [1] 22

4.1 MLP architecture used, with input of 784 and d = 134 794 28
4.2 Overview of experiments 28
4.3 Parameter settings for each algorithm that reached 5 % of initial error the fastest with a maximum of 10 % crashed executions, based on 10 independent runs of all algorithms with step size, $\eta \in [0.001, 0.009]$ and batch size, $B \in \{16, 32, 64, 128, 256, 512\}$. Fastest converging setting reported as Step size and Batch size derived from figures 4.5-4.12, A.1-A.4. The #Parameter combinations indicates the number of settings reaching 5 %-convergence within 10 % of the time fastest execution with a maximum of 10 % crashed executions. Crashes represent the total number of crashed executions over the whole test range (540 executions). 41

1
Introduction

Stochastic Gradient Descent (SGD) is a numerical optimization algorithm widely used for common optimization problems in data analytics and Machine Learning (ML), especially for large datasets using *Artificial Neural Networks* (ANN). Today ANNs can solve complex tasks and be found in applications such as language translation, speech recognition, object recognition, face recognition and more [4]. ANNs originate from the work of W. McCulloch and W. Pitts, [5], when they in 1943 proposed one of the first models on how a neuron might work. Promising early success such as the perceptron learning simulation by [6] in 1960 resulted in a debate about the capabilities of ANNs. It was concluded that a simple XOR problem would not be solvable with ANNs [7, 8]. The progress within the field stagnated as the research decreased. In the mid-80s, ANNs started to regain attention again with the US-Japan Joint Conference on Cooperative/Competitive Neural Networks, the American Institute of Physics started Neural Networks for Computing, and the Institute of Electrical and Electronic Engineer's had their first International Conference on Neural Networks. The growing computational capabilities of computers and the proposal of backpropagation for gradient calculation made practical training of multi-layer ANNs possible.

ANNs build on the concept of biological neurons, where the model of a neuron is called a perceptron. Several perceptions can be used in connected layers, forming an ANN that can be trained on different ML tasks. A perceptron consists of a weight, a bias, and a non-linear activation function. The network will produce an output for a given input, and this output can be compared to an expected output through a loss function. From this point, the training process becomes a numerical optimization problem where the sets of weights and biases producing the lowest error are the target. Solving this optimization problem can be done using SGD.

Larger datasets can generally allow for more complex tasks and better generalizing capabilities of the model. However, training ANN models can be time-consuming, especially for large datasets. Generally, different parameters, architecture, and algorithms for the model need to be tested before a satisfactory result is reached. Therefore, it is desirable to speed up training, both to reduce training time and in order to use larger datasets. The time to reach a satisfactory result is referred to as convergence time.

There are several ways in which this can be done, such as transferred learning (e.g., start the training with pre-trained base parameters for the model), train several models with different settings concurrently, and make use of parallelism for each training phase. This thesis will focus on parallelism within each model training. However, all of these techniques can be used together in a complete development

process.
Parallelism within each model training can further be divided into four main categories:

1. Training several models concurrently that then is used as an ensemble of models forming one model
2. Training several models concurrently that are aggregated into one model
3. Make use of data parallelism within the model by parallelizing computations in each iterative update of the model
4. Parallelize the iterative update process of the model

This thesis will focus on the fourth way of parallelizing the training phase, and more specifically for a shared memory system using the SGD as the optimization algorithm. SGD takes a step in the negative gradient direction for each iteration. The stochasticity comes from randomly sampling a subset of the data for the gradient calculation. Each update thus builds on the previous state/position. SGD has two main parameters that need to be selected before the optimization process, *step size* and *batch size*. The step size is a constant that scales the size of the step taken in each iteration. The batch size is the size of the subset sampled from the data used to calculate the gradient.
The intuitive way of parallelizing any iterative process is to process multiple instances at each iteration and then average their result at the end of each iteration. This process is in the context of parallel SGD, known as Synchronous Parallel SGD (SyncSGD). The other approach is to omit the averaging and let each worker update their result when finished with its calculations. This approach is known as Asynchronous Parallel SGD (AsyncSGD).
Averaging the results between each iteration implies synchronization between each iteration, limiting performance in parallel systems. Research has under certain assumptions shown theoretical performance benefits of AsyncSGD, and empirical tests have also shown performance benefits of AsyncSGD over SyncSGD [2, 9, 1].

1.1 Statement of the problem

SyncSGD in its simplest form entails limited scalability [10, 11]. This limitation comes from the waiting time introduced when different workers calculate gradients at different speeds. AsyncSGD does not have this problem since there is less synchronization between gradient calculations. However, this reduction of synchronization is also the origin of the challenges with AsyncSGD.
Gradient calculations can be made on stale (old) parameters with AsyncSGD. Calculations based on old parameters will introduce noise into the model, increasing the total number of steps required until convergence. The stale gradient calculations originate from the introduction of two or more workers. When two or more workers concurrently read and iteratively updates the shared state, one or more updates from other workers may be applied during the gradient computation, causing the first worker to have an old view of the shared state. Some proposed methods for reducing the effect of gradient calculations on stale parameters are adaptive step size, delay compensating terms, variance reduction [11, 12, 13, 14]. All of these methods are in some way trying to capture the staleness in the mathematical model, e.g.,

changing the updating rule in order to reduce the number of updates required to reach a certain level of precision.
Another dimension that can be analyzed is how the access to the shared state is coordinated, i.e., *fine-grained synchronization*. The shared state is the current state of the model, e.g., the value of all weights and biases of the model. When multiple workers are introduced in AsyncSGD, the access to the shared state needs to be coordinated, which is referred to as fine-grained synchronization. With more workers, the contention for the shared state increases, and depending on the fine-grained synchronization method, this will have different effects on training.
Hyperparameters such as batch size and step size have a significant impact on training time. For AsyncSGD, the impact of different hyperparameters is related to the level of parallelism and the fine-grained synchronization used. In current literature, within the field of AsyncSGD, this is something that often is overlooked. Most papers in the field use a baseline model for hyperparameter tuning and then use these settings across multiple algorithms and levels of parallelism. This thesis will study how the selection of batch size and step size affect AsyncSGD with different fine-grained synchronization methods.

1.2 Purpose of the study and hypotheses

The goal of this study is to deepen the understanding of AsyncSGD. This is done by analyzing how convergence is affected by different fine-grained synchronization schemes used, i.e., mutex locks, read-write locks, lock-free implementations (HOGWILD! and *Leashed-SGD*) [9, 2, 1], various backoff schemes, and how they relate to the mini-batch size and step size.
More specifically, the effects of fine-grained synchronization are analyzed and compared in scalability and convergence. How the underlying hardware, e.g., memory access, affects convergence and staleness distribution for ANNs are also analyzed. The use of read-write locks as a fine-grained synchronization scheme is proposed together with three backoff schemes for *Leashed-SGD* and the lock-based AsyncSGD. These sub-questions are addressed in this together with how they relate to the synchronization method used:

i How does the memory model affect convergence and staleness?
ii How does the selection of mini-batch size and step size affect convergence?
iii How does the selection of mini-batch size and step size affect staleness?
iv Can a backoff scheme help reduce contention for the shared state and thus improve convergence or reduce staleness?

1.3 Limitations and Delimitations

This thesis focus on asynchronous parallel SGD. The motivation for asynchrony is presented in Section 2. The focus is on shared memory systems and is limited to the available systems at Chalmers.

2
Theory and Background

The following chapter introduces the relevant concepts, theory, and background. It starts with describing the ordinary (sequential) stochastic gradient descent and its use in deep learning. Furthermore, the chapter also describes how SGD can be parallelized and what challenges parallelization introduces. In Table 2.1 a brief explanation of the notations used in this thesis is presented.

Table 2.1: Notations used

Notation	Meaning
f	Non-negative target/error/loss function
θ	Vector of learnable parameters
η	Learning rate
ϵ	Precision indicator for convergence criteria
$O^{(l)}$	Output vector from layer l of an ANN
σ	Non-linear activation function
N	Number of hidden layers in a DNN
v_τ	The stale view of the state θ
τ	Staleness
D	The dataset
d	The dimension of the learnable parameters, $\|\theta\|$
t	SGD iteration
B	Mini-batch size
m	Number of concurrent threads
w	Weight
b	Bias

2.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is an iterative numerical optimization algorithm that repeatedly takes a step in the negative gradient direction of a function $f(\theta)$. The function f often represents the error of the model, and the goal is to minimize f, as illustrated in Equation 2.1. θ represents the learnable metrics of the model. The function f are often based on the maximum likelihood for a set of independent observations that are summed up [15], see Equation 2.2. $f_B(\theta)$ represents a subset

of the data, known as a *mini-batch*.

$$\underset{\theta}{\text{minimize}} \; f_D(\theta) \tag{2.1}$$

$$f_D(\theta) = \frac{1}{N} \sum_{B=1}^{N} f_B(\theta) \tag{2.2}$$

Solutions to Equation 2.1 may be found using SGD, defined as Equation 2.3 with mini-batches, B, sampled randomly from the dataset. The parameter, $\eta > 0$ is known as the *step size* and $t \in \mathbb{Z}^+$, indexes the iteration/step of the algorithm. The updating of θ, Equation 2.3, is repeated until an acceptable tolerance is reached, $f(\theta) < \epsilon$, this is referred to as ϵ-convergence. A single update of, θ, according to Equation 2.3 are referred to as one SGD iteration.

$$\theta^{(t+1)} = \theta^t - \eta \nabla f_B(\theta^t) \tag{2.3}$$

Figure 2.1 illustrates SGD optimization over a two dimensional target function, $f_D(\theta) = f_D(w_1, w_2)$. The SGD steps are represented by the red line in the figure, where each step is indicated by a point.

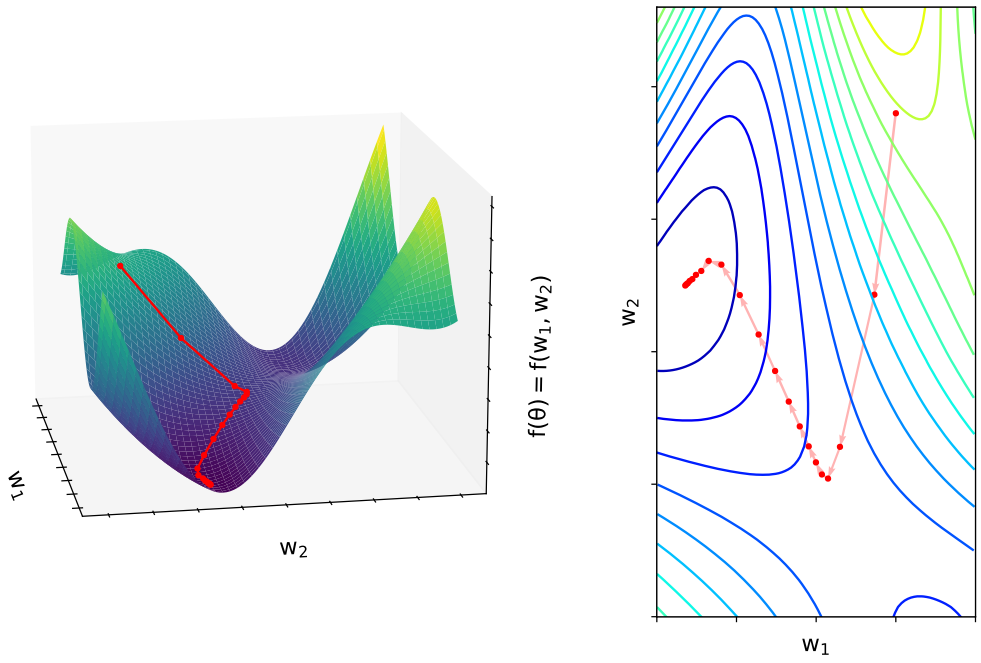

Figure 2.1: Illustration of a Stochastic Gradient Descent iterative optimization algorithm for a two-dimensional target function. SGD repeatedly takes a step in the negative gradient direction of the target function

SGD originates from gradient descent, which uses the whole dataset for gradient calculations instead of mini-batches. The random sampling of mini-batches used in SGD has the benefit of making each iteration execute faster. The random sampling of mini-batches is especially beneficial when the dataset is large so that fitting the

whole dataset in memory is not possible. Another benefit from sampling mini-batches randomly is that SGD thereby have a chance of handling some non-convex target functions. SGD can be used for various numerical optimization problems. One problem where SGD has proven particularly useful is in Artificial Neural Networks (ANN) training, especially with large datasets.

2.1.1 Metrics of interest

In order to evaluate the performance, we need to define what metrics that are of interest. Intuitively for any iterative optimization algorithm, it is of interest to know the number of steps required to reach a certain accuracy, ϵ-convergence. It is also of interest to know the total wall-clock time to reach that accuracy, given that the algorithm converges.
The convergence rate describes the rate that the computational error is approaching 0 as the number of iterations is approaching infinity. The *big-O* notation describes at what asymptotic rate a numerical method converges.
The most relevant metric to consider when evaluating the SGD is the overall *convergence* rate, i.e. the wall-clock time until ϵ-convergence. The convergence rate can be decomposed as the product of *statistical efficiency* and *computational efficiency*, as suggested by [1, 16] where statistical and computational efficiency can be defined as follows.

(i) *Statistical efficiency* is the number of SGD iterations required until ϵ-convergence.
(ii) *Computational efficiency* is the number of SGD iterations per time unit

By decomposing the convergence rate into two products and measure both overall wall-clock time and at least one of the two, statistical efficiency or computational efficiency, it is possible to identify from where the change in performance comes.

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

Table 2.2 shows an overview of the most important metrics used when evaluating different algorithms in this work.

Table 2.2: Overview of the most important metrics

Notation	**Meaning**
Convergence rate	Time to reach ϵ-convergence, i.e. wall-clock time until $f(\theta) \leq \epsilon$
Statistical efficiency	Number of SGD iterations required until ϵ-convergence
Computational efficiency	Number of SGD iterations per time unit
Stability	Percentage of successful executions

2.1.2 Parameters and hyperparameters

Hyperparameters are assigned a value before the optimization process, such as mini-batch size, learning rate, or starting values for θ, e.g., θ^0. The selection of these parameters can affect the learning process, e.g., convergence rate and stability. Stability in this context is defined as the ratio between successful and failed training processes.
The starting point, θ^0, is usually selected at random. Since the starting point affect the convergence it might be necessary to run the algorithm several times to reach a sufficient ϵ-convergence [15].
The size of the mini-batch is another parameter that affects the convergence rate. A larger mini-batch makes for a better gradient approximation and can use inherent concurrency when evaluating the gradient. However, increasing the mini-batch size also increases memory consumption if all observations in the mini-batch are to be processed concurrently. Selecting a mini-batch too small will not utilize the inherent concurrency, and choosing it too large can be problematic in three ways. The first is the available memory; the model needs to fit in the memory during training. The second is generalization; a larger mini-batch reduces the stochasticity, which, depending on the dataset and optimization task, can reduce generalization capabilities of the model [17]. The third is computation time; a larger batch size increases the computation time for each gradient compared to a smaller batch size. The reduced stochasticity with larger batch sizes can also increase convergence time, depending on the dataset and optimization task [18, 19]. When it comes to generalization, techniques like dropout can be used to increase generalization.
The selection of mini-batch size depends on the underlying hardware, and for multicore systems selecting a mini-batch smaller than some minimum value will not reduce computation time. Thus the statistical efficiency is increased, and the computational efficiency is constant up to some value of the mini-batch size. The optimal mini-batch size depends on the application, and finding the optimal mini-batch size can be challenging and time-consuming. Figure 2.2 adopted from T. Ben-Nun and T. Hoefler, [3], shows the performance and accuracy of SGD with respect to the mini-batch size for a fixed number of steps. The figure illustrates three regions with region B as the desired one. From this figure, we can observe that a small batch size, e.g., region A, results in lower performance and higher validation error. Going from region A to region B, we can observe a reduced slope for the performance curve, indicating that the most performance gain is in the lower range of the batch size. We can also see that a large batch size, e.g., region C, results in degrading accuracy, e.g., increased validation error.

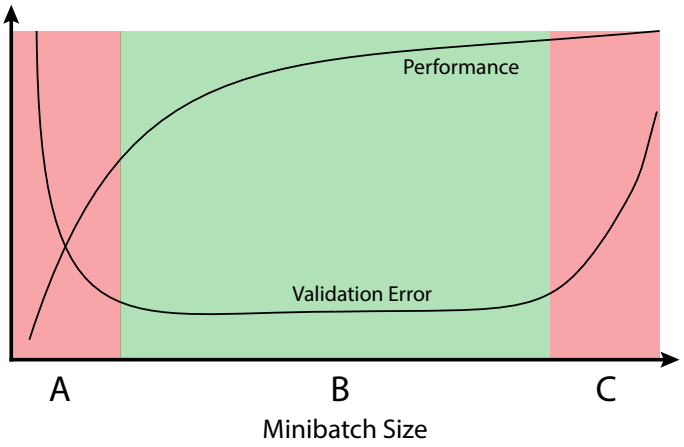

Figure 2.2: Performance and Accuracy of SGD with respect to mini-batch size for a fixed number of steps (illustration), figure adopted from [3]

The rule used to update θ also affects both the statistical and computational efficiency. As presented in Section 2.1, the regular update rule, Equation 2.3, referred to as learning rate in Table 2.3 are the simplest form of update rule. There are several other versions of this update rule that try to increase statistical efficiency. Table 2.3 shows an overview of the most commonly used rules. A more complex updating rule can increase the statistical efficiency, but this often comes at the price of lowering the computational efficiency. How the updating rule performs depends on the dataset and optimization task.

The choice of step size, η, affects the convergence rate. A larger step size can make the algorithm converge faster by taking longer steps. However, if the learning rate is selected too large, the algorithm can become unstable and diverge. On the other hand, a small learning rate requires more steps to reach ϵ-convergence. There is also an increased risk that the algorithm gets stuck in a local minimum and never reaches ϵ-convergence with a smaller step size. The choice of updating rule can reduce these risks, but the step size still affects the overall convergence.

Table 2.3: Some of the most commonly used updating rules, table adopted from [3]

Method	**Formula**	**Definitions**
Learning rate	$\theta^{(t+1)} = \theta^t - \eta\nabla(f_B\theta^t)$	
Adaptive Learning Rate	$\theta^{(t+1)} = \theta^t - \eta_t\nabla(f_B\theta^t)$	
Momentum	$\theta^{(t+1)} = \theta^t + \mu(\theta^{(t)} - \theta^{(t-1)}) - \eta\nabla(f_B\theta^t)$	
Nesterov Momentum	$\theta^{(t+1)} = \theta^t + v_\tau$	$v_{t+1} = \mu v_t - \eta\nabla f_B(\theta^t - \mu v_t)$
AdaGrad	$\theta_i^{(t+1)} = \theta_i^t - \frac{\eta\nabla\theta_i^t}{\sqrt{A_{i,t}+\epsilon}}$	$A_{i,t} = \sum_{\tau=0}^{t}(\nabla f_B(\theta_i^t))^2$
RMSProp	$\theta_i^{(t+1)} = \theta_i^t - \frac{\eta\nabla\theta_i^t}{\sqrt{A'_{i,t}+\epsilon}}$	$A'_{i,t} = \beta A'_{t-1} + (1-\beta)(\nabla f_B(\theta_i^t)^2)$
Adam	$\theta_i^{(t+1)} = \theta_i^t - \frac{\eta M_{i,t}^{(1)}}{\sqrt{M_{i,t}^{(2)}+\epsilon}}$	$M_{i,t}^{(m)} = \frac{\beta_m M_{i,t-1}^m + (1-\beta_m)(\nabla f_B(\theta_i^{(t)}))^m}{1-\beta_m^t}$

2.1.3 Artificial Neural Networks

ANN consists of multiple artificial neurons, called perceptrons, inspired by the biological neurons in our brain. A deep artificial neural network consists of several interconnected layers of perceptrons, as shown in Figure 2.3. Each perceptron is associated with a bias, and one or more input connections are associated with a weight, w. Each layer can be parameterized by a weight matrix and a bias vector. Together all weights and biases are represented by the vector θ. The numeric values of the parameters in θ are learned through Equation 2.3. There are several types of architects for ANNs where *Multi-Layer Perceptron* (MLP) and *Convolutional Neural Network* (CNN) are among the most commonly used.

