UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Convergence of Deep Learning and High Performance Computing: Challenges and Solutions

Doctoral Thesis

Albert Njoroge Kahira

Supervisors: Dr. Leonardo Bautista Gomez
Dr. Rosa M Badia

Dissertation submitted to the Department of Computer Architectures (DAC) in partial fulfilment of the requirements for degree of Doctor of Philosophy in Computer Architectures

Barcelona (Spain)
July 2021

Obra Social "la Caixa"

European Commission | Horizon 2020 European Union funding for Research & Innovation

The project that gave rise to these results received the support of a fellowship from the "la Caixa" Foundation (ID **100010434**). The fellowship code is **LCF/BQ/DI17/11620059**.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. **713673**.

To my mum and dad
For being the most passionate and supportive people throughout my education

&

To all scientists
We stand on your shoulders and see the future

Acknowledgments

This thesis is the result of years of hard work that would not have been possible without the support of numerous people. In this inconclusive list, I would like to mention a few of those people.

First, I would like to express my deep gratitude to my supervisors, Dr Leonardo Bautista and Dr Rosa Badia. Their comments, feedback and encouragement, made this possible. I am grateful for the opportunities they created to develop myself as a researcher and for the many talks we had during lunch hours and in conferences. And for the times I was convinced about quitting, they taught me never to give up.

I would also like to thank my pre-defence and defence committee; Prof. Jordi Torres, Dr Marc Casas, Dr Antonio Pena, Dr Naoya Maruyama and Dr Prasanna Balaprakash. I truly appreciate the time they took to read my thesis and the feedback that helped shape the final document.

I would like to thank Dr Mohamed Wahib, Dr Truong Thao Nguyen and everyone at Matsouka Lab for the three months I spent in Japan. They were great hosts that guided me through what ended up becoming the core part of my thesis. I would also like to thank my undergraduate and graduate professors, especially Dr Gulay Yalcin, who encouraged me to pursue a PhD and introduced me to research. I am forever indebted to her.

My sincere gratitude to the Workflows and Distributed Computing group members at Barcelona Supercomputing Center(BSC). Especially Dr Francisco Javier Conejero, who helped me fix all the problems I had with PyCOMPSs and taught me a lot about distributed computing. I would like to thank everyone at Barcelona Supercomputing Center who helped me in one way or another. From my relocation to Barcelona and the many administrative procedures that followed.

I am grateful for the friends and colleagues we spent years together in the same office at BSC. Especially members of the Computer Architecture For Parallel Paradigms Group. Julian Pavon, Kai Keller, Dr Behsad Selami, Sawsane Ochtal, Saber Nabavi, Elvis Rojas, Mikel Cortes(who pro-read the Spanish summary) and the group leaders, Dr Osman Unsal and Dr Adrian Christal. Though I belonged to a different group, I felt at home in the Nexus 1 302 office and later in the Til.lers office.

I would like to thank the la Caixa Foundation for giving me the incredible opportunity to do a PhD in an excellent facility. The staff at la Caixa foundation were extremely helpful in both the transition and the years that followed through numerous training. I would also like to thank the over 70 la Caixa fellows who started this PhD together, especially those in Barcelona. I am genuinely humbled to call myself a la Caixa fellow and be part of the la Caixa family.

I am privileged to have incredible people in my life who have been really supportive and walked with me through this journey: Dr Milena Sulzbach, Nikolaos Triantafyllou, Elena Marban, Paul Njoroge, Joel Moriasi, Aisha Navizale and especially Polet Njeri, who put up with me in my worst times.

Finally, I would like to thank my family, who, despite having no idea what I do, supported me nevertheless and encouraged me to finish. I am incredibly grateful to my late father, who was my pillar in my education, encouraging me all through and helping me discover my true potential. Deepest gratitude to my mother for the support and listening to me on the many times I told her how stressed I was.

Abstract

Deep Learning has achieved outstanding results in many fields and led to groundbreaking discoveries. With the steady increase in datasets and model sizes, there has been a recent surge in Machine Learning applications in High-Performance Computing (HPC) to speed up training. Deep Neural Network (DNN) frameworks use distributed training to enable faster time to convergence and alleviate memory capacity limitations when training large models or using high dimension inputs. However, training DNN in HPC infrastructures presents a unique set of challenges: scalability, IO contention, network congestion and fault tolerance. Solving these problems is particularly challenging and unique due to DL applications' nature and the history of adaptation of DL in HPC. This thesis addresses scalability and resilience challenges by looking at different parts of the Machine Learning Workflow.

We first address hyper-parameters optimisation (HPO), which is one of the most time consuming and resource-intensive parts of a Machine Learning Workflow. We present a HPO scheme built on top of PyCOMPSs, a programming model and runtime which aims to ease the development of parallel applications for distributed infrastructures. We show that PyCOMPSs is a robust framework that can accelerate the process of Hyperparameter Optimisation across multiple devices and computing units. We perform a detailed performance analysis showing different configurations to demonstrate the effectiveness of our approach.

We then analyse the compute, communication, and memory requirements of DNNs to understand the trade-offs of different parallelism approaches on performance and scalability. We leverage our model-driven analysis to be the basis for an oracle utility that can help detect the limitations and bottlenecks of different parallelism approaches at scale.

While significant effort has been put to facilitate distributed training by DL frameworks, fault tolerance has been largely ignored. We examine the checkpointing implementation of popular DL platforms. We evaluate the computational cost of checkpointing, file formats and file sizes, the impact of scale, and deterministic checkpointing. We provide discussion points that can aid users in selecting a fault-tolerant framework to use in HPC. We also provide take-away points that framework developers can use to facilitate better checkpointing of DL workloads in HPC.

Keywords: *Machine Learning*, *High Performance Computing*, *Parallel and Distributed Computing*.

Resumen

El Deep Learning ha logrado resultados sobresalientes en muchas aplicaciones y ha dado lugar a descubrimientos revolucionarios. Con el aumento constante del tamaño de las colecciones de datos y de los modelos, ha habido un reciente desarrollo de aplicaciones de Machine Learning en computación de alto rendimiento (HPC) que se enfocan en reducir el tiempo de entrenamiento de los modelos diseñados. Las librerías de Deep Neural Networks (DNN) utilizan el entrenamiento distribuido para reducir el tiempo de convergencia y aliviar las limitaciones de capacidad de memoria al entrenar modelos grandes o al utilizar entradas de gran dimensión. Sin embargo, capacitar a DNN en infraestructuras de HPC presenta una serie única de desafíos: escalabilidad, contención de E / S, congestión de la red y tolerancia a fallas. Resolver estos problemas es particularmente desafiante y único debido a la naturaleza de las aplicaciones DL y la historia de adaptación de DL en HPC. Esta tesis aborda los desafíos de escalabilidad y resiliencia al analizar el flujo de trabajo completo del Machine Learning.

Primero abordamos la optimización de hiper-parámetros (HPO), que es una de las partes del flujo de trabajo de Machine Learning que consume más tiempo y recursos. Presentamos un esquema HPO construido sobre PyCOMPSs, un modelo de programación que tiene como objetivo facilitar el desarrollo de aplicaciones paralelas para infraestructuras distribuidas. Demostramos que PyCOMPSs es un marco robusto que puede acelerar el proceso de optimización de hiper-parámetros en múltiples dispositivos y unidades informáticas. Realizamos un detallado análisis de rendimiento que muestra diferentes configuraciones para demostrar la efectividad de nuestro enfoque.

Luego, analizamos los requisitos de computación, comunicación y memoria de las DNN para comprender las compensaciones de los diferentes enfoques de paralelismo en el rendimiento y la escalabilidad. Aprovechamos nuestro análisis basado en modelos como base de una utilidad de Oracle que puede ayudar a detectar las limitaciones y los cuellos de botella de diferentes enfoques de paralelismo a escala.

Si bien se ha realizado un esfuerzo significativo para facilitar el entrenamiento distribuido por los marcos de DL, la tolerancia a fallas se ha ignorado en gran medida. Examinamos la implementación de puntos de control de plataformas DL populares. Evaluamos el costo computacional de los puntos de control, los formatos y tamaños de los archivos, el impacto de la escala y los puntos de control deterministas. Proporcionamos puntos de discusión que pueden ayudar a los usuarios a seleccionar un marco tolerante a fallas para usar en HPC. También proporcionamos puntos de referencia que los desarrolladores de marcos pueden utilizar para facilitar un mejor control de las cargas de trabajo de DL en HPC.

Palabras clave: *Machine Learning*, *High Performance Computing*, *Parallel and Distributed Computing*.

Table of Contents

List of Figures

Acronyms

AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
DNN	Deep Neural Networks
GPU	Graphics Processing Unit
TPU	Tensor Processing Unit
ANN	Artificial Neural Network
HPC	High Performance Computing
IoT	Internet of Things

Chapter 1

Introduction

We stand on the brink of a technological revolution that will fundamentally alter the way we live, work and relate to one another

Klaus Schwab

Machine Learning is a set of methods that can automatically detect patterns in data and then use the uncovered patterns to predict future data or other kinds of decision-making [1]. In the last two decades, Machine Learning (ML) has become a major driving force for Artificial Intelligence (AI) and has gained popularity across numerous domains, which has, in turn, led to countless ML applications [2] [3]. ML now powers many industries such as self-driving cars, recommendation systems, computer vision and natural language processing. As a result, ML has changed and will continue to the way we live, communicate, shop, travel, healthcare and even politics.Deep Learning (DL) is a subset of ML based Artificial Neural Network (ANN). DL has become a method of choice, and Deep Neural Networks (DNN) has achieved superior accuracy than traditional methods in areas such as computer vision.

The idea of machines that can learn and therefore acquire some level of intelligence has been around since the 1950s when Alan Turing first proposed the Turing Test in his paper, Computing Machinery and Intelligence [4]. Over the

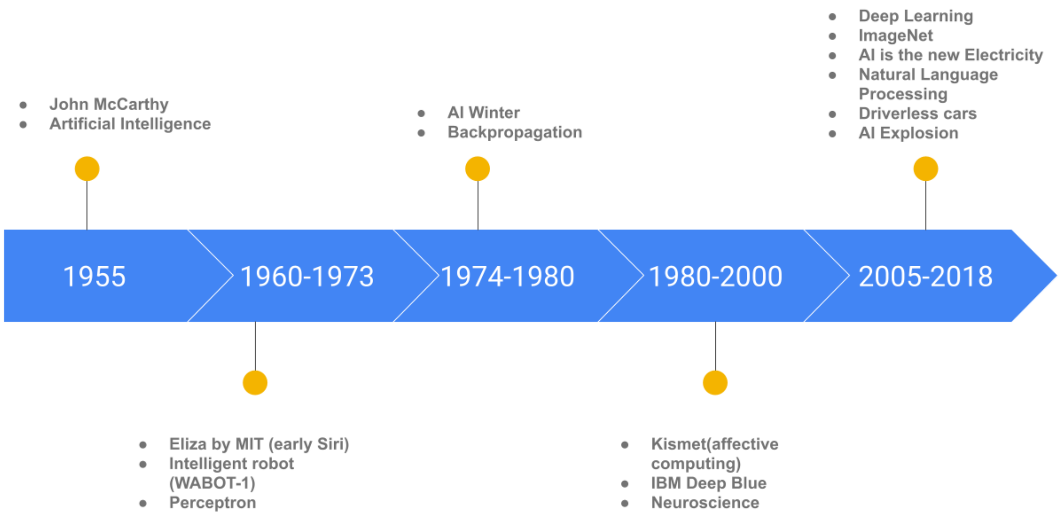


Figure 1.1: History of Artificial Intelligence

years, scientists and engineers have worked relentlessly to achieve some form of intelligence, as Turing described. Early attempts, such as Eliza by MIT Artificial Intelligence Laboratory, showed that intelligence such as natural language understanding was complex and could not be scripted. However, later inventions would be based on those founding principles, and ML would emerge as the critical driver of AI. Figure 1.1 shows a brief history of Artificial Intelligence, highlighting key achievements and inventions since the 1950s. Even though the progress has been continuous, key breakthroughs have happened in the last two decades. This is due to several factors that we address in details below.

First is **groundbreaking learning algorithms** starting with the **Perceptron**. A Perceptron is a mathematical model of a biological neuron and the building block of ANN, first introduced in 1957 by Frank Rosenblatt. Despite much enthusiasm at its inception, the perceptron did not achieve much and faced criticism because it could not learn complex patterns. It led to a period of little or no research in ANN, popularly called the AI winter. It later emerged that most of the limitations of the perceptron could be overcome by combine several perceptrons. This was called Multi-Layer Perceptron (MLP) and could solve more complex problems, including the XOR function. Combined with **Back-propagation** training algorithm by Rumelhart et al. 1986 [5] it led to renewed interests in Artificial Neural Networks. Further

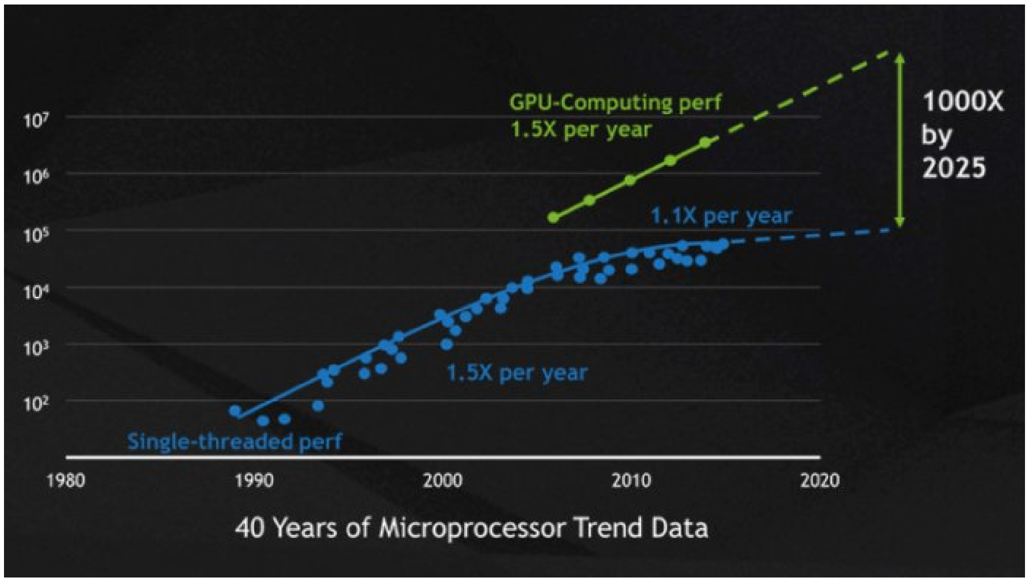


Figure 1.2: 40 Year Microprocessor Trend
Source: NVIDIA

advancements such as better activation functions, more complex optimisation algorithms and combining more ANN layers eventually led to Deep Learning which is today responsibly for the remarkable results across multiple domains.

Second is **developments in hardware**. In 1965, Gordon Moore predicted that the number of transistors in Integrated Circuits (IC) would double each year. He would later revise the prediction to 2 years, and this prediction has held to date. However, challenges such as overheating to increased density led Processors designers to develop multi-core chips. This continuous increase in computing power meant that early developments in Neural Networks could now be implemented. Despite these developments and increased computing power, it was impossible to train large models or use the vast amounts of data that were becoming increasingly available. In early 2009, [6] showed that GPUs, traditionally developed for gaming and graphics applications, could significantly scale learning algorithms and surpass multi CPUs. This could be done by leveraging the fact that GPUs are massively parallel and ideal for matrix multiplication. These early work would later lead to the development of more advanced Graphics Processing Unit (GPU) and most recently Tensor Processing Unit (TPU) that can significantly bring down the training time of advanced learning algorithms. Figure 1.2 created by Nvidia shows hardware trends in the last 40 years.

Third is **increased amounts of data**. Since the mainstream adaptation of Information Systems and widespread use of the internet, the amount of data generated every day has increased tremendously. Today, it is estimated that we generate about 2.5 Quintillion bytes of data every day. For instance, more than 300 Million Photos are uploaded on Facebook every day, and millions of emails are sent daily. The amount of data generated will continue to increase with technologies such as Internet of Things (IoT) and driver-less cars. These massive amounts of data are at the core of improved accuracy that surpasses humans in complex tasks such as computer vision that we witness today. In the paper, The unreasonable effectiveness of data [7], Halevy et al. argued that even with reasonably average algorithms, more data significantly improved the performance of a Machine Learning Model. With this data, ML algorithms can identify complex relationships in data and generate meaningful insights that would otherwise be impossible or not computationally feasible using traditional statistical approaches.

1.1 Motivation

DNNs today achieve outstanding results in a wide range of applications, including image recognition, video analysis, natural language processing [8], understanding climate [9], and drug discovery [10], among many others. Mathuriya et al. [11] with the Cosmoflow project showed the usefulness of Deep Learning at scale to measure cosmological parameters from density fields. Deep Learning is therefore a potential tool to determine the physical model that describes our universe. Most recently, DeepMind used DL to create AlphaFold [12], a program that can solve one of biology's most significant challenges, accurately predicting protein structures from their amino-acid sequence. This discovery can revolutionise medicine and drug development by understanding the nature of diseases and molecules that make up the human body.

Recent trends in DL include huge DNN models and very large training datasets [13, 14, 15]. This is in the quest to increase solution accuracy and solve more challenging problems. In addition, applying Deep Learning (DL) in new domains, such as health care and scientific simulations, introduces larger data samples and more complex DNN models [16]. When trained over a large amount of data, these models' sheer size and complexity make them harder to converge in a reasonable amount of time. Furthermore, despite significant breakthroughs in GPU memory,

memory size remains relatively small with respect to model sizes. For example, Huang et al. [15] reported that GPUs memory has only increased from 12 GB in 2014 to 40 GB in 2020 while the DL models increased approximately 36x in terms of the number of parameters. Due to these reasons, parallelism in DL is inevitable.

Parallalelism in Deep Learning has generally focused on 4 aspects; Instruction Level Parallelism(ILP) and other low level parallelism(e.g. parallelel convolutions), Data Parallelism, Model Parallelism, and most recently Hyper Parameter Optimisation(HPO). Low level parallelism techniques came into prominence after Alex et al [17] demonstrated that Deep Learning models could be trained faster and efficiently using GPU implemenation of the convolution operation. Prior to that, GPUs were traditionally used for graphics intensive applications such as gamming applications. To leverage multiple GPUs and reduce training times, Raina et al [6] developed general principles for massively parallelizing unsupervised learning tasks using GPUs. Today, multi GPU training generally is generally done with either Data Parallelism or Model Parallelism. Due to the large number of hyper-parameters required in DNNs, HPO process has also been parallelised to search through the huge parameter space. While early efforts in parallellism focused on intra-node parallelism, this soon became insufficient due to the limited number of computing devices (e.g. GPUs) that can be packed in a single node. As a result, DL has now turned to High Performance Computing (HPC)

Large-scale training on HPC systems or clusters of GPUs is now becoming increasingly common to achieve faster training time for larger models and datasets [13] and alleviate memory constraints. Training ML models in these systems cuts months or even weeks of training to a few hours or even minutes and facilitates faster prototyping and research in ML. Figure 1.3 shows the relative speedups, the accuracy, and the number of nodes used in notable milestones of training DNNs, specifically for 90-epoch training of ResNet50 with ImageNet. It is evident and clear that HPC is now the key driver of DL. In fact, in the last few years, we have seen HPC systems such as AI Bridging Cloud Infrastructure (ABCI) built explicitly for DL workloads. HPC systems' power will continue to increase, with an eventual breakthrough to an exascale system expected in 2023 as shown in figure 2.5. However, achieving maximum performance and capitalising on the power of HPC is not obvious.In the following section, we discuss the challenges in the convergence of DL and HPC.

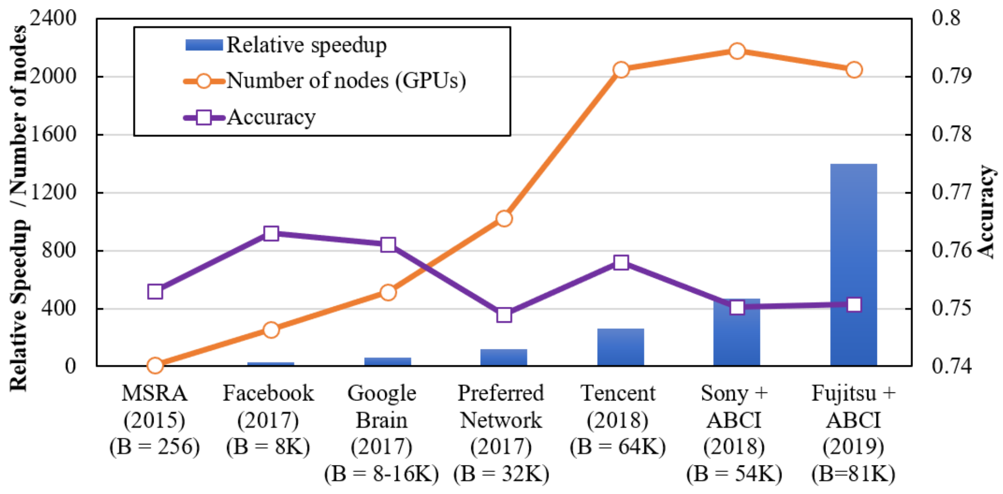


Figure 1.3: Relative speedup, accuracy and number of nodes(GPUs) in recent years

1.2 Challenges

The first challenge is **Scaling**. Scaling Machine Learning in current HPC systems and future exascale systems is a complex problem due to DL applications' unique nature, the history of adaptation of DL in HPC and typical HPC challenges such as communication, IO contention, network congestion and fault tolerance. For instance, data parallelism, which involves duplicating the model into multiple nodes/GPUs with each model processes a mini-batch and an Allreduce collective operation for weight update, is well established and almost the default method for parallelism in DL. However, despite promising results from data parallelism in scaling training across multiple nodes, as the number of computing nodes increases, e.g., up to 2048 GPUs for Resnet-50, the communication becomes a bottleneck due to the Allreduce collective operation. Furthermore, data parallelism is impractical for large models since the model has to be replicated in every device. On the other hand, getting good performance with model parallelism given multiple computing devices is non-trial and non-obvious. Furthermore, there are multiple stages in a Machine Learning Workflow that all require different scaling strategies. For instance, scaling hyperparameter optimisation requires a different set of tools and methods. Scaling and extracting the optimum performance from HPC systems for ML applications, therefore, demands advancement along multiple research directions such as model/data parallelism, model/data compression, distributed optimisation algorithms for DL convergence, synchronisation strategies, efficient communication and specific hardware acceleration.

The second challenge is **Performance Modelling** and consequently deciding the right parallelism strategy. Performance modelling is one way to understand the scaling behaviour of parallel applications[18]. Using performance modelling, we can understand how performance changes as different parameters change. This helps us to understand scaling limitations, performance bottlenecks and the right parallelism strategy. Several parallelism strategies have been proposed over the years, and combinations of two or more also exist. Researchers spend too much time trying out different parallelisation strategies, which often becomes obsolete with a change in model architecture or dataset. There is no clear way to determine which approach to use for a given model or dataset. On the other hand, performance modelling for DL is an extremely difficult task that involves analysing memory, communication and performance of multiple systems and testing with a multitude of models.

The third challenge is **Framework Limitations**. Deep Learning frameworks are the core of Deep Learning and need to process multi-dimensional and multi-channel data at scale. However, most DL frameworks are not built with HPC in mind, and therefore, they are not able to leverage the full power of HPC. Table 1.1, shows some of the most common DL frameworks. We observe that most of the DL frameworks are not HPC ready(i.e require extensive engineering effort to support distributed training) and almost always require another library to support distributed computing in HPC.

The fourth challenge is **Fault tolerance**. HPC clusters are susceptible to unrecoverable hardware and software failures that can ruin days or even weeks of training time. As we move towards exascale, meantime to failure is expected to increase [19]. Most Deep Learning frameworks implement some fault tolerance mechanism such as checkpointing to save the training state at a certain point. However, these are not implemented with HPC in mind. Furthermore, for DL, it is not just a defence mechanism against failures but a fundamental component of training DL models used in techniques such as transfer learning. It is therefore important to understand how existing frameworks cope with failures, especially in HPC.

Framework	**Language/API**	**Features**	**HPC Ready**
Tensorflow	C++, Python, Java	Eager execution Computational graph model	✗
PyTorch	C++, Python	Dynamic Graphs Strong GPU support Define by run	✗
Chainer	Python	Define by run Highly intuitive	✓
MXNet	C++, Python, Java	Lean Flexible and Scalable	✓
Keras	Python	Easy Python Intergration Easier programability	✗
CNTK	C++, Python, C#	Support for Apache Spark Simple intergration with Azure Cloud	✓
DL4J	Java	Robust and flexible Support for apache Hadoop and Spark	

Table 1.1: Selected Deep Learning Frameworks and Key Features

1.3 Objectives

Optimum utilisation of HPC for DL is a complex endeavour that demands advancement along multiple research directions. It also requires tools to aid researchers and ML developers in abstracting distributed computing's intricate details and focusing more on building better ML applications. This thesis looks at the convergence of DL and HPC. The **objectives** are two-fold. On one hand, the objective is to study the challenges, limitations and bottlenecks, and limitations of current and future HPC systems when running DL workloads; on the other hand, it is to provide and engineer tools that can alleviate these challenges. Below we discuss the objectives in details.

The first objective of this thesis is to provide a tool to perform Hyperparameter Optimisation(HPO) at scale. HPO is the process of finding the right set of parameters for a DL model from a large space of parameters. These parameters include learning rate, number of layers, batch size, etc. Due to the large search space, HPO takes a long time and involves multiple runs. The objective is to develop a tool that can massively scale this process in a HPC cluster and search through the vast space in a short time, in turn reducing the prototyping time.

The second objective is to develop a tool to aid in the selection of the right parallelism strategy when draining DNNs in HPC. Developing such a tool is complex because performance modelling in HPC requires factoring in many variables and applications have to run along many other applications from different users with different requirements which introduces congestion. To develop the performance model, we first define the main parallel strategies and use that analysis as the basis of the tool. We then implement all the parallel strategies to test the tool.

The third objective is to spark research interest in checkpointing for DL workloads in HPC and provide research directions for checkpointing tools. As DL workloads become increasingly common in HPC, we observe that checkpointing implementations in DL frameworks that have for long been considered sufficient as no longer sufficient. We explore different DL frameworks to identify key insights that can be used to develop more advanced schemes.

1.4 Thesis Contributions

In this section, we provide a summary of the key contributions and chapters of this thesis. These contributions are presented as three separate self sustaining chapters based on three papers published in peer reviewed journals and conferences. Each chapter contains several contributions and findings. Combined, these chapters address the challenges and solutions in the convergence of Deep Learning and High Performance Computing.

Chapter 2 gives a background of Deep Learning, HPC and a survey of ML in HPC. This chapter is essential in that it covers the key concepts required to understand the contents of this thesis. It helps to make this document a self sustaining document. The related work covered is brief because we provide further related work in each chapter.

