ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

# Exploring Distributed Learning Algorithms with Communication Optimization

Bachelor Thesis

Samuel Bohl

July 2022

Supervisors: Prof. Ce Zhang, Dr. Binhang Yuan
Department of Computer Science, ETH Zürich

# Abstract

In this thesis I conduct a survey on state-of-the-art distributed learning algorithms regarding communication optimization and introduce a distributed machine learning system called Bagua. Next, I analyze a specific decentralized algorithm called RelaySGD and examine the communication mechanism, as well as the convergence guarantees and benefits regrading communication optimization. Furthermore, I implement RelaySGD within Bagua and experimentally verify my implementation in terms of system speedup and statistical accuracy of different data distributions between machines. Lastly, I discuss the results and some potential improvements.

# Contents

Chapter 1

# Introduction

With the growing popularity of machine learning over the past decade, the problems solved by machine learning have become more complex and require increasing amounts of data. Due to this growth in both the models used and the data scale, training machine learning models on single computers is becoming increasingly difficult. Popular datasets such as Open Images V4 [12] and ImageNet [5] exceed 500GB and 150GB respectively.

Distributed learning aims to solve this problem by splitting the required optimization computation and/or training data across multiple machines. In distributed learning, models are trained across multiple machines, where the communication overhead between the workers poses the main limiting factor for scalability. This is also visible in the increasing number of publications in recent years, aiming to improve this bottleneck using different system relaxations methods, broadly classified into 3 types. Lossy communication compression, asynchronous communication or decentralized communication. These techniques aim to ”relax” one or more bottlenecks imposed in a system, such as communication bandwidth, latency, synchronization cost, etc. [16]

This thesis aims to explore one particular decentralized communication algorithm called RelaySGD [23], implemented within a distributed machine learning system called Bagua. Bagua is a machine learning training acceleration framework that initially supports the communication compression pattern, allowing for more of a focus on the algorithm level without having concerns about the communication engine. [6] This algorithm tackles a key challenge in decentralized learning, the handling of differences between the workers' local data distributions.

Finally, I want to evaluate the algorithm in different real-world settings against other decentralized algorithms in Bagua.

Chapter 2

# Preliminaries

This chapter aims to provide a short introduction to the concepts that will be used in the thesis. First, I will briefly introduce distributed machine learning and its parallelization methods. Then I will present distributed learning algorithms and the different communication paradigms and relaxation techniques used. Finally, I present Bagua [6], a distributed machine learning system that I used to implement the algorithms of interest.

## 2.1 Distributed Machine Learning

With the growing demand and adoption of artificial intelligence over the past decade, the amount of data and the complexity of the associated models is increasing, outpacing the increase in computing power. Therefore, there is a need to distribute the machine learning workload across multiple machines and to transform the previously centralized system into a distributed system. Distributed machine learning can be parallelized in two different ways, each addressing a different bottleneck. [22]

### 2.1.1 Model Parallel

The first parallelization strategy is called model parallelism. Here the model is split across multiple devices, each containing a copy of the entire dataset. This becomes useful when the model size becomes too large to fit into the fast memory of a single device. Unfortunately, this does not necessarily reduce the training time, as the devices containing the split model layers must sequentially communicate their computed values to the device containing the next layer. [16]

### 2.1.2 Data Parallel

In data parallelism, the data is distributed across multiple devices and then synchronize the models using different strategies and algorithms. This approach is useful when the size of the dataset becomes a training bottleneck. Figure 2.1 further illustrates the difference between data and model parallelism. [16]

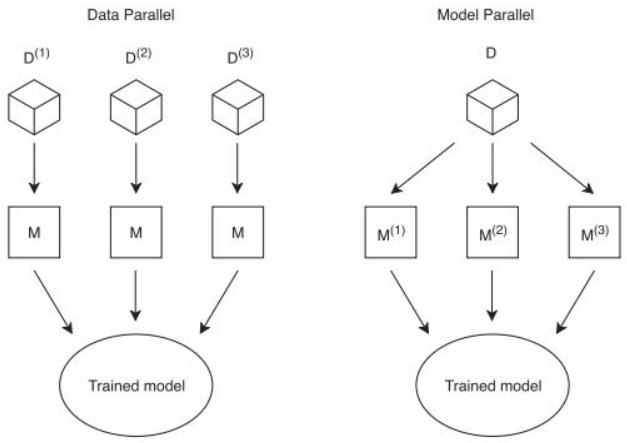



**Figure 2.1:** Illustration of parallelism in distributed machine learning. In the data parallelism approach, multiple instances of the same model are trained on different subsets of the training dataset. With model parallelism, a single model is distributed across multiple nodes. [22]

## 2.2 Distributed Learning Algorithms

This section describes the various system architectures and relaxation methods that can be used to design distributed learning algorithms.

