UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ANA LUISA VERONEZE SOLÓRZANO

A Practical Evaluation of Parallel and Distributed Deep Learning Frameworks

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof. Dr. Lucas Mello Schnorr

Porto Alegre
Dezembro 2021

CIP — CATALOGING-IN-PUBLICATION

Veroneze Solórzano, Ana Luisa

A Practical Evaluation of Parallel and Distributed Deep Learning Frameworks / Ana Luisa Veroneze Solórzano. – Porto Alegre: PPGC da UFRGS, 2021.

84 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2021. Advisor: Lucas Mello Schnorr.

1. Distributed Deep Learning, Performance Analysis, HPC, DDL Frameworks. I. Mello Schnorr, Lucas. II. Título.

“A ship in port is safe, but that is not what ships are built for.”

— GRACE HOPPER

ACKNOWLEDGEMENTS

I want to thank my advisor, professor Lucas Mello Schnorr. First, for the opportunity of working with him as his advised student. Second, for all the support and patience during my research projects. Thank you for guiding me during my research and sharing insightful comments and resources. I hope to share the tip of several icebergs with the world throughout my research in the following years.

To my parents, who were supportive in these challenging times of a pandemic, especially to my sister, who was always excited listening to me explain my research topics even though she does not understand much of it.

To all the friends I made during my journey in Computer Science. The path is much more comfortable if you feel you belong to it. Thank you for the virtual conversations and pieces of advice.

To CAPES (Coordination for the Improvement of Higher Education Personnel) for granting me the scholarship that supported my research. To the jury members for this work and all comments and suggestions presented.

The experiments used during my research were carried out using the Parque Computacional de Alto Desempenho (PCAD) at UFRGS and the Grid'5000 testbed supported by Inria, CNRS, RENATER, and other organizations.

ABSTRACT

The computational power growth in the last years and the increase of data to be processed contributed to researchers in deep learning update their models to use distributed training. Distributed Deep Learning (DDL) is essential for solving large-scale problems faster and accurately using multiple devices to run the model in parallel. This strategy brings challenges to improving the training performance without losing accuracy and without increasing the overhead of exchanging data between host and devices. Frameworks for DDL have become popular alternatives in the last years for training using multiple devices, running on top of usual machine learning libraries. They are advantageous for final users since they require only a few extra lines of code to a single-node model script. However, from a High-Performance Computing (HPC) perspective, it is challenging to evaluate distributed training performance since the frameworks hide the implementation's details. The use of performance analysis methodologies and visualization tools common to the HPC field can benefit DDL frameworks' users to choose the best framework for their model and can benefit DDL frameworks' developers by providing insights on how to optimize their applications. This work presents a performance analysis and comparison of two modern frameworks: Horovod, one of the most popular DDL frameworks used worldwide, and Tarantella, a recent framework with the same parallel strategy as Horovod but with a different all-reduce algorithm and distributed library. Our results showed that combining HPC and Machine Learning tools to evaluate the performance of DDL can enrich the findings and identify bottlenecks in the frameworks. Horovod presented higher scaling efficiency than Tarantella, with a difference of almost 50% in their efficiency scaling from four to twelve GPUs. Although Horovod all-reduce algorithm trains faster than Tarantella, the last presented higher model accuracy. Using a temporal aggregation, we also identified the exact time spent computing and communicating during training, which can benefit developers in improving the frameworks. Since our approach is implemented at the DDL framework level, it can also be used to analyze the performance of other neural network models.

Keywords: Distributed Deep Learning, Performance Analysis, HPC, DDL Frameworks.

Avaliação Prática de Frameworks para Deep Learning Paralelo e Distribuído

RESUMO

O aumento do poder computacional e de dados disponíveis nos últimos anos contribuiu para que pesquisadores em Aprendizado Profundo atualizassem seus modelos para usar treinamento distribuído. Aprendizado Profundo Distribuído (APD) é essencial para resolver problemas de grande escala mais rapidamente e de forma precisa usando múltiplos dispositivos para treinarem em paralelo. Esta estratégia traz desafios em otimizar o desempenho do treino sem perder a acurácia e sem gerar sobrecarga com comunicações entre servidor e dispositivos. Frameworks para APD se tornaram uma alternativa para treinar redes neurais, executando sobre bibliotecas de Aprendizado de Máquina (AM). Esses frameworks são vantajosos para usuários finais, pois requerem algumas novas linhas de código no script não-distribuído. No entanto, da perspectiva the Computação de Alto Desempenho (CAD), a avaliação do treinamento distribuído é um desafio, pois os frameworks escondem detalhes de suas implementações. O uso de metodologias aplicadas em análise de desempenho e ferramentas para visualização comuns à área de CAD podem beneficiar usuários dos frameworks a escolherem o melhor para seu modelo, e também desenvolvedores dos frameworks a identificarem indicativos de como otimizá-los. Este trabalho apresenta uma avaliação de desempenho e comparação entre dois modernos frameworks para APD: Horovod, um dos mais populares usado mundialmente, e Tarantella, mais recente com a mesma estratégia de paralelização que o Horovod, mas com diferentes algoritmos e padrões para comunicação em sistemas distribuídos. Os resultados mostram que combinar ferramentas de CAD e de AM para avaliar o desempenho de frameworks para APD enriquecem a análise de desempenho e ajudam a identificar gargalos nos frameworks. Horovod apresentou a maior eficiência escalando de quatro à oito GPUs, com uma diferença de quase 50% em relação ao Tarantella. Embora o algoritmo do Horovod treine mais rápido do que o do Tarantella, este apresentou maior acurácia do modelo. Usando agregação temporal, pode-se identificar o tempo gasto com computação e com comunicação, o que pode beneficiar desenvolvedores a melhorarem seus frameworks. Nossa abordagem pode ser usada para análise de desempenho de diversos modelos de redes neurais artificiais, pois foi implementada a nível dos frameworks.

Palavras-chave: Distribuição de Aprendizado Profundo, Análise de Desempenho, HPC.

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Intelligence
API	Application Programming Interface
CNN	Convolutional Neural Network
CUPTI	NVIDIA CUDA Profiling Tools Interface
cuDNN	NVIDIA CUDA Deep Neural Network library
DDL	Distributed Deep Learning
DL	Deep Learning
DNN	Deep Neural Network
GASPI	Global Address Space Programming Interface
GD	Gradient Descent
GPI-2	Global address space Programming Interface
HPC	High-Performance Computing
MNIST	Modified National Institute of Standards and Technology database
ML	Machine Learning
MPI	Message-Passing-Interface
MSGD	Minibatch Stochastic Gradient Descent
NN	Neural Network
NVSMI	NVIDIA System Management Interface
NCCL	NVIDIA Collective Communication Library
OpenMP	Open Multi-Processing
PCAD	Parque Computacional de Alto-Desempenho
SGD	Stochastic Gradient Descent
SPMD	Single-Program Multiple-Data

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Artificial Neural Networks are mathematical models used by computers to handle real-world problems inspired by the brain's capacity to perform perceptual and cognitive tasks (LAINE, 2003). It can be used for scientific research problems involving, for example, object detection, image classification, genomics, and text categorization studies (SZEGEDY; TOSHEV; ERHAN, 2013; KRIZHEVSKY; SUTSKEVER; HINTON, 2017; ZHANG; ZHOU, 2006).

Deep Neural Networks (DNN), as the human brain's biological neural network, also have to learn from unknown data using hierarchical training. This training uses mathematical rules to determine changes in the neuron's connectivity. The network computes predictions considering what it learned, so the first levels of the network improve the learning of the deeper levels (LECUN; BENGIO; HINTON, 2015). Linear algebra plays an important role in Machine Learning (ML) since the networks and parameters are represented as matrices and vectors, suffering multiplications, transposing, pooling, and convolution operations.

Training a DNN using big datasets can take days. Despite the training accuracy and efficiency, researchers are also concerned about increasing the training's performance execution. The use of parallel and distributed environments is essential to increase the performance of applications that deal with many data and perform exhaustive computations. Moreover, DNN models perform several matrices multiplications that can be effectively implemented in a GPU (OH; JUNG, 2004).

Different approaches have been presented over the last years to accelerate the learning process. Initially, they rely on exploring the network parameters variation, such as loss and activation functions (KARLIK; VEHBI, 2011). The next approaches started exploring architectural innovations on the network design, including increasing the depth and the width of models (ZAGORUYKO; KOMODAKIS, 2016). Libraries that are references in ML, as TensorFlow and Theano, currently support the training distribution over parallel resources in multi-node environments (ABADI et al., 2016; MA HEAND MAO; TAYLOR, 2017). However, they require a deeper knowledge of parallel and distributed models using programming paradigms as CUDA and MPI, and also new algorithms to handle distributed computing.

Distributed Deep Learning (DDL) frameworks can take advantage of large-scale hybrid systems (e.g., CPU and GPU) to speed up the training of DNN without extra ef-

fort from the programmer side (BEN-NUN; HOEFLER, 2019). These tools run over ML frameworks to distribute a single-device training to a multi-node system with multiple devices (GPUs, TPUs, and CPUs). New DDL frameworks are being launched constantly due to the fast development of the field with support to different ML frameworks, using different parallel programming paradigms and parallelization strategies. Horovod and Tarantella are two modern frameworks for systems with CPUs and GPUs that offer an interface to run over ML libraries like Keras and TensorFlow and distribute serial training to multiple computational nodes adding few lines of code (SERGEEV; BALSO, 2018; CCHPC, 2020). They share similarities in their installation, usage, and distribution strategy, but they use different all-reduce algorithms and distributed libraries.

Performance evaluation on DDL usually focuses on weak scalability to more devices and measures training time and accuracy. There is a lack of evaluations and comparisons of parallel and distributed frameworks from a High-Performance Computing (HPC) perspective. As we mentioned, the frameworks usually hide details about the distribution implementation, facilitating the user configuration using a few lines of code. From the perspective of HPC programmers, it is a challenge to perform an in-depth analysis of how tools as Horovod and Tarantella are taking advantage of the distributed approaches and the devices compared to usual parallel and distributed applications.

With a practical evaluation and methodology to use state-of-the-art HPC tools and visualization methods, we break the framework's "black box" and better understand the performance results, the resources usage, the frameworks synchronization algorithms, and methods used to distribute the training. We performed experiments over four clusters with varied NVIDIA GPU models using a Convolutional Neural Network (CNN) with varied batch size using strong-scaling. We found that Horovod presents higher scaling efficiency than Tarantella, up to 50% faster, and processes batches faster than Tarantella, optimizing the time synchronizing data. Tarantella, on the other hand, presented higher accuracy for most cases and also a faster initialization time. Using space/temporal visualizations of the profiler and tracing measurements, we identified bottlenecks in the Tarantella implementation. Furthermore, correlating measures from different tools, we depicted the time spent with computations in the GPUs during each epoch. These results can benefit DDL frameworks' developers to improve their tools.

1.1 Motivation

The advances of heterogeneous supercomputers and availability of big datasets, motivated researchers in DL to innovate the field of AI by accelerating the training when distributing it to multiple devices in an easy way to use. The advantage of using DDL frameworks from the user side is that they can be applied over a training script for a single device by adding a few extra lines of code. However, from the researchers' and developers' perspectives, these frameworks hide details of their implementation, such as synchronization algorithms, communication strategies, computing time and devices usage. Performance evaluations of DDL frameworks usually focus on their scalability to more devices and focus on measuring and evaluating the training time and accuracy (RAVIKUMAR; S., 2020). Considering the fast advances in the field, with new DDL frameworks being created every year, and new advances on using accelerators to process linear algebra operations more efficiently, it is required a performance analysis of DDL from an HPC perspective along with the training efficiency evaluation. Furthermore, it would be necessary to start comparing these frameworks over different devices, number of workers, and model parameters. This work performs a practical evaluation and comparison of the modern DDL frameworks Horovod and Tarantella, using state-of-the-art HPC and ML tools and methodology for performance analysis of distributed applications. We propose the use of tracing and data science to evaluate the frameworks' strengths and weaknesses and depicts their distributed implementation. An evaluation as that reveal uncovered bottlenecks that impact the performance of the frameworks, and can also benefit users to decide which framework to use to train Convolutional Neural Networks (CNN) on multi-GPUs and multi-node systems.

1.2 Contributions

This research proposes a performance analysis and comparison of two modern DDL frameworks: Horovod and Tarantella. We applied a new methodology to evaluate and compare the frameworks' performance correlating measurements obtained with state-of-the-art HPC and ML tools. The main contributions of this research are:

- Experiments on four clusters multi-GPUs with different GPU cards for up to 12 devices using the Lenet-5 CNN, and the MNIST dataset for evaluating Horovod

and Tarantella training performance. We use strong-scaling varying the batch sizes for 100 epochs. We evaluate the training time, the training accuracy, the GPUs usage, and the scaling efficiency to more GPUs.

- Experiments using the NVProf tracing tool to obtain events within GPUs for both frameworks. We also propose a Horovod instrumentation with Score-P to get information about the MPI operations used in the distributed training.
- A method to correlate measurements from HPC and ML tracing and profiling tools and visualizations using data science. We used temporal synchronization to correlate results captured in different tools and a temporal aggregation to measure the frameworks' training time in the GPUs.

1.3 Document Structure

This document is organized as follows. Chapter 2 presents a background on DL and DDL, including methods and strategies for distributing the training. Chapter 3 presents the related work on evaluating the performance of DDL frameworks and addresses several existing frameworks and their characteristics. Chapter 4 presents the methodology used to evaluate the framework's performance, our experimental design, and approaches for profiling and tracing the experiments. Chapter 5 presents the experimental results for a CNN model for Horovod and Tarantella in three different NVIDIA GPU models. Chapter 6 presents our approach to collect and analyze execution traces, bringing insightful aspects of the framework's functionality. Finally, Chapter 7 presents the conclusions and future works based on our findings.

2 BACKGROUND

Deep Learning (DL) is an area of Machine Learning that became popular in the last fifteen years due to the increase in the amount of data available for training, and the improvements in the hardware and in the software used for training (LECUN; BENGIO; HINTON, 2015; DENG; YU, 2014; GOODFELLOW; BENGIO; COURVILLE, 2016). DL consists of applying algorithms with linear algebra operations on multiple processing layers of a model to train it using an input dataset with supervised, semi-supervised, or unsupervised learning (DENG; YU, 2014).

Supervised learning is a method where we feed our model with representative data of the category being analyzed, more specifically, data that is labeled (SATHYA; ABRAHAM et al., 2013). For example, for an image classification model to classify cats and dogs, we first train the network with many images of cats and many images of dogs to make the network learn so it can later perform classification with new images. Unsupervised learning receives unlabeled data, so it applies a heuristic to identify patterns in the input data and learn from trial. Semi-supervised learning combines few labeled data with several unlabeled data.

In this Chapter, we present a background of DL, from the fundamentals until the advances that lead to DDL frameworks creation. Section 2.1 presents the fundamentals of DL, Section 2.2 chronologically presents the advances in DL until it became a field explored in HPC, Section 2.3 presents DDL methods and strategies for distributing training.

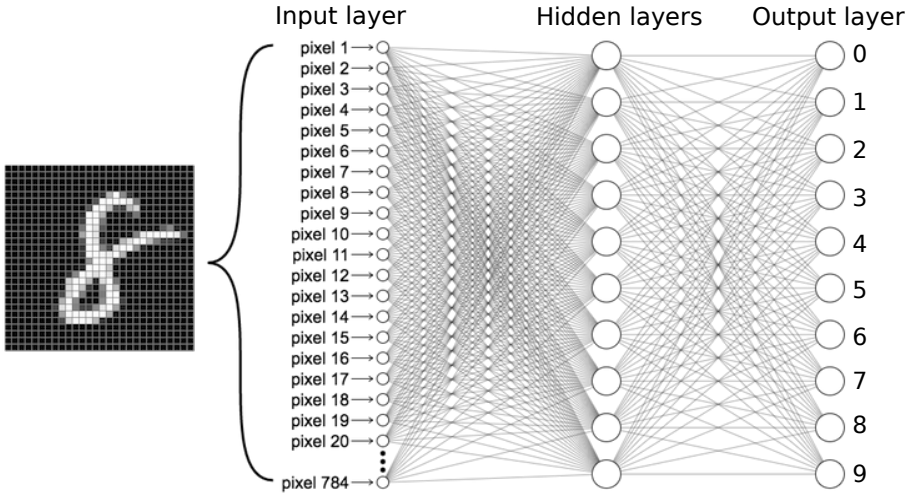
2.1 Fundamentals of Deep Learning

Artificial Neural Networks (ANN) are networks of interconnected neurons that represent mathematical functions to transform input data into the desired output, inspired by our biological neural network (SATHYA; ABRAHAM et al., 2013). The main contribution of ANN is the low level of programming complexity to solve complex and nonlinear problems in recognition, diagnosis, classification, predictions, and filtering (GRAUPE, 2013). For example, in a classical ML problem to classify images, the input data can be a vector where each vector element represents a pixel, and a pixel represents a neuron of the network.

There are different neural network architectures, with variations in their processing layers, how neurons are interconnected, and how they communicate. The most pop-

ular are feed-forward networks, composed of layers with several artificial neurons that communicate with each other to achieve the best prediction. Figure 2.1 represents an example of a Deep Feed Forward network for recognizing hand-written digits as a number from 0 to 9. The first layers, closest to the input layer from left to right, are also called the lower levels, and the last layers, closer to the output layer, are also known as the highest levels. There are three specific layers: the input layer, hidden layers, and an output layer. The input layer receives the training's input data; in this case, each pixel of the input image is mapped to a neuron. Each neuron receives and outputs to all neurons a value representing its activation, if 1, or deactivation, if 0. It goes until achieving the output layer. A neuron can be represented by a perceptron or a sigmoid. A sigmoid is the most common neuron variation created to achieve more efficient learning by accepting values between 0 and 1, so slight prediction updates can result in subtle changes in values.

Figure 2.1 – Example of a deep neural network architecture.

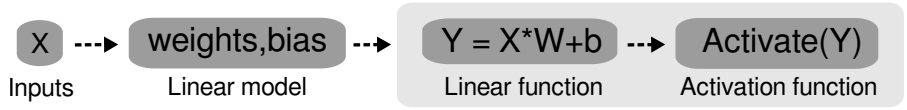


Source: adapted from (ML4A, 2018)

The activation of each neuron considers the *bias* and *weights* randomly initialized. A weight is expressed as a real number that determines the "strength" of the connections of the neurons. The bias determines how activated the neuron should be when passing its information through the network, helping to speed up the prediction. Linear Regression is one of the most popular algorithms for predictive analysis based on given input variables. Figure 2.2 represents a linear regression model applied to a NN for a single neuron. The light gray delimits the neuron, which receives inputs associated with weights and biases and applies a linear function. Equation 2.1 depicts the linear function, where N represents the total inputs, $X = \{X_n \mid 1 < n \leq N\}$, with N weights associated to them, $W = \{W_n \mid 1 < n \leq N\}$. The output is applied to an **activation function** that will convert the value to a new signal deciding what information will pass to the next neuron. There are

different activation functions available, depending on the goal of the NN.

Figure 2.2 – Linear regression model applied in ANN.



Source: The Author

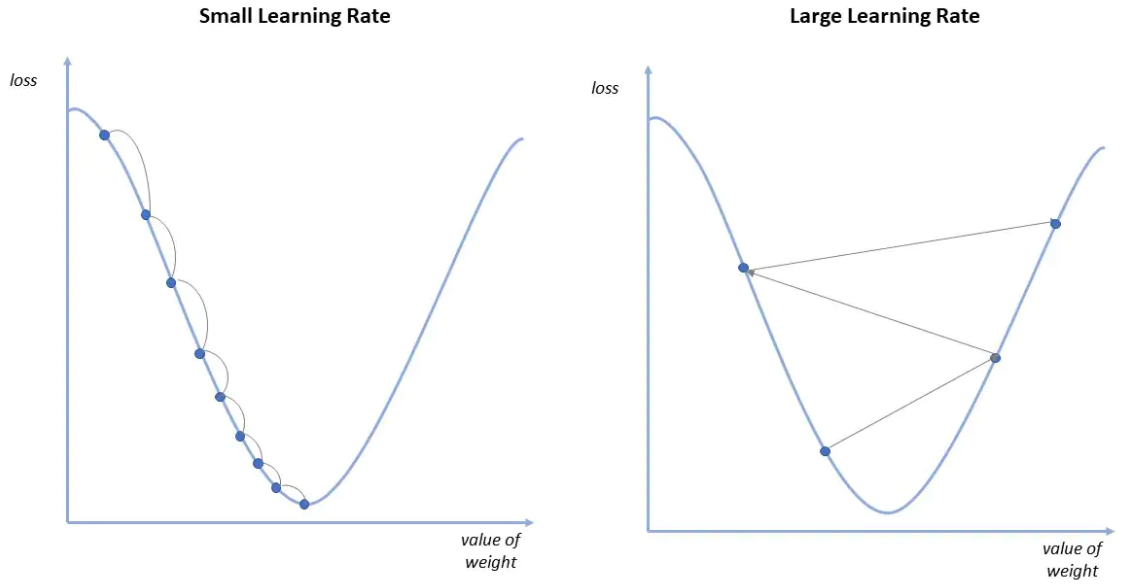
$$Y = \sum_{n=1}^{N} X_n \times W_n + b \tag{2.1}$$

We use a **cost function**, or loss function, to update the activation and make networks learn through data. It shows the error between the predicted output and the actual output, determining how bad is the network classifying data using the current weights and bias. The most common algorithm to minimize the cost function in a multi-layer network is the Gradient Descent (GD). It considers a **learning rate**, and the derivative of the loss function in terms of the weights and bias to minimize the loss function, the error of the prediction (BENGIO, 2012). We can then adjust our weights based on that minimization when completing a pass through the network. Figure 2.3 presents the learning rate, which is a technique to speed the convergence to a minimum. It represents the steps taken to reach the minimum of the cost function, so a small learning rate, as shown in the left plot, will be more efficient in reaching a minimum. Still, the training will take longer to achieve the minimum, and a large learning rate in the right side plot will converge faster but can overshoot the minimum.

Considering the vast amount of data available, the more used version of Gradient Descent today is the Minibatch Stochastic Gradient Descent (SGD), which splits the dataset into small samples of `N` examples, and compute the cost function over the samples until pass through all datasets (LECUN; BENGIO; HINTON, 2015). So for a pass-through 1000 data samples and minibatch sizes of 100, there will be ten iterations to pass and average all minibatch per epoch in the model. Equation 2.2 presents the SGD idea, where w are the weights, initialized as the same for all minibatches, $\nabla l(x, w_t)$ is the loss function, B is a minibatch sampled from x, n is the minibatch size, which makes it equivalent to the GD if set to 1, η is the learning rate, and t indicates the index of the iteration (GOYAL et al., 2018).

$$w_{t+1} = w_t - \eta \frac{1}{n} \sum_{x \in B} \nabla l(x, w_t) \tag{2.2}$$

Figure 2.3 – Convergence for a small and a large learning rate.



Source: (EDUCATION, 2020)

SGD is more computationally efficient than passing all dataset in the network and storing it in memory. When n increases, we can benefit from the matrix operations computed in the selected device, generating more stable results regarding the accuracy, and we perform fewer updates per computation, but it makes the model converge slower (BENGIO, 2012). A smaller n will perform more updates, converge faster, but it may underuse the devices. The minibatch size, also called only "batch size", is usually defined by the user. Its choice depends on the system utilized, such as the memory limitations to fit a minibatch in the memory.

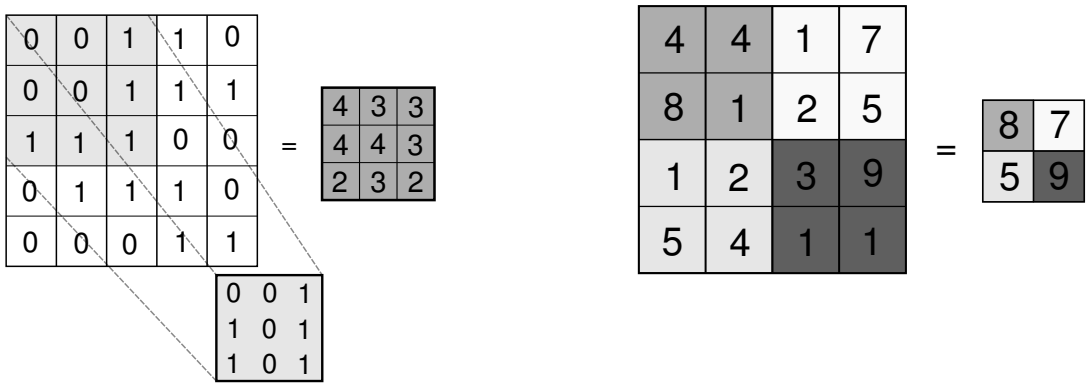
The previous process is called **forward-propagation**, where the inputs from lower hidden layers impact the signals of neurons in higher layers. A **back-propagation** process is used to update the network's weights, based on the cost function. This process performs a pass through the network from the highest layers to the lower ones to update the gradients based on the averaged loss calculated during the forward-propagation (RUMELHART; HINTON; WILLIAMS, 1986). The forward and backward pass of the entire input data over the network is called an **epoch**. Full training runs hundreds of epochs to achieve the best prediction.