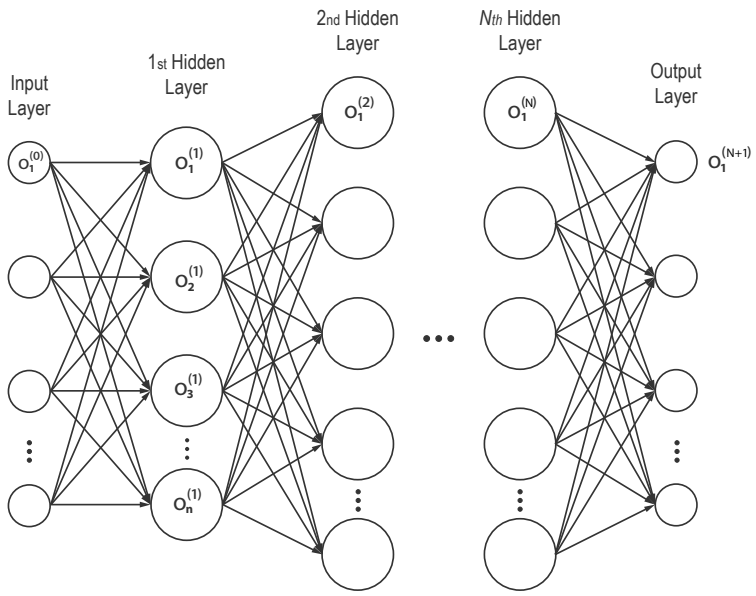

Figure 2.3: Example of a feedforword DNN architecture with N hidden layers and n input-layers. The output of each layer is the input of the next layer as illustrated

MLP consists of densely connected layers, i.e., each perceptron directly connects to every perceptron in the next layer. The incoming value (the previous layer's output) is multiplied with a corresponding weight at each perceptron. The incoming values are summed up and, then the bias is subtracted. After that, the aggregated value is run through a non-linear activation function to produce an output to the next layer, see Equation 2.4. σ is referred to as the activation function. A commonly used activation function is ReLU ($\sigma_{ReLU} = max(0, x)$).

$$o^{(l)} = \sigma(\sum_{i=1}^{n} w_i \cdot o_i^{(l-1)} - b) \tag{2.4}$$

CNN does not have densely connected layers but, instead, CNN uses a sparse network architecture as illustrated in Figure 2.4. CNN consists of layers that convolve the input with learnable filters. The general structure of a CNN can be divided into four main sections as shown below and in Figure 2.4.

(i) Convolution

(ii) Pooling
(iii) Flattening
(iv) Full Connection

CNN is commonly used in classification tasks for image recognition. CNN has fewer learnable parameters compared to MLP.

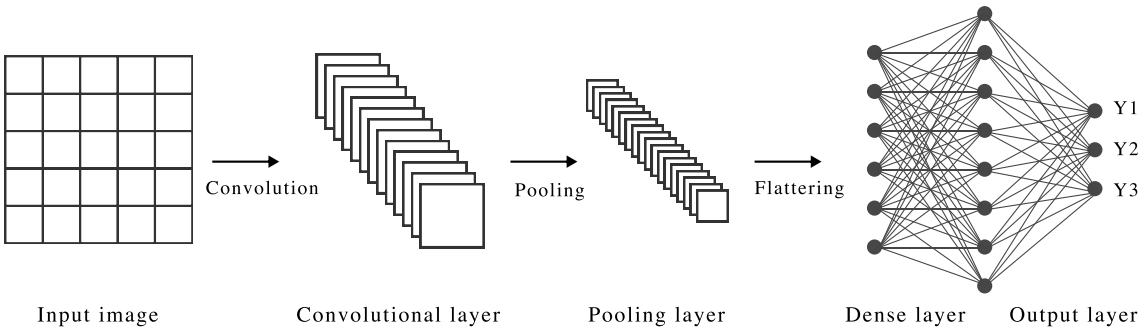

Figure 2.4: Example of a CNN architecture

2.1.4 The training process

The training process is similar for both MLP and CNN and can be divided into four main steps that are repeated until ϵ-convergence is reached:

(i) Select a mini-batch at random from the dataset
(ii) Pass the selected mini-batch trough the network (forward pass)
(iii) Calculate the output error and use backpropagation to estimate the gradient $\nabla f_B(\theta)$ (backward pass)
(iv) Use the update rule, Equation 2.3, to update the learnable parameters of the network, e.g. the weights and biases

When the whole dataset has been passed through the network, it is called an epoch. In practice, it usually takes multiple epochs to reach ϵ-convergence. It is also common to shuffle the data between each epoch and then select the mini-batches sequentially [20] instead of selecting the mini-batches at random each iteration.

2.2 Parallel Stochastic Gradient Descent

The increasing amount of available data and growing demand for data analysis have led to an increased demand for more efficient systems that can utilize modern many-core processing architecture and larger distributed systems. Parallel algorithms can intuitively use such systems more efficiently. Therefore, parallelization of the SGD has been given an increased amount of attention lately. Although gradients calculated in parallel will increase the throughput of calculated gradients this, does not necessarily imply improvements of the total execution time to reach ϵ-convergence. This is because of the inherent sequential nature of the SGD; each update is dependent on its previous value. This makes parallel gradient calculations non-trivial, requiring synchronization after each iteration to not break the original

SGD algorithm's semantics. This method is referred to as *Synchronous Parallel SGD (SyncSGD)*. Synchronization is costly and tends to limit the scalability of a parallel system; therefore, it is of interest to investigate how lowering the requirement for synchronization affects parallel SGD. This approach is known as *Asynchronous Parallel SGD (AsyncSGD)*. AsyncSGD does not require synchronization after each gradient calculation, and it has been shown, under certain conditions, that AsyncSGD converges faster and scales better than SyncSGD [2, 11].
The challenge with SyncSGD originates from synchronization. Threads that are slower with their gradient calculation will delay all other threads. These slower threads are referred to as stragglers, and the impact of stragglers can significantly affect the convergence time of SyncSGD.
AsyncSGD, on the other hand, reduces the negative effects of stragglers by not requiring synchronization at each iteration. However, relaxing the synchronization requirement after each iteration changes the semantics of the algorithm, it is no longer guaranteed that each update is based on the same view of the state, θ.
By not synchronizing after each gradient calculation, there is a risk of gradient calculations being done on stale views of the state, θ. There have been theoretical and empirical studies on how gradient calculations on stale views of the state affect the convergence and efficiency, together with proposals on how to reduce the effect of gradient calculations on stale parameters, [12, 9, 21, 11]. In Table 2.4 an overview of the notation and terminology used regarding concurrent operations can be seen.

Table 2.4: A brief overview of notation and terminology used regarding concurrent operations.

Notation	**Meaning**
Starvation-freedom	Every thread wanting to enter a critical section eventually succeed [22], implies deadlock freedom.
Lock-based synchronization	Blocking progress condition. A lock is used to protect access to limited resources.
Lock-freedom	At least one thread is making progress at all time, starvation is allowed [22].
Wait-freedom	All threads is making progress at all time, starvation is not allowed
Consistent	Read operations return a consistent snapshot, [1], e.g. a read operation can not return a partially updated state.

2.2.1 Parallel computer architecture and parallelization

The underlying hardware system used for training affects how the parallelization can be done. The system used for training can roughly be categorized into single-machine and multi-machine. The single-machine system often uses shared memory,

while the multi-machine system often uses distributed memory [3].
Parallelization of the SGD can be done both on shared memory systems and distributed systems. This thesis focus on single-machine shared memory systems, but many general concepts can be transferred to distributed systems. A shared memory system has a global shared memory that stores the data of an application and can be accessed by all processors or cores of the hardware systems [23]. Communication between threads is done through shared memory, where one thread writes to a shared variable, and another reads that variable [23]. Access to shared data needs to be coordinated by synchronization between threads [23].
Multi-core architectures are shared memory systems where there are multiple processor cores on the same computer chip. Typically each processor has a private L1 cache and a shared L2 or L3 cache where processors can communicate [22].
Multi-threaded architecture allows a single processor to execute two or more threads concurrently [22]. Modern computers combine multi-core with multi-threading, where there can be multiple individual multi-threaded cores on the same chip [22].
Shared memory systems can further be divided into:

1. In *Uniform Memory Access* (UMA) all identical processors have equal access time (latency) and access speed.
2. In *Non Uniform Memory Access* (NUMA), the processors are divided into clusters where each cluster has its own local memory. Accessing memory from outside the cluster requires going through an interconnection resulting in different accessing times depending on where the data is located in memory relative to the CPU requesting the data.

Figure 2.5 shows an example of UMA and NUMA architectures.

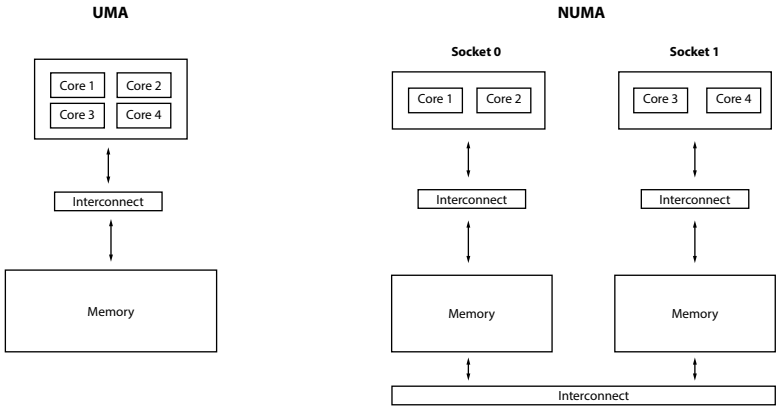

Figure 2.5: Example of a UMA(left) and NUMA(right) architecture

2.2.2 Synchronous parallel SGD

Synchronous parallel SGD (SyncSGD) use the concept of regular sequential SGD. However, at each iteration, multiple threads or nodes read the state, θ, select a random sample from the dataset and calculate the gradients, $\nabla f_B(\theta^t)$, locally. The

threads then synchronize by averaging the resulting local gradients and then updates the global state, θ, according to Equation 2.3. In [11] K. Bäckström *et.al.* argues that SyncSGD in its original version is statistically equivalent to sequential SGD with larger mini-batch size. With this view, SyncSGD does not break the semantics of the SGD, and most of the theoretical convergence guarantees and empirical results for SGD hold even for SyncSGD.
As argued in Section 2.2, synchronization after each iteration impose challenges for scalability. The challenge in scalability comes mainly because each iteration is limited to the slowest thread, stragglers. The presence of stragglers becomes a bottleneck for SyncSGD [3].
Stale-Synchronous Parallelism (SSP) can be used to reduce the impact of stragglers. SSP relaxes the semantics of SGD by allowing asynchronous updates and only synchronizes the threads after a maximum number of steps have been performed [3]. Stragglers are more prominent in heterogeneous systems making this approach work especially well for these systems [3].
Even though SSP reduces the problems with stragglers slowing down the progress, halting threads will still cause the whole system to halt indefinitely in the synchronization phase [24]. The *n-softsync* method, proposed by [25], partially addresses this issue by allowing for updates with only n threads contributing to the update, thus relaxing the synchronization.

2.2.3 Asynchronous parallel SGD

The Asynchronous approach removes the coarse-grained synchronization, e.g., the gradient averaging between steps, allowing for asynchronous updates of the shared state. Asynchronous SGD implies that while an update is being calculated by one thread, multiple concurrent updates are calculated by other threads. As mentioned in Section 2.2, this changes the semantics of the algorithm, and there have been several works dedicated to understanding the effect of asynchrony [12, 9, 21, 11]. The stale view of the state can be represented as $v_t = \theta^{t-\tau}$, where τ is the staleness; this gives us the following update rule.

$$\theta^{(t+1)} = \theta^t - \eta \nabla f_B(v_t) \tag{2.5}$$

Asynchronous reads and updates of the global state, θ, enable better computational efficiency with a higher degree of parallelism. However, because of the shared global state, θ, the contention of shared memory access increases with increased parallelism. At a certain point, the computational efficiency no longer increases with the increasing number of threads [1]. Increased parallelism also increases the gradient updates based on stale parameters, and therefore reduces the statistical efficiency.
In [9], S. Chaturapruek et al. investigated the effects of gradient calculations on stale parameters. S. Chaturapruek et al. showed, under assumptions such as convexity, that the noise introduced by gradient calculations on stale parameters are asymptotically negligible compared to the noise introduced by the stochasticity [9]. I. Mitliagkas et al., [21], further extended the understanding of AsyncSGD by relaxing the assumptions made in [9]. In [21], I. Mitliagkas et al. show that running SGD asynchronously can be viewed as adding a momentum term to the SGD updating

rule. There is no assumption on the convexity of the target function in their work, i.e., it is applicable to deep learning. However, there are assumptions on bounded staleness and number of threads [21].
In [25], W. Zhang et al. investigate the effects of learning rate and staleness, proposing a version of AsyncSGD with staleness adaptive step size for n-softsync protocol. This work is extended by K. Bäckström et al. in [11], by proposing a new distribution model that better captures the staleness.
As mentioned above, there are two different synchronization levels; first, there is the algorithmic level, e.g., asynchronous or synchronous, which can be referred to as *coarse-grained* synchronization. Then, there is the thread coordination for AsyncSGD, which can be referred to as *fine-grained* synchronization. AsyncSGD can be implemented in several different ways, both when it comes to coarse-grained and fine-grained synchronization.
Coarse-grained synchronization ranges from synchronous parallel SGD to fully asynchronous parallel SGD with intermediate methods such as SSP and n-softsync. Fine-grained synchronization focuses on thread coordination, e.g., how shared resources are accessed.
Different updating rules have different properties. These properties can indirectly or directly affect synchronization and convergence. For instance, can updating rules focus on statistical efficiency, computational efficiency or reduce the adverse effects of staleness.

Fine-grained synchronization and synchronization primitives

In each SGD step (each iteration of Equation 2.5), the shared state, θ, is first read and then updated, resulting in two critical sections. The first when reading the shared state and the second when updating the shared state. Fine-grained synchronization is used to coordinate the access of these critical sections. The goal of the fine-grained synchronization is to increase scalability by reducing contention at the critical sections. This is done with different synchronization primitives for access to the shared state, θ. One of the simplest forms of synchronization primitives is blocking synchronization.
Blocking synchronization uses some locking mechanism to prevent multiple threads from accessing the same limited recourse simultaneously. Blocking synchronization implies that any thread's delay can delay other threads [22]. In practice, this implies that a thread put to sleep by the OS-scheduler can block the progress of the program.
A mutually exclusive (mutex) lock can be used to protect a critical section by only allowing for one thread in a critical section at a time. A thread is only allowed to enter the critical section when holding the lock, making updates to the shared state perfectly safe.
Since there are no modifications to the shared state, θ, in the first critical section, we can allow multiple reads concurrently, e.g., there is no need for mutual exclusion on that critical section. A *read-write lock* (rw-lock) allows for multiple concurrent reads, and an exclusive write lock [26]. Readers need to lock out other writers and readers since a write modifies the object [22]. rw-locks locks can be designed differently, prioritizing readers, the writers, or unspecified priority. What priority that is specified leads to different trade-offs with regards to progress guarantees.

If priority is given to the readers, this allows for maximum concurrency but can lead to starvation if contention is high. The priority can be either strong or weak. Whenever a writer releases the lock for a strong priority, any blocking reader will always acquire it. For a weak priority, multiple readers may hold the lock, and even though a writer is waiting for the lock to be released, new readers can acquire the lock. If priority is given to the writers, the readers can no longer acquire the lock if a writer is waiting for the lock. The problem with starvation for the writers is thereby addressed but with a trade-off in concurrency.

In systems with contention, backoff can be used to reduce contention. The contention is reduced by backing off instead of persistently try to access the contended state. For example, if a thread tries to acquire a lock that is not available in the case of a lock-based system, it will back off before trying to acquire the lock again. Some of the most commonly used backoff schemes are linear, exponential, and random backoff. For linear and exponential backoff, the initial backoff is selected at random. For each failed attempt to acquire the lock, the backoff time is increased linearly or exponentially up to a maximum backoff time. For random backoff, the backoff time is just selected randomly for each thread.

Another type of synchronization is non-blocking synchronization. Non-blocking synchronization does not make use of locks to synchronize threads accessing limited resources. There are three main types of non-blocking synchronization. The first and weakest progress guarantee is the *obstruction-freedom*. A method is said to be obstruction-free if, executed in isolation, it finishes in a finite number of steps [22]. The obstruction freedom ensures that not all threads can be blocked by a sudden delay of one or more threads. The *Lock-Free* progress condition implies that at least one thread is making progress at all times, e.g. system progress. A method is said to be lock-free if it guarantees that infinitely often, some method call finishes in a finite number of steps [22]. Lock-freedom thus implies obstruction-freedom, but obstruction-freedom does not guarantee lock-freedom. *Wait-free* progress guarantee has the strongest progress condition and guarantees that every thread that takes steps makes progress. More formally, a method is wait-free if each method call finishes in a finite number of steps, independently of how its execution is interleaved with steps of other concurrent method calls [22].

HOGWILD!, [2], is one of the most widely-used AsyncSGD algorithms for shared memory, multi-core machines [27]. HOGWILD! is a lock-free implementation of AsyncSGD, which is achieved by letting threads component-wise atomic update the shared global state θ without any locks, proposed by F. Niu et. el. [2]. The HOGWILD! algorithm has shown to be efficient in training ML models, especially when gradient updates are sparse [28]. For sparse gradient updates, HOGWILD! has shown a near-linear speedup in convergence [2]. However, when relaxing the sparsity assumption the convergence bound increases with $\sqrt{d}$ [29]. In [1], K. Bäckström et al. proposed a consistent lock-free implementation of the AsyncSGD aimed to keep the benefits of lock freedom while keeping the consistency, especially for higher dimension problems. Consistency in this context refers to read operations returning a consistent snapshot of the shared state θ, e.g., a read operation can not return a partially updated θ as in an inconsistent algorithm. With the *Leashed-SGD* framework, they showed promising results with 20 % up to four times improvements in

convergence time compared to regular AsyncSGD, and HOGWILD! [1].
The data access for AsyncSGD and HOGWILD! is similar to each other, while *Leashed-SGD* uses a different structure where the read of the shared state and update to the shared state have been separated as illustrated in Figure 2.6.

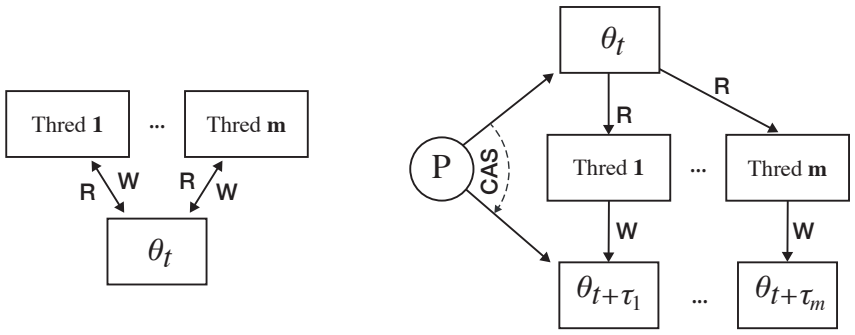

Figure 2.6: Data access for AsyncSGD and HOGWILD! (left) and *Leashed-SGD* (right). AsyncSGD uses locks to ensure mutual exclusion to the shared state, HOGWILD! uses component-wise atomic read and write of the shared state. In Leashed-SGD each thread only access θ_t through a read operation. The update are then calculated locally and stored at a new memory location that becomes a candidate for $\theta_{t+\tau}$, figure adopted form [1].

Leashed-SGD uses a *compare and swap* (CAS) operation in a retry-loop for updating the global state. A CAS operation is a type of *read modify write* instruction and is guaranteed to be atomic. It compares the content of a memory location with a given value and only modifies the value of that memory location if they are the same. In Leashed-SGD a persistence bound is introduced to the CAS retry-loop, indicating a maximum number of CAS retries for each update.

2.2.4 A note on hyperparameters and parallel SGD

In Section 2.1.2 the training process and hyperparameters are discussed for sequential SGD. Here the influence of parallelization will be taken into account. Since SyncSGD can be viewed as sequential SGD with larger batch sizes [1, 28], the focus will be on AsyncSGD.
S. Chaturapruek et al., [9], investigate L1, L2 and L3 cache misses for batch size $B \in \{1, 10\}$ for dense linear regression problems. They show that B=10 reduces cache incoherency, resulting in halved epoch time for ten threads compared to B=1 [9].
Most papers in the field of AsyncSGD use a baseline model for hyperparameter tuning. Since the selection of hyperparameters highly affects the convergence of the algorithms, it is essential to understand how different settings affect the different types of algorithms. This work aims to increase the understanding of how batch sizes and step sizes for different fine-grained synchronization schemes affect convergence and staleness by empirically analyzing various settings for the batch and step size.
Memory access can also affect convergence and scalability, as discussed in Section 2.2.1. The research on this topic is, to the best of my knowledge, limited. HOG-

WILD! suffer from performance degradation when NUMA increases communication time, according to H. Zhang et al., [27]. An important note is that their results are based on linear SVM tasks and not ANN. Therefore, the optimization space is different, where ANNs generally have more parameters than SVM, and each SGD step will be more computational intensive for ANNs.