Chapter 3 presents a Hyperparameter Optimisation Scheme developed on top of PyCOMPSs to aid in scaling Hyperameter Optimisation(HPO) in HPC. We first conducted a literature review of existing work on ML in HPC and realised that, despite being a key component of the ML workflow, existing HPO tools are primarily built for a single node. However, HPO is one of the most time and compute consuming phases of the ML workflow. A careful evaluation also shows that HPO is, in some sense, a low hanging fruit for scaling because it essentially involves running the same experiment with different parameters to identify the best performing. Therefore, we can run each experiment on a node with the possibility of running as many experiments as there are nodes concurrently. We leverage a task-based programming model called PyCOMPSs and propose a scheme that significantly accelerates HPO in clusters. This scheme is framework agnostic, robust, scalable and with all essential features for HPO, such as early stopping and visualisation dashboards.

Chapter 4 is the core of this thesis and provides paraDL. We first present formal definitions of main parallel strategies in CNN training and then propose *paraDL*, an oracle that projects the ideal performance of distributed training of CNNs. To create paraDL, we first define the main parallel strategies include hybrid strategies. We conduct a comprehensive analysis of the compute, communication and memory footprint for training CNNs. We then use this as the basis of the oracle and validate this by implementing all strategies defined and conduct exhaustive experiments up to thousands of GPUs. We show the utility of paraDL in exposing performance and scalability trade-offs achieving accuracies of up to 97.5%. This utility can be used by Machine Learning developers, framework developers and system architects. We also point out other observations such as the rise of hybrid parallelism and the need for distributed inference.

Chapter 5 covers checkpointing for ML in HPC. This is the third and final contribution of this thesis and is essentially a study of fault tolerance of ML applications in HPC. As mentioned earlier, though HPC can reduce weeks of training, HPC systems are susceptible to both hardware and software errors that could ruin this training. For this reason, checkpointing is a widely used technique in HPC to save the state of computation at time intervals and resume in case of failure. Though checkpointing is implemented in almost all frameworks, the implementations are not suitable for HPC, as our study shows. The first reason is that they introduce a considerable overhead because checkpointing is mainly performed by a single node. The second reason is model and data parallelism requires different checkpointing techniques, while most of the frameworks only support checkpointing for data parallelism. With the increasing growth of model parallelism, deep learning frameworks will require more sophisticated checkpointing implementations such as those available for typical HPC applications. Our contribution is we show the overheads that currently exist in checkpointing and identify possible areas of improvement.

Finally, in chapter 6 we recap the key contributions of this thesis and provide some details of the work behind the scenes that led to this thesis. We also provide details of future work. ***All tools developed and proposed in this thesis are open source and publicly available to the research community. All implementations for testing purposes are also publicly available***. This is also a contribution of this thesis.

Chapter 2

Background

In this chapter, we present a detailed background of Deep Learning and key concepts around the topic that are relevant to understand the remainder of this thesis.

2.1 Deep Learning

Deep Learning is a subset of Machine Learning that uses Deep Neural Networks (DNNs). To understand DNNs, we first need to understand Artificial Neural Networks(ANNs), the building units of DNNs. ANNs are computational model inspired by neurons. ANN consists of interconnected neurons that process information. A standard ANN consists of an input layer, a hidden layer and an output layer. DNN are ANNs with many hidden layers that can solve many computationally complex problems such as character recognition, image compression, speech recognition, and computer vision. DNNs are attractive for these sort of problems due to their continuous learning behaviour from given data.

The current state of the art DL results from years of research and development by both the engineering and research community. From the pioneering work by [20] that lead to the concept of Artificial Neural Networks (ANN) to the groundbreaking work by Paul J. Werbos that created the backpropagation algorithm and renewed interest in Artificial Intelligence. Backpropagation led to Deep Neural Networks (DNN) and Deep Learning (DL) that would later outperform humans in computer vision using Convolutional Neural Networks (CNN) [21]. Today, DL is widely used and can outperform humans in many tasks such as image and object recognition and recently playing Go [22].

Training DNNs is hard and generally involves iteratively updating the weights of the layers to reduce the error on prediction. This is done in two steps, **forward propagation** and **backward propagation**. In forward propagation, the network is fed an input, and the model predicts the output. With this output and known ground truth (in the case of supervised learning), a backward pass is performed, and weights are updated using an optimisation algorithm such as Stochastic Gradient Descent(SGD). When training a DNN from scratch, weights are first randomly initialised. However, newer techniques such as **transfer learning** can reduce training times by starting with a previously trained model.

Another important step in training DNNs is Hyperparameter Optimisation(HPO), sometimes called Hyperparameter search. This is the process of finding the correct/best combination of parameters to achieve optimum accuracy. Such parameters include batch size, number of layers in a model, learning rate or even the optimisation algorithm(optimiser). This generally involves training a model multiple times and observing the accuracy and loss curves.

2.2 Parallelism in Deep Learning

Due to their nature, DNNs are computationally very intensive applications. When huge data sets are involved, training DNNs could take days. Because of this, DNNs are now deployed in high-performance computing (HPC) systems where they leverage the high computational power and massive parallelism offered by HPC systems. It is widely agreed that ANN and AI applications' future implementations will continue to be deployed in HPC systems and large data-centres. When training a specific DNN model on an HPC system, there are two prominent strategies for parallelizing the training phase of DL: *data* and *model* parallelism.

2.2.1 Data Parallelism

In data parallel, the model is duplicated into multiple nodes/GPUs. Each model process a mini-batch and computes its gradients and loss for the data it possesses. The models then share gradients through all reduce to obtain average global gradients. Though preferred for its ease of implementation and good results, data

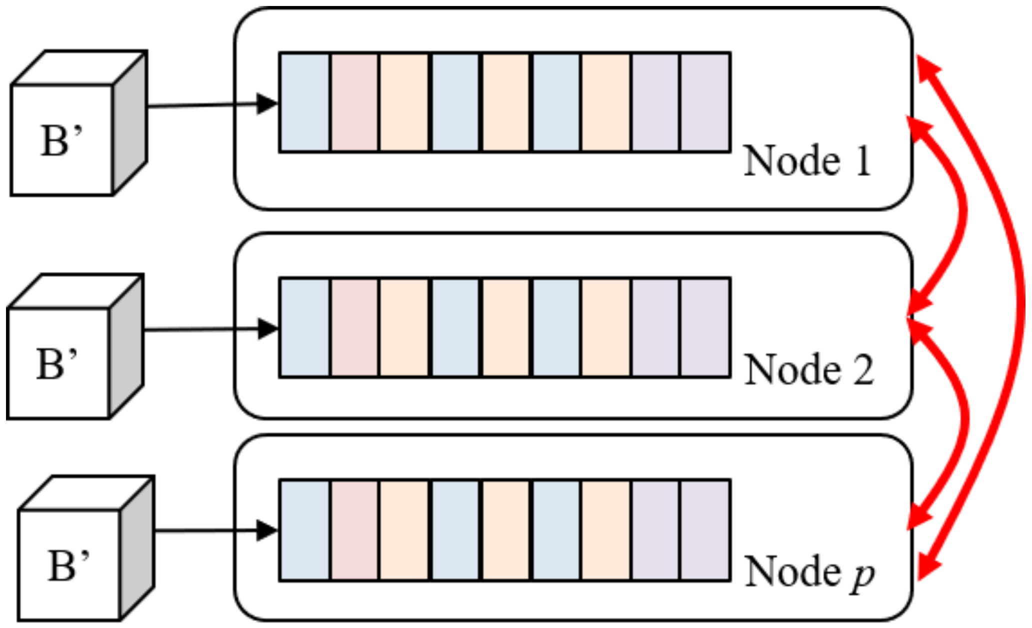


Figure 2.1: Data Parallelism

parallelism has several challenges, such as accuracy degradation with increasing batch size. Communication overhead from Allreduce operation and limited model sizes because models can't be bigger than Node/GPU memory are also limiters for data parallelism. Figure 2.1 shows data parallelism. The rectangles represent the different layers which the red arrows represent communication.

2.2.2 Model Parallelism

Model parallelism divides the model into disjoint subsets, and each subset is assigned on a dedicated GPU. Each GPU is liable for the updates of the designated model layers. Model parallelization is appropriate when the model is huge to be fitted into a single GPU due to memory capacity. Nevertheless, split the model into subsets is not an easy task because it could generate load imbalance issues limiting the scaling [23, 24]. There are several approaches to splitting a model, such as vertically (shown in figure 2.3), horizontally (shown in figure 2.2), across channels/filters or combining one or two methods called hybrid. Model parallelism is discussed extensively in Chapter 4

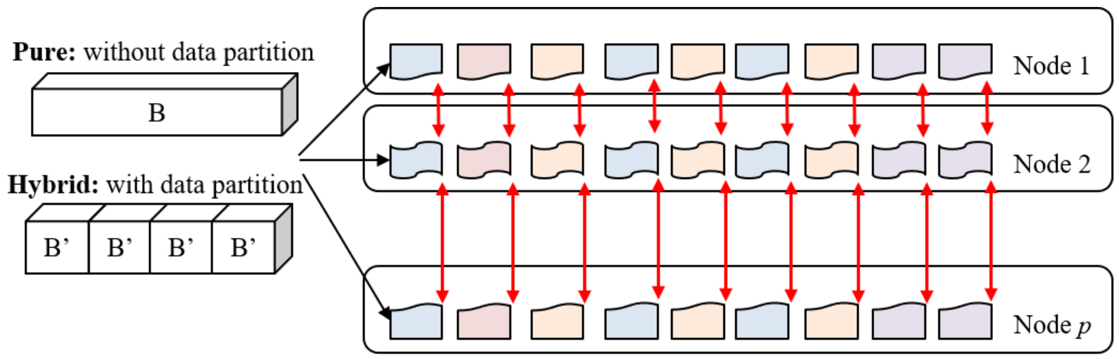


Figure 2.2: Horizontal Model Parallelism

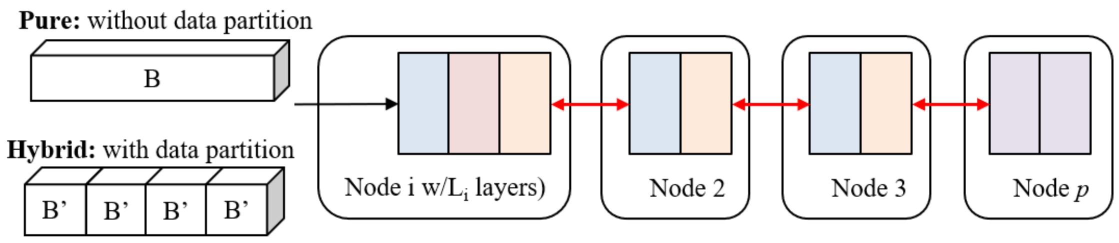


Figure 2.3: Vertical Model Parallelism

2.3 Deep Learning Frameworks

Over the years, numerous DL frameworks have been developed to support training and deployment and DL applications. Almost all major DL frameworks provide some support for distributed training of DNNs. The choice of a DL framework is critical when training ML models in HPC. As mentioned in chapter 1, most DL frameworks are require modifications to fully optimise HPC. In this work, we focus on 3 state of the art frameworks common in HPC ; Chainer [25], PyTorch [26], and TensorFlow [27, 28]. All these DL frameworks provide support for GPUs and allow libraries like CUDA, CuDNN and NCCL. In this section we highlight selected frameworks that are used in this work.

2.3.1 Chainer

Chainer [29] is an open-source Python framework introduced in 2015. The creators of Chainer define it as a ”powerful, flexible and intuitive deep learning framework”. It uses a Defined-by-Run scheme, i.e. Chainer stores the computing history rather than the logic of programming. In the ”Define phase”, a computational graph is constructed (instantiation of a neural network object based on a model definition). Then, in the ”Run phase”, the model is trained by minimizing the loss function using optimization algorithms. To support GPUs, Chainer implements CuPy (Open-source matrix library accelerated with NVIDIA CUDA) [30] that is a package similar to NumPy. For parallelism and especially multi-node parallelism, Chainer comes packaged with ChainerMN, [31] that allows multi-node distributed DL parallel training and use of technologies such as NVIDIA NCCL and CUDA Aware MPI.

2.3.2 PyTorch

PyTorch [26] is a tensor DL framework based on the Torch framework [32] and is deeply integrated with Python. PyTorch performs executions of dynamic tensor computations with GPU acceleration and automatic differentiation. Also, PyTorch provides an array-based programming model implementing the NumPy library, extending the Python multiprocessing module (e.g., moving tensor data between processes through shared memory and not through the communication channel), and allows the implementation of data parallelism in distributed training with CUDA and CuDNN support. Though Tensorflow supports Multi-node execution, most practitioners rely on Horovod(discussed below) for multi-node execution.

2.3.3 Tensorflow

TensorFlow [27, 28] is an open-source library for machine learning that supports various applications with a focus on training and inference in deep neural networks. It originated f the Google Brain project and was based on a system called DistBelief [33]. It is cross-platform and can run on multiple architectures made up of both CPUs, GPUs, or combinations. Additionally, it can run on mobile devices, embedded platforms, and TPUs (tensor processing units), which are specialised hardware to perform calculations with tensors. TensorFlow’s name is derived from multidimensional data arrays called tensors, with which computations are performed to express them as dataflow graphs. This graph represents both the computation in an algorithm and the state in which the algorithm operates. TensorFlow uses highly optimised pre-existing libraries such as cuBLAS (matrix multiplication) or CuDNN to obtain better performance during training. In more recent versions, TensorFlow includes by default the high-level API Keras [34] which allows for more agile and faster experimentation using abstractions and building blocks.

2.3.4 Horovod

Horovod is a distributed deep learning training framework for TensorFlow, Keras, PyTorch, and Apache MXNet.It implements distributed training using MPI as a communication mechanism. The first implementation was based on the bandwidth-optimal ring-allreduce algorithm proposed by Baidu [35]. This framework performs gradient reduction across models and employs inter-GPU communication via the MPI Allreduce algorithm with the possibility of using NCCL (NVIDIA Collective Communications Library). NCCL implements multi-GPU and multi-node collective communication primitives that are optimised for NVIDIA GPUs [36].

2.4 Datasets

Several datasets are used in this thesis. In this section we provide the details of the datasets though each dataset is also discussed in the relative chapters.

2.4.1 Mnist

MNIST[37] is one of the most popular datasets used in Machine Learning education and research. It is a database of handwritten digits from the US Census Bureau with a training set of 60,000 samples and a test set of 10,000 samples. The images are 28x28 pixels (784 features) and each image is labelled with the digit it represents. MNIST is no longer considered an ideal dataset for DL as it is relatively easy to train and more advanced datasets are available.

2.4.2 Cifar

Cifar10 and Cifar100 [38] are labeled subsets of the 80 million tiny images dataset. They were collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class.

2.4.3 ImageNet

ImageNet [39] is a large-scale ontology of images built upon the backbone of the WordNet structure. ImageNet aims to populate the majority of the 80,000 synsets of WordNet with an average of 500-1000 clean and full resolution images. The dataset has over 14 million images spread over 21 thousand groups or classes (synsets). Some of the key milestones and breakthoughs in DL were in the ImageNet Large Scale Visual Recognition Challenge, or ILSVRC which is an annual computer vision competition held between 2010 and 2017.

2.4.4 CosmoFlow

The CosmoFlow project aims to process large 3D cosmology datasets on modern HPC platforms [40]. The CosmoFlow dataset which consists of data from around 10,000 cosmological N-body dark matter simulations. All data for this set is stored in the HDF5 format, with one file per universe/simulation (1GB per file). A sample of the 3D cube is shown in Figure 2.4

2.5 High Performance Computing

Computing plays a critical role in science. The use of computers to solve complex scientific problems and simulations led to high-performance computing systems to provide more computing power through distributed computing. To achieve this, large problems are broken down into smaller problems, and each is computed in a separate computing unit. Information is exchanged between the compute units through a network and specific software such as Message Passing Interface (MPI).

HPC generally involves aggregating hundreds or thousands of computing nodes. Though early supercomputers were primarily based on CPUs, recent developments have seen a significant increase in GPU based supercomputers or a combination with specific accelerators. Since 1993, there is a list of the 500 most powerful computer systems that is maintained and updated twice a year. Figure 2.5 shows performance trends in HPC since 1990s. This work was primarily conducted on two state of the art supercomputers. Below we discuss the details of each.

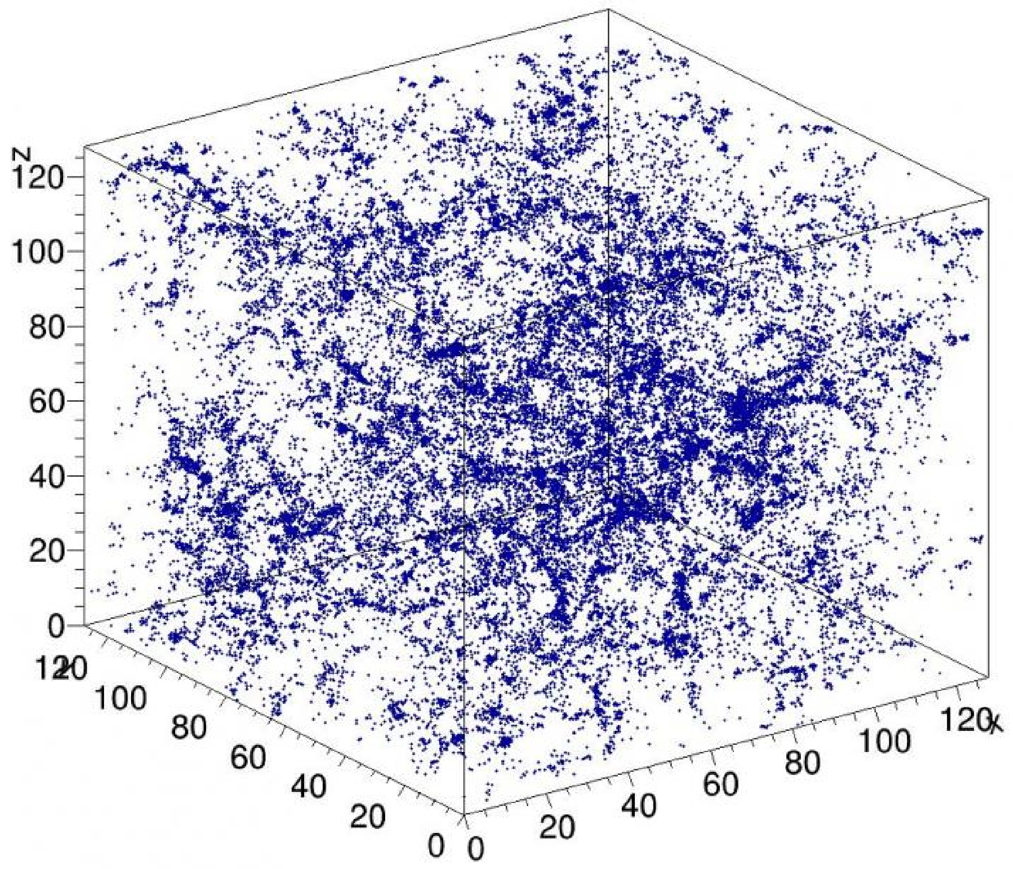


Figure 2.4: CosmoFlow Data Sample

2.5.1 MareNostrum

MareNostrum4 is a supercomputer based on Intel Xeon Platinum processors from the Skylake generation. It is a Lenovo system composed of SD530 Compute Racks, an Intel Omni-Path high performance network interconnect and running SuSE Linux Enterprise Server as operating system. Its current Linpack Rmax Performance is 6.2272 Petaflops. This general-purpose block consists of 48 racks housing 3456 nodes with a grand total of 165,888 processor cores and 390 Terabytes of main memory. Compute nodes are equipped with:

- 2 sockets Intel Xeon Platinum 8160 CPU with 24 cores each @ 2.10GHz for a total of 48 cores per node
- L1d 32K; L1i cache 32K; L2 cache 1024K; L3 cache 33792K
- 96 GB of main memory 1.880 GB/core, 12x 8GB 2667Mhz DIMM (216 nodes high memory, 10368 cores with 7.928 GB/core)

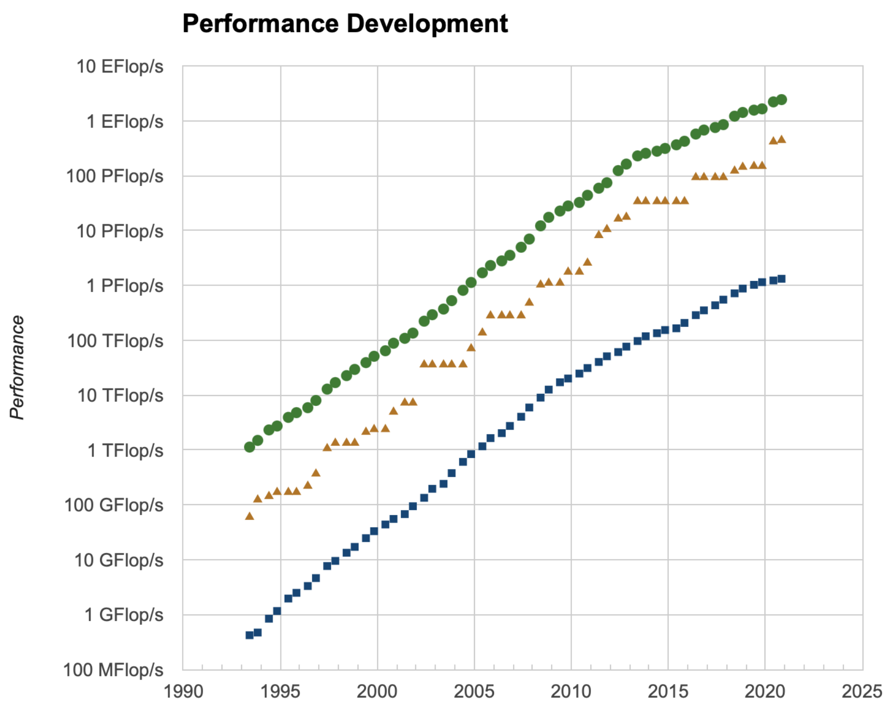


Figure 2.5: Top 500 Performance Development
Source: Top 500

- 100 Gbit/s Intel Omni-Path HFI Silicon 100 Series PCI-E adapter
- 10 Gbit Ethernet
- 200 GB local SSD available as temporary storage during jobs

Most of the experiments on MareNostrum supercomputer were performed on the CTE-Power cluster. CTE-POWER is an experimental cluster based on IBM Power9 processors, with a Linux Operating System and an Infiniband interconnection network. CTE-POWER has 54 compute node, each with the following configurations:

- 2 x IBM Power9 8335-GTH @ 2.4GHz
- 512GB of main memory distributed in 16 dimms x 32GB @ 2666MHz
- 2 x SSD 1.9TB as local storage
- 2 x 3.2TB NVME
- 4 x GPU NVIDIA V100 (Volta) with 16GB HBM2.
- Single Port Mellanox EDR
- GPFS via one fiber link 10 GBit

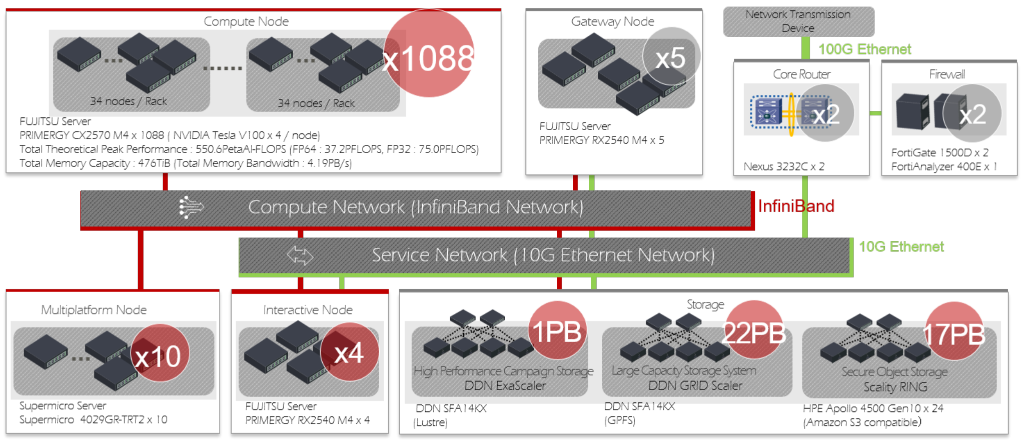


Figure 2.6: ABCI System Architecture

2.5.2 ABCI

AI Bridging Cloud Infrastructure (ABCI), is an open computing infrastructure for both developing AI technology and bridging AI technology into the industry and the real world, constructed and operated by National Institute of Advanced Industrial Science and Technology (AIST). Figure 2.6 shows the architecture of ABCI. The ABCI system consists of 1,088 compute nodes with 4,352 NVIDIA V100 GPU accelerators and other computing resources, shared file systems and ABCI Cloud Storage with total capacity of approximately 40 PB, InfiniBand network that connects these elements at high speed, firewall, and so on. It also includes software to make the best use of these hardware.

2.6 Related Work

Though each chapter discusses related work specific to that chapter, in this section, we present related about Machine Learning in HPC in general. [41] developed develop a procedure for setting minibatch size and choosing computation algorithms. They also derived lemmas for determining the quantity of key components such as the number of GPUs and parameter servers.

Several researchers have also looked into fault tolerance for Deep Learning in HPC. Amatya et al [42] specifically addressed the problem of permanent faults and highlighted the need for fault tolerant MPI specification. They presented an in-depth discussion on the suitability of different parallelism types (model, data and hybrid); a need (or lack thereof) for check-pointing of any critical data structures; and most importantly, consideration for several fault tolerance proposals (user-level fault mitigation (ULFM), Reinit) in MPI and their applicability to fault tolerant DL implementations.

Asaadi et al [43] conduct a comparative study of Deep Learning frameworks in HPC environments. They compare performance of different frameworks and support for different architectures. They also study the HPC-specific features of provided by different frameworks.

On performance modelling, Yan et al [44] observed that the correct choice for model and data partitioning and overall system provisioning is highly dependent on the DNN and distributed system hardware characteristics. These decisions currently require significant domain expertise and time consuming empirical state space exploration. They developed a performance models that quantify the impact of this partitioning and provisioning decisions on overall distributed system performance and scalability. They then used the performance models to build a scalability optimizer that efficiently determines the optimal system configuration that minimizes DNN training time.

