École Polytechnique Fédérale de Lausanne

A Comparative Evaluation of Decentralized Learning Algorithms using Realistic Real-world Traces

by Lubor Budaj

Master Thesis

Approved by the Examining Committee:

prof. Anne-Marie Kermarrec
Thesis Advisor

Erwan Le Merrer
External Expert

Martijn de Vos
Rishi Sharma
Sayan Biswas
Thesis Supervisors

EPFL IC IINFCOM SaCS
Scalable Computing Systems
BC Building, Station 14
CH-1015 Lausanne

July 8, 2024

Acknowledgments

Firstly, I want to thank my supervisors Martijn de Vos, Rishi Sharma and Sayan Biswas for their support and dedication to the project. During our weekly meetings, they were able to nudge me towards the right direction for the project and were up to help when I needed it.

Secondly, I am grateful to the SaCS laboratory as a whole for providing me with the computational resources necessary for the completion of the project.

In addition, I used the thesis template[1] made by HexHive laboratory.

Lausanne, July 8, 2024 Lubor Budaj

[1] https://github.com/HexHive/thesis_template

Abstract

With machine learning (ML) models consisting of billions of parameters and being trained with data spread across multiple machines, there is a need to train the models in a distributed way. Decentralized Learning (DL) offers a scalable solution to this problem which avoids a singular point of failure in terms of a central server. Each participating node incorporates received models into its model, trains it and shares it with other nodes according to a selected DL algorithm while avoiding sharing its private data with any other node. Although the performance of the DL algorithms was previously studied, it was either done in a theoretical setting where each participant is of equal, or similar, performance or the evaluation compares the algorithms only to common baselines such as FedAvg and D-PSGD. However, in a real-life scenario, there can be significant differences in terms of computing or network capabilities of different participating nodes, especially when concerning DL on mobile devices.

We use traces collected from mobile devices, which capture their computing and networking speeds, to emulate a realistic setting of DL. We evaluate the DL algorithms in terms of achieved model quality, convergence speed, and computation and communication efficiencies in a DL simulator. The simulator provides an equal ground for comparison while giving us more control over the execution. We evaluate and explain the performance of the DL algorithms in two common ML tasks in three different settings. We show that in a heterogeneous environment, asynchronous algorithm outperforms their synchronous counterparts. However, there is no one-fit-all algorithm - in different tasks, different algorithms excel. In addition, we present a new asynchronous DL algorithm, Heterogeneous Gossip Learning (HGL), which incorporates node heterogeneity into its decision process. We find that HGL converges to the best model while achieving the highest communication efficiency from all evaluated DL algorithms in the recommendation on the MovieLens data set.

Contents

Acknowledgments 2

Abstract 3

1 Introduction 6

2 Background 9
2.1 Decentralized Learning . . . 9

3 Simulator 11
3.1 Overview . . . 11
3.2 Simulation . . . 11
3.2.1 Events . . . 12
3.2.2 Tasks . . . 12
3.2.3 Bandwidth scheduler . . . 13
3.3 Execution . . . 14
3.3.1 Coordinator . . . 14
3.3.2 Brokers . . . 14
3.4 Traces . . . 14
3.5 Contributions . . . 15
3.5.1 Features . . . 16
3.5.2 Bug fixes . . . 17

4 Algorithms 18
4.1 Decentralized Parallel Stochastic Gradient Descent . . . 18
4.2 Epidemic Learning . . . 19
4.3 Asynchronous Decentralized Parallel Stochastic Gradient Descent . . . 20
4.4 Gossip Learning . . . 22
4.5 Super Gossip Learning . . . 23

5 Evaluation 25
5.1 Hardware . . . 25
5.2 Experiment design . . . 26

5.3 Image classification on IID CIFAR-10 . . . 28
5.4 Image classification on non-IID CIFAR-10 . . . 31
5.5 Recommendation on MovieLens . . . 33
5.6 Discussion . . . 36

6 Heterogeneous Gossip Learning **39**
6.1 Motivation . . . 39
6.2 Algorithm . . . 39
6.3 Evaluation . . . 41

7 Conclusion and Future Work **42**
7.1 Future Work . . . 42

Bibliography **44**

Chapter 1

Introduction

Ever since the popularization of deep learning [11], it has been used for a myriad of tasks. Applications such as text generation use large language models, which can consist of billions of parameters. For example, the largest version of Llama 2 [18] contains seventy billion parameters and it took almost two million GPU hours to train it. It is infeasible to train such large models on a singular machine, thus it is necessary to distribute the workload.

On another note, machine learning (ML) models are not only trained in the server setting. Data from the source, such as smartphones or various Internet of Things accessories, can be directly used to train the global model on the device. Federated Learning [16] (FL) has been a popular approach for training across multiple participating mobile devices. Each participating node has its own private data and contributes to the global model. It receives a model from the central server, performs a gradient update on the model and sends back the updated model without ever sharing its data with the server.

However, this approach suffers from the common problems of centralized architectures. Firstly, it is susceptible to a single point of failure. If the central server were to fail, no training would be possible as the participants would not receive the global model. Secondly, this approach is not scalable beyond the number of nodes the central server can handle. This can be caused either by insufficient network or computing capabilities of the central server. Lastly, there is a privacy concern because a central entity monitors the entire operation. It is shown that the private data of a participating node can leak just through its gradient updates [21].

For these reasons, Decentralized Learning (DL) emerged as an alternative to FL. In DL, there is no central server. Each participating node receives models from its peers and merges them with its own model. Then it performs one or several training steps and shares its new model with other nodes. At any time, whether the node is supposed to share its model, aggregate it with an incoming model or perform a training step is decided by rules, which constitute a DL algorithm.

There are two types of DL algorithms - synchronous and asynchronous ones. The synchronous algorithms wait for all other nodes to finish the training round before proceeding to the next one. On the other hand, the asynchronous algorithms have no concept of rounds. The nodes do not wait for other nodes to finish their training. Instead, they proceed with execution based on the given DL algorithm.

Although the performance of different DL algorithms was previously studied, they are often evaluated only in comparisons with baselines such as FedAvg [16] and D-PSGD [14], but not with each other. For example, based on the existing results, we expect both EL [2] and AD-PSGD [15] to perform better than D-PSGD, but we do not know how they compare to each other. Furthermore, the experiments were often done in a rather theoretical setting where each participant has an equal or similar computing and networking performance. The evaluation being performed on heterogeneous mobile devices was previously rarely considered.

It is not necessary to conduct such an evaluation on mobile devices. Instead, we can utilize standard computer hardware by using traces, which capture the computing and networking speed of different participating mobile devices. For this purpose, SaCS lab developed a Decentralized Learning Simulator [1]. The simulator allows us to conduct a fair evaluation in an equal setting for each algorithm. Furthermore, by utilizing traces the evaluation emulates a real-life scenario. Although there are existing frameworks that serve a similar purpose such as DecentralizePy [3] and FedScale [12], they either do not support the inclusion of traces, or they support only FL algorithms

In this project, we implement different DL algorithms in the DL simulator. We emulate a real-life scenario using traces collected from mobile devices [8, 9]. We evaluate algorithms' performance in two different ML tasks in three settings. Firstly, we examine image classification using the CIFAR-10 [10] data set. We consider both IID and non-IID data distributions among the participating nodes. Secondly, we evaluate a recommendation task on the Movielens [4] data set. Using the algorithms' singularities, we explain the differences in their performances in terms of achieved testing loss or accuracy over time and computing and communication efficiencies.

We will show that the best-performing algorithm in image classification on CIFAR-10 is AD-PSGD in both IID and non-IID settings. It converges fast while avoiding over-fitting thanks to distributing the nodes into active and passive groups. However, in a task where it is easier to find convergence, such as in the recommendation task on the MovieLens data set, it performs poorly due to over-fitting. Moreover, we will show that asynchrony is worthy. There is an asynchronous algorithm, Gossip Learning, which outperforms all synchronous algorithms in every tested scenario. Overall, we will show that there is no one-fit-all algorithm and each algorithm excels in different tasks.

In addition, we present a new asynchronous algorithm, Heterogeneous Gossip Learning (HGL), which compared to the other algorithms utilizes differences in computing speeds among participating nodes. In HGL, the probability of a node receiving a model depends on its computing speed, such that on average each node receives a single model per training step. In the recommendation

task on the MovieLens data set, its resulting models achieve the smallest average testing loss across all the algorithms.

Overall, we make the following contributions:

- We implement the DL algorithms in a common framework - the DL simulator, which provides an equal ground for their evaluation.
- Utilizing the traces, we evaluate the algorithms in a realistic setting of DL on mobile devices in three different experiments.
- We introduce a new asynchronous DL algorithm HGL, which is designed to consider nodes' heterogeneity. It is the best-performing algorithm in the recommendation task on the MovieLens data set.

We divide this work into seven chapters. We start by summarizing Decentralized Learning in chapter 2. In chapter 3 we introduce the simulator which we used to run the algorithms. Here we also describe the traces we used. In chapter 4 we introduce the existing DL algorithms. We introduce our evaluation setup, evaluate the algorithms, and discuss the results in chapter 5. We propose and evaluate HGL in chapter 6. Lastly, we conclude our findings and suggest future work in chapter 7.

Chapter 2

Background

In this chapter, we describe the necessary background to understand this work, namely Decentralized Learning. We assume the reader already has a basic understanding of machine learning (ML).

2.1 Decentralized Learning

Decentralized Learning (DL) is a collaboration of nodes in the training of ML model without sharing their private data. Compared to Federated Learning [16] (FL), DL removes the need for a central server. Furthermore, in DL each node has its own model, which it can share with other nodes. There are several advantages to this approach. Firstly, it increases fault tolerance. The learning process in FL depends on the central server, which coordinates the training. If the central server were to fail, the nodes would not be able to continue training without it. On the other hand, if a node fails in the DL setting, the other nodes can continue training without it. Similarly, the central server is a bottleneck of the whole operation. By the nature of its task, the central server has to send and receive models from all participating nodes. When training large models consisting of billions of parameters, the transfers can take a significant amount of time, possibly causing congestion on the central server. In contrast, DL spreads out the communication volume over the participating nodes. No single node needs to communicate with all others at any particular time, removing the bottleneck and making the learning more scalable. Lastly, with the removal of the central server, we also remove the central monitoring point. There is no node in DL with information on all traffic exchanged during the learning process. This has a positive effect on privacy.

More formally, there are n nodes. Each node i has its private data set D_i. The loss of a model x on a data sample ξ is denoted by $f(x,\xi)$. DL aims to find an assignment of values to model x to minimize the objective function:

$$\min_{x \in R^d} \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}_{\xi \sim D_i} \left[f(x, \xi) \right]$$

A node in DL can either train its model, aggregate the received model with its own or send its model to other nodes. The exact procedure is determined by the DL algorithm used for the learning. There are two types of DL algorithms - synchronous and asynchronous ones.

The synchronous DL algorithms are characterized by a reoccurring synchronization phase after each round of training. If a node finishes the round before others, it waits for all other nodes before proceeding to the next round. The benefit of this approach is that each node will perform an equal number of training rounds, hence its potential contribution to the model is also equal. However, the faster nodes will have to wait for the slower ones, which can be a significant setback in the presence of nodes with highly varied performance. Decentralized Parallel Stochastic Gradient Descent [14] (D-PSGD) is a staple of synchronous algorithms and DL in general. D-PSGD utilizes a static topology, where each node in every round trains and exchanges its model with its neighbours. Afterwards, it averages its own model with the received ones to create its new model for the next round. Epidemic Learning [2] improves on this concept by making the topology dynamic - the topology changes after every round. This helps to spread the model updates faster and increases the convergence speed.