In the previous example, we showed a simple model where we map each image to a neuron, but it is infeasible to deal with thousands of images. Convolutional Neural Networks (CNN), also called ConvNets, are a popular class of neural networks used for image classification due its lower computing time and high accuracy than other models (JORDÀ; VALERO-LARA; PEÑA, 2019). The first CNN was created in 1989, called LeNet, and

added convolutional layers to the hidden layers of the network model (LECUN et al., 1989). The convolutional layers process at least three basic steps: convolution, pooling, and activation functions. In these layers, the CNN performs image transformations to be less memory costly to pass through the network and benefit from GPUs ability to perform matrix operations.

Figure 2.4a represents the convolution operation. Each convolution layer applies filters to detect a pattern in the image, such as edges, corners, and objects. The filter is represented as an N × N matrix that slides over the input image pixels performing a convolution, updating its values based on the dot product with the image pixels. This way, the input image will be transformed into a feature map, or activation map, which is a matrix containing the image's highlights, with the same or reduced size or dimensions. The feature map is passed to a pooling layer, represented in Figure 2.4b. The pooling applies an N × N filter over the feature map and gets the maximum value of all pixels inside the pooling filter (*max pooling*) or the average of the values (*average pooling*). It reduces the map dimensions by a factor of N. After these operations, the non-linear activation functions are applied to these reduced maps at the fully connected layer to obtain the weighted sum of multiple input elements.

Figure 2.4 – Example of the convolution (a) and pooling (b) operations.



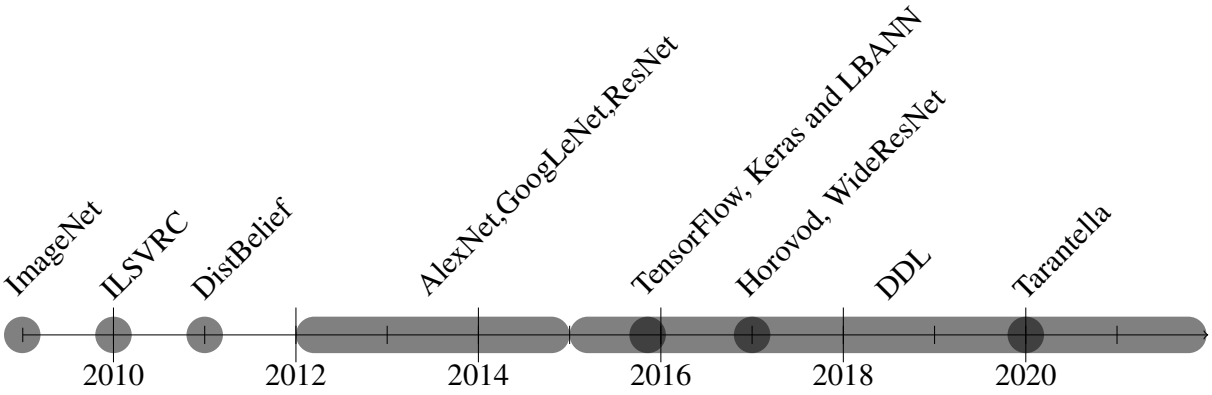
(a) Convolution operation. (b) Pooling operation using max pooling.

Source: The Author

2.2 Advances in Deep Learning

In 1999 the launch of NVIDIA GPUs had changed the DL industry. The first CNN was already invented in 1980, followed by the ConvNet and LeNet-5, a popular CNN still used nowadays for image classification. Then, developers could improve their models to take advantage of GPUs' capabilities of processing matrix multiplications. Figure 2.5 presents a timeline of the main contributions in DL that precedes the appearance of DDL.

Figure 2.5 – A brief timeline of the contributions created in the field of Deep Learning.



Source: The Author

In 2009, the ImageNet was created, the first big data for computer vision containing 2.3 million colored images of different classes organized in a hierarchical structure using subtrees (DENG et al., 2009). With more data to evaluate, new neural networks architectures started being developed and improved. In 2010, an annual competition to evaluate these networks started called the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It evaluates how accurately new models classify ImageNet. AlexNet, a CNN launched in 2012 with eight layers, was one of the new networks that achieved good results in the contest. It has more convolutional layers than LeNet, and makes better usage of multiple GPUs, resulting in a faster convergence (KRIZHEVSKY; SUTSKEVER; HINTON, 2017). GoogLeNet proposed Inception, a 22 convolution layers and five pooling layers network that presented new features to improve training performance, such as reducing the dimensions of the input pixels computing convolutions faster (SZEGEDY et al., 2015). ResNet, created by the Microsoft Research team, got first place on the ILSVRC 2015 (HE et al., 2016). It uses more layers, 152 total, and it allows lower layers to exchange data to deeper ones, and this way, it avoids losing the gradients values throughout the forward passing. WideResNet, created in 2017, proposes decreasing the depth and increasing the width of a residual network to improve its accuracy, keeping a fifty-layer

network (ZAGORUYKO; KOMODAKIS, 2016). It decreases the depth and increases the width and the number of parameters used during training, presenting better performance than ResNet for ImageNet.

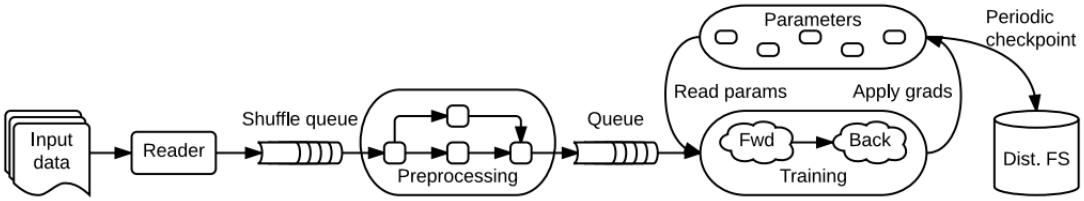
Aligned with the increase in the GPUs usage for processing CNNs, and big data input for the training, more sophisticated models and advances in DL were created. In 2011, one of the first frameworks for DL called DistBelief was developed by the Google Brain team (DEAN et al., 2012). DistBelief brought the concept of using worker servers and parameter servers to update the model parameters and be flexible to run on top of a GPU or CPU. They also started offering parallel computing to train using one or multiple workers in the same cluster. The more popular frameworks today are Caffee, PyTorch, MXNET, Theano, Keras, and TensorFlow.

Keras is a library for DL launched in 2015, written in Python to support CNN and Recurrent Neural Networks (RNN) (KERAS, 2015). It provides a high-level API with a simpler interface for the final user to configure optimizers, and assign operations for accelerators, for example. Keras is used as a wrapper for TensorFlow, abstracting the training configuration to make it easier to use and extend. TensorFlow is the continuous work of DistBelief launched in 2015, and it has become a widely used open-source framework for DL (ABADI et al., 2016). It is implemented in C++ for large-scale and heterogeneous systems, supporting models implemented in C++, Python, Java, and other programming languages. It supports one or multiple devices GPUs, TPUs, and CPUs with x86 and ARM architectures. It also offers a visualization toolkit called TensorBoard to profile and visualize the performance and accuracy of the training. Tensorflow has robust documentation, which helps users to implement their experiments.

Tensorflow uses a dataflow graph to represent the computations, communications, dependencies, and the algorithm state, presented in Figure 2.6. It first reads and performs a preprocessing of the input data, then it builds the model and starts the training. It can collect checkpoints throughout execution, so users can stop and return back to a longer training using the same parameters values. This dataflow scheme allows users to partition the data and distribute it to be independently computed by different workers in parallel. The edges of a TensorFlow graph, also called *tensors*, represent the outputs or inputs of the computations. Tensors are multidimensional matrices used in DL to represent inputs and outputs updated during training inside a NN. In the TensorFlow architecture, tensors will "flow" along with the graph suffering operations along the way. The vertices represent mathematical operators such as matrix multiplications and convolutions. Each vertice can

have zero dimension tensors or N-dimensional tensors as input or output.

Figure 2.6 – TensorFlow dataflow graph representation.



Source: (ABADI et al., 2016)

With bigger datasets, efficient networks, and frameworks to facilitate the training, DDL frameworks started being created. The frameworks' main challenge to scale to multiple nodes is to minimize the overhead with the extra computations, to take advantage of multi-accelerator devices, and to keep a high accuracy. Some of the new frameworks for DDL are: LBANN launched in 2015 by the Lawrence Livermore National Laboratory (ESSEN et al., 2015), Horovod launched by Uber in 2017, and Tarantella launched in 2020 by the Fraunhofer Institute for Industrial Mathematics. Nowadays, DDL research is increasing, so developers can take advantage of modern accelerator devices, and users from different fields can benefit from this tools.

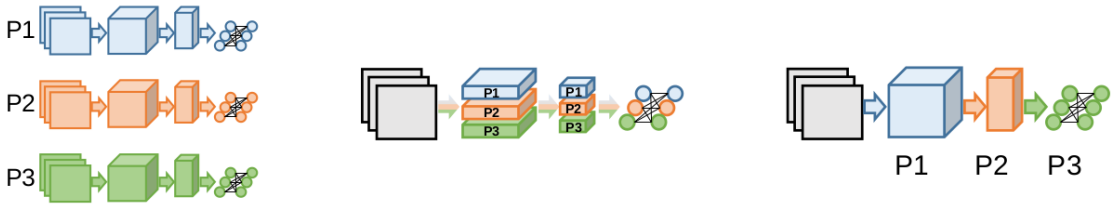
2.3 Distributed Deep Learning

As we presented in the previous Section 2.2, DDL frameworks for distributing the training across computational resources are becoming popular in the last years (BENNUN; HOEFLER, 2019). DNNs present aspects that motivate parallel training, such as the SGD dataset division in minibatches, presented in Section 2.1. Minibatches represent independent portions of the input processed by workers. Workers need to communicate at a given moment to exchange data so the parameters update are based on gradients averaged about all the input. This is a common scenario in parallel computing, were we partition a problem into many tasks and distribute them accross multiple workers, and implement synchronization points if necessary.

In DDL, three parallelization strategies were created to scale a sequential training to multi-GPUs nodes. They consider the main challenges when designing DL frameworks, which are massive communications among processes, the storage of the dataset and the model on each worker depending on the strategy used, and the performance improvements to reduce the processing time and keep or increase the model accuracy. Figure

2.7 presents the strategies: Data Parallelism (a), Model Parallelism (b), and Layer Pipelining (c). P1, P2, P3, and the colors represent the devices, the squares represent the dataset, the cubes represent the network structure, and the connected circles represent the fully-connected layer. The devices can be CPUs, GPUs, FPGAs, or other hardware we are performing the training.

Figure 2.7 – Partitioning strategies for distributed deep learning: data parallelism, model parallelism, and layer pipelining.



(a) Data Parallelism (b) Model Parallelism (c) Layer Pipelining

Source: (BEN-NUN; HOEFLER, 2019)

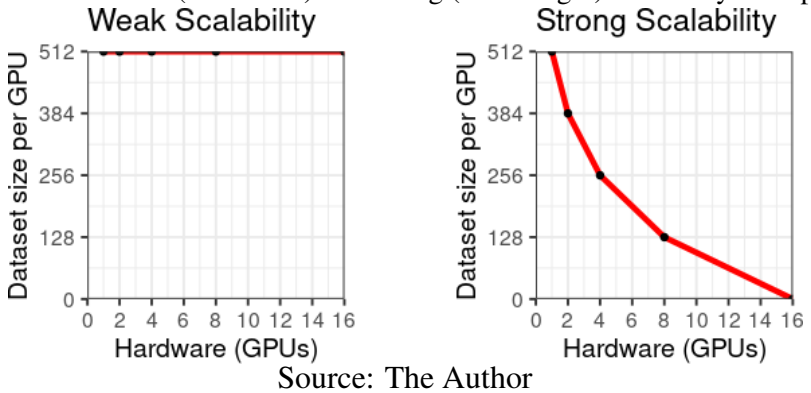
- **Data parallelism** consists of splitting the dataset into chunks and distribute them among the devices, the workers. Each device will process its chunk of data in parallel, needing to store the entire network model in each worker, so the only communication point is at the end of an epoch. At the end of an epoch, the workers average the gradients or parameters calculated in each to update the weights based on the training of the entire dataset. This strategy is useful for overcoming insufficient GPU memory for training with larger datasets, dividing it and distributing it to the devices, or achieving the desired accuracy faster by distributing the same data, but shuffled, for each device. It is also the most straightforward strategy to implement.
- **Model parallelism** consists of partitioning the network structure to be computed by the devices in parallel. Each device processes some of the layers of the entire network, getting a copy of the whole dataset. A challenge in this strategy is to achieve good load balance across devices since some layers present different computations. Also, it brings additional communication and synchronization steps at each forward and backpropagation pass. This strategy is helpful to deal with too large models for a single GPU. It is also harder to implement, usually via multithreading in the same machine or across nodes and devices via message passing to decrease the latency of data exchange.
- **Pipelining**, Pipelining is used to overlap communications with computations between training steps, such as forwarding evaluation, backpropagation, and weight

update, mitigating processor idle times. For example, while a process performs communications, another independent process can implement the forward evaluation phase. This strategy is feasible since a model has a fixed number of communication points, and the source and destination processors are always known. The disadvantages of this strategy are the difficulty of implementing since each layer will require particular implementations and the low latency if not fully utilizing the system.

- **Hybrid**: combines more than one of the strategies above. The more usual is using data parallelism with model parallelism.

For being easier to implement, data parallelism is a common strategy in DDL frameworks. A concern when using data parallelism is distributing the data among workers, if it should use weak or strong scalability regarding the dataset distribution for more devices. Figure 2.8 presents an example of the weak and strong scalability used in DDL training. The weak scalability, represented in the left plot, assumes that we keep the amount of work for each device fixed while increasing the hardware used. If we have a fixed dataset of 512, each extra GPU will process this amount of data. In the strong scalability, represented in the right plot, we increase the number of hardware available for computing and fix locally the amount of work for each extra device added. This approach can lead to performance improvements to train the same problem using more devices. In this case, we increase the number of GPUs and divide the dataset according to the number of devices available.

Figure 2.8 – Weak (on the left) and strong (on the right) scalability examples.



Source: The Author

Choosing weak or strong scaling depends on the training goals. Weak scaling is used to achieve the desired accuracy faster since we will train over more data in more devices. Strong scaling increases the performance regarding execution time, but it can achieve lower throughput and result in underused devices (OR; ZHANG; FREEDMAN,

2020). A work used strong scaling to train a CNN for images classification in Xeon based systems (DAS et al., 2016). They achieved almost a linear scalability adding the double of workers from 2 to 128, using 512 batch sizes. A smaller batch size of 256 also scales almost linearly until 64 devices. Another work evaluated the training accuracy for weak and strong scaling in DDL (CUNHA et al., 2018). It showed that weak scaling fails to converge, resulting in different accuracy without adjusting learning rates and other parameters. Moreover, strong scaling presented good scalability achieving the same accuracy as the sequential training.

DDL strategies are still evolving. New DDL frameworks are being created or improved, so they can accelerate training without impacting the convergence approaches to minimize the error of the predictions. Some aspects of the training should be noted when scaling for more devices, such as the goals of the training, the network model used, and the parallel strategy used by the frameworks. The next chapter depicts the state-of-the-art DDL frameworks, their characteristics, and parallel strategies. We also present the existing performance analysis approaches found in the literature to evaluate these new tools and how they motivated our research.

3 RELATED WORK

Several frameworks are being implemented and evaluated in recent years due to the fast innovations in the field. The following sections present an overview of some available DDL frameworks and toolkits, other works that proposed a performance analysis comparison of distributed frameworks, and the most-used tools for evaluating distributed training performance. Moreover, we justify the choice of the HPC tools for our evaluation and how the current efforts and methodologies are related our new approach.

3.1 Distributed Deep Learning Frameworks

During the last years, several DDL frameworks and toolkits were created for accelerating and facilitating the training of large models and huge datasets. They can be targeted to specific scenarios, such as dealing with big data systems using data parallelism or to train over big datasets (DAI et al., 2019; OOI et al., 2015), targeted to specific devices, or to be more or less user-friendly. This section presents recent DDL frameworks focusing on the frameworks aimed at users without deep knowledge of distributed systems and works that evaluate and compare these frameworks.

3.1.1 Frameworks for Distributed Training

Frameworks to accelerate deep learning using multiple nodes in large-scale HPC systems are still being created or actively improved. The first distributed training options offered new features to users with knowledge of ML, who also needed to learn new concepts on how to distribute layers and phases of the training to workers on multiple nodes. In recent years, new frameworks have been created to prevent users from acquiring a deep knowledge of high-performance systems to improve their training performance. These frameworks abstract the system configurations to quickly transform a single-node training into a multi-node training. Table 3.1 summarizes the properties of the frameworks we will present that run on top of popular ML libraries providing a user-friendly API.

Horovod was developed by the Uber Engineering team for easy-to-use data parallelism distribution in Python on top of TensorFlow, Keras or PyTorch (SERGEEV; BALSO, 2018). In the backend, Horovod uses the Message-Passing-Interface (MPI) and

pthreads for parallel computing. LBANN (Livermore Big Artificial Neural Network) is a toolkit for accelerating the training of large neural networks on high-performance machines using data, model, and pipelining parallelism (ESSEN et al., 2015). It uses open-source libraries most developed by their team: DiHydrogen and Hydrogen, for distributing linear algebra operations; Aluminum, for HPC communication using MPI and NCCL; Elemental, for distributing matrix-matrix and matrix-vector computations and the BLAS library for node-local thread-level parallelism. Since it is focused on the internal usage at the Lawrence Livermore Laboratory (LLNL), the instructions on how to install and use, and documentation of LBANN and dependent software are focused on the laboratories' facilities. Tarantella is a new tool developed by the Competence Center High-Performance Computing that applies data parallelism using the Global Address Space Programming Interface (GASPI) standard for distributed computing (CCHPC, 2020). It is open-source and provides a solid documentation and contact to the developers.

Whale is a recent framework aimed to train giant models using data, model, and pipelining parallelism strategies. It supports TensorFlow and NVIDIA GPUs, but it is not available online for usage (JIA et al., 2021). Orca is part of the BigDL 2.0 project, and it provide a easy-to-use library to scale single node training in Python to more nodes (SONG et al., 2020; DAI et al., 2019). Since it is focused on distributing big data, Orca is configured to run on Apache Spark, an analytics engine for large scale data processing, and Ray, more focused on machine learning applications.

Table 3.1 – Distributed Deep Learning Frameworks Overview.

Framework	Implementation	ML Framework Support	Parallel Strategy	Open-source
Horovod	Python and C++	TensorFlow, Keras, PyTorch, MXNet	Data	✓
LBANN	C++	PyTorch	Data, Model, Pipelining	✓
Orca	Python	TensorFlow, PyTorch, Keras	Data	✓
Tarantella	Python and C++	TensorFlow	Data	✓
Whale	Python	TensorFlow	Data and Model	

Source: The Author

Some ML frameworks and DDL toolkits also support distributed training under more user configuration and knowledge of distributed systems and parallelism paradigms. TensorFlow offers two different distributed learning approaches. The Parameter Server Strategy was the first one created based on the technique of having one dedicated server to receive data from workers and orchestrate the communication to the other workers to update parameters (DEAN et al., 2012). And the recent strategy released as a stable API in the TensorFlow 2.4.0 version in December of 2020 is called Multi Worker Mirrored Strategy. As Horovod, this strategy is based on Baidu's all-reduce algorithm that uses

no dedicated server to exchange data (GIBIANSKY, 2017). It is an open-source code in Python and C++ with support for data, model parallelism, and asynchronous and synchronous communication. However, it is less intuitive to apply since the user needs to configure the workers and parameter server and learn new concepts. SINGA was created by Apache for training big models such as convolutional and recurrent neural networks, over large datasets, written in C++ and Python (OOI et al., 2015). It is an open-source tool, implements data parallelism and uses the NCCL library over its only API model, and not run over others existing ML frameworks.

PyTorch was developed by Facebook in C++ and Python and programmable in Python (PASZKE et al., 2019). It supports model and data parallelism, and can be used with Apex, a tool to enabled mixed precision and optimize the use of NVIDIA GPUs using NCCL (NVIDIA, 2020a). Microsoft Cognitive Toolkit (CNTK) was developed by Microsoft in Python and C++ with data parallelism, is one of the first toolkits for distributed training, but is no longer actively developed (SEIDE; AGARWAL, 2016). There is also ChainerMN, an extension of the Chainer DL framework for distributed deep learning (AKIBA; FUKUDA; SUZUKI, 2017), and MXNet framework, developed by Apache that implements data and model parallelism (Chen et al., 2015).

3.1.2 Frameworks Evaluation and Comparison

Recent works that present a complete literature reviews and comparisons of DDL frameworks consider aspects such as parallelization strategy used, programming language supported, the use of communication overlapped with computation, if it is easy to use, and portability to other architectures (RAVIKUMAR; S., 2020; HASHEMINEZHAD et al., 2020). Overall, works that perform practical comparisons focus on the training scalability and efficiency.

SHI et al. (SHI et al., 2018) evaluate the performance of Caffe-MPI, CNTK, MXNet, and Tensorflow in a four-node cluster with 4 NVIDIA Tesla P40 per node. Their methodology consists of selecting the SGD algorithm models and running tests for analyzing their performance, then implementing optimizations in the SGD model, performing more tests, and finally analyzing the previous results in different environments. Although this work explores the tools' specificities, considering their parallelization algorithms, parameter configurations, and scalability, they uncover the evaluation of the new DDL frameworks and focus on the running time performance evaluation only. Another work

evaluated four frameworks across HPC resources and compared their efficiency, walltime, and speedup using CPUs and GPUs (4 Tesla V100 available per node), with MPI, for up to 12 GPUs (MAHON et al., 2020). They evaluated Horovod with TensorFlow, Keras, Horovod with Keras, Pytorch with a Gloo backend, and MXNET. PyTorch presented the best results for all setups, followed by Horovod with TensorFlow. As the authors reported, once you have a serial code for DL, using Horovod requires very little coding compared to the other frameworks used, so it is highly recommended for users without deep knowledge of DL. They did not evaluate the distribution for multiple nodes for Keras and MXNet since it was less user-friendly to configure.

Another work explored less common tools and architectures to evaluate the weak and strong scaling of distributed frameworks. They used Cray systems and compared Horovod, Horovod-MLSL, which uses the Intel Machine Learning Scaling Library instead of MPI, and the Cray Programming Environment plugin for ML (KURTH et al., 2019). They evaluated the execution time and accuracy of the training and concluded recommended Horovod with MPI because it brought the best weak scaling and because it is available for most systems. Similarly, another work compared Caffe2, ChainerMN, CNTK, MXNet, and TensorFlow using cloud computing with Amazon Web Service (LIU et al., 2018). They used a ResNet-50 model on the CIFAR-10 dataset with synchronous and asynchronous SGD updates, running instances with no dedicated host. For the multi-node evaluation, they only use one GPU per node and compared training speed, reporting that the Tensorflow version requires more code changes to configure the multi-GPUs environment due to the parameter-server setup. MXNet and ChainerML show the best scalability among the frameworks.

No recent works have compared the modern DDL frameworks performance using up-to-date HPC tools and methodologies for distributed applications. None of them correlate profiling and tracing to compare the performance of the frameworks for the same DL model using temporal performance analysis, crossing results among different profiling and tracing tools. The same happens for works that evaluate a specific DDL framework. They apply one or two tools for performance analysis, such as Horovod Timeline, Keras callbacks, NVProf, NVSMI and NVProf, Horovod Timeline and cProfile, to profile the application focusing on overall measuring about execution time and efficiency (LAANAIT et al., 2019; CUNHA et al., 2018; SHI; CHU, 2017; WU et al., 2021; WU et al., 2018). These tools have advantages but also some limitations, as we will present in the following section.

3.2 Performance Analysis of Distributed Frameworks

Performance analysis of DL models usually runs a single experiment, which can take days to finish, and evaluate their accuracy, efficiency, and execution time using callbacks or profiler tools. The new advances of the field started exploring high-performance systems to partition a training phase on multi-GPUs, and more recently, to multiple computational nodes. Even with this advance in the DL field, the performance analysis methods and tools used are still similar to what has been used for training in one device or multiple devices in a single node. This section presents the most-used performance analysis tools nowadays for DL and DDL frameworks, their advantages, and limitations.