Several surveys have also been done. For instance [45] performed a broad and thorough investigation on challenges, techniques and tools for scalable DL on distributed infrastructures. This incorporates infrastructures for DL, methods for parallel DL training, multitenant resource scheduling and the management of training and model data. They also analysed and compared 11 current open-source DL frameworks and tools and investigate which of the techniques are commonly implemented in practice. Finally, we highlight future research trends in DL systems that deserve further research

In another survey, [46]. BEN-NUN et al addressed the challenge of training DNN and described the problem from a theoretical perspective, followed by approaches for its parallelization.They presented trends in DNN architectures and the resulting implications on parallelization strategies. Finally they reviewed and modeled the different types of concurrency in DNNs: from the single operator, through parallelism in network inference and training, to distributed deep learning.They also discussed asynchronous stochastic optimization, distributed system architectures, communication schemes, and neural architecture search. Based on those approaches, they extrapolated potential directions for parallelism in deep learning

Chapter 3

Accelerating Hyperparameter Optimisation with PyCOMPSs

As mentioned in Chapter 2, several parts of the Machine Learning workflow can be parallelised and accelerated by HPC. In this chapter, we present a HPO scheme built on top of PyCOMPSs, a programming model and runtime which aims to ease the development of parallel applications for distributed infrastructures. We show that PyCOMPSs is a powerful framework that can accelerate the process of Hyperparameter Optimisation across multiple devices and computing units. We also show that PyCOMPSs provides easy programmability, seamless distribution and scalability, key features missing in existing tools. Furthermore, we perform a detailed performance analysis showing different configurations to demonstrate the effectiveness of our approach.

3.1 Introduction

To facilitate and accelerate both research and application of ML, numerous tools and libraries have been developed. ML frameworks such as Tensorflow [47], PyTorch [48] and Caffe [49] enable developers to train and deploy complex models. Most of these advanced frameworks focus on training and deployment. However, before training, two important things have to be decided. The architecture of the model and the configuration of the model. Arriving at the right model for a specific dataset is a complex process, that does not only require skilled engineers but is also time consuming and compute intensive. Interestingly though, the time consumed to train the model as well as the effectiveness (accuracy) of the model heavily depends on a set of parameters selected prior the training procedure. This set of parameters is called hyperparameters.

The process of finding the correct combination of parameters for a certain model is called Hyperparameter Optimisation (HPO), sometimes refereed to as Hyperparameter Tuning (HPT). It is one of the key parts in a machine learning workflow. The most common hyperparameters include number of epochs, batch size, learning rate, optimiser, and sometimes specific model parameters such as number of layers. Finding the correct combination of these parameters is non trivial and manual tuning is both difficult and sometimes impossible as the best solution is not always obvious. Furthermore, in most cases HPO takes the longest time as it is not only computationally intensive but also involves multiple trainings. As such, there has been significant research interest in HPO. Research in this domain has generally taken two paths, 1) Algorithms for HPO 2) Tools to implement and execute these algorithms. Further details of both are discussed in section 3.2, but our focus is on the latter.

Numerous tools for HPO have been developed as is evident in section 3.2. An in-depth look into most of these tools reveals 3 major issues that this chapter will address. First, most existing HPO tools are sequential. Those that are parallel constrain the user to a single node and those that span multiple nodes involve complex cluster configuration. Considering that most, if not all, HPO algorithms are embarrassingly parallel, these processes can be significantly accelerated by exploiting both parallelism and distributed execution. Second, recent trends in DL show an increase in the size of models. This not only translates to an increase in the number of hyperparameters(to magnitudes of hundreds) but also requires skillful partitioning and usage of available computing resources as one model can span across several devices and take training time in magnitude of days. Long execution times also raises the important question of fault tolerance.

Third, there has been a proliferation of Machine Learning applications in High Performance Computing (HPC). However, not much has changed in ML workflows, especially for HPO. Since HPO is an integral part of the ML workflow, there is a need for a HPO scheme and tool that can leverage the full power of HPC such as very high inter-node communication, HPC file systems, heterogeneous computing and scalability. However such a tool should not come with a steep learning curve or increased overhead in programmability. A study of current trends and existing literature reveals that an ideal HPO tool should therefore have the following characteristics.

- Parallel : Intra-node task parallelization
- Distributed : Distribute tasks across multiple nodes
- Scalable : Speed up as the number of nodes increases
- Robust : Guarantee a certain degree of fault tolerance
- Framework agnostic: It should not be constrained to a specific framework
- It should also provide essential features such as early stopping and visualisation dashboards to enable researchers make sense of the output.

Taking into consideration the above mentioned, this chapter presents a robust HPO scheme, built on top of PyCOMPSs, to accelerate HPO. PyCOMPSs [50] is the Python binding of COMPSs, a programming model and runtime which aims to ease the development of parallel applications for distributed infrastructures, such as Clusters and Clouds. A detailed description of PyCOMPSs is provided in section 3.3. The contributions of this chapter can be listed as follows.

- We present a robust scheme for HPO in HPC clusters and grid with minimum changes to the code.
- We implement grid search and random search using PyCOMPSs to demonstrate the usage.
- We present an alternative tool for both HPO and other ML workloads that are embarrassingly parallel.

The remainder of this chapter is divided as follows, section 3.2 gives a look into existing tools and previous work on HPO, section 3.3 introduces the PyCOMPSs framework, in section 3.4, we explain how to implement HPO using PyCOMPSs, then we provide details of the experiments performed in section 3.5 and discuss the results in section 3.6 . Finally section 3.7 gives a conclusion and future work.

3.2 Background and Related Work

In this section, we give a brief background of ML and HPO. We then review and discuss existing tools for HPO. The list covered is by no means exhaustive but every attempt has been made to cover the most popular tools.

3.2.1 Background

Even though the idea of a machine capable of learning and mimicking human intelligence was proposed in the early 1950s, its only recently that we have seen significant progress and commendable results. One factor for this is the invention or Artificial Neural Networks (ANNs) and back-propagation, the algorithm used to train these ANNs. The other factor is a major surge in the amount of data available. These combined with increased computing power have made Deep Learning a major research research topic in Computer Science. A subset of this research has been tools and algorithms for HPO.

In algorithms for HPO, the most popular ones are Exhaustive Grid Search and Random Search. Exhaustive Grid search involves trying out all possible combinations and comparing the result using a metric such as loss or accuracy. This approach is feasible when there is a small set of hyperparameters. However, it becomes impossible and unrealistic with a larger search space. Random search [51] was proposed by Bergestra et al and has become more common. Rather than search through the entire search space, combinations of parameters are picked randomly. Empirical results show that random research is more efficient than grid search and arrives at parameters that are good or better at a fraction of the time required by grid search.

Though random search is a superior algorithm in many cases, several other approaches have been proposed. Gaussian Process and Tree-structured Parzen Estimator were proposed by Bergstra et al [52] for Deep Belief Networks. Bayseian optimisation is another approach that essentially builds a surrogate model to approximate the ideal trained model by using different hyperparameters. It's practical usage and implementation is presented by Snoek et al [53] .The tools discussed below implement one or several of these algorithms.

3.2.2 State of the Art

Madrigal et al [54] did a review of existing tools HPO using a computer vision application. They analysed and compared 4 tools for multiple object tracking applications: MCMC, SMAC, TPE and Spearmint. These tools were analysed in terms of stability, performance and usability with the goal of helping making informed decisions when choosing a tool and method for HPO. We discuss other tools not mentioned in that work.

Scikit-learn [55] is perhaps one of the most popular machine learning tools. It combines many of the state of the art algorithms in an easy to use way. Scikit-learn provides both exhaustive grid search and randomized parameter optimisation and uses cross validation to evaluate the best performing parameters. Furthermore scikit-learn computations can can also be run in parallel by setting the number of jobs. However, scikit-learn does not provide multi-node support and is not efficient for complex tasks such as deep learning.

Sherpa [56] is a hyper-parameter optimisation tool geared towards HPO for computationally expensive tasks such as deep learning. It includes several HPO algorithms such as random search, grid search, Bayesian optimisation and local search. Even though Sherpa is intended to run in a multi node environment, doing so requires scheduler and mongoDB. Besides the extra overhead introduced by MongoDB, scheduler configuration is a complex task in HPC.

Shadho [57] developed by Kinnison et al is a general purpose massively scalable hardware-aware distributed hyperparameter optimisation tool. Shadho calculates the relative complexity of each search space and monitors performance on the learning task over all trials. These metrics are then used as heuristics to assign hyperparameters to distributed workers based on their hardware. Shadho achieves double the throughput of a standard distributed hyperparameter optimization framework by optimizing SVM for MNIST using 150 distributed workers.

Hyperopt [58] by Bergstra et al is another tool for serial and parallel HPO over awkward search spaces. It includes random search and Tree of Parzen Estimators (TPE) algorithms. Like Sherpa, HyperOpt also requires mongoDB for parallel execution.

Kopt and Tolos are HPO tools specifically built for Keras. Kopt is based on Hyperopt and requires mongoDB to parallelise on multiple workers. Both are constrained to specific frameworks.

Tune [59] is a unified framework for model selection that allows straightforward scaling in large clusters. Each training is referred to as a trial and an experiment is a collection of trials. Tune is built on top of Ray [60] framework

Google Cloud Machine Learning Engine is part of the larger family of Google products for machine learning. However, the product is heavily dependent on Google infrastructure and is neither open source nor free.

As we shall show in the following sections, PyCOMPSs not only enables the design of more complex workflows with little programming effort, it also handles job management, data transfers dependencies and reuse of memory objects from one task to the next if they use the same object. These key features are not only missing from existing tools, but implementing them in existing job schedulers such as slurm requires multiple reservations and a serious developers effort.

3.3 PyCOMPSs

PyCOMPSs [50] is a task based programming model that enables the parallel execution of existing Python sequential applications, with minimal impact on the development effort, in distributed environments. To do this it offers an interface for parallelizing that uses Python decorators to identify the methods to be considered as tasks, and a small API for synchronization. Formally, PyCOMPSs is the Python binding of COMPSs [1] (Figure 3.1), which relies on the COMPSs runtime and communicates with it whenever a task is detected or its execution requested.

In order to enable the parallelization, the runtime builds a data dependency graph of the tasks that make up the application at execution time. To this end, the task parameters and its direction are taken into account to determine the dependencies among tasks. The runtime is responsible of keeping track of the tasks and respect the dependencies in order to guarantee the validity of the execution, that is, to produce the same result as if is executed sequentially. Consequently, the runtime that can exploit the inherent parallelism of the application at task level and can execute the application in a distributed environment, such as grids, clusters, clouds, and container managed clusters. To achieve this, the runtime is able to schedule the tasks in the available computational resources, acting as an interface with the different computing resources, and transferring the data when needed. COMPSs also supports Java and C++ applications.

The mechanism that PyCOMPSs provides to declare a method as a task is the *@task* decorator, which can be used over any function, instance method or class

[1] compss.bsc.es

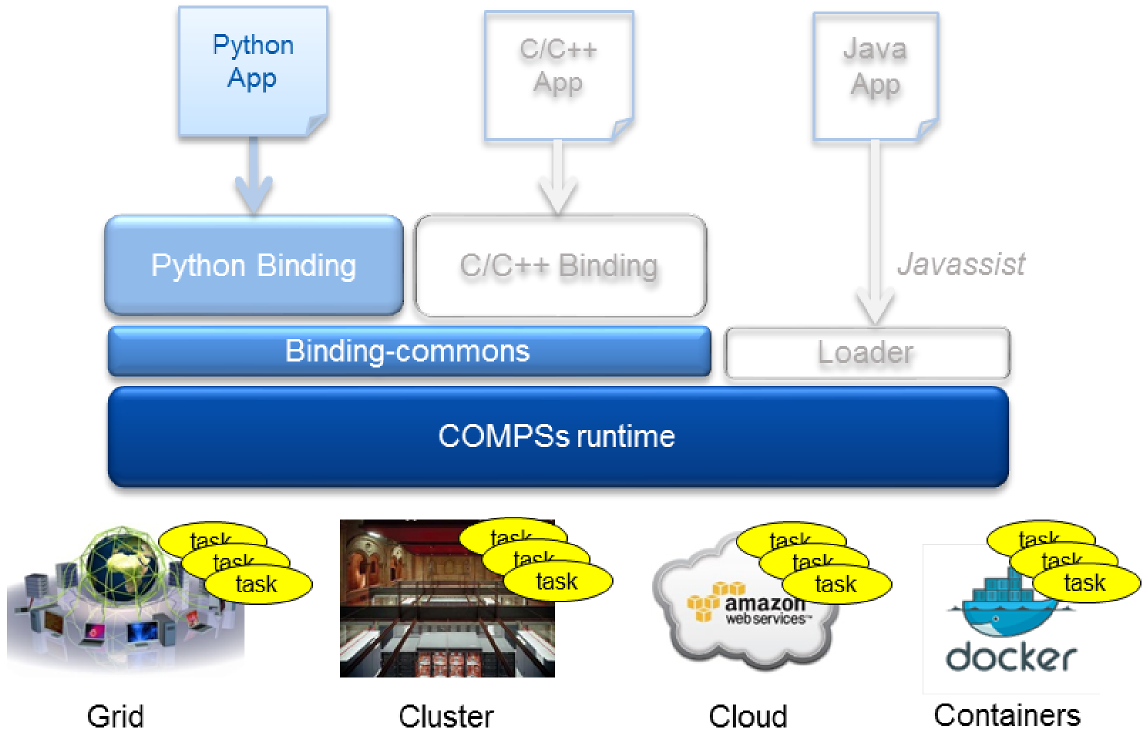


Figure 3.1: COMPSs Architecture

method. In this decorator, hints to specify characteristics of the function parameters or hints for the scheduler can be included. For example, the *returns* keyword into the *@task* decorator can be used to specify the type/number of return elements, the name of a parameter with its type (e.g., FILE) or their directionality (e.g., IN, OUT, INOUT), and *priority=True* for the scheduler so that it tries to schedule that task as soon as possible, among others.

Moreover, PyCOMPSs also provides a set of decorators which can be placed on top of *@task* in order to: define task constraints *@constraint*, define the task as an external binary, MPI or OmpSs executable *@binary, @mpi, @ompss respectively*, declare multiple implementations for the same task *@implement* (this decorator allows the runtime to choose the most appropriate task considering the resources), nesting tasks *@compss*, or even multi node tasks *@multinode*.

Listing 3.2 shows an example of the *experiment* task, which receives an IN parameter (*config* - since its direction is not explicitly defined, default is taken) and returns a single integer value. In addition, a constraint has been defined, declaring that the task requires one core and one GPU.In a nutshell, key strengths of PyCOMPSs that empower its usage for HPO, include the following:

Programmability: PyCOMPSs follows the natural Python way of programming and all a user has to do is add decorators to existing code. Furthermore, in the absence of PyCOMPSs, the program executes sequentially as it would and all PyCOMPSs directions are ignored. This is particularly important because new users can get up and running within no time hence encouraging adaptability.

Seamlessly Distributed: Cluster management and distributed computing can be a complex task that adds a significant overhead to machine learning researchers. PyCOMPSs takes the burden of cluster configuration from the researcher. For instance, if the user wants to use multiple nodes for HPO, they only need to set the number of nodes for the entire job and PyCOMPSs seamlessly manages this resources and allocates tasks based on the requirements of each task.

Resource Management: PyCOMPSs manages all available resources accordingly. The user can exclusively determine the type and number of computing resources for a particular task. Furthermore and very important for machine learning, PyCOMPSs supports heterogeneous resources. As such, for compute intensive deep learning applications, each task can will be assigned a number of CPUs and a GPU. If further, a task has built-in parallelism, PyCOMPSs will not interfere with this. PyCOMPSs also enforces CPU and GPU affinity and therefore prevents tasks from competing for same resources.

Fault Tolerance: A sequential application has a single point of failure. Depending on where the failure happens, this could be a wastage of both time and resources. For long running applications such as HPO, its important to ensure continuity in case of failure. Fault tolerance in PyCOMPSs is supported in two ways. If a task fails for whatever reason, an attempt is made to start the task again. Secondly if a computing unit fails or becomes unavailable for whatever reason, PyCOMPSs restarts this task in another computing unit. This is especially important in machine learning where some tasks are bound to fail during execution due to long execution times.

3.4 Approach

In this section present our approach to implement HPO using PyCOMPSs. First we explain how to structure the application, then give programming details and finally we explain what PyCOMPSs does behind the scenes to distribute the application.

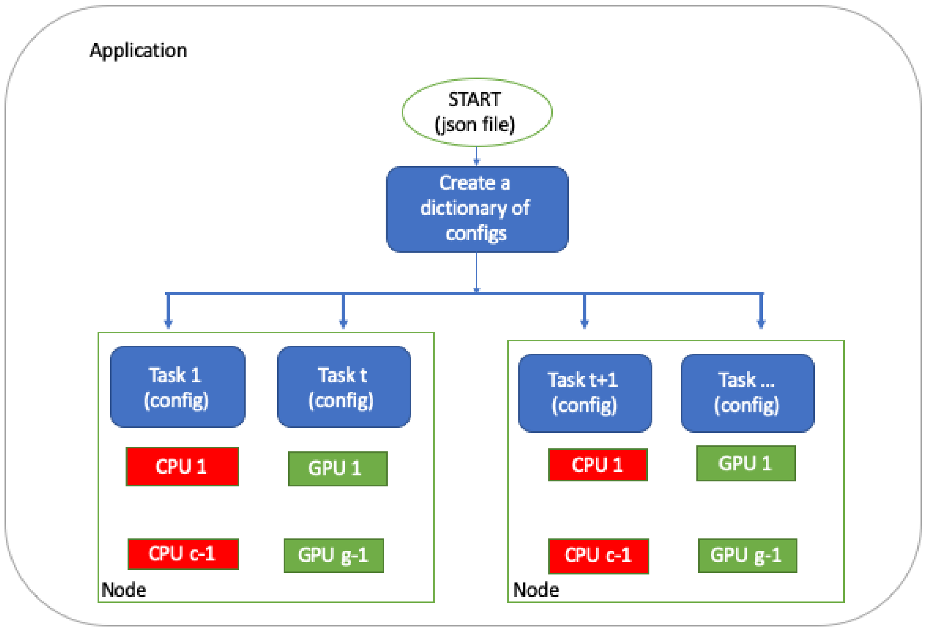


Figure 3.2: Application Structure

As mentioned earlier, HPO is essentially running multiple trainings with different configurations to determine the one that generalises best. In most cases, training doesn't have to run all the way to the end as one can tell how a model is training after several epochs/iterations. However, there are instances of premature convergence and Deep Double Descent [61]. Therefore it is important to test a wide set of parameters. Multiple runs are generally independent of each other. Traditionally, one would just launch one training after the other and make observations. If they have several computers available, one could launch multiple trainings on different computers. We build our HPO tool based on these principles.

On the general structure, at the very top we have an ***application***, which is the entire HPO process. A JSON file containing all the hyperparameters and their values is passed to this application at start. Training and observing a model is an experiment and can be defined as a ***task*** in PyCOMPSs terms. The application will therefore be made up of multiple tasks that will be executed either on the same node or across multiple nodes. Each task requires a unique set of hyperparameters, we call this ***config***, that is passed to the task as a parameter. This configs are generated, depending on the algorithm selected, from the list of hyperparameters contained in the JSON file that was passed to the application. A sample config file is shown in Listing 3.1. A high level overview of the structure and flow is shown in figure 3.2.

Listing 3.1: A simple config file

```
{
    ”optimizer”: [”Adam” ,”SGD”, ”RMSprop”],
    ”num_epochs”: [20, 50, 100],
    ”batch_size”: [32, 64, 128]
 }
```

From the structure show in Figure 3.2 programming entails very few changes to existing sequential code. To make each experiment a task, executable in parallel with other tasks, we simply add the ***@task*** decorator from the PyCOMPSs API before the method/function. We also use the ***@constraint*** decorator to assign the type and number of computing resources to each task, e.g CPU or GPU. We then launch the tasks in a loop passing a different config to each task. The code in Listing 3.2 summarises how to implement HPO in PyCOMPSs.

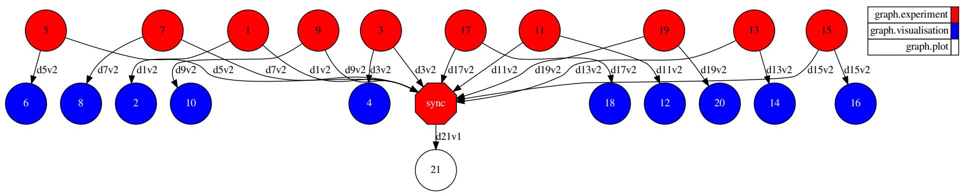


Figure 3.3: Tasks graph

When we launch this application using using PyCOMPSs (to launch we use ***runcompss application.py json_file***), a dynamic graph is created and all dependencies are established. A sample graph for one of the experiments is shown in Figure 3.3. PyCOMPSs then assigns computational resources based on the requirements for each task. When not using a Parallel File System (PFS) such as IBM's General Parallel File System (IBM GPFS) then the data required by the task is copied to the specif node that the task will be executed. Otherwise all tasks can read and write to the PFS. Its important to note that most HPC clusters are equipped with PFS.If no further resources are available, tasks wait for the resources.

Tasks are then executed in workers independently and in parallel to completion. A task can utilise its own internal parallelism if its designed to do so, for instance, Tensorflow executes tensor operations in parallel. Furthermore, for tasks that require multiple GPUs within the same node, this could be easily achieved using the Frameworks internal data/model parallel support. In fact, this could also be a tunable parameter. The case for trainings that require multiple GPUs across multiple nodes has not been explored in this work. In case a task fails for whatever reason (such as node failure), the runtime tries to start the same task in the same node, if it fails again, its restarted in another node. This way, PyCOMPSs ensures fault tolerance. The failure of task does not affect the other tasks unless there are some dependencies.

Listing 3.2: Implementing HPO with PyCOMPSs

```
# PyCOMPSs modulesls
from pycompss.api.task import task
from pycompss.api.api import compss_wait_on
from pycompss.api.constraint import constraint

def create_model(config):
    # New model created every time with different parameters
    # Model parameters can be set here from the config file (i.e optimisers)
    ...
    ...
    return model

@constraint(processors=[{'ProcessorType':'CPU', 'ComputingUnits':1}, {'ProcessorType':'GPU', 'ComputingUnits'
     :1}])
@task(returns=int)
def experiment(config):
    # Trainign parameters can be set here (ie No of Epochs)
    model = create_model(config)
    model.train(config)
    return val_acc

def main():
    args = get_args()
    configurations = process_config(args.config)

    for config in configurations:
        experiment_result = experiment(config)
        results.append(experiment_result)
    results = compss_wait_on(results)

if __name__ == '__main__':
    main()
```

On completion, each task returns the result which can be a performance measure such as validation loss or accuracy and training history. For immediate and interactive action, the performance measure returned can be visualised using another task. When all tasks are completed, we plot the graphs showing the performance of each experiment. To do this, we use the ***comps_wait_on*** over the list of results of all the experiments that synchronizes and ensures all results are available for plotting.

3.5 Experiments

To demonstrate the usage and effectiveness of our scheme, we designed and performed several experiments using popular machine learning benchmarks , MNIST [37] and CIFAR 10 [38]. The experiments are performed at the MareNostrum 4 supercomputer. Each node has two Intel Xeon Platinum chips, each with 24 processors, a total of 48 per node. For GPU implementations, we perform experiments on both MinoTauro cluster, which has 2 K80 NVIDIA GPU Cards and 2 Intel Xeon E52630 v3 (Haswell) 8-core processors and CTE IBM POWER9 cluster which has 2 x IBM Power9 8335-GTH @ 2.4GHz (3.0GHz on turbo, 20 cores and 4 threads/core, total 160 threads per node and 4 x GPU NVIDIA V100 (Volta) with 16GB HBM2.

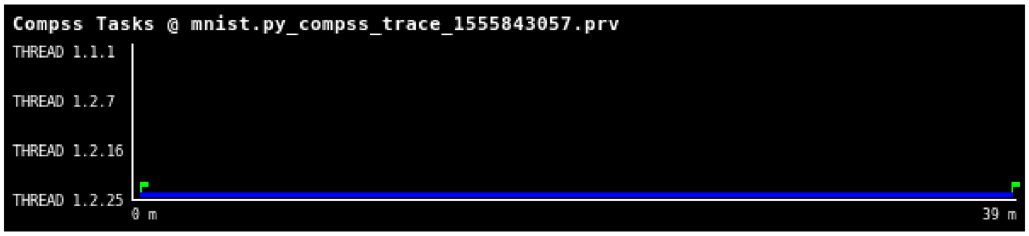


Figure 3.4: Running a single task on a single core

When tracing is set (this is done using a simple flag), PyCOMPSs generates a set of traces that help in application analysis. This is because PyCOMPSs is instrumented with Extrae, an instrumentation package that captures information during the program execution and generates paraver traces. Paraver [62] is a powerful tool that provides detailed quantitative analysis of program performance. We run the first set of experiments with tracing set. From the traces generated, X axis is the time while Y axis is the resource (i.e cores and nodes).

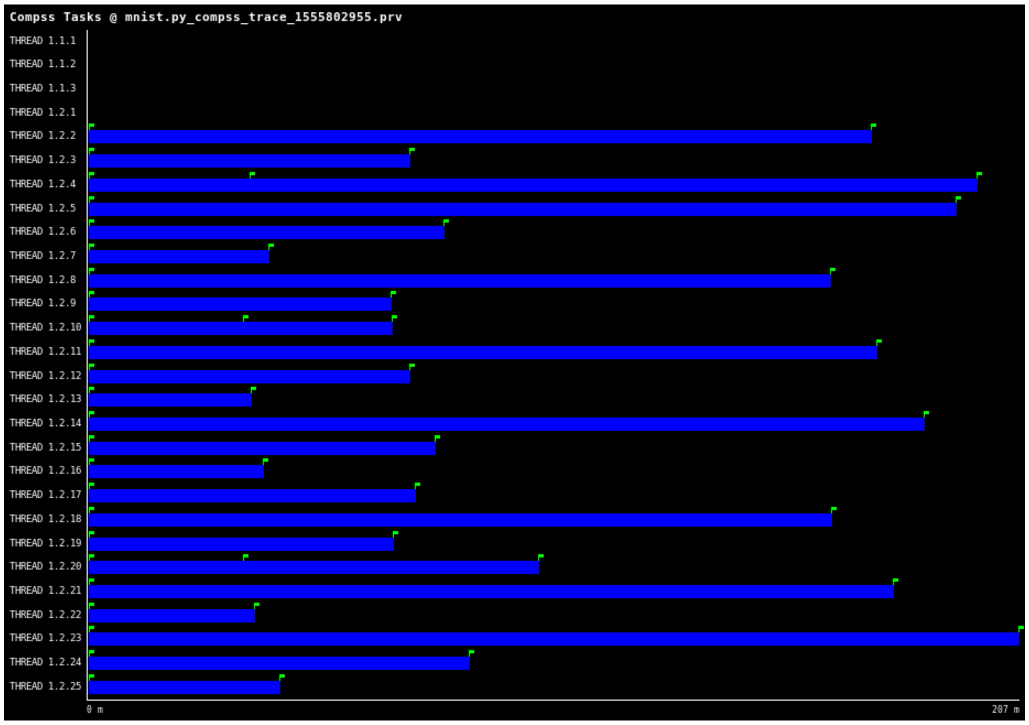
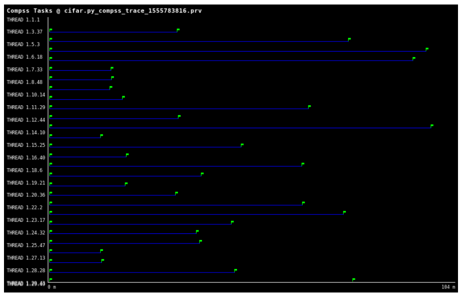


Figure 3.5: Multiple tasks on a single Node

The first experiment is to make sure and show that each task respects the resources given and CPU affinity is enforced. For this, we launch just one task and assign one core in a node with 48 cores. This is done using the MNIST data set as it is not a very compute intensive task. The traces are shown in the Figure 3.4. The task takes around 29 mins to run to completion and its constrained to a single core. Even though tensorflow's default behavior is to span across all available resources, PyCOMPSs is able to enforce CPU affinity and the application has access to only the resources allocated.