2.3 Problems and challenges

In this section, the problems and challenges of parallel SGD will be highlighted. The section is divided into three main parts, starting with convergence and asynchrony, then moving over to scalability, and lastly, testing and benchmarking.

2.3.1 Convergence and asynchrony

AsyncSGD introduces questions about statistical efficiency, e.g., how is the convergence affected by asynchrony. As mentioned in Section 2.2, asynchrony increases computational efficiency while the statistical efficiency is reduced. The trade-off between statistical and computational efficiency is highly dependent on the application. The reason for reduced statistical efficiency is the noise introduced by gradient calculations based on stale parameters. Methods such as delay compensation, variance reduction, and adaptive step size, [14, 13, 11] can reduce the effect of gradient calculation on stale parameters. With adaptive step size, [11], showed an average speedup of $\times 1.5$ compared to regular AsyncSGD. Leashed-SGD, [1] showed improvements in wall-clock time to convergence from 20 % up to four times improvement compared to AsyncSGD and HOGWILD!.

2.3.2 Scalability

The statistical efficiency is reduced by asynchrony with an increasing number of threads, as mentioned in the previous section. Different techniques such as delay compensation, variance reduction, adaptive stepsize, momentum can potentially reduce the statistical penalty for gradient calculations on stale parameters, allowing for increased scalability of AsyncSGD.
For regular sequential SGD, observations in the mini-batch can potentially be calculated concurrently, making use of the data parallelism. For parallel SGD, depending on the hardware system used, all available hardware resources might be used to run concurrent SGD iterations. Then, the system will not fully use the gained data parallelism with larger batch sizes. Another challenge for asynchronous updates is memory contention. As more threads are introduced, there is increased competition to access the shared state, θ. As described in Section 2.2.3, different synchronization primitives for fine-grained synchronization can reduce contention of the critical sections and increase scalability. K. Bäckström et. al shows in [1] that coordination of threads with the Leashed-SGD framework increases the overall convergence rate and scalability compared to uncoordinated reads and writes to θ, like in the HOGWILD! framework.

2.3.3 Testing and Benchmarking

Since the performance of different algorithms is highly dependent on the dataset, hardware, and architecture used, comparison between papers can be challenging. Another challenge with testing and benchmarking is that selection of hyperparameters might not have the same effect on every algorithm. Optimizing each algorithm with respect to hyperparameters and optimization tasks is a very time-consuming and resource-intensive task that might not be feasible. This work aims to give an increased understanding of how synchronization schemes affect the selection of hyperparameters.

3
Methods

In this chapter, methods used and evaluated in the thesis are presented, starting with the algorithms used for testing and then the dimensions in which the experiments have been performed. There are five main dimensions in which the AsyncSGD is explored in this thesis. A short summary of these a presented in Table 3.1 and in Table 4.2 corresponding overview of the results to each step can be found. The goal of this thesis is to answer the questions presented in Section 1.2. This is done through the five steps presented in Table 3.1 and 4.2.

Table 3.1: Overview of the dimensions analyzed

Step	Study description
S1	***Memory model and parallel SGD.*** Here the impact of UMA and NUMA are explored in terms of convergence time. Further more are the statistical and computational efficiency analyzed together with the staleness distribution to get full view of how the memory model affect convergence.
S2	***Convergence and hyperparameter selection.*** Here the effects of hyperparameter selection are analyzed in terms of convergence time and stability. This is done for different fine-grained synchronization schemes and level of parallelism.
S3	***Staleness and batch size.*** Here the staleness distribution are analyzed for various levels of parallelism and batch size
S4	***Comparison over tuned settings.*** The algorithms are compared using hyperparameter settings resulting in the fastest convergence time for each individual algorithm and level of parallelism while still maintaining a stability of 90 %.
S5	***Backoff.*** Three backoff schemes are introduced for the lock-based algorithms and *Leashed-SGD*. These are tested in terms of convergence rate using the settings obtained from **S2**.

3.1 Algorithms and fine-grained synchronization

In order to understand how step size and batch size relate to fine-grained synchronization, three different algorithms were selected for comparison, with each algorithm with different fine-grained synchronization. The fine-grained synchronization methods can be evaluated from their progress guarantees or in terms of consistency, Table 2.4. Table 3.2 shows the algorithms used for testing and how they are categorized in terms of progress guarantees and consistency.

Table 3.2: Progress guarantees and consistency for the algorithms used in testing, table adopted from [1]

Algorithm	**Progress guarantees**	**Consistent**
AsyncSGD [9]	lock-based	yes
HOGWILD! [2]	lock-free	no
Leashed-SGD [1]	lock-free	yes

All algorithms were implemented using the *ParameterVector* data structure proposed by [1]. The data structure can be instantiated as local or shared among threads and supports reading and submitting updates to the shared state. Algorithm 1 shows an overview of the ParameterVector data structure. The data structure does not implement synchronization for protecting reads of updates. Instead, this is left for the algorithmic implementation [1].

Algorithm 1 ParameterVector datastructure, [1]

```
1:  Float[d] theta
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:     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]
```

Besides regular AsyncSGD with a mutex lock, another version was implemented using a rw-lock, resulting in four different algorithms for comparison. rw-locks allow multiple concurrent reads but only one write at a time, as described in Section 2.2.3. The hypothesis was that shared reads would reduce contention on the shared state and thereby allow for better scalability. The implementation of the rw-lock was done using the POSIX rw-lock [26]. Algorithm 2 shows an overview of the lock-based AsyncSGD algorithm using the ParameterVector structure from Algorithm 1.

Algorithm 2 Lock-based AsyncSGD, [1]

```
1:  GLOBAL ParamVector PARAM
2:  GLOBAL Float η                                          ▷ step size
3:  GLOBAL Lock shared_state                    ▷ mutex lock / rw-lock

    Initialization:
4:  PARAM ← new ParamVector
5:  PARAM.rand_init()

    Each thread:
6:  local_grad ← new ParamVector
7:  local_param ← new ParamVector
8:  repeat
9:      Fetch a random mini-batch
10:     shared_state.lock()                              ▷ lock / read lock
11:     local_param.theta = copy(PARAM.theta)            ▷ Make local copy of θ
12:     shared_state.unlock()                            ▷ release lock
13:     local_grad.theta ← comp_grad(local_param.theta)
14:     shared_state.lock()                              ▷ (exclusive) lock on θ
15:     PARAM.update(local_grad.theta, η)
16:     shared_state.unlock()                            ▷ release lock
17: until convergence
```

For HOGWILD!, the algorithm is very similar, with the difference that locks are removed and read/write to the shared state θ needs to be performed atomically. This is also why read operations of the shared state in HOGWILD! can not guarantee to return a consistent view of the state. *Leashed-SGD*, on the other hand, uses a different structure. Algorithm 3 shows an overview of *Leashed-SGD*; for a more detailed explanation of *Leashed-SGD*, see [1].

Algorithm 3 *Leashed-SGD* [1]

```
1: GOLBAL ParamVector **P                          ▷ address to latest pointer
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()
12:    until break

    Initialization:
13: init_pv ← new ParamVector()
14: init_pv.rand_init()
15: P ← &init_pv

    Thread i:
16: local_grad ← new ParamVector
17: repeat
18:    latest_param ← latest_pointer()
19:    local_grad.theta ← comp_grad(latest_param.theta)
20:    latest_param.stop_reading()
21:    new_param ← new ParamVector()
22:    Int num_tries ← 0
23:    repeat
24:        latest_param ← latest_pointer()
25:        new_param.t ← latest_param.t
26:        latest_param.stop_reading()
27:        new_param.update(local_grad.theta, η)
28:        succ ← CAS(P, latest_param, new_param)
29:        if succ then
30:            latest_param.stale_flag ← true
31:            latest_param.safe_delete()
32:        else
33:            num_tries ← num_tries + 1
34:            if num_tries > T_p then
35:                delete new_param
36:                break
37:    until succ
38: until convergence
```

3.2 Memory model and parallel SGD

How threads are scheduled on the system can have an impact on convergence and staleness. Therefore a comparison of uniform memory access (UMA) and non-uniform memory access (NUMA) was made before exploring the impact of hyper-parameters and fine-grained synchronization. Using a system with two sockets and eight cores on each socket made it possible to schedule a maximum of eight threads on one socket (e.g., UMA) or split them between the two sockets (e.g., NUMA). The convergence time, computational efficiency, staleness distribution for individual threads, and the total number of updates were measured for UMA and NUMA configuration using seven and eight threads. The reason for using seven threads was so that handling of the global state would be done without hyper-threading on the same socket as the other threads for UMA. By doing this with eight and seven threads, it is possible to see if this has any effects on convergence.
For hyperparameter selection, the initial test was done with a batch size of 64 and a step size of 0.005. In the initial experiment, the difference between UMA and NUMA is the focus. If the difference in performance between UMA and NUMA is insignificant, then the hyperparameter exploration can be done using either UMA or NUMA. On the other hand, if the memory access impacts convergence, then hyperparameter exploration will be done on both UMA and NUMA.

3.3 Convergence and hyperparameter selection

The selection of learning rate and batch size can have a high impact on convergence time and stability, as stated in Section 2.1.2. A parameter search was done for the algorithms presented in the previous section to explore what impact parallelism and fine-grained synchronization have on different hyperparameter settings. The parameter search was done over stepsize, $\eta \in [0.001, 0.009]$ and batch size $B \in [16, 512]$, for 2, 4, 8, 16 and 32 threads. The time to reach a convergence of 5 % of the initial loss, $f(\theta^0) \approx 2.3$, was selected as a convergence criterion. Each run was limited to 60 s, executions that did not manage to reach 5 %-convergence in that time are reported as diverge. The step size and batch size with the fastest convergence time for each level of parallelism was selected for comparison between the algorithms, see Table 4.3 and Figure 4.19 and 4.20.
Since the memory access had a noticeable impact on convergence, mainly from computational efficiency, the hyperparameter exploration for 2, 4, and 8 threads was done using both UMA and NUMA architecture.

3.4 Staleness and hyperparameter selection

The total staleness distribution for each algorithm, setting, and level of parallelism was also measured to see how the staleness is affected by different settings. Since the step size does not affect staleness, the focus will be on how batch size affects staleness for different synchronization schemes and level of parallelism. Increasing the batch size implies longer time to compute each update, and less updates for

each epoch. The batch size therefor have a direct impact on the contention on the shared state, and potentially the staleness. This will be tested for the different synchronization schemes presented in Section 3.1.

3.5 Backoff

The three backoff schemes mentioned in Section 2.2.3 were implemented for the lock-based versions of AsyncSGD, e.g., RW-lock and mutex lock. The backoff was implemented by using a non-blocking try-lock version of the locks in a while loop with the backoff as described by Algorithm 4. Each thread creates a pointer to a base backoff class from which the specific backoff class is created (e.g., linear, exponential, or random).

Algorithm 4 Backoff implementation for lock-based AsyncSGD

```
1: while try_lock() do
2:     backoff → start_backoff()
```

For *Leashed-SGD*, the backoff was implemented in the CAS retry-loop, e.g., between lines 38 and 39 in Algorithm 3.
When introducing a backoff schemes the contention for the shared state are potentially changed and this could affect the selection of hyperparameters and what settings that results in the fastest convergence time. Furthermore, will a backoff scheme introduce backoff specific hyperparameters such as maximum backoff, initial backoff distrubution and increments in backoff time for linear backoff. Due to limitations in time the settings obtained from the initial hyperparameter search was used when evaluating the backoff. The backoff specific parameters was initially explored, however due to limitations in time this exploration was limited.

4
Empirical study

In this chapter, a summary of the results is presented. Additional data such as the complete hyperparameter search can be found in Appendix A. The experiment setup and hardware used are presented in Section 4.1. Table 4.2 shows a summary of the experiments and where to find corresponding results.

4.1 Experiment setup

The empirical study has been performed in five main steps as stated in Section 3. This was done in order to answer the questions stated in Section 1.2.
A c++ framework implemented by K. Bäckström for shared-memory parallel SGD training, [1], was extended to include a rw-lock version of AsyncSGD and the three backoff schemes in Section 3.5 for both lock-based versions of AsyncSGD and *Leashed-SGD*. The framework builds on the MiniDNN library and relies on Eigen, [30], and OpenMP [31]. MiniDNN is a lightweight c++ library for DNN training built on top of Eigen, a c++ template library for linear algebra.
All tests were performed on a 3.40 GHz Intel(R) Xeon(R) E5-2687W v2 system with 16 cores distributed on two sockets, each with eight cores, all supporting hyper-threading. The MNIST dataset was used during all experiments [32].
The MNIST dataset contains 60 000 images of handwritten digits for training and 10 000 images for testing, [32]. Each image is in grayscale format and has a size of 28×28 pixels containing one digit that has been normalized and centered in the image. Since the dataset contains numbers that are to be classified, the network's output layer has a size of 10, and since each image is represented in grayscale, the input to the network is of size $28 \times 28 = 784$. Figure 4.1 shows an example of some digits from the MNIST dataset.

Figure 4.1: MNIST handwritten digits example

For the MLP architecture, three hidden layers with the size 128 were used during testing. This architecture was selected for its simplicity, and other architecture structures might produce higher accuracy for this classification task. Table 4.1 shows an overview of the MLP architecture.

Table 4.1: MLP architecture used, with input of 784 and d = 134 794

Layer #	Type	# Neurons	Act. func.
1-3	Dense	128	ReLU
4	Dense	10	Softmax

In Table 4.2 a summary of the experiments done and where to find the corresponding results are shown. The experiments are divided into five steps. In **S1** the effects of UMA and NUMA memory models used are analyzed in-depth for seven and eight threads using the same hyperparameter settings ($\eta = 0.005$, $B = 64$). **S1** partially addresses the question of how the memory model affects convergence and staleness. In order to answer the question more definitively, the comparison was extended to a range of hyperparameters ($\eta \in [0.001, 0.009]$, $B \in [16, 512]$) over 2, 4, and 8 threads, **S1, S2**, focusing on convergence rate and staleness distribution.

How convergence is affected by the selection of batch size and step size for different fine-grained synchronization schemes are analyzed by the hyperparameter search in **S2** together with the convergence comparison in **S4**. In **S3** the effects different batch sizes have on staleness are analyzed.

In the last step, **S5**, the three backoff schemes presented in 3.5 are compared to their corresponding algorithms not using a backoff, addressing the question if backoff can reduce contention for the shared state.

The proposed rw-lock and how it compares to the other fine-grained synchronization schemes are addressed through all of the steps **S1-S5**.

Table 4.2: Overview of experiments

Experiment overview								
Step	**Architecture**	**Description**	**N.o. threads** (m)	**Memory model**	**Precision** (ϵ)	**Step size** (η)	**Batch size** (B)	**Outcome**
S1	MLP	Memory model	7, 8	UMA, NUMA	2 %	0.005	64	figures 4.2-4.4
S2	MLP	Parameter Search	2-32	UMA (2-8), NUMA (2-32)*	5 %	0.001-0.009	16, 32, 64, 128, 256, 512	figures 4.5-4.12, Table 4.3, figures A.1-A.4
S3	MLP	Staleness and batch size	2-32	UMA (2-8), NUMA (2-32)*	-	0.001-0.009	16, 32, 64, 128, 256, 512	figures 4.13-4.18
S4	MLP	Comparison over optimal settings	7, 8	UMA, NUMA	2 %	**	**	figures 4.19-4.21
S5	MLP	Backoff and convergence	7, 8, 16, 32	-	2 %	**	**	figures 4.22-4.23

* OS scheduler used for m > 8
** Settings gained from parameter search (Table 4.3) used

4.2 Memory model and convergence

Here the results from comparing UMA to NUMA are presented. Starting with convergence rate, computational and statistical efficiency for seven threads, then followed with eight threads. All results are based on ten independent executions. The settings used was batch size, $B = 64$ and step size $\eta = 0.005$. The box contains the 1^{st} and 3^{rd} quantile, outliers are indicated as points. Crashed executions are reported as crashed, and executions that did not manage to reach ϵ-convergence within 60 s are reported as diverge.

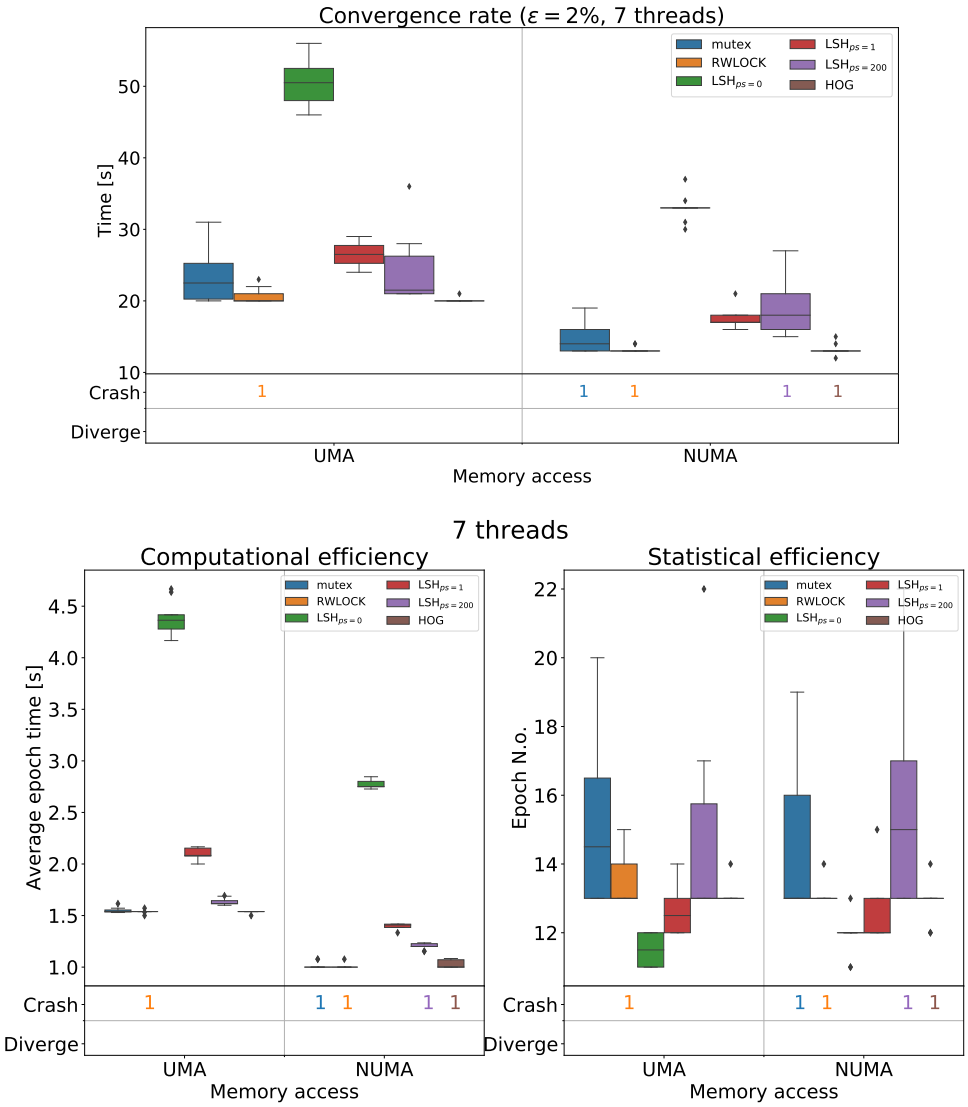

Figure 4.2: Thread scheduling comparison for 7 threads. NUMA refers to 4 threads scheduled on socket 0 and 3 threads scheduled on socket 1, and UMA refers to 7 threads scheduled to socket 0. Time to reach 2 %-convergence (top). Average epoch time (e.g. Computational efficiency) (bottom left) and number of epoch to reach 2 %-convergence (e.g. statistical efficiency) (bottom right). Step size $\eta = 0.005$ and batch size $B = 64$.