### 2.2.1 Distributed Learning Systems

The communication paradigms on which distributed learning algorithms are built use different abstractions of the underlying distributed system architecture. The two most popular distributed system abstractions for deep learning are parameter servers and the allreduce paradigm. [10]

#### Parameter Server

The parameter server is one of the most popular architectures for distributed stochastic gradient descent. The set of all computing devices is divided into a small set of one or several parameter servers and the rest into worker nodes. The worker nodes periodically synchronize their parameters with the parameter server, which then contains the global model. Figure 2.2 illustrates this architecture. Since each worker has only one peer to communicate with, this can be an improvement over decentralized architectures, where each worker has to communicate with more than one peer. It also means that the network bandwidth and latency can become a major bottleneck as the number of workers increases. [16]

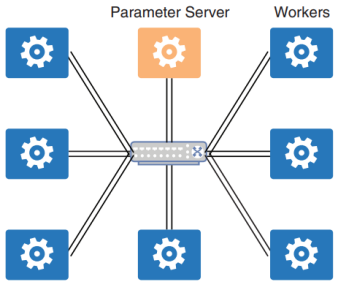



**Figure 2.2:** Simplified illustration of a parameter server architecture with a single parameter server (orange) and worker nodes (blue). [16]

#### Allreduce

One way to circumvent the potential network congestion problem posed by the Parameter Server architecture is to use the Allreduce operator to synchronize the parameters across all compute nodes. Since the optimization and implementation of the Allreduce operator has been studied by the HPC community for decades, there are multiple communication efficient implementations and typologies for different cases. For example, the Tree Allreduce implementation minimizes the network bandwidth but maximizes latency, while Butterfly Allreduce minimizes latency. Communication frameworks will use different implementations to optimize given certain network parameters. [16, 24]

### 2.2.2 Relaxation Techniques

Distributed learning algorithms can use various system relaxation techniques to improve their training performance by addressing the underlying architecture bottlenecks I briefly mentioned in the previous section. These techniques can be roughly divided into 3 relaxation patterns: Lossy communication compression, asynchronous communication and decentralized communication. [16]

#### Lossy Communication Compression

Lossy Communication Compression aims to accelerate the expensive step of gradient exchange in distributed SGD. Instead of exchanging the gradient as 32 bit floating point numbers, the gradient is first quantized into a lower precision representation before the communication step. As a result, the time needed to transfer the gradient decreases. If, for example, the gradient is quantized into an 8 bit fixed point representation (instead of 32 bit),

the transfer time is theoretically reduced by 400%. However, this involves a trade-off in terms of accuracy because quantization is a lossy compression technique. The extent of the performance drop depends on the chosen quantization algorithm. An extreme example of quantization is SignSGD [3]. This algorithm compresses the gradient into a single bit to represent the sign of the gradient. It is obviously less accurate, but it works surprisingly well compared to other existing algorithms. [16]

#### Asynchronous Communication

With asynchronous (or delayed) communication, workers continue training without waiting to fully synchronize with each other during the synchronization step. Since the synchronization step is usually blocking, asynchronous communication reduces the time required for the synchronization step, which can be a major bottleneck in large clusters, especially when some workers are slower than others. This can also be a potential downside in terms of accuracy when there is a significantly slower worker $w_0$ and the other workers might train on for several iterations without an update from the slower worker $w_0$.

#### Decentralized Communication

As the name suggests, algorithms using this relaxation technique aim to decentralize communication between the different workers. There is therefore no parameter server and so the allreduce communication paradigm is used instead. Several network topologies can be used to define the neighbors of each worker. The simplest topology would be a ring. Here, each worker communicates with his left and right neighbors, significantly reducing the latency overhead to $O(1)$. Different network topologies involve a trade-off between bandwidth and latency. [16]

## 2.3 Bagua

Bagua is a deep learning acceleration framework for PyTorch developed by Kuaishou Technology's AI Platform and ETH Zurich's DS3 Lab. In recent years, researchers in the machine learning community have proposed a wide range of techniques to reduce the communication overhead through several system relaxations. However, existing systems only support a subset of these optimizations and therefore cannot take advantage of all possible optimizations as a result. Bagua was built to bridge the gap between the new theory and current systems to improve real-world performance. It provides MPI-style communication operations and a flexible system abstraction that supports all of the state of the art system relaxation techniques, including those described in the previous section. Various other system and communication

optimizations allow Bagua to outperform existing systems such as PyTorch-DDP [13], Horovod [19] and BytePS [9] in more real-world scenarios by a factor of up to 2x in various tasks under different network conditions. [6]

Bagua's communication primitives, combined with various hooks that allow for any communication or computation at various points in every training step, allow developers to have control over almost every detail of data-parallel distributed training, including what to communicate, when to communicate, and how to update the models. New algorithms in Bagua automatically benefit from its system optimizations such as memory management, execution management, and communication-computation overlap, allowing developers to take full advantage of the algorithm without a compromise caused by inefficient implementation. With this innovative design, algorithm developers can now easily create, test and benchmark their distributed learning algorithms. [2]

### 2.3.1 System Optimizations

A key component of Bagua is the execution optimizer, which, given a neural network and a distributed learning algorithm, automatically schedules and optimizes the computations and communication primitives performed when computing each layer. Bagua explores the following three optimization techniques. [6]

#### Overlap Communication and Computation

The overlap of communication and computation is a key optimization as part of accelerating various distributed algorithms. Bagua is able to overlap communication primitives along with the calculations required by an algorithm, such as compression/decompression and the model updates. Bagua automatically analyzes the computation graph that includes the in-place tensor operations and tensor communication primitives. [6]

#### Tensor Bucketing and Memory Flattening

Another optimization of the efficiency of communication and parallel computing is merging small tensors into bucket. This is usually done to better utilize the network bandwidth, when otherwise a lot of small tensors have to be sent. Now only one bigger message containing multiple smaller tensors is sent instead. Once the computation graph is split into buckets, Bagua conducts fusion over those buckets. After determining the partitioning of the buckets in the first backward propagation, Bagua carefully aligns the model parameters and gradients into a bucket in a continuous memory space. This continuous or flattened view of memory allows the computation unit to utilize parallelism more effectively. [6]

### Hierarchical Communications

Hierarchical communication is particularly useful for heterogeneous network connections, such as bandwidth differences between GPUs within a server and the bandwidth between servers. Bagua creates hierarchical communication on two levels: intra-node and inter-node. An intra-node connection usually has a significantly higher bandwidth compared to inter-node connection, therfore intra-node communication is preferred over inter-node communication whenever possible. Bagua's communication primitives have been optimized based on this abstraction. [6]

Chapter 3

# Related Work

In this chapter, I introduce two decentralized distributed learning algorithms that we will later use to compare the performance with respect of the heterogeneity of the data distribution. The first algorithm, Decentralized Parallel Stochastic Gradient Descent (D-PSGD), is already implemented in Bagua and will be used to evaluate the performance of the second algorithm, RelaySGD [23].