3.2.1 TensorBoard

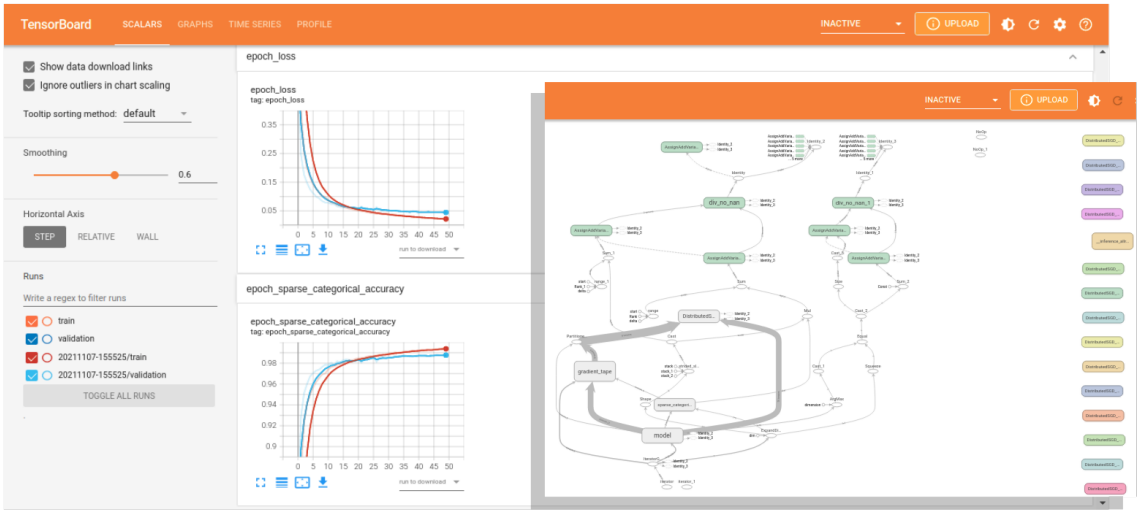
TensorBoard is the visualization toolkit for TensorFlow to profiling and tracing machine learning experimentation (BRAIN, 2015). It can be downloaded as a standard Python package with the TensorFlow API, and the user needs to add the TensorBoard into the callbacks used by the fitting function. It generates two output directories containing measures about `training` and `validation`, and it uses the web browser to present the visualizations. Tensorflow has robust documentation, which helps users to implement their experiments.

Figure 3.1 presents two TensorBoard tabs, the one in the background is called Scalars and presents interactive plots for loss and accuracy along epochs or execution time, and the other in the front is the Graphs tab, which shows an interactive graph of the network model used. There are also other tabs to explore, such as Time Series that shows the metrics in Scalars per rank, more detailed, and Histograms that shows the weights, biases, or other tensors as they change over time. This information can be used to detect overfitting using the plots for training and validation loss or to visualize the dataset images, text or audio (VOGELSANG; ERICKSON, 2020).

For getting GPU level information, TensorBoard uses CUPTI [1], a CUDA Profiling Tool Interface to get traces and profile of experiment at GPU level. The tool started supporting multi GPUs profiling and tracing in the end of 2020, but only for sampling mode, where we perform on-demand profiling by setting the workers IP addresses and

[1] https://docs.nvidia.com/cuda/cupti/index.html

Figure 3.1 – TensorBoard user interface showing the Scalars (background image) and the Graphs (front image) tabs.



Source: The Author

the time interval to be profiled [2]. To get the Profiling information we need to install the TensorBoard plugin and configure it in the code. It can profile the training for a range of batches specified by the user, and generates an event trace for each node that launched an instance of TensorFlow and it detects the GPUs set visible for TensorFlow.

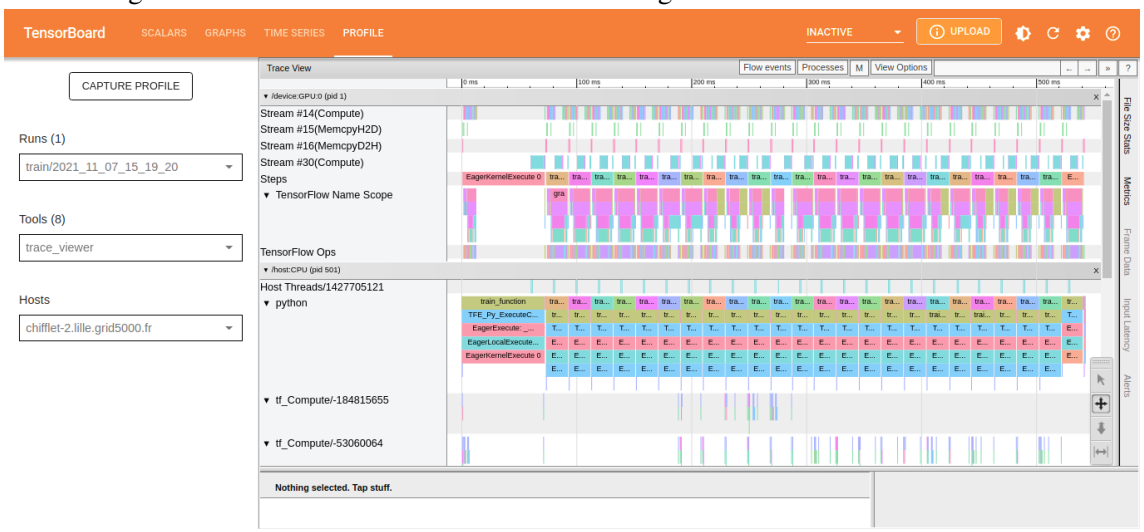
Figure 3.2 presents the Profile tab, which depicts the devices' usage and kernels execution. It shows the computing time at the host and at the device, the compilation time, communication time, the computing time per kernel, a memory profiler for the devices, an experimental tool to show a performance analysis summary that diagnostic the training bottlenecks, and trace visualization. The trace visualization shows the events per stream in the GPU and CPU. It is hard to detect overlapped operations, and we can only open one panel per node. The training needs to be distributed using the TensorFlow APIs for multi nodes systems to detect and show them in the same panel about a node.

3.2.2 NVIDIA Profiler and GPUs monitoring

NVIDIA System Management Interface (nvidia-smi or NVSMI), is a command-line tool to collect information from NVIDIA GPUs (NVIDIA, 2011). It is used to evaluate the devices power, frequency, memory utilization, and GPUs utilization. It is a command-line tool to monitor defined queries about the devices in a defined time interval, and it can output the monitoring results in the CSV file format.

[2] https://github.com/tensorflow/tensorflow/releases/tag/v2.4.0-rc4

Figure 3.2 – TensorBoard user interface showing the Profile tab for trace viewer.



Source: The Author

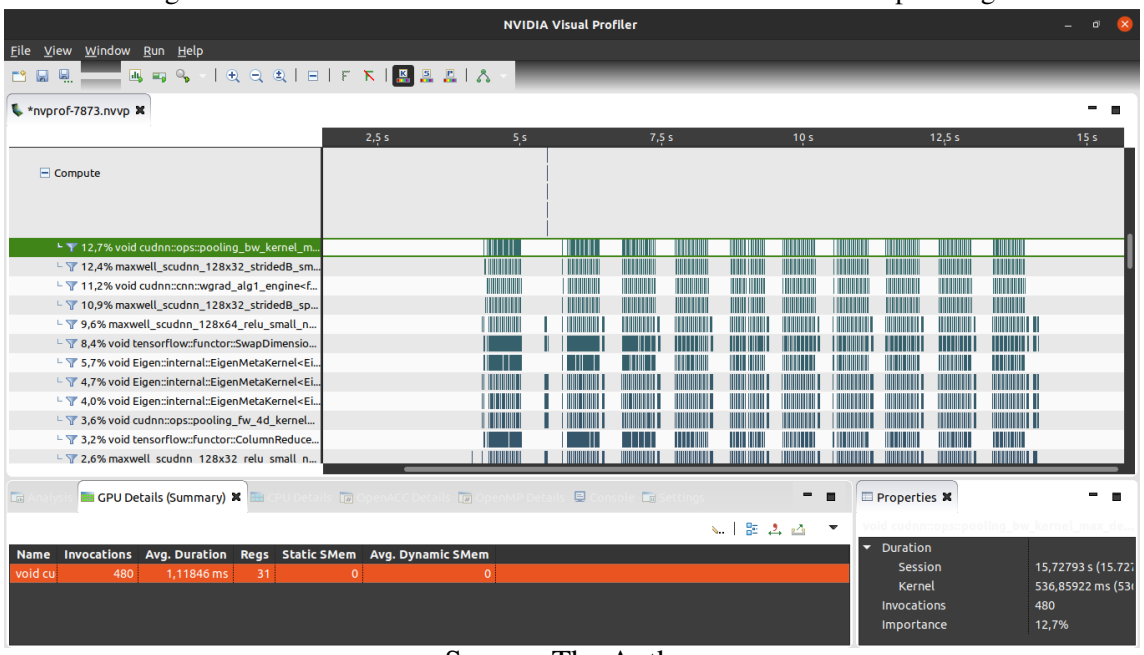
NVIDIA Visual Profiler (NVProf) is part of the CUDA toolkit as a profiling and tracing tool intended to optimize the performance of CUDA applications. NVProf collects all events running in NVIDIA GPUs, enabling the user to define profiling regions to avoid profiling all experiments. NVProf output is stored as a SQLite relational database and can be open using the NVProf interface, that needs to be locally installed as presented in Figure 3.3. As TensorBoard, it also uses CUPTI to collect the profiling information, so it shows similar results regarding the GPUs usage. The opened tab display the profiling information for one worker in a distributed training with Tarantella. The main panel shows the event calls per computing operation, streams, and thread. It also shows the data copy between host and devices and can generate traces in the CSV format. NVProf was discontinued in 2019 to the creation of the NVIDIA Nsight Tool[3], a new tool that offers the same visualizations and profiling workflow, but with new features, such as support for larger profiles (>2GB) without slowing down the visualization as NVProf, and support to the most recent visualization tool with support to modern NVIDIA GPUs.

3.2.3 Callbacks in Python

Profiling at the Python level can gather training information about execution time, parameters, and the training status at runtime. Python cProfile is a tool used at command line to report statistics about the time spent with Python functions and methods (FOUNDATION, 2016). For DDL evaluations, it only shows the total time of the ML framework,

[3]https://developer.nvidia.com/nsight-systems

Figure 3.3 – NVIDIA Visual Profiler user interface for one node profiling.



Source: The Author

without details of how it behaves during the training (WU et al., 2018)

Keras callbacks are a feature from the Keras library for DL written in Python (KERAS, 2015). It provides a set of functions to get information about training during runtime. Users can control the early stopping feature, checkpoints, or use custom functions to get weights, accuracy, loss, and execution time, called at specific times during execution per epoch, iteration, or for the whole training. It is a great tool to investigate the efficiency of the training and how fast it converges. TensorFlow offers an abstraction of the Keras callbacks more easy-to-use to pass to the Keras methods for fitting, evaluation, and prediction for training, testing, and validation phases.

3.3 Discussion about DDL Frameworks and Performance Analysis Tools

Considering the DDL frameworks to abstract distributed training over ML libraries, LBANN's advantage is the implementation of all parallel strategies, but it is highly dependent on the frameworks targeted to LLNL systems, with a lack of details on how to configure to other clusters. LBANN Spack package presented several inconsistencies in the software versions for the clusters we tested, which were challenging to configure and use. Whale is not open-source, so it blocked us from using and exploring its code. Horovod and Tarantella are both implemented using the same programming language and offer the same parallel strategy. They are both open-source and have support

for TensorFlow and CUDA.

Keras callbacks and TensorBoard offer important insights on specific aspects of the training. NVProf is targeted to the system, being a popular profiler in HPC for GPU-based applications. Keras callbacks are useful to evaluate the training efficiency over a specific time interval. Still, there is a lack of details on how the model uses the devices for training or communicating inside the time frame representing an epoch duration. NVProf opens one file for each worker at a time, preventing us from contrasting the worker's execution. The recent support by TensorBoard for multiple nodes does not provide a visualization of all workers involved in the training in the same panel. Testing for Horovod, Tensorboard generated one trace file per node, but it only shows one CPU and one GPU in each node, even using more. For Tarantella, TensorBoard only generates traces for the node from where we launch the script.

Recent works to evaluate and compare the performance of DDL frameworks focus on model accuracy, scalability, execution time, and resource usage. This information is usually analyzed individually, with different tools, visualization plots, and panels. Also, the results are shown as an overall representation of the performance per epoch or per the number of workers used. From the discussed works and tools, none correlated information gathered from different tools to analyze the training in a temporal evaluation, considering HPC methodology approaches. No methodology or tool present in the same panel a temporal aggregation of the devices computing time and correlate it to callbacks at the Python level to understand what those values encompass. As we presented, new tools and methodologies are required to evaluate the usage of high-performance devices by DDL frameworks.

4 MATERIAL AND METHODOLOGY

This chapter presents the tools and methodology to analyze and compare the performance of Horovod and Tarantella, presented in Section 4.1, to distribute the training in different NVIDIA GPU models. Our evaluation methodology used a convolutional neural networks model with SGD optimizer, varying the batch size training parameter, the GPUs models, and scaling for more GPUs in the clusters. The experiments presented in this work were carried in three clusters from the Grid'5000 (Cappello et al., 2005) testbed and one in the PCAD[1].

Table 4.1 presents the selected clusters. Hype is hosted in the PCAD and Chifflot, Chifflet, and Gemini at the Grid'5000. The Tesla V100 GPUs are the only ones with Tensor Cores available, a technology from NVIDIA that performs more matrix operations per GPU clock. While a CUDA Core computes one single value in a 1×1 multiplication per clock, a Tensor Core computes a 4×4 matrix multiplication and can add a third matrix of 16 or 32 floating points per clock. DNN training benefit from this multiplication and sum of matrices for its linear algebra operations in the layers. If available, the NVIDIA CUDA Deep Neural Network library (cuDNN) is configured to use Tensor Cores, increasing the training throughput while keeping the accuracy. NVLink is a feature added in the board architecture of Tesla P100 and Tesla V100 GPUs to provide higher bandwidth between devices than using PCIe. It is available in Gemini, with a links speed of 25.781 GB/s.

#+BEGIN_EXPORT latex

Table 4.1 – Specifications for the clusters used, all with Intel CPUs and NVIDIA GPUs.

Specification	Hype	Chifflot	Chifflet	Gemini
Location	PCADUFRGS	Grid'5000	Grid'5000	Grid'5000
CPUs (per node)	2x Xeon E5-2650 v3	2x Xeon Gold 6126	2x Xeon E5-2680 v4	4x E5-2698 v4
Nodes	2	6	8	2
Cores	10	12	14	20
Memory	128GB	192GB	768GB	512GB
Frequency	2.3GHz	2.6GHz	2.4GHz	2.2GHz
GPUs (per node)	2x Tesla K80	2x Tesla P100	2x GTX 1080Ti	8xTesla V100
CUDA cores	2,496	3,840	3,584	5,120+640 Tensor
Frequency	560MHz	1190 MHz	1481MHz	1230 MHz
Memory	12GB GDDR5	12GB GDDR5	11GB GDDR5X	16 GB HBM2
Network	Ethernet	2x 25 Gbps	2x 10Gbps	10 Gbps+3x 100Gbps InfiniBand

Source: The Author

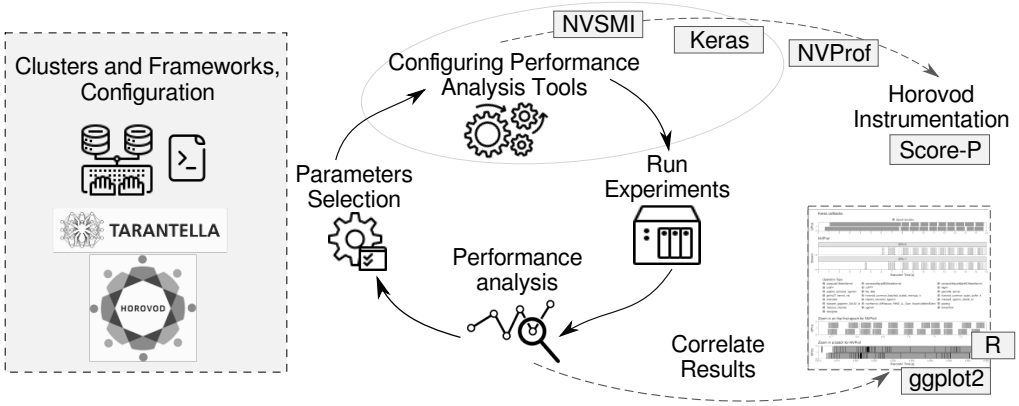
The clusters have operating system Debian 4.19.160-2, GCC version 8.3.0, Open-MPI 4.5.0, GPI-2 1.5.0, which is an open source implementation of the GASPI standard, CUDA 11.3.1, and Python 3.7.3. We use Anaconda virtual environments to manage the

[1] http://gppd-hpc.inf.ufrgs.br/

python packages, and Spack to manage software required in the different micro architectures (GAMBLIN et al., 2015; SOFTWARE, 2016). We use Horovod version 0.22.1 and Tarantella version 0.7.0. For data processing and analysis, we use the R programming language with the package tidyverse[2] for the data science processing and the package ggplot2 to generate visualizations.

Figure 4.1 presents an overview of our methodology workflow. We first selected Tarantella and Horovod and configured them in the clusters used. We applied an incremental methodology, performing constant experiments and post-execution evaluations and analysis using visualizations with R and ggplot2. We selected parameters for varying the batch size, the input size, and the number of workers in training. After finishing the experiments for a complete first methodology cycle collecting measurements with NVSMI and Keras for overall performance analysis, we started applying Keras and NVProf to collect execution tracing and profiling. We also applied the Score-P trace system for Horovod experiments. Performance analysis using data science and visualization was carried after running the experiments.

Figure 4.1 – Our methodology workflow. The black arrows represent the main workflow cycle, constantly reproduced. The dashed arrows represent new steps added incrementally. First, by performing experiments with NVSMI and Keras, with NVProf, and with Score-P for Horovod. The other incremental step regards our performance analysis using data science and visualization.



Source: The Author

The performance analysis experiments generate three output files: a CSV file for monitoring the devices' usage, a CSV file containing callbacks related to the model training stages, and a log file with the performance results for the training obtained with Keras function at the Python level. We monitored the GPUs usage using the NVSMI. We collected the following set of parameters every second: GPU index, timestamp, the power draw for the board, the frequency of the Steaming Multiprocessors, the frequency of mem-

[2]https://cran.r-project.org/web/packages/tidyverse/index.html

ory and graphics clocks, the PCIe bus ID, memory used, memory available, GPU utilization, memory, utilization, and temperature. The training stages were obtained with Keras callbacks functions, which capture the training status in specific moments, such as at the initialization and end of an epoch or a batch.

These tools help us to understand the overall performance of the training and measure the devices' usage. However, none of them provides enough information to understand which events are computed in the devices during training. For example, with Keras callbacks, we have the duration of an epoch, but there are no details of what is happening inside this time interval. NVProf and Nsight captures traces for NVIDIA GPUs. Since our approach did not generate much data, we used our previous experience with NVProf to our analysis. With this, our approach was to correlate the results from the Keras callbacks and NVSMI with the results with NVProf, so we have a temporal vision of what is happening inside the frameworks' "black-box". Moreover, we used the Score-P library, in its version 6.0, to record traces focusing on the MPI communication for Horovod (KNÜPFER et al., 2012).

4.1 Selected Frameworks

Tarantella and Horovod are open-source frameworks that implement data parallelism and support TensorFlow and Keras training on GPUs and CPUs using CUDA-based implementation. They are both written in C++ and Python programming languages but use different programming paradigms for distributed computing. They also have solid documentation and developers actively available to help. This section presents the DDL frameworks selected for this work: Horovod (Section 4.1.1) and Tarantella (Section 4.1.2). We will present the framework's characteristics, usage, parallelization strategy, and algorithms used, and a brief overview of the most important aspects of each framework selected for this work in Section 4.1.3.

4.1.1 Horovod

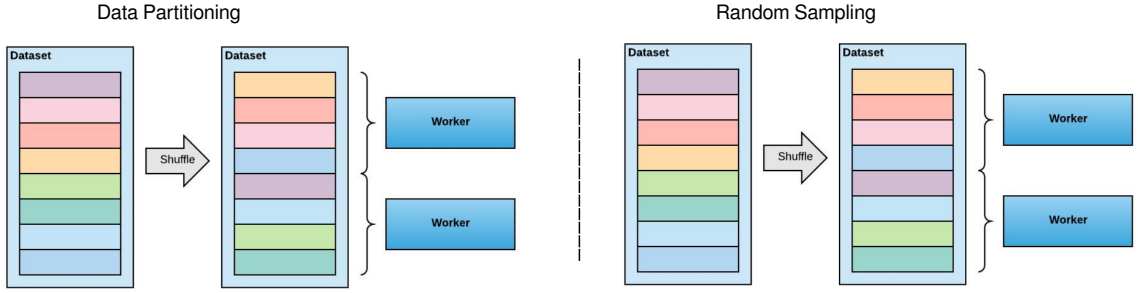
Horovod is an open-source library written in C++ and in Python, with support for data parallelism, developed by Uber in 2017 (SERGEEV; BALSO, 2018). It has support to run on top of TensorFlow, Keras, PyTorch, or Apache MXNET. It can be downloaded

as a stand-alone python package or installed and compiled via source code, available in the official repository: <https://github.com/horovod/horovod>. Horovod's documentation presents strong support on how to install and use the framework for all libraries and software required, but little information on concepts and about the algorithms implemented: <https://horovod.readthedocs.io/en/stable>. It has been constantly updated, and it has been compatible with all TensorFlow versions since it was launched.

The framework uses MPI to find the machines and coordinate operations from hosts to devices. If available, the reduction operations and communication across devices use the NVIDIA Collective Communication Library (NCCL) to perform collective communication between NVIDIA GPUs and multi-nodes over PCIe and NVLink interconnections. Otherwise, it requires communications between the host and devices. It works with proprietary and open-source MPI implementations, and for GPU programming, it uses CUDA.

Horovod launches multiple copies of the training script, one for each worker. Horovod developers recommend the data distribution strategies presented in Figure 4.2. In data partitioning, users must implement the data partitioning strategy by passing the desired `dataset_size` considering the number of workers. In random sampling, the workers randomly read data from the total dataset, the same amount of data shuffled differently. The script recommendation to use Horovod implements random sampling. Therefore, to achieve any acceleration with more workers, we need to split data across workers by manually implementing the data distribution strategy.

Figure 4.2 – Recommended strategies for data distribution in Horovod.

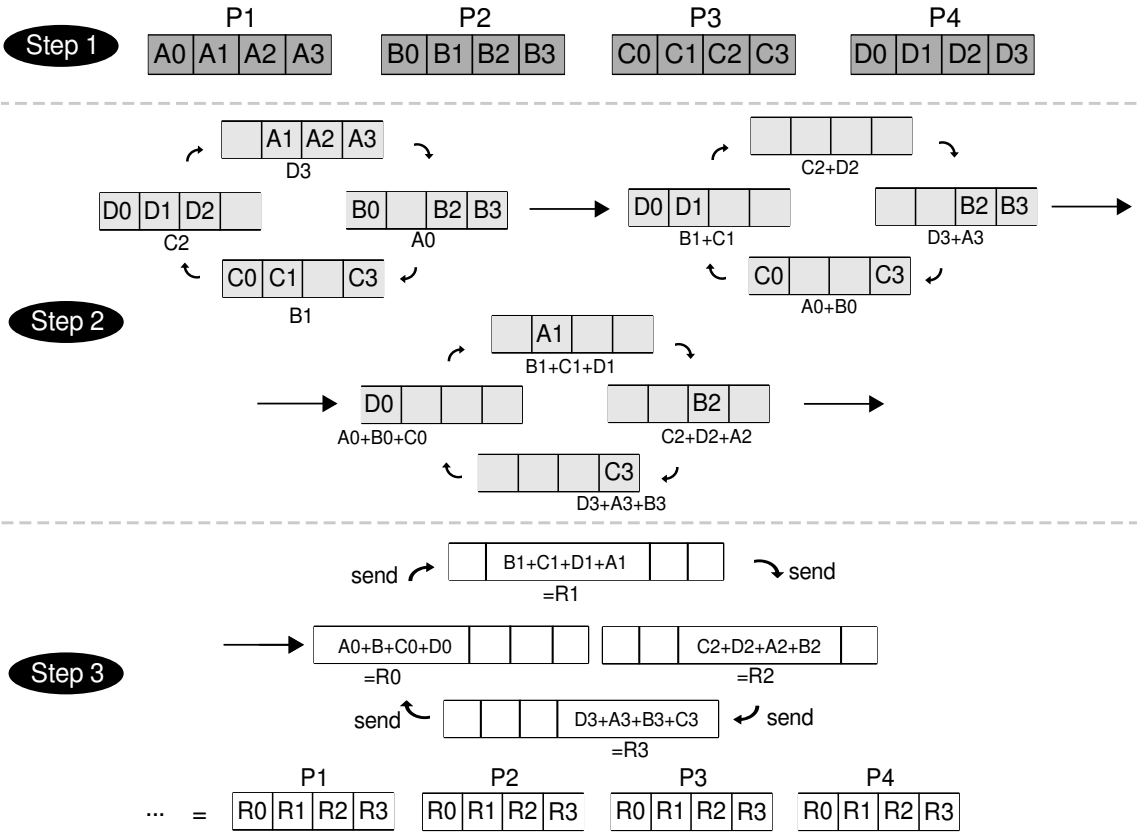


Source: (SERGEEV, 2017)

Horovod implements a ring-allreduce algorithm where the gradients computed on different devices are averaged by the workers and distributed to all nodes for weights update during backpropagation. This approach improves performance compared to the TensorFlow strategy, which allocates one process to be the parameter server, responsible for averaging gradients calculated in different processes, called workers and broadcasting

them to all (SERGEEV; BALSO, 2018). The Horovod reduction algorithm was inspired in an article published by Baidu (GIBIANSKY, 2017). Figure 4.3 presents the algorithm idea, divided into three steps. In the first step, the user's manual dataset partition guarantees each process (P1, P2, P3, and P4) will work over part of the data. The data shuffling implemented with TensorFlow ensure the processes will deal with different data. Each device will compute the gradients (A, B, C, and D) for their chunks during the epochs. The all-reduce algorithm starts in the second step, where the processes share their gradients with their neighbor using a ring communication and reduce the received data with the gradients for that chunk. Each process will have reduced one chunk gradients at the end of this step. In the third step, each process shares its reduced gradients chunk with the others.