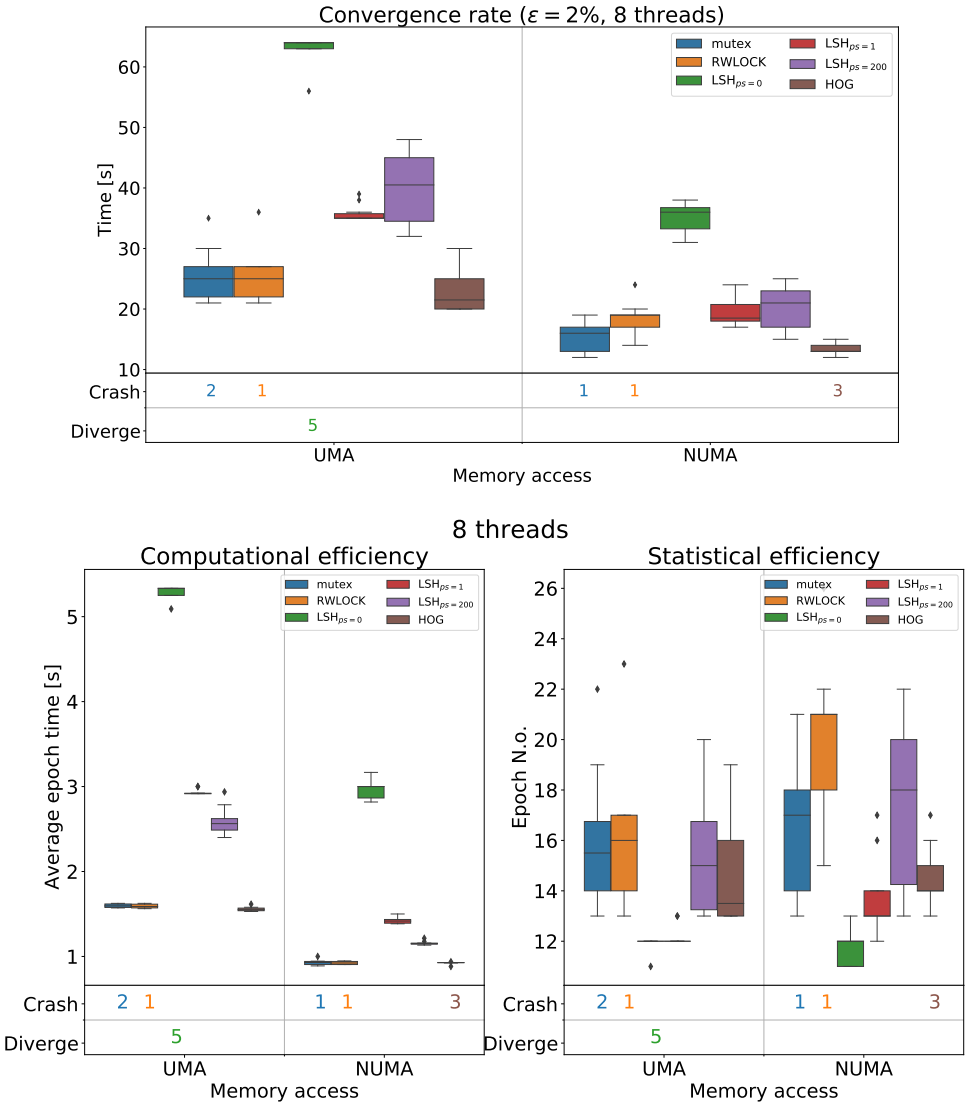

Figure 4.3: Thread scheduling comparison for 8 threads. NUMA refers to 4 threads scheduled on socket 0 and 4 threads scheduled on socket 1 and UMA refers to 8 threads scheduled to socket 0. Average epoch time (e.g. Computational efficiency) (left) and number of epoch to reach 2 %-convergence (e.g. statistical efficiency) (right). Step size $\eta = 0.005$ and batch size $B = 64$.

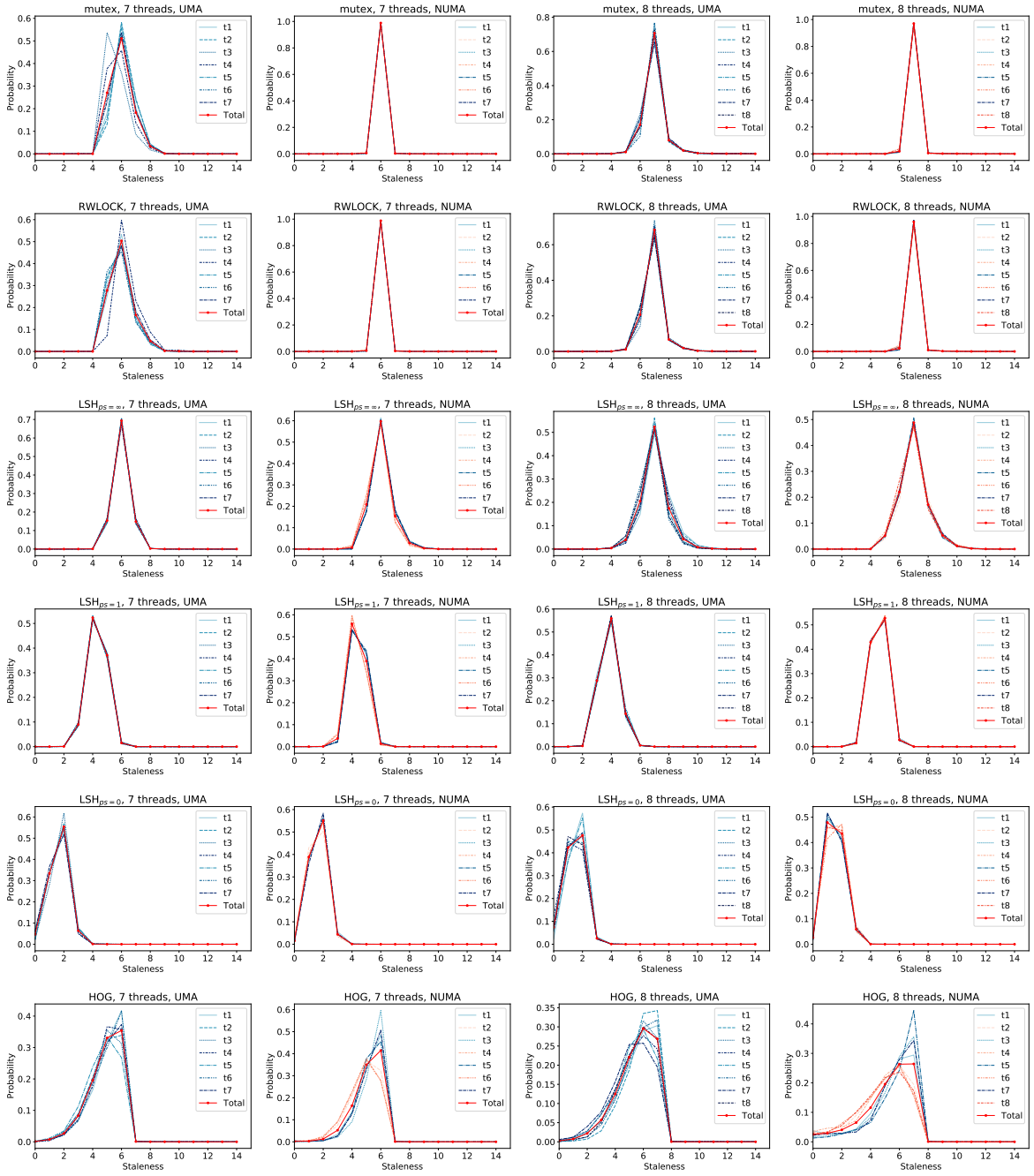

Figure 4.4: Staleness distribution for eight and seven threads with UMA and NUMA memory model. t1-t8 indicate each individual thread and *Total* are the total staleness. Blue represent threads scheduled on socket 0 and orange threads scheduled on socket 1. Batch size, $B = 64$ and step size, $\eta = 0.005$.

4.3 Convergence analysis

Figure 4.5-4.12 shows a selection of the results from the hyperparameter search, *Leashed-SGD* with persistence bound of zero and one are located in Appendix A. All results in this section are based on ten independent executions, and each execution was limited to 60 s. The heatmaps (figure 4.5-4.12) shows time to reach ϵ-convergence of the initial error, $\theta^0 \approx 2.3$, where $\epsilon = 5\,\%$. Crash indicates the number of times the algorithm crashed and diverge the number of times the execution failed to reach ϵ-convergence. PS indicates the persistence bound of *Leashed-SGD*. AsyncSGD with mutex lock, rw-lock is referred to as *mutex*, and *RWLOCK* respectively. The section starts with presenting the results from each algorithm over 2-32 threads and for UMA and NUMA using heatmap representations of mean time to reach 5 %-convergence for different hyperparameter settings, the number of crashed and diverged executions are also presented.
In Table 4.3 a summary of the settings reaching 5 %-convergence the fastest for each level of parallelism, memory model, and algorithm is presented. The table is derived from figures 4.5-4.12, A.1-A.4. Over the test range ($\eta \in [0.001, 0.009]$, $B \in [16, 32, ..., 512]$) there are 54 possible combinations of hyperparameter settings. As a measurement of parameter selection robustness, the number of parameter combinations within 10 % of the training time with the fastest convergence was measured. From this, all parameter combinations with more than 10 % crashed executions were excluded. They indicate the range of hyperparameters resulting in an optimal or close to optimal convergence time for that level of parallelism and algorithm. The time for this range is also reported in the table. The total number of crashed execution over the whole test range is reported, indicating the algorithm's general stability over that level of parallelism.

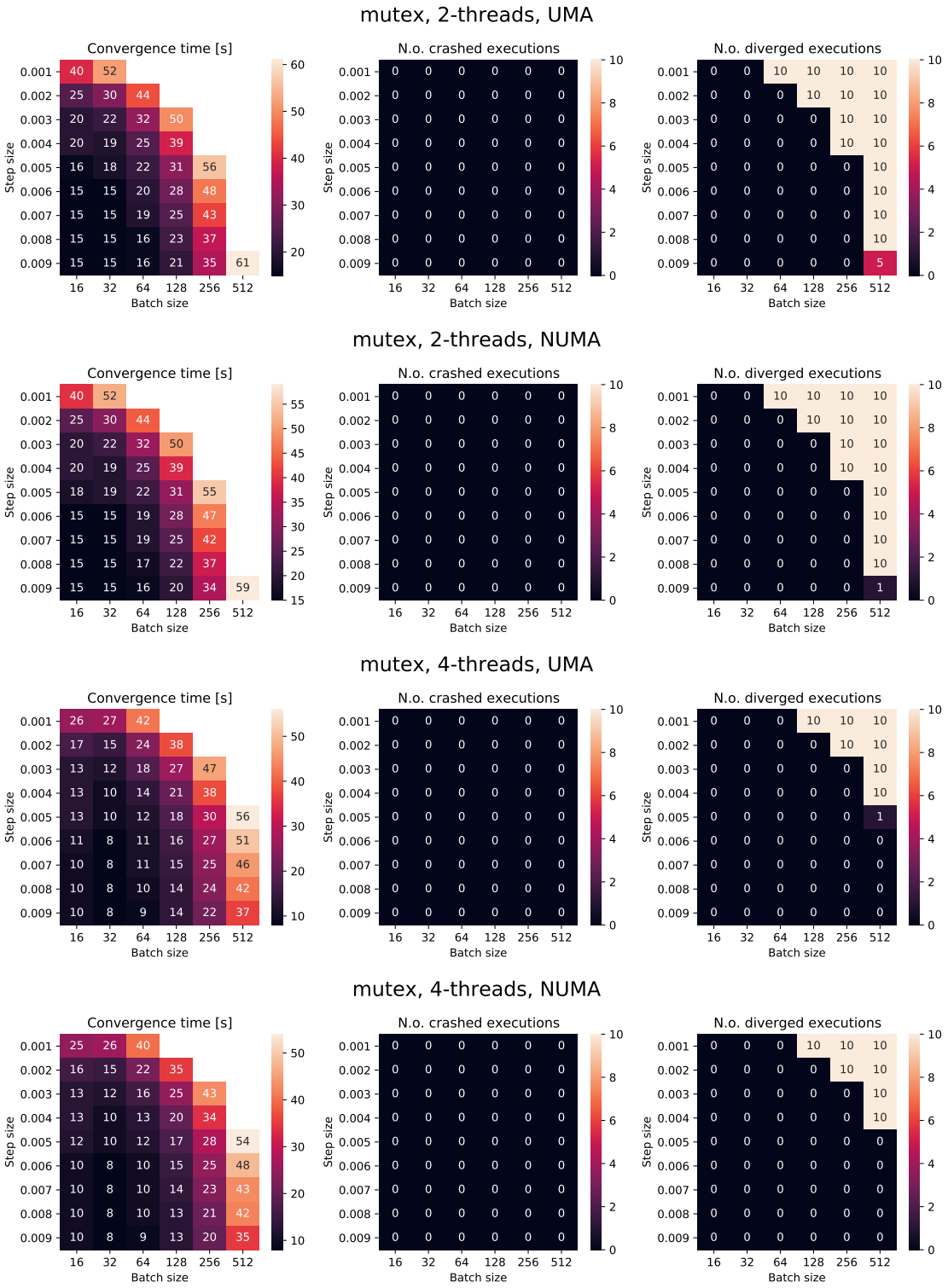

Figure 4.5: Convergence time to reach 5 % of initial loss for AsyncSGD with mutex lock, 2 and 4 threads for UMA and NUMA. Crashed executions indicate number of crashed executions. Based on ten independent runs.

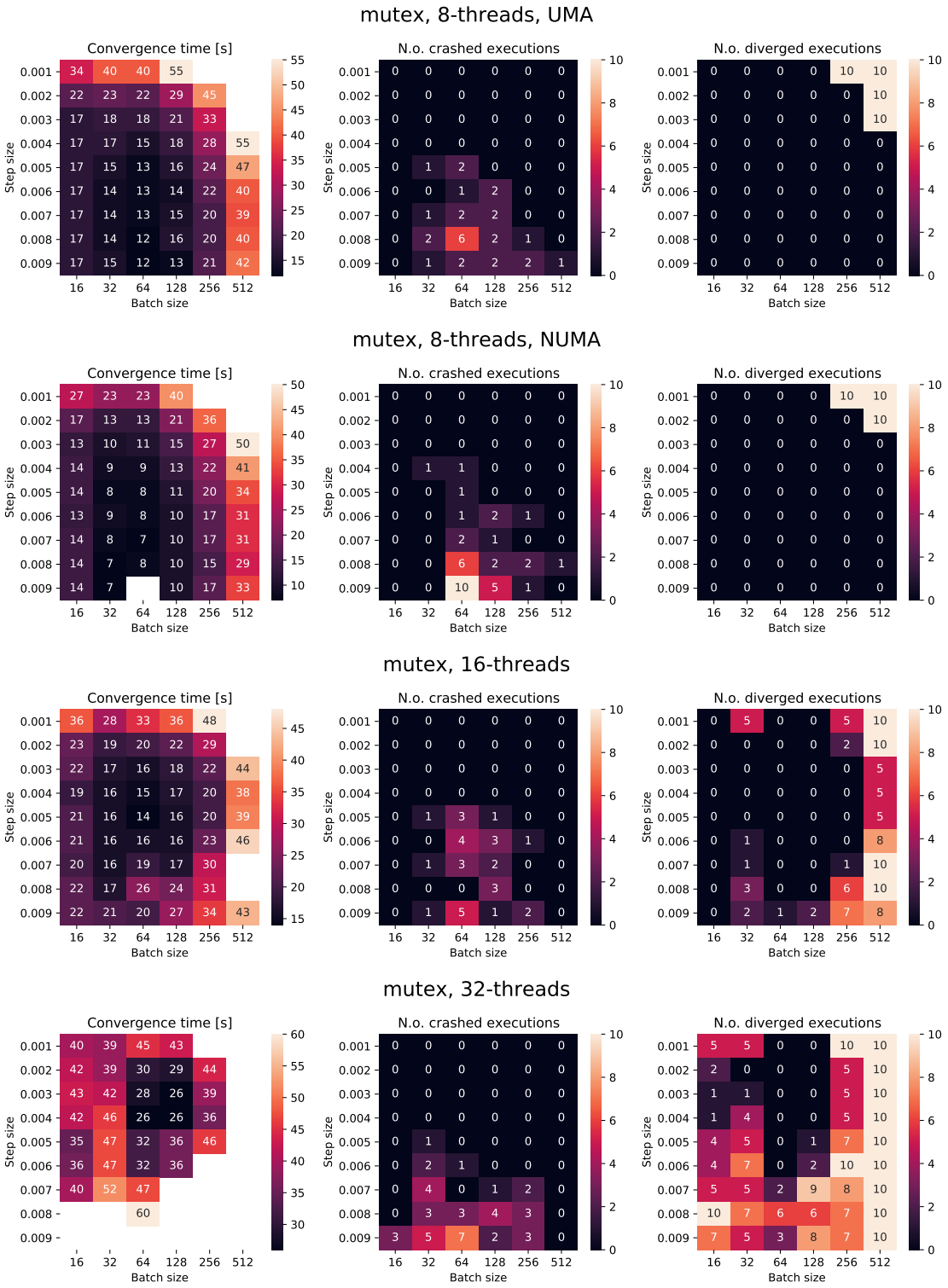

Figure 4.6: Convergence time to reach 5 % of initial loss (left) for AsyncSGD with mutex lock, 8, 16 and 32 threads, UMA and NUMA for 8 threads and OS scheduler for 16 and 32 threads. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs.

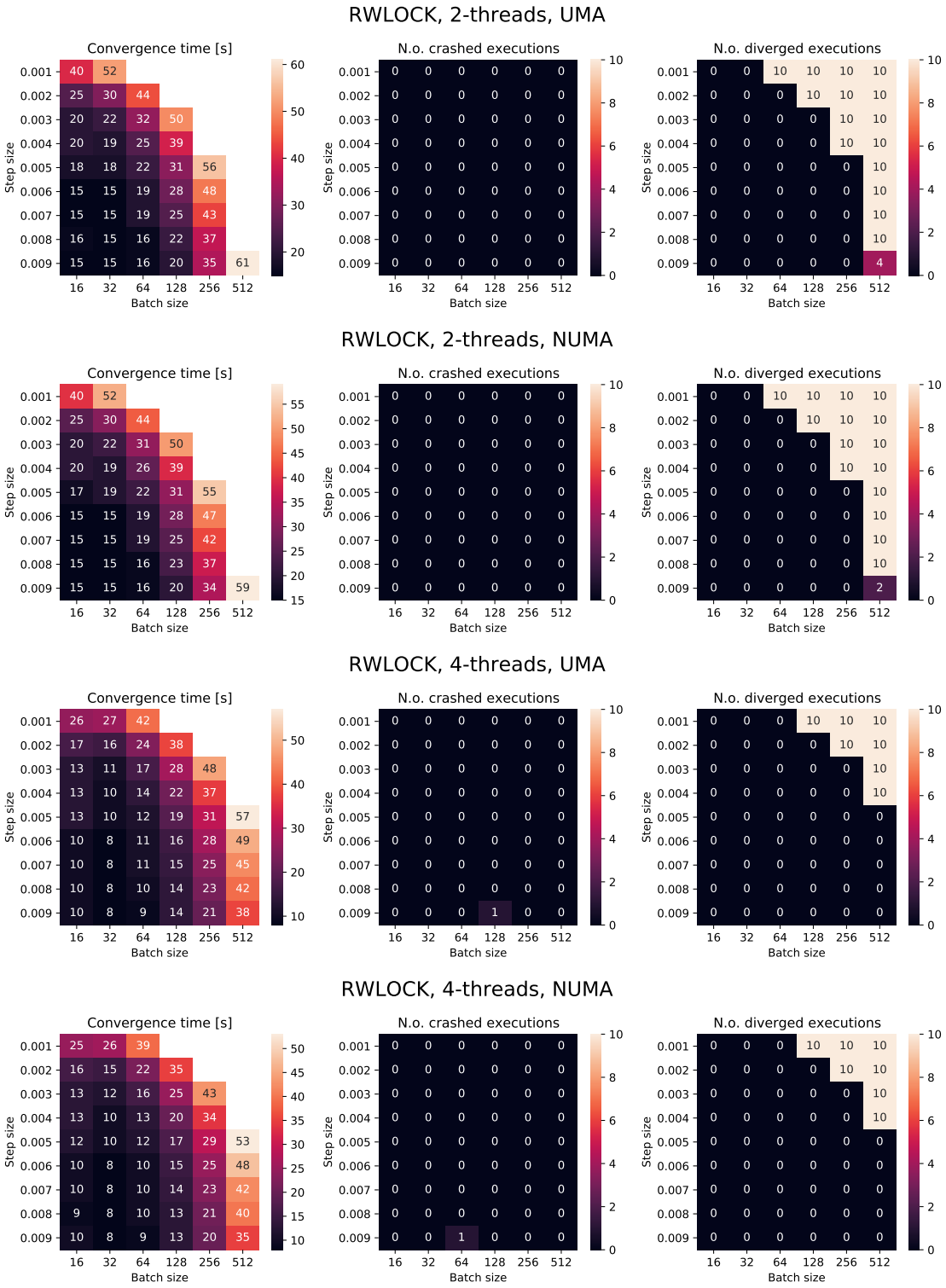

Figure 4.7: Convergence time (left) to reach 5 % of initial loss for AsyncSGD with mutex lock, 2 and 4 threads for UMA and NUMA. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs.