In contrast, the asynchronous DL algorithms do not have a concept of rounds and consequent synchronization steps. In the case of nodes of heterogeneous performance, asynchrony can lead to significantly higher resource utilization by letting the faster nodes perform more training steps instead of waiting. However, the resulting model does not necessarily have to be of higher quality than the one trained using a synchronous algorithm, especially in the context of non-IID data distribution over the nodes. Asynchronous Decentralized Parallel Stochastic Gradient Descent [15] (AD-PSGD) is an asynchronous algorithm, where nodes train, exchange and average their models at the rate independent of other nodes. In Gossip Learning [5, 6] (GL), nodes periodically send their models to a random other node. Whenever a node receives a model, it aggregates the model into its own and performs a training step.

Chapter 3

Simulator

The core component in our project is the Decentralized Learning Simulator [1][1], which has been developed in SaCS lab to evaluate the DL algorithms. In this chapter, we explain the operation of the simulator in two stages - simulation and execution. In addition, we describe the traces we use with the simulation. Lastly, we list our contribution to the simulator project.

3.1 Overview

The main purpose of the simulator is to emulate a DL network consisting of multiple participating nodes. The simulator provides the algorithms with an equal ground, on which we can fairly evaluate them. Its key feature is the support for traces, which determine the computing and networking speeds of the participating nodes, enabling us to simulate a real-life scenario. The simulation gives us more control over the execution, allowing us to perform the execution on hardware independent of the mobile devices from which the traces were collected. The execution of the training is implemented in PyTorch [17] and can run on both CPU and Cuda and on one or multiple machines. We can divide the operation of the simulator into two parts - simulation and execution.

3.2 Simulation

The first stage is the simulation, which is computationally less demanding. Its responsibility is to create a dependence graph containing all tasks, which are going to be executed later.

[1] https://github.com/sacs-epfl/decentralized-learning-simulator

3.2.1 Events

The algorithmic logic is conveyed through an event-based system. Each event is assigned a node, a function, data and time. When a node generates an event, the simulator puts the event into the event queue based on the event's time. The simulator executes events in order. When it is time to execute the given event, the event's function is called with the data contained in the event. The function call can itself result in the scheduling of new events. In this way, the simulator continues to execute events, which generate new events, until the stopping condition, either the time elapsed or the number of rounds, is reached and new events are no longer scheduled. The simulator currently supports the following events: initialization, start of a training step, training finish, start of a data transfer, finish of an outgoing data transfer, incoming model, aggregation, dissemination of a model, performance testing of a current model and starting of a round. However, not all algorithms make use of all event types.

3.2.2 Tasks

The aim of the events is not only to schedule other events but also the tasks. Each task has a function, data, inputs tasks and outputs tasks. The role of the simulator is to put a task into a directed acyclic graph, such that the task follows its input tasks. For instance, in Gossip Learning, when a node receives a model, it will aggregate the incoming model with its own. Aggregation is the merging of two or more models using linear combination, where the weights of the models are determined based on the specific algorithm and sum up to one. We call this process averaging in case of equal weights. In that case, the node will generate an aggregation task with two input models - the node's own model and the received one. These two models have to be the results of some other tasks, which would be put by the simulator as inputs of the newly generated aggregation task. We can see aggregations tasks put in the graph in Figure 3.1. We observe that each aggregation has two input tasks - the received model and the node's current model. Whenever a task is put into the graph, its output tasks have not yet been generated. There are five types of tasks: training, aggregation, testing, gradient computation and gradient update, but most algorithms make use of only the first three. Each of the tasks will call its respective function during the execution phase.

Each event is scheduled at a particular time determined before the scheduling. Consider the start of a training step event. In the most general case, it will create a training task with input being the current model and pass it to the simulator. Then, it will schedule a training finish event. Depending on the algorithm, the start of a training step event can have other side effects. Supposing the current time is t_0 and training takes t_1 time units, the training finish event will be scheduled at time $t_0 + t_1$. The training time is calculated using several factors. Firstly, there is an underlying speed of the node executing the event, which is taken from the traces. Other factors influencing the length of training are batch size and the number of local steps performed during a single training step. On the other hand, we assume that some events, such as aggregation, run in an instant. We reason that the time it

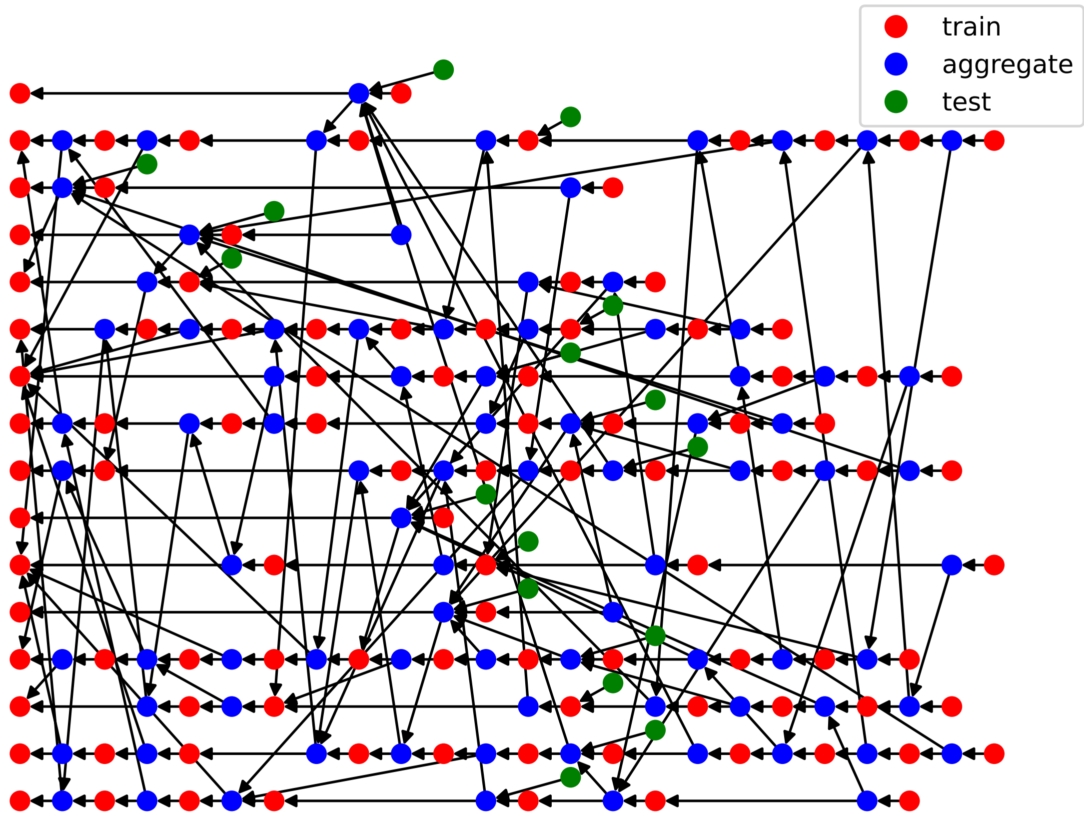

Figure 3.1: Compute graph of gossip learning simulated on 16 nodes. The filled circles represent the tasks. The y coordinate of a task determines the node. The x-axis represents the order. An arrow from a task B to the task A shows that the task B depends on the result of the task A.

would take to make a linear combination is negligible compared to the gradient computation during the training.

3.2.3 Bandwidth scheduler

During an event call, each node can communicate and exchange models with the other nodes. When exchanging models, the total transfer time is determined using a bandwidth scheduler. The bandwidth scheduler tracks incoming and outgoing transfers from the given node. Whenever there is a request for a transfer, based on the available bandwidth of the two nodes involved in the transfer, the bandwidth scheduler determines the maximum possible transfer speed. The overall length of the transfer depends on the transfer speed and model size. On the other hand, if there is no bandwidth available, a transfer request waits in a queue. We assume the nodes have knowledge of the bandwidth capabilities of other nodes and do not attempt to transfer using more bandwidth

than necessary.

3.3 Execution

During the execution phase, the simulator executes all the accumulated tasks. This usually takes significantly longer than the initial simulation. We have already introduced the behaviour of the aggregation task in subsection 3.2.2. The training task calls a function which performs one or several local steps on the node's model. The gradient is computed in a stochastic way from the node's data. The responsibility of a testing task is to call a function to measure the loss and accuracy of the node's model at the given time on the testing data set. In addition, during this phase, we measure different performance-oriented metrics such as memory and CPU usage to help us optimize the execution.

3.3.1 Coordinator

The coordinator is the same entity as the simulator, but in this phase, we will refer to it as the coordinator. The coordinator aims to distribute the workload across the brokers. Once the simulation is over, the coordinator waits for all brokers to connect. Afterwards, the coordinator assigns nodes to brokers and sends them the task graph. The coordinator then determines the starting tasks - the tasks without any inputs - and schedules them on the responsible brokers. Subsequently, the coordinator waits to receive all sink tasks - tasks without any output. When this happens, the coordinator sends a shutdown signal to all brokers and the execution is over.

3.3.2 Brokers

The responsibility of a broker is to schedule tasks on workers. When the coordinator initializes a broker, it spawns one or multiple workers, each representing a single thread running in parallel. Whenever a worker finishes a task, the result of the task is passed to whichever broker needs it. This is determined based on the node the task belongs to. Additionally, if there is a task with all of its inputs already computed, the broker will schedule it on a worker.

3.4 Traces

The key feature of the simulator is the support of traces, which allows us to simulate a real-life scenario of different devices participating in decentralized learning. Each node is assigned a sample from the distribution containing computing and network capabilities. In total, the distribution

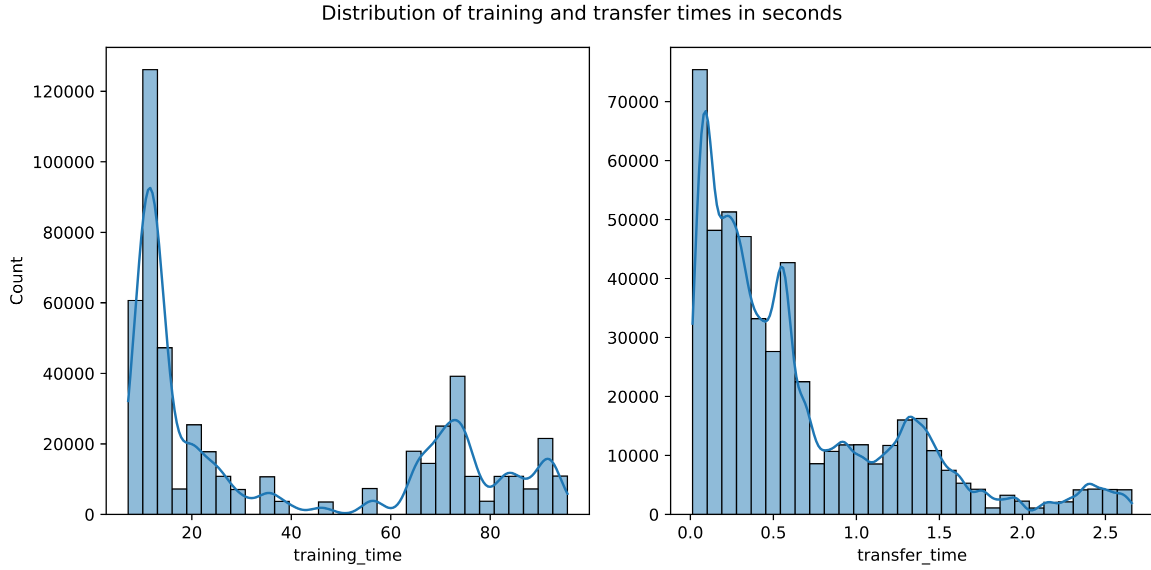

Figure 3.2: Distributions of training and transfer times in seconds for training of GN-LeNet model on CIFAR-10 data set. A single training includes 5 local training steps with a batch size of 32.

contains 500000 samples. We use the traces from FedScale [12], the simulator for Federated Learning. The traces related to the computing efficiency of the mobile devices for the training of deep learning models were collected for AI benchmark [9], and the network transfer speeds by MobiPerf measurements [8].