Figure 4.3 – The Horovod ring-allreduce algorithm, for four processes (P1, P2, P3, and P4), each working over part of the dataset, computing gradients (A, B, C and D).



Source: The Author

The total communication cost of the ring all-reduce algorithm is $2 \times ((\frac{N}{P}) \times (P - 1))$, where N is the number of chunks and P is the number of workers. Considering $\frac{N}{P} \times (P-1)$ for the reduction phase, and $\frac{N}{P} \times (P-1)$ for exchange the reduced gradients. This algorithm uses the NCCL library, which supports point-to-point primitives execution multi-GPU and multi-node, such `send`, `receive`, `scatter`, `gather`, and `all to`

`all`, and optimize operations such as merge, reduction, and aggregation.

In practice, the ring-allreduce algorithm can generate several all reduce operations for application with many tensors, increasing the time spent with communication (SERGEEV; BALSO, 2018). The first versions of Horovod implemented a greed fusion algorithm to do small all-reduce operations when detecting tensors that are ready to be reduced, so gradient transfer can be done right after the calculation is finished (HOROVOD, 2019). Horovod then proposed the Tensor Fusion technique to fuse small all-reduce tensors operations into larger ones, and this way, interleave communication with computation and gain performance. As presented in the Horovod documentation, the Tensor Fusion works by determining the tensors available to be reduced, allocating a fusion buffer to store the tensors, running the all reduce operation over the buffer, copying the results to the output tensors, and repeating it until detecting more tensors to reduce that fit in the buffer. This results in fewer reduction operations at a time. The Tensor Fusion buffer size can be adjusted by the user.

Distributing a training script with Horovod requires a few lines of code, as presented in Listing 4.1 (more information in the comments of the code snippet). We need to import the library to the script, initialize Horovod, and pin each GPU available to a process. We scale the learning rate linearly with the number of workers to improve the model convergence rate compared to a single device training, as presented in (GOYAL et al., 2018). Then we wrap the optimizer, in this case, the Stochastic Gradient Descent, to use the Horovod optimizer, which will apply the ring-allreduce algorithm. A wrapper function is a subroutine in a software library to call a second subroutine with additional computation. To guarantee the parameters and weights are correctly initialized, we use a Horovod callback function to broadcast global variables from rank 0. The Horovod execution can be done by calling a python script `horovodrun` or directly using the `mpirun` command. We pass the number of workers and the host file as parameters as in usual MPI programs.

Listing 4.1 – Example of using Horovod in a training script with TensorFlow and Keras.

```
...
# Import Horovod library
import horovod.tensorflow as hvd

#Initialize Horovod
hvd.init()
```

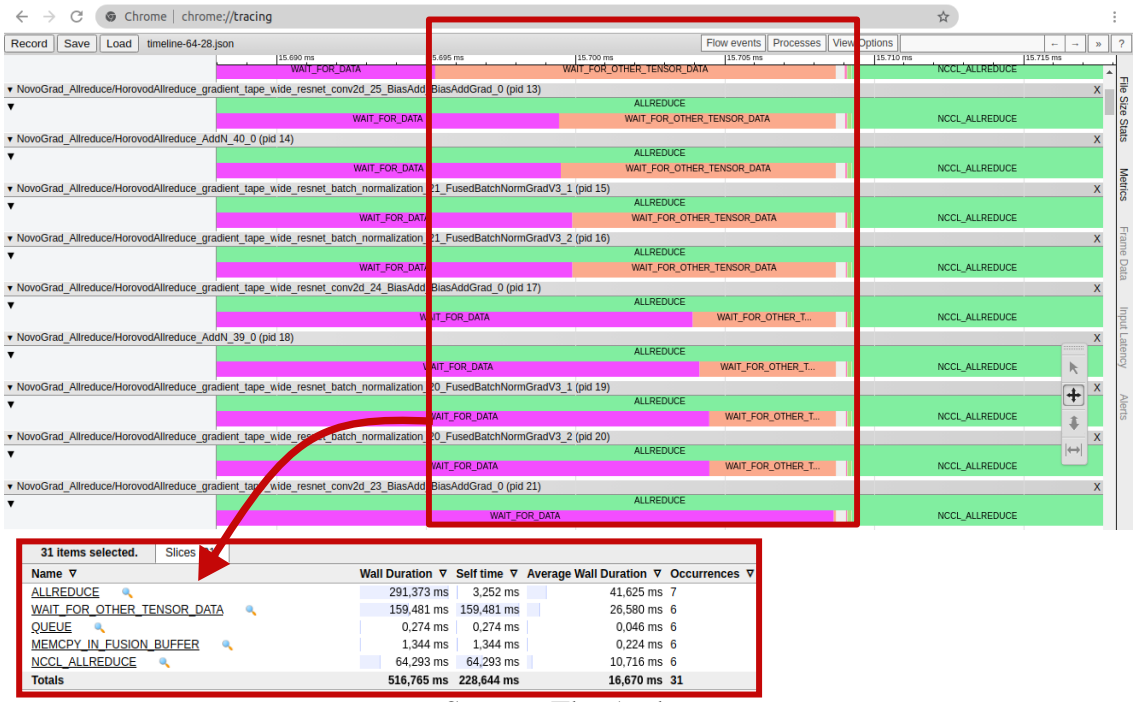
```
# Check GPUs available and pin one per process
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    tf.config.experimental.set_visible_devices(
        gpus[hvd.local_rank()], 'GPU')
...
# Create the Keras model
...
# Add the Horovod Distributed Optimizer to distribute the
# gradients and set the gradients averaging allgather
# the model update
opt = keras.optimizers.SGD(
        learning_rate=args.learning_rate * hvd.size())
opt = hvd.DistributedOptimizer(opt)

# Compile the model using the Horovod optimizer
model.compile(optimizer = opt, ... )

# Broadcast initial variable states from rank 0 to all
# other processes
callbacks = [
        hvd.callbacks.BroadcastGlobalVariablesCallback(0),]
...
```

Horovod also offers a native profiler tool called Horovod Timeline. When passing the parameter to collect the timeline data, the execution generates a .JSON that is compatible with the Google Chrome `chrome://tracing` viewer, as seeing in Figure 4.4. It offers an interactive view where the user can pan in the visualization, zoom parts of the timeline, use a checkbox to select the operations displayed, and select specific regions to get detailed textual information. In the Figure, the region selected in the red annotation opens a new panel with the total execution time and the number of calls and execution time for each operation inside the region. Log visualization is helpful to evaluate the training efficiency and the training time. It facilitates the identification of epochs, MPI and NCCL operations, and bugs related to Horovod. However, it hides information about the C++ operations to launch the ring-allreduce algorithm and the device's utilization. It also presents limitations to open and navigate in large log files in the browser. A training in four workers in 100 epochs and 100 batch size generated a trace with more than 300MB, for example.

Figure 4.4 – The Horovod Timeline view.



Source: The Author

4.1.2 Tarantella

Tarantella is an open-source framework developed at the Competence Center for High-Performance Computing, part of the Fraunhofer Institute for Industrial Mathematics (CCHPC, 2020). It is written in C++ and Python with support to TensorFlow models in clusters with CPU and GPU, or CPU only. The major goal of the tool is to provide strong scaling efficiency in an easy-to-use way, even for people without parallel programming knowledge. Using Tarantella, users can distribute the training by adding a few code lines to their model without configuring parallel computing details.

The framework implements the data parallelism strategy, using synchronous communication to keep results as accurate as training in a single device. It uses the GPI-2[3] communication library, based in the Global Address Space Programming Interface (GASPI) API and high-performance communication between devices, with support for CPUs, GPUs and FPGAs (GRÜNEWALD; SIMMENDINGER, 2013). GASPI implements one-sided asynchronous communication, allowing Tarantella to overlap the communication during all-reduce with computation in the `backpropagation`. As MPI, GASPI uses a C++ interface to implement collective operations.

Tarantella's approach for data parallelism implements a class that subclasses the

[3]https://github.com/cc-hpc-itwm/GPI-2

Keras `model.fit` function, used to train the model for a fixed number of epochs. This way, Tarantella implements its data partitioning while benefiting from the Keras callbacks and batches distribution. The Keras fit function receives the dataset as raw data or in the `tf.data.Dataset` format. However, Tarantella fit allows datasets only in the `tf.data.Dataset` format, representing a large set of tensor elements that passed through transformations (shuffling, batching) before applying the model. The dataset size must be a multiple of the batch size for correct data partitioning.

Tarantella data parallelism is implemented automatically, or by explicitly passing `tnt_distribute_dataset = True` to its `model.fit` function. The dataset size is always fixed, and it is split among the participating ranks. That means that the more ranks are available, the more images you can process concurrently, and thus the faster your training (for one single epoch) should be. Tarantella uses the `shard` function to divide the data. It creates a dataset of size `1/num_shards` associated with the index parameter, called as shown in Listings 4.2.

Listing 4.2 – Dataset partitioning operation in Tarantella.

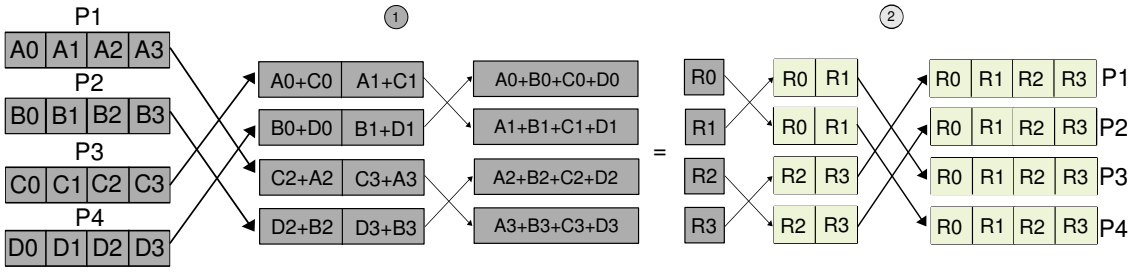
```
dataset = dataset.shard(
        num_shards = self.num_ranks,
        index = self.rank)
```

Tarantella uses a strategy called `micro-batches`, to automatically scales the minibatch sizes with the number of devices used. It splits each minibatch into a number of called `micro-batches` of size equal to the minibatch divided by the total number of ranks, and distribute the `micro-batches` to be run in parallel. To apply strong scaling, where we increase the number of workers and keep the input constant, users need to increase the minibatch size linearly with the number of ranks. For example, if for a single rank you use a minibatch size of 128, then for `N` ranks, you should use $128 \times N$. Also, the minibatch size has to be a multiple of the number of ranks to run because of the `micro-batch` default implementation.

Tarantella averages the gradients calculated in different workers at the end of an epoch using reduce-scatter and allgather algorithms. Figure 4.5 presents the algorithms used by Tarantella to perform the gradients reduction. It is divided into two all-to-one, and one one-to-all collective algorithms pointed in the Figure: (1) Butterfly Reduce-Scatter, which is a combination of the reduce and the scatter communication models, and (2) the Butterfly Allgather, which is a combination of a gather and a broadcast. In the first phase, the processes share chunks of their data with others and reduce the values obtained,

resulting in a total communication cost of $\log P \times \alpha + s \times \beta$, where `P` is the number of processes, `s` the message size, α the latency and β the bandwidth. In the second phase, the processes perform a reverse communication to have the same reduced gradients, with an equal communication cost of $\log P \times \alpha + s \times \beta$. This way, the total cost of this approach is $2 \times (\log P \times \alpha + s \times \beta) + reduction$.

Figure 4.5 – The Tarantella Butterfly Reduce-Scatter and Butterfly Allgather algorithms.



Source: The Author

Listings 4.3 presents an example of the Tarantella usage. It automatically broadcast the initial weights of the DNN to all workers without user intervention. It also identifies the available GPUs and makes them visible to TensorFlow automatically during the framework initialization, being easier to use than Horovod. It only requires importing the library and wrapping the Keras model to use the Tarantella model, using the `tnt.Model` function. Tarantella detects and initializes the devices and broadcast the DNN initial weights automatically. To run Tarantella, we need to call an executable script called `tarantella` and, similarly to the Horovod execution using the MPI standard, we need to pass the number of devices per node, the hostnames, and the execution script. Under `tarantella`, the program calls the GASPI API, with an `gaspi_run` command.

Listing 4.3 – Example of using Tarantella in a training script with TensorFlow and Keras.

```
...
# Import Tarantella library
import tarantella as tnt
...
# Create a Keras model
...
# Create a Tarantella model from the Keras model
model = tnt.Model(lenet5_model_generator())

# Compile the wrapped model
model.compile(optimizer = opt, ... )
...
```

4.1.3 Frameworks Overview

Table 4.2 summarizes the characteristics of the frameworks. We select Horovod and Tarantella based on the programming languages they are implemented, support for GPUs, NCCL, and cuDNN, for multi-node environments with multi-GPUs, and available documentation. Despite that, they use different distribution libraries and all-reduce algorithms for the same data parallelization strategy. So we can evaluate the performance of each implementation and compare them for the same experimental setup. Horovod is a more stable framework, in its 65th release, popular in several courses and tutorials on DDL (NVIDIA, 2020b; Microsoft, 2020; AWS, 2020), while Tarantella is a most recent framework, in its 4th public release. Also, Tarantella automatically initializes the GPUs with TensorFlow and provides a data partitioning approach considering the number of the available ranks. Horovod users need to partition the data, explicitly call the broadcast function and initialize the devices using TensorFlow.

Table 4.2 – Overview of the selected frameworks: Horovod and Tarantella.

Feature	Horovod	Tarantella
Year of launch	2017	2020
Distributed library	MPI	GPI
Programming language	Python and C++	Python and C++
Parallelization strategy	Data parallelism	Data parallelism
Data distribution	Manually	Automatically
Devices support	CPU, GPU	CPU, GPU
Interconnection	Ethernet, Infiniband	Ethernet, Infiniband
All-reduce Algorithm	Ring-AllReduce	Butterfly
Versions used	0.22.1	0.7.0

Source: The Author

Horovod and Tarantella implement data parallelism, where we copy the DNN model for each allocated device and average all partial gradients during the *backpropagation* phase. Equation 4.1 shows that the total gradients calculated with the $\partial Loss$ over the training parameters ∂w, is equivalent to computing local gradients in parallel in `P` workers (LE et al., 2018). In Equations 4.2 we represent that the total dataset, represented by `s`, can also be represented as the sum of partial and independent data from this dataset ($d_1 ... d_P$), calculated in different `P` devices. Equation 4.3 proves that the data parallelism strategy results in the same gradients as using serial training. In this Equation, $\partial f(x_i, y_i)$ represents the partial derivatives of the loss function with respect to the weights ∂w (VIVIANI et al., 2019). We assume the same parameters and model in each device

but different chunks of data.

$$\frac{\partial \text{ Loss}}{\partial w} = \frac{1}{P}\Big[\frac{\partial l_1}{\partial w} + \frac{\partial l_2}{\partial w} + \cdots + \frac{\partial l_P}{\partial w}\Big] \tag{4.1}$$

$$\begin{aligned} s &= (d_1 + ... + d_P) \\ \frac{s}{P} &= d_1 = ... = d_P \end{aligned} \tag{4.2}$$

$$\begin{aligned} \frac{\partial \text{ Loss}}{\partial w} &= \frac{\partial\Big[\frac{1}{s}\sum_{i=1}^{s} f(x_i, y_i)\Big]}{\partial w} = \frac{1}{s}\sum_{i=1}^{s}\frac{\partial f(x_i, y_i)}{\partial w} \\ &= \frac{d_1}{s}\frac{\partial\Big[\frac{1}{d_1}\sum_{i=1}^{d_1} f(x_i, y_i)\Big]}{\partial w} + \cdots + \frac{d_P}{s}\frac{\partial\Big[\frac{1}{d_P}\sum_{i=d_{P-1}+1}^{d_{P-1}+d_P} f(x_i, y_i)\Big]}{\partial w} \\ &= \frac{d_1}{s}\frac{\partial l_1}{\partial w} + \cdots + \frac{d_P}{s}\frac{\partial l_P}{\partial w} = \frac{s}{P}\frac{1}{s}\frac{\partial l_1}{\partial w} + \cdots + \frac{s}{P}\frac{1}{s}\frac{\partial l_P}{\partial w} \\ &= \frac{1}{P}\Big[\frac{\partial l_1}{\partial w} + \frac{\partial l_2}{\partial w} + \cdots + \frac{\partial l_P}{\partial w}\Big] \end{aligned} \tag{4.3}$$

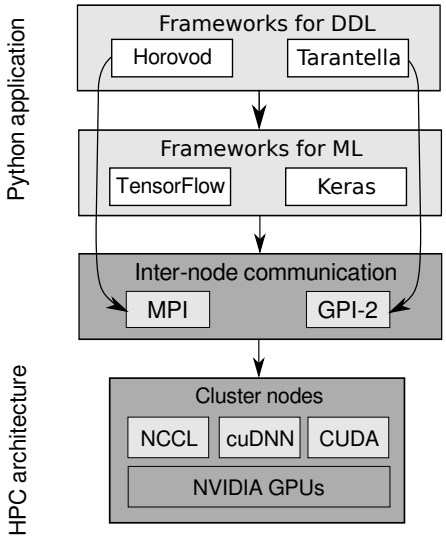
4.2 Frameworks Stack

Figure 4.6 presents the execution stack for Horovod and Tarantella. At the Python application level, we define the framework chosen in an existing training script using Tensorflow and Keras. We need to add and modify some lines of code to guarantee the model will use distributed training for the workers selected in the command line. The inter-node communication to train the network using clusters multi-node with multiple GPUs will use an API for distributed computing. Horovod uses MPI and Tarantella GPI-2. The frameworks are responsible for managing the resources used to train ML frameworks. At the HPC architecture level, the frameworks use CUDA and can use the NCCL and cuDNN NVIDIA libraries to optimize operations in NVIDIA GPUs.

4.3 Model and Parameters Selection

Image classification is one of the most representative and popular models for performance evaluation since it requires a higher computational process, memory, and execution time to achieve more accurate results. We selected a Lenet-5 model with the Modified National Institute of Standards and Technology (MNIST) dataset (LECUN et

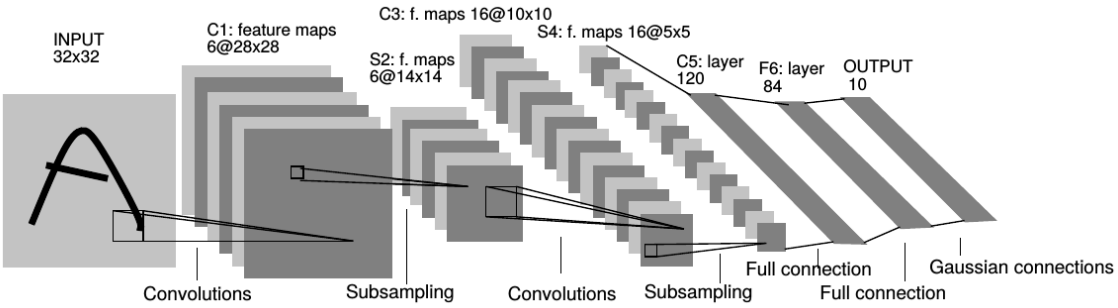
Figure 4.6 – Distributed Deep Learning frameworks execution stack.



Source: The Author

al., 1998). Figure 4.7 presents the LeNet-5 architecture, implemented as five layers containing the training parameters, including three convolutional layers (C1, C3, and C5), two sub-sampling layers (S2, S4), and one fully-connected layer (F6). The convolutions apply filters of 5×5 size. The first convolution layer (C1) results in six feature map of 28×28 size. In the second layer (S2), the feature map is reduced by half its size. In the second convolutional layer (C3), we have 16 filters, resulting in 10×10×16 filtered map. Then another sub-sampling (S4), reducing the size to 5×5×16. Finally, a last convolutional layer with 120 filters resulting in 120 values. MNIST is a popular image classification dataset composed of 60,000 handwritten digits images, more 10,000 examples for testing, of size 28×28 (784 pixels) and 10 output classes (LECUN; CORTES, 2010).

Figure 4.7 – LeNet-5 architecture.



Source: (LECUN et al., 1998)

In CNNs, data is split by the sample dimension and gradients computed in parallel are averaged after each epoch. (DETTMERS, 2016). Smaller batch sizes are recommended for achieving higher accuracy, so the gradients are averaged over fewer data

(MASTERS; LUSCHI, 2018). Bigger batch sizes result in gradients less representative of the entire dataset, achieving lower accuracy. Considering our goal to compare the frameworks algorithms and strategies to train in different devices, we selected different minibatch sizes, which can reflect the parallelism decisions inside a minibatch, between minibatches, and between epochs for each framework on each GPU model. For the minibatch sizes selection, we consider the data parallelism for Tarantella, where the minibatch size must be a multiple of the number of ranks because of the `micro-batches` approach, as explained in Section 4.1.2. Otherwise it drops the remainder data. Also, we considered the maximum minibatch size according to the GPUs memory, and the number of GPUs available to have at least one batch per GPU in all cases.

Table 4.3 present the minibatch sizes selected for the Lenet-5 model and the number of minibatches processed per epoch. We use 54,000 images for training and 6,000 images for validation. Validation is a process to evaluate the model fit at the end of each epoch and tune the model hyperparameters if necessary. It is recommended to choose validation data different from the training and testing data to result in unbiased training. Keras directly incorporate these values in the `model.fit` function. For testing the model, we used 10,000 different images. Since we want to measure the performance gains scaling the training for more devices in terms of execution time, we use strong scaling, which can achieve the same accuracy as the sequential training for small datasets, as in the case of MNIST (CUNHA et al., 2018).

Table 4.3 – Minibatch sizes for the Lenet-50 with MNIST dataset.

Minibatch size	Batches per epoch
100	540
180	300
360	150
720	75
1500	36
2250	24

Source: The Author

We configured full-factorial experimental designs using Jain's methodology for computer systems performance analysis (JAIN, 1991). The factors are grouped into two categories: (i) related to the training and (ii) related to the environment. For the first, we consider the DL frameworks, models, datasets, and minibatch sizes. For the second, we consider the number of nodes and GPUs. Different from most works that analyze

a single experiment, we performed ten repetitions for each case with experiments order listed randomly. It resulted in 84 experiments. We use a learning rate of 0.01, linearly scaling by the total number of workers, since it helps to match the accuracy and learning curves between using small and large minibatches using SGD models (GOYAL et al., 2018).

We run the frameworks over TensorFlow, since Tarantella only has support for it, and it is a well documented tool, with several tutorials available on the internet. We used `bash` scripts to launch our experiments with each full-factorial entry and parse the proper parameters for each experiment considering the number of workers. We set the same seed to be used by TensorFlow in the weights initialization and the dataset shuffling for each experiment. It restricts the same results for the same experiment repetition and case for different clusters and frameworks. Listing 4.4 presents our approach to randomize the dataset.