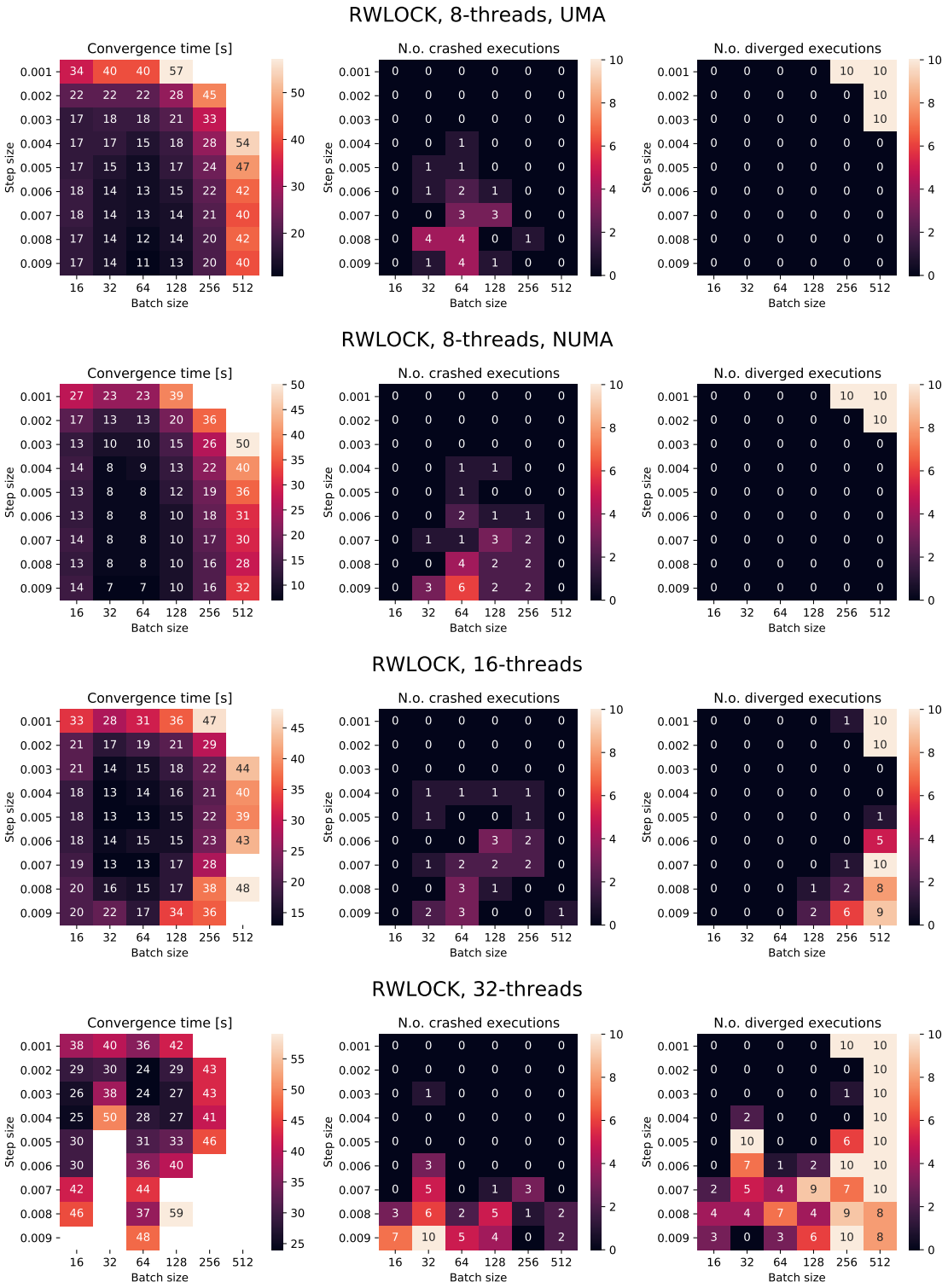

Figure 4.8: Convergence time to reach 5 % of initial loss (left) for AsyncSGD with rw-lock, 8, 16 and 32 threads, UMA and NUMA for 8 threads and OS scheduler for 16 and 32 threads. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs.

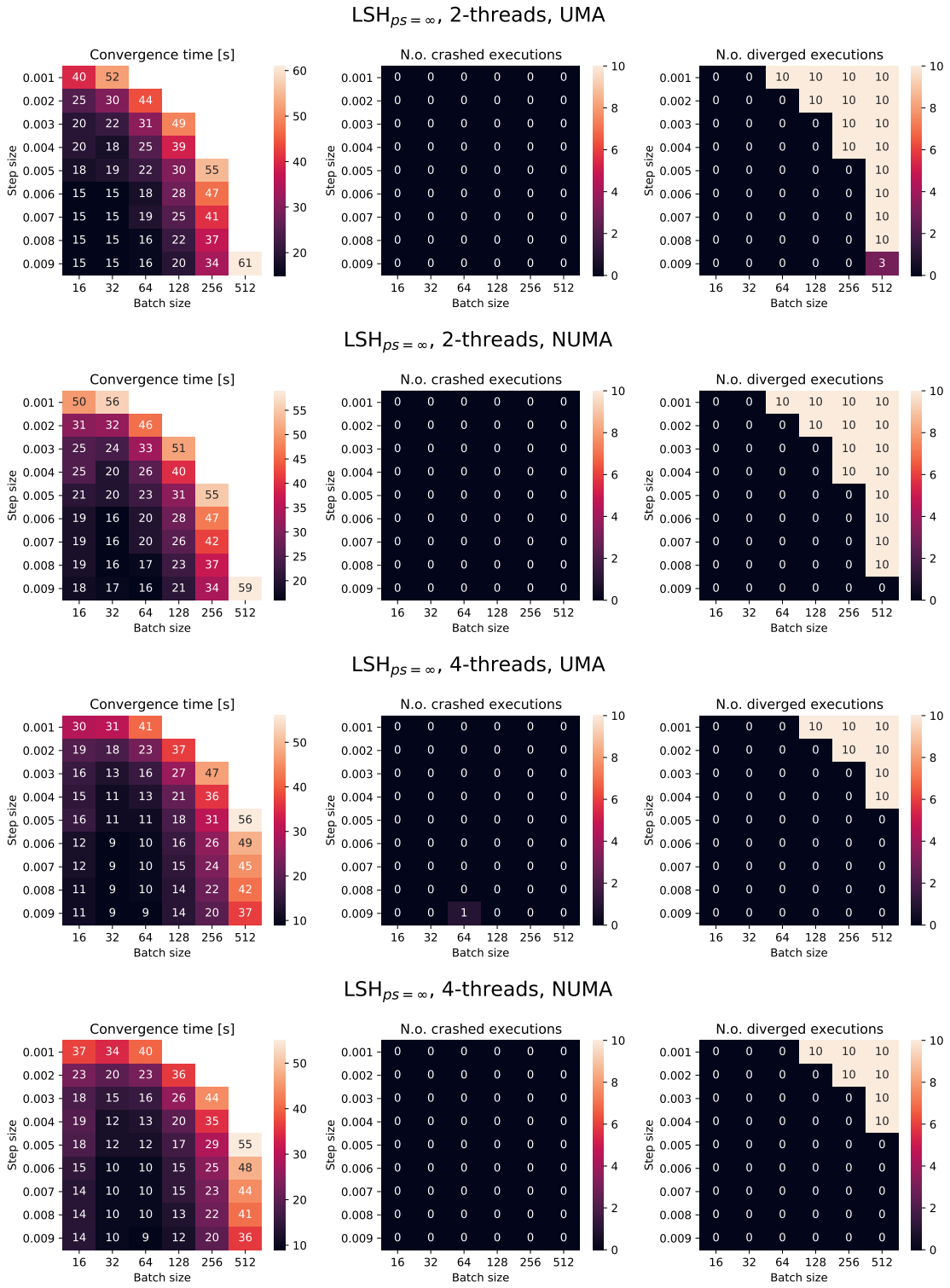

Figure 4.9: Convergence time (left) to reach 5 % of initial loss for *Leashed-SGD* with persistence bound $ps = \infty$, 2 and 4 threads for UMA and NUMA. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs.

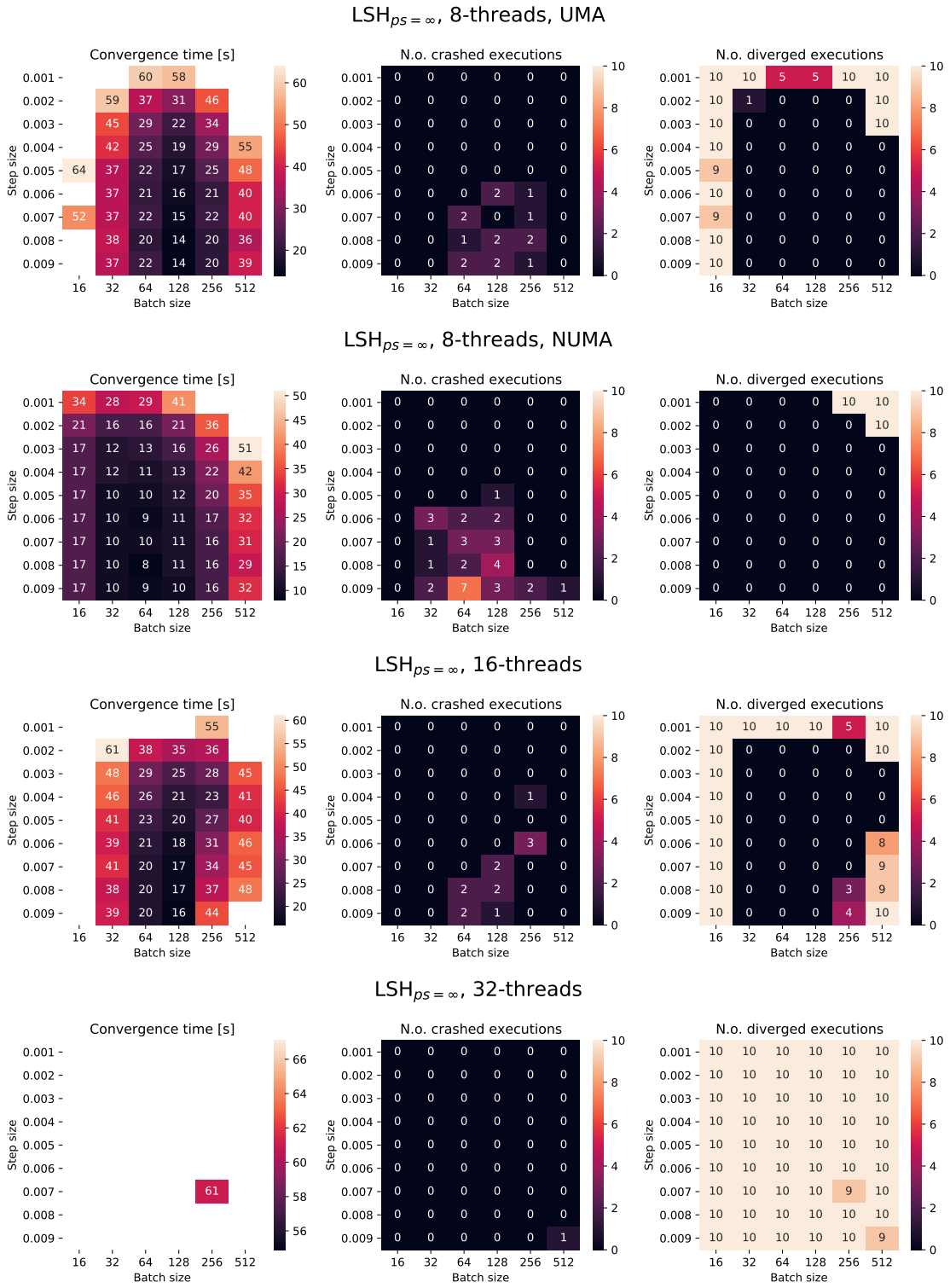

Figure 4.10: Convergence time to reach 5 % of initial loss (left) for *Leashed-SGD* with persistence bound $ps = \infty$, 8, 16 and 32 threads, UMA and NUMA for 8 threads and OS scheduler for 16 and 32 threads. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs.

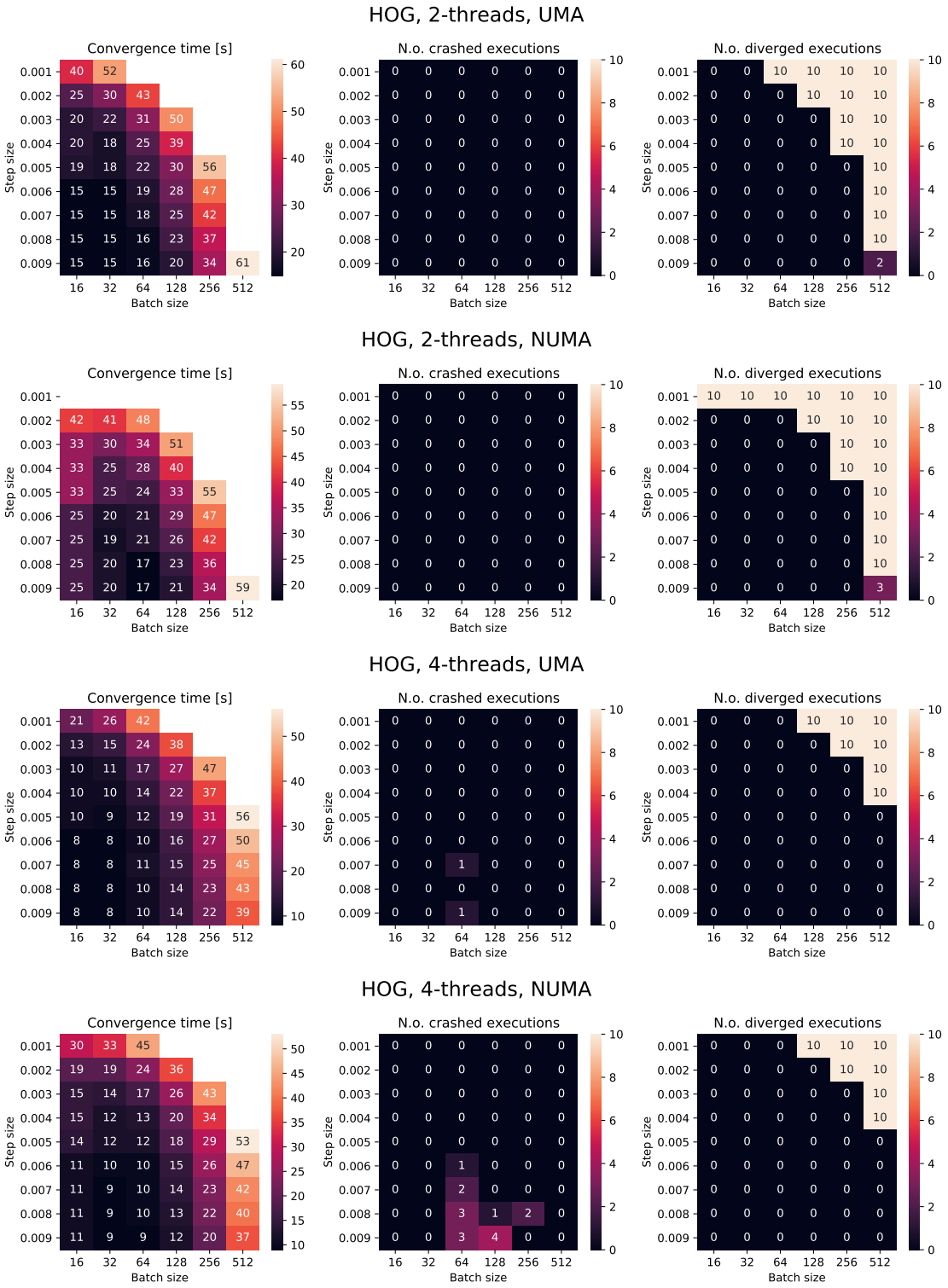

Figure 4.11: Convergence time (left) to reach 5 % of initial loss for HOGWILD!, 2 and 4 threads for UMA and NUMA. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs.

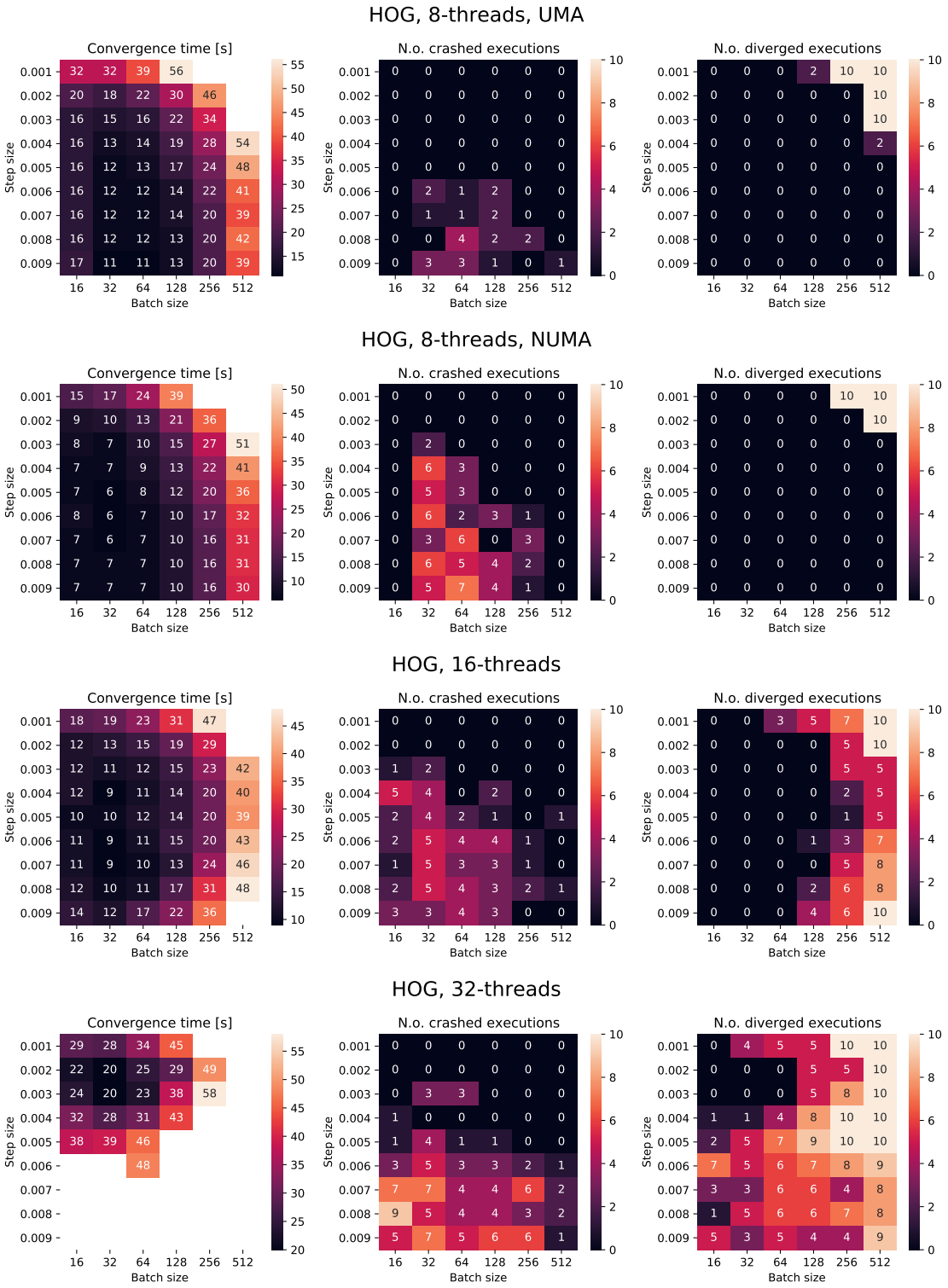

Figure 4.12: Convergence time to reach 5 % of initial loss (left) for HOGWILD!, 8, 16 and 32 threads, UMA and NUMA for 8 threads and OS scheduler for 16 and 32 threads. Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs.