We can see the distribution of computing and transfer times for the GN-Lenet model on CIFAR-10 data set in Figure 3.2. We see that the capabilities of the mobile devices are vastly different. Especially in the case of the training times, the overall distribution seems to be made of two separate distributions, one for the fast and one for the slow devices. There are very few devices with training speeds in between these two distributions. On the other hand, the distribution of network traces seems to be more stable, but there are still significant differences between the devices. When comparing the two distributions, we notice that the duration of the training seems to be much longer. We expect it to have a larger effect on total execution time.

3.5 Contributions

As part of this project, we have made numerous contributions to the simulator. We first list the added features and then the bug fixes.

3.5.1 Features

- Our main contribution is the implementation of 5 decentralized learning algorithms within the simulator. This includes client and simulation classes. In addition, we also made unit tests for all of these algorithms. We note that our implementation of Epidemic Learning is heavily based on the already existing implementation of D-PSGD in the simulator.
- We implemented abstract asynchronous client and simulation classes, which will make it easier to implement other asynchronous algorithms in future.
- We added elapsed time as a stopping condition and made periodic testing based on the time elapsed. The periodic testing is implemented using a new testing event.
- We introduced a disseminate event, which periodically sends a model to a random other node.
- We added the capability of plotting the compute graph and the resulting loss and accuracy after the execution.
- We implemented an option for aggregation to use custom weights.
- We added tasks and implemented functions to compute the gradient and perform a gradient update. Before, both of these tasks were united within a training task. Now there is an option to separate them if the algorithm requires it.
- We added logging of various measurements, such as the proportion of time spent computing, number of transfers between each two nodes, etc., which are then saved to files. These logs can be utilized later for analysis.
- We introduced an option to use stragglers. Stragglers are the slowest nodes which are further handicapped in their computing speed. We can utilize stragglers to observe the behaviour of an algorithm in presence of extremely slow nodes.
- We made the execution of the coordinator more efficient when receiving a result for a task.
- We added support for the MovieLens data set. We note that our implementation is heavily based on the existing implementation in DecentralizePy [3]. We only made the changes necessary for the class to work in the simulator.
- In addition we made the simulator generate and execute tasks in batches. Instead of the long simulation and execution phase, as described earlier in this section, the simulation stops when a certain amount of tasks generated is reached. Then, these tasks are executed and the simulation phase continues again. The two stages follow each other until there are no tasks to generate. This approach increases memory efficiency for long simulations when there are millions of tasks. We note that this feature is still experimental [2] and has not yet been pushed to the main repository.

[2] https://github.com/fondefjobn/decentralized-learning-simulator/tree/batch_simulator

3.5.2 Bug fixes

- We now make a copy of a model when testing. Otherwise, there was a possibility of a crash on some platforms.
- Before, the name of the task was a hash. Two different tasks could receive the same hash, which would cause the tasks to be incorrectly assigned input and output tasks. We fixed this issue by numbering the tasks instead.
- We now catch a rare concurrency error which can happen when training on a GPU. Otherwise, the exception would result in a crash. We note that this bug was not caused by the simulator but by the other library.
- We fixed a rare race condition in the existing implementation of D-PSGD algorithm, which occurred if a node received a model from the next round at the same microsecond as it performed aggregation in the current round.
- We make sure that the simulator properly exits the execution in a dry run setting. During the dry run, only the simulation part runs without computing the results of the tasks.
- We fixed the partitioners not properly using seed and alpha parameters.
- Brokers now create a dictionary to save the measurements if it does not exist. This would cause a crash if the broker was run from a different directory than the coordinator.
- We fixed the broker sometimes sending an incorrect IP address in its hello message. This error was not caused by the simulator but by the library responsible for fetching the IP address of the network device connected to the internet. We opted to use another library instead.
- We fixed a race condition when multiple brokers attempted to write metrics to the same file. Now there is a separate file for each node. When all tasks are computed, the coordinator merges the files into a single metrics file.
- We fixed momentum not being used in successive training tasks.

Chapter 4

Algorithms

In this chapter, we describe the existing DL algorithms we used in our evaluation. Often, the exact implementation is not clear solely based on the algorithm. This is especially true when a node executes multiple local steps in a row. Hence, where deemed necessary, we provide and explain our implementation decisions of the algorithm in question. Furthermore, we mostly stay true to the original algorithm, however, we make small performance improvements where appropriate. We will first introduce synchronous DL algorithms and then move towards asynchronous ones. In addition to the existing DL algorithms, we introduce Super Gossip Learning, which was previously developed at the SaCS lab.

4.1 Decentralized Parallel Stochastic Gradient Descent

Decentralized Parallel Stochastic Gradient Descent [14] (D-PSGD) is a prime example of a synchronous DL algorithm. In each round. the algorithm performs the two repeating steps. At first, each node updates its local model. Then it averages the updated model with the model of its neighbours from the same round. The complete algorithm can be seen in Figure 4.1.

In our implementation, we assume that the underlying topology is $\log_2 N$ regular, where N is the number of nodes. As the stopping criterion, compared to the pseudo-code in Figure 4.1, we do not use the number of rounds passed but elapsed simulated time. This helps us compare D-PSGD to the asynchronous algorithms, which have no concept of rounds. Nevertheless, it is important to note that our implementation still uses a concept of rounds for variable assignments, e.g. a node in the given round will accept for aggregation only the models from the current round.

The biggest difference in our implementation compared to the original algorithm [14] is that we swapped the gradient update and aggregation steps. The reason for this switch is that in our setup

Algorithm 1: D-PSGD on i_{th} node

Input: initial model $x_{i,0}$, local data set ξ_i, number of rounds K, learning rate γ, nodes are placed in an underlying connected topology, where N_i is the set of neighbours of i

1 **for** *each round k from 0 to* $K-1$ **do**
2 Sample a batch $\xi_{i,k}$ from ξ_i;
3 Compute the gradient $g_{i,k} = \nabla F(x_{i,k}, \xi_{i,k})$;
4 Update the model $x_{i,k+\frac{1}{2}} = x_{i,k} - \gamma \cdot g_{i,k}$;
5 Send the model $x_{i,k+\frac{1}{2}}$ to all neighbours $n \in N_i$;
6 Wait to receive model $x_{n,k+\frac{1}{2}}$ from all neighbours $n \in N_i$;
7 Average the models from the neighbouring nodes $x_{i,k+1} = \frac{1}{|N_i|+1}(\sum_{n \in N_i} x_{n,k+\frac{1}{2}} + x_{i,k+\frac{1}{2}})$;

Figure 4.1: D-PSGD algorithm

we perform multiple local steps, meaning that in a single round, we update the model multiple times, but perform only a single aggregation. For each of the multiple model updates, we first calculate the gradient, therefore these two steps have to follow each other. Another trick is that a node does not have to wait for all other nodes if it wants to proceed to the next round. It is enough to wait only for all of its neighbours, allowing the node to proceed with aggregation and reach the next round. However, in practice, this method results only in a small performance benefit, where the fast nodes, depending on the topology, can be a few rounds ahead. In the case of $\log_2 N$ regular topology, it is at most one or two rounds.

4.2 Epidemic Learning

In principle, Epidemic Learning [2] (EL) functions similarly to D-PSGD. The twist is that the underlying topology is not static but changes each round. As the nodes communicate with different nodes in every round, the model updates are spread faster, which leads to theoretically faster convergence [2]. The complete algorithm can be seen in Figure 4.2. There are two ways of generating a new topology - EL-Oracle and EL-Local.

EL-Oracle generates a s-regular topology, where s is a hyperparameter. As suggested in the original paper [2], we set $s = \log_2 N$. This is the same approach we used in our D-PSGD implementation, but EL-Oracle repeats the topology generation every round.

On the other hand, in EL-Local each node, each round independently samples s other nodes, which will receive its model. Hence, different nodes can receive a different number of models in the same round or even no models at all. It is shown that this approach has the same convergence speed

Algorithm 2: EL on i_{th} node

Input: initial model $x_{i,0}$, local data set ξ_i, number of rounds K, learning rate γ, set of all nodes S, sample size s

1 **for** *each round k from 0 to* $K-1$ **do**

2 Sample a batch $\xi_{i,k}$ from ξ_i;

3 Compute the gradient $g_{i,k} = \nabla F(x_{i,k}, \xi_{i,k})$;

4 Update the model $x_{i,k+\frac{1}{2}} = x_{i,k} - \gamma \cdot g_{i,k}$;

5 Sample s nodes from $S \setminus \{i\}$ according to EL-Oracle or EL-Local;

6 Send $x_{i,k+\frac{1}{2}}$ to all sampled nodes;

7 Wait to receive $|N_{i,k}|$ models $x_{n,k+\frac{1}{2}}$, where $N_{i,k}$ is a set of nodes sending its model to i at round k;

8 Average the received models $x_{i,k+1} = \frac{1}{|N_{i,k}|+1}(\sum_{n \in N_{i,k}} x_{n,k+\frac{1}{2}} + x_{i,k+\frac{1}{2}})$;

Figure 4.2: Epidemic Learning algorithm

as EL-Oracle [2]. There are two main ways to implement EL-Local. Either, a node knows in each round how many models it will receive, e.g. thanks to a common seed, and it will wait to receive all the expected models, before proceeding with the aggregation. Alternatively, there is a specified time limit of how long the node will wait to receive the models. Afterwards, the node will aggregate regardless of whether there is some other node still sending its model to the node in the given round. We opted for the first approach, which is better suited for our evaluation scenario with highly variant traces and no assumed failures.

4.3 Asynchronous Decentralized Parallel Stochastic Gradient Descent

As the first asynchronous algorithm, we introduce Asynchronous Decentralized Parallel Stochastic Gradient Descent [15] (AD-PSGD). The distinctive feature of the algorithm is that whenever a node receives a model, it replies with its own model. After the subsequent averaging, the two nodes participating in the exchange will have the same model. The complete algorithm can be found in Figure 4.3.

Whenever a node initiates the model exchange, it waits until it receives a reply. However, if all nodes were to initiate the exchange at the same time, it would cause a deadlock with all the nodes stuck waiting to receive a model from the other party. Therefore, AD-PSGD avoids the deadlock by dividing the nodes into two groups - the active and the passive nodes. The active nodes initiate the communication and can send their model only to a passive node. A passive node cannot initiate the communication. When a passive node receives the model, it immediately replies with its own model and averages the received model with its own. The division effectively creates a bipartite

Algorithm 3: AD-PSGD on i_{th} active node

Input: initial model x_i, local data set ξ_i, learning rate γ, set of all passive nodes S

1 **repeat**
2 Sample a batch $\hat{\xi}_i$ from ξ_i;
3 Compute the gradient $g_i = \nabla F(x_i, \hat{\xi}_i)$;
4 Update the model $x_i = x_i - \gamma \cdot g_i$;
5 Sample a node j from S;
6 Send x_i to node j;
7 Wait to receive x_j from j;
8 Average the received model $x_i = \frac{x_i + x_j}{2}$;
9 **until** *the running time elapses*;

Algorithm 4: AD-PSGD on j_{th} passive node

Input: initial model x_i, local data set ξ_i, learning rate γ

1 **repeat**
2 Sample a batch $\hat{\xi}_j$ from ξ_j;
3 Compute the gradient $g_j = \nabla F(x_j, \hat{\xi}_j)$;
4 Update the model $x_j = \hat{x}_j - \gamma \cdot g_j$ Note that $\hat{x}_j$ might be different than x_j;
5 Wait to receive x_i from any active node i;
6 Send x_j to node i;
7 Average the received model $x_j = \frac{x_i + x_j}{2}$;
8 **until** *the running time elapses*;

Figure 4.3: AD-PSGD algorithm

communication topology.