The next experiment is to see how tasks are distributed across cores in a single node. For this we launch the full MNIST HPO experiment using grid search. From the configuration file, 27 different experiments are created. The search space is number of epochs, batch size the optimiser, 3 parameters for each. Since the worker takes half of the cores in a node, 24 cores are left for the tasks. As such, not all tasks will run in parallel. However, the next task is assigned a computational


(a) 28 Nodes

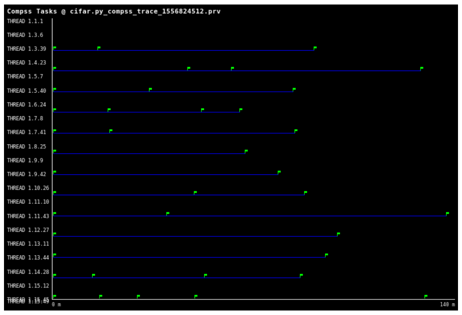

(b) 14 Nodes

Figure 3.6: Multiple tasks on multiple nodes

unit as soon as one is available as shown by the event flags. The traces are shown in Figure 3.5.

The final experiment is to demonstrate HPO across multiple nodes. For this we choose a much bigger dataset, CIFAR10. A total of 27 experiments are created to be distributed across 27 nodes. However, during job submission, we request an extra node for the worker to make sure that all the tasks run in parallel. We assign 48 cores to each task (the total number of cores in a node) and let Tensorflow take care of internal parallelism. We also repeat this experiment with half the number of nodes. The traces are shown in the Figure 3.6.

Both tracing and graph generation create a performance overhead. These two features can easily be turned off by a simple flag when launching the application. Ideally we do not need the traces in HPO, but they are important to show deeper details of the application. As such, we repeat the second and last experiment with traces turned off. We also execute the same experiment on the GPU cluster, as ideally, training is done on GPU. We also repeat the experiments with different GPU and CPU configurations. In these experiments we only measure the execution time. Results for those executions are presented in Section 3.6.

3.6 Results and Discussion

In this section we provide detailed analysis of our tool and the results of the experiments. We first do a performance analysis and resource utilisation by looking at the traces generated. The objective is to show the effectiveness of PyCOMPSs in task parallelism and resource management. The first experiment tested that PyCompss can properly manage the hardware resources available in the supercomputer.

3.6.1 Application Analysis

From Figure 3.5, several observations can be made. First, the tasks take different times to complete with some taking almost half the time. This is due to the different number of epochs from the configuration file. Second, from the event flags, 24 tasks were started at the same time. The remaining tasks are started as soon as a new resource is available, in this case cores 4, 10 and 16 from node 2. The entire application takes 207 minutes. However, as will be shown later, this is not entirely necessary and the process can be stopped as soon as one task achieves a specified accuracy.

In Figure 3.6(a), each task runs on its own node as specified and all tasks run in parallel. The first node seems empty as it is used by the worker. Like in the previous case, some tasks finish earlier than others. This means that it is possible to run the same application with half the number of nodes for almost the same amount of time as the nodes remain idle for the tasks that complete. This is shown in Figure 3.6(b). Clearly, this is a better utilisation of resources. It is important to note that no code changes are required to run across multiple nodes, the user just has to request more nodes when submitting the job. Scaling from a single node to multiple nodes is seamless.

Figure 3.9 shows the time taken to complete the MNIST (CPU nodes) and CIFAR (GPU node) experiment using different configurations. By increasing the number of cores for each task, there is a continuous decrease in the time taken. However in the case of a single node, the time starts to increase after 4 cores. This is because assigning more cores than the total available means that some tasks will be waiting for resources. If the total number of cores available is the same or close to the number of cores requested by the application, the execution becomes sequential and therefore takes a longer time. One should therefore increase the number of nodes as they increase the number of cores per task to create a bigger pool of resources. This is evident when using two nodes as the time taken by the application continues to decrease.

There is a very significant time difference in the GPU Node. The GPU node has 4 GPUs and 160 cores. We assign each task a single GPU (therefore only 4

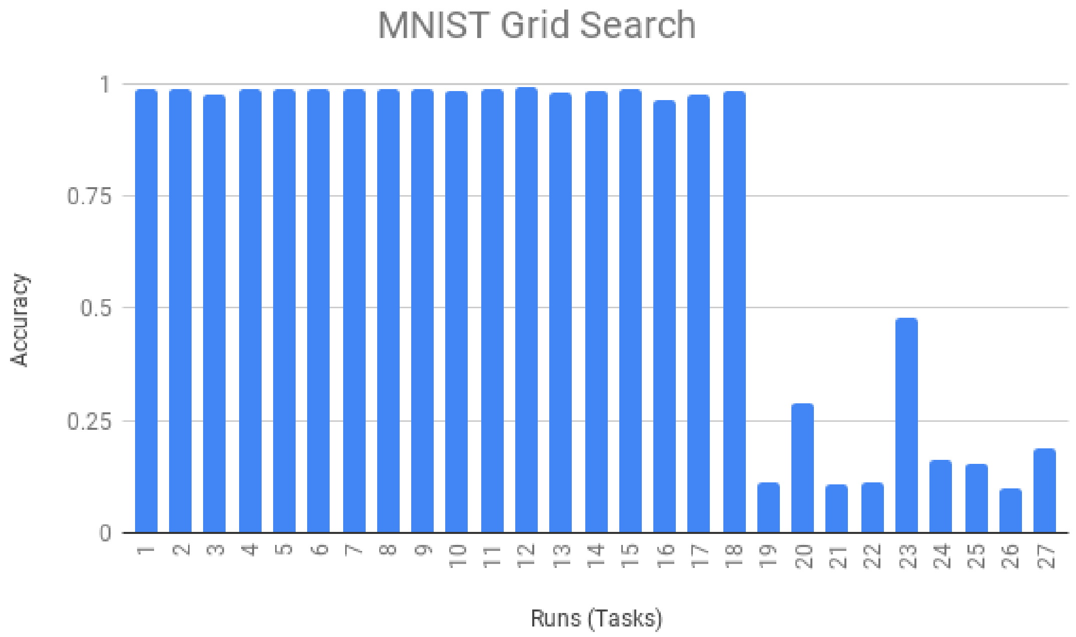


Figure 3.7: MNIST Hyperparameter optimisation using Grid Search

parallel tasks) and continuously increase the number of CPU cores. When using a single core, the time taken is even higher than that of CPU node. The explanation for this is that even though deep learning is significantly accelerated by GPUs, in our set up data preprocessing takes place in the CPU. Therefore a powerful GPU with just a single core is irrelevant as it will be idle more of the time. Increasing the number of cores brings down the time for the entire HPO process to less than an hour even though only 4 tasks run in parallel. Also important to note is that for the GPU node, we run the CIFAR10 dataset, which is much bigger in size. This is to create a noticable difference as the MNIST dataset is too small.

3.6.2 HPO Results

When all the tasks are done, we plot the results the same figure for easier comparison. Figure 3.7 shows the result of HPO for MNIST dataset after the entire application has completed. In this run, our config has the following hyperparameters: Optimizers (Adam ,SGD,RMSprop), Epochs(10, 20, 50) and Batch size (32, 64, 128). MNIST is a relatively simple application that generalises well after just a few epochs. Most of the combinations of hyperparameters are able to attain above

90% accuracy. For such task, early stopping is of paramount significance as it makes no sense to continue with other tasks after one has achieved the desired accuracy.

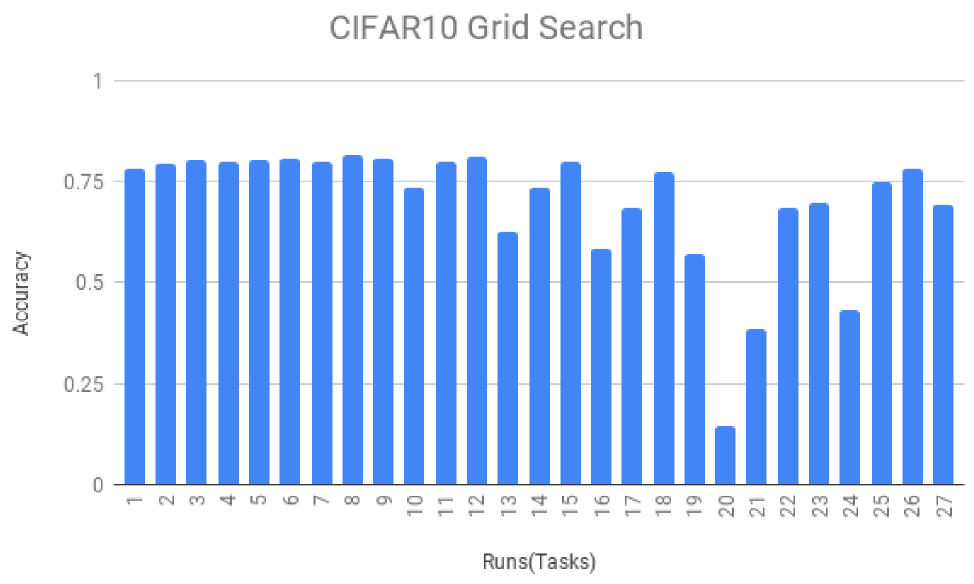


Figure 3.8: CIFAR10 Hyperparameter optimisation using Grid Search

CIFAR 10 is a slightly bigger and more complex benchmark in comparison with MNIST. Figure 3.8 shows the results of HPO for CIFAR 10 dataset. Most of the experiments perform well on the given hyperparameters. We include even hyperparameter combinations that would obviously give low accuracy to show and visualise such cases as they provide useful information as well. As mentioned earlier, random search would be a better alternative in this case as its possible to determine a good set of hyperparameters with just a few experiments.

3.6.3 Discussion

In this chapter, our main focus was to provide a scheme and tool for HPO in HPC clusters. Though we have demonstrated the usage using primarily one algorithm, this scheme provides the user with the flexibility to choose and implement any HPO algorithm. Furthermore, even though all the experiments are implemented with Tensorflow, our scheme does not constrain the user to any framework. We focus on structuring the application rather than the inner details of the application.

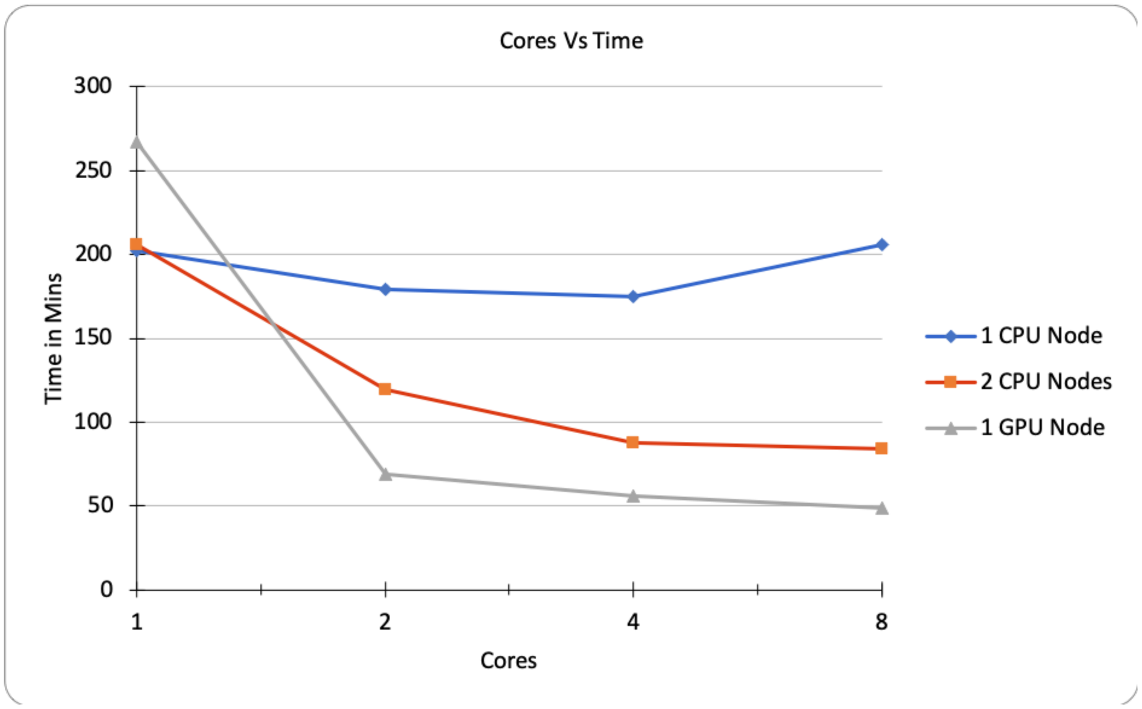


Figure 3.9: Time Vs Cores

Besides easy programmability as evident in listing 3.2, our scheme scales with an increase in the number of both cores and nodes. We tested scalability up to 27 nodes as shown in the traces. The time taken for the entire HPO even when using an algorithm such as grid is drastically brought down with an increase in the number of resources. Furthermore, our tool provides the much needed flexibility in resources management. By specifying the number of GPUs and CPUs for a task, one can come up with an optimal number and type of resources depending on the task.

3.7 Conclusion

In this chapter we have presented a HPO scheme based on PyCOMPSs as an alternative tool for Hyperparameter Optimisation. We have shown that we can span multiple trainings across multiple nodes in a supercomputer and reduce the entire HPO process to days or hours instead of weeks. We have shown that our scheme is not only simple and easy to implement but provides all the features for a HPO tool discusses in section 3.1.

We hope to provide researchers with an alternative tool to accelerate HPO in complex infrastructures such as supercomputers and cloud. Furthermore, we provide a framework agnostic tool with a easy implementation. Even though we intend to support exhaustive grid search and random search, different algorithms can easily be implemented. We further present PyCOMPSs as a framework that enables seamless distributed computing to the machine learning community to facilitate further discussions and innovation around the subject. For future work, we are developing a library that puts together all key algorthms in HPO in an easy to use way. This library will enable the user to perform HPO over any search space by simply calling a function and specifying the algorithm.

Chapter 4

An Oracle for Guiding Large-Scale Training of Deep Neural Networks

With the steady increase in datasets and model sizes, model/hybrid parallelism is deemed to have an important role in the future of distributed training of DNNs. In this chapter, we analyse the compute, communication, and memory requirements of Deep Neural Networks (DNNs) to understand the trade-offs of different parallelism approaches on performance and scalability. We leverage our model-driven analysis to be the basis for an oracle utility which can help in detecting the limitations and bottlenecks of different parallelism approaches at scale. We evaluate the oracle on six parallelization strategies, with four CNN models and multiple datasets (2D and 3D), on up to 1024 GPUs. This chapter is the core of this thesis

4.1 Introduction

DNNs are achieving outstanding results in a wide range of applications, including image recognition, video analysis, natural language processing [8], understanding climate [9], and drug discovery [10], among many others. In the quest to increase solution accuracy, researchers are increasingly using larger training datasets as well as larger and deeper DNN models [13, 14, 15]. In addition, applying Deep Learning (DL) in new domains, such as health care and scientific simulations, introduce larger data samples and more complex DNN models [16]. Those trends make the DNN training computationally expensive for a single node. Therefore, large-scale parallel training on high-performance computing (HPC) systems or clusters of GPUs is becoming increasingly common to achieve faster training time for larger models and datasets [13].

When training a specific DNN model on an HPC system, one of the challenges is to figure out the optimal large-scale parallel strategy. There are two prominent strategies for parallelizing the training phase of DL: *data* and *model* parallelism. It is important to note that despite the early investigation of model parallelism in DL [63], those efforts were premature and remained far from production deployments, since data parallelism was simple and sufficient. However, the growth in datasets and models far outgrows the increase in compute capability [15]. Accordingly, scaling data parallelism can be limited by the memory capacity and the communication overhead.

First, we elaborate on the memory capacity issue. In data parallelism, the entire model is duplicated for each compute node. Therefore, training larger and deeper neural networks have to deal with the memory capacity limits. A notable case is in the area of language modeling at which models are increasingly approaching $O(100B)$ parameters [64] (ex: GPT-3 has 175B parameters [65]). In addition, for sample sizes with higher dimensions, e.g., 3D scientific data sets [40] and videos, the memory capacity would also limit the number of samples that can be concurrently processed by a GPU [66]. Hence, restricting the scaling of data parallelism.

Second we elaborate on the communication overhead. A major bottleneck when scaling data parallelism is the large-message Allreduce collective communication for the gradient exchange at the end of each iteration [67, 68]. As the number of computing nodes increases, e.g., up to 2048 GPUs for Resnet-50 as in, the communication becomes a bottleneck due to the Allreduce collective operation. Several active efforts try to optimize the Allreduction collective algorithm for supporting large messages on specific network architectures in HPC systems [68, 69, 70]. Researchers also reported reducing the size of transmitted data such as quantization [71, 72] and sparsification [73, 74, 75] by using fewer bits to represent the model weights or gradients (known as quantization) [71, 72], or by skipping the transfer of the trivial gradients and instead only share the significant gradients (known as sparsification) [73, 74, 75]. However, even with those algorithms, communication remains a bottleneck when the size of the models increases. Moving to model or hybrid parallelism is one of the ways to reduce this communication overhead [76].

The third issue is the degradation in convergence accuracy when the mini-batch size increases [77, 78]. In recent years, many successful efforts have been dedicated to increase the mini-batch size without (or with small) loss in accuracy in DNNs [79, 80, 67, 68]. For example, work in [68] reports that the mini-batch size reach up to 81K, when training ImageNet [81] on Resnet-50 using Stochastic Gradient Descent (SGD) with up to 2048 GPUs. It helps to reduce the training time from 29 hours, on a single V100 GPU, down to 74.7 seconds [68]. However, there is no empirical evidence or theoretical proof that ensures that techniques such as LARS [82] and LAMB [83], can work well with other different datasets and models.

Approaches	Components					Training Phases				Additional Remarks
	AP	TA	PS	FR	SY	IO	FB	GE	WU	
Optimization methods	○	●	○	○	○	-	-	-	✓	Second order methods [84] (fewer epochs to converge, but longer iterations and more memory).
Normalization	●	○	○	○	○	-	✓	-	-	Cross-GPU Batch-normalization [85] (requires extra communication) and Group Normalization [86].
Pre-trained model	●	●	○	○	○	-	✓	-	-	A big model, that is pre-trained on a big generic dataset, such as Google BiT [87] can be fine-tuned for any task, even if only few labeled samples are available.
Allreduce optimization	○	○	●	○	●	-	-	✓	-	Reduce the communication time by considering the specific network architectures of HPC systems [68, 69, 70] (data parallelism)
Sparsification	○	●	●	○	○	-	✓	✓	✓	Reducing the computation volume [88] or/and communication message size [75] by skipping the computing/transferring of non-important weights/gradients, instead only performing on the significant ones (mainly for data parallelism).
Memory optimization	○	○	○	●	○	-	✓	✓	✓	Reduce required memory by using lower precision (quantization) [71], gradient checking point [89], out-of-core methods [90],
Network architecture	○	○	○	○	●	✓	-	✓	-	Increasing number of GPUs intra node, i.e., up to 16 GPUs in DGX-2. High-throughput inter-node network topology such as HyperX [91] or BiGraph [92].
Model/hybrid parallelism (our target in this work)	○	○	●	○	○	✓	✓	✓	✓	Castello et al. [93] analyzed the communication trade-offs in some model parallel strategies. Hybrid parallelism are proposed in [94] (spatial with data) and [76] (channel/filter with data). [95, 96] explored different parallelization schemes on per-layer basis.

Table 4.1: Recent progress (and examples) in scaling distributed training of DNNs, across different training components and phases.

Given those issues with data parallelism and the growing scale of training, researchers are tackling different bottlenecks across the different components necessary for distributed DNN training. Table 4.1 shows a summary of the recent approaches in scaling distributed DNN training, split into components and training phases. Training components: **AP**- application (models and datasets), **TA**-training algorithms, **PS**-parallel strategies (computation and communication), Training phases: **IO**-I/O and pre-processing, **FB**-a forward and backward propagation, **GE**-the gradient exchange (if needed) and ●: related components. ✓: related training phases. **Despite those efforts, data parallelism is not feasible for all cases.** Thus, it is important to understand the limitations and scalability of large scale model and hybrid parallelism training of DNNs.

In this work, we focus on the HPC aspects of scaling six different strategies for model and hybrid parallelism in CNNs distributed training. Innovations in DL theory (e.g., optimizers) are out of the scope of this thesis. While most works in the literature focus on improving the performance of one single parallelism strategy for one specific framework; our study functions as the basis for a tool, named *ParaDL*, capable of modeling and predicting the performance of a large set of configurations for CNN distributed training at scale. In addition, ParaDL also helps to reveal the practical limits and bottlenecks of different parallel strategies in CNN training .

Our main contributions in this work are as follow:

- We formally define the main parallel strategies (See Section 4.3), including hybrid ones, and provide a comprehensive analysis of the compute, communication, and memory footprint when training CNNs for inputs of any dimension.

- We propose an oracle (ParaDL) A demo of ParaDL is available at the following link: `http://tiny.cc/paraDL`. that projects the ideal performance of distributed training of DNNs, broken down by training phases. This helps in favouring a parallel strategy on a given system and aids in identifying optimisation opportunities in frameworks or system libraries.

- We implement all parallelization strategies to validate our model, except for: a) data parallelism (already supported by most DL frameworks), and b) using an existing pipeline implementation.

- We show the utility of ParaDL in exposing performance and scalability trade-offs. The accuracy of ParaDL (86.74% on average and up to 97.57%) is demonstrated by conducting a wide range of experiments for different CNN models, parallel strategies, and datasets, on up to $1,024$ GPUs (See Section 5.5).

4.2 Background and Notation

As mentioned in Chapter 2, DNNs are made up of a network of neurons (represented as nodes) that are organised in layers (a model). A DNN is trained by iteratively updating the weights of connections between layers in order to reduce the error in prediction of labelled datasets. That is, for a given dataset of D samples, a DNN is trained to find out the model weights w for which the loss function L is minimized. Distributed training of a DNN can be divided into four phases: (**IO**) I/O and pre-processing, (**FB**) a forward phase at which the samples pass through the entire network, followed by a backward phase (back propagation) to compute the gradients, (**GE**) the gradient exchange (if needed) and (**WU**) updating the weights. Specifically, the samples are picked up from the dataset randomly in batches of size B (mini-batch).

The training process is then performed on those batches of samples iteratively by using an optimization algorithm such as the SGD, in which, weights are updated with a learning rate ρ via $w^{iter+1} \leftarrow w^{iter} - \rho \frac{1}{B} \sum_{i \in Batch} \left(\frac{dL}{dw} \right)_i$. The process is then repeated, until convergence, in epochs that randomize the order at which the input is fed to the network.

To optimize for the performance and efficiency of training in large-scale, researchers introduce improvements to the methods, algorithms and design across entire training components which includes: **AP**- application (Deep Learning models and datasets), **TA**-training algorithms (ex: SGD or second order methods), **PA**-parallel strategies (model of computation and communication), **FR**-framework and **SY**-systems.

We summarise our notation in Table 4.2. In a G-layer CNN model, a convolution layer l mainly needs these tensors:

- The input of layer l with N samples, each sample include C_l channels, each channel is a tuple of d-dimension: $x_l[N, C_l, X_l^d]$. In a 2-dimension layer, we replace X_l^d with $[W_l, H_l]$, i.e., $x[N, C_l, W_l \times H_l]$. In a clear context, we omit the layer index l and the dimension d, i.e., $x[N, C, X]$.

- The output (activation) of layer l with N samples and F_l output channel $y_l[N, F_l, Y_l^d]$.
- The weight $w_l[C_l, F_l, K_l^{dim}]$ with F_l filters. Each filter has C_l channels and size of K_l^d. In some case, we omit the filter size (also known as kernel size), e.g., $w_l[C_l, F_l]$.
- The activation gradients $\frac{dL}{dy_l}[N, F_l, Y_l^d]$.
- The weight gradients $\frac{dL}{dw_l}[C_l, F_l, K_l^d]$
- The input gradients $\frac{dL}{dx_l}[N, C_l, X_l^d]$.

We adapt non-Conv layers with the above tensors (we extend the notation in [76]). For channel-wise layers, such as pooling or batch-normalization, we require no further adaption. A fully-connected layer with input $x[N, C, W \times H]$ and F output can be expressed as a convolution layer where the size of filters is exactly similar to the size of the input layer, i.e., $w[C, F, W \times H]$, with padding and stride set to 0 and 1, respectively. Thus, the output will become $y[N, F, 1 \times 1]$.For element-wise layers, such as ReLU, the number of filters F is equal to the number of channels C. For layers without weight, such as pooling and ReLU, the weight becomes $w[C, F, 0]$.It is also possible to express a fully-connected layer with a flattened strategy (although we do not use such way in this work). Assume that the output of the previous layer is $y_{pre}[N, F_{pre}, W \times H]$. We flattened the input, output as $x[N, C = F_{pre} \times W \times H, 1 \times 1]$, $x[N, F, 1 \times 1]$, respectively. Thus the weight becomes $w[C, F, 1 \times 1]$.