Table 4.3: Parameter settings for each algorithm that reached 5 % of initial error the fastest with a maximum of 10 % crashed executions, based on 10 independent runs of all algorithms with step size, $\eta \in [0.001, 0.009]$ and batch size, $B \in \{16, 32, 64, 128, 256, 512\}$. Fastest converging setting reported as Step size and Batch size derived from figures 4.5-4.12, A.1-A.4. The #Parameter combinations indicates the number of settings reaching 5 %-convergence within 10 % of the time fastest execution with a maximum of 10 % crashed executions. Crashes represent the total number of crashed executions over the whole test range (540 executions).

Threads	Memory model	Algorithm	Step size (10^{-3})	Batch size	#Parameter combinations	Time [s]	Crashes
2	UMA	Mutex	9	32	11	15-16	0
			9	32	10	15-17	0
		RW-lock	9	32	10	15-16	0
			9	32	11	15-17	0
		$LSH_{PS=\infty}$	9	32	10	15-16	0
			9	64	7	16-18	0
		$LSH_{PS=1}$	9	32	11	15-17	0
			8	64	7	16-18	0
		$LSH_{PS=0}$	8	64	5	21-23	0
			9	64	4	25-27	0
		HOG	9	32	10	15-16	0
			9	64	3	17-19	0
4	UMA	Mutex	9	32	5	8-9	0
			9	32	5	8-9	0
		RW-lock	9	32	5	8-9	1
			9	32	6	8-9	1
		$LSH_{PS=\infty}$	9	64	8	9-10	1
			9	64	8	9-10	0
		$LSH_{PS=1}$	9	64	8	9-10	1
			9	64	5	9-10	4
		$LSH_{PS=0}$	9	64	4	20-22	0
			9	64	3	19-21	0
		HOG	9	32	9	8-9	2
			9	16	4	9-10	16
8	UMA	Mutex	6	64	3	13-14	30
			9	32	8	7-8	37
		RW-lock	5	64	6	13-14	28
			8	32	7	8-9	35
		$LSH_{PS=\infty}$	7	128	2	15-17	16
			5	64	5	10-11	37
		$LSH_{PS=1}$	8	128	3	16-18	9
			8	64	4	9-10	22
		$LSH_{PS=0}$	9	64	2	27-28	0
			9	64	7	16-18	3
		HOG	8	32	8	12-13	25
			9	16	6	7-8	77
16	*	Mutex	4	64	10	15-17	31
		RW-lock	5	64	7	13-14	28
		$LSH_{PS=\infty}$	9	128	2	16-18	13
		$LSH_{PS=1}$	9	64	5	20-21	9
		$LSH_{PS=0}$	9	128	6	25-28	0
		HOG	4	64	5	11-12	82
32	*	Mutex	3	128	5	26-29	44
		RW-lock	3	64	4	24-26	60
		$LSH_{PS=\infty}$	-	-	-	-	1
		$LSH_{PS=1}$	8	128	5	53-58	2
		$LSH_{PS=0}$	8	128	4	49-54	1
		HOG	2	32	2	20-22	118

* OS scheduler used

4.4 Staleness distribution and batch size

The following figures shows the total staleness distribution for 8 and 32 threads for different batch sizes. The step size do not affect staleness and was therefor set to a value in the middle of the ranged used during testing, e.g. $\eta = 0.005$.

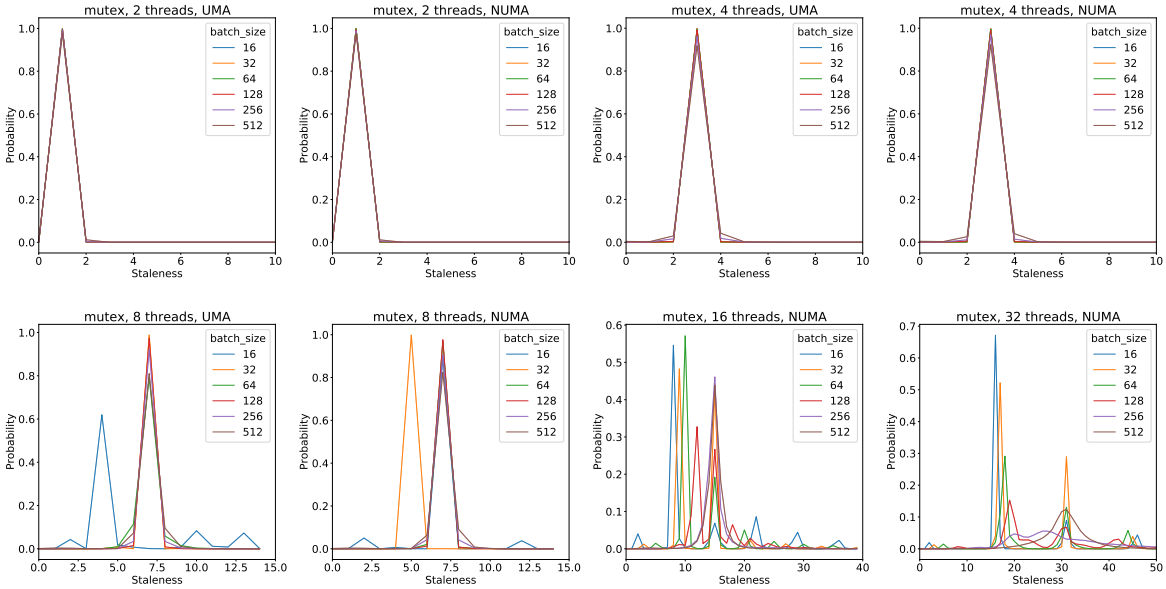

Figure 4.13: Staleness distribution for AsyncSGD with mutex lock, based on the average of 10 independent executions.

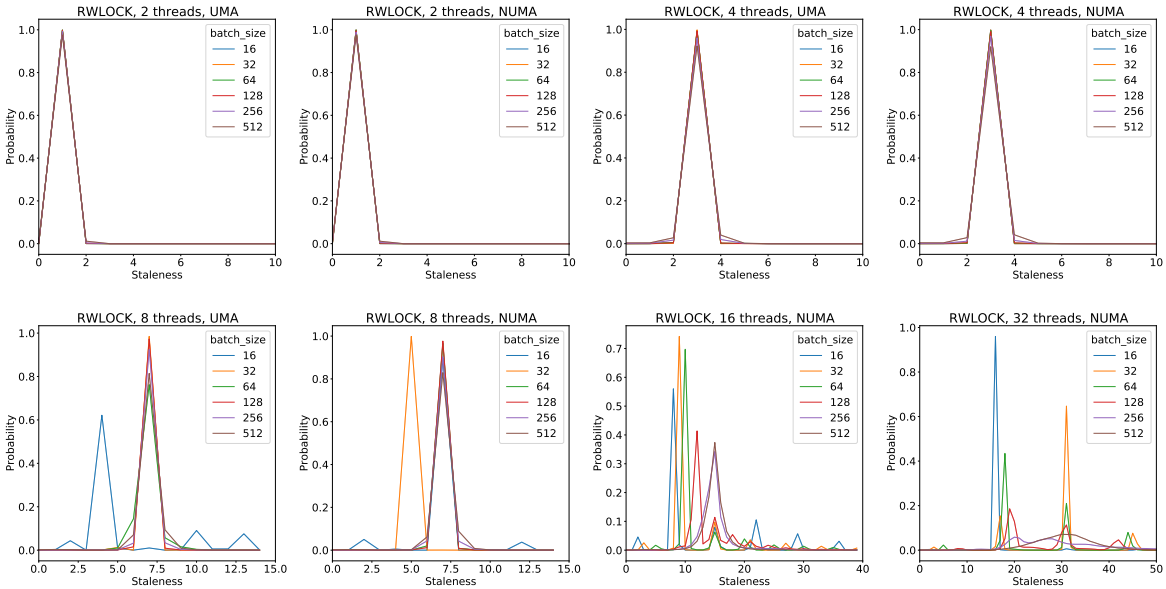

Figure 4.14: Staleness distribution for AsyncSGD with RW-lock lock, based on the average of 10 independent executions.

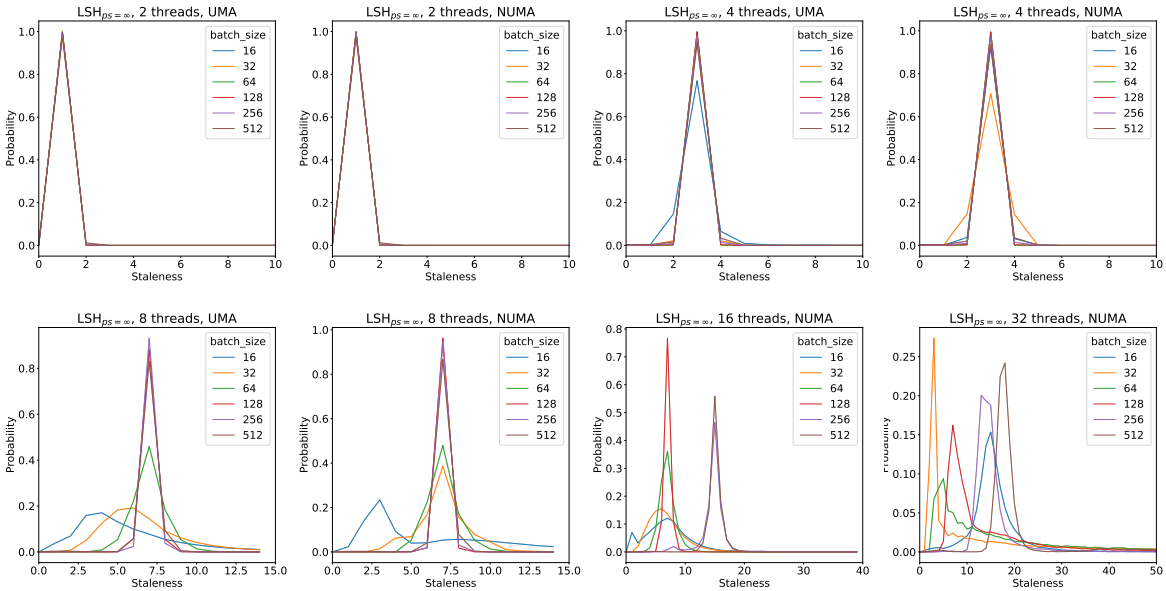

Figure 4.15: Staleness distribution for Lsashed-SGD [1], with persistence bound $ps = \infty$, based on the average of 10 independent executions.

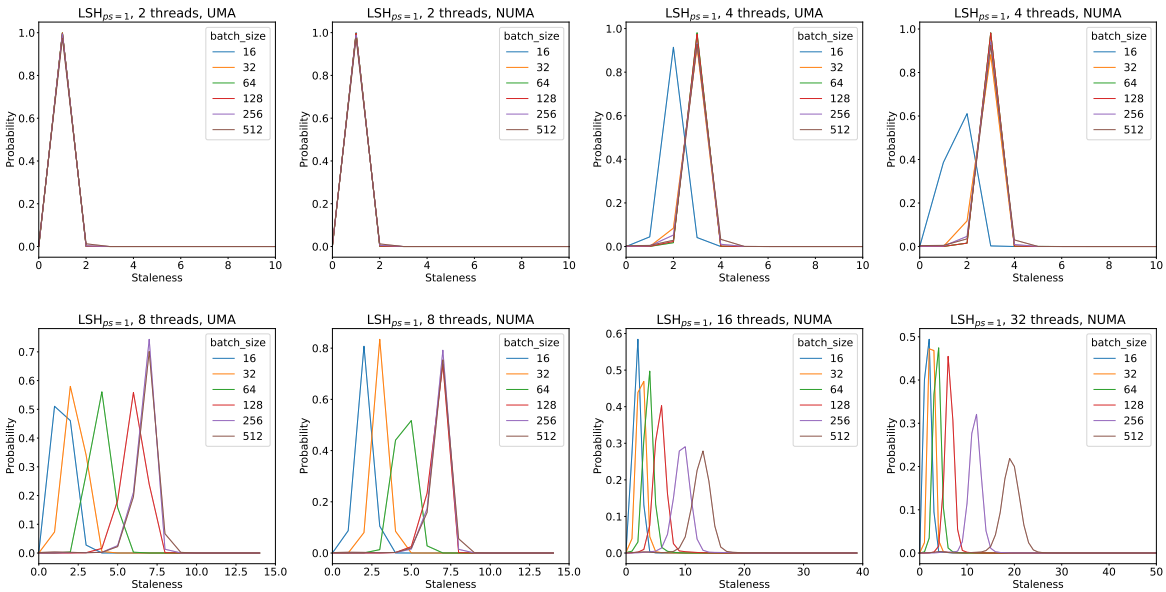

Figure 4.16: Staleness distribution for Lsashed-SGD [1], with persistence bound ps=1, based on the average of 10 independent executions.

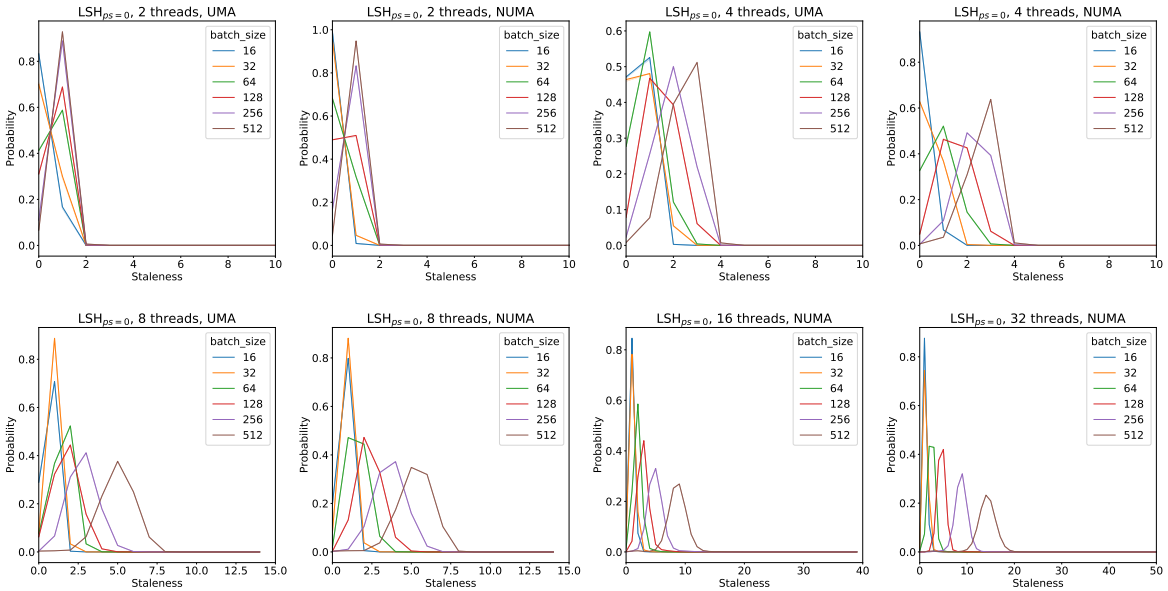

Figure 4.17: Staleness distribution for Lsashed-SGD [1], with persistence bound ps=0, based on the average of 10 independent executions.

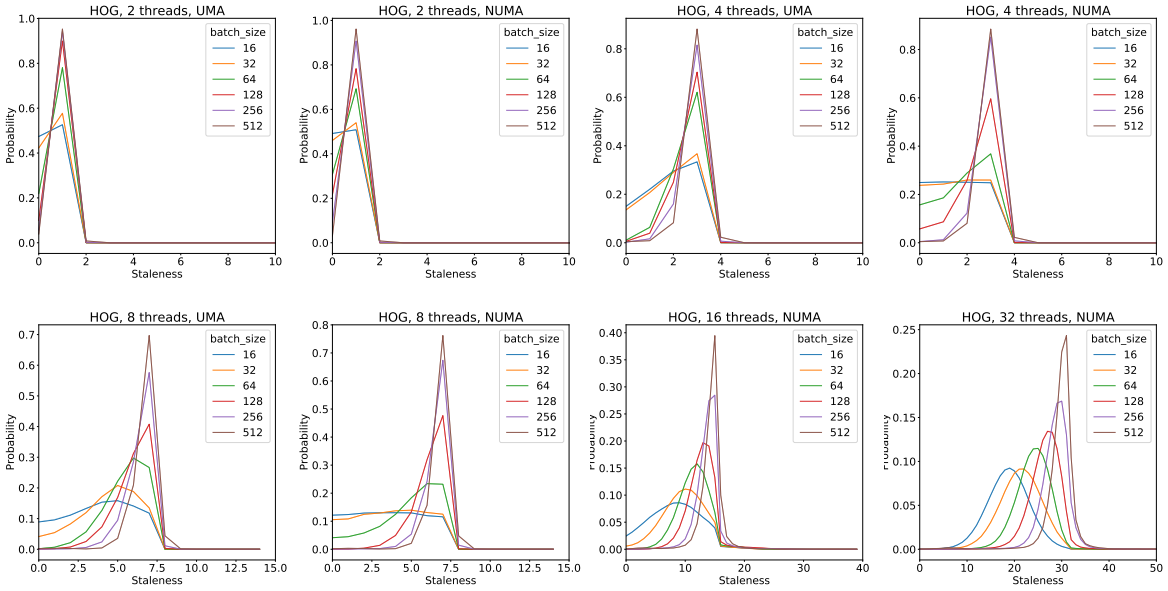

Figure 4.18: Staleness distribution for HOGWILD! [2], based on the average of 10 independent executions.

4.5 Convergence comparison

In this section a comparison of the the algorithms are done based on the settings from Table 4.3. The comparison are done by analyzing convergence rate for 2-32 threads. Then statistical and computational efficiency, individual staleness for each thread, and number of updates will be analyzed for eight threads.

The following figures shows time to ϵ-convergence, where $\epsilon = \{10\%, 5\% \text{ and } 2\%\}$ of the initial loss, for AsyncSGD with mutex lock, RW-lock, HOGWILD! and *Leashed-SGD* with the settings gained from the parameter search, e.g. Table 4.3. The box contains the 1^{st} and 3^{rd} quantile from 10 independent executions, outliers are indicated as points. The step size (η) and batch size (B) used for each algorithm are presented in the legend and in Table 4.3.