Listing 4.4 – TensorFlow functions used to shuffle and batch the dataset.

```
train_dataset = tf.data.Dataset.from_tensor_slices(
        (x_train, iy_train))
train_dataset = train_dataset.shuffle(
        len(x_train), shuffle_seed)
train_dataset = train_dataset.batch(
        args.batch_size)
train_dataset = train_dataset.prefetch(
        buffer_size = tf.data.experimental.AUTOTUNE)
```

The `shuffle` function is used to ramdomize the elements of the dataset using a seed. We pass the size of the entire dataset since the shuffle creates a buffer with `buffer_size` of the first parameter, which ideally has the same size of the dataset. In our experiments, we set the `shuffle_seed` according to the GPU identificator, and repetition, to ensure the same repetition with the same amount of GPUs will train over the same data for different frameworks and batch sizes. The `batch` function will group the dataset into batches of size `minibatch size`, set as argument by the user in the run line. It results in an additional outer dimension to the dataset of size `batch_size`. They recommend to use `drop_remainder` parameter set to *true*, to deal with dataset not multiple of the batch size. The default is *false*, and can lead to an unknown dimension for the entire dataset type. Finally, the `prefetch` function is used to increase the performance of memory access, improving latency and throughput. It prefetches later elements from the dataset while the current element is being processed. If the

dataset was previously batched, it uses a buffer of size `buffer_size`. We apply the `tf.data.experimental.AUTOTUNE` option, which will prompt the tf.data runtime to tune the number of values prefetched dinamically at runtime. Then we pass the `train_dataset` to the Tarantella model.fit function, as presented in Listings 4.5.

Listing 4.5 – Keras model.fit function used to train the batched model.

```
model.fit(train_dataset,
        use_multiprocessing = False,
        callbacks=callbacks,
        tnt_distribute_dataset = True,
        validation_data = val_dataset,
        epochs = args.number_epochs,
        verbose = 0)
```

5 RESULTS: COMPARING HOROVOD AND TARANTELLA

Our performance analysis methodology consisted of performing experiments with the selected models and environments considering the DDL framework's algorithms and strategies. We compared Horovod and Tarantella in three clusters `chifflot`, `chifflet`, and `gemini`, using a full-factorial design with different batch sizes, with one network model, and one dataset. We selected 100 epochs for the training, which was sufficient for achieving high accuracy in an acceptable execution time.

All experiments collected the Keras callbacks and NVSMI traces. The difference in runtime with and without the logs was small, up to 4% of the entire execution time for the smaller batch size for Tarantella and up to 2.5% for Horovod. Due to the variability in the overhead, in practice, this difference remains unnoticeable. Also, the callbacks generate small datasets, from 20KB to 80KB for the NVSMI log, and from 40KB to 8MB for the callbacks.

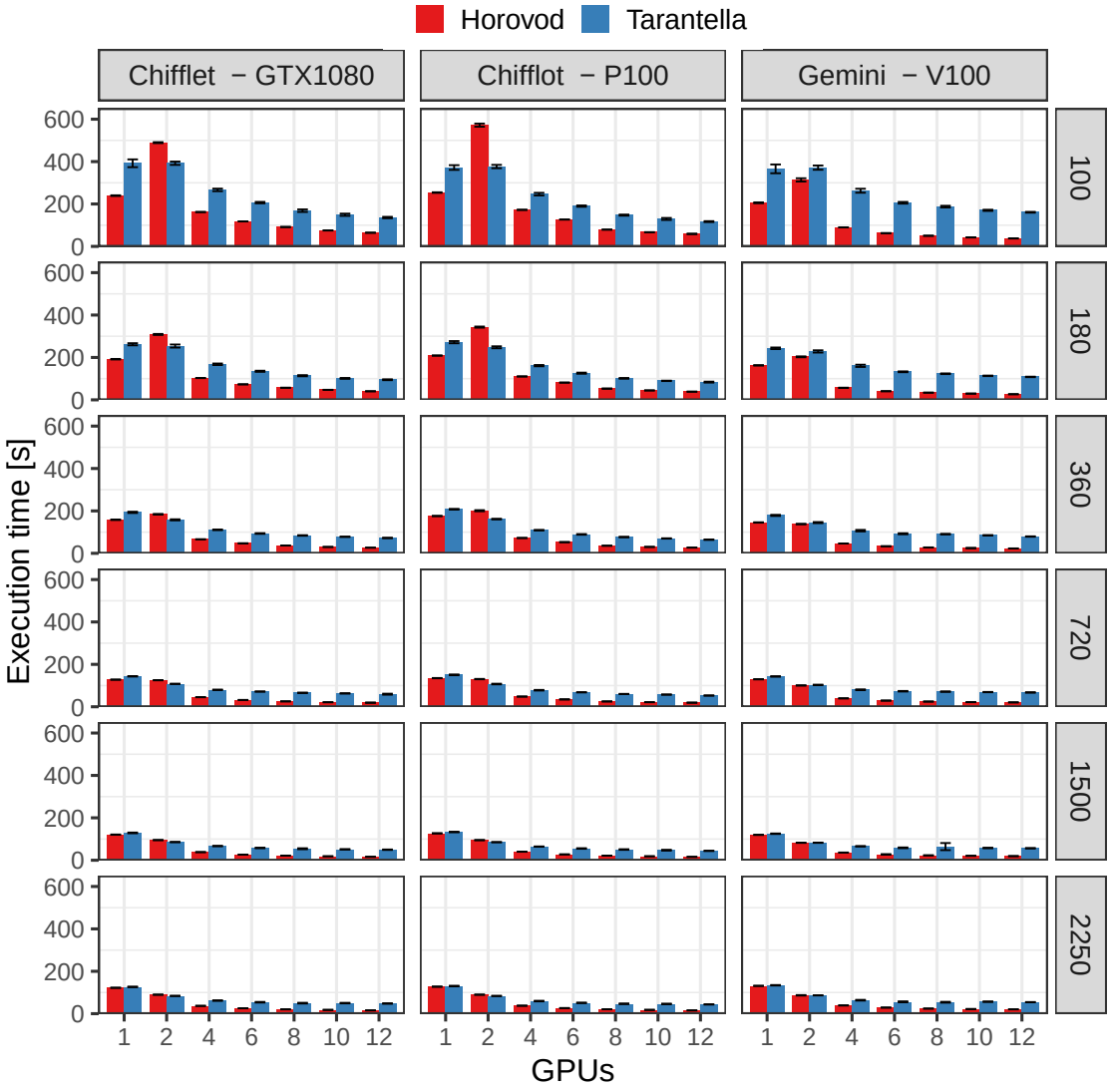
The main contribution of this Chapter is a comparison of two DDL frameworks that applies data parallelism for distributing the training for up to 12 workers in three different clusters. Horovod presented the best performance regarding execution time, achieving higher efficiency scaling than Tarantella. On the other hand, Tarantella achieved higher and more stable training accuracy. We found a bottleneck in Tarantella's implementation when leaving the framework set up for sequential training and not distributing test batches. The time with initialization of the frameworks before starting the model fitting is similar for both frameworks. Horovod made higher usage of GPUs during training, but this value drops for the cluster Gemini, with the most recent GPU cards. Using Keras callbacks, we found that Horovod faster training is because it computes an epoch faster. It also revealed a space for improvement in Horovod initialization time compared with Tarantella.

Section 5.1 presents the training time comparison between Horovod and Tarantella for our full-factorial design. Section 5.2 shows the frameworks scaling efficiency when adding more GPUs. Section 5.3 compares the loss and accuracy to verify if distributing the training can impact the model prediction. Section 5.4 presents our findings using Keras callbacks for profiling the training.

5.1 Training Time Performance Analysis

Figure 5.1 presents the mean execution time for all setups, where each vertical facet represents a cluster and the horizontal facets represent the minibatch sizes. We use a confidence interval of 99.97% for 10 repetitions each, showing the mean time and the error bar in black. Horovod presented the best performance using four or more GPUs compared to Tarantella for all cases. It also scales until 12 GPUs, different from Tarantella, which perform similarly when increasing from 6 to 12 devices.

Figure 5.1 – Execution time for LeNet-5 over MNIST. Each facet represents a batch size.

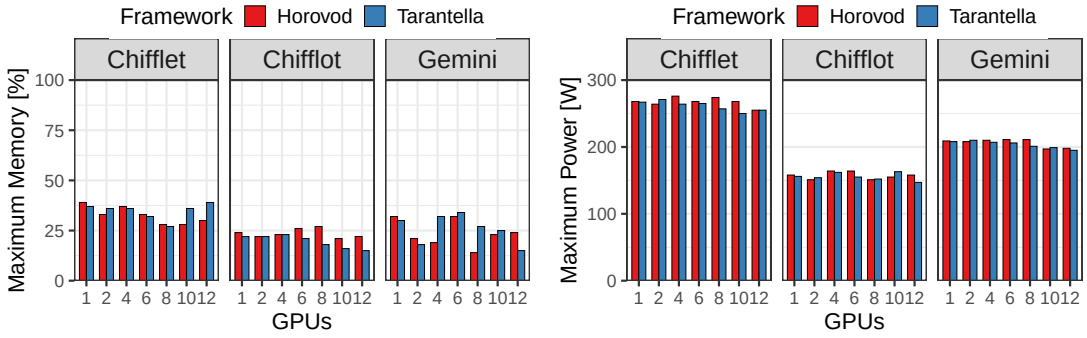


Source: The Author

Comparing the clusters used, even having the overhead of making a copy of the entire model in each worker to run a distributed training with data parallelism, the frameworks achieved similar results for all GPU models, except for using two GPUs in a 100

batch size. This similarity happens since the MNIST dataset size does not explore the GPUs memory and the Lenet-5, a model with few layers, does not explore all the GPUs power to run. Figure 5.2 presents the maximum power and memory achieved for the case that explores these factors the most, with a batch size of 2250 images. We had less than 50% maximum use of the total memory for the GPUs, and less than 300W consumed for all GPUs. The variation in the maximum power can be explained by the frequency for the GPU models in each cluster, the lower frequency for Chifflot of 1190MHz limited the power to less than 200W, and the highest power for the Chifflet was possible due to its frequency of 1481MHz. The power variation did not impact the overall execution time.

Figure 5.2 – Peak memory and power for one experiment with batch size 2250.



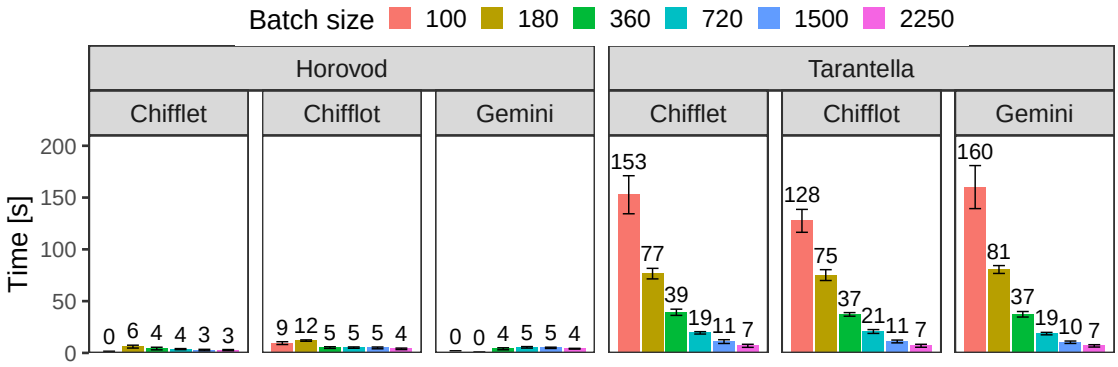
Source: The Author

The batch size defines the number of image samples that will be propagated through the network during training at the same time. Smaller batches finish faster, but they increase the latency of data sent from host and device and between devices, without taking advantage of the GPUs processing potential and taking longer time to execute. Using larger batches decrease the execution time and increase the throughput by processing more data at a time. The GPUs will perform fewer iterations per epoch to pass through all dataset, decreasing the communication between devices. We notice this behavior in Figure 5.1, where the smaller batches took longer to execute for the all number of GPUs set, compared to bigger batch sizes. A network traffic analysis about this higher throughput using smaller batches was reported before for a CNN architecture in three nodes with CPUs (ASPRI; TSAGKATAKIS; TSAKALIDES, 2020).

We run the experiments with one GPU using the frameworks to evaluate their overhead comparing to the pure Tensorflow execution, since Horovod, for example, reports no overhead if applied to a single-worker execution. Figure 5.3 presents the overhead of using the frameworks for a non-distributed training with one GPU compared to the pure TensorFlow implementation. Horovod, as expected, had a similar execution time as the

experiments using pure TensorFlow. The smallest batch sizes of 100 and 180 for Gemini, and 100 for Chifflet present the same execution time for Horovod and pure TensorFlow, considering the measurement variability in the experiments with 10 repetitions. Tarantella added a significant overhead of up to 160 seconds for the smaller batch size in the Gemini cluster. Investigating the Tarantella source code, we observed it does not implement a verification if the training is using more than one GPU. Tarantella initializes the Gaspi library and starts distributing the training for the GPUs listed by `tnt.get_size()` normally.

Figure 5.3 – Overhead of using the DDL frameworks for a single GPU training.

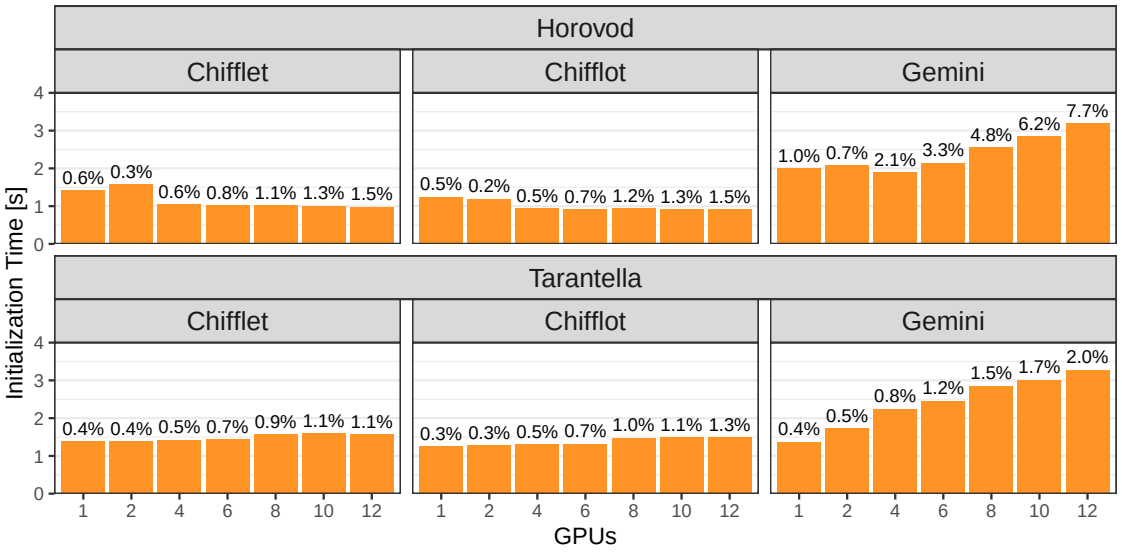


Source: The Author

We supposed that the extra time in Tarantella for one GPU could affect other experiments if related to an initialization phase, especially when scaling for more devices. Figure 5.4 presents the total time between the training script initialization and the beginning of the first batch for an experiment with a batch size of 100 images. For each setup, we show the percentage it represents over all execution time. Before starting the script, the frameworks' initialization time was measured separately, around 0.14 seconds for Horovod and 0.33 for Tarantella. The frameworks spent similar time before training, Horovod spent 7.7% of the time with initialization for 12 GPUs at Gemini, which is a high percentage since the training time was very low, as shown in Figure 5.1. This visualization indicates that Tarantella's extra execution time compared to Horovod is not related to its initialization.

The Horovod execution time increased with minibatch size 100, when going from one node with one GPU to two nodes, one GPU each, as presented in Figure 5.1. Figure 5.3 shows that Horovod had no overhead in a non-distributed training, which means its execution time is dedicated on training. This execution time increase can be explained by Horovod start applying its reduction algorithm and NCCL operations when using distributed training. Figure 5.5 presents the maximum GPUs usage achieved during training captured with NVSMI. For a DL training, the higher the usage of a GPU the better the

Figure 5.4 – Time spent with initialization for an experiment with batch size 100, and the percentage of time it represents over the execution makespan.



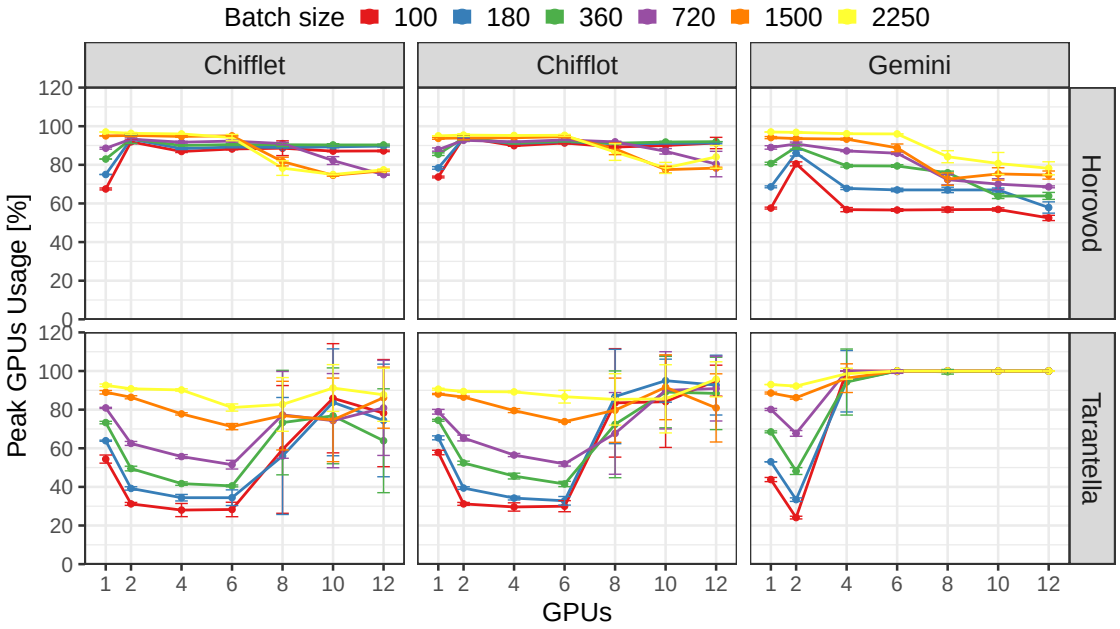
Source: The Author

algorithm is, spending fewer time with the work outside the devices with communication, for example. Here we confirm Horovod starts more the device when distributing the training in Chifflet, and Chifflot clusters. An unexpected behavior in the Gemini cluster was that Horovod made fewer usage of the GPUs when distributing to more devices. Tarantella on the other side, presented higher GPU usage when adding more than four GPUs to in Gemini, training. Since it runs over a non-distribute training, the peak GPU drops going from one to two GPUs. It also presented a high variation in the Chifflet and Chifflot clusters for more than eight GPUs.

5.2 Frameworks Scaling Efficiency

Using multiple GPUs located in different computational nodes can lead to data transfer overhead. To guide our investigation of how the frameworks could be improved in their current implementation, we represented their expected gains if they scale linearly based on the serial training, presented in Figure 5.6. It is calculated as the ratio between the time training in one GPU with the framework over the number of devices added. Horovod approached the expected gain for 4 GPUs or more for larger batches, with more than 180 images (Figure 5.6a). For smaller batches, with more data transferred through the network, Horovod performance in the Gemini cluster, with the newest GPUs used, is significantly higher than in the other clusters. Performance for Chifflet and Chifflot

Figure 5.5 – Horovod and Tarantella peak GPUs usage when scalling more devices and larger batches.



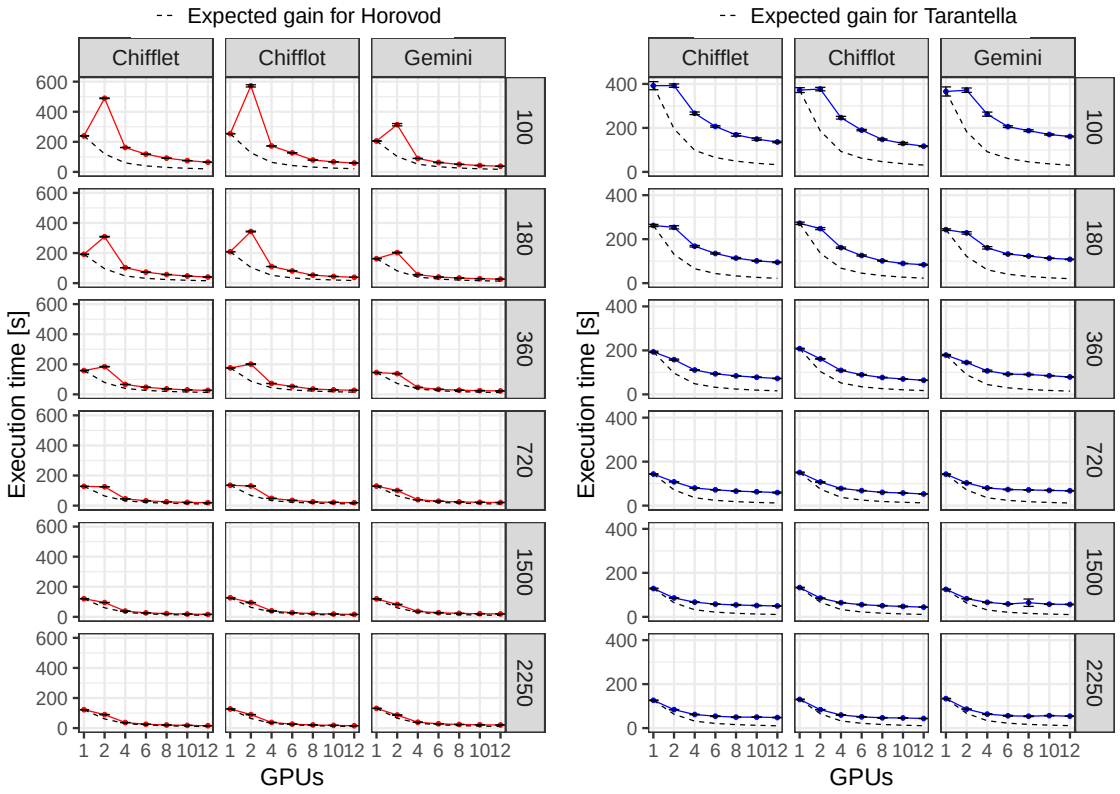
Source: The Author

could still improve the performance until ≈105 seconds for 6 nodes and ≈81 seconds for 6 nodes.

Tarantella (Figure 5.6b), could gain from ≈26 seconds to almost ≈170 seconds if achieved the expected gain for 4 GPUs. Of course, a linear speedup is hard to accomplish since adding more devices will increase the communication overhead and the time processing synchronization algorithms. Still, when comparing Tarantella with Horovod, we notice that the last approximates the expected gain, especially for larger batch sizes. For Tarantella, experiments in Gemini with batches of size 720 or greater take longer execution time than using the other clusters. The Gemini cluster has two nodes with eight GPUs each and four GPUs per NUMA node, impacting long memory latency compared to the other clusters, with two GPUs per NUMA node. Even though Horovod benefit from using the V100 GPU cards. Figure 5.7 compares the scaling efficiency for Horovod and Tarantella. We used the Equation 5.1 to calculate it, where T_1 is the execution time with one GPU, and T_n with n GPUs. In our strong scaling approach, Horovod achieved higher efficiency for all cases with more than 6 GPUs, with a difference of almost 50% in the efficiency between Tarantella and Horovod, indicating that Tarantella could potentially improve its performance.

$$E = \frac{T_1}{n \cdot T_n} 100\% \tag{5.1}$$

Figure 5.6 – Execution time (coloured lines) and expected gains (black dashed lines) for the frameworks if they scale linearly for more GPUs.

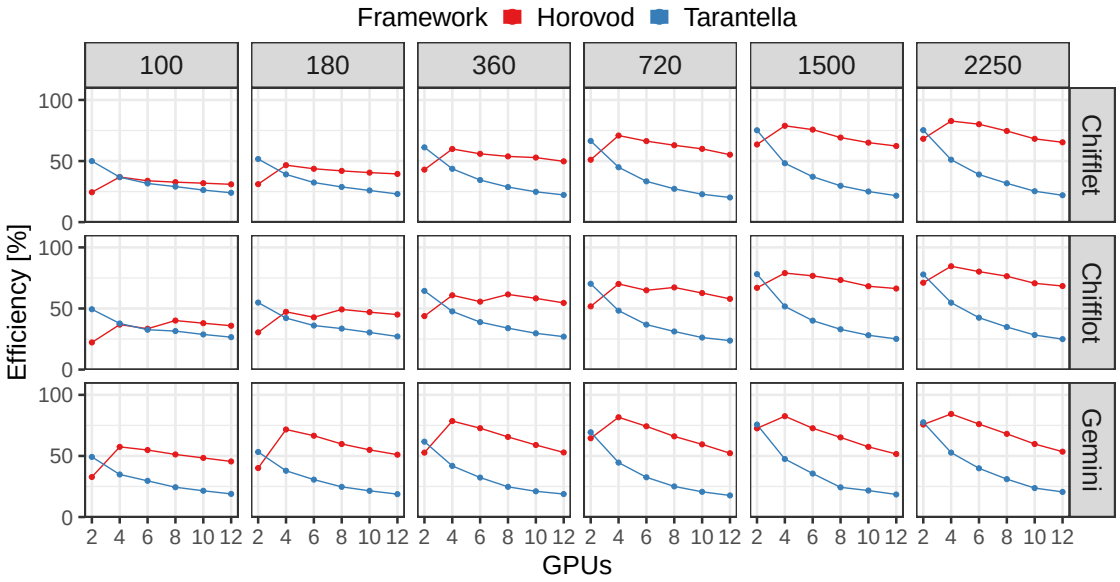


(a) Expected gain for Horovod.