The sequential implementation of CNN requires the following steps for each layer :

(IO) $x[N, C, X] \leftarrow IO(\text{dataset, B})$ in the first layer

(FB) $y[N, F, Y] \leftarrow FW(x[N, C, X], w[C, F, K])$

(FB) $\frac{dL}{dx}[N, C, X] \leftarrow BW_{data}(\frac{dL}{dy}[N, F, Y], w[C, F, K])$

(FB) $\frac{dL}{dw}[C, F, K] \leftarrow BW_{weight}(\frac{dL}{dy}[N, F, Y], x[N, C, X])$

At the end of each iteration, weights of all layers are updated:

(WU) $w[C, F, K] \leftarrow WU(\frac{dL}{dw}[C, F, K], \rho)$

4.2.1 Related Work

Gholami et al. [94] investigated the efficiency of hybrid data and spatial parallelism. Castello et al. [93] analyzed the communication trade-offs in some model parallel strategies. Jia et al. [96] showed that different parallelism strategies for different layers can give some performance advantage, for some models. Dryden et al. [76] proposed the use of channel/filter parallelism with data parallelism. They demonstrated improved scaling of ResNet50/ImageNet, mainly due to the effectiveness of intra-node segmented collectives. Huang et al. [15] proposed the use of pipeline parallelism for a single node with multi-GPUs. Recent work [95, 96] explored different parallelization schemes on per-layer basis. The parallelization strategy then becomes an optimization problem to choose the best solution among different permutations of layer-wise parallelization options. The authors suggested a graph search heuristic to find optimal strategies. However, the search heuristic can be a bottleneck since it grows exponentially with the number of potential configurations per layer. Oyama et al [97] presented scalable hybrid-parallel algorithms for training large-scale 3D convolutional neural networks and s showed that good weak and strong scaling can be achieved using up to 2K GPUs. In summary, the related work for studying parallelism in training has typically been limited in scope to applying individual parallel strategies on specific choices of model/dataset.

4.3 Strategies for Distributed Training

Training DNNs using a single processing element (PE) is computationally expensive, e.g., training ResNet-50 over a single V100 GPU requires 29 hours. Hence distributed training on HPC systems is common for large models and datasets. Parallelizing of training process should be done by splitting different dimensions. In this work, we cover four basic parallel strategies that differ in the way we split the data and model dimensions in the training of CNNs: (1) distributing the data samples among PEs (*data parallelism*), (2) splitting the data sample by its spatial dimension such as width or height (*spatial parallelism*) [98], (3) vertically partitioning the neural network along its depth (*layer parallelism*) and overlapping computation between one layer and the next layer [15] (also known as *pipeline parallelism*), and (4) horizontally dividing the neural network in each layer by the number of input and/or output channels (*channel and filters parallelism*) [94, 98]. In addition, a combination of two (or more) types of the mentioned parallelism strategies is named as *hybrid parallelism* (e.g., Data+Filter parallelism and Data+Spatial parallelism, or *df* and *ds* respectively for short, are some examples of hybrid parallelism).

In this work, when presenting tensors such as x, y, and w, we use the $*$ symbol to present a dimension for which its values are replicated between processes. To emphasize that a tensor's dimension is partitioned among different PEs, we use the number of processes p. For example, in data parallelism, $x[p, *, *]$ implies that the input x is split equally in dimension N (number of samples) and partitioned to p PEs. The other dimensions such as C and X are replicated. The arrow $\xleftarrow{Allreduce}$ presents Allreduce communications.

It is important to note that the notation and analysis in this chapter is general to input tensors of any dimension (1D, 2D, and, 3D). Input tensors of higher dimensions are also valid in our analysis since they can be represented as 3D tensor with the extra dimensions as component vector(s) (e.g. CosmoFlow [66] has 4D input represented as a 3D tensor plus a vector at each cell). Finally, the parallel strategy can alternatively be viewed as a domain decomposition problem: a recurring problem in HPC applications. Accordingly, we formulate the notation and analysis to be interpretable as domain decomposition schemes.

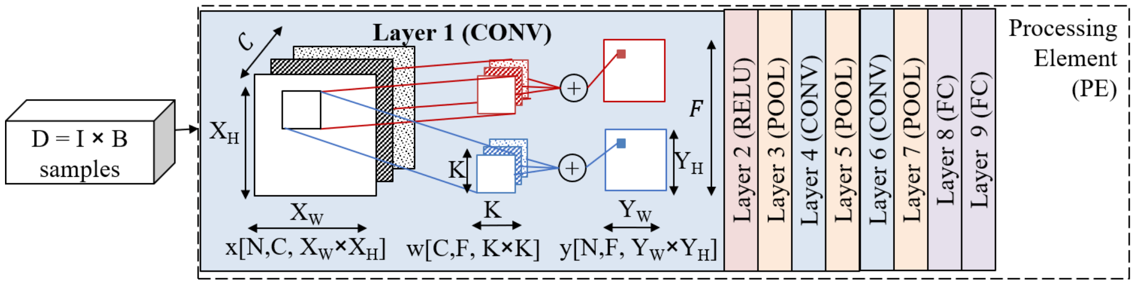


Figure 4.1: Sequential implementation on a single PE

4.3.1 Data parallelism

The entire model is replicated on p different PEs, e.g., GPUs (Figure 4.2) and the dataset is scattered into sub-datasets to each PE. Then the forward and backward

D	Data set size
B	Mini batch size
I	Number of iterations per epoch. $I = \frac{D}{B}$
E	Number of epochs
G	Number of layers
x_l	Input of a layer l
y_l	Output (activation) of layer l
w_l	Weight of layer l
W_l / H_l	Width / Height of input of layer l
C_l	Number of input channels of layer l
F_l	Number of output channels of layer l, e.g., number of filters in conv. layer
FW_l / BW_l	Forward / Backward propagation action of layer l
$[A_1, \ldots, A_n]$	n-dimensions array with size of $A_1 \times A_2 \times \cdots \times A_n$
X_l^d	a d-dimension tuple (array) presents an input channel. In a 2-D convolution layer, X_l^2 is a Cartesian product of $W_l \times H_l$
Y_l^d	a d-dimension output channel
K_l^d	a d-dimension filter. In a 2-D convolution, $K_l^2 = K \times K$
p	Total number of processes elements (PEs)
S	Number of segments in pipeline parallelism
α	Time for sending a message from source to destination
β	Time for injecting one byte of message into network
δ	Number of bytes per item, e.g., input, activation, weight
γ	Memory reuse factor

Table 4.2: Parameters and Notation

phases are computed independently, using those different partitions of the dataset, i.e., in a micro-batch $B' = \frac{B}{p}$ at each iteration. In the gradient exchange phase, an Allreduce operation is required to aggregate the weight gradients, i.e., $\sum_{i=1}^{p} \left(\frac{dL}{dw}\right)_i$. We define operations at the processing element i in data parallelism as:

(IO) $(x)_i[p,*,*] \leftarrow IO(\text{sub-dataset}_i, B')$ in the first layer.

(FB) $(y)_i[p,*,*] \leftarrow FW(x_i[p,*,*], w[*,*,*])$

(FB) $(\frac{dL}{dx})_i[p,*,*] \leftarrow BW_{data}((\frac{dL}{dy})_i[p,*,*], w[*,*,*])$

(FB) $(\frac{dL}{dw})_i[*,*,*] \leftarrow BW_{weight}((\frac{dL}{dy})_i[p,*,*], (x)_i[p,*,*])$

(GE) $\frac{dL}{dw}[*,*,*] \xleftarrow{Allreduce} \sum_{i=1}^{p} \left((\frac{dL}{dw})_i[*,*,*]\right)$

(WU) $w[*,*,*] \leftarrow WU(\frac{dL}{dw}[*,*,*])$

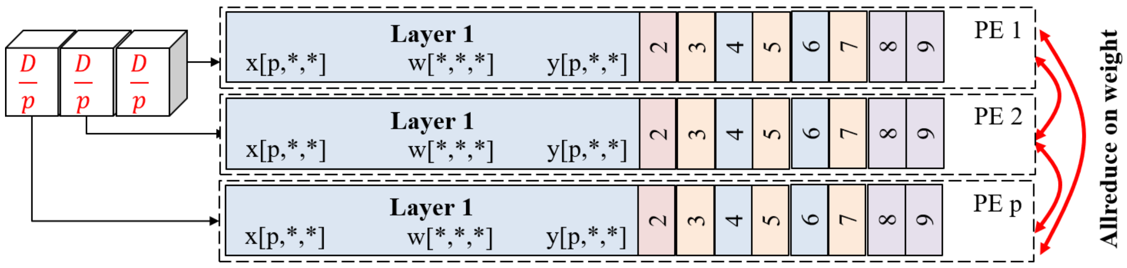


Figure 4.2: Data parallelism

4.3.2 Spatial parallelism (height-width-depth)

All the PEs work on the same batch of samples. First, one leader PE loads those samples at each iteration and then distributes to other PEs. Note that, the spatial dimension H, W (and D as in 3-D convolution layer), of x, y, $\frac{dL}{dx}$ and $\frac{dL}{dy}$ are split among p PEs (Figure 4.3). That is $p = pw \times ph \times pd$ where pw, ph, $pd \leq W$, H, D, respectively. Each process thus performs the forward and backward operation locally. For a convolution layer, when a filter of size $K \times K$ where $K > 1$ is placed near the border of a partition, each PE requires remote data for computing. Thus, a small number (e.g., $\frac{K}{2}$) of rows and/or columns will be transferred from logically-neighboring remote PEs (halo exchange) [98]. The exchanged data size (i.e., halo(x_l)) depends on how each spatial dimension is split, and the stride length. For example, a processing element i needs a halo exchange for its partial input $(x)_i$ to get $(x)_{i+}$ when computing the output $(y)_i$ in the forward phase. In the backward phase, the computation of $(\frac{dL}{dx})_i$ requires a halo exchange on the corresponding $(\frac{dL}{dy})_i$. To compute the weight gradients requires the $(x)_{i+}$, yet no more halo exchange is required since the exchanged values of $(x)_i$ can be reused. In the weight update phase an Allreduce is performed for the sum of $\frac{dL}{dw}$.

(IO) $x[*,*,*] \leftarrow IO(\text{dataset}, B)$

(IO) $(x)_i[*,*,p] \xleftarrow{Scatter} x[*,*,*]$ in the first layer.

(FB) $(x)_{i+}[*,*,p] \xleftarrow{halo} (x)_i[*,*,p]$

(FB) $(y)_i[*,*,p] \leftarrow FW((x)_{i+}[*,*,p], w[*,*,*])$

(FB) $(\frac{dL}{dy})_{i+}[*,*,p] \xleftarrow{halo} (\frac{dL}{dy})_i[*,*,p]$

(FB) $(\frac{dL}{dx})_i[*,*,p] \leftarrow BW_{data}((\frac{dL}{dy})_{i+}[*,*,p], w[*,*,*])$

(FB) $(\frac{dL}{dw})_i[*,*,*] \leftarrow BW_{weight}((\frac{dL}{dy})_i[*,*,p], (x)_{i+}[*,*,p])$

(GE) $\frac{dL}{dw}[*,*,*] \xleftarrow{Allreduce} \sum_{i=1}^{p}\left((\frac{dL}{dw})_i[*,*,*]\right)$

(WU) $w[*,*,*] \leftarrow WU(\frac{dL}{dw}[*,*,*])$

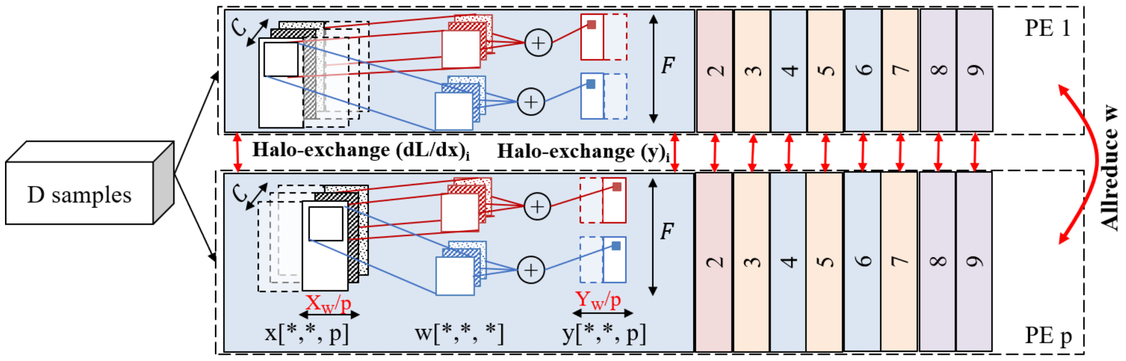


Figure 4.3: Spatial parallelism

4.3.3 Model-horizontal parallelism (filter/channel)

A model parallel variant in which each layer of the neural network model is equally divided by the number of output (filters F) or input channels (channels C) and distributed on p PEs. Each PE keeps a portion of the weights of a given layer and partially computes the output in both the forward and backward phases. For example, the filter parallelism of a convolution layer [94] is illustrated in Figure 4.5. Each PE i keeps $\frac{F}{p}$ filters and computes $\frac{F}{p}$ corresponding channels of the output activation. That is, $|(y)_i| = N \times |Y| \times \frac{F}{p}$. After finishing the forward computation of each layer, the PEs have to share their local output, i.e., $y = \bigcup_{i=1}^{p}(y)_i$ (via an Allgather operation). After finishing the backward computation of each layer, the processes also have to share their gradient of the input (pass it to the preceding layer), i.e., $\frac{dL}{dx} = \sum_{i=1}^{p}(\frac{dL}{dx})_i$ (an Allreduce operation[1]). Because each PE performs the weight-update on its portion of weights, the gradient-exchange phase is skipped.

[1]In the backward phase, because a given layer $l-1$ only requires to use one partition of the layer l's input gradients, i.e., $\frac{dL}{dx}[*,p,*]$, it is possible to perform a Reduce-Scatter instead of an Allreduce operation [76].

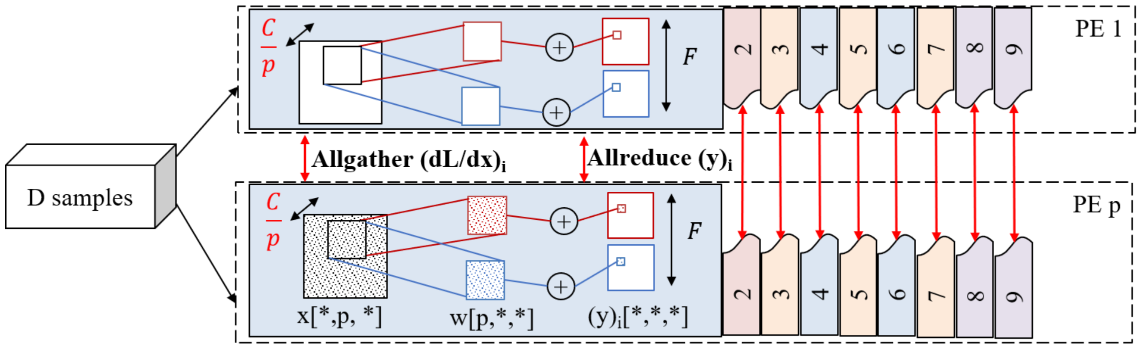


Figure 4.4: Channel parallelism (partition the model horizontally)

(IO) $x[*,*,*] \leftarrow IO(\text{dataset}, B)$

(IO) $(x)_i[*,*,*] \xleftarrow{Bcast} x[*,*,*]$ in the first layer.

(FB) $(y)_i[*,p,*] \leftarrow FW((x)_i[*,*,*], w[*,p,*])$

(FB) $y[*,*,*] \xleftarrow{Allgather} \bigcup_{i=1}^{p} ((y)_i[*,p,*])$

(FB) $(\frac{dL}{dx})_i[*,*,*] \leftarrow BW_{data}((\frac{dL}{dy})_i[*,p,*], w[*,p,*])$

(FB) $\frac{dL}{dx}[*,*,*] \xleftarrow{Allreduce} \sum_{i=1}^{p} (\frac{dL}{dx})_i[*,*,*]))$

(FB) $\frac{dL}{dw}[*,p,*] \leftarrow BW_{weight}((\frac{dL}{dy})_i[*,p,*], (x)_i[*,*,*])$

(WU) $w[*,p,*] \leftarrow WU(\frac{dL}{dw}[*,p,*])$

Channel parallelism [76] (Figure 4.4) is similar to filter parallel strategy but it requires an Allreduce in the forward pass and Allgather in the backward pass.

(FB) $(y)_i[*,*,*] \leftarrow FW((x)_i[*,p,*], w[p,*,*])$

(FB) $y[*,*,*] \xleftarrow{Allreduce} \sum_{i=1}^{p} ((y)_i[*,*,*])$

(FB) $(\frac{dL}{dx})_i[*,p,*] \leftarrow BW_{data}((\frac{dL}{dy})_i[*,*,*], w[p,*,*])$

(FB) $\frac{dL}{dx}[*,*,*] \xleftarrow{Allgather} \bigcup_{i=1}^{p} ((\frac{dL}{dx})_i[*,p,*])$

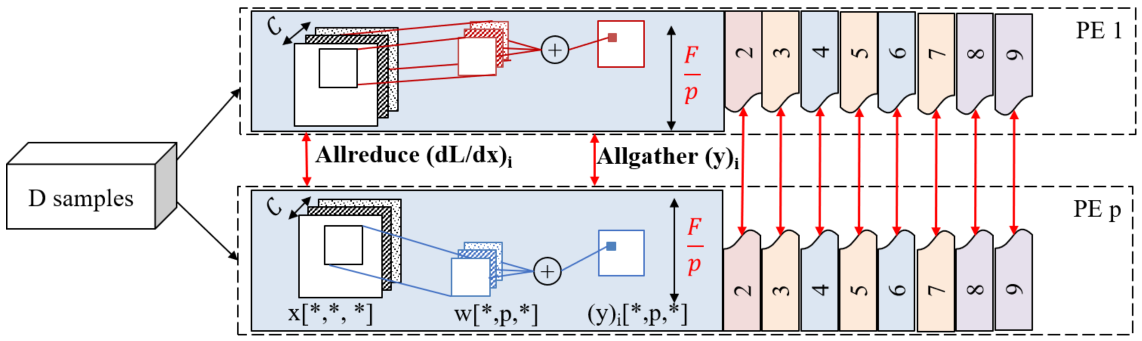
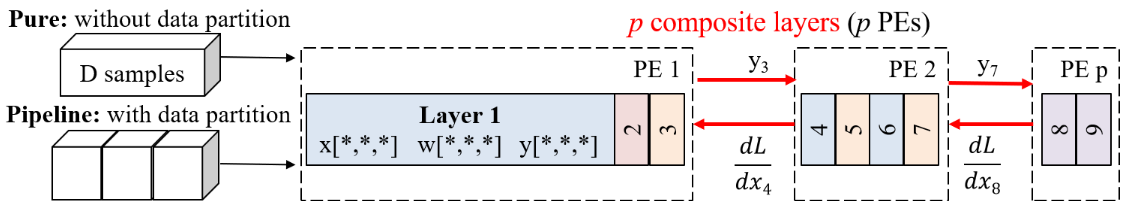


Figure 4.5: Filter parallelism (partition the model horizontally)

4.3.4 Model-vertical (layer) parallelism

A model parallel variant at which the CNN is partitioned across its depth (number of layers G) into $p \leq G$ composite layers, where each composite layer is assigned into one PE, as shown in Figure 4.6. We consider the pipeline implementation of this model parallelism (first proposed by GPipe [15]). The mini-batch is divided into S segments of size $\frac{B}{S}$. In each stage, the forward computation of a composite layer i-th on a data segment s is performed simultaneously with the computation of composite layer $(i+1)$-th on the data segment $s-1$ and so on. The backward computation is done in reversed order.



Pure: without data partition
D samples
Pipeline: with data partition
p composite layers (p PEs)
PE 1
PE 2
PE p
Layer 1
x[*,*,*] w[*,*,*] y[*,*,*]
y_3
y_7
$\frac{dL}{dx_4}$
$\frac{dL}{dx_8}$
2
3
4
5
6
7
8
9

Figure 4.6: Layer parallelism (partition the model vertically)

4.3.5 Hybrid parallelism

We have defined four different main parallel strategies which split the dimension N, $W \times H$ $(\times D)$, F, C, and G, respectively. Without loss of generalization, a layer also can be split by the size of kernel $K \times K$. However, in practice K is so small that parallelizing by dividing K would not give any benefit. Therefore, we focus on the mentioned main strategies. A hybrid parallelism is a combination of two (or more) strategies. For example, Figure 4.7 illustrates the data+filter parallelism.

In which, p PEs are arranged into $p1$ groups of size $p2 = \frac{p}{p1}$. This hybrid strategy implements the filter parallelism inside each group and data parallelism between groups. For a PE $1 \leq i \leq p2$ in a group $1 \leq j \leq p1$:

(IO) $(x)_j[p1,*,*] \leftarrow IO(\text{sub-dataset}_j, B')$

(IO) $(x)_{ij}[p1,*,*] \xleftarrow{Bcast} (x)_j[p1,*,*]$ in the first layer
Filter parallelism inside a group of $p2$ PEs:

(FB) $(y)_{ij}[p1,p2,*] \leftarrow FW((x)_{ij}[p1,*,*], w[*,p2,*])$

(FB) $y_j[p1,*,*] \xleftarrow{Allgather} \bigcup_{i=1}^{p2} \left((y)_{ij}[p1,p2,*]\right)$

(FB) $(\frac{dL}{dx})_{ij}[p1,*,*] \leftarrow BW_{data}((\frac{dL}{dy})_{ij}[p1,p2,*], w[*,p2,*])$

(FB) $(\frac{dL}{dx})_j[p1,*,*] \xleftarrow{Allreduce} \sum_{i=1}^{p2} \left(\frac{dL}{dx})_{ij}[p1,*,*]\right))$
Data parallelism between $p1$ groups :

(FB) $(\frac{dL}{dw})_j[*,p2,*] \leftarrow BW_{weight}((\frac{dL}{dy})_{ij}[p1,p2,*], (x)_{ij}[p1,*,*])$

(GE) $\frac{dL}{dw}[*,p2,*] \xleftarrow{Allreduce} \sum_{j=1}^{p1} \left((\frac{dL}{dw})_j[*,p2,*]\right)$

(WU) $w[*,p2,*] \leftarrow WU(\frac{dL}{dw}[*,p2,*])$

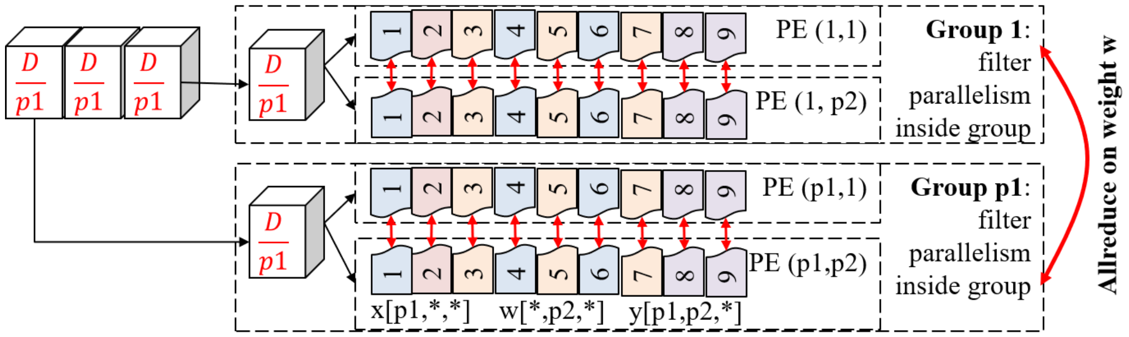


Figure 4.7: Hybrid parallelism (example of filter on top of data parallelism)

Another example of hybrid parallelism is the combination of data and spatial or channel parallelism [76]. Furthermore, the hybrid strategy could be more complex when applying different parallel strategies for different layers [99, 95]. For instance, the parallel strategy of Transformer networks in Megatron-LM [100] partitions a general matrix multiply (GEMM) with filter-wise and then partitions the next GEMM channel-wise.

4.4 Performance and Memory Analysis

In this section, we investigate the performance and memory requirements of different parallelism strategies when training a DNN on a specific system.

4.4.1 Sequential

The training time of one epoch in the sequential implementation (serial) of a CNN includes only the time for computation:

$$\begin{aligned} T_{serial} &= \sum_{1}^{I} B \sum_{l=1}^{G} \Big(FW_l + BW_l\Big) + \sum_{1}^{I}\sum_{l=1}^{G}\Big(WU_l\Big) \\ &= D\sum_{l=1}^{G}\Big(FW_l + BW_l\Big) + \frac{D}{B}\sum_{l=1}^{G}(WU_l) \end{aligned} \tag{4.1}$$

Considering the memory footprint:

$$M_{serial} = 2\delta \sum_{l=1}^{G}\Big(B(|x_l| + |y_l|) + |w_l|\Big) \tag{4.2}$$

In the following, we estimate the total training time and maximum memory per PE for the mentioned basic parallelism strategies and one hybrid strategy. Details about other assumptions and restrictions factored in these estimations are discussed in section 4.5.2.

In this strategy, the training time includes both computation and communication time. Each PE processes a micro batch size $B' = \frac{B}{p}$ in this case. The time for computing at layer l in one iteration for forward and backward phase is $\frac{1}{p}$ of the single-process. Thus the total computation time in one epoch becomes:

$$\begin{aligned} T_{data,comp} &= \sum_{1}^{I}\sum_{l=1}^{G}\Big(\frac{B}{p}(FW_l + BW_l) + WU_l\Big) \\ &= \frac{D}{p}\sum_{l=1}^{G}(FW_l + BW_l) + \frac{D}{B}\sum_{l=1}^{G}(WU_l) \end{aligned} \tag{4.3}$$

Because PEs have to share their gradients at the end of each iteration, the time for communication is $\frac{D}{B}T_{ar}(p, \sum_{l=1}^{G}|w_l|)$. i.e., an Allreduce operation with a ring-based algorithm, the time for communication is:

$$T_{data,comm} = 2\frac{D}{B}(p-1)\Big(\alpha + \frac{\sum_{l=1}^{G}|w_l|}{p}\delta\beta\Big) \tag{4.4}$$

Clearly, data parallelism has the benefit of reduction in computation time by $\frac{1}{p}$ at the price of communication time.