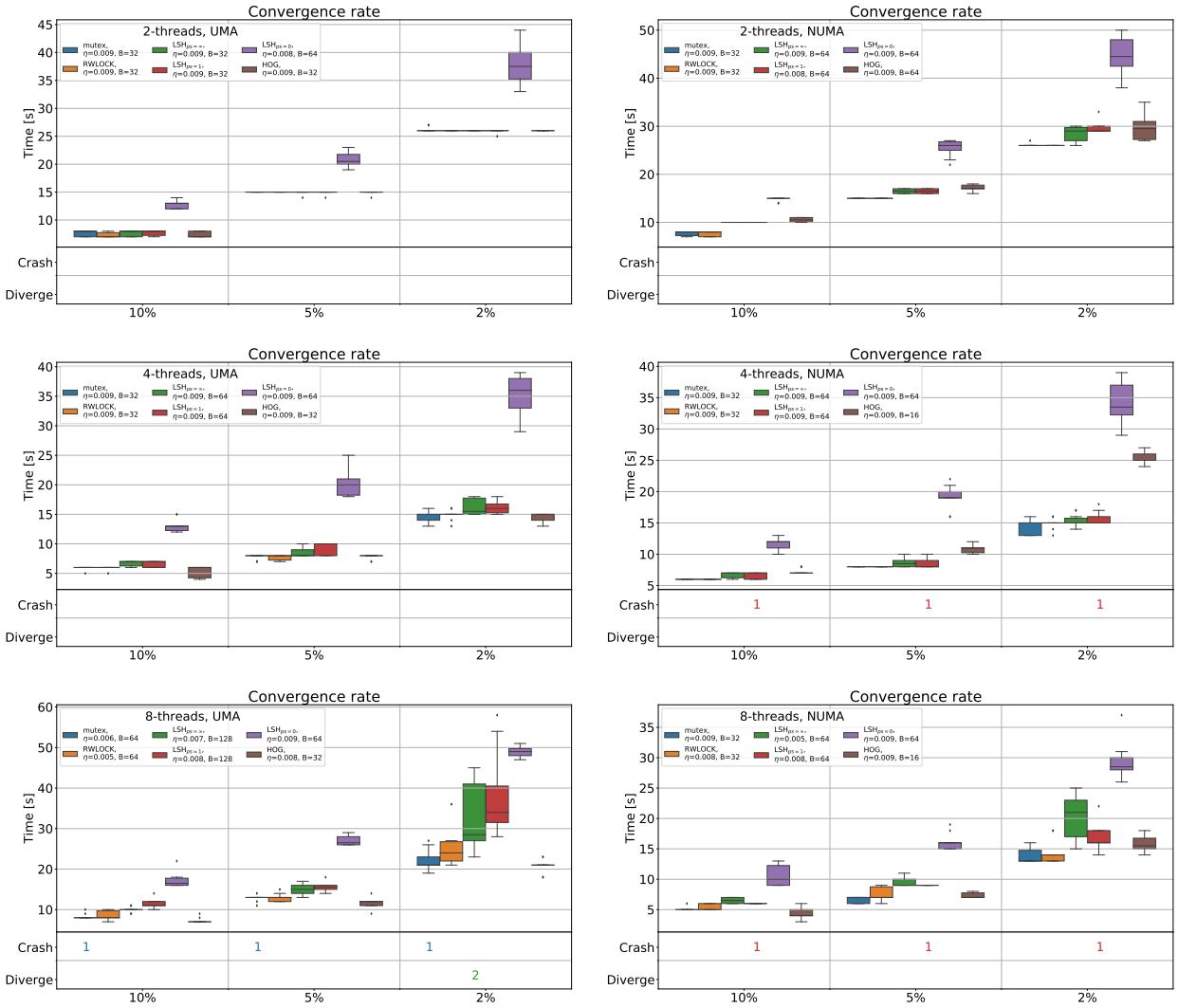

Figure 4.19: Convergence rate for MLP with $\epsilon = \{10\%, 5\%, 2\%\}$ of the initial error, maximum level of parallelism m=8 and minimum level of parallelism m=2. UMA (left), NUMA (right), step size (η) and batch size (B) are indicated in the legend and in Table 4.3. Diverge indicate the number of times respective algorithm failed to reach ϵ-convergence within 60 seconds and crash indicate the number of times the execution crashed. PS indicate the persistence bound for *Leashed-SGD*. Based on ten independent runs of each setting.

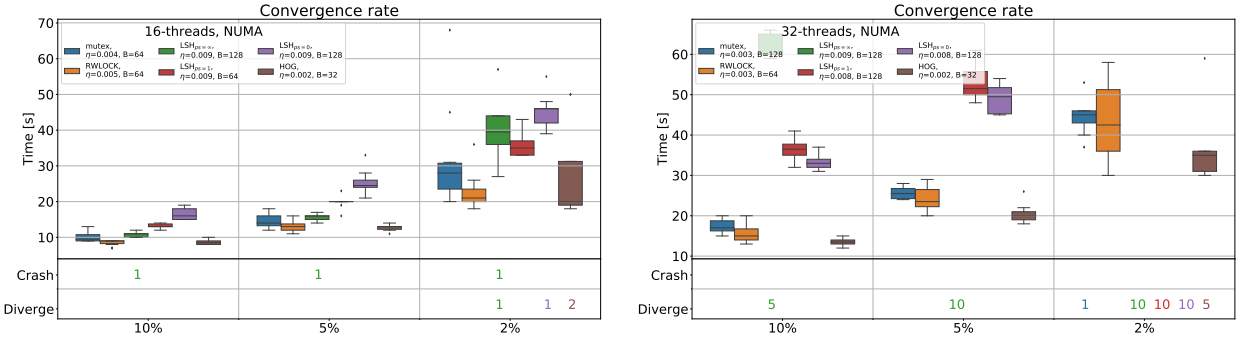

Figure 4.20: Convergence rate for MLP with $\epsilon = \{10\%, 5\%, 2\%\}$ of the initial error, for 16 and 32 threads. Step size (η) and batch size (B) are indicated in the legend and in Table 4.3. Diverge indicate the number of times respective algorithm failed to reach ϵ-convergence within 60 seconds and crash indicate the number of times the execution crashed. PS indicate the persistence bound for *Leashed-SGD*. Based on ten independent runs of each setting.

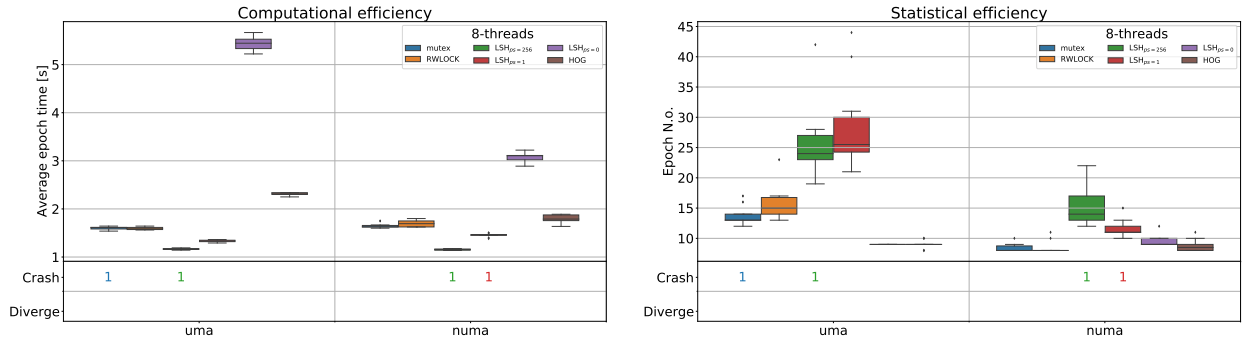

Figure 4.21: Average epoch time (e.g. computational efficiency, left) and number of epochs to reach 2 %-convergence (e.g. statistical efficiency, right) for eight threads. Diverge indicate the number of times respective algorithm failed to reach ϵ-convergence within 60 seconds and crash indicate the number of times the execution crashed. PS indicate the persistence bound for *Leashed-SGD*. Based on ten independent runs, settings presented in Table 4.3

4.6 Backoff

The backoff schemes presented in Section 3.5 are compared in Figure 4.22, the settings used are presented in Table 4.3.

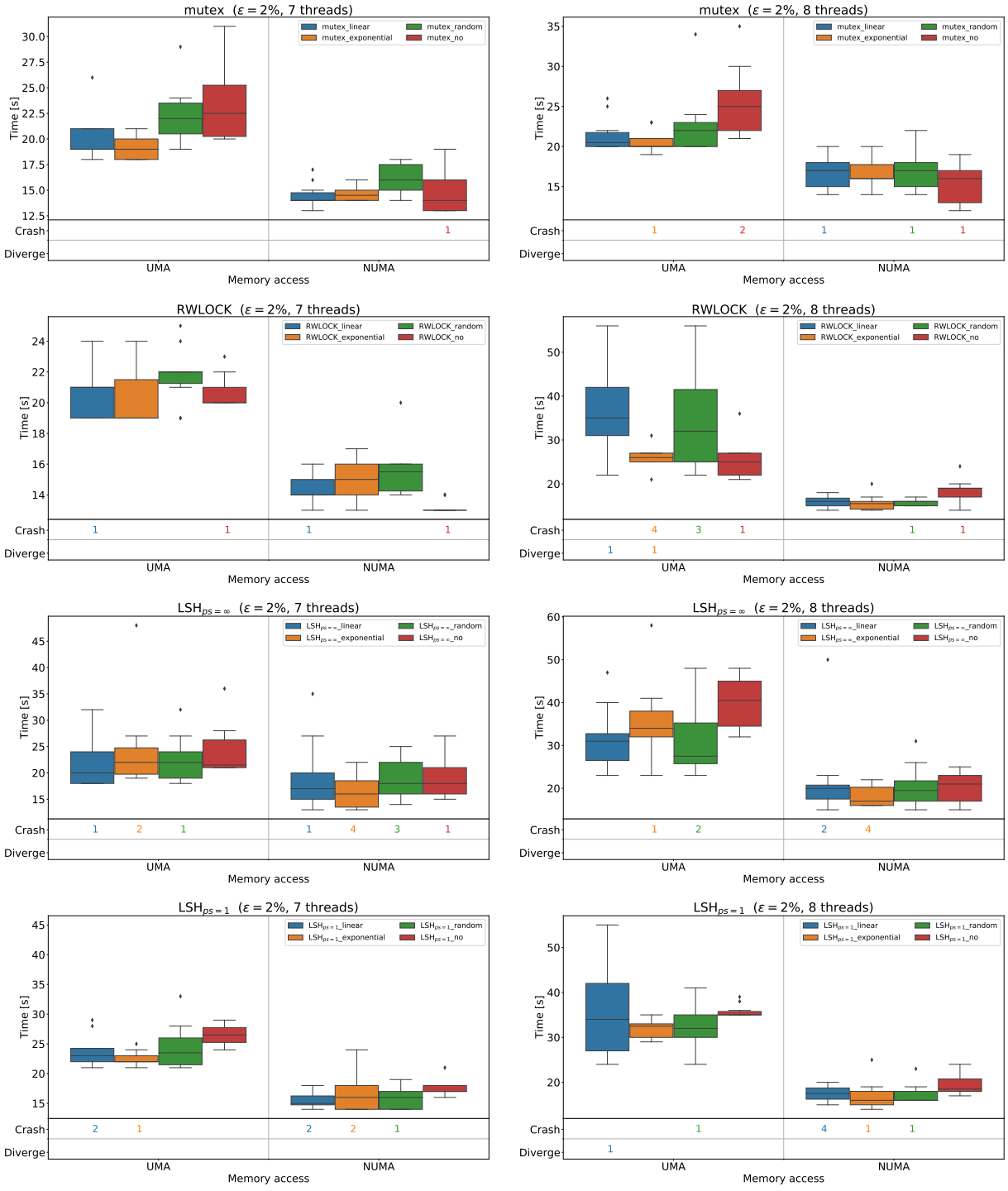

Figure 4.22: Convergence rate with linear, exponential, random and no backoff, seven (left) and eight (right) threads. ϵ =2 %, settings used are presented in Table 4.3.

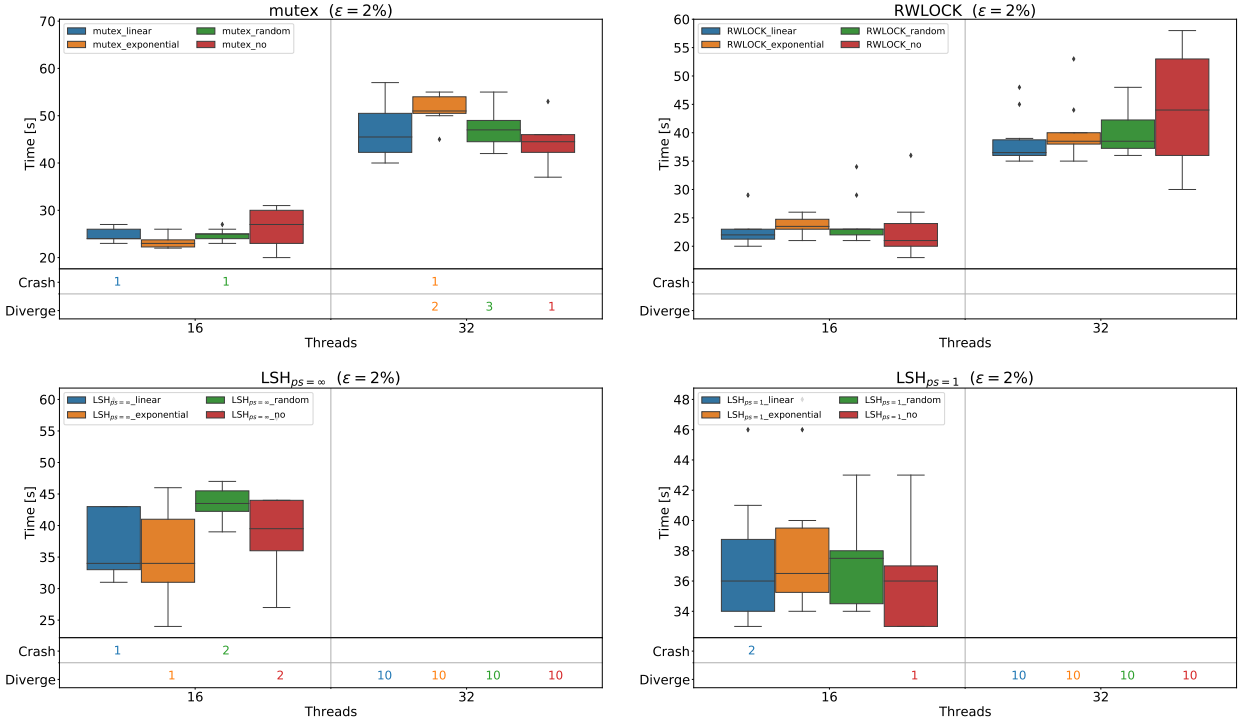

Figure 4.23: Convergence rate of AsyncSGD (mutex- and rw-lock) and *Leashed-SGD* with linear, exponential, random and no backoff, 16 (left) and 32 (right) threads. ϵ =2 %, settings used are presented in Table 4.3.

4.7 Discussion of Experiments

S1. *Scheduling and memory model:* Here, the difference in convergence for UMA and NUMA are compared. In this step hyperparameters are selected in the middle of the test range, e.g. $\eta = 0.005$ and $B = 64$. The system is limited to a maximum of eight threads for UMA; both eight and seven threads were tested. The results are presented in figures 4.2-4.3, where the convergence rate, statistical and computational efficiency is presented. We can see that NUMA reaches 2 %-convergence faster than UMA for all algorithms, both for seven and eight threads. The speedup from NUMA compared to UMA is between 60 and 80 %. When analyzing from where the improved convergence comes, we see that all speedup comes from computational efficiency. The Statistical efficiency is more or less unchanged, with some reduction in statistical efficiency for NUMA compared to UMA. One possible explanation for this is because the increased communication time for NUMA contributes to reduced contention for the shared state θ. In Figure 4.4 the staleness distribution is analyzed. We can see that for the lock-based algorithms and Leashed-SGD the staleness is concentrated at the one less than the number of threads $(m-1)$, which is expected. For HOGWILD! eight and seven threads with NUMA, the staleness is grouped in two, one with all threads scheduled on socket zero (blue lines) and the other scheduled on socket one (orange lines). An explanation on why this is only present for HOGWILD! is that the overhead introduced by the other fine-grained synchronization schemes is larger than the increased communication between threads. The differ-

ence between seven and eight threads is nothing more than the expected differences when introducing one additional thread.

S2. *Convergence and hyperparameter selection:* Here, we analyze how hyperparameter selection affects convergence for different levels of parallelism. Since the memory model made a difference in convergence the testes is done for both memory models, the results can be found in figures 4.5-4.12. In Table 4.3 we have a list of the fastest converging hyperparameter settings for each algorithm. From Table 4.3 we can see that the memory model makes less difference when parallelism is low, e.g., $m = \{2, 4\}$, and that UMA tends to converge slightly faster than NUMA for $m = 2$. We can also see that the selection of hyperparameters is highly dependent on the level of parallelism and individual differences between the different algorithms. We can see an increase in crashed executions with increasing parallelism, especially for the lock-based algorithms and HOGWILD!. Generally, the selection of hyperparameters is more limited as parallelism increases because of the increased crashed and diverged executions. In terms of stability, we can see that Leashed-SGD stands out compared to the other algorithm with significantly fewer crashed executions. If we only consider convergence time, HOGWILD! generally is the fastest algorithm to reach ϵ-convergence for a higher level of parallelism, $m > 8$. However, HOGWILD! is also the algorithm with the most crashed executions, especially for a higher level of parallelism. The step size for the lock-based algorithms and HOGWILD! needs to be significantly reduced when parallelism increases. The reason for this is likely because of the increased staleness when parallelism increases. An update with high staleness can have a negative impact on convergence. Since the step size essentially scales the contribution of each update, each update based on a stale view of the state will not have as high of an impact. Comparing the two lock-based algorithms, we can see some improvements in terms of convergence time using the rw-lock compared to the mutex lock for a higher level of parallelism, $m > 8$.

S3. *Staleness and batch size:* Here, we investigate the effects parallelism have on staleness with different batch sizes, the results are presented in figures 4.13-4.18. We also compare the staleness for UMA and NUMA. We can see that the memory model have little effect on staleness, except for lock-based AsyncSGD with eight threads where we can see a difference for batch size 16 and 32. The lock-based algorithms have almost identical staleness distribution except for higher parallelism ($m = \{16, 32\}$), where we have some differences. Increasing parallelism generally results in higher staleness which is expected, and we can also see a larger difference between different batch sizes with increasing parallelism. For leashed and HOGWILD!, staleness is increased with increasing batch size.

S4. *Convergence comparison:* Here we compare the different algorithms using the hyperparamter settings resulting in the fastest convergence time with no more then 10 % crashed executions. This is done for two to 32 threads and the time to reach ϵ-convergence ($\epsilon = \{10\,\%, 5\,\%, 2\,\%\}$) are presented in figures 4.19-4.20. The overall fastest time to reach 2 %-convergence was obtained using eight threads, NUMA, where the lock-based algorithms had the fastest convergence time.

S5. *Backoff:* Here, we analyze if backoff can be used to reduce contention of the shared state resulting in faster convergence time or increased scalability. The results are presented in figures 4.22. We can see some improvements using backoff, except

for rw-lock, where the backoff had a slightly negative effect on convergence. However, further testing and tuning of backoff-specific parameters such as maximum backoff time and initial backoff are needed. The introduction of a backoff might impact how hyperparameters should be selected, and therefore, testing with additional hyperparameter settings is desirable. Additionally, backoff specific parameters introduced need to be tuned.

5
Conclusions and future work

This thesis empirically analyzes how the memory model used affects convergence for different fine-grained synchronization schemes in a deep learning application. Other works studying the impact on the memory model are limited. One study on lower-dimensional tasks using SVMs indicates that increased communication for NUMA increases convergence time compared to UMA [27]. The results from experiment **S1, S2, S4** indicate that NUMA actually can reduce time to convergence for $m > 4$. The increased convergence rate mainly comes from increased computational efficiency, indicating that the increased communication time might reduce contention for the shared state.

Hyperparameter selections for AsyncSGD under varying parallelism and different fine-grained synchronization schemes are also analyzed. In current literature within the area of AsyncSGD, hyperparameter tuning is often done on a single level of parallelism and for a single baseline algorithm. Then the same settings are applied over different levels of parallelism. How hyperparameter settings affect convergence for different fine-grained synchronization schemes is essential to understand the trade-offs between different algorithms fully. Hyperparameter tuning can also be a time-consuming task, even for regular SGD. With parallel SGD and AsyncSGD in particular, this becomes even more challenging, which further motivates the study of hyperparameters' impact on convergence. We can see that the level of parallelism and fine-grained synchronization scheme impact hyperparameter selection. Leashed-SGD is, in general, more stable than the other algorithms, while HOGWILD! tends to converge the fastest. For the lock-based algorithms and HOGWILD!, the batch size should be slightly increased with increasing parallelism, and the step size should be reduced, especially when moving over eight threads.

For a higher level of parallelism, the proposed rw-lock is a competitive option having an improved convergence time compared to the mutex lock and *Leashed-SGD* while maintaining significantly less number of crashed executions than HOGWILD!.

Introducing a backoff scheme to the lock-based algorithms and *Leashed-SGD* shows improvements in convergence for some algorithms and settings. However, further testing and hyperparameter tuning are needed. Investigating how backoff affects convergence is a natural next step. Another step for future works includes extending the study to different ANN architectures and other datasets.

Bibliography

[1] K. Bäckström, I. Walulya, M. Papatriantafilou, and P. Tsigas, "Consistent Lock-free Parallel Stochastic Gradient Descent for Fast and Stable Convergence," in *35th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, (New Orleans, Louisiana USA), 5 2021.

[2] F. Niu, B. Recht, C. Ré, and S. J. Wright, "HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," in *Advances in Neural Information Processing Systems 24* (J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, eds.), pp. 693–701, Curran Associates, Inc., 2011.

[3] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys*, vol. 52, 8 2019.

[4] J. Egger, A. Pepe, C. Gsaxner, and J. Li, "Deep Learning-A first Meta-Survey of selected Reviews across Scientific Disciplines and their Research Impact," *CoRR*, 2020.