Compared to the algorithm in Figure 4.3, the implementation of AD-PSGD is more nuanced. Due to the waiting steps of both the active and the passive nodes, there could be a significant waiting period, hindering the benefits of asynchrony. The authors of AD-PSGD solve this problem by making the passive peer reply back and average immediately when it receives the model. However, this causes another problem. What if the passive node receives a model at the same time as it is computing the gradient to update its model? In such a case, the node continues to compute the gradient. At the same time, it averages the received model with its own and puts the result to $\hat{x}_j$. Once the gradient computation is finished, it will apply the gradient update to the averaged model $\hat{x}_j$. To implement this behaviour we created new types of tasks in the simulator - gradient computation and gradient update. This is the only algorithm that utilizes them.

The last thing to solve is the waiting behaviour of the passive nodes. Since a passive node replies and averages whenever it receives a model, the waiting and averaging could be removed from the main execution loop. However, in such a case the node could perform many training steps, without

ever receiving a model, which is not the intention of the algorithm. We opted for a hybrid approach. Whenever a passive node receives a model, it increments its training budget. Contrarily, when it trains, it decrements the budget. If a node has already used up its budget, it will wait with training until it receives a model. Otherwise, it will continue to the next iteration without waiting.

4.4 Gossip Learning

Another asynchronous algorithm is Gossip Learning [5, 6] (GL), which adopts the gossip protocol to decentralized learning. The nodes spread their model by periodic gossiping to a random other node. The basic idea of the algorithm is simple - each node periodically sends its model to a random other node. Whenever it receives a model, it aggregates it with its own and performs one training step. You can observe the complete algorithm in Figure 4.4.

GL uses an aged-based aggregation function. The function assigns weights to the aggregated models based on the nodes' ages. When a node aggregates, its new age is set to the maximum age of the two nodes involved. Furthermore, when a node trains, it increments its age. In this way, the models that are the results of many training steps will have a higher weight during future aggregations. This should result in faster convergence by avoiding outdated models from the nodes, which have models not updated as many times.

The algorithm for GL neglects an important implementation detail - what should happen when a node receives a model when it has just received another one and is currently in training? It would not make sense to start another training when one is already running. We could store every received model and aggregate them all when the training finishes, but that is the idea behind the Super Gossip Learning algorithm, which we will introduce later. However, completely ignoring the received models would be a very wasteful option. We chose a middle-ground approach. We save the received model for later, but at any time keep at most one model waiting for the end of the current training. We found that this small improvement results in much higher resource utilization of GL.

During our testing, we encountered another problem. If two nodes were to send their model to the same node, the receiving node would always aggregate the one it received first, depending on the possible transfer speeds. Then, the node would be busy with training and not aggregate the second model. We thought this would be an unfair treatment for one of the nodes, therefore we added a random delay between 0 and α second to each node before it starts its periodic sending. In this way, the sending period of every node has a different shift and on average each node's model has an equal chance to be aggregated when sent to another node. Another problem is that the algorithm has a hyperparameter α, which has to be tuned. Since our primary focus is to maximize the performance given the available resources, we set α to the smallest period, the networks of the nodes can handle. As we will see in chapter 5, this results in poor communication efficiency of GL. However, the higher values of α would result in poor computing utilization and therefore slower

Algorithm 5: GL on i_{th} active node

Input: initial model x_i, local data set ξ_i, learning rate γ, set of all nodes S, period α

```
1  repeat
2  |  Wait α seconds;
3  |  Sample a node j from S \ {i};
4  |  Send x_i to node j;
5  until the running time elapses;

6  Procedure OnReceiveModel(x_j):
7  |  Aggregate x_i and x_j and put the result into x_i;
8  |  Sample a batch ξ̂_i from ξ_i;
9  |  Compute the gradient g_i = ∇F(x_i, ξ̂_i);
10 |  Update the model x_i = x_i − γ · g_i;

11 Procedure Aggregate(x_i, a_i, x_j, a_j):
12 |  c = x_j / (x_i + x_j);
13 |  a = max(a_i, a_j) + 1;
14 |  x_i = (1 − c) · x_i + c · x_j;
```

Figure 4.4: Gossip Learning algorithm

convergence. Another problem with this hyperparameter is that if the network conditions were to change during the execution, the selected value of α might no longer be suitable.

4.5 Super Gossip Learning

Super Gossip Learning [19] (Super GL) is an asynchronous algorithm, which was previously internally developed in the SaCS lab. It aims to correct the deficiencies of GL. Super GL removes the periodic sending and the hyperparameter connected to it. Instead, the sending is performed in the main loop. Super GL introduces the concept of a queue into GL. Whenever a node receives a model, it puts it into the queue. When it is time for aggregation, all models from the queue will be aggregated with the local model and the queue is cleared. Super GL uses the same aged-based aggregation scheme as GL. The complete algorithm can be seen in Figure 4.5.

Super GL introduces a set of implementation questions of its own. Firstly, a node may receive a newer model from the node, which has already put its older model in the queue. In that case, the newer model essentially makes the previous one redundant, therefore we replace the old one with the newer one in the queue. Another issue is in the main loop. If it is time to aggregate and a node has not received any model after the previous aggregation, its queue is empty. In such a case, it is

Algorithm 6: Super GL on i_{th} active node

Input: initial model x_i, local data set ξ_i, learning rate γ, set of all nodes S, an empty queue q_i

1 **repeat**
2 Sample a batch $\hat{\xi}_i$ from ξ_i;
3 Compute the gradient $g_i = \nabla F(x_i, \hat{\xi}_i)$;
4 Update the model $x_i = x_i - \gamma \cdot g_i$;
5 Aggregate x_i with all models in the queue q_i;
6 Sample a node j from $S \setminus \{i\}$;
7 Send x_i to node j;
8 **until** *the running time elapses*;

9 **Procedure** `OnReceiveModel(`x_j`)`**:**
10 put x_j into the queue q_i;

11 **Procedure** `Aggregate(`x_i, a_i, x_j, a_j`)`**:**
12 $c = \frac{x_j}{x_i + x_j}$;
13 $a = max(a_i, a_j) + 1$;
14 $x_i = (1 - c) \cdot x_i + c \cdot x_j$;

Figure 4.5: Super Gossip Learning algorithm

not clear whether the node should wait to receive a model or continue without any aggregation. We tested both approaches. The results with the waiting option were significantly worse. The waiting approach results in the algorithm barely performing any learning, therefore we opted for the non-waiting implementation. The cause of this issue is that the model sharing is conducted in the main loop. However, waiting for aggregation would also happen in the main loop. Therefore, if a node is waiting, it cannot share its model with other nodes. Eventually, only one node can compute, while all others are waiting for the node to send them a model. However, the node can send the model only to one of the waiting nodes before it is waiting itself.

Chapter 5

Evaluation

In this chapter, we evaluate the five existing algorithms introduced in chapter 4. To avoid plot repetition, the plots in this chapter also include the results related to Heterogeneous Gossip Learning, but we will discuss them only in chapter 6. In total, we evaluated the algorithms in three different experiments involving two common ML tasks. At first, we consider an image classification task on the CIFAR-10 [10] data set. We consider both IID and non-IID class distributions over the nodes. Secondly, we evaluate the algorithms in the recommendation task on the MovieLens [4] data set. In this chapter, we first describe our evaluation setup in terms of hardware and the experiment design. Then we show the results of our evaluation.

5.1 Hardware

We conducted the experiments on several machines. Firstly, there are six servers, which were assigned to us by the SaCS lab. Five of them are equipped with 64GB of memory and Intel Xeon E-2288G. The remaining one has 128GB of memory and Intel Xeon CPU E5-2630 v3. Each of these CPU consist of eight cores. As explained in the chapter 3, the simulation would run only on a single machine, which runs the coordinator. All other machines would run a broker, which connects to the coordinator and executes the tasks. During our experiments, we found that to maximize the performance it is not beneficial to utilize all the available threads. Since there could be other processes running on the server, we do not spawn as many workers as there are threads available on the CPU. Instead, we delegate only six threads for this purpose. If we were to spawn more workers, there would be too much context switching, hindering the performance. Despite us conducting the experiments on six servers, the training of large neural networks still took a significant amount of time. The reason is that the training was done only on a CPU since the servers were not equipped with a GPU.

One of the biggest challenges we encountered was the significant running time of the experiments. We used two additional laptops equipped with GPUs Nvidia GeForce RTX 3060 and RTX 4060 to speed up the execution. The first of the two was used only semi-regularly because it served us as our primary development device. The usage of GPUs significantly shortened the running time of the experiments. In the end, the laptop with RTX 4060 was responsible for the majority of experiments, despite having the same running time as the servers.

5.2 Experiment design

When designing the experiments, we aimed to capture different scenarios where DL can be used. We initially planned to conduct experiments with over a hundred participating nodes, however, due to the limitations in computing power, we scaled it down to 32 participating nodes. To reduce frequent communication between the nodes, each node performs five local steps whenever it trains. Moreover, to speed up the convergence we use a momentum of 0.9. We optimize the models using stochastic gradient descent with a batch size of 32 data points. In total, we try six different learning rates, namely 0.001, 0.005, 0.01, 0.05, 0.1 and 0.25 and for each experiment and algorithm we show the best-performing one. We run the experiments with two different seeds and report the average. However, we did not advance the learning rates, which performed very poorly with the first seed, into the experiment with a second seed.

We evaluated the algorithms on two data sets - CIFAR 10 [10] and MovieLens [4]. We engage in an image classification task in the experiment involving the CIFAR-10 data set, CIFAR-10 contains sixty thousand tiny colour images of size 32x32. Images are divided into ten classes, each class having 6000 samples. 50000 images are reserved for the training set and 10000 for the test set.

During the execution, we allocate the images to the nodes using two different distributions. Firstly, there is the IID distribution, where a node has in its data set each image with equal probability. With this distribution, each node has the same number of images. Secondly, in the non-IID case, the samples are distributed according to the Dirichlet distribution with $\alpha = 0.1$. This simulates a real-life scenario, where different nodes specialize in different classes. The distribution according to one seed is shown in Figure 5.1. We can see that most nodes contain significantly more samples of one or two classes than the images of other classes. Due to the nature of the distribution, there is a large variety of number of images per node. For instance, in Figure 5.1 node 9 has only 96 images, while node 24 over 5000. As the model, we use GN-LeNet [7], which is inspired by the original LeNet [13] network. GN-LeNet uses group normalization layers [20] to tackle the problems of other normalization techniques when facing a non-IID class distribution. Since we engage in a classification task, as the loss function we use cross-entropy. We simulate the execution of each algorithm for over 80 hours.

Secondly, we run a recommendation task with the MovieLens data set. The MovieLens data

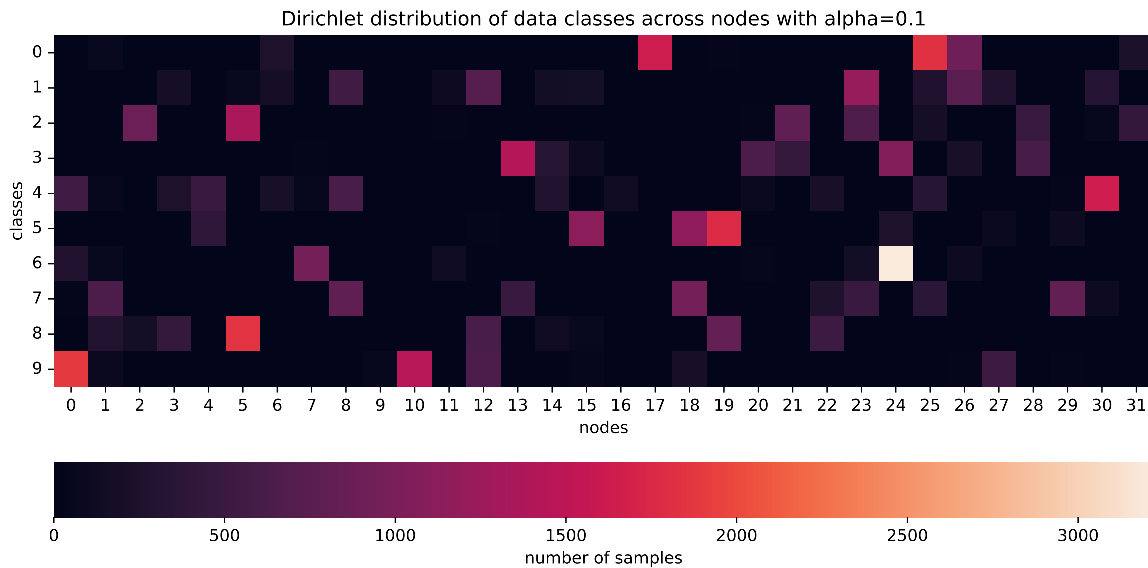

Figure 5.1: Dirichlet distribution of data classes of CIFAR-10 data set across thirty-two nodes with $\alpha = 0.1$.

set consists of 100000 ratings of movies on a scale from 0.5 to 5. In total, there are 610 users, who rated 9724 movies. We distribute the users across the nodes. Each node is responsible for an equal amount of users but some nodes may have an extra user due to a reminder. Since different users rated a different number of movies, the number of ratings is not equal across the nodes. However, the differences are not as extreme as in the case of non-IID distribution for CIFAR-10. For each user, we allocate 30% of its ratings as the test set. In this experiment, as a model, we use a matrix factorization and mean squared error as the loss function. We run the recommendation task for almost 140 simulated hours.