Considering the memory footprint, in data parallelism we duplicate the entire model on p different PEs. Each PE processes a partition of the dataset in a microbatch of $B' = \frac{B}{p}$ samples. A layer l mainly needs memory to store its input $B'|x_l|$, activation $B'|y_l|$, weights $|w_i|$, the gradients $B'|\frac{dL}{dx_l}|$, $B'|\frac{dL}{dy_l}|$, and $|\frac{dL}{dw_l}|$. Some models with a huge number of parameters may require significant memory for ”bias” such as in a fully-connected layer. We consider to not formally show such memory requirement to make our analysis easy to follow. Overall, if each item of the input, activation, weight and gradients are stored in δ bytes, the maximum required memory at one PE is:

$$\begin{aligned} M_{data} &= \sum_{l=1}^{G} \delta(B'(|x_l| + |y_l|) + |w_l| + B'(|\frac{dL}{dx_l}| + |\frac{dL}{dy_l}|) + |\frac{dL}{dw_l}|) \\ &= 2\delta \sum_{l=1}^{G} \Big(\frac{B}{p}(|x_l| + |y_l|) + |w_l|\Big) \end{aligned} \tag{4.5}$$

4.4.2 Spatial parallelism

As mentioned in the previous section, the spatial dimensions of x, y, $\frac{dL}{dx}$ and $\frac{dL}{dy}$ are split among p PEs so that the memory at one PE is:

$$M_{spatial} = 2\delta \sum_{l=1}^{G} \Big(B\frac{(|x_l| + |y_l|)}{p} + |w_l|\Big) \tag{4.6}$$

Because each PE performs a computation with the size of the spatial dimensions as a fraction $\frac{1}{pw}$, $\frac{1}{ph}$, and $\frac{1}{pd}$ of the sequential implementation. This reduces the computation time of forward and backward phase of a layer by $p = pw \times ph \times pd$ times Thus, the computation time is:

$$\begin{aligned} T_{spatial,comp} &= \sum_{1}^{I} B \sum_{l=1}^{G} (\frac{FW_l}{p} + \frac{BW_l}{p}) + \sum_{1}^{I} \sum_{l=1}^{G} \Big(WU_l\Big) \\ &= \frac{D}{p} \sum_{l=1}^{G} (FW_l + BW_l) + \frac{D}{B} \sum_{l=1}^{G} (WU_l) \end{aligned} \tag{4.7}$$

The communication time includes the time to perform the Allreduce operation to share the weight gradients (similar to data parallelism) and the time to perform the halo exchange of each layer. For a layer l, a PE needs to send/receive the halo regions with the logically-neighboring PE(s). Thus the total time for halo exchange is

$$\begin{aligned} T_{spatial,halo} &= 2\frac{D}{B} \sum_{l=1}^{G} (T_{p2p}(B(\text{halo}(|x_l|))) + T_{p2p}(B(\text{halo}(|\frac{dL}{dy_l}|)))) \\ &= 2\frac{D}{B} \sum_{l=1}^{G} (2\alpha + B\delta\beta(\text{halo}(|x_l|) + \text{halo}(|\frac{dL}{dy_l}|))) \end{aligned} \tag{4.8}$$

In which halo$()$ presents the size of data exchanged per batch. The exchanged data size depends on how each spatial dimension is split.

4.4.3 Layer parallelism

In this strategy, a DNN model is split into p composite layers (or group). Let g_i denote the group assigned to PE i. That is, each PE i keeps G_i layers of the model given that $\sum_{i=1}^{p} G_i = G$. Let FW_{G_i}, BW_{G_i}, and WU_{G_i} denote the time for performing the forward, backward, and weight update computation of group i, i.e., $FW_{G_i} = \sum_{l \in g_i}(FW_l)$, $BW_{G_i} = \sum_{l \in g_i}(BW_l)$, and $WU_{G_i} = \sum_{l \in g_i}(WU_l)$.

Pure implementation processes a batch of B samples at the first node and then sequentially pass the intermediate activation (gradients) through all p nodes in each iteration. Hence, the time for computation is:

$$\begin{aligned} T_{layer,comp} &= \sum_{1}^{I}(B\sum_{i=1}^{p}(FW_{G_i} + BW_{G_i}) + \sum_{i=1}^{p}(WU_{G_i})) \\ &= D\sum_{l=1}^{G}(FW_l + BW_l) + \frac{D}{B}\sum_{l=1}^{G}(WU_l) \end{aligned} \tag{4.9}$$

This approach does not reduce the computation time but it is helpful if the memory footprint at one node is limited. In practice, a pipeline implementation is used to reduce the computation time.

In a *pipeline implementation*, the mini-batch is divided into S segments of size $\frac{B}{S}$. In one stage, the computation of a layer group (or PE) g_i on a data segment s is performed simultaneously with the computation of layer group g_{i+1} on the data segment $s-1$, and so on. Thus, the time for each stage can be approximated by the maximum computation time of layer groups, i.e., $\max_{i=1}^{p}(FW_{G_i})$ or $\max_{i=1}^{p}(BW_{G_i})$. In general, a pipeline implementation of p PEs with S data segments requires $(p + S - 1)$ stages per iteration that leads to the total computation time of one epoch as:

$$T_{pipe,comp} \approx \frac{D(p+S-1)}{S}(\max_{i=1}^{p}(FW_{G_i}) + \max_{i=1}^{p}(BW_{G_i} + \max_{i=1}^{p}(WU_{G_i})) \tag{4.10}$$

Considering the communication in this strategy, each PE i has to pass forward/-backward the output/input's gradients to the next/previous PE in a peer-to-peer communication scheme which costs $T_{p2p}(B|y_{G_i}|)$ and $T_{p2p}(B|\frac{dL}{dx_{G_i}}|)$, where y_{G_i} and x_{G_i} denote the output of the last layer and input of the first layer of a group layer g_i, respectively. In the pipeline fashion, the communication time of each stage can be approximated by $\max_{i=1}^{p-1} T_{p2p}(B|y_{G_i}|)$ and $\max_{i=2}^{p} T_{p2p}(B|\frac{dL}{dx_{G_i}}|)$. In the case of $|x_l| = |y_{l-1}|$, the total time for communication in one epoch ($I = \frac{D}{B}$ iterations) is summarized in:

$$T_{pipe,comm} \approx 2\frac{D(p+S-2)}{B}\Big(\max_{i=1}^{p-1}(\alpha + \frac{B}{S}|y_{G_i}|\delta\beta)\Big) \tag{4.11}$$

For the memory footprint, because each PE i stores a different set of layers, the maximum required memory in one PE is:

$$M_{pipe} = 2\delta \max_{i=1}^{p}\Big(2\sum_{l=1}^{G_i}(B(|x_l| + |y_l|) + |w_l|)\Big) \tag{4.12}$$

4.4.4 Filter parallelism

In this strategy, the computation time is reduced p times, yet the time for communication at a layer l becomes more complex, since it includes (1) an Allgather at the forward phase (except layer G)[2] that costs $T_{ag}(p, \frac{B|y_l|}{p})$, and (2) an Allreduce at the backward phase (except layer 1) that costs $T_{ar}(p, B|\frac{dL}{dx_l}|) = T_{ar}(p, B|x_l|)$. In the case of $|x_l| = |y_{l-1}|$, the total time for communication in $I = \frac{D}{B}$ iterations is:

$$T_{filter,comm} = 3\frac{D}{B}(p-1)\sum_{l=1}^{G-1}(\alpha + \frac{B|y_l|}{p}\delta\beta) \tag{4.13}$$

In this strategy, each PE keeps only $\frac{1}{p}$ the filters (weight) of each layer. However, PE i needs to communicate with other PEs to share its local partial activations, hence requiring memory to store the entire activation $|y_k|$. The required memory at each PE is:

$$M_{filter} = 2\delta\sum_{l=1}^{G}\Big(B(|x_l| + |y_l|) + \frac{|w_l|}{p}\Big) \tag{4.14}$$

4.4.5 Channel parallelism

Similar to the filter parallelism, channel parallelism splits the DL models horizontally, i.e., by the number of input channels C. Thus, the computation time, and the required memory at each PE are same as those of filter parallelism

$$M_{channel} = 2\delta\sum_{l=1}^{G}\Big(B(|x_l| + |y_l|) + \frac{|w_l|}{p}\Big) \tag{4.15}$$

[2]Each process i transfers $|(y_l)_i[*, p, *]| = \frac{|y_l|}{p}$ values for one sample in layer l, and a total of $B\frac{|y_l|}{p}$ values for the entire batch.

$$T_{channel,comp} = T_{filter,comp} = \frac{D}{p}\sum_{l=1}^{G}\Big(FW_l + BW_l\Big) + \frac{D}{pB}\sum_{l=1}^{G}(WU_l) \tag{4.16}$$

The communication is performed in a different pattern that includes (1) an Allreduce at the forward phase (except layer G) that costs $T_{ar}(p, B|y_l|)$, and (2) an Allgather at the backward phase (except layer 1) that costs $T_{ag}(p, \frac{B|\frac{dL}{dx_l}|}{p})$. Similar to filter parallelism, we get the total communication time:

$$T_{channel,comm} = 3\frac{D}{B}(p-1)\sum_{l=1}^{G-1}(\alpha + \frac{B|y_l|}{p}\delta\beta) \tag{4.17}$$

4.4.6 Hybrid parallelism (Data + Filter)

We consider an example of hybrid parallelism: the combination of data and filter parallelism in which we use $p1$ data parallelism groups in $p = p1 \times p2$ PEs. We apply filter parallelism inside each group and data parallelism between groups. Each group will process a partition of the dataset, i.e., $\frac{D}{p1}$ samples. Each PE then keeps one part of filters of each layer, e.g., $\frac{F}{p2}$ filters, so that the required memory is:

$$M_{df} = 2\delta\sum_{l=1}^{G}\Big(\frac{B}{p1}(|x_l| + |y_l|) + \frac{|w_l|}{p2}\Big) \tag{4.18}$$

Each PE hence performs $\frac{1}{p2}$ of the computation at each layer with a mini-batch of $\frac{B}{p1}$. The computation time is:

$$\begin{aligned} T_{df,comp} &= \sum_{1}^{I}\frac{B}{p1}\sum_{l=1}^{G}\Big(\frac{FW_l}{p2} + \frac{BW_l}{p2}\Big) + \sum_{1}^{I}\sum_{l=1}^{G}(\frac{WU_l}{p2}) \\ &= \frac{D}{p}\sum_{l=1}^{G}\Big(FW_l + BW_l\Big) + \frac{D}{Bp2}\sum_{l=1}^{G}(WU_l) \end{aligned} \tag{4.19}$$

In this strategy, the communication includes intra-group and inter-group communication, which correspond to the cases of filter and data parallelism. The total communication time of one iteration includes $T_{ag}(p_2, \frac{B|y_l|}{p2})$ and $T_{ar}(p_2, B|\frac{dL}{dx_l}|)$ at each layer and $T_{ar}(p_1, \sum_{l=1}^{G}\frac{|w_l|}{p2}|)$ when update, respectively. The total communication time becomes:

$$\begin{aligned} T_{hybrid,comm} = 3\frac{D}{B}(p2-1)\sum_{l=1}^{G-1}(\alpha + \frac{B|y_l|}{p}\beta) + \\ 2\frac{D}{B}(p1-1)(\alpha + \frac{\sum_{l=1}^{G}|w_l|}{p}\beta) \end{aligned} \tag{4.20}$$

We summarize our analytical model in Table 4.3.

	Computation Time T_{comp}	**Communication Time** T_{comm}	**Maximum Memory Per PE**	**Number of PEs** p
Serial	$D\sum_{l=1}^{G}\Big(FW_l+BW_l\Big)+\frac{D}{B}\sum_{l=1}^{G}(WU_l)$	0	$2\gamma\delta\sum_{l=1}^{G}\Big(B(\lvert x_l\rvert+\lvert y_l\rvert)+\lvert w_l\rvert\Big)$	$p=1$
Data	$\frac{D}{p}\sum_{l=1}^{G}(FW_l+BW_l)+\frac{D}{B}\sum_{l=1}^{G}(WU_l)$	$2\frac{D}{B}(p-1)\Big(\alpha+\frac{\sum_{l=1}^{G}\lvert w_l\rvert}{p}\delta\beta\Big)$	$2\gamma\delta\sum_{l=1}^{G}\Big(\frac{B}{p}(\lvert x_l\rvert+\lvert y_l\rvert)+\lvert w_l\rvert\Big)$	$p\leq B$
Spatial	$\frac{D}{p}\sum_{l=1}^{G}\Big(FW_l+BW_l\Big)+\frac{D}{B}\sum_{l=1}^{G}(WU_l)$	$2\frac{D}{B}\Big((p-1)(\alpha+\frac{\sum_{l=1}^{G}\lvert w_l\rvert}{p}\delta\beta)+$ $\sum_{l=1}^{G}\big(2\alpha+B(\text{halo}(\lvert x_l\rvert)+\text{halo}(\lvert\frac{dL}{dy_l}\rvert))\delta\beta\big)\Big)$	$2\gamma\delta\sum_{l=1}^{G}\Big(B\frac{(\lvert x_l\rvert+\lvert y_l\rvert)}{p}+\lvert w_l\rvert\Big)$	$p=pw\times ph\leq$ $\min_{l=1}^{G}(W_l\times H_l)$
Layer (Pipeline)	$\frac{D(p+S-1)}{S}\Big(\max_{i=1}^{p}(FW_{G_i})$ $+\max_{i=1}^{p}(BW_{G_i})\Big)+\max_{i=1}^{p}(WU_{G_i})\Big)$	$2\frac{D(p+S-2)}{B}\Big(\max_{i=1}^{p-1}\big(\alpha+\frac{B}{S}\lvert y_{G_i}\rvert\delta\beta\big)\Big)$	$2\gamma\delta\max_{i=1}^{p}\Big(\sum_{l=1}^{G_i}\big(B(\lvert x_l\rvert+\lvert y_l\rvert)$ $+\lvert w_l\rvert\big)\Big)$	$p\leq G$
Filter	$\frac{D}{p}\sum_{l=1}^{G}\Big(FW_l+BW_l\Big)+\frac{D}{Bp}\sum_{l=1}^{G}(WU_l)$	$3\frac{D}{B}(p-1)\sum_{l=1}^{G-1}(\alpha+\frac{B\lvert y_l\rvert}{p}\delta\beta)$	$2\gamma\delta\sum_{l=1}^{G}\Big(B(\lvert x_l\rvert+\lvert y_l\rvert)+\frac{\lvert w_l\rvert}{p}\Big)$	$p\leq\min_{l=1}^{G}(F_l)$
Channel	$\frac{D}{p}\sum_{l=1}^{G}\Big(FW_l+BW_l\Big)+\frac{D}{Bp}\sum_{l=1}^{G}(WU_l)$	$3\frac{D}{B}(p-1)\sum_{l=1}^{G-1}(\alpha+\frac{B\lvert y_l\rvert}{p}\delta\beta)$	$2\gamma\delta\sum_{l=1}^{G}\Big(B(\lvert x_l\rvert+\lvert y_l\rvert)+\frac{\lvert w_l\rvert}{p}\Big)$	$p\leq\min_{l=1}^{G}(C_l)$
Data + Filter	$\frac{D}{p}\sum_{l=1}^{G}\Big(FW_l+BW_l\Big)+\frac{D}{Bp2}\sum_{l=1}^{G}(WU_l)$	$3\frac{D}{B}(p2-1)\sum_{l=1}^{G-1}(\alpha+\frac{B\lvert y_l\rvert}{p}\delta\beta)+$ $2\frac{D}{B}(p1-1)(\alpha+\frac{\sum_{l=1}^{G}\lvert w_l\rvert}{p}\delta\beta)$	$2\gamma\delta\sum_{l=1}^{G}\Big(\frac{B(\lvert x_l\rvert+\lvert y_l\rvert)}{p1}+\frac{\lvert w_l\rvert}{p2}\Big)$	$p=p1\times p2\leq$ $B\times\min_{l=1}^{G}(F_l)$

Table 4.3: Computation, Communication, and Memory Analysis Summary (per epoch)

4.5 Performance Projection of Different Parallel Strategies

4.5.1 Overview of ParaDL

In this section we introduce our oracle (*ParaDL*) . Through the information that we can get beforehand, such as the dataset, model, supercomputer/cluster system specification, and user's constraints (e.g., maximum number of involved PEs), ParaDL calculates the computation and communication time to project the overall performance (as described in Figure 4.8). If the strategy differs as the number of nodes increases, ParaDL would breakdown the execution time of different strategies as the number of PEs changes, i.e. scaling the number of PEs. ParaDL can used for the following purposes:

- Suggesting the best strategy for a given CNN, dataset, and resource budget (especially when data parallelism is not feasible).
- Identifying the time and resources to provision from a system (we partially relied on ParaDL in this thesis for that purpose when conducting our empirical experiments in Section 5.5).
- Comparison of projections with measured results to detect abnormal behavior (we relied on ParaDL for this purpose in our analysis of network contention in Section 4.5.3).
- Identifying limitations of parallel strategies, shortcomings of frameworks, and bottlenecks in systems (we relied on ParaDL for this purpose in our discussion in Section 4.6.3).

- As an education tool of the parallel strategies that would improve the understanding of parallelism in DL

Frameworks that are used for DL are comprised of complex and interleaved layers of optimized functions. A pure analytical model of parallel strategies in CNNs would, therefore, be impractical. In this chapter we adopt a hybrid analytical/empirical modeling approach at which we: (i) use analytical modeling for functional requirements driven by the parallelism strategies (Section 4.5.3), and (ii) empirical parametrization for functions not related to the parallel strategy being deployed (more details in Section 4.5.4). Finally, we quantify the accuracy of the oracle with a large empirical evaluation in Section 5.5.

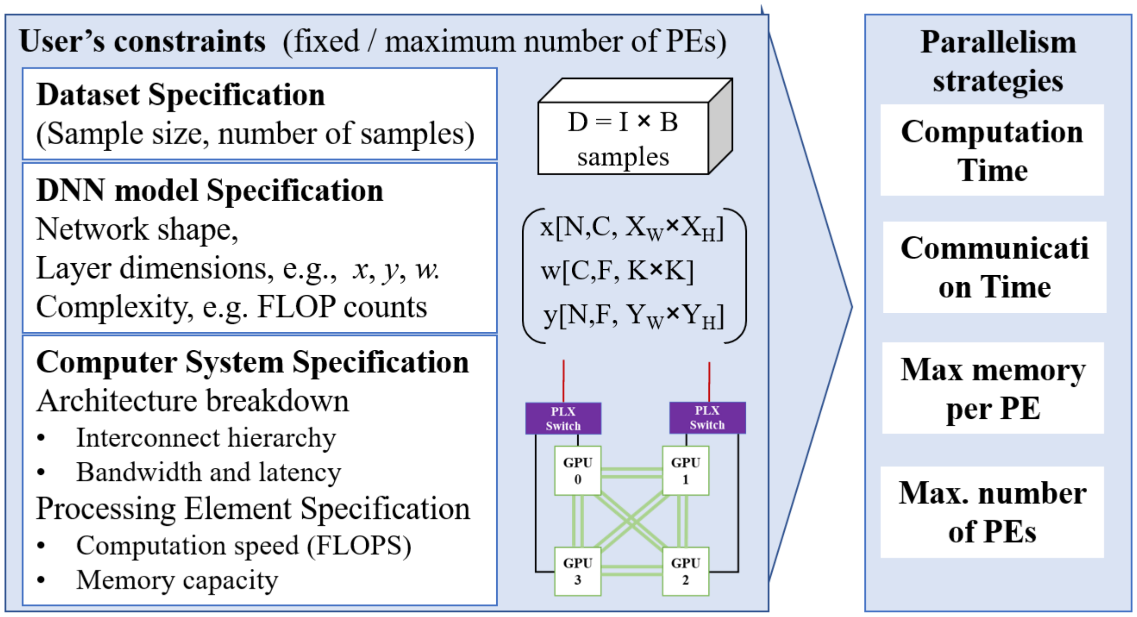


Figure 4.8: Overview of ParaDL

4.5.2 Assumptions and Restrictions

The study in this chapter is based on the following assumptions.

Targeted models and datasets: our study covers all types of layers used in production CNNs, and could hence be used for projecting the performance of any production CNN model, not just the models we evaluate in the paper. We also support the input (i.e. samples) to be of any dimension (as shown in Table 4.2).

Training time and memory estimation: Our study focuses predominantly on the computation and communication time of the CNN training, thus we assume that all the training data is available in memory before starting the training process. In other words, in this model we do not include the time for I/O.

One could conservatively estimate the memory required on a per layer basis by assuming the memory buffers of the output of layer l are different from the memory buffers for the input of layer $l+1$, however, in reality both buffers being the same. Additionally, in reality there is a variety of optimizations that frameworks implement to reduce the memory used (See Table 4.1). Since those optimization methods are complexly intertwined and depend on the framework implementations, without loss of generalization, we propose a practical memory requirement estimation. More specifically, we start out from the naive memory projection that aggregates layers, then we reduce that conservative upper bound to reflect the actual memory optimizations happening inside frameworks. We introduce a memory reuse factor γ. The actual minimum required memory, after all memory reuse optimizations are applied, can be estimated by multiplying total naive required memory by γ. This memory reuse factor can be derived from several elaborate studies on model-level and layer-level memory profiling of CNNs [101, 102].

Parallel strategies: all results in this paper, unless otherwise stated, are for the de facto scaling approach in DL: weak scaling. The mini-batch size scales with the number of PE, hence the number of samples per PE remains constant. In addition, unless mentioned, we do not actively optimize for changing the type of parallelism between different layers in a model, i.e., different layers do not have different parallel strategy. However, there can be cases at which a different type of parallelism is used, in order to avoid performance degradation. For instance, the fully connected layer in spatial parallelism is not spatially parallelized, since that would incur high communication overhead for a layer that is typically a fraction of the compute cost of convolution layers [99].

4.5.3 Performance and Memory Projection

In this section, we estimate the total training time in one epoch and maximum memory per PE for the mentioned main parallel strategies, including hybrid. Let FW_l, BW_l denote the time to perform the computation of forward and backward propagation for one sample and let WU_l denote the time for weight update per iteration at layer l [3]. $T_{ar}(p, m)$, $T_{ag}(p, m)$, and $T_{p2p}(m)$ stand for the time of transferring a data buffer of m-size between p PEs via an Allreduce, Allgather, and a peer-to-peer scheme, respectively. In data parallelism, the training includes both computation and communication time. Each PE processes a micro batch size $B' = \frac{B}{p}$ in this case. The time for FW and BW in one iteration is $\frac{1}{p}$ of the single-process. Thus the total computation time in one epoch becomes:

$$T_{data,comp} = \frac{D}{p}\sum_{l=1}^{G}(FW_l + BW_l) + \frac{D}{B}\sum_{l=1}^{G}(WU_l) \tag{4.21}$$

Because PEs have to share their gradients at the end of each iteration, the time for communication is $\frac{D}{B}T_{ar}(p, \sum_{l=1}^{G}|w_l|)$. Considering the memory footprint, in data parallelism we duplicate the entire model on p different PEs. Each PE processes a partition of the dataset in a microbatch of $B' = \frac{B}{p}$ samples. A layer l needs memory to store its input $B'|x_l|$, activation $B'|y_l|$, weights $|w_i|$, the gradients $B'|\frac{dL}{dx_l}|$, $B'|\frac{dL}{dy_l}|$, and $|\frac{dL}{dw_l}|$. Some models with a huge number of parameters may require significant memory for "bias" such as in a fully-connected layer. We consider to not formally show such memory requirement to make our analysis easy to follow. Overall, if each item of the input, activation, weight and gradients are stored in δ bytes, the maximum required memory at one PE is:

$$M_{data} = \sum_{l=1}^{G}\delta(B'(|x_l| + |y_l|) + |w_l| + B'(|\frac{dL}{dx_l}| + |\frac{dL}{dy_l}|) + |\frac{dL}{dw_l}|) \tag{4.22}$$

In theory, the computation time can be estimated by observing the dataset and CNN model (e.g., FLOP counts and the computation speed of each PE). For modeling the communication time, there exists various derived / specific analytical performance models, e.g., as in the survey of Rico Gallego et. al. [103]. To keep the performance modeling generic, we choose to use the Hockney $\alpha - \beta$ model. In which, the peer-to-peer communication time of transferring a message of size m is modeled by $T_{p2p}(p, m) = \alpha + m\beta$. Time for a message send from a source to a destination is α (also known as startup time) and the time to inject one byte of data into the network is β. We follow the common practice in DL communication libraries such as NCCL [36] to use a ring-based algorithm for all the collective communication operation with large message sizes and a tree-based algorithm for small message sizes. In the ring-based algorithm, a logical ring is first constructed among p PEs based on the system network architecture. Then, each PE partitions

[3] In pipeline, each PE i keeps G_i layers of the model given that $\sum_{i=1}^{p} G_i = G$. Let FW_{G_i}, BW_{G_i} and WU_{G_i} denote the time for performing the forward, backward, and weight update computation of group i per sample.

its m-size data buffer into p segments of size $\frac{m}{p}$. Each PE then sends one data segment to the successive PE and receive another segment from the preceding PE along the ring, i.e., a total of $p-1$ steps for Allgather and $2(p-1)$ steps for Allreduce. Thus $T_{ar}(p,m)$ and $T_{ag}(p,m)$ can be modeled by $2(p-1)(\alpha+\frac{m}{p}\beta)$ and $(p-1)(\alpha+m\beta)$, respectively. Based on this communication model, we also estimate the total training time in one epoch and the maximum memory required per PE for the mentioned parallel strategies. We summarize our analytical model in Table 4.3 [4].

Contention modelling: Ideally, in a system without contention, the start up time α of a given pair is estimated as the total switching latency, which depends on the number of intermediate switching elements. In addition, β is the inverse of the minimum link bandwidth on the routing path between two PEs (the bottleneck link). However, network congestion is one of the biggest problems facing HPC systems today, affecting system throughput and performance. To address the contention effects we introduce the use of a contention penalty coefficient ϕ, which divides the bandwidth of a link by the number of communication flows ϕ sharing this link at each step of collective communications [105]. In our analytical model, we only consider the self-contention caused by all the communication flows of the training process itself, e.g., a link is shared between different groups in hybrid parallelism strategies. The contention coefficient can be estimated analytically by using dynamic contention graphs [106]. It is important to note that we do not intent to model the contention caused by congested networks due to a large number of applications running at the same time in a shared system. Such kind of external contention affects all parallelism strategies and do not reflect the baseline fundamental performance of each parallelism strategy. In addition, the baseline performance predicted by our analytical model can be complemented with a congestion impact factor, which can be empirically estimated as in [107], in order to predict the real-world performance in production environments.

[4]When message sizes are small, communication time with tree-based algorithm can be estimated as $2(\log(p)+k)(\alpha+m\frac{m}{2k}\beta)$ where a message is divided into k chunks to communicate in a pipeline [104].

4.5.4 Empirical Parametrization

As mentioned earlier, we rely on a hybrid of analytical modeling and empirical parametrization for ParaDL. To reduce the impact of noise associated with black-box empirical modeling [18], we segment the experiments used to inspect the target parameters. We are thus able to distinguish between effects of noise on the measurements and actual runtime change because of parameter influence. The empirical parameters are (as defined in Table 4.2):

- **Computation parameters** (FW_l, BW_l, and WU_l): It is important to note that processors, CPUs and GPUs, rarely perform close to their peak performance. We empirically profile the average computation time per sample of each layer (or group of layers) on the target architecture to get a more accurate result. Such profiling can be performed easily and quickly beforehand. Furthermore, the empirical compute time, per a given layer on a given processor, is available in DL databases of models [101].
- **Communication parameters** (α and β): The interconnect hierarchy of modern computing systems, the algorithms used by communication libraries, and the communication technologies (such as GPUDirect [108]) may lead to differences in the latency and bandwidth factors α and β. Thus, we empirically measure the communication time of collective communication patterns, such as Allreduce, with different message size, number of involved processing elements on a specific computing system. Those empirical measurements can be derived from well-known tools for performance of systems, e.g., OSU Micro-Benchmarks or NCCL-test [109]. We then use those benchmark results to interpolate α and β.
- **Memory parameters**: In order to make the model more realistic and reflect memory optimizations, we use the memory reuse factor γ as mentioned in Section 4.5.2.