## 3.1 D-PSGD

D-PSGD [14] averages the model using a symmetric doubly stochastic peer weight matrix $W$, where the value of $W_{ij}$ contains the weight of how much node $i$ can affect node $j$. If two nodes are disconnected $W_{ij} = 0$, so the network does not need to be fully connected, therefore sparse topologies can be used. Algorithm 1 describes the algorithm in two simplified steps. Line 3 averages the model with all its neighboring nodes and then line 4 updates the local model. The gradient on line 4 $\gamma \nabla f_i(x_i^{(t)})$ can be computed in parallel to the model averaging step on line 3. In the last step of line 7, the average of all worker nodes $X = \frac{1}{n} \sum_{i=1}^{n} x_i^{(T)}$ is taken as the final averaged model.

**Algorithm 1** D-PSGD

**Input** model weight $X$, peer weight matrix $W$, learning rate $\gamma$

1: **for** node $i$ **in parallel**
2: **for** $t = 0, 1, ..., T-1$ **do**
3: $x_i^{(t+1/2)} = \sum_{j=1}^{n} W_{ij} x_j^{(t)}$
4: $x_i^{(t+1)} = x_i^{(t+1/2)} - \gamma \nabla f_i(x_i^{(t)})$
5: **end for**
6: **end for**
7: $X = \frac{1}{n} \sum_{i=1}^{n} x_i^{(T)}$

This algorithm is implemented in Bagua as `DecentralizedAlgorithm` and I will use this implementation later to observe convergence behavior within heterogeneously distributed data and compare it with RelaySGD.

### 3.1.1 Communication mechanism

This communication mechanism is also known as gossip averaging. In every timestep all workers average their local model $x_i$ with the received models from its neighbors $x_j$ ($j \in N_i$),

$$x_i^{(t+1)} = W_{ii} x_i^{(t+1/2)} + \sum_{j \in N_i} W_{ij} x_j^{(t)}$$

using averaging weights defined by a gossip matrix $W \in \mathbb{R}^{nxn}$. The gossip matrix is chosen in such a way that the weights asymptotically converge to $\frac{1}{n}$, distributing all updates uniformly over the workers. As briefly mentioned earlier, $W$ is a symmetric doubly stochastic matrix, which means that (i) $\forall i, j\ W \in [0, 1]$, (ii) $\forall i, j\ W_{ij} = W_{ji}$ and $\sum_j^n W_{ij} = 1$. [21, 23]

$$x_i^{(t+1)} = x_i^{(t+1/2)} + \sum_{j \in N_i} W_{ij} x_j^{(t)}$$

### 3.1.2 Convergence

D-PSGD converges at a rate of $O(\frac{\sigma}{\sqrt{nT}} + \frac{(n\zeta^2)^{\frac{1}{3}}}{T^{\frac{2}{3}}} + \frac{1}{T})$ where $\zeta^2$ is the data variance between all workers, $\sigma^2$ is the data variance within each worker, $n$ is the number of workers, and $T$ is the number of iterations. [21] When $\zeta^2 = 0$ and $\sigma^2 = 0$, the data is distributed in an equal optimum. The values of $\zeta^2$ and $\sigma^2$ both grow with increasing data heterogeneity, where e.g. $\zeta^2 = 1$ would be a very heterogeneously distributed dataset. Given a sufficiently large number of iterations $T$ and thr additional assumption that the data is uniformly distributed, the rate converges to $O(\frac{1}{\sqrt{nT}} + \frac{1}{T})$. [14]

## 3.2 RelaySGD

Decentralized algorithms usually use some version of gossip messaging to average the models of the different workers. RelaySGD, on the other hand, approximates uniform model averaging

$$x_i^{(t+1)} = \frac{1}{n} \sum_{j=1}^{n} x_j^{(t+1/2)}$$

without the gossip multiplication step and instead relays the model weight messages to all workers with a delay of $\tau_{ij}$, the minimum number of network hops between nodes $i$ and $j$:

$$x_i^{(t+1)} = \frac{1}{n} \sum_{j=1}^{n} x_j^{(t-\tau_{ij}+1/2)}$$

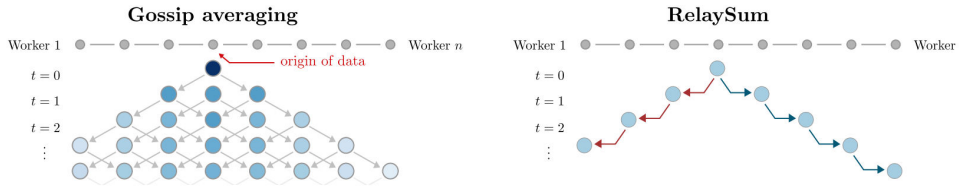



**Figure 3.1:** The figure on the left shows, from the perspective of the fourth worker in a chain network, how gossip averaging slowly spreads information through the network. In RelaySum, the messages are relayed without weighting, resulting in the uniform delivery of the information to every worker. When multiple workers broadcast simultaneously (not shown), RelaySum sums their messages up and uses the same bandwidth as gossip averaging. [23]