In our evaluation we aim to answer the following research questions:

- Do the asynchronous algorithms perform better than the synchronous ones in the setting with unequally performing nodes?
- Which algorithms perform well at which setting? Is there an algorithm performing the best in every setting or are different algorithms suitable for different tasks? Why?
- Are the algorithms with the best performance also the most efficient in resource utilization?

5.3 Image classification on IID CIFAR-10

We will first evaluate the algorithms' performance and then the efficiency. In Figure 5.2 there is testing accuracy over time of different algorithms. We observe that initially, Super GL converges the fastest, but at approximately 15 hours mark Super GL is overtaken by AD-PSGD, which keeps the lead until the end. We explain the initial fast convergence of Super GL using the heterogeneity of traces. We have seen in Figure 3.2 that both the computing and networking performance of different nodes can be vastly different. In Super GL, the faster nodes send more models to other nodes than the slow ones. This is a direct result of the Super GL algorithm, where the number of sent models is directly related to the number of times a node trains, which depends on its computing speed. We can see a visualization of this behaviour in Figure 5.11. The heatmaps show the number of times a node utilizes a model from other nodes in its aggregations. In the Super GL heatmap, we observe a horizontal pattern, which indicates that a model of a node has an equal probability of being utilized by any other node. However, different colours of the rows show that models from nodes are utilized at different rates. As a result, the slow nodes converge faster because they receive on average more models from faster nodes, which were trained more times. Although there is a faster convergence, this behaviour has a detrimental effect. The consequence is over-fitting towards the data held by the faster nodes.

The worst performers are synchronous algorithms D-PSGD and EL-oracle. This is not a surprise. Due to heterogeneity in computing and network speeds, faster nodes have to wait a significant amount of time for the slow ones. More significantly, the speed of the slowest node determines the speed of the entire learning. All other nodes have to wait for the slowest one to finish training before progressing to the next round. Hence, if even slower nodes were present, these algorithms would perform even worse.

GL converges slower than Super GL, but over time it manages to equal it. In GL, every node sends models at an equal rate, therefore GL does not suffer from the same type of over-fitting as Super GL. This can again be seen in Figure 5.11. The aggregation heatmap of GL has a vertical pattern, which indicates that for any node, the probability of aggregating a model from another node is uniform across all present nodes. However, different nodes perform a different number of aggregations, which is determined by their computing speed. This is more desirable behaviour.

Choosing a single best-performing learning rate can be difficult when we consider multiple optimization criteria, namely the convergence speed and the best-achieved testing accuracy. Therefore, we measure the time required to reach a certain testing accuracy target. For each target, we show the earliest time the target is reached with any learning rate. Therefore, different learning rates may be used to reach different targets, even when considering the same algorithm. We chose four target testing accuracies between 70% and 82%.

In Figure 5.3 we see the results. All asynchronous algorithms reach at least a testing accuracy of 80%, while the synchronous one can reach only the 75% target. We note that the algorithms could

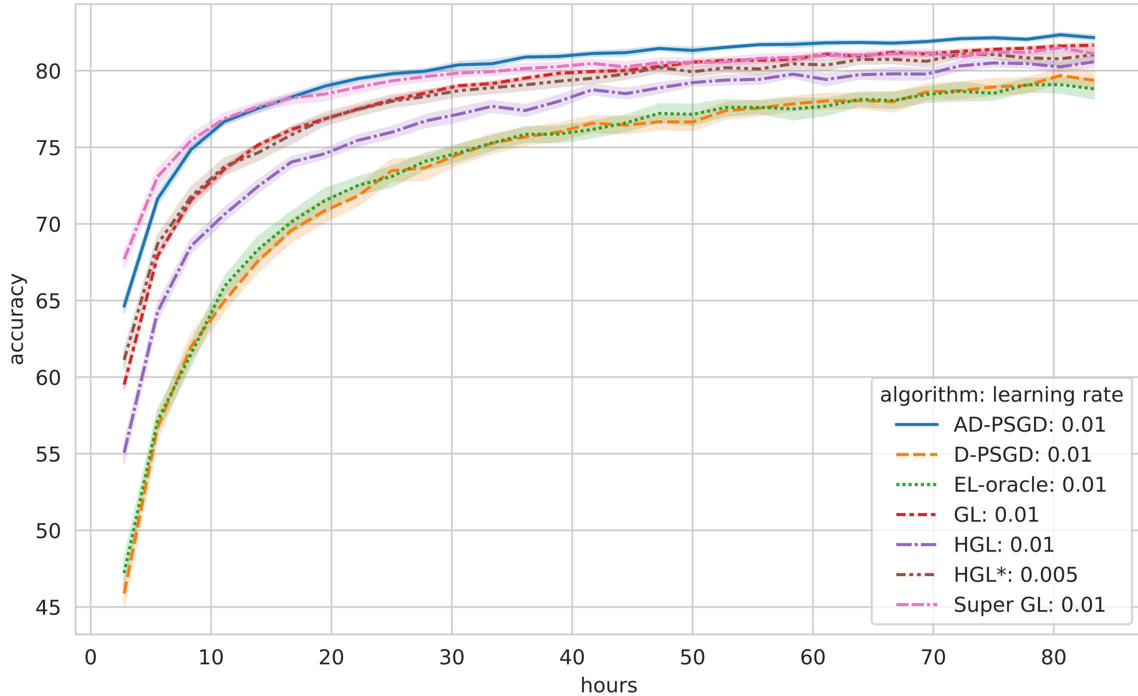

Figure 5.2: Testing accuracy reached over the elapsed simulated time in image classification on the CIFAR-10 data set with IID class distribution on 32 nodes. The error area marks the standard deviation of the testing accuracy of different nodes. For each algorithm, we selected the best-performing learning rate.

reach even higher targets if they were left to run for a longer amount of time. Interestingly, the time to reach 75% target is approximately double the time to reach 70%, while the time to reach 80% is approximately triple the time to reach 75% for all algorithms except Super GL. As we explained earlier Super GL initially converges very fast, but then slows down due to over-fitting. As the only algorithm, AD-PSGD manages to reach the 82% target.

We believe AD-PSGD performs the best due to the way it tackles node-performance heterogeneity. As we already mentioned, Super GL favours fast nodes. The fast nodes send more models than the slow nodes, which will eventually lead to over-fitting. On the other hand, in GL every node sends an equal amount of models, but the utilization of received models depends on the node's computing speed. This leads to a slower convergence as even the nodes who trained their model only a few times have an equal chance of participating in other nodes' models. AD-PSGD mixes these two approaches thanks to its division of the nodes into active and passive groups. Similarly to Super-GL, the model sending rate of an active node depends on its performance capabilities. Assume a fast node is the active group. Thanks to the faster computing and transfer speeds, it will send more models than the slow active nodes. This helps AD-PSGD converge faster because the nodes with models trained more times have a higher sending rate. On the other, if a fast node is

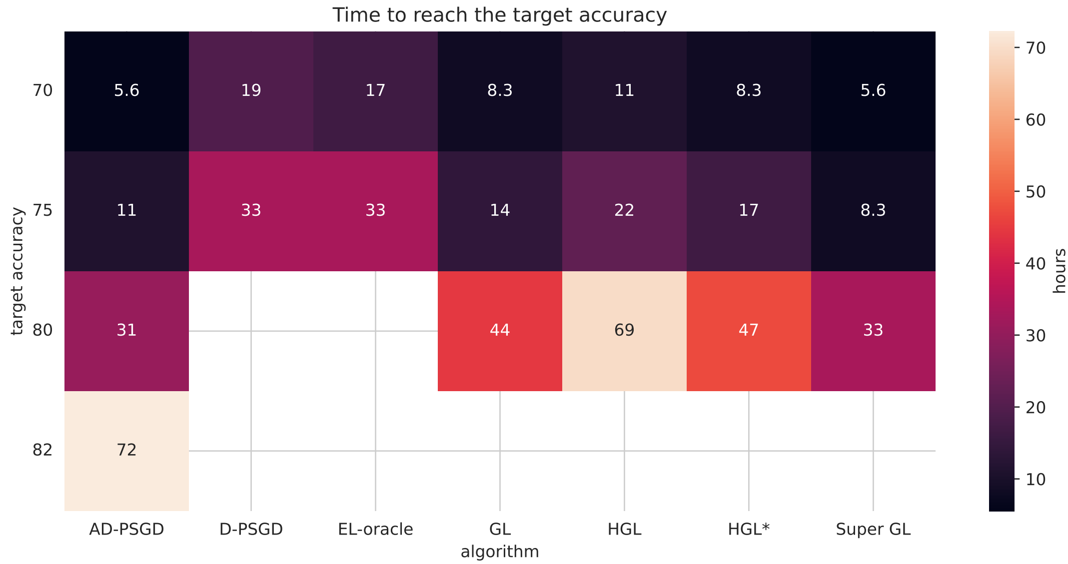

Figure 5.3: Time required in hours to reach the mean target testing accuracy in image classification on the CIFAR-10 data set with IID class distribution on 32 nodes. The missing values indicate that the accuracy was not reached in the given time frame. For each combination of algorithm and target, we selected the best-performing learning rate.

passive, it will spread its model at an equal rate to the other passive nodes. A passive node sends its model to another node only when it receives a model and the probability of any passive node receiving a model is equal for all passive nodes. Unlike in Super GL, a passive node's sending rate is not dependent on its performance capabilities. This helps to limit the effect of over-fitting due to the fast nodes spreading more models while keeping the benefit of a faster convergence.

We can see the model utilization heatmap of AD-PSGD in Figure 5.11. We can observe multiple patterns. Firstly, the heatmap is reflected along the $x = y$ axis (note that the y axis goes from top to bottom). This is caused by the passive nodes only sending their models as a reply to the received one. Secondly, we see that not all nodes communicate with all the other nodes. In AD-PSGD, only communication between an active and passive node is possible. Lastly, we can observe that some nodes have a vertical pattern but others have a horizontal one. The type of the pattern depends on whether a node is active or passive. Active ones have a horizontal pattern because they send at a rate, which depends on their computing and network speeds, similar to Super GL. Passive nodes have a vertical pattern - they receive at an equal rate no matter their performance.