(b) Expected gain for Tarantella.

Source: The Author

Figure 5.7 – Efficiency scaling for more GPUs.



Source: The Author

Horovod efficiency improves when increasing the batch sizes, 80% for Chifflet and Chifflot for 2250 batch size, and 76% for 1500 batch size. The same happens for

Tarantella, but more slightly, with ≈41% for Chifflet and Chifflot, 2250 batch size, and ≈39% for 1500 batch size. Both frameworks achieve lower scaling efficiency for more than 6 GPUs for minibatch sizes of 100 and 180 since the batches are not large enough to enable further gains. Horovod is ≈84% faster for 4 GPUs in all clusters for 2250 batch size, the highest efficiency, and for 12 GPUs achieved 68% in Chifflot, 65% for Chifflet, and 53% in Gemini. Tarantella's maximum scaling efficiency is 78% for Chifflot with 1500 batch sizes with 2 nodes, and it is only ≈23% faster for 12 nodes in all clusters. Overall, Horovod efficiency is higher than Tarantella for 4 GPUs or more, and both frameworks scale up to 12 GPUs for a batch size larger than 720. Chapter 6 presents reasons of why Tarantella is slower than Horovod even with the same DDL strategy.

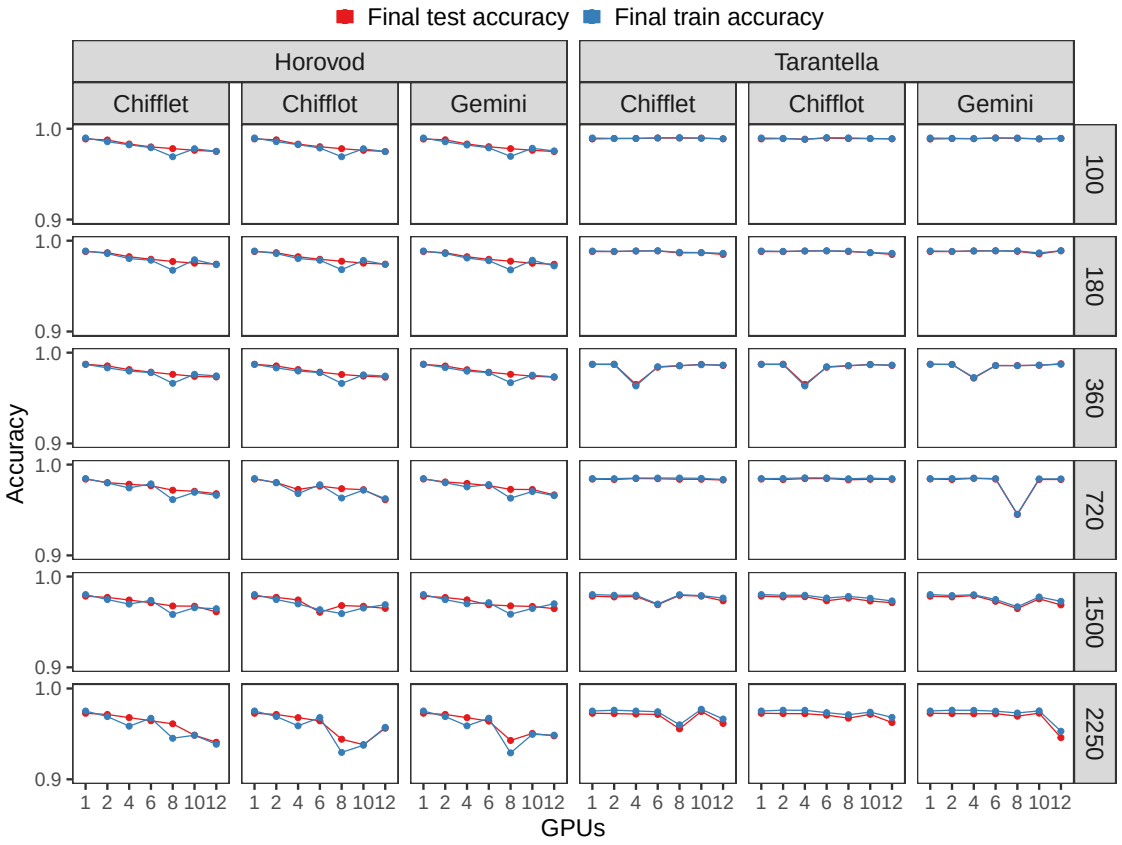
5.3 Loss/accuracy

Even though our research focuses on evaluating and comparing the performance of the frameworks from an HPC perspective, we believe it is important to measure the accuracy of the models since it will impact the users' choice of the best DDL framework for CNNs. The **accuracy** represents the percentage of the data correctly classified from the total data used for data classification problems. The **loss** presents the distances between true and predicted values, the probability of making correct classifications. Increasing the batch size, we increase the number of data treated simultaneously per iteration, so the prediction of many examples will be less representative than using smaller batches, which is why we have a lower accuracy when using larger batches.

Figure 5.8 presents the final accuracy achieved for training and for the test phase, which uses 10,000 images different from the training. Horovod decreased its accuracy more than Tarantella when scaling, and in general, Tarantella achieved higher and more stable accuracies for the setups. There are some very low cases for 360 and 720 batch sizes only that stand out. We did not tuned hyperparameters for scaling the global batch size up as recommended when focusing on the model accuracy (GOYAL et al., 2018).

Figure 5.9 presents the final train and test loss. The Horovod loss achieved a higher variation for 8 and 10 nodes with all machines and batch sizes. The test loss kept a similar value for Tarantella, but not for Horovod, behaving better than during training. Even though we shuffle the dataset so the same experiment case with different frameworks will work over the same dataset order, we can not control the averaging algorithms and optimizations for distributed training with the frameworks. Also, we notice an anoma-

Figure 5.8 – Test and train accuracy obtained at the execution end for Horovod and Tarantella in all setups.

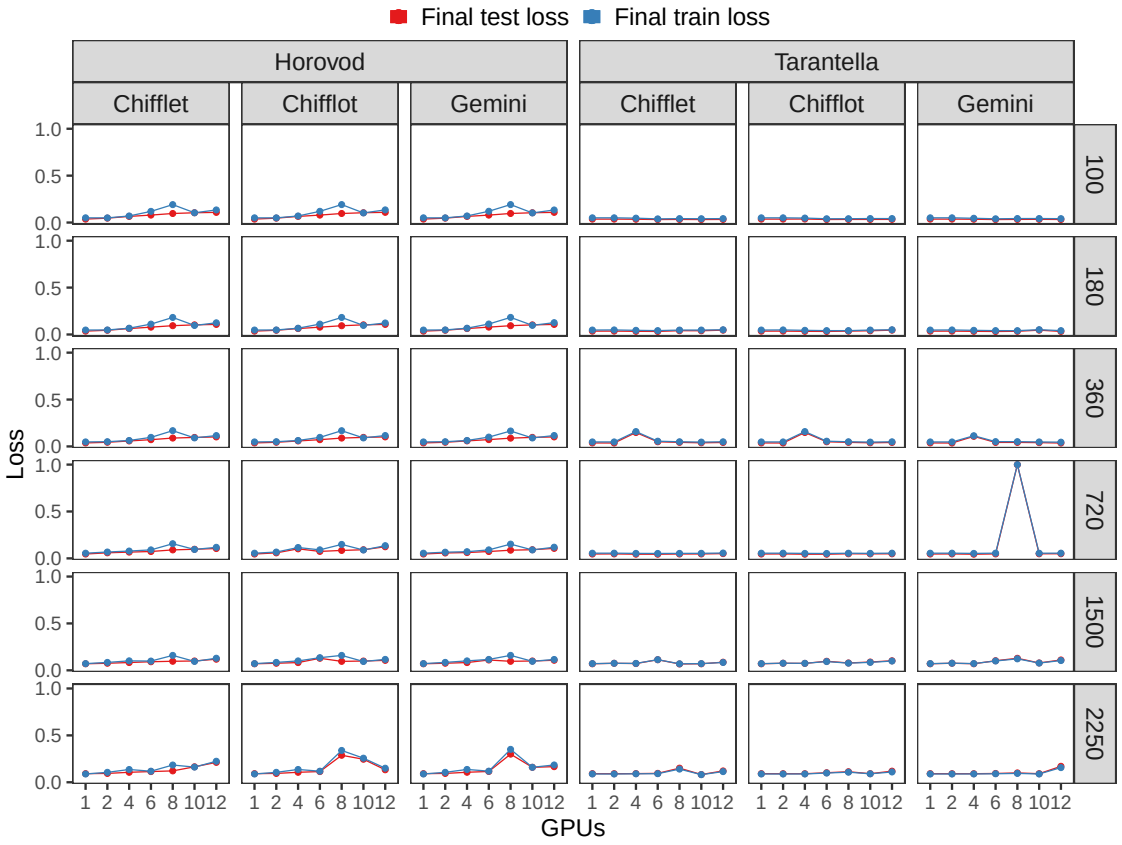


Source: The Author

lous case for Tarantella with 8 GPUs in the Gemini clusters, with the highest loss. We conclude that Horovod's prediction for the image classification model is less stable than Tarantella scaling for more devices, and in some cases, also scaling for larger batches, while Tarantella behaved well for all cases, for the train and test predictions.

Figure 5.10 shows the accuracy and the loss variation through the training epochs for the smallest and the bigger batch sizes, on all machines, for 4 and 12 GPUs. We only show the first 50 epochs out of 100 since it stops converging after 40 epochs. As expected, predictions over small batch sizes converge faster than averaging gradients over bigger batch sizes. Loss started presented slight variability for 12 GPUs with 100 batch sizes with Tarantella and higher variability for 2250 batch sizes. Using fewer devices, they achieved a similar convergence rate. For the larger batch size, both frameworks presented several loss and accuracy variations until epoch 20. Still, they converge almost at the same time, between epoch 20 and 30. The huge loss for the first epochs using batch size 2250 is also expected due to several data being classified. Tarantella kept more stable accuracy when scaling to more devices (Figure 5.8), but when looking at the accuracy for

Figure 5.9 – Test and train loss obtained at the execution end for Horovod and Tarantella in all setups.



Source: The Author

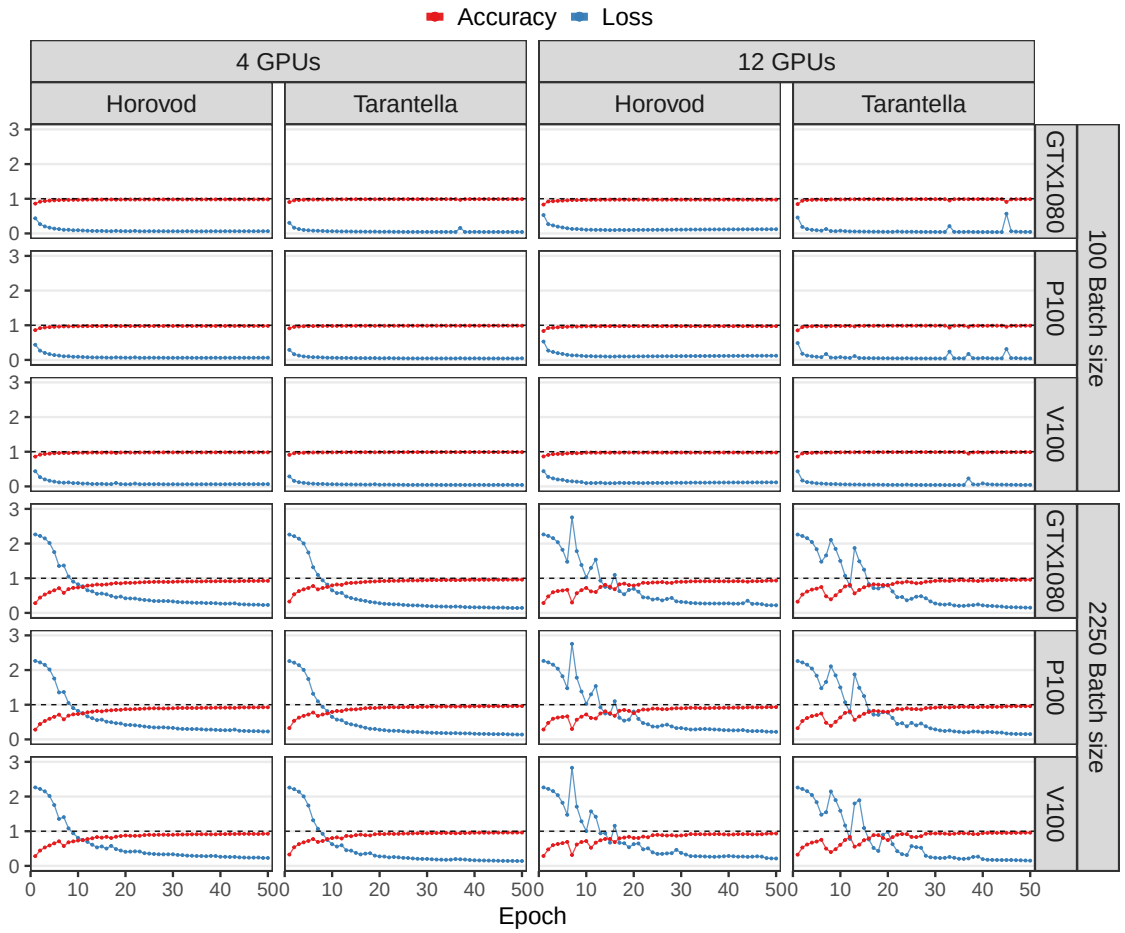
each epoch along with the training time, Tarantella presented more variability to converge than Horovod.

5.4 Training workflow

We saw in the previous Sections (5.1 and 5.2) that Horovod scales better than Tarantella and has a lower total execution time in all cases for more than four workers. This section depicts this total execution time for Horovod and Tarantella using the Keras callbacks to get the time spent processing epochs and batches to identify the synchronization methods used by each framework. With this, we can understand the training workflow for DDL frameworks and start classifying the time spent with computation and communication.

Table 5.1 presents the execution time processing batches during testing (**Test Batches**), during training (**Train Batches**), and the overall execution time of each case (**Total Time**). We also present the time difference (**Difference**) between the Total Time,

Figure 5.10 – Training loss and accuracy comparison for Horovod and Tarantella for some cases throughout the epochs. Lower loss is better. Higher accuracy is better, up to 1.



Source: The Author

not considering the Train and Test Times. We selected the oldest and most recent GPU versions, with the smallest and biggest batch size for 2 and 12 GPUS. Our goal was to understand if Horovod is faster than Tarantella for processing the batches faster or if the time processing batches are similar, so the difference comes from other reasons. Both frameworks decrease their total execution time when scaling for more workers. For time computing batches during training, Horovod is 9.88 times faster for Chifflot and 8.82 times faster for Gemini going from 2 GPUs to 12 with 100 batches size. The maximum that Tarantella goes is 3.73 times faster for 100 batch sizes in Chifflot. We notice that the time testing drops when scaling the training in all cases for Horovod, but it remains similar or even increases for Tarantella. It indicates that Tarantella does not distribute the testing batches as it does for the training batches.

Using Keras callbacks, we can start to investigate the amount of time that is not spent with computing during distributed training. In the “Difference” column, we present the amount of time the experiment is not processing batches. Scaling DDL for more

Table 5.1 – Total execution time processing batches during testing (**Test Batches**), during training (**Train Batches**), the total execution time for running the experiment (**Total Time**), and the time that the experiment is not computing batches (**Difference**).

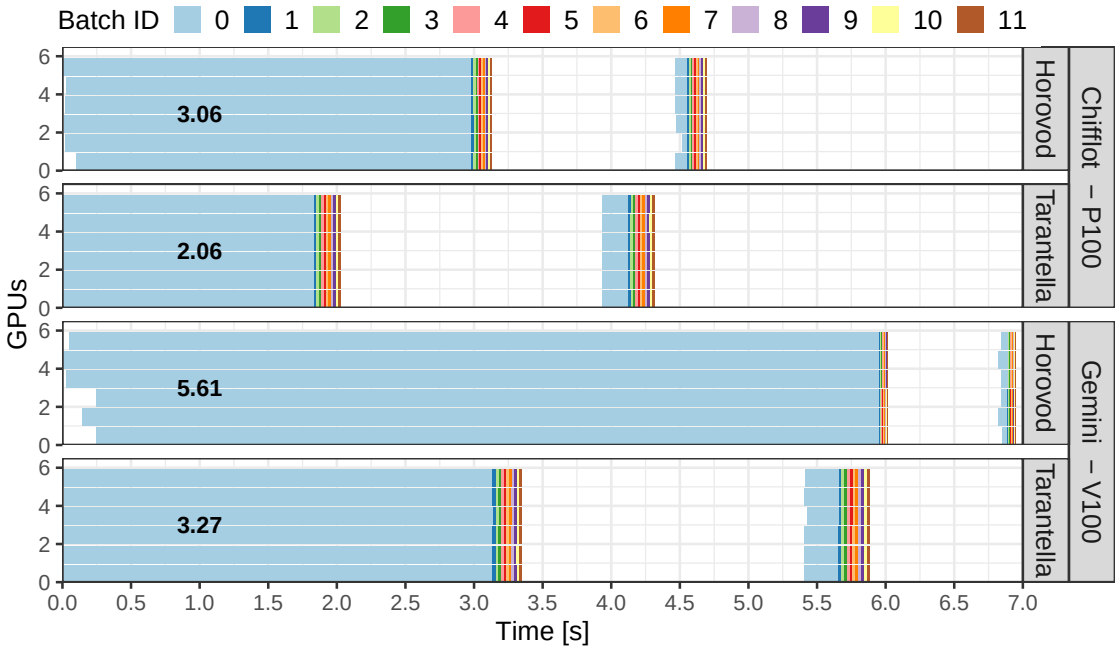
Horovod						
Cluster	Batch	GPUs	Test Batches (TrB)	Train Batches (TeB)	Total Time (TT)	Difference TT - (TrB + TeB)
Chifflot	100	2	6.89	**544.93**	564.69	12.87
Chifflot	100	12	0.98	**55.21**	**59.36**	3.17
Chifflot	2250	2	2.68	76.42	90.96	**11.86**
Chifflot	2250	12	0.61	13.09	**16.6**	**2.9**
Gemini	100	2	7.41	**299.37**	320.61	13.83
Gemini	100	12	1.36	**33.94**	**41.97**	6.67
Gemini	2250	2	2.01	73.61	88.51	12.89
Gemini	2250	12	0.64	17.54	**23.93**	5.75
Tarantella						
Chifflot	100	2	11.68	**342.75**	370.1	15.67
Chifflot	100	12	10.67	**91.79**	**117.6**	15.14
Chifflot	2250	2	4.44	66.7	84.85	13.71
Chifflot	2250	12	4.42	26.25	**45.29**	14.62
Gemini	100	2	13.2	340.29	370.42	**16.93**
Gemini	100	12	15	128.43	**164.28**	**20.85**
Gemini	2250	2	3.24	69.53	88.11	**15.34**
Gemini	2250	12	4.35	34.9	**58.78**	**19.53**

Source: The Author

workers can accelerate training time since each worker will process the smallest amount of data, but it also increases the time spent with communication among devices to update the gradients. When we scale with Horovod, the time not processing batches also drops, up to 4.08 times slower for Chifflot with 2250 batch size. It indicates Horovod optimizations for Horovod communication have a good result over our model. However, for Tarantella, the time not processing batches remained similar for Chifflet, and it even increases for Gemini.

To understand what was causing the differences for Horovod and Tarantella in different clusters, we present the batches execution for the first epochs in Chifflot and Gemini using a temporal visualization. Figure 5.11 shows the first two epochs and batches for batches of size 100, and Figure 5.12 the first three epochs for 2250 batch size, both with 6 GPUs. The colors represents the batches for each case, 4 batches for 2250 batch size, and 11 out of 45 for 100 batch size (5400 images total). The first batch of the first epoch took much longer than the others to execute, that indicates the initialization time is also considered training execution time for Keras callbacks. Only with the traces, we can not know for sure. Also, the first batch of other epochs took longer, as we can more easily identify in Figure 5.12 since it only computes two batches of size 2250 per epoch.

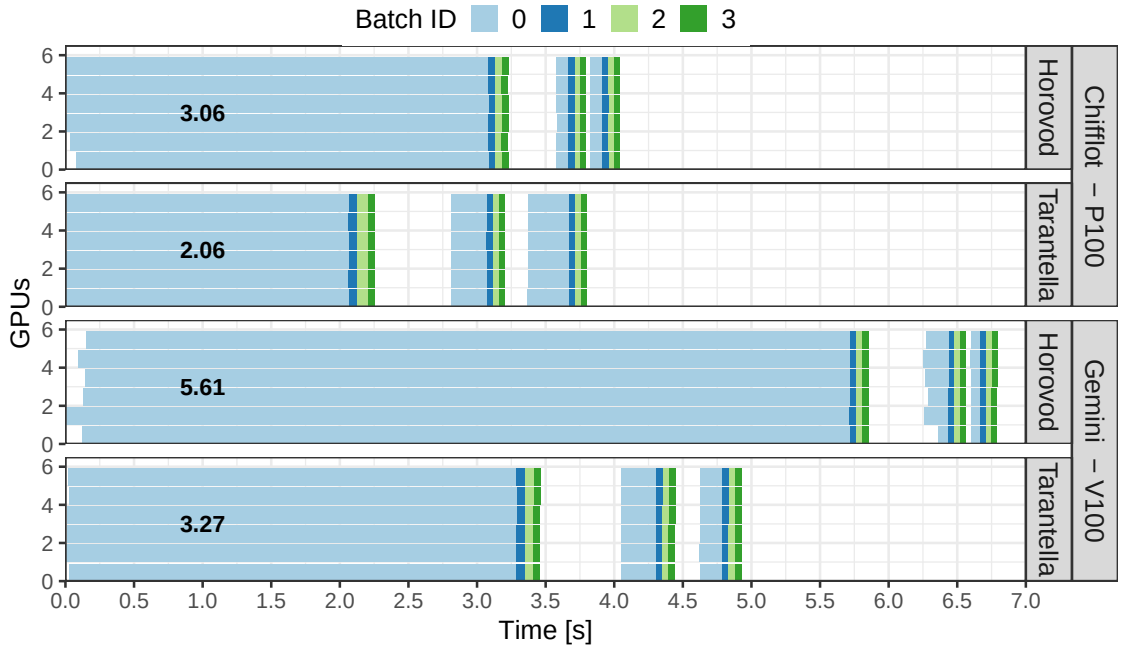
Figure 5.11 – Space/Time view for the first two batches with a batch size of 100 and 6 GPUs.



Source: The Author

We saw in pink in Table 5.1, that Horovod runs faster for 2250 batch size in Chifflot than in Gemini. In the plots, we notice this behavior is due to the faster initialization

Figure 5.12 – Space/Time view for the first three batches with a batch size of 2250 and 6 GPUs.



Source: The Author

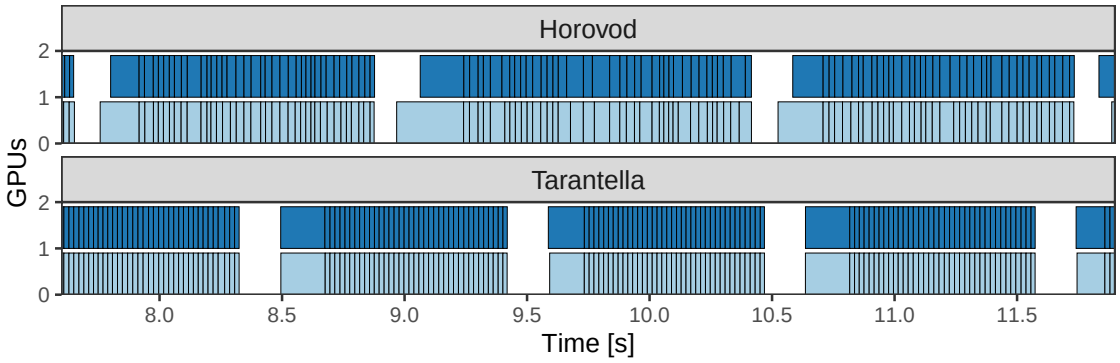
time, which is 1.8 times faster, since the time processing other epochs is similar. The opposite happens for a smaller batch of 100 (yellow values in the table) for Horovod. The total training is faster in Gemini, even though the initialization in Chifflot is 1.97 times faster than Gemini, but the time computing the other epochs for a small batch size is faster at Gemini. That is because V100 GPUs are faster than P100, bringing optimizations to computer tensor operations. The colored values in Table 5.1 for Tarantella show it computes faster for both cases in Chifflot. For smaller batch sizes, the total time processing an epoch is similar, but, as Horovod, the initialization time in Gemini is very time demanding. Tarantella also had a long time between epochs than Horovod, with around 2 seconds between the first and second epoch at Gemini for batch size 100.