It is important to emphasise that the empirical parameters in our model are invariant to the implementation of the parallelism strategies, i.e., values of empirical parameters could change when moving from one framework to another, yet values of the analytical parameters would not. Finally, to simplify the portability of ParaDL between different frameworks and systems, we include the following with the ParaDL utility: a) detailed instructions of using the benchmarks used for gathering

Table 4.4: Implementation Overview(✓: customized; - : untouched)

Parallelism strategy	Conv	Pooling	BNorm / LNorm	ReLU	FC
Data	-	-	-	-	-
Spatial	✓	✓	-	-	✓
Filter / Channel	✓	-	-	-	✓

the empirical parameters we use, and b) pointers to DL model and layers databases from which the user could get empirical breakdown of compute and memory requirement at the granularity of layers.

4.5.5 Implementation

Implementation Details

We implement data, channel, filter, spatial and hybrid parallelism strategies using ChainerMN [31] for distributed execution. Although ChainerMN provides a built-in implementation for data parallelism and some minimum level of support for model parallelism, it is not sufficient for testing all the parallel strategies we study here (the same insufficiency also goes for PyTorch, TensorFlow, and others). Substantial engineering effort was required to modify and extend the existing implementation and create **ChainerMNX** to support all forms of parallelism. This extension included modifying existing communicators meant for data parallelism to support hybrid parallelism. We also extended existing convolutional layers to support filter/channel/spatial convolutions[5]. We mark our implementation for each type of targeted layers and parallel strategies in Table 4.4.

More specifically, we use the default implementation of Chainer for data parallelism. Since the size of each dimension (i.e., H, W, and/or D) limits the parallelism of spatial strategy, in this work, we implement the spatial strategy for some first layers of a given model until adequate parallelism is exposed while still maintaining the maximum required memory per node within memory capacity. We then implement an Allgather to collect the full set of activations before passing it to the following layers which perform similar to the sequential implementation. For example, we aggregate after the final convolution layer (before a fully-connected

[5]The code is publicly available here `https://github.com/ankahira/chainermnx`

layer) in VGG16, ResNet-50 and ResNet-152. For CosmoFlow, we aggregate after the second convolution/pooling layer because most of required memory footprint and compute are in those first two layers.

The minimum number of input channels C at each layer limits the parallelism of channel strategy, e.g., only 3 input channels for ImageNet. In this work we implement the channel parallelism from the second layer. For hybrid strategies such as Data+Filter (or Data+Spatial), we map the data parallelism inter-node. This implementation is also used by Dryden et al. [76]. We leverage ChainerMN with MPI support for both inter-node and intra-node communication. To perform an Allreduce and update the gradients in hybrid strategies, we use ChainerMN's multi-node optimizer which wraps an optimizer and performs an Allreduce before updating the gradients. For the Data+Spatial parallelism, we perform a reduce inside each node to a leader GPU, then perform an Allreduce between the leaders. These two Allreduce involve different parallelization techniques (i.e., spatial in local and data in global). For the Data+Filter parallelism, we perform a segmented Allreduce, e.g., disjoint subsets of GPU run Allreduces on different sets of the weights, i.e., number of subsets equals to number of GPUs per node.

Accuracy and Correctness

We aim at making sure our implementation of different parallelism strategies have the same behaviour as data parallelism. We first compare the output activations/-gradients (in forward/backward phases) of each layer (value-by-value) to confirm that the parallelization artifacts, e.g., halo exchange, do not affect the correctness. Note that these new implementations change only the decomposition of the tensors, and do not change any operator or hyper-paramaters that have an impact on accuracy.

Second, in this work we assure that batch normalization (BN) layers are supported in all parallel strategies since the accuracy of training can be affected by the implementation of the BN layers [110, 85]. More specifically, for the data parallelism strategy, the typical implementations of batch normalization in commonly used frameworks such as Caffe, PyTorch, TensorFlow are all unsynchronized. This implementation leads to data being only normalized within each PE, separately. In typical cases, the local batch-size is usually already large enough for BN layers to function as intended. Yet in some cases, the local batch-size will be only 2 or

4 in each PE, which will lead to significant sample bias, and further degrade the accuracy. In this case, we suggest to use the synchronized BN implementation as mentioned in [85], which requires a communication overhead for computing the global mini-batch mean. For the spatial strategy, although performing batch normalization on subsets of the spatial dimensions has not been explored, to the authors knowledge, this computation requires no significant adjustment [98]. More specifically, BN is typically computed locally on each PE on its own portion of the spatially partitioned data.

In the filter and channel parallelism strategy, since all PEs keep the same set of activations after performing the Allgather operation at each layer, the BN layer could be implemented as in the sequential strategy. It could be implemented in a centralized fashion, e.g., one PE performs the BN and then sends the result to other PEs. Alternatively, each node could redundantly compute the BN layer (distributed approach). In this work, we use the distributed approach which does not require any communication overhead.

4.6 Evaluation

In this section, we describe how we conduct a wide range of experiments to show the accuracy and utility of ParaDL in projecting the performance of distributed training of CNN models under different parallel strategies, including hybrid ones. We compare ParaDL projection results to the empirical measurements on a multi-petaflop HPC system with thousands of GPUs. In addition, we characterize the bottlenecks and limitations of different parallelization strategies, and highlight relevant findings observed with the help of ParaDL. Finally, we discuss how the ParaDL can aid framework developers and system builders in i) guiding the implementation and possible optimizations of different parallel strategies in existing DL frameworks; and ii) advising system architects on the best co-design choices for their system depending on the workloads they plan to run.

4.6.1 Methodology

Selected Models and Datasets

We choose different CNN models and datasets with different characteristics that affect performance and memory requirements. They are summarised in Table 4.5.

Table 4.5: Models and Datasets Used in Experiments

Model	Dataset	#Samples (Size)	# Param.	#Layers
ResNet-50 [21]	ImageNet [81]	1.28M (3$\times 226^2$)	$\approx$ 25M	50
ResNet-152 [21]			$\approx$ 58M	152
VGG16 [111]			$\approx$ 169M	38
CosmoFlow [66]	CosmoFlow [40]	1584 (4$\times 256^3$)	$\approx$ 2M	20

Evaluation Environment

Experiments are performed on the ABCI supercomputer, a multi-petaflop supercomputer, with two Intel Xeon Gold 6148 Processors and four NVIDIA Tesla V100 GPUs (16GB of memory per GPU) on each compute node. The GPUs are connected intra-node to the CPUs by PLX switches and PCIe Gen3 x16 links (16 GBps), and together by NVLink (20 GBps). The compute nodes are connected in a 3-level fat-tree topology which has full-bisection bandwidth, and 1:3 over-subscription for intra-rack and inter-rack, respectively (two InfiniBand EDR, e.g., 12.5 GBps, per compute node and 17 compute nodes per rack).

Configurations of Experiments

We perform the experiments of the parallel strategies using the framework Chainer [29] (v7.0.0), ChainerMN [31] (the multi-node varient of Chainer), and CUDA (v10.0). We also use the PyTorch (v1.5) implementation for the pipeline strategy [112]. We implement all communication functions based on Nvidia's NCCL library [36] (v2.4.8.1). The exceptions are at which we use MPI (OpenMPI v2.1.6): a) the halo exchange of the spatial strategy since P2P communication interfaces are not supported by NCCL[6], and b) the Allgatherv for the spatial strategy since NCCL does not support Allgatherv.

[6]The latest version of NCCL now supports P2P communications.

An important performance factor is efficient device utilization of GPUs. Thus, we conducted a series of test runs for each type of parallelism and DL model to identify the optimal number of samples per GPU (or node) that would efficiently utilize the device. We observed that the performance drops significantly when we train using a higher samples/GPU number than the optimal one. This occurs when the computational load becomes too large to effectively utilise a single GPU. For CosmoFlow with spatial strategies, since we use only one sample per node, i.e., 0.25 samples/GPU: we could not have the freedom to tune the parameter b. This is often the case for models using large 3D input datasets (increasingly common in scientific computing), where data parallelism is simply not an option. Given that it was not possible to get the empirical layer by layer time for CosmoFlow running a sequential implementation with the 512^3 data size, we used 256^3 sample sizes and multiplied the computation time by 8. We confirmed with measurements that the strategy was accurate.

4.6.2 ParaDL's Projection and Accuracy

This section discusses how close is the projection of the ParaDL oracle in comparison with the measured experiments. It is a complex task to accurately project the performance of DL training, especially when scaling. More specifically, the following factors have a significant effect on performance: contention on the PFS, the effectiveness of the pipeline used for asynchronous data loading, network contention, implementation quality, and overheads of solution fidelity book-keeping. That being said, in this section we aim to demonstrate that the presented oracle, despite the complexities mentioned above, reasonably represents the reality of measured runs on an actual system (especially when scaling up to 1024 GPUs). In this comparison, we focus only on the computation and communication time of the main training loop (the most time consuming part) and exclude other times from this study such as I/O staging.

Figures 4.9, 4.10, 4.11, 4.12, 4.13, 4.14 show the oracle's projections versus the measured runs for different parallel strategies using three different models (a fourth model is shown in Figure 4.15). The label above each column shows the *projection accuracy*. The x-axis is the number of GPUs. Filter/channel are strong scaling.$^{(*)}$Values are total time since pipeline parallelism [112] overlaps the computation and communication.

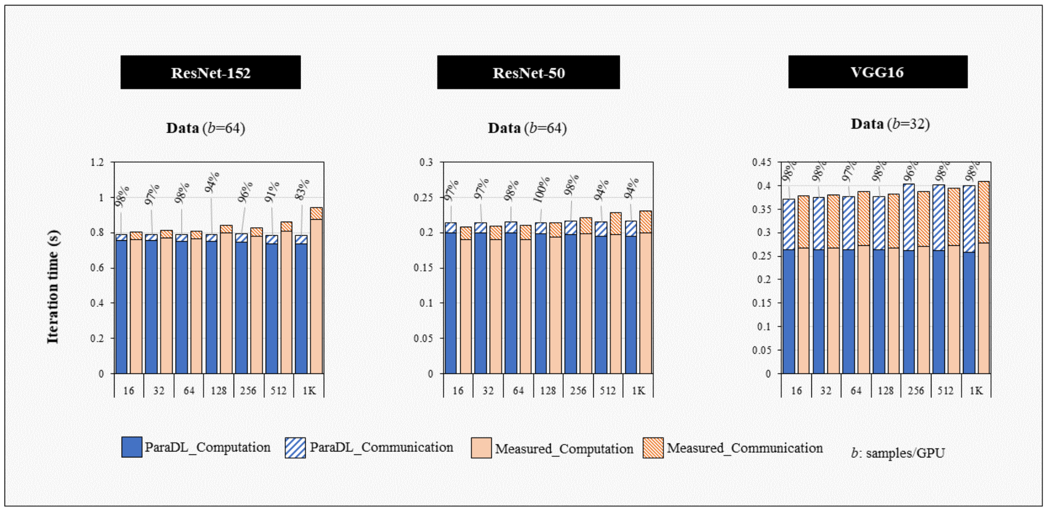


Figure 4.9: Prediction Accuracy of ParaDL with Imagenet for Data Parallelism

We ran all the permutations of possible configurations but plot only some of them because of space limitations. Each figure is divided in three columns, one for each CNN model. The parameter b shows the mini-batch size for each case. As mentioned in Section 5.4, the mini-batch size is set to achieve the highest device occupancy on GPUs. The x-axis shows the number of GPUs, up to the scaling limit of the specific parallel strategy (e.g., maximum number of filters). More specifically, we scale the tests from 16 to 1024 GPUs for data and hybrid parallelism, from 4 to 64 GPUs for filter/channel parallelism, and up to 4 GPUs for pipeline parallelism. The y-axis shows the iteration time for each case. The iteration time is calculated as an average of 100 iterations excluding the first iteration which normally involves initialization tasks. To get a more detailed analysis, we decompose the execution time into computation and communication. The oracle prediction is shown in blue as stacked bars, i.e., computation+communication, and the measured empirical results are shown in orange. In this figure, we report the best communication times obtained during our experiments, as this represents the peak performance the hardware can deliver and leave aside occasional delays due to external factors such as network congestion coming from other apps, system noise and, overheads due to correctable errors, among others. A detailed analysis on network congestion is given on Section 4.6.3. The labels above each column show the *projection accuracy* in percentage, i.e., 1 - ratio of the absolute value of the difference with respect to the total measured time. Similarly, Figure 4.15 shows the accuracy for CosmoFlow in the case of Data+Spatial parallelism. Note that the reason CosmoFlow is only run on the Data+Spatial hybrid

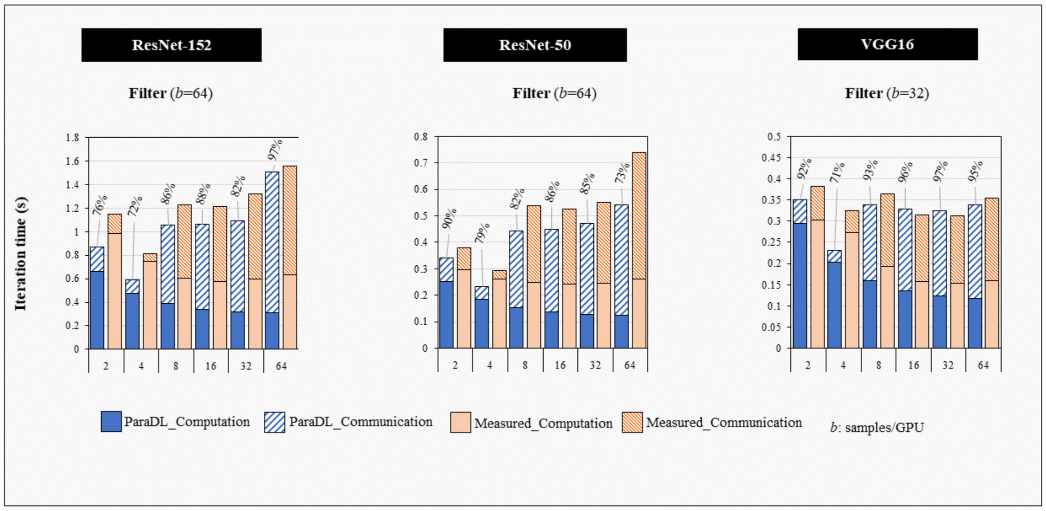


Figure 4.10: Prediction Accuracy of ParaDL with Imagenet for Filter Parallelism

configurations is because the sample size is so large that it cannot run with any other parallel strategy.

The accuracy of ParaDL predictions for the different parallel strategies are 96.10% for data parallelism, 85.56% for Filter, 73.67% for Channel, 91.43% for Data+Filter, 83.46% for Data+Spatial and 90.22% for pipeline across all CNN models. In general, this represents an overall accuracy of 86.74% for ParaDL, across all parallelism strategies and CNN models, and up to 97.57% for data parallelismon VGG16. Cosmoflow shows an accuracy of 74.14% on average.

It is important to note that the overall average accuracy drops significantly due to some few outliers in which the communication time measured is substantially higher than the prediction from ParaDL. For instance Data+Spatial for ResNet-152 with 512 GPUs shows an accuracy of 62% due to network congestions. Interestingly, the same configuration with 1024 GPUs shows a much higher accuracy (i.e., 83% for Data+Spatial ResNet-152) including the communication part, demonstrating that the ParaDL oracle is highly accurate, even at large scale (i.e., 1024 GPUs). Section 4.6.3 includes a detailed analysis of the network congestion leading to the few outliers where the machine was oversubscribed. Note that the communication time of ParaDL for Data+Filter shown in Figure 4.12 is calculated with a contention penalty coefficient of $2\times$, e.g., contention caused by two disjoint Allreduces that share the same InfiniBand link for inter-node communication. The high accuracy reported show that our analytical model fits well to the real performance.

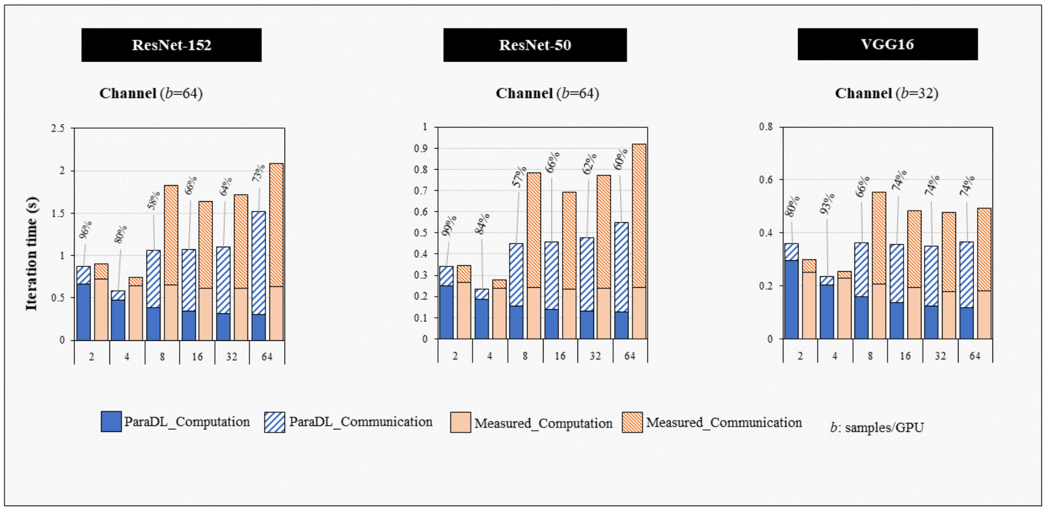


Figure 4.11: Prediction Accuracy of ParaDL with Imagenet for Channel Parallelism

4.6.3 Parallelism Limitations and Bottlenecks

In this section we use a combination of observations from ParaDL projections and empirical results to highlight some important points: (i) inherent to the parallel strategies themselves (limitations), and (ii) those caused by other components such as the framework implementation or system architecture (bottlenecks). This helps users in avoiding these limitations, and framework programmers in prioritizing their efforts for improvements. We group these limitations and bottlenecks into four categories.

Communication

It is well-known in literature that parallel training introduces communication overhead. Those overheads have different forms and patterns for different parallel strategies. There is the gradient exchange at the end of each iteration in data and spatial parallelism (GE-Allreduce). There can also be extra communication in the forward and backward passes of other parallel strategies: the layer-wise collective communication in filter/channel (FB-Allgather and FB-Allreduce), layer-to-layer communication in pipeline (FB-layer), and the halo exchange in spatial (FB-Halo).

Gradient Exchange: Similar to data parallelism, spatial requires a gradient exchange to aggregate the weights. This collective communication, i.e., Allreduce, has significant impact on performance, and can become a limitation. Another point worth mentioning is

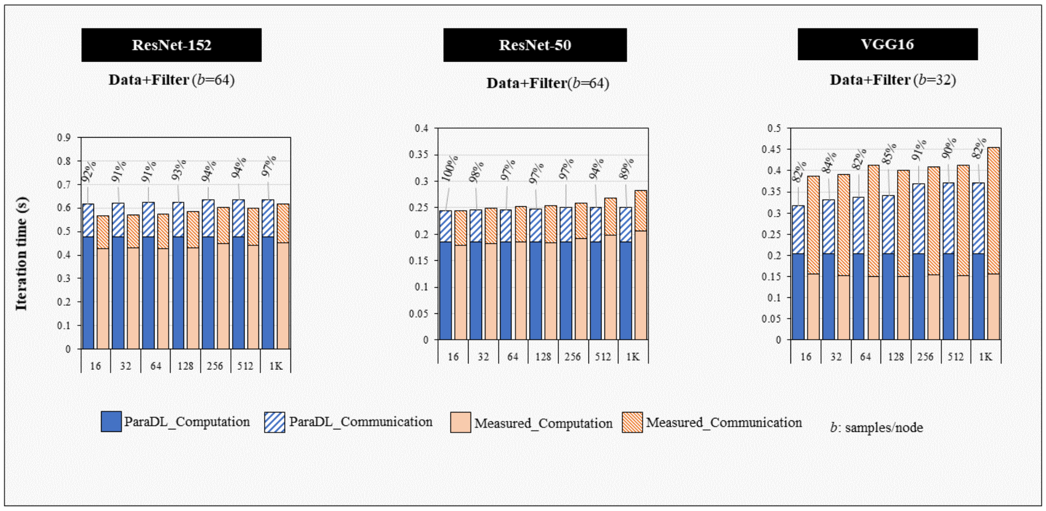


Figure 4.12: Prediction Accuracy of ParaDL with Imagenet for Data+Filter Parallelism

the hierarchal implementation of Allreduce in hybrid strategies such as Data+Filter (*df*) and Data+Spatial (*ds*). These two types of Allreduce (data and hybrid parallelism) are different as they involve different parallelization techniques. In *ds*, we perform a reduce inside each node to leader GPUs first, then perform a global Allreduce between the leaders. However such implementation leads to a higher overhead, e.g., time for Allreduce is more than $2\times$ as those of data. Alternative ways to address this issue are to use multiple leaders instead of only one [70] or to use segmented allreduces, i.e., smaller, concurrent allreduces among disjoint sets of GPUs [76]. We use the former strategy for *ds* and the latter for *df*. These methods are not trivial to implement and they require significant engineering effort. ParaDL has proven accurate at modeling those communications and can be used to choose an implementation strategy based on ParaDL's projected communication times.

Layer-wise collective communication: unlike data parallelism, filter and channel parallelism require multiple collective communication rounds at each layer. The communication time depends on the activation size $\times$ batch size, i.e., $\mathcal{O}(B\sum_{l=1}^{G}|y_l|)$, as well as the depth of DL model, i.e., $\sum_{l=1}^{G}(p-1)\alpha$ as reported in our analytic model. In our experiments with ImageNet, even though the total activation sizes are smaller than the number of weights, yet with a batch size of ≥ 32 samples, the communication time of filter/channel is larger than that of data parallelism (See Figure 4.11 and 4.10) . Note that because this communication overhead is attributed to the forward and backward phases, Allreduce optimization techniques such as sparsification are no-longer valid. Instead, a hybrid which combines

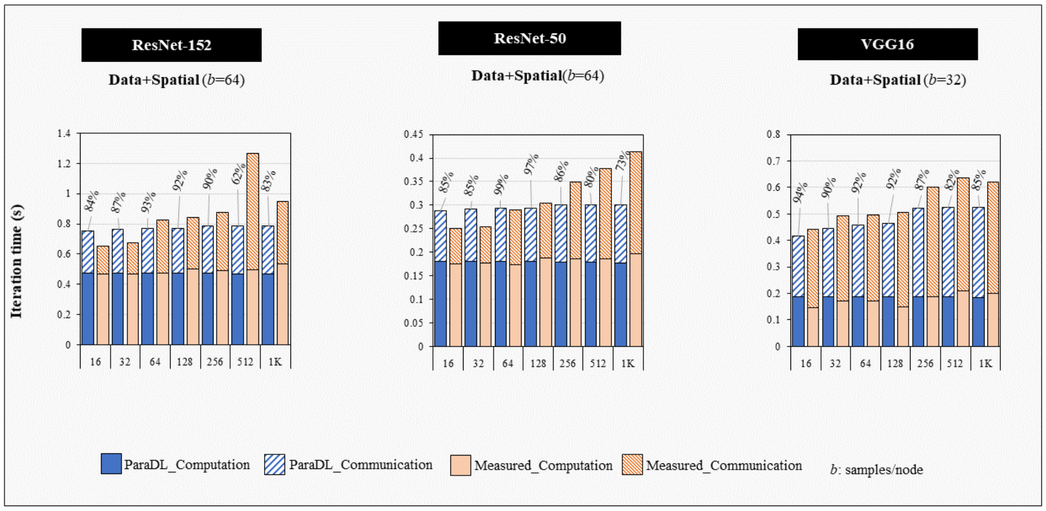


Figure 4.13: Prediction Accuracy of ParaDL with Imagenet for Data+Spatial Parallelism

filter and channel (plus data) parallelism may help to mitigate this limitation by reducing the number of communication calls [100] or using a smaller segmented Reduce-Scatter [76].

Peer to Peer communication: The halo exchange in spatial parallelism and the activation passing between composite layers in pipeline are performed in a P2P fashion, which are expected to have small communication times. However, paraDL shows that the communication time of FB-Halo is non-trivial and this was confirmed by the empirical results. For example, in ResNet-50, 128 GPUs, the time of FB-Halo is approximately 60% of the gradient exchange Allreduce, which is substantially higher than initially expected. This bottleneck appears because the framework uses the MPI library instead of NCCL (NCCL allows GPUs to communicate directly instead of via CPU, i.e., GPUDirect). We plugged different network parameters in paraDL (See Section 4.5.4) for MPI and NCCL and we confirmed the difference, both theoretically and empirically.

Network Congestion: In our empirical experiments, we try to avoid the issue of congestion as much as possible by running several times for each data point. However, as shown in some results, we still observe network congestion when approaching 1K GPUs. We did a detailed analysis for several of the runs. In Figure 4.17 we show the time for Allreduce communication for data parallelism of ResNet-50 with 512 GPUs and an Allgather communication for filter parallelism of VGG16 with 64 GPUs. We noticed that most data points align well with the expected theoretical bandwidth predicted by our analytical model (blue line), yet network congestion caused by other jobs in the system can lead to some outliers that push the average communication time up to four times higher

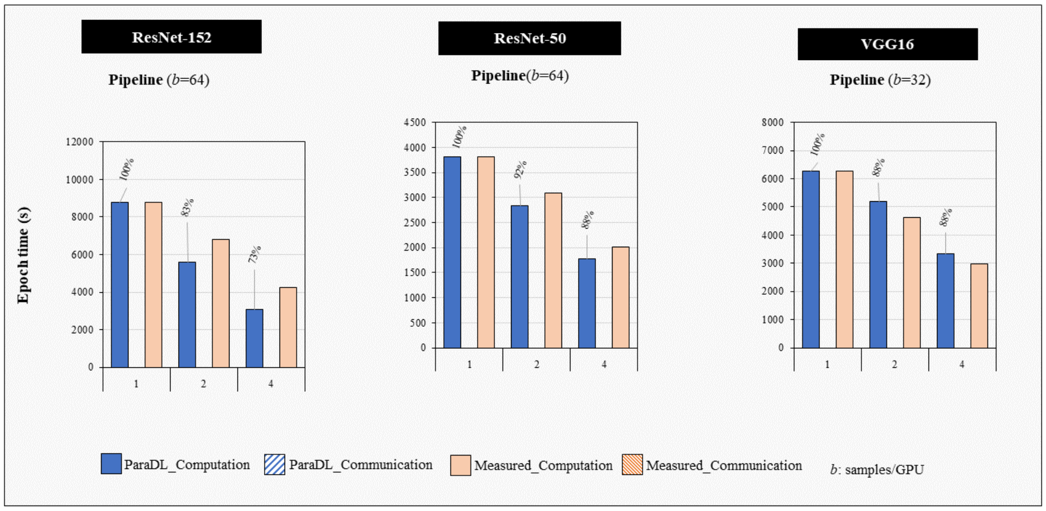


Figure 4.14: Prediction Accuracy of ParaDL with Imagenet for Pipeline Parallelism

than expected. This overhead can not be avoided especially for large-scale training, e.g., 100s-1000s GPU, in a shared HPC system. It could, however, be mitigated at the system level by switching to a full-bisection bandwidth rather than having 1:3 over-subscription.