Although our primary objective is to study the algorithms' performance given the available resources, we also examine the efficiency in resource utilization. In Figure 5.4 we see the testing accuracy over the computation and communication cost. Observing the computing efficiency curves, perhaps the most surprising result is that the synchronous algorithms are less efficient, despite computing for roughly only 30% of the total time. The computing efficiency curve of AD-

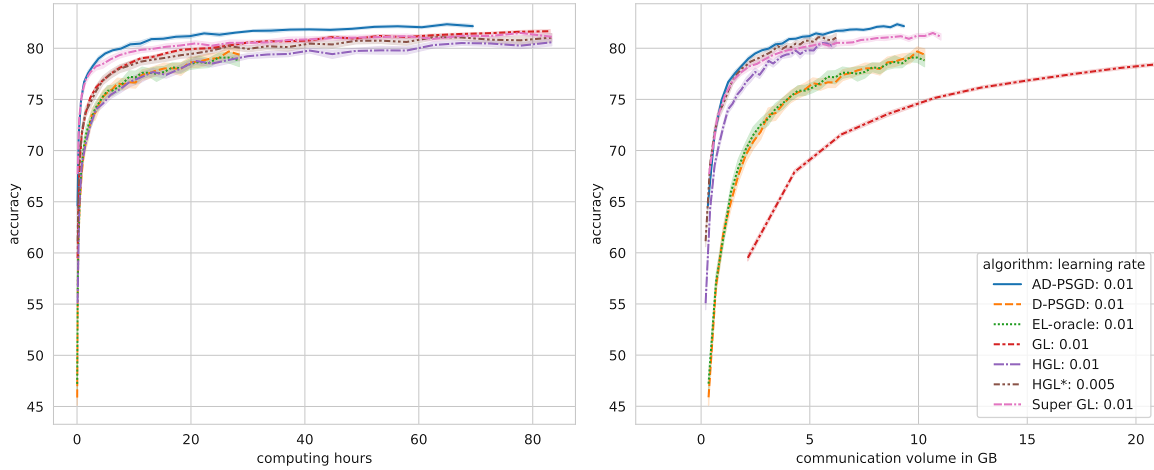

Figure 5.4: In the left plot, there is testing accuracy reached over the hours spent computing In the right plot, there is testing accuracy over the communication volume in GB. The plots are the result of image classification on the CIFAR-10 data set with IID class distribution on 32 nodes. The error area marks the standard deviation of the testing accuracy of different nodes. For each algorithm, we selected the best-performing learning rate.

PSGD shows not only that it is the most efficient but also that it computes only for approximately 85% of the time. Other asynchronous algorithms compute at all times. This makes the performance results of AD-PSGD even more impressive.

Observing the communication efficiency curves we notice that GL is very inefficient. As we mentioned earlier, this is caused by our tuning of the hyperparameter α to the smallest value the network could handle. This leads to higher performance, which we primarily optimized, but at the cost of heavy communication. This time we expect the synchronous algorithm to be less efficient because a node sends its model to multiple other nodes in each round. In this metric, AD-PSGD is again the most efficient algorithm.

5.4 Image classification on non-IID CIFAR-10

The setting of the non-IID experiment is identical to the IID version, except for the data distribution. We describe the data distribution in section 5.2.

In Figure 5.5, we observe similar trends compared to the IID setting. Super GL performs the best early on but is eventually overtaken by AD-PSGD, which achieves the highest testing accuracy. Secondly, again the synchronous algorithms perform the worst. We can justify these claims with the same reasoning as in the case of IID distribution. GL also perform similarly relative to the other algorithms.

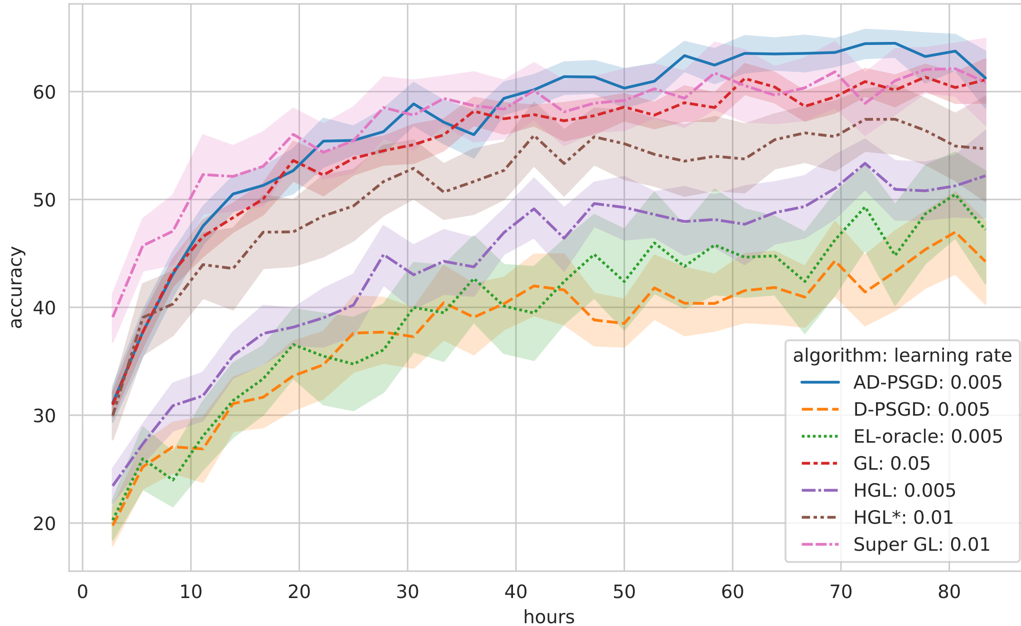

Figure 5.5: Testing accuracy reached over the elapsed simulated time in image classification on the CIFAR-10 data set with non-IID data distribution on 32 nodes. The error area marks the standard deviation of the testing accuracy of different nodes. For each algorithm, we selected the best-performing learning rate.

However, with non-IID data distribution, the models of different nodes are significantly different. This is shown by the large error bars in achieved testing accuracy. Furthermore, the convergence is less stable and the testing accuracy reached is much smaller. There are frequent observable peaks and floors in the testing accuracy curves for all the algorithms.

We use a similar target scheme as in the previous section to evaluate the algorithms' performance with many learning rates in a single plot. However, we use smaller targets reflective of the performance in the non-IID case. The results can be seen in Figure 5.6. Firstly, we notice improved performance of EL-Oracle compared to DP-SGD. It indicates that changing the topology every round pays off in the case of non-IID data distribution. The reason is that different nodes not only have different data but also quite different models. We can see the effect in more detail in Figure 5.11. A node in El-Oracle receives models from other nodes at an equal rate but in D-PSGD it communicates only with its neighbours. Furthermore, based on this metric, AD-PSGD and Super GL perform similarly for targets smaller or equal to 60%, but Super-GL cannot reach higher targets due to over-fitting.

AD-PSGD, as the only algorithm, achieves a 65% target. However, in Figure 5.5, AD-PSGD did not reach a testing accuracy of 65% at any time. Our explanation is two-fold. We use multiple seeds and the convergence is not stable. Therefore, AD-PSGD never crossed the line of 65% because the two different seeds did not reach it at the same time stamp. The first one reached it at some timestamp

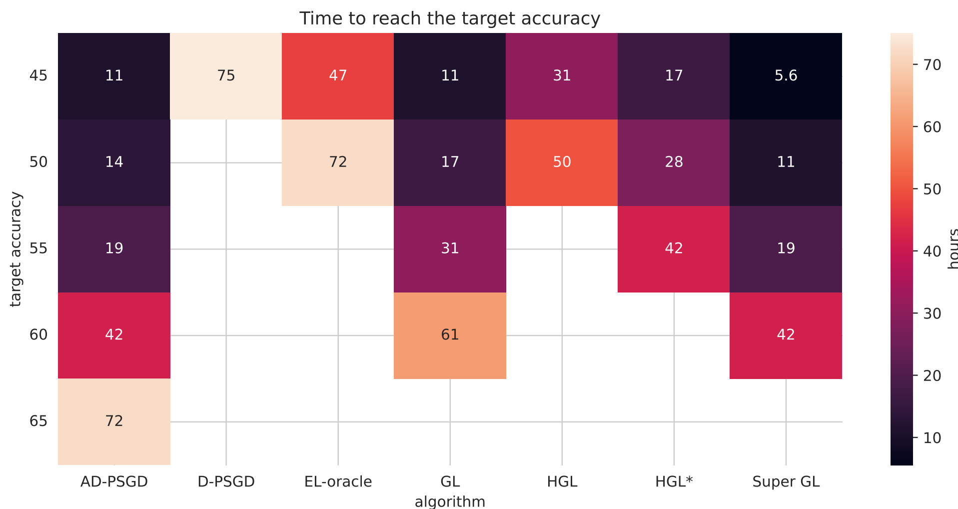

Figure 5.6: Time required in hours to reach the mean target testing accuracy in image classification on the CIFAR-10 data set with non-IID data distribution on 32 nodes. The missing values indicate that the accuracy was not reached in the given time frame. For each combination of algorithm and target, we selected the best-performing learning rate.

but then dropped down. The second reached it too, but at a different time. However, from the perspective of the target metric, both seeds managed to reach the target testing accuracy.

There are computing and communication efficiency plots in Figure 5.7. Regarding efficiency, the algorithms compare to each other similarly to the IID case. In communication efficiency, GL managed to reach the efficiency of synchronous algorithms, but those three are still the least efficient. Again, AD-PSGD is the most efficient in both computation and communication.

5.5 Recommendation on MovieLens

We use a similar setting and metrics to evaluate the algorithms on the recommendation task on the MovieLens data set.

In Figure 5.8 there is an average testing loss reached over hours of simulated training for different algorithms. When using the MovieLens data set, we plot the loss instead of accuracy, there is no concept of distinct classes, unlike in the case of classification. We selected the best-performing learning rates for each algorithm. However, it is even more difficult than in the previous experiments to select a single best-performing learning rate representing each algorithm. Some learning rates converge faster but do not reach as small loss, while others converge to a slightly smaller optimum, but take significantly longer. For this reason, we will focus our discussion on the target metric, which uses the best-performing learning rate for each target separately.

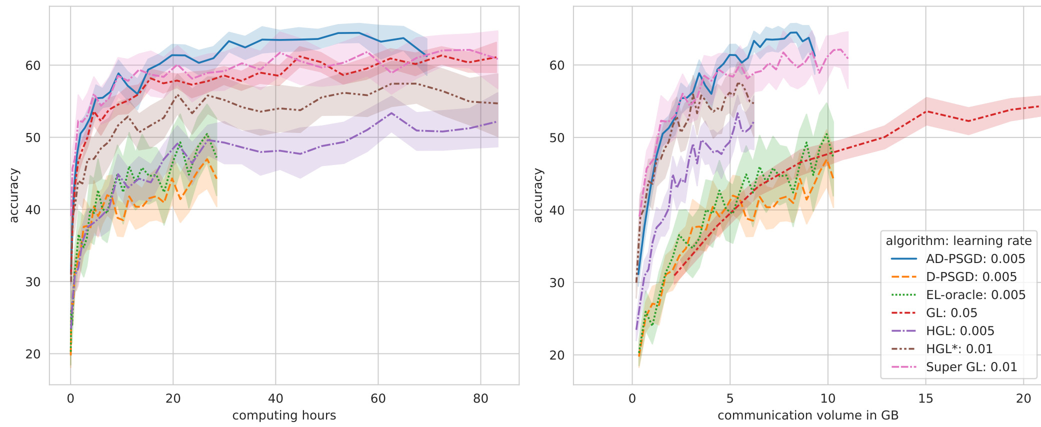

Figure 5.7: In the left plot, there is testing accuracy reached over the hours spent computing In the right plot, there is testing accuracy over the communication volume in GB. The plots are the result of image classification on the CIFAR-10 data set with non-IID class distribution on 32 nodes. The error area marks the standard deviation of the testing accuracy of different nodes. For each algorithm, we selected the best-performing learning rate.