The GPU warm-up time is a possible reason for a long time in the first batch of each epoch. Without a persistence mode enabled, the GPUs are unloaded when their file descriptor is not open by another process, generating a possible delay when initializing the GPUs with TensorFlow (TESSER; MARQUES; BORIN, 2021). Looking at Figure 5.11, Horovod time for the first batch of the second epoch is almost the same for processing the other batches. Also, the execution time for the first batches differs between frameworks. Therefore, it is necessary an investigation of watch this initialization time represents.

Zooming into the temporal visualizations of the callbacks, we identify the frameworks synchronization between epochs. Figure 5.13 presents a time interval for training

with 720 minibatch size in 2 GPUs. Both frameworks had a synchronization point at the end of an epoch when they average gradients among all. Horovod let workers that updated their gradients start processing the next epoch, different than Tarantella, which also looked to have a synchronization point at the beginning of an epoch. This approach decreases the time between epochs for Horovod, as shown in Figures 5.11 and 5.12.

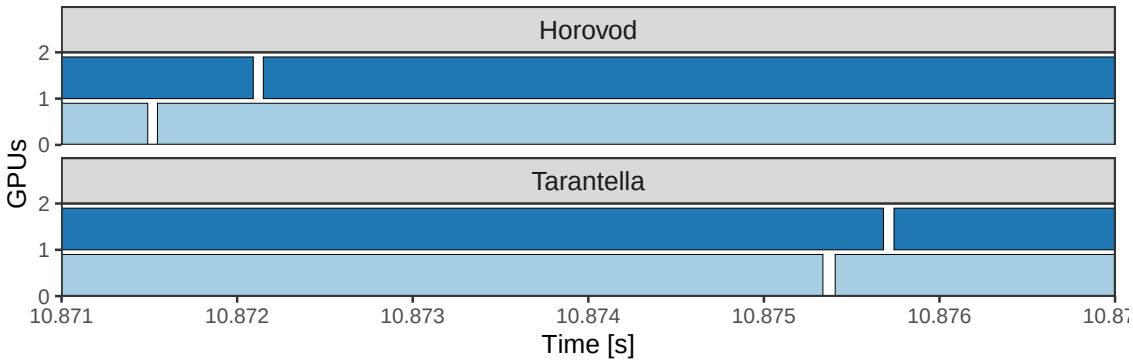
Figure 5.13 – The frameworks synchronization for workers between epochs detected with Keras callbacks.



Source: The Author

Zooming into one batch processing for the same case in Figure 5.14, we confirmed that there is no synchronization between workers for processing the batches. Even using the micro-batches strategy, Tarantella still has long synchronization points between epochs. We believe they implemented Tarantella with micro-batches since they are working on providing pipelining parallelism in the tool, which is not available in the version used. Further investigation needs to be done to identify what is considered the first batch for high-level profilers in ML.

Figure 5.14 – The frameworks synchronization for workers between batches detected with Keras callbacks.



Source: The Author

6 RESULTS: BREAKING THE FRAMEWORKS BLACK-BOX

Measuring accuracy and execution time is important in DL to evaluate the efficiency of a model. Using distributed training adds new concerns when evaluating DL, to understand if the training is taking advantage of the computing resources and minimizing the communication overhead, adding more workers. We applied tools for performance analysis of distributed applications on Horovod and Tarantella and correlated with results presented by the Keras Callbacks. We used execution traces, profiling, and visualization to break the DDL framework's black-box, written in Python and C++/C programming languages. Section 6.1 introduces Score-P and presents our results using it with Horovod for code instrumentation. Section 6.2 presents our findings combining the frameworks instrumentation with NVProf to our evaluation.

6.1 Instrumenting Deep Learning with Score-P

Score-P is a performance measurement infrastructure to instrument applications with different programming paradigms (KNÜPFER et al., 2012). It generates event traces in the Open Trace Format Version 2 (OTF2), and profile in the CUBE4 format (KNÜPFER et al., 2012). Score-P allows instrumentation at the user level, through manual code instrumentation, or it can collect events of multi-process, thread-parallel and accelerator-based application.

A literature review of parallel and distributed deep learning frameworks applied in DNNs shows that the Message Passing Interface (MPI) is the most-used communication standard for implementing parallel strategies in DDL (BEN-NUN; HOEFLER, 2019). That is the case of Horovod, but not Tarantella, which uses GPI-2. Score-P has support for MPI, but not for GPI-2. Extrae has a branch [1] developed by the Supercomputing Center in Barcelona (BSC) with support to GPI-2/Gaspi. We installed and tested, but it fails to collect Tarantella traces. A possible reason is that both frameworks wrap the C++ code with Python. For Score-P we can get MPI information for Horovod only if configuring Score-P Binding Python[2], a Score-P implementation for Python. In this work we performed compiler-based instrumentation only for Horovod with Score-P.

Horovod instrumentation with Score-P required installing and configuring Score-P

[1] https://github.com/bsc-performance-tools/extrae/tree/GASPI
[2] https://github.com/score-p/scorep_binding_python

and scorep-binding-python, and manually installing and configuring Horovod with support to Score-P tracing. We used Score-P 6.0 from the official VI-HPS repository[3] and installed it with support to the `gcc-compiler plugin`. After, we installed the Score-P Binding Python, which recognizes the installation of Score-P. Horovod is a standalone Python library, so usually installed with pip, the package installer for Python.

We downloaded and extracted the Horovod source code to our approach and configured the compilation file to use Score-P for MPI and CUDA instead of pure MPI and CUDA compilers. Horovod uses CMake to compile the code, so a usual addition of the "scorep" command as a prefix for the compiler call does not work. We used the Score-P Compiler Wrapper [4], which replaced the application's compiler with the corresponding wrapper with Score-P. For a code wrapped in Python, only instrumenting at C++ level does not work. Score-P requires the `scorep_subsystems` variable defined in an instrumented executable, which will not add it if is not instrumented at the Python level.

Fit large amounts of data on small screens is a challenge in the performance visualization of parallel applications (SCHNORR; LEGRAND, 2013). An execution collecting all traces for events in C++ and Python for two GPUs, batch size of 64 and 150 epochs generated a trace of 3.4GB, which demands time and computational power to process and analyze using visualizations. It also correlates with the limitations presented for Horovod Timeline in Section 4.1.1.

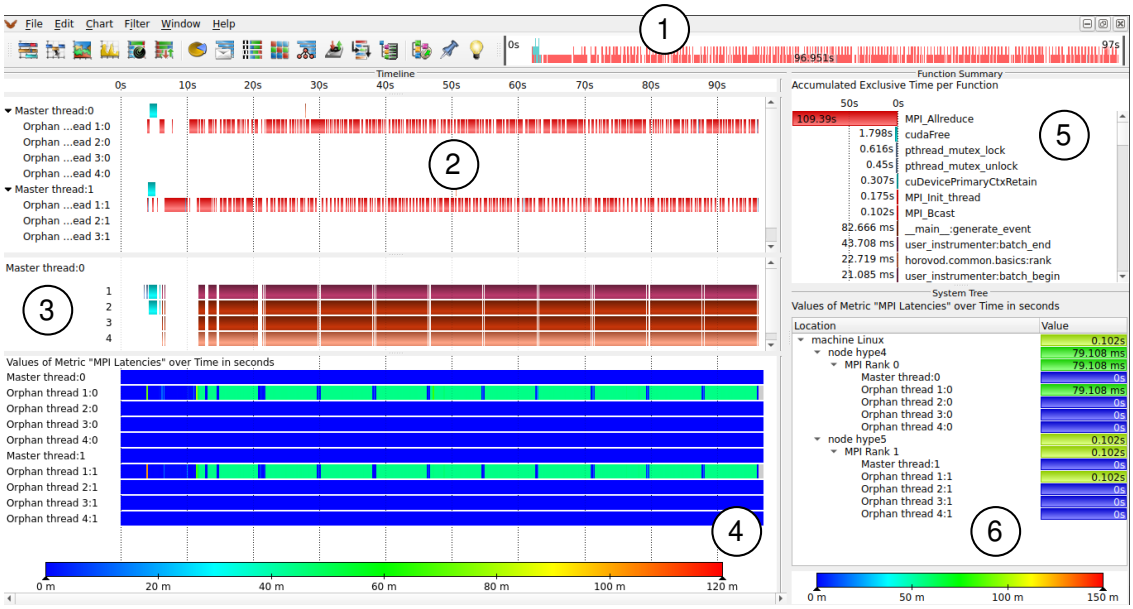
Vampir is an event trace visualization tool to OTF2 files, as outputed by Score-P, with spatial and time visualizations such as Gantt charts, bar charts, and summaries to show the trace events information. Tools as Vampir are insightful to identify patterns in parallel applications and the communication flow between processes. Visualization for Horovod with Vampir for an experiment with two GPUs, one node each, in the Hype cluster with 720 batch size in 10 epochs is presented in Figure 6.1. Panel 1 shows an overview of the application execution time, Panel 2 presents it per process, and Panel 3 presents the different levels of function calls in a bar chart for a single process where we identify the 10 epochs. Panel 4 presents the MPI latencies for each process, where we identify that each process launches a thread to control the MPI operations, with one rank having higher latency, and Panel 6 presents a system tree with accumulated latencies for each process. Panel 5 presents the accumulated execution time for each function, and we obtain that from the 97 seconds running, 109 seconds (sum of both ranks) were spent with MPI all reduce operations. If they processed similarly, as shown in Panel 2, it is

[3] https://www.vi-hps.org/cms/upload/packages/scorep/

[4] https://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/html/scorepwrapper.html

most than half of the execution time with reductions. However, for a DDL framework trace analysis, Vampir limits correlating the Score-P measurements with the tracing at the Python level. This limitation motivated our analysis using data processing to generate our own visualizations.

Figure 6.1 – Vampir visualization tool for the Score-P traces collected for Horovod.



Source: The Author

We used two nodes of the Hype cluster setting, two NVIDIA Tesla K80 each, to perform experiments with ten epochs. We converted the Score-P trace file to CSV using otf2utils [5], so we could process with R. Table 6.1 presents the total training time duration, and the total time spent with the `MPI_Allreduce` operation for each node for one execution with batch size 720, which is representative enough. The percentage of time with MPI Allreduce represents 73.62% to 82.30% of the total training time in the respective ranks. These high values come from the Horovod feature called Tensor Fusion, that performs constant asynchronous all-reduce operation during the training, as presented in Section 4.1.1. TensorFlow stores the tensors ready to be reduced in a queue, and Horovod launches a thread to combine these tensors and reduce them in one reduction operation. This process can make the training faster since it uses the time TensorFlow is computing operations in the backward propagation step to advance the reductions processing (KURTH et al., 2019). Still, overlapping the computations with communications needs to be well used since spending more than 80% of the time with reductions can indicate a performance bottleneck.

[5] https://github.com/schnorr/otf2utils

Table 6.1 – Horovod time with MPI_Allreduce for 720 batch size in 4 GPUs (2 nodes).

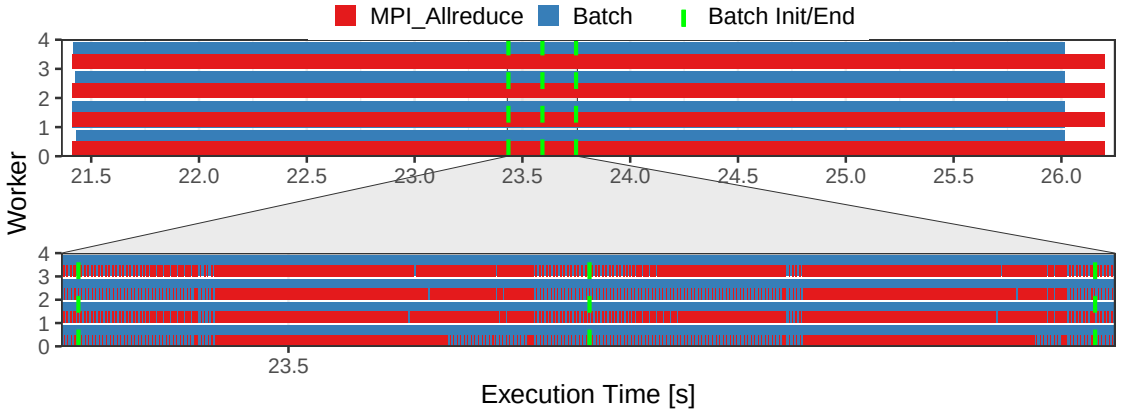
MPI Rank	Training (seconds)	MPI_Allreduce (seconds)	Percentage of Time
0	84.84	62.46	73.62%
1	85.19	70.11	82.30%
2	84.97	65.34	76.89%
3	85.12	69.63	81.80%

Source: The Author

The Tensor Fusion technique is suitable to improve the performance of large messages. Tensor Fusion disabled for a batch size 2250 increases ≈10% of the execution time, and for a batch size 720, it decreases ≈3.5% for 4 workers in 10 epochs. Fused tensors can better use the network bandwidth, but the time to start the all-reduce is delayed depending on the buffer size, impacting the total execution time (DAS et al., 2016). It explains the performance decrease for our experiment with a batch size of 720 and the default buffer of 64MB for small messages. Lenet-5 uses 54,000 images of size 32×32 for the MNIST dataset, which is small compared to other popular networks as ResNet-50 with Imagenet dataset, with more than 10 million images of size 224x224x3. Testing the Tensor Fusion feature with the last model will be considered in our future work.

Each epoch of the total training will suffer the same computations but over updated parameters. From a performance analysis perspective, we can trace a few epochs and get a representative picture of the total training execution. Figure 6.2 presents a temporal visualization of the `MPI_Allreduce` operations that took place during one epoch for training with 4 workers and batch size 1500 using Horovod. The blue color in the background represents the batches process captured with Keras callbacks. The `MPI_Allreduce` operations are executed throughout all batches with small and bigger data volume. We zoomed in a time slice to investigate the reductions inside two batches. The vertical green lines represent the time beginning and ending a batch, which occurs subsequently. We identified a pattern where `MPI_Allreduce` processes bigger messages inside each batch, starting around the middle of its execution, and that the rank 0 processes fewer reductions inside the batch. Right after the batch beginning and end, all ranks process faster reductions, making them narrow in the plot. This Figure exposes the challenge of fitting one epoch training in a visualization, even for a small dataset, and zooming into a few batches.

Figure 6.2 – MPI_Allreduce operations during one epoch of the training in red. In blue, in the background, the batches execution. The green lines represent the delimitation of a batch initialization and end. We zoom for two batches.



Source: The Author

6.2 Instrumenting Deep Learning with NVProf

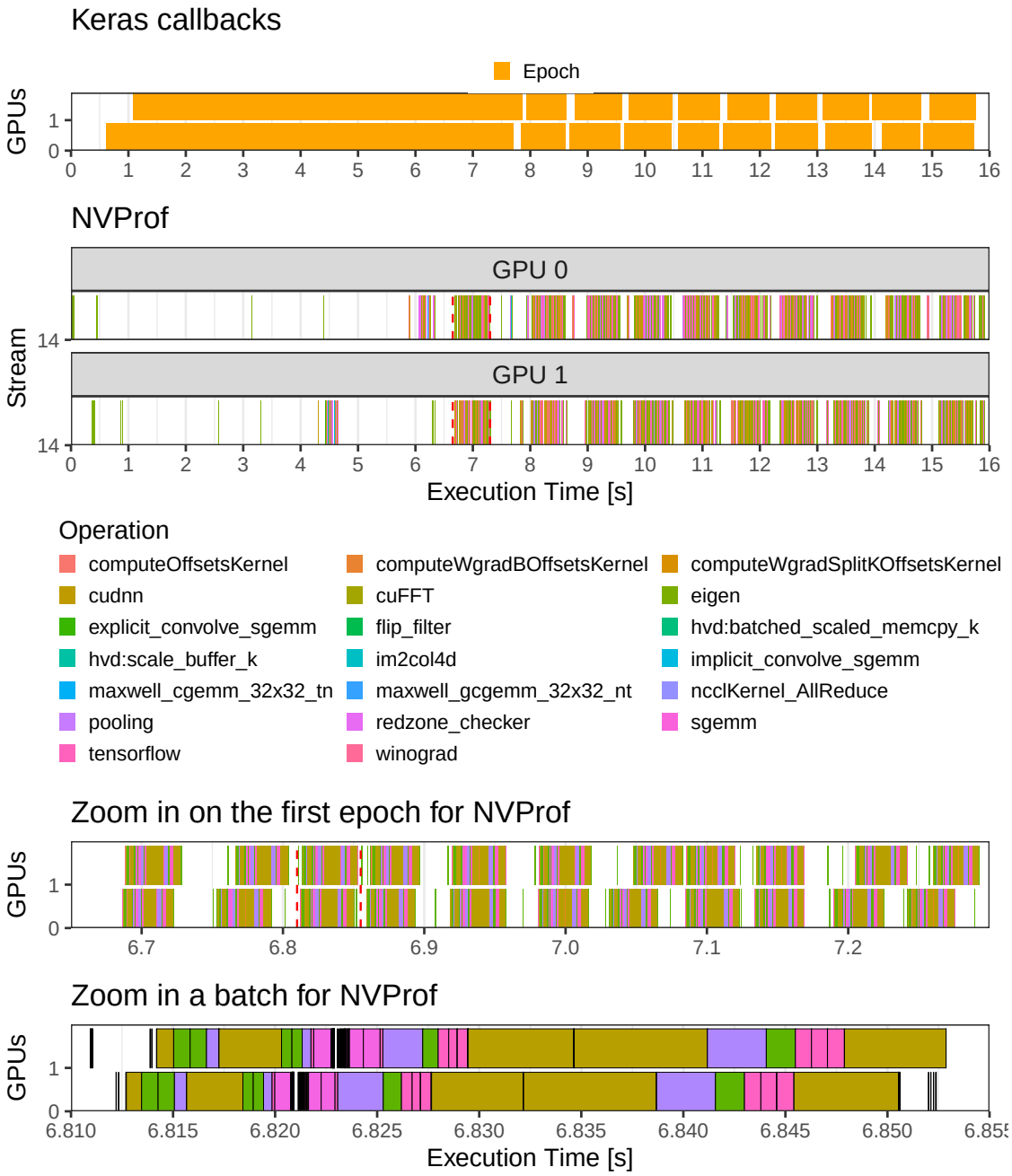
NVProf can output traces in the CSV file format or its default file format, with a .nvvp extension, which stores the CUDA operations as an SQLite file, interpreted by the NVProf visualization tool. Considering our data analysis using R, our first approach was to output CSV files. However, it shows the relative time of each event without revealing what they are relative to. It is essential to know the base timestamp of all measurement tools to correlate information as we collected for the Keras callbacks. We found it is a particularity of the traces in the CSV format, so our solution was also to output a .nvvp file and use the RSQLite package[6] to process the data with R and get the absolute timestamp.

Figures 6.3 and 6.4 present a correlation between the results obtained with Keras callbacks and NVProf. The panel on the top shows the epochs with Keras callbacks for the whole execution. Below, the NVprof panel shows the GPU Stream processing, with synchronized X-axis to the Keras callbacks. The two panels in the bottom present a zoom into the first epoch for NVProf, and into one batch of this epoch. The vertical dotted lines in red mark the zoomed parts. Looking at the two panels on the top, we identified the computing time at the devices during what is considered an epoch to Keras. The callbacks encompass more than the actual training time in the devices for all epochs and both frameworks. Works that only use tracing at the high-level programming language skip the first iterations or the first epoch of the training to avoid measuring the initialization time (LIU et al., 2018; SHI et al., 2018). The presented correlation confirmed we can identify

[6]https://db.rstudio.com/databases/sqlite/

the actual training time using the NVProf and consider the entire training time.

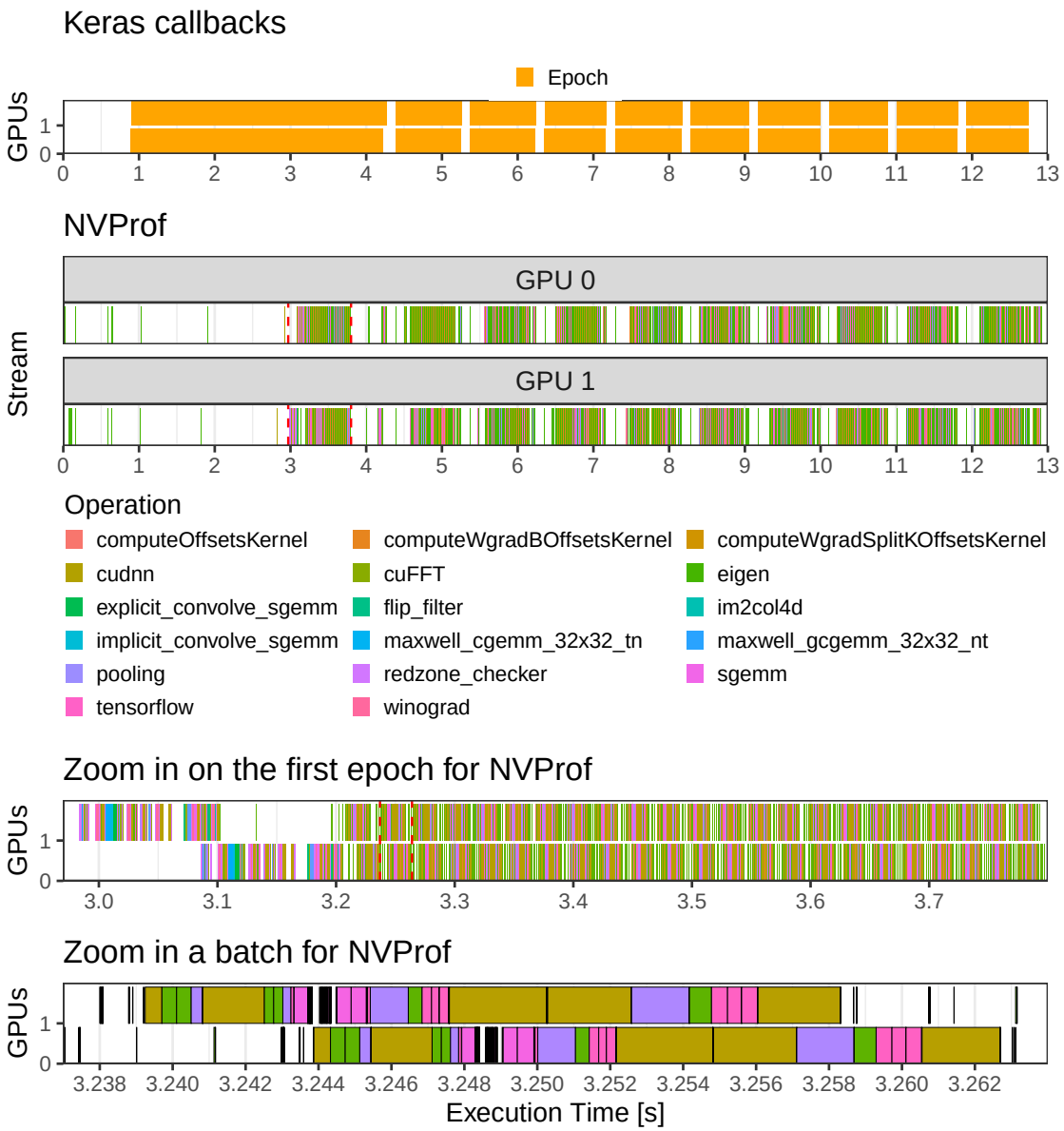
Figure 6.3 – Correlating Keras callbacks and NVProf traces to investigate the trainining for Horovod.



Source: The Author

Horovod's experiment runs 10 epochs with 12 batches in each. In Figure 6.3, in the NVProf panel, we notice the origin of the longer initialization time for Horovod. Before the first red dashed line, the time computing in GPU is 0.29 seconds, so most of the 6.65 seconds of initialization before the first epoch is performed in the host. Zooming in the first epoch, we noticed the asynchronous execution of the batches. We also identified only 11 batches because the first batch of this epoch, which does GPUs initialization, runs

Figure 6.4 – Correlating Keras callbacks and NVProf traces to investigate the trainining for Tarantella.