### 3.2.1 Communication Mechanism

This can be achieved using the RelaySum communication mechanism which provides delayed message sums $s_i = \sum_{j=1}^{n} m_j^{(t-\tau_{ij})}$ to each worker $i$ in a tree network, where the message $m_j^{(t)}$ is created by worker $j$ at time $t$. The following decomposition illustrates this mechanism in a chain topology:

$$s_i^{(t)} = \sum_{j=1}^{n} m_j^{(t-\tau_{ij})} = m_i^{(t)} + \underbrace{\sum_{j=1}^{i-1} m_j^{(t-\tau_{ij})}}_{\text{messages from the left}} + \underbrace{\sum_{j=i+1}^{n} m_j^{(t-\tau_{ij})}}_{\text{messages from the right}}$$

Messages from the left $\sum_{j=1}^{i-1} m_j^{(t-\tau_{ij})}$ are sent a a single message $m_{(j-1)\to j}$ from worker $j-1$ to worker $j$ and vice versa for the messages from the right. This allows neighboring workers to compute these messages from the messages they received from their neighbors in the previous time steps. Figure 3.1 illustrates the difference between gossip averaging and RelaySum further. Algorithm 2 shows how this communication mechanism is generalized on tree networks for RelaySGD.

#### Topologies

RelaySGD can be implemented in any tree network. I have already mentioned the simplest tree network, a chain, which is fine for illustration purposes. However, as the communication delay grows linearly with the number of workers $O(N)$, there is a need for a different topology to solve this potential bottleneck. A binary tree has a height of $O(\log N)$, which is also the maximum delay in this case, therefore this is a better option where there is an increasing number of workers. For the third topology, I used double binary trees [18]. The main idea behind using double binary trees is to use two

**Algorithm 2** RelaySGD [23]

**Input** model weight $x$, counter $c$, learning rate $\gamma$, tree network $N$

**for** $t = 0, 1, ..$ **do**
  **for** node $i$ **in parallel**
    $x_i^{(t+1/2)} = x_i^{(t)} - \gamma \nabla f_i(x_i^{(t)})$ (or Adam and/or momentum)
    **for** each neighbor $j \in N_i$ **do**
      Send $m_{i\to j}^{(t)} = x_i^{(t+1/2)} + \sum_{k \in N_i \setminus j} m_{k\to i}^{(t-1)}$ (relay messages)
      Send corresponding counters $c_{i\to j}^{(t)} = 1 + \sum_{k \in N_i \setminus j} c_{k\to i}^{(t-1)}$
      Receive $m_{j\to i}^{(t)}$ and $c_{j\to i}^{(t)}$ from node $j$
    **end for**
    $\bar{n}_i^{(t+1)} = 1 + \sum_{j \in N_i} c_{j\to i}^{(t)}$ ▷ sum up all received counts
    $x_i^{(t+1)} = \frac{1}{\bar{n}_i^{(t+1)}} \left( x_i^{(t+1/2)} + \sum_{j \in N_i} m_{j\to i}^{(t)} \right)$ ▷ average local model
  **end for**
**end for**

different topologies for different parts of the model. It is possible to communicate odd coordinates using a balanced binary tree A and to communicate the even coordinates using the complimentary tree B. With this combination of two trees, RelaySGD only needs additional constant memory equivalent to at most 2 model copies, and it sends and receives the equivalent of 2 models. [23]

### 3.2.2 Convergence

Thijs Vogels et. al. [23] showed in their paper that the algorithm theoretically converges where the dominant term $O(\frac{\sigma^2}{n\varepsilon^2})$ matches with the dominant term in the convergence rate of the centralized ('all-reduce') mini-batch SGD, and thus can not be improved. They also showed that RelaySGD's presented convergence results are independent of data heterogeneity $\zeta^2$ unlike other decentralized algorithms like D-PSGD.

Independence from data heterogeneity is the main advantage of using this algorithm over other decentralized algorithms. I will show this experimentally in the last chapter of this thesis. In the paper, the authors show that when there was a high data variance between workers, RelaySGD significantly outperformed D-PSGD and even $D^2$ [21] which had previously shown to be robust to this kind of data variance. Thanks to the RelaySum communication mechanism, this algorithm achieves this with the same communication volume per step as the typical gossip averaging mechanism.

Chapter 4

# Implementation

This chapter describes the RelaySGD implementation and all additional components that I needed for the experiments in the next chapter. The code can be found on Github https://github.com/samuelbohl/RelaySGD.

## 4.1 Data Distribution

Since RelaySGD promises promising performance benefits over heterogeneously distributed data, I needed a distributed data sampler suitable for heterogeneously distributed data. The PyTorch [17] distributed package provides a `DistributedSampler` class, which is suitable in most cases, as it distributes the data evenly and also offers the ability to pre-shuffle the dataset to achieve a better and more homogeneous distribution across all devices. Since we want the exact opposite, we need our own distributed sampler that can sample the data heterogeneously. For this purpose, I build a PyTorch Sampler subclass `DistributedHeterogeneousSampler`.

I follow the state of the art approaches [23, 15, 25, 7] to model the distribution of non-iid data using the Dirichlet distribution $Dir(\alpha)$. A lower $\alpha$ indicates greater heterogeneity in the distribution of the data across different devices. I used the code from Vogels et. al. [23] (`distribute_data_dirichlet()`) to generate non-iid Dirichlet distributed indices using the following parameters: $\alpha$, the number of target classes and the number of devices. Then I grouped the indices into evenly sized packages which were then each returned to the corresponding device as python lists wrapped inside of an iterator when the `__iter__()` method of the sampler was called by a default PyTorch dataloader.