[5] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, 12 1943.

[6] F. Rosenblatt, "Perceptron Simulation Experiments," *Proceedings of the IRE*, vol. 48, 3 1960.

[7] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 1969.

[8] M. Arbib, "Review of 'Perceptrons: An Introduction to Computational Geometry' (Minsky, M., and Papert, S.; 1969)," *IEEE Transactions on Information Theory*, vol. 15, 11 1969.

[9] 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 Information Processing Systems* (C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, eds.), vol. 28, pp. 1531–1539, Curran Associates, Inc., 2015.

[10] 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, 12 2016.

[11] 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)*, (Los Angeles, CA, USA), pp. 16–25, IEE, 12 2019.

[12] A. Agarwal and J. C. Duchi, "Distributed Delayed Stochastic Optimization," in *Advances in Neural Information Processing Systems* (J. Shawe-Taylor,

R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, eds.), pp. 873–881, Curran Associates, Inc., 2011.

[13] Z. Huo and H. Huang, "Asynchronous Mini-Batch Gradient Descent with Variance Reduction for Non-Convex Optimization," in *The Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)*, pp. 934–946, 2017.

[14] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu, "Asynchronous Stochastic Gradient Descent with Delay Compensation," in *Proceedings of the 34th International Conference on Machine Learning* (D. Precup and Y. Whye Te, eds.), (International Convention Centre, Sydney, Australia), pp. 4120–4129, PMLR, 8 2017.

[15] Christopher M. Bishop, *Pattern Recognition and Machine Learning*. 233 Spring Street, New York, NY 10013, USA: Springer Science+Business Media, 2006.

[16] Y. Ma, F. Rusu, and M. Torres, "Stochastic gradient descent on modern hardware: Multi-core CPU or GPU? Synchronous or asynchronous?," in *Proceedings - 2019 IEEE 33rd International Parallel and Distributed Processing Symposium, IPDPS 2019*, pp. 1063–1072, Institute of Electrical and Electronics Engineers Inc., 5 2019.

[17] N. Shirish Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. Tak Peter Tang, "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima," in *5th International Conference on Learning Representations ICLR*, (Toulon, France), OpenReview.net, 4 2017.

[18] 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.

[19] D. Masters and C. Luschi, "Revisiting Small Batch Training for Deep Neural Networks," 4 2018.

[20] J. Haochen and S. Sra, "Random shuffling beats SGD after finite epochs," in *Proceedings of the 36th International Conference on Machine Learning* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, pp. 2624–2633, PMLR, 09–15 Jun 2019.

[21] I. Mitliagkas, C. Zhang, S. Hadjis, and C. Re, "Asynchrony begets momentum, with an application to deep learning," in *54th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2016*, pp. 997–1004, Institute of Electrical and Electronics Engineers Inc., 2 2017.

[22] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, vol. 1. 225 Wyman Street, Waltham, MA 02451, USA: Morgan Kaufmann, 1 ed., 2012.

[23] T. Rauber and G. Rünger, *Parallel programming: For multicore and cluster systems*. Springer Berlin Heidelberg, 2 ed., 1 2013.

[24] K. Bäckström, "Adaptiveness and lock-free synchronization in parallel stochastic gradient descent." Lic. thesis, Chalmers University of Technology and Gothenburg University, Gothenburg, 2021.

[25] 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 (IJCAI-16)*, (New York, New York, USA), pp. 2350–2356, AAAI Press, 2016.

[26] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7," *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951, 2018.

[27] H. Zhang, C. J. Hsieh, and V. Akella, "HogWild++: A new mechanism for decentralized asynchronous stochastic gradient descent," in *Proceedings - IEEE International Conference on Data Mining, ICDM*, pp. 629–638, Institute of Electrical and Electronics Engineers Inc., 1 2017.

[28] F. Lopez, E. Chow, S. Tomov, and J. Dongarra, "Asynchronous SGD for DNN training on shared-memory parallel architectures," in *Proceedings - 2020 IEEE 34th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020*, pp. 995–998, Institute of Electrical and Electronics Engineers Inc., 5 2020.

[29] D. Alistarh, C. De Sa, and N. Konstantinov, "The convergence of stochastic gradient descent in asynchronous shared memory," in *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pp. 169–177, Association for Computing Machinery, 7 2018.

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

[31] 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.

[32] Y. LeCun, C. Cortes, and C. J.C. Burges, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/.

A

Appendix A

A.1 Convergence analysis

In this section the results from the parameter search for *Leashed-SGD* with persistence bound, ps=1 and ps=0, are presented. The results are based on ten independent runs. The convergence precision was selected to 5 %-convergence and the maximum time 60 s.

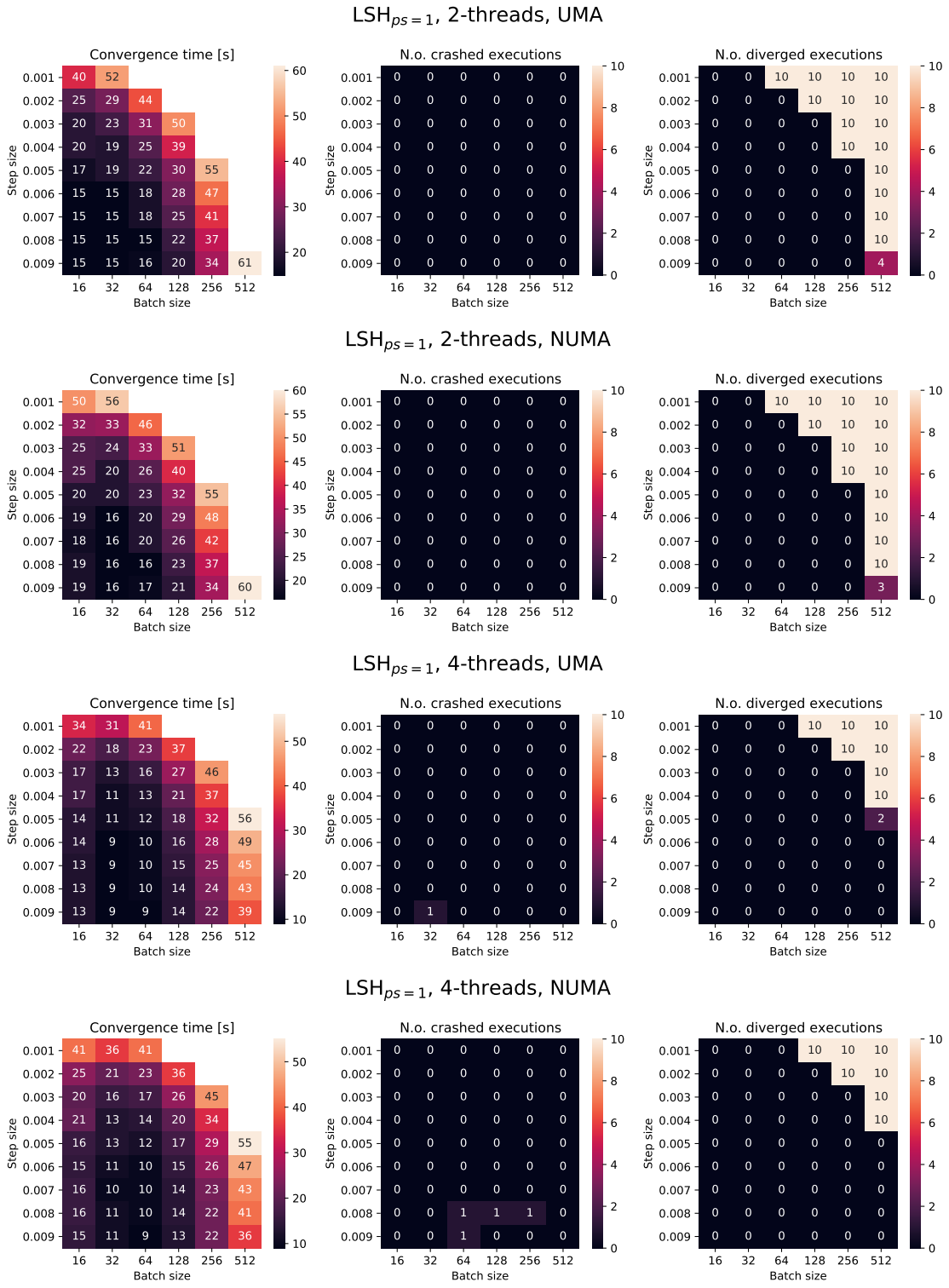

Figure A.1: Heat map of Leashed-SGD with persistence bound 1, time to reach 5 %-convergence (left). Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. For 2 and 4 threads.

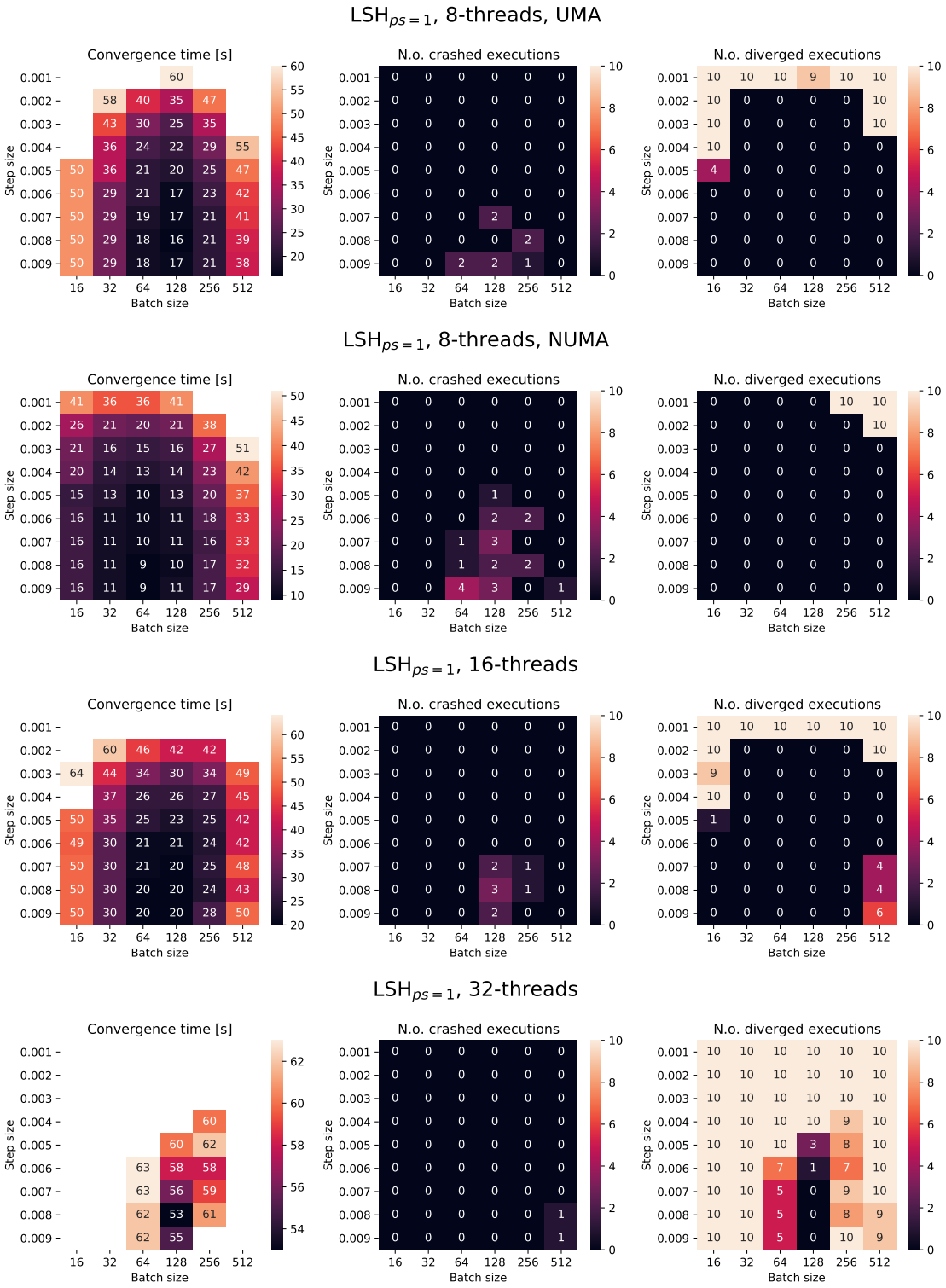

Figure A.2: Heat map of Leashed-SGD with persistence bound 1, time to reach 5 %-convergence (left). Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. For 8, 16 and 32 threads.

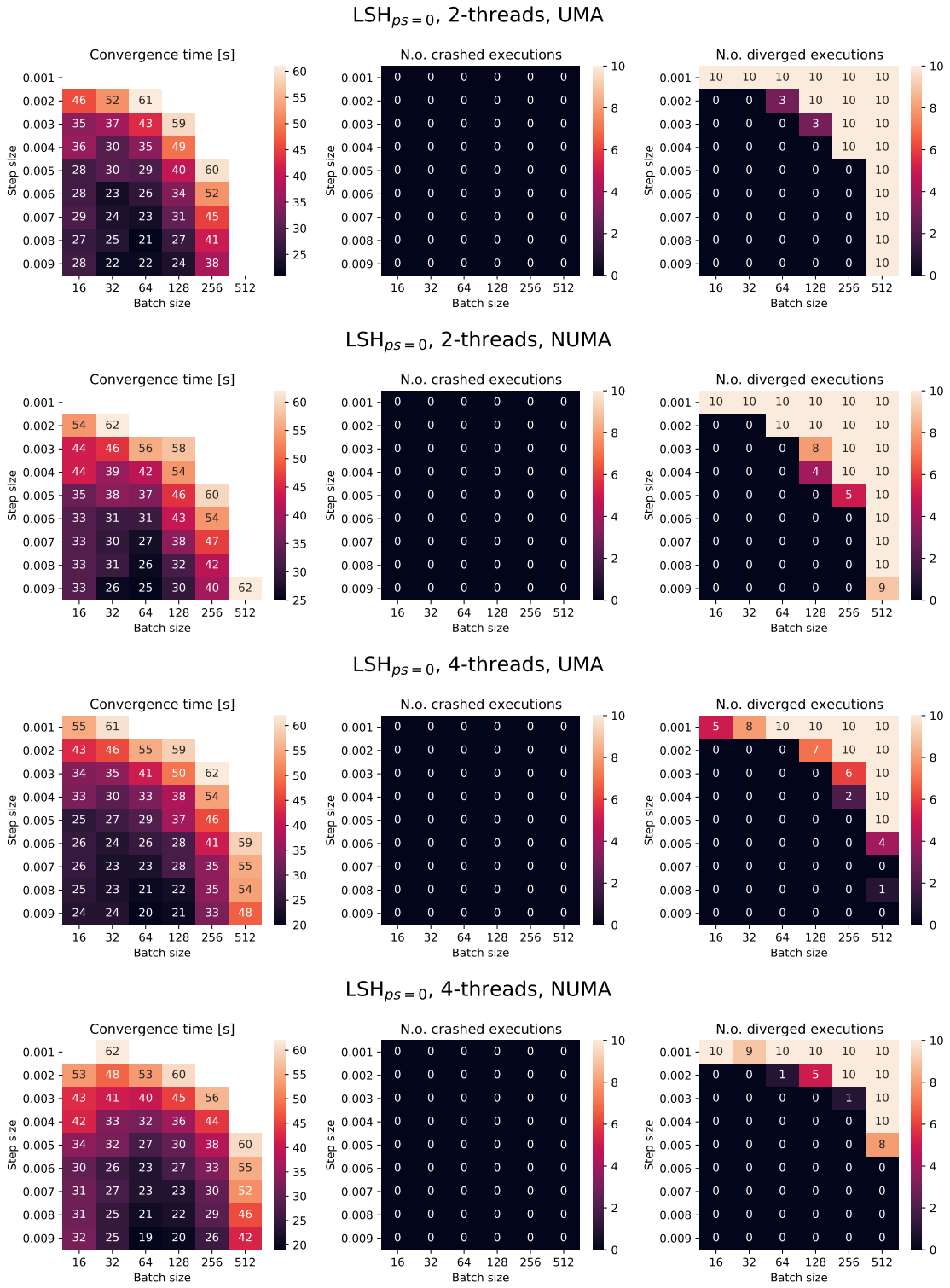

Figure A.3: Heat map of Leashed-SGD with persistence bound 0, time to reach 5 %-convergence (left). Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. For 2 and 4 threads.

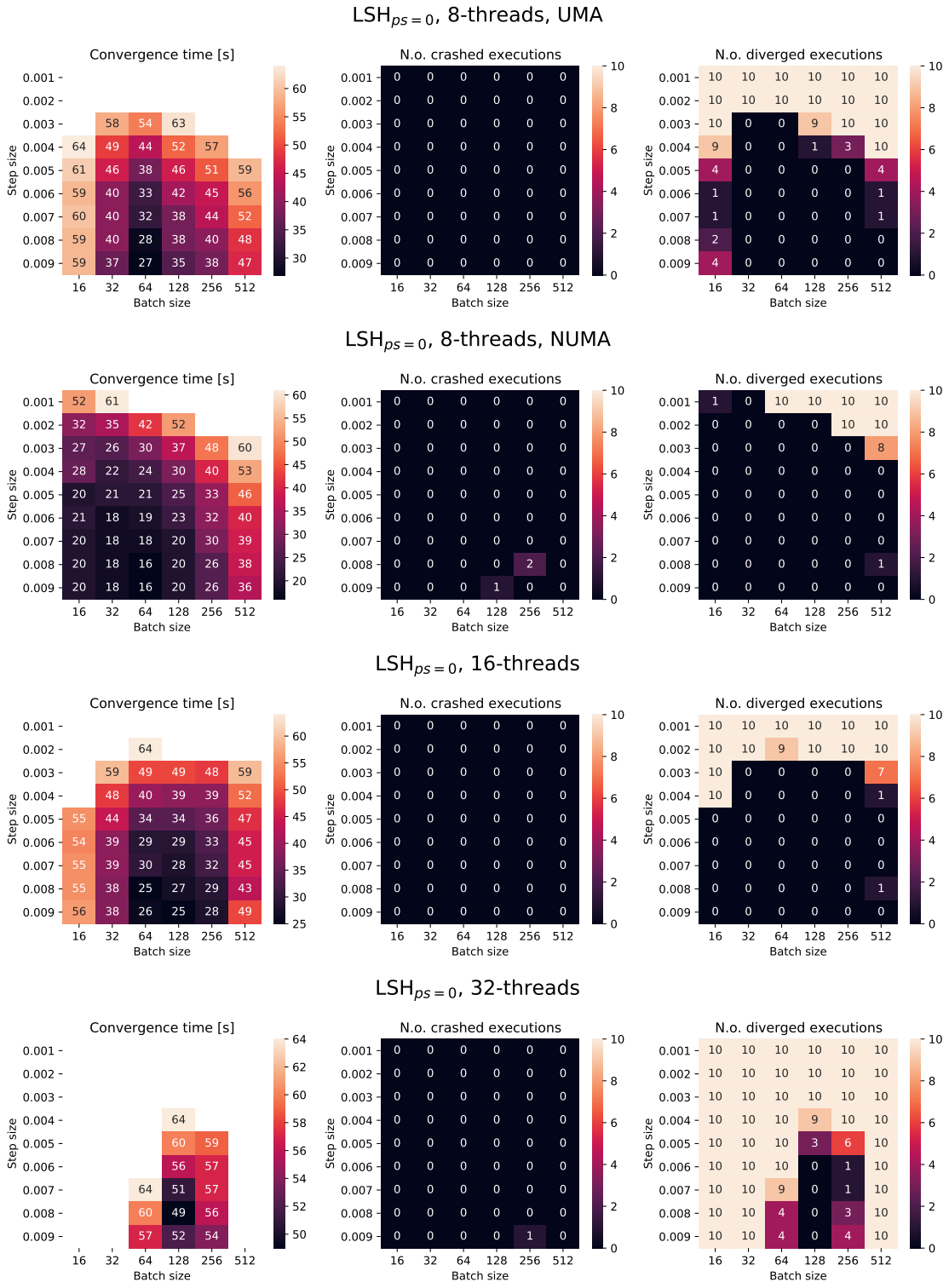

Figure A.4: Heat map of Leashed-SGD with persistence bound 0, time to reach 5 %-convergence (left). Crashed executions (middle) indicate number of crashed executions and number of executions that failed to reach 5 %-convergence are reported as diverged executions (right). Based on ten independent runs. For 8, 16 and 32 threads.