Source: The Author

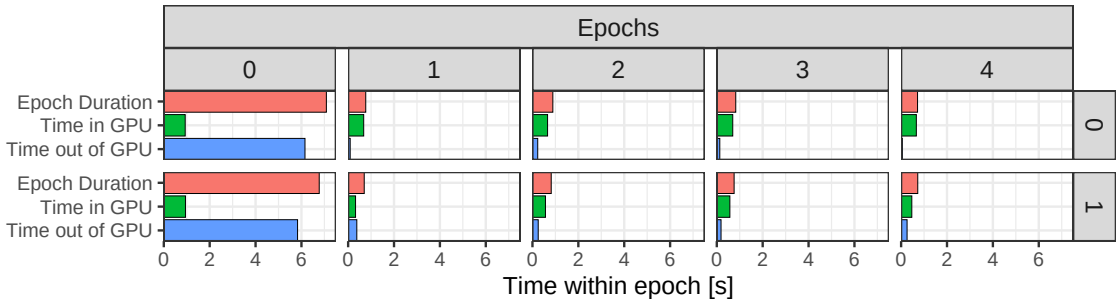
before the time interval selected. Zooming in one batch in the last panel, we can also identify the operations performed per batch. They are mostly cuDNN operations, followed by pooling operations, Eigen operations, a library to perform linear algebra operations, TensorFlow operations, and SGEMM for matrix-matrix operations in single precision.

Tarantella's experiment runs 10 epochs with 24 batches in each. The two panels on the top of Figure 6.4 showed the shorter initialization time for Tarantella compared to Horovod. From the first 3 seconds of initialization, only 1.42 milliseconds are inside the devices. Zooming into one batch, we identified the 24 batches and that the interval between the first and the second batches is smaller compared to Horovod, where the first

batch was computed between 4 and 5 seconds (Figure 6.4). We also noticed synchronicity between batches on different workers. Zooming into one batch, we identified the same operations performed inside a batch for Horovod, as expected for the training using the same model.

We used the Keras callbacks and the NVProf measurements to perform a temporal aggregation considering the time of an epoch. Figure 6.5 presents the results for Horovod and Figure 6.6 for Tarantella. The X-axis represents the total time executing an epoch, operations inside the GPU, and outside the GPU, and there are only two ranks. We summed the time with operations in GPU and got the time processing out of the devices for each epoch. The long time processing outside the GPUs for the first epoch reflects the initialization time considered by the Keras callbacks. Here we confirmed that Horovod processes more time inside and outside of the devices to work on initialization, which leaves space for investigating if initialization in Horovod could be improved. The other epochs performed very similarly for Tarantella, while Horovod spent almost all the epoch time processing inside the GPU 0 for epochs 1, 3, and 4. It is a result of the asynchronous reductions performed by the framework. Synchronicity in Tarantella needs to be explored so that it can reduce the time out of the GPU.

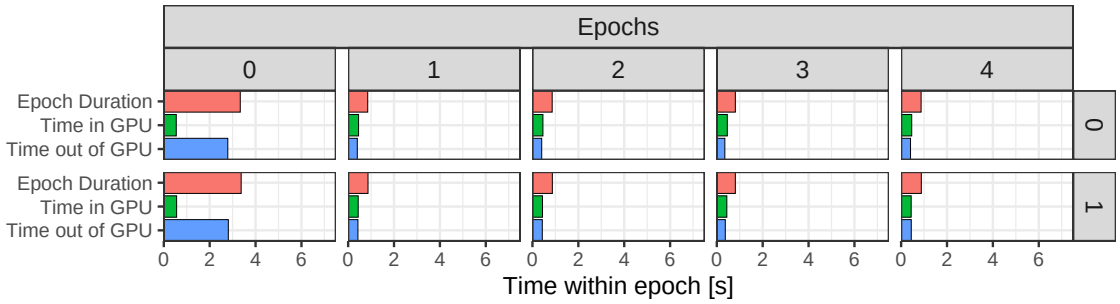
Figure 6.5 – Training time processing in GPU and outside the devices for each epoch using 2 GPUs distributed with Horovod.



Source: The Author

The presented results contribute to the performance analysis of DDL training, confirming that tracing events at the device level is complementary to the profiling at the high-level programming language only. Performing a temporal synchronization of the execution time of the Keras callbacks and the CUDA operations obtained with NVProf reveals the time performing the calculation. With a temporal aggregation, we can also measure during the training steps (epochs and batches). Moreover, tracing the DDL frameworks reveals the standard's overhead for message passing to average gradients computed in different nodes.

Figure 6.6 – Training time processing in GPU and outside the devices for each epoch using 2 GPUs distributed with Tarantella.



Source: The Author

7 CONCLUSIONS

With the increase of datasets to feed DNN and advances in DL and HPC, DDL frameworks started to be created. They facilitate the usage by people without experience in distributed systems. Still, they hide details about their algorithms and parallel strategy, wrapping the final code to be easily installed using high-level programming languages. For being very recent, there is a lack of studies comparing the performance of these frameworks. Even more, to correlate information from different performance analysis tools for different hardware and software configurations. We selected Horovod and Tarantella frameworks for DDL and state-of-the-art ML and HPC performance analysis tools to characterize the frameworks. We presented a methodology to perform temporal analysis correlating information from different tracing and profiling tools.

Horovod is one of the most-used tools for DDL, with support to more ML frameworks. It proposes improvements in its ring-allreduce algorithm that performs asynchronous communications inside and epoch to accelerate the reduction computations. Because of that, it makes higher usage of the devices. Horovod's longer time with initialization is compensated by its optimizations to process batches faster. It presented higher scaling efficiency for strong scaling than Tarantella for our experiments, with a 48% difference for the larger batch size of 2250 in the Chifflot and Chifflet clusters.

Tarantella is a comprehensive tool for users starting in DDL. It requires even fewer configuration than Horovod to distribute a sequential code and provides complete documentation. However, it limited our performance analysis with state-of-the-art tracing tools as Score-P to get the communication time with the GPI-2 pattern. Tarantella's interface limits command-line configurations. We needed to modify the application source code to test TensorBoard, perform measurements with NVProf, and communicate among nodes in the Grid'5000 tested with the OAR task manager that uses the *OARSH* patter to connect to resources, instead of the usual SSH for what Tarantella was configured.

For a small CNN as Lenet-5, the GPU version is not significant important when choosing between Tarantella and Horovod. If using a more complex network, with more layers requiring more matrix operations, we could benefit from Tensor Cores technologies present in recent devices and both NCCL and cuDNN usage by Horovod. Both frameworks stop scaling after six GPUs, since the dataset is not large enough to enable further gains. Correlating measurements with Keras Callbacks and NVProf enhance the performance analysis of Tarantella and Horovod, providing better application execution

comprehension. We could identify the time the frameworks spend with computation inside and outside the device during the epochs. Which can benefit frameworks developers on finding performance improvements spots, such as for improving the Horovod initialization and the Tarantella batches processing. We proposed a temporal aggregation for measuring the computing time in the GPUs considering the time processing an epoch.

Our methodology based on popular ML and HPC tools measurements is easy to use for researchers with some HPC experience. Since our approach is implemented at the DDL framework level, it can also be used to analyze the performance of other neural network models. It should be considered that we had access to a dedicated environment to perform our experiments. We acknowledge that the available computational resources limit DDL research. Another approach could be considered for limited access to resources, such as using cloud environments and virtual machines. However, it should consider the resources sharing and communication bottlenecks in these environments.

7.1 Future work

Performance analysis of DDL still needs investigation. New parallel strategies are being applied to frameworks, and new frameworks are being created, but no standard methodology or specific tools to analyze the performance of the frameworks considering HPC were proposed. A temporal aggregation approach correlating different tracing and profiling measurements can benefit frameworks developers to evaluate their tools. As future work, we plan to create a more automatic methodology, perhaps using a high-level programming language, so users without experience with DDL can benefit from tracing tools at the system-level, and get a diagnosis about their training using multiple nodes. An in-situ analysis considering the first epochs of a distributed training can evaluate longer training-time models without generating much data to be analyzed.

Furthermore, we aim to evaluate Horovod, Tarantella, and other DL frameworks with support to distributed computing. We will consider varied NN models, more complex CNNs, and larger datasets. In these future experiments, we will consider the usage of optimizations at the GPU level, such as using Tensor Cores and cuDNN during training.

7.2 Publications

- **Solórzano, A. L. V.**, Schnorr, L. M. "Proposta de Avaliação Prática de Frameworks para a Distribuição de Redes de Aprendizado Profundo". In Anais da XXI Escola Regional de Alto Desempenho da Região Sul (ERAD/RS-2021), pp. 129-130. Porto Alegre: SBC. doi:10.5753/eradrs.2021.14803

The following papers were also published during my research. They are not directly related to the Distributed Deep Learning field, but they are a result of other projects in High-Performance Computing I was involved with. Furthermore, the methods employed and published in these papers have been part of our main investigation in DDL.

- **Solórzano, A. L. V**.; Charão, A. S. “BlocklyPar: from sequential to parallel with block-based visual programming”. In the IEEE Frontiers in Education Conference (FIE 2021). 1-8. DOI: 10.1109/FIE49875.2021.9637261.
- **Solórzano, A. L. V.**; Schnorr, L. M.; Navaux, P. O. A. “Temporal Load Imbalance on Ondes3D Seismic Simulator for Different Multicore Architectures”. In the 2020 International Conference on High Performance Computing Simulation (HPCS’20).
- **Solórzano, A. L. V.**; Nesi, L. L.; Schnorr, L. M. “Using Visualization of Performance Data to Investigate Load Imbalance of a Geophysics Parallel Application”. In: Practice and Experience in Advanced Research Computing (PEARC). ACM, NY, USA, 518–521. DOI: 10.1145/3311790.3400844.
- **Solórzano, A. L. V.**; Nesi, L. L.; Schnorr, L. M. "Evaluation of Load Imbalance Metrics for Homogeneous and Heterogeneous Platforms." In: Concurrency and Computation: Practice and Experience (CCPE) (Submitted).

REFERENCES

ABADI, M. et al. Tensorflow: A system for large-scale machine learning. In: **Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation**. USA: USENIX Association, 2016. (OSDI'16), p. 265–283. ISBN 9781931971331.

AKIBA, T.; FUKUDA, K.; SUZUKI, S. Chainermn: Scalable distributed deep learning framework. In: **Proceedings of Workshop on ML Systems in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)**. [S.l.: s.n.], 2017.

ASPRI, M.; TSAGKATAKIS, G.; TSAKALIDES, P. Distributed training and inference of deep learning models for multi-modal land cover classification. **Remote Sensing**, v. 12, n. 17, 2020. ISSN 2072-4292. Available from Internet: <https://www.mdpi.com/2072-4292/12/17/2670>.

AWS. **Deep Learning AMI**. 2020. <https://docs.aws.amazon.com/dlami/latest/devguide/activate-horovod.html>. Accessed: 2020-11-10.

BEN-NUN, T.; HOEFLER, T. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 52, n. 4, aug 2019. ISSN 0360-0300. Available from Internet: <https://doi.org/10.1145/3320060>.

BENGIO, Y. Practical recommendations for gradient-based training of deep architectures. In: MONTAVON, G.; ORR, G. B.; MÜLLER, K.-R. (Ed.). **Neural Networks: Tricks of the Trade: Second Edition**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 437–478. ISBN 978-3-642-35289-8. Available from Internet: <https://doi.org/10.1007/978-3-642-35289-8_26>.

BRAIN, G. **TensorBoard: TensorFlow's visualization toolkit**. 2015. <https://www.tensorflow.org/tensorboard>. Accessed: 2021-11-06.

Cappello, F. et al. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In: **The 6th IEEE/ACM International Workshop on Grid Computing, 2005.** Washington, US: IEEE, 2005. p. 8.

CCHPC. **Tarantella: Distributed Deep Learning Framework**. [S.l.]: Competence Center for High Performance Computing, 2020. <https://github.com/cc-hpc-itwm/tarantella>. Accessed: 2020-11-14.

Chen, T. et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. **arXiv preprint arXiv:1512.01274**, 2015.

CUNHA, R. L. de F. et al. An argument in favor of strong scaling for deep neural networks with small datasets. **CoRR**, abs/1807.09161, 2018. Available from Internet: <http://arxiv.org/abs/1807.09161>.

DAI, J. J. et al. Bigdl: A distributed deep learning framework for big data. In: **Proceedings of the ACM Symposium on Cloud Computing**. New York, NY, USA: Association for Computing Machinery, 2019. (SoCC '19), p. 50–60. ISBN 9781450369732. Available from Internet: <https://doi.org/10.1145/3357223.3362707>.

DAS, D. et al. Distributed deep learning using synchronous stochastic gradient descent. **ArXiv**, abs/1602.06709, 2016.

DEAN, J. et al. Large scale distributed deep networks. In: **Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1**. Red Hook, NY, USA: Curran Associates Inc., 2012. (NIPS'12), p. 1223–1231.

DENG, J. et al. Imagenet: A large-scale hierarchical image database. In: **2009 IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2009. p. 248–255.

DENG, L.; YU, D. Deep learning: Methods and applications. **Found. Trends Signal Process.**, Now Publishers Inc., Hanover, MA, USA, v. 7, n. 3–4, p. 197–387, jun. 2014. ISSN 1932-8346. Available from Internet: <https://doi.org/10.1561/2000000039>.

DETTMERS, T. 8-bit approximations for parallelism in deep learning. In: **4th International Conference on Learning Representations, Conference Track Proceedings**. [s.n.], 2016. Available from Internet: <http://arxiv.org/abs/1511.04561>.

EDUCATION, I. C. **Gradient Descent**. 2020. <https://www.ibm.com/cloud/learn/gradient-descent>. Accessed: 2021-06-22.

ESSEN, B. V. et al. LBANN: Livermore big artificial neural network hpc toolkit. In: **Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments**. New York, NY, USA: Association for Computing Machinery, 2015. (MLHPC '15). ISBN 9781450340069.

FOUNDATION, P. S. **Python Profilers**. 2016. <https://docs.python.org/2/library/profile.html>. Accessed: 2021-06-20.

GAMBLIN, T. et al. The spack package manager: Bringing order to hpc software chaos. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: Association for Computing Machinery, 2015. (SC '15). ISBN 9781450337236. Available from Internet: <https://doi.org/10.1145/2807591.2807623>.

GIBIANSKY, A. **Bringing HPC Techniques to Deep Learning**. 2017. <https://web.archive.org/web/20180128132031/http://research.baidu.com/bringing-hpc-techniques-deep-learning>. Accessed: 2020-10-01.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. Cambridge, MA, USA: MIT Press, 2016. <http://www.deeplearningbook.org>.

GOYAL, P. et al. **Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour**. 2018.

GRAUPE, D. **Principles of artificial neural networks**. [S.l.]: World Scientific, 2013.

GRÜNEWALD, D.; SIMMENDINGER, C. The gaspi api specification and its implementation gpi 2.0. In: **In 7th International Conference on PGAS Programming Models**. [S.l.: s.n.], 2013. v. 243.

HASHEMINEZHAD, B. et al. Towards a scalable and distributed infrastructure for deep learning applications. **2020 IEEE/ACM Fourth Workshop on Deep Learning on Supercomputers (DLS)**, IEEE, Nov 2020. Available from Internet: <http://dx.doi.org/10.1109/DLS51937.2020.00008>.

HE, K. et al. Deep residual learning for image recognition. In: **2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)**. [S.l.: s.n.], 2016. p. 770–778.

HOROVOD. **Tensor Fusion**. 2019. <https://horovod.readthedocs.io/en/stable/tensor-fusion_include.html>. Accessed: 2021-10-29.

JAIN, R. **The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling**. [S.l.]: Wiley New York, 1991.

JIA, X. et al. **Whale: Scaling Deep Learning Model Training to the Trillions**. 2021.

JORDÀ, M.; VALERO-LARA, P.; PEÑA, A. J. Performance evaluation of cudnn convolution algorithms on nvidia volta gpus. **IEEE Access**, v. 7, p. 70461–70473, 2019.

KARLIK, B.; VEHBI, A. Performance analysis of various activation functions in generalized mlp architectures of neural networks. **International Journal of Artificial Intelligence and Expert Systems (IJAE)**, v. 1, n. 4, p. 111–122, 2011.

KERAS. **Keras**. 2015. <https://github.com/keras-team/keras>. Accessed: 2020-11-26.

KNÜPFER, A. et al. Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir. In: BRUNST, H. et al. (Ed.). **Tools for High Performance Computing 2011**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 79–91. ISBN 978-3-642-31476-6.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 60, n. 6, p. 84–90, may 2017. ISSN 0001-0782.

KURTH, T. et al. Tensorflow at scale: Performance and productivity analysis of distributed training with horovod, mlsl, and cray pe ml. **Concurrency and Computation: Practice and Experience**, v. 31, n. 16, p. e4989, 2019. E4989 cpe.4989.

LAANAIT, N. et al. **Exascale Deep Learning for Scientific Inverse Problems**. 2019.

LAINE, A. Neural networks. In: ____. **Encyclopedia of Computer Science**. GBR: John Wiley and Sons Ltd., 2003. p. 1233–1239. ISBN 0470864125.

LE, T. D. et al. Involving cpus into multi-gpu deep learning. In: **Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering**. New York, NY, USA: Association for Computing Machinery, 2018. (ICPE '18), p. 56–67. ISBN 9781450350952. Available from Internet: <https://doi.org/10.1145/3184407.3184424>.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **Nature**, v. 521, 2015.

LECUN, Y. et al. Backpropagation applied to handwritten zip code recognition. **Neural Computation**, v. 1, n. 4, p. 541–551, 1989.

LECUN, Y. et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, v. 86, n. 11, p. 2278–2324, 1998.

LECUN, Y.; CORTES, C. MNIST handwritten digit database. 2010. Available from Internet: <http://yann.lecun.com/exdb/mnist/>.

LIU, J. et al. Usability study of distributed deep learning frameworks for convolutional neural networks. In: . [S.l.: s.n.], 2018.

MA HEAND MAO, F.; TAYLOR, G. W. Theano-mpi: A theano-based distributed training framework. In: **Euro-Par 2016: Parallel Processing Workshops**. Cham: Springer International Publishing, 2017. p. 800–813. ISBN 978-3-319-58943-5.

MAHON, S. et al. Performance analysis of distributed and scalable deep learning. In: **2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)**. [S.l.: s.n.], 2020. p. 760–766.

MASTERS, D.; LUSCHI, C. **Revisiting Small Batch Training for Deep Neural Networks**. 2018.

Microsoft. **Deep learning with Horovod for distributed training**. 2020. <https://docs.microsoft.com/en-us/learn/modules/deep-learning-with-horovod-distributed-training/>. Accessed: 2020-11-10.

ML4A. **Looking inside neural nets**. 2018. <https://ml4a.github.io/ml4a/looking_inside_neural_nets/>. Accessed: 2021-06-22.

NVIDIA. **NVIDIA System Management Interface**. 2011. <https://developer.download.nvidia.com/compute/DCGM/docs/NVSMI-367.38.pdf>. Accessed: 2020-05-10.

NVIDIA. **Apex**. [S.l.]: NVIDIA, 2020. <https://github.com/nvidia/apex>. Accessed: 2020-11-14.

NVIDIA. **Deep Learning at Scale with Horovod**. 2020. <https://courses.nvidia.com/courses/course-v1:DLI+L-FX-23+V1/about>. Accessed: 2020-05-10.

OH, K.-S.; JUNG, K. Gpu implementation of neural networks. **Pattern Recognition**, v. 37, n. 6, p. 1311 – 1314, 2004. ISSN 0031-3203. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0031320304000524>.

OOI, B. C. et al. Singa: A distributed deep learning platform. In: **Proceedings of the 23rd ACM International Conference on Multimedia**. New York, NY, USA: Association for Computing Machinery, 2015. (MM '15), p. 685–688. ISBN 9781450334594.

OR, A.; ZHANG, H.; FREEDMAN, M. J. Resource elasticity in distributed deep learning. In: DHILLON, I. S.; PAPAILIOPOULOS, D. S.; SZE, V. (Ed.). **Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020**. mlsys.org, 2020. Available from Internet: <https://proceedings.mlsys.org/book/314.pdf>.

PASZKE, A. et al. Pytorch: An imperative style, high-performance deep learning library. In: **Thirtd-three Annual Conference on Neural Information Processing Systems (NeurIPS)**. [S.l.: s.n.], 2019.

RAVIKUMAR, A.; S., H. A comprehensive review and evaluation of distributed deep learning on cloud environments. In: **Journal of Critical Reviews**. [S.l.]: Journal of Critical Reviews, 2020. p. 9519–9538.

RUMELHART, D.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **Nature**, v. 323, p. 533–536, 1986.

SATHYA, R.; ABRAHAM, A. et al. Comparison of supervised and unsupervised learning algorithms for pattern classification. **International Journal of Advanced Research in Artificial Intelligence**, Citeseer, v. 2, n. 2, p. 34–38, 2013.

SCHNORR, L. M.; LEGRAND, A. Visualizing more performance data than what fits on your screen. In: **Tools for High Performance Computing 2012**. [S.l.]: Springer, 2013. p. 149–162.

SEIDE, F.; AGARWAL, A. Cntk: Microsoft's open-source deep-learning toolkit. In: **Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. New York, NY, USA: Association for Computing Machinery, 2016. (KDD '16), p. 2135. ISBN 9781450342322. Available from Internet: <https://doi.org/10.1145/2939672.2945397>.

SERGEEV, A. **Uber's Distributed Deep Learning Journey**. 2017. <https://www.hpcadvisorycouncil.com/events/2018/stanford-workshop/pdf/DayTwo_Wed21Feb2018/ASergeev_UberEng_DistributedDeepLearning_KeynoteDayTwo_21Feb2018.pdf>. Accessed: 2021-07-24.

SERGEEV, A.; BALSO, M. D. Horovod: fast and easy distributed deep learning in TensorFlow. **arXiv preprint arXiv:1802.05799**, 2018.

SHI, S.; CHU, X. Performance modeling and evaluation of distributed deep learning frameworks on gpus. **CoRR**, abs/1711.05979, 2017. Available from Internet: <http://arxiv.org/abs/1711.05979>.

SHI, S. et al. Performance modeling and evaluation of distributed deep learning frameworks on gpus. In: **2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)**. [S.l.: s.n.], 2018. p. 949–957.

SOFTWARE, C. **Anaconda Software Distribution**. 2016. <https://anaconda.com/>. Accessed: 2020-05-10.

SONG, J. et al. Cluster serving: Distributed model inference using big data streaming in analytics zoo. In: . [S.l.]: USENIX Association, 2020.

SZEGEDY, C. et al. Going deeper with convolutions. In: **2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)**. [S.l.: s.n.], 2015. p. 1–9.

SZEGEDY, C.; TOSHEV, A.; ERHAN, D. Deep neural networks for object detection. In: BURGES, C. J. C. et al. (Ed.). **Advances in Neural Information Processing Systems**. [S.l.]: Curran Associates, Inc., 2013. v. 26, p. 2553–2561.

TESSER, R. K.; MARQUES, A.; BORIN, E. Selecting efficient vm types to train deep learning models on amazon sagemaker. In: **2021 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. Los Alamitos, CA, USA: IEEE Computer Society, 2021.

VIVIANI, P. et al. Deep learning at scale. In: **2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)**. [S.l.: s.n.], 2019. p. 124–131.

VOGELSANG, D. C.; ERICKSON, B. J. Magician's corner: 6. tensorflow and tensorboard. **Radiology: Artificial Intelligence**, v. 2, n. 3, p. e200012, 2020. PMID: 33937828. Available from Internet: <https://doi.org/10.1148/ryai.2020200012>.

WU, J. et al. Performance analysis of graph neural network frameworks. In: **2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.: s.n.], 2021. p. 118–127.

WU, X. et al. Performance, power, and scalability analysis of the horovod implementation of the candle nt3 benchmark on the cray xc40 theta. In: **Proceedings of SC18 Workshop on Python for High-Performance and Scientific Computing**. [S.l.: s.n.], 2018.

ZAGORUYKO, S.; KOMODAKIS, N. Wide residual networks. In: WILSON, E. R. H. R. C.; SMITH, W. A. P. (Ed.). **Proceedings of the British Machine Vision Conference (BMVC)**. BMVA Press, 2016. p. 87.1–87.12. ISBN 1-901725-59-6. Available from Internet: <https://dx.doi.org/10.5244/C.30.87>.

ZHANG, M.-L.; ZHOU, Z.-H. Multilabel neural networks with applications to functional genomics and text categorization. **IEEE Transactions on Knowledge and Data Engineering**, v. 18, n. 10, p. 1338–1351, 2006.