## 4.2 Algorithms

I have already introduced the two algorithms I want to compare, D-PSGD and RelaySGD, in the related work section. I have also previously mentioned that the D-PSGD is already implemented within Bagua as `DecentralizedAlgorithm`, therefore there is no need for me to implement this algorithm. Since RelaySGD is approximating uniform model averaging, it may be useful to compare it with a baseline uniform model averaging algorithm. This section provides the implementation details for a baseline allreduce algorithm and the RelaySGD algorithm.

### 4.2.1 Allreduce Baseline

To provide a basis for the comparison of the algorithms, we need a good baseline algorithm that does not degrade in performance when data is heterogeneously distributed among workers. To achieve this, we can use uniform model averaging which is equivalent to all-reduce averaging in a fully connected network:

$$x_i^{(t+1)} = \frac{1}{n} \sum_{j=1}^{n} x_j^{(t)}$$

Algorithm 3 describes the allreduce baseline algorithm using mathematical notation.

**Algorithm 3** Allreduce Baseline

**Input** model weight $x$, learning rate $\gamma$

1: **for** $t = 0, 1, ..$ **do**
2: **for** node $i$ **in parallel**
3: $x_i^{(t+1/2)} = x_i^{(t)} - \gamma \nabla f_i(x_i^{(t)})$ ▷ Local model update (SGD)
4: $x_i^{(t+1)} = \frac{1}{n} \sum_{j=1}^{n} x_j^{(t+1/2)}$ ▷ Average with all models
5: **end for**
6: **end for**

As mentioned in the preliminaries section, *Bagua* provides different hooks that can be used to do computations and communication and very specific stages during the forward and backward propagation steps. For this algorithm I used the `post_optimizer_step_hook` to do the uniform model averaging after the optimization step described on Line 3 of Algorithm 3. The averaging step is done in one line with Bagua's `allreduce_inplace` method using the averaging reduce operator. I also used 2 helper methods `pack` and `unpack` to squeeze the model into a one-dimensional tensor to simplify the calculations that need to be performed. Algorithm 4 describes this hook in more detail.

**Algorithm 4** Bagua - Allreduce Baseline, Post Optimizer Step Hook

**Input** PyTorch optimizer *optimizer*

1: $x_i \leftarrow$ extracted params from *optimizer*
2: $x\_i\_buffered \leftarrow$ pack($x_i$)
3: allreduce_inplace($x\_i\_buffered$, $AVG\_Operator$)
4: $x_i \leftarrow$ unpack($x\_i\_buffered$)
5: *optimizer* $\leftarrow$ *optimizer* with overwritten averaged parameters $x_i$

### 4.2.2 RelaySGD

To implement RelaySGD, I again used the `post_optimizer_step_hook`. Algorithm 5 describes the steps inside of the hook in more detail. Here I also used the helper methods `pack` and `unpack` similar to Algorithm 4. Also, I use another helper method `sum_without_nb` which takes a dictionary of tensors and adds them all except for the element where the key given as the second parameter *nb*. The list of neighbors *Neighbors* is initialized with the current worker id when the algorithm is loaded and the dictionaries containing the received messages, `recv_messages` and `recv_counts`, are initialized empty.

**Algorithm 5** Bagua - RelaySGD, Post Optimizer Step Hook

**Input** PyTorch optimizer *optimizer*

$x_i \leftarrow$ extracted params from *optimizer*
$x\_buffered \leftarrow$ pack($x_i$)
**for** $nb \in Neighbors$ **do**
    bagua_send($nb$, sum_without_nb($recv\_messages$, $nb$) + $x\_buffered$)
    bagua_send($nb$, sum_without_nb($recv\_counts$, $nb$) + 1)
    $recv\_messages[nb] \leftarrow$ bagua_recv($nb$)
    $recv\_counts[nb] \leftarrow$ bagua_recv($nb$)
**end for**
$n \leftarrow 1+$ sum($recv\_counts$)
$x\_buffered \leftarrow \frac{1}{n}$ ($x\_buffered$ + sum($recv\_messages$))
$x_i \leftarrow$ unpack($x\_buffered$)
*optimizer* $\leftarrow$ *optimizer* with overwritten averaged parameters $x_i$

#### Topologies

For the implementation of the different topologies, I created a `Topology` superclass that contains a `neighbors` method which expects a worker ID and returns a list of this worker's neighbors. The three implemented topologies `ChainTopology`, `BinaryTreeTopology` and `DoubleBinaryTreeTopology` are all subclasses of the `Topology` superclass and essentially differ only in the `neighbors` method.

**ChainTopology** Given a worker ID $n$ the `neighbors` method of `ChainTopology` returns a list of the left and right neighbors, $n-1$ and $n+1$. If it is the leftmost worker $n = 0$, then it simply returns a list with the right neighbor 1 and vice versa for the rightmost worker.

**BinaryTreeTopology** We are using a complete binary tree [4] as topology because is greatly simplifies the calculation of the parent end child node

indices. For the given worker id $n$ and the total number of workers $N$ the children are $2n$ and $2n + 1$ if they are smaller than $N$ and its parent is $n/2$, if $n > 0$.