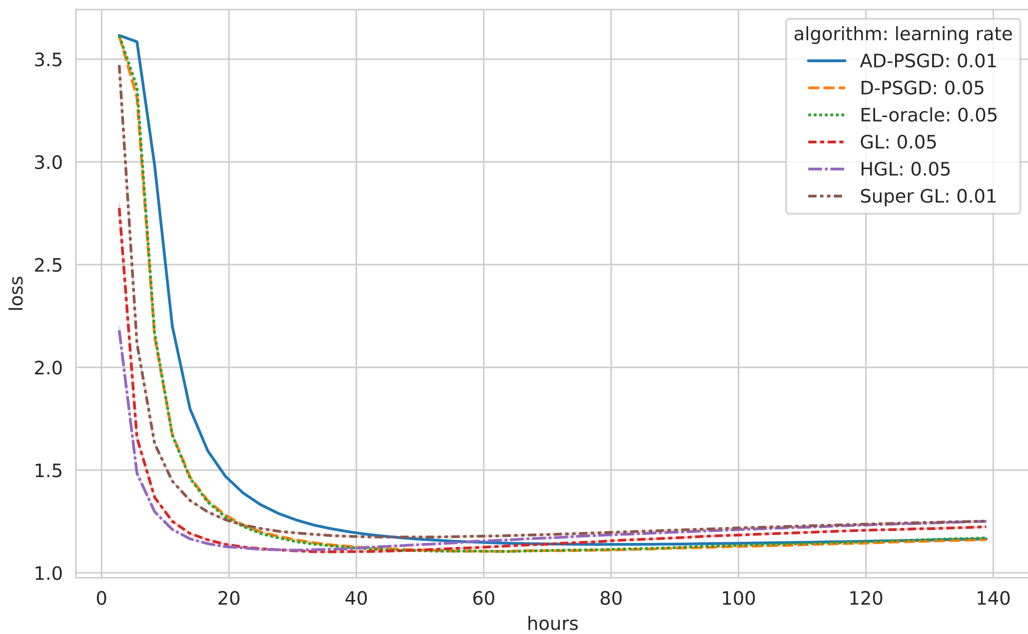

Figure 5.8: Testing loss reached over the elapsed simulated time in a recommendation task on the MovieLens data set on 32 nodes. The error area marks the standard deviation of the testing loss of different nodes. For each algorithm, we selected the best-performing learning rate.

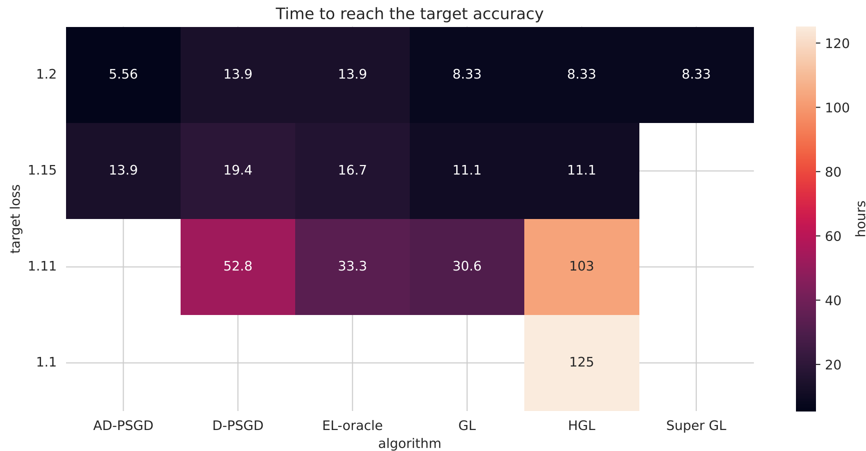

Figure 5.9: Time required in hours to reach the mean target testing loss in a recommendation task on the MovieLens data set on 32 nodes. The missing values indicate that the loss was not reached in the given time frame. For each combination of algorithm and target, we selected the best-performing learning rate.

In Figure 5.9 we see the times in hours required to reach the given mean target loss. The results are quite different compared to image classification on the CIFAR-10 data set. In the previous experiments, we saw that AD-PSGD and Super GL were two of the best-performing algorithms, however, this is not the case for the recommendation on the MovieLens data set. They initially converge fast but are not able to reach smaller testing loss targets. We attribute this to the sending rate of the fast nodes, who in the early stages possess better models and help speed up the convergence. However, in the MovieLens experiments, it is easier for all the algorithms to converge to a good solution. Therefore the benefit of a higher sending rate of fast nodes is unneeded and only results in eventual over-fitting.

In contrast to AD-PSGD and Super GL, the synchronous algorithms D-PSGD and EL-Oracle converge slower but can reach better targets. Similarly to the classification on the non-IID CIFAR-10 date set, EL-Oracle converges faster than D-PSGD. The best-performing algorithm is GL, which can reach the solution equivalent in testing loss to the synchronous algorithms but at a faster pace. Thanks to a high computing utilization, it is faster than the synchronous algorithms but at the same does not over-fit as other asynchronous algorithms due to its equal sending rate.

There are efficiency metrics in Figure 5.10. In terms of computing efficiency, GL and the synchronous algorithms are the most efficient. Super GL and AD-PSGD are the least efficient. They compute for the entire duration but do not reach a good solution. Synchronous algorithms are also the most efficient in communication. As we mentioned earlier, GL is not as efficient in communication due to its frequent model sending. However, its performance in this metric is not as bad as in the case of the image classification experiments on CIFAR-10.

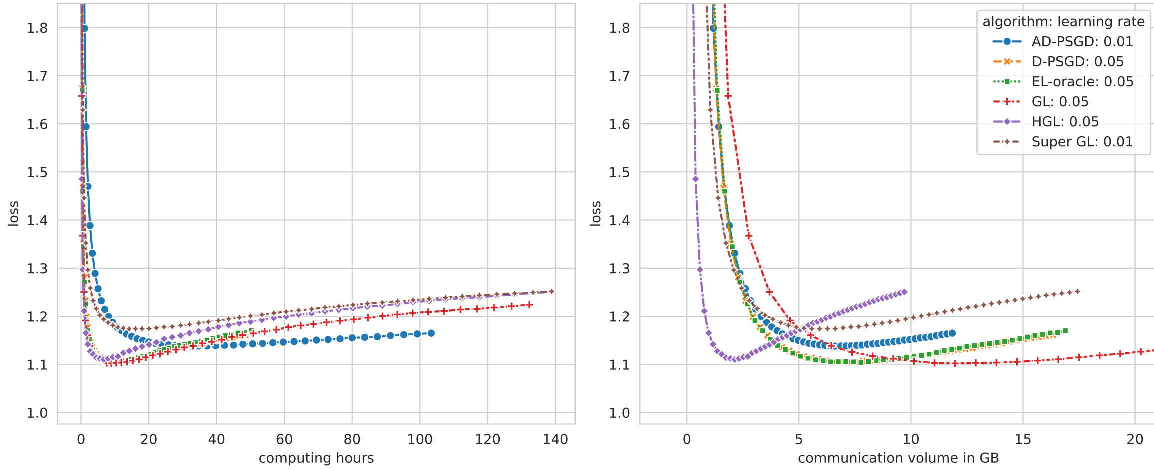

Figure 5.10: In the left plot, there is testing loss reached over the hours spent computing In the right plot, there is testing loss over the communication volume in GB. The plots are the result of a recommendation task on the MovieLens data set on 32 nodes. The error area marks the standard deviation of the testing loss of different nodes. For each algorithm, we selected the best-performing learning rate.

5.6 Discussion

We have seen the evaluation of the performance and efficiency of the DL algorithms in three different scenarios. Overall, no algorithm is the best in every setting. Depending on the task, different algorithms perform better than others.

In image classification on both CIFAR-10 data distributions, the best performer is AD-PSGD. It reaches the highest mean testing accuracy and is the most efficient in both efficiency metrics. We attribute this to the split of nodes into two groups. The active fast nodes help the model to converge faster, while the fast passive nodes are curtailed to limit over-fitting compared to Super GL. However, its performance is not as great in the recommendation task on the MovieLens data set, where it is easier for the models to converge and other nodes do not need help from the fast-sending nodes.

In the MovieLens recommendation, the best performer is GL. It reaches the smallest testing loss target in the shortest amount of time. It achieves almost full computing utilization, while not over-fitting as other asynchronous algorithms. However, because of the small value we set for the hyperparameter α, the communication is more costly compared to the other algorithms. With a higher value of α it would be more communication-efficient but at the cost of slower convergence.

Lastly, we have shown that asynchrony is useful in scenarios of nodes with heterogeneous performance. In such a scenario, asynchronous algorithms can achieve much higher resource utilization. GL achieves better performance in all three experiments than both D-PSGD and EL-

Oracle. Even though GL results in a higher communication volume than the synchronous algorithms, the burden is manageable for the nodes' networks in our experiments.

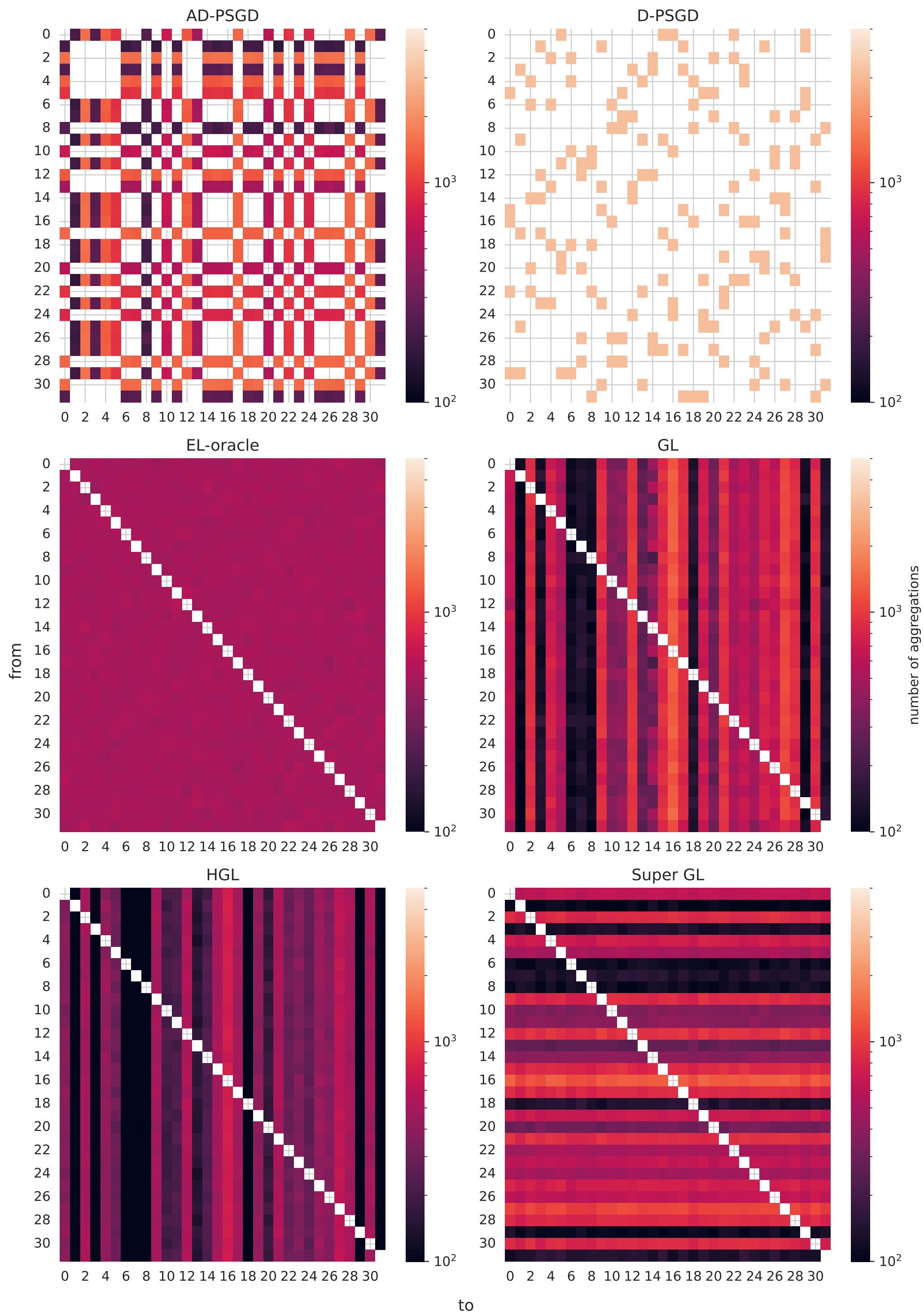

Figure 5.11: Heatmaps showing the number of models utilized from other nodes in the CIFAR-10 experiment.