**DoubleBinaryTreeTopology** The `neighbors` method of this topology class returns 2 lists of neighbors. The first list is equivalent to the list that `BinaryTreeTopology` returns. Therefore, `DoubleBinaryTreeTopology` is a subclass of `BinaryTreeTopology`. Its `neighbors` methods first calls the `neighbors` method of the superclass, then calculates the indices of the complementary tree of the complete binary tree and returns the two lists. If $n = 0$ is the root node of a complete binary tree, then $n = N - 1$ is the root node of the complementary complete binary tree. The indices of the child and parent nodes are calculated accordingly.

Chapter 5

# Experimental Evaluation

In this final section, I evaluate the performance of the RelaySGD implementation. I first examine the performance differences of RelaySGD's 3 implemented topologies and then compare RelaySGD with Bagua's implementation of D-PSGD and the allreduce baseline across different data distributions. Then I benchmark the three algorithms in terms of system speedup. Finally, I discuss the results, limitations and possible improvements.

## 5.1 Experimental Setting

One of the objectives of this thesis is to verify the implementation of RelaySGD within Bagua. For this reason, I use the same dataset and model as the authors of the RelaySGD paper: Cifar-10 [11] with the VGG-11 [20] architecture. The model was trained for 200 epochs and the best learning rate was selected using a naive grid search over the interval $[0.01 - 0.1]$. Since the training time for these experiments is relatively long (1-2 hours for each run), I only performed the grid search for 50 epochs. The experimental evaluation is performed on a single node with 8 NVIDIA TITAN Xp GPUs.

## 5.2 Effect of network topology

I have mentioned three different topologies in previous chapters that can be used to implement RelaySGD: Chain, Binary Tree, and Double Binary Trees. In this experimental setup, the double binary trees outperformed the binary tree and chain network by about 0.5% under all of the three data distributions. The difference in terms of accuracy is shown in Figure 5.1.

This result is somewhat expected. With only 8 GPUs, the amount of messages exchanged with neighbors between a chain and a binary tree is not that different, therefore the performance is only marginally better. But the double binary trees allow for communication with more devices at the same time as the chain or the binary tree, therefore there is a small increase in terms of accuracy.

## 5.3 Effect of data heterogeneity

In the previous chapter I introduced the Dirichlet distribution $Dir(\alpha)$, where $\alpha$ is the parameter that determines heterogeneity. $\alpha = 1$ generates a homoge-

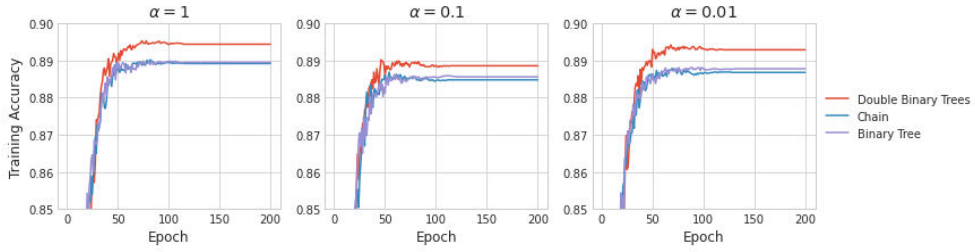



**Figure 5.1:** Performance of RelaySGD on three different network topologies (chain, binary trees and double binary trees) and under three data distributions ($\alpha = 1$ homogeneous, $\alpha = 0.1$ heterogeneous, $\alpha = 0.01$ very heterogeneous). Learning rates are tuned to reach the highest accuracy in 50 epochs on the CIFAR-10 dataset trained on VGG11 with 8 GPUs.

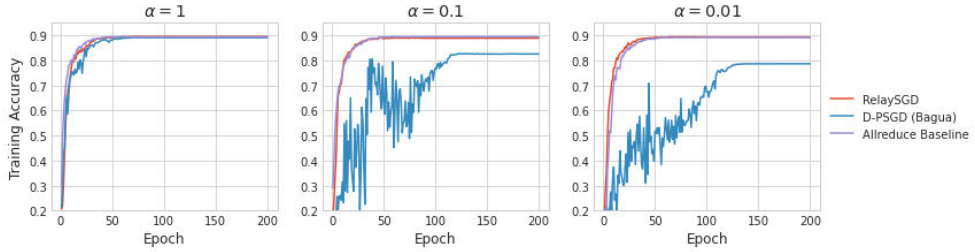



**Figure 5.2:** Performance of RelaySGD, D-PSGD and allreduce model averaging and under three data distributions ($\alpha = 1$ homogeneous, $\alpha = 0.1$ heterogeneous, $\alpha = 0.01$ very heterogeneous). Learning rates are tuned to reach the highest accuracy in 50 epochs on the CIFAR-10 dataset, trained on VGG11 with 8 GPUs.

neous distribution, $\alpha = 0.1$ a heterogeneous distribution and 0.01 a very heterogeneous distribution. In this section, I compare D-PSGD and RelaySGD with the allreduce baseline algorithm with respect to those 3 levels of data distribution heterogeneity. Figure 5.2 shows that as heterogeneity increases, the performance of RelaySGD and the allreduce baseline algorithm does not seem to degrade. On the other hand, it can clearly be seen that D-PSGD has a much higher volatility and also converges with much lower accuracy.

Unfortunately, Figure 5.2 does not clearly show the difference between RelaySGD and the allreduce model averaging algorithm. Figure 5.3 shows a more detailed comparison of these two algorithms. Here it can be seen that in some cases, RelaySGD slightly exceeds the allreduce baseline. This is likely due to the special topology and update behavior of double binary trees.

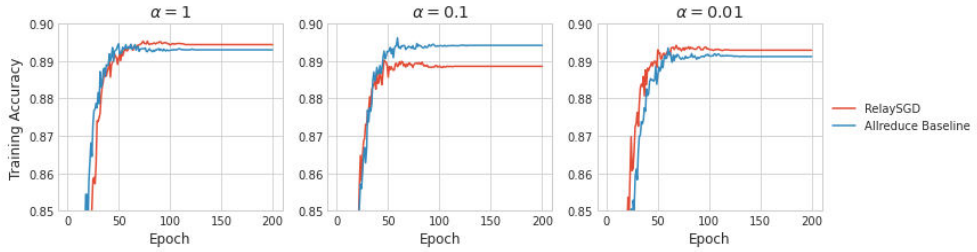



**Figure 5.3:** Performance of RelaySGD and allreduce model averaging and under three data distributions ($\alpha = 1$ homogeneous, $\alpha = 0.1$ heterogeneous, $\alpha = 0.01$ very heterogeneous). Learning rates are tuned to reach the highest accuracy in 50 epochs on the CIFAR-10 dataset, trained on VGG11 with 8 GPUs.

## 5.4 System speedup

The Bagua repository [6] provides a `synthetic_benchmark.py` script to benchmark system speedup. I also adopted this script to run the algorithm. Table 5.1 shows the results. It is clear that RelaySGD does not provide a system speedup under the network conditions within a single node. D-PSGD's communication step is implemented in the Rust backend of Bagua, and the `allreduce_inplace` operation that I used in the `AllreduceAlgorithm` implementation, is quite efficient, as I mentioned in a previous chapter. RelaySGD's communication mechanism is implemented only in Python, so implementing it in Rust may increase the system speedup.

**Table 5.1:** Total img/sec on 8 GPUs using the ResNet50 [8] model.

| Algorithm | Topology | Throughput |
|---|---|---|
| Allreduce (baseline) | fully connected | **1346.5** $\pm$ 10.5 |
| D-PSGD (Bagua) | fully connected | **1334.7** $\pm$ 37.4 |
| RelaySGD | chain | **1208.5** $\pm$ 6.4 |
| RelaySGD | binary tree | **1010.6** $\pm$ 10.4 |
| RelaySGD | double binary trees | **847.3** $\pm$ 5.1 |

Chapter 6

# Conclusion

Similar to the authors from the RelaySGD paper [23], I observe there to be a significantly higher accuracy for RelaySGD compared to D-PSGD for more hydrogenous data distributions, which is very close to the ideal allreduce baseline, indicating that this algorithm is optimal in terms of convergence. I also found that the double binary tree topology of RelaySGD performs better compared to the other topologies. That being said, RelaySGD is not necessarily the best solution for training in a datacenter setting, as it is easy to produce a homogeneous distribution of data in most cases. This algorithm is probably more appropriate for federated learning, where heterogeneous data distributions are more common. It would be interesting to see experiments that demonstrate the performance of RelaySGD in a federated learning setting.

Furthermore, it would be interesting to see the performance speed benchmark under different network conditions, especially with more devices. The handshaking theorem [1] states that the sum of degrees of the vertices of a graph is twice the number of edges. Depending on the topology used for the allreduce operation, there is a lower bound for the messages sent of $2(2n-2)$ for tree allreduce and $n^2$ for the naive allreduce. This means that RelaySGD only sends $2(n-1)$ messages with a binary tree topology or $2(2n-2)$ times half of the model for double binary trees. So in total, it is theoretically only sending at most half of the number of models compared to the full allreduce model averaging. As mentioned in the previous section, I was not able to observe this theoretical speedup because my RelaySGD implementation in Python does not properly leverage Baguas communication primitives, since I ama using a novel communication mechanism. Accordingly, I would need to implement the relay communication mechanism in the Rust backend of Bagua. Therefore, with an efficient implementation in Rust, RelaySGD can potentially optimize bandwidth, It can be argued that in this case it has a place in a data center environment, especially when bandwidth is a bottleneck.

Finally, it would also be interesting to see experiments examining the robustness of RelaySGD, e.g. how it tolerates randomly dropped messages.

Chapter 7

# Acknowledgements

I would like to thank Prof. Ce Zhang for the opportunity to work on this project and Binhang Yuan for his guidance and feedback during our weekly meetings. I wish to extend my special thanks to Yafen Fang for her continuous technical support and help in understanding Baguas system.

Appendix A

# Learning Rate Tuning

To find the optimal learning rate for each of the three algorithms, I ran a naive grid search for 50 epochs each. The following tables show the accuracy of each algorithm with different learning rates, trained on 8 GPUs on the Cifar-10 dataset and with the VGG-11 model.

**Table A.1:** Allreduce beasline algorithm in Bagua

| learning rate $\gamma$ | $\alpha = 1$ | $\alpha = 0.1$ | $\alpha = 0.01$ |
|---|---|---|---|
| 0.01 | 0.8688 | 0.8683 | 0.8691 |
| 0.025 | 0.8858 | 0.8810 | 0.8822 |
| 0.05 | **0.8923** | 0.8878 | 0.8902 |
| 0.075 | 0.8887 | **0.8944** | **0.8926** |
| 0.1 | 0.8905 | 0.2320 | 0.8894 |

**Table A.2:** D-PSGD in Bagua

| learning rate $\gamma$ | $\alpha = 1$ | $\alpha = 0.1$ | $\alpha = 0.01$ |
|---|---|---|---|
| 0.01 | 0.8668 | 0.5543 | 0.2970 |
| 0.025 | 0.8789 | 0.6472 | **0.5257** |
| 0.05 | 0.8846 | **0.8074** | 0.3379 |
| 0.075 | **0.8872** | 0.4863 | 0.1000 |
| 0.1 | 0.8837 | 0.2851 | 0.1000 |