Chapter 6

Heterogeneous Gossip Learning

In this chapter, we introduce a new asynchronous DL algorithm Heterogeneous Gossip Learning (HGL). The idea to create a new algorithm came to our minds when we were examining the behaviour of the other DL algorithms in the presence of nodes of heterogeneous performance. We are going to state our motivation behind the algorithm, then present the algorithm itself and at last evaluate it.

6.1 Motivation

We set our goal to design an algorithm which would perform great in a setting with nodes of highly varied performance and non-IID data. We want all nodes to compute as often as possible, therefore we choose an asynchronous design. However, at the same time, we would like to avoid overfitting coming from the side of faster nodes. In a way, GL satisfies these design goals, however, it suffers from a different problem. There is a hyperparameter α, which determines how often a node shares its model. If α is relatively small, there will be high computing utilization, but also a high communication volume. This is not suitable for a setting with slow networks. On the other hand, if we set α to a relatively large value, the computing utilization will drop significantly. Therefore, we would like to design an algorithm that behaves similarly to GL, but has no hyperparameter and has a communication cost in line with the other algorithms.

6.2 Algorithm

Based on our design goals we came up with the following algorithm that can be seen in Figure 6.1. First of all, we use a similar main loop to Super GL. All nodes compute at all times. Each node has a queue where it puts the models waiting to be aggregated. In this way, the received models cannot

Algorithm 7: HGL on i_{th} active node

Input: initial model x_i, local data set ξ_i, learning rate γ, an empty queue q_i, set of computing speeds of different nodes S, where s_j is a training speed of j

1 Run `Disseminate()` in a new thread;
2 **repeat**
3 Sample a batch $\hat{\xi}_i$ from ξ_i;
4 Compute the gradient $g_i = \nabla F(x_i, \hat{\xi}_i)$;
5 Update the model $x_i = x_i - \gamma \cdot g_i$;
6 Aggregate x_i with all models in the queue q_i;
7 **until** *the running time elapses*;

8 **Procedure** `Disseminate()`**:**
9 Initialize distribution σ_i, such that the probability of choosing node j is proportional to its speed $p_j = \frac{s_j}{\sum s \in S \setminus \{s_i\}}$;
10 Sample a node j from the distribution σ_i;
11 Send x_i to node j;
12 Wait $mean(S \setminus \{s_i\})$ seconds;

13 **Procedure** `OnReceiveModel(`x_j`)`**:**
14 put x_j into the queue q_i;

Figure 6.1: Heterogeneous Gossip Learning Algorithm

be overwritten if the node is not yet ready to aggregate, unlike in GL. However, compared to Super GL, the sending of models is not handled in the main loop, but in a separate thread. We made this change because the connection of computing speed with the sending rate is the main reason behind the over-fitting of Super GL.

With model sending, we employ a similar strategy to GL with two changes. Firstly, each node sends the model to others at an equal rate but the receiving node is not sampled uniformly. Instead, the sampling distribution is based on the computing speed of the receiver, favouring the faster nodes. Therefore, the faster nodes are more likely to be sampled and receive the model. Our idea is that if faster nodes are left unconstrained, in the same amount of time they will compute many more training steps than the slow nodes and over-fit their model to their data. If they receive more models, they will aggregate them into their model, limiting the impact of over-fitting. On the other hand, the slow nodes will receive on average a smaller amount of models. Their models will aggregate with other models less often, meaning they can converge towards the model partly based on their data, instead of being forced to another direction by aggregating an influx of other models. However, this will result in slower convergence of the slow nodes. Secondly, we remove difficult-to-set hyperparameter α and instead use a mean training time as a waiting period in sending. These two changes mean that all nodes will on average receive a single model per training step. In this way,

every node will have an equal chance to contribute to the models of others.

To implement HGL we need to know the computing speeds of all the participating nodes. In our implementation, we assume an oracle with this knowledge. In practice, when the oracle is not available, the nodes can share their computing speeds with other nodes, e.g. as meta-data when sharing models. An advantage of this approach is that if the node's computing speed changes during the execution, e.g. the mobile device's user starts engaging in another activity, the algorithm can adapt to the new setting on the go.

6.3 Evaluation

We evaluate HGL in the same three scenarios as other DL algorithms in chapter 5. HGL does not perform very well in image classification on the CIFAR-10 data set. As we have explained earlier, these two experiments favour the algorithms that can converge faster using the higher sending rate of the fast nodes. To mitigate this issue we develop an alternative version of the algorithm - HGL*, where the receiving node is uniformly sampled. This brought some performance benefits. HGL* performs similarly to GL in the case of IID data distribution, as seen in Figure 5.3. However, in the non-IID data distribution, both HGL and HGL* lag behind all other asynchronous algorithms, but they still perform better than the synchronous algorithms, as seen in Figure 5.6.

On the other hand, HGL performs very well in the recommendation task on the MovieLens data set. In fact, it is the best-performing algorithm. As the only algorithm, it could reach the smallest loss target (see Figure 5.9), but only after a significant amount of time. Furthermore, it is the most efficient algorithm in communication and one of the most efficient in computation cost, as shown by Figure 5.10.

The algorithm fulfils our expectation of performing similarly to GL. For the tasks where GL does not perform that well, such as in image classification on CIFAR-10, HGL performs even worse. On the other hand, in the recommendation task on the MovieLens data set where GL shines, HGL can converge to the models with an even smaller testing loss. In every case, it removes the need to tune the hyperparameter and HGL is much more communication-efficient than GL.

Chapter 7

Conclusion and Future Work

We have evaluated different decentralized learning algorithms in a realistic heterogeneous setting in three experiments on two data sets. We have shown that in the classification task on the CIFAR-10 data set, AD-PSGD reaches the highest testing accuracy in the given time frame in both IID and non-IID settings, while being the most efficient in both computation and communication. We have attributed this to its distribution of the nodes into two groups, which allows for fast convergence and avoids significant over-fitting. On the other hand, in the recommendation task on the MovieLens data set, Gossip Learning has shown the best performance from the existing algorithms. It reaches the same mean target testing loss as the synchronous algorithms at the cost of significant communication volume. We conclude that there is no one-fit-all algorithm. In different scenarios, different algorithms perform the best. However, we have shown the benefits of asynchrony in the presence of nodes of varied computing and networking performance. In every examined scenario there is an asynchronous algorithm, namely Gossip Learning, that over-performs the synchronous ones.

Furthermore, we have introduced a new decentralized learning algorithm HGL, which considers the differences in computing speeds of the participating nodes. The algorithm performs similarly to Gossip Learning while being more communication-efficient and removing the hyperparameter of GL. In the recommendation task on the MovieLens data set, HGL reaches the smallest testing loss, while being efficient in both computation and communication.

7.1 Future Work

There is a plethora of possibilities for future work. The first option is a direct extension of this work, evaluating scenarios with a greater number of participating nodes in more types of ML tasks on different data sets. This would provide an even more comprehensive evaluation of the performance

of decentralized learning algorithms. Another option is to include different types of scenarios, e.g. a node or network failure during the run or the presence of extremely slow nodes. The simulator currently does not support all of these scenarios, which would have to be implemented.

In the case of AD-PSGD, in the presence of heterogeneous nodes, on average there is the same proportion of the fast computing nodes in both active and passive groups. Using this fact we explained its great performance in the image classification on the CIFAR-10 data set. It would be interesting to see how the node assignment to the two groups impacts the performance.

Regarding HGL, theoretical justification, such as a convergence proof, can be shown. Secondly, the algorithm could also be expanded to depend on nodes' network capabilities. Based on the traces we used, the main limit factor was the computing speed, therefore we did not include it in the algorithm.

Lastly, there are possible improvements in the simulator itself. Although we have implemented computation of tasks in batches, our implementation is not the most memory efficient and not bug-free when multiple brokers are employed. The simulator currently struggles with network congestion, causing the simulation to become very slow. Although this was not a limit factor in our scenarios, the algorithms in the bandwidth scheduler should be improved to better deal with a high volume of traffic. Furthermore, we suggest splitting the simulation class into two classes, one responsible for simulation and one for coordination, to increase the clarity of each class. Another useful feature would be to add an option to send partial models.

Bibliography

[1] Martijn De Vos. *Decentralized Learning Simulator*. `https://github.com/sacs-epfl/decentralized-learning-simulator`. 2024.

[2] Martijn De Vos, Sadegh Farhadkhani, Rachid Guerraoui, Anne-Marie Kermarrec, Rafael Pires, and Rishi Sharma. "Epidemic Learning: Boosting Decentralized Learning with Randomized Communication". In: *Advances in Neural Information Processing Systems* 36 (2024).

[3] Akash Dhasade, Anne-Marie Kermarrec, Rafael Pires, Rishi Sharma, and Milos Vujasinovic. "Decentralized learning made easy with DecentralizePy". In: *Proceedings of the 3rd Workshop on Machine Learning and Systems*. 2023, pp. 34–41.

[4] F Maxwell Harper and Joseph A Konstan. "The movielens datasets: History and context". In: *Acm transactions on interactive intelligent systems (tiis)* 5.4 (2015), pp. 1–19.

[5] István Hegedűs, Gábor Danner, and Márk Jelasity. "Decentralized learning works: An empirical comparison of gossip learning and federated learning". In: *Journal of Parallel and Distributed Computing* 148 (2021), pp. 109–124.

[6] István Hegedűs, Gábor Danner, and Márk Jelasity. "Gossip learning as a decentralized alternative to federated learning". In: *Distributed Applications and Interoperable Systems: 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings 19*. Springer. 2019, pp. 74–90.

[7] Kevin Hsieh, Amar Phanishayee, Onur Mutlu, and Phillip Gibbons. "The non-iid data quagmire of decentralized machine learning". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 4387–4398.

[8] Junxian Huang, Cheng Chen, Yutong Pei, Zhaoguang Wang, Zhiyun Qian, Feng Qian, Birjodh Tiwana, Qiang Xu, Z Mao, Ming Zhang, et al. "Mobiperf: Mobile network measurement system". In: *Technical Report. University of Michigan and Microsoft Research* (2011).

[9] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. "Ai benchmark: All about deep learning on smartphones in 2019". In: *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE. 2019, pp. 3617–3635.

[10] Alex Krizhevsky, Vinod Nair, Geoffrey Hinton, et al. "The CIFAR-10 dataset". In: *online: http://www.cs.toronto.edu/ kriz/cifar.html* 55.5 (2014), p. 2.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[12] Fan Lai, Yinwei Dai, Sanjay Singapuram, Jiachen Liu, Xiangfeng Zhu, Harsha Madhyastha, and Mosharaf Chowdhury. "Fedscale: Benchmarking model and system performance of federated learning at scale". In: *International conference on machine learning*. PMLR. 2022, pp. 11814–11827.

[13] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[14] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. "Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent". In: *Advances in neural information processing systems* 30 (2017).

[15] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. "Asynchronous decentralized parallel stochastic gradient descent". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 3043–3052.

[16] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. "Communication-efficient learning of deep networks from decentralized data". In: *Artificial intelligence and statistics*. PMLR. 2017, pp. 1273–1282.

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

[18] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. "Llama 2: Open foundation and fine-tuned chat models". In: *arXiv preprint arXiv:2307.09288* (2023).

[19] Katerina Vitsaxaki. *Asynchronous algorithms for Decentralized Learning*. `https://github.com/sacs-epfl/async-dl`. 2023.

[20] Yuxin Wu and Kaiming He. "Group normalization". In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 3–19.

[21] Ligeng Zhu, Zhijian Liu, and Song Han. "Deep leakage from gradients". In: *Advances in neural information processing systems* 32 (2019).