**Table A.3:** RelaySGD in Bagua

| | Double Binary Trees | | | Binary Tree | | | Chain | | |
|---|---|---|---|---|---|---|---|---|---|
| lr $\gamma$ | $\alpha = 1$ | $\alpha = 0.1$ | $\alpha = 0.01$ | $\alpha = 1$ | $\alpha = 0.1$ | $\alpha = 0.01$ | $\alpha = 1$ | $\alpha = 0.1$ | $\alpha = 0.01$ |
| 0.01 | 0.7060 | 0.4642 | 0.3093 | 0.6343 | 0.5392 | 0.7231 | 0.7381 | 0.7135 | 0.7597 |
| 0.025 | 0.8568 | 0.8634 | 0.8638 | 0.8512 | 0.8575 | 0.8649 | 0.8553 | 0.8646 | 0.8644 |
| 0.05 | 0.8759 | 0.8757 | 0.8785 | 0.8762 | 0.8729 | 0.8819 | 0.8781 | 0.8763 | 0.8798 |
| 0.075 | 0.8833 | 0.8870 | 0.8848 | 0.8853 | **0.8885** | **0.8862** | 0.8850 | **0.8881** | 0.8792 |
| 0.1 | **0.8896** | **0.8927** | **0.8868** | **0.8884** | 0.1000 | 0.1000 | **0.8872** | 0.1000 | **0.8866** |

# Bibliography


[1] "handshaking lemma". *The Concise Oxford Dictionary of Mathematics*. https://www.oxfordreference.com/view/10.1093/oi/authority.20110803095919242, [Online; accessed 08.07.2022].

[2] Bagua contributors. How to create a new algorithm - Bagua tutorials, 2022. https://tutorials.baguasys.com/how-to-create-a-new-algorithm/, [Online; accessed 04.07.2022].

[3] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. signSGD: Compressed optimisation for non-convex problems. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 560–569. PMLR, 10–15 Jul 2018.

[4] Paul E. Black. "complete binary tree". *Dictionary of Algorithms and Data Structures*, 2016. https://xlinux.nist.gov/dads/HTML/completeBinaryTree.html, [Online; accessed 04.07.2022].

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[6] Shaoduo Gan, Xiangru Lian, Rui Wang, Jianbin Chang, Chengjun Liu, Hongmei Shi, Shengzhuo Zhang, Xianghong Li, Tengxu Sun, Jiawei Jiang, Binhang Yuan, Sen Yang, Ji Liu, and Ce Zhang. Bagua: Scaling up distributed learning with system relaxations, 2021.

[7] Chaoyang He, Songze Li, Jinhyun So, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, Li Shen, Peilin Zhao, Yan Kang, Yang Liu, Ramesh Raskar, Qiang Yang, Murali Annavaram, and Salman Avestimehr. Fedml: A research library and benchmark for federated machine learning. *CoRR*, abs/2007.13518, 2020.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[9] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020.

[10] Peter H. Jin, Qiaochu Yuan, Forrest N. Iandola, and Kurt Keutzer. How to scale distributed deep learning? *ArXiv*, abs/1611.04581, 2016.

[11] Alex Krizhevsky. Learning multiple layers of features from tiny images, 2009.

[12] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, Tom Duerig, and Vittorio Ferrari. The open images dataset v4. *International Journal of Computer Vision*, 128(7):1956–1981, mar 2020.

[13] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training, 2020.

[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 I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[15] Tao Lin, Sai Praneeth Karimireddy, Sebastian Stich, and Martin Jaggi. Quasi-global momentum: Accelerating decentralized deep learning on heterogeneous data. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 6654–6665. PMLR, 18–24 Jul 2021.

[16] Ji Liu and Ce Zhang. *Distributed Learning Systems with First-Order Methods*. 01 2020.

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach,

H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[18] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009. Selected papers from the 14th European PVM/MPI Users Group Meeting.

[19] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[20] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[21] Hanlin Tang, Xiangru Lian, Ming Yan, Ce Zhang, and Ji Liu. $d^2$: Decentralized training over decentralized data. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4848–4856. PMLR, 10–15 Jul 2018.

[22] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. A survey on distributed machine learning. *ACM Comput. Surv.*, 53(2), mar 2020.

[23] Thijs Vogels, Lie He, Anastasia Koloskova, Sai Praneeth Karimireddy, Tao Lin, Sebastian U Stich, and Martin Jaggi. Relaysum for decentralized deep learning on heterogeneous data. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

[24] Huasha Zhao and John F. Canny. Sparse allreduce: Efficient scalable communication for power-law data. *CoRR*, abs/1312.3020, 2013.

[25] Zhuangdi Zhu, Junyuan Hong, and Jiayu Zhou. Data-free knowledge distillation for heterogeneous federated learning. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 12878–12889. PMLR, 18–24 Jul 2021.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

---

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

Exploring Distributed Learning Algorithms with Communication Optimization

**Authored by** (in block letters):

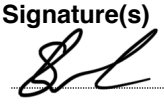
*For papers written by groups the names of all authors are required.*

| **Name(s):** | **First name(s):** |
|---|---|
| Bohl | Samuel |
| | |
| | |
| | |
| | |

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

| **Place, date** | **Signature(s)** |
|---|---|
| Zurich, 31.07.2022 |  |
| | |
| | |
| | |
| | |

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*