Memory Capacity

We highlight specific cases when memory requirements become an issue in different strategies.

Redundancy in Memory: Different memory redundancies could emerge in different parallel strategies. In the spatial and channel/filter strategies the activations (i.e. input/output channels) are divided among nodes, however this does not reduce the memory requirements of holding the weight tensors since their weights are not divided among PEs (as in our analytical model). This becomes an issue for larger models. One alternative proposed in ZeRO [64] is to split the weights as well as the activations. However, this comes at the cost of extra communication of 50% since two Allgathers of the weights are needed in the forward and backward pass. In pipeline, the memory required for a single layer could be prohibitive. For example, ComsoFlow's first Conv layer generates more than 10GB of activation tensor when the input size is 4×512^3. Accordingly for those kind of models the pipeline strategy would be unfeasible and one has to resort to other parallel strategies (e.g., use Data+Spatial for CosmoFlow as shown in Figure 4.20). Additionally, since samples are feed in a pipelined fashion, the memory required is proportional to the

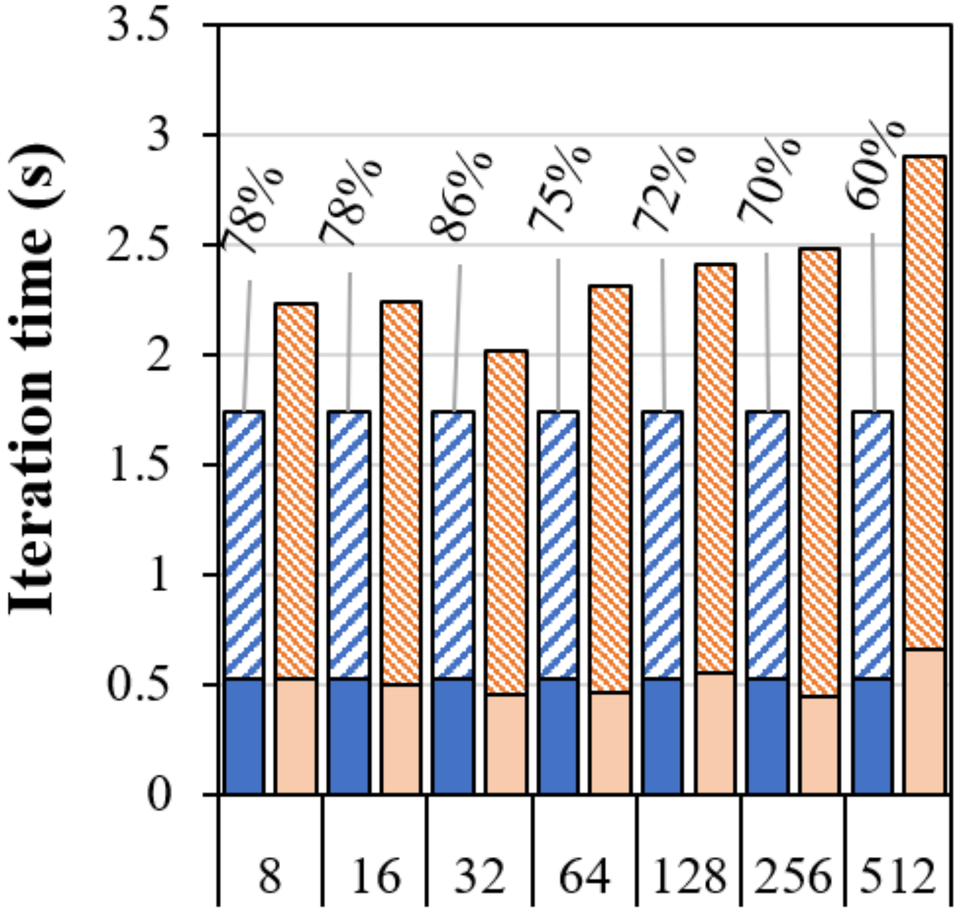


Figure 4.15: Prediction Accuracy of ParaDL with CosmoFlow for Data+Spatial

number of stages, and would hence become a bottleneck in deep pipelines, unless we apply gradient checkpointing at the boundary of the partition [15, 113], which comes with the overhead of recomputing the activations within each partition.

Memory Manager: DL frameworks typically include a memory manager to reduce the overhead of frequent malloc and free. Since GPUs typically operate in asynchronous execution model, there are many CUDA kernels being launched at any given time in training. We observed a disparity between ParaDL and measured performance that could be attributed to stalling kernels. Upon inspection, we observed kernels that were launched asynchronously often stall when requesting memory (in, Chainer as well as PyTorch). The launched asynchronously launched kernels waiting for memory to be available leads to heavy fluctuations in performance. Guided by ParaDL to identify the fine-grained location of the performance gap, we confirmed, for instance, that the implementation of

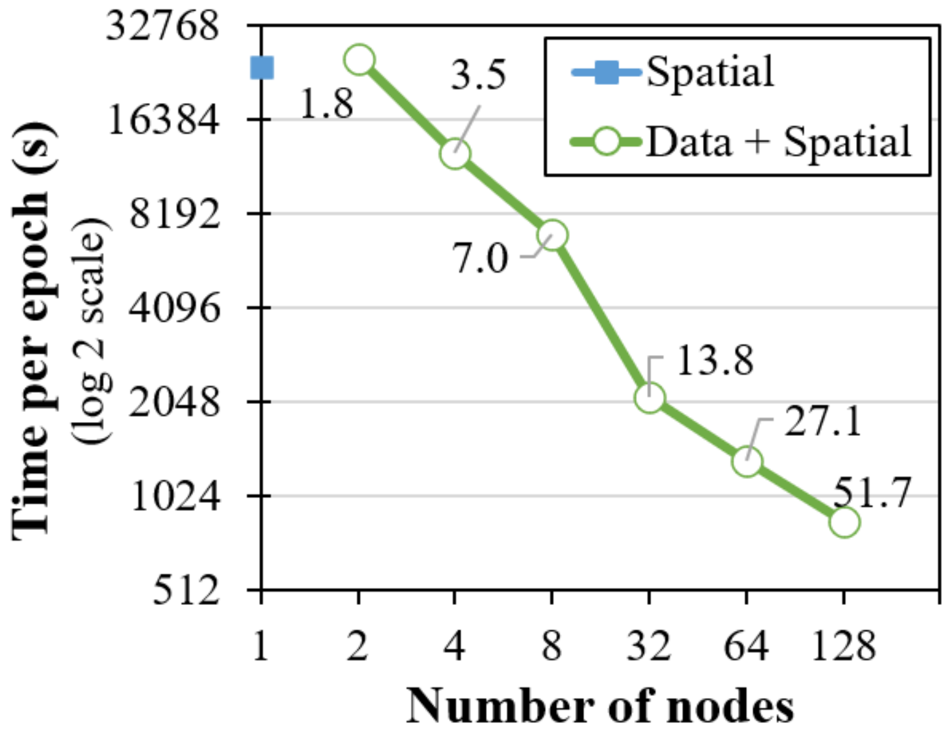


Figure 4.16: Spatial + data scaling with CosmoFlow. The labels show the speedup ratio of spatial+data over the pure spatial strategy

Data+Spatial parallelism of VGG16 (64 GPUs) could avoid out-of-memory issues at the cost of a performance degradation of 1.5$\times$.

Computation

We highlight the following limitations. *Weight update*: most compute time in training typically goes to the forward and backward pass. However, we observed with our analytical model that for larger models the weight update starts to become a significant portion of the compute time. For instance, we measured weight update to take up to 15% for the VGG16 (shown in Figure 4.18). Larger DL models, using ADAM optimizer in specific, may need a higher computation time for weight. Especially, large Transformer based models report up to 45% time on weight update and more than 60% extra memory requirements since ADAM requires four variables per weight [90]. One alternative to address this is to shard the weight update among GPUs across iterations, and Allgather the weights before forward/backward passes [114].

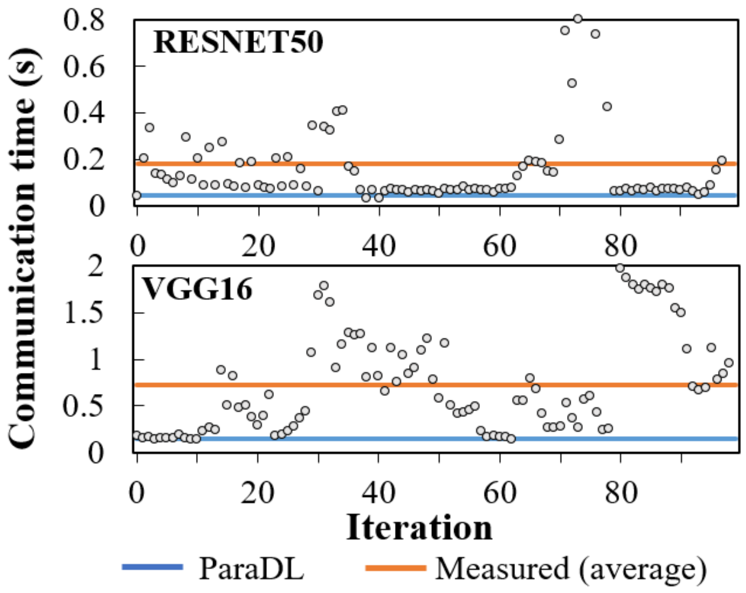


Figure 4.17: Network congestion of ResNet-50, 512 GPUs, data parallelism (upper) and VGG16, 64 GPUs, filter parallelism (lower).

Not only FW and BW require huge computation. For a big model, WU become a problem. This points related to all the strategies. NEED a figure show the relative time of (weight update) over the total time. (Figure 4.18).

Workload Balancing: Pipeline can outperform data parallelism with less communication by using P2P rather than a collective communication. However, it is crucial that all stages in the pipeline take roughly the same amount of time, since the training time of a pipeline is limited by the slowest stage. Indeed, there may be cases where no simple partitioning across the GPUs achieves perfect load balance (e.g. networks with non-linear connections). To further improve load balancing, a straight forward approach is to use data parallelism inside a stage, i.e., hybrid of pipeline plus data [113].

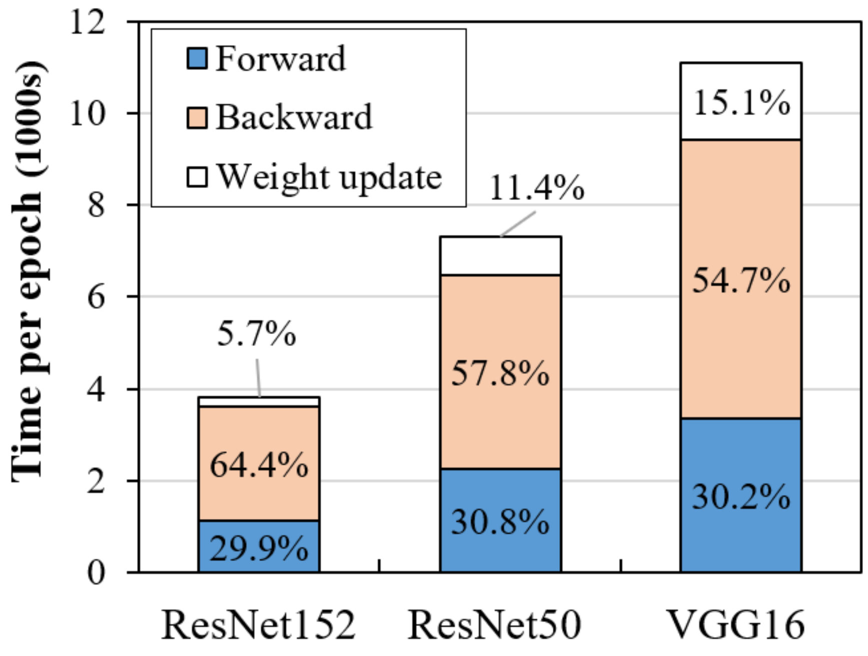


Figure 4.18: Computation time per epoch with PyTorch. Weight update is not trivial in large models and dataset.

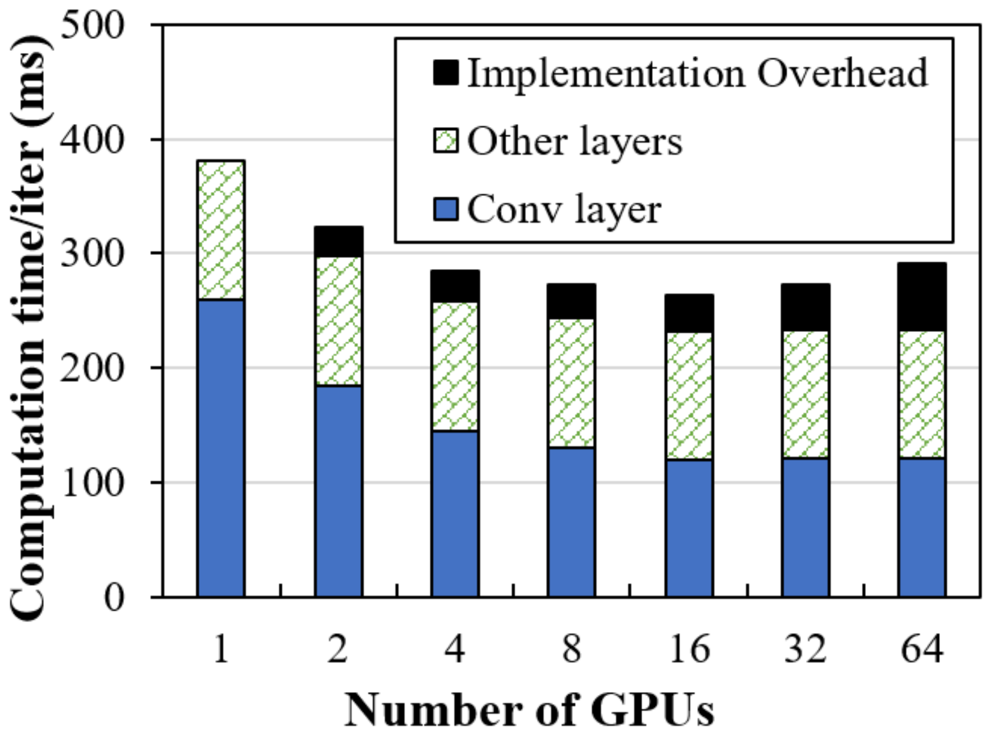


Figure 4.19: Computation breakdown of filter parallelism, ResNet-50. Implementation of convolution layers does not scale well.

Computation Redundancy: This section discusses limits that could arise from computational redundancy that is introduced for different parallel strategies. Using ParaDL we found out that there was a gap between analytical result and the measured time in filter/channel. Looking in detail, we saw that this was an implementation issue in the framework including two factors i) the convolution layer does not always scale as expected and ii) the computation overhead, such as split/concat, is non-trivial. These non-trivial implementation overheads are shown in Figure 4.19.

Scaling limitation

When scaling, there is a limit on the number of GPUs for each of the model parallel strategies (last column of Table 4.3). For example, p can not exceed the minimum number of filters of a layer in the model, i.e., 64 in the case of VGG16 and ResNet-50 with filter parallelism. Hybrid approaches have a better scaling than those of pure model parallelisms. For example, as shown in Figure 4.20, using Spatial+Data hybrid is an effective scalable alternative (despite communication inefficiencies), since it scales both in performance and GPU count (i.e. one could simply expand the data parallel pool as much as new nodes are added). Indeed, the curve shows a perfect scaling (note the logarithmic y-axis).

4.6.4 Other Observations

We briefly discusses further notable point of distributed DNNs.

The Rise of Hybrid Parallelism

As mentioned, each of four basic parallelism strategies has its own limitations. Using the hybrid strategies (Data+Model) helps to break/mitigate those limitations, e.g., memory issue of data parallelism and communication and scaling limitation of model parallelism (Section 4.6.3). As more datasets from the HPC field start to be trained by DL frameworks, this type of hybrid parallel strategies will become increasingly relevant because data parallelism will simply be not enough, as shown in the case of CosmoFlow and its good scaling with *ds* in Figure 4.20. In addition, hybrid strategies may drive to a better solution in terms of performance. In accordance with other recent reports [76], there are cases where data+filter (*df*) hybrid can outperform data parallelism at large scale, as we also observed in some of our experiments (we also noticed scenarios where pipeline outperforms data parallelism).

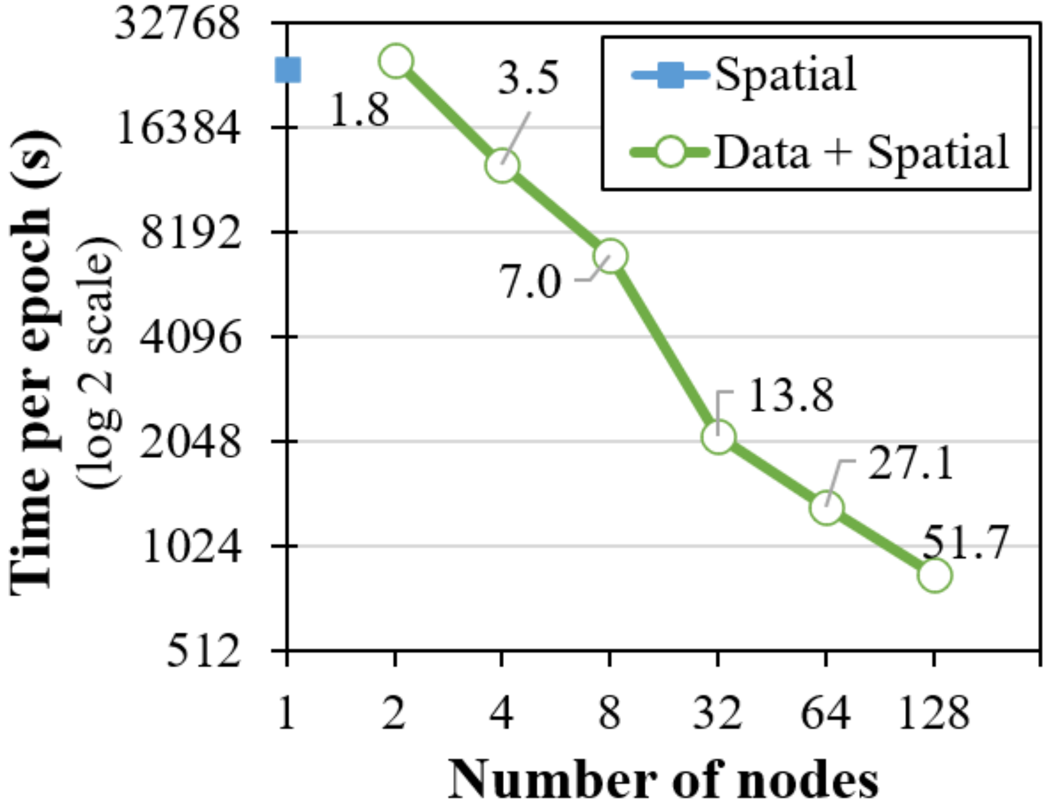


Figure 4.20: Spatial + data scaling with CosmoFlow. The labels show the speedup ratio of spatial+data over the pure spatial strategy.

Staging and I/O Overhead

Figure 4.21 shows the staging overhead per iteration of different cases of (DL model, parallel strategy). Because each case loads different number of samples/GPU, we expect to see differences between ResNet-50 and VGG16 (64 vs. 32 samples) and a similar difference between ResNet-50 and ResNet-152 (both use 64 samples). We noticed during our experiments that in the spatial and filter parallelism implementations, the samples are loaded on a leader process, which then distributes (part of) the samples to the other GPUs. This staging process evidently adds a significant amount of time, especially in a large-scale run or when the sample sizes are large. For example, I/O time can be as high as $6.7\times$ of communication and computation times in the case of CosmoFlow. Note that this is not a fundamental limitation of spatial/model parallelism, but simply an implementation issue of the data loaders of the underlying framework. (i.e., Chainer). Other frameworks (e.g., TensorFlow, PyTorch) also have the same issue.

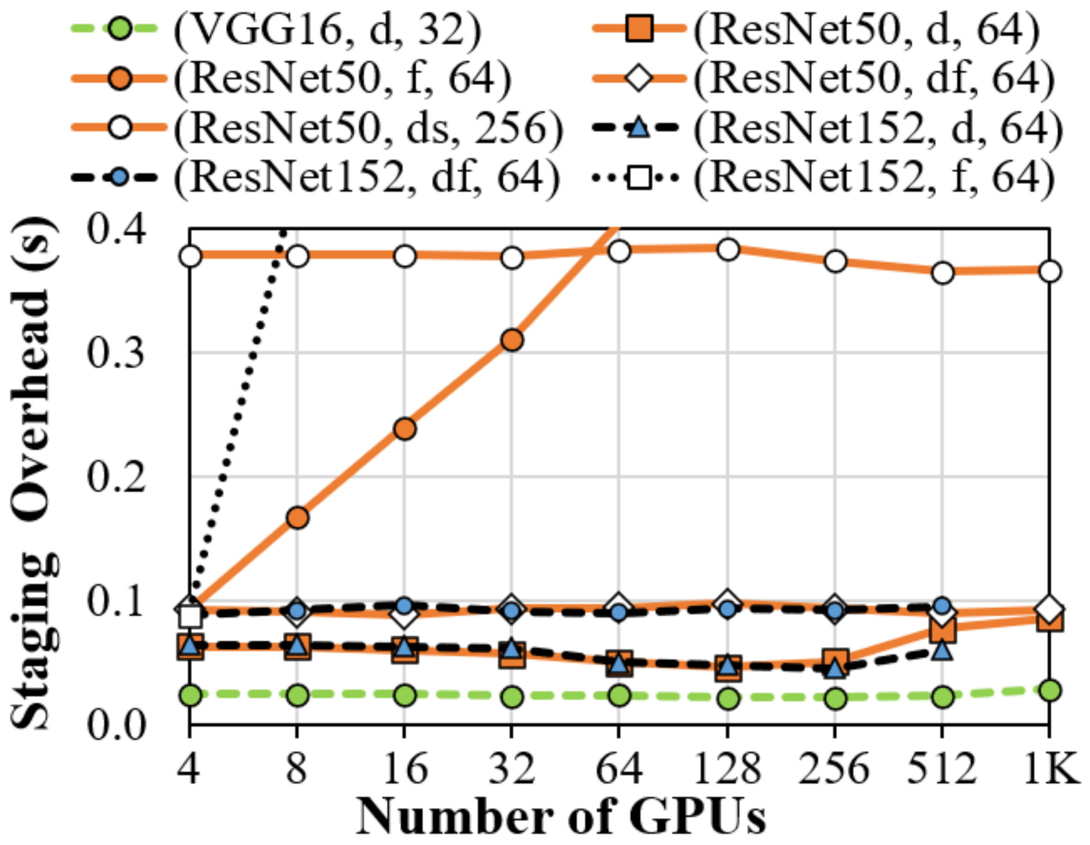


Figure 4.21: Staging overhead. Labels show (model, parallel strategy, samples/GPU). d:data, f:filter, df:(data+filter), ds:(data+spatial).

Distributed Inference

Inference at large scale is becoming increasingly demanded, given that for large models the inference would also be distributed [65]. In smaller models, when latency of inference matters in an application, the inference could also be distributed (e.g., real-time prediction of Tokamak disruptions in magnetically-confined thermonuclear plasma experiments [115]). Some of the limitations and bottlenecks of distributed training discussed previously also appear in distributed inference (See lines marked with **Y** in column **I** of Table 4.6).

4.6.5 Summary of our Analysis

In this section we summarise our main findings. A compact version of the limitations and bottlenecks that paraDL helped us uncover is summarised in Table 4.6. Those limitations/bottlenecks in the table may appear (Y/N) in distributed inference (**I**). Related parallel strategies ($\times$): **d**-data, **s**-spatial, **p**-pipeline, **f/c**-filter and channel, **df**-hybrid Data+Filter, **ds**-hybrid Data+Spatial. **FR**-Framework, **SY**-System. Training phases: **IO**-I/O and pre-processing, **FB**-a forward and backward propagation, **GE**-the gradient exchange (if needed) and **WU**-updating the weights.

- Our analytical model projects ideal performance and gives consistent accuracy (over 85% in average) on a wide range of models and datasets on up to 1K GPUs.
- We analytically identify different bottlenecks that can appear in different parallel strategies due to communication patterns that compensate for different ways to split the tensors, and confirm those predictions empirically (See Table 4.6).
- We identify memory and computational pressure that arise from different redundancies in different parallel strategies.

Category	**L/B**	**Para. Strategies**						**FR**	**SY**	**Training Phases**				**I**	**Remarks**
		d	**s**	**p**	**f/c**	**df**	**ds**			**IO**	**FB**	**GE**	**WU**		
Communication	L	×	×	-	-	×	×	○	○	-	-	✓	-	N	Gradient-exchange
	L	-	-	-	×	×	-	○	○	-	✓	-	-	Y	Layer-wise comm.
	B	-	×	×	-	-	×	●	○	-	✓	✓	-	Y	P2P communication
	B	×	×	×	×	×	×	○	●	-	✓	✓	-	Y	Network Congestion
Memory Capacity	B	×	×	×	×	×	×	○	●	✓	✓	✓	✓	Y	Memory Redundancy
	B	×	×	×	×	×	×	●	○	✓	✓	✓	✓	Y	Memory Stalling
Computation	L	×	×	×	×	×	×	○	○	-	-	-	✓	N	Weight Update
	L	-	-	×	-	-	-	○	○	-	✓	-	✓	Y	Workload Balancing
	B	-	-	-	×	×	-	●	○	-	✓	-	✓	Y	Comp. Redundancy
Scaling	L	×	×	×	×	×	×	○	○	✓	✓	✓	✓	Y	Number of PEs

Table 4.6: Summary of detected limitations (**L**) and bottlenecks (**B**) with the related training phases (✓) and components (●).

4.7 Conclusion

In this chapter, we propose an analytical model for characterizing and identifying the best technique of different parallel strategies for CNN distributed training. We run a wide range of experiments with different models, different parallel strategies and different datasets for up to 1,000s of GPUs and compare with our analytical model. The results demonstrate the accuracy of ParaDL, as high as 97.57% , and 86.74% on average accuracy across all parallel strategies on multiple CNN models and datasets on up to 1K GPUs.

The analytical model helped us uncover limitations and bottlenecks of parallel training of CNNs (summary in Table 4.6). We analytically identify different bottlenecks that can appear in different parallel strategies due to communication patterns that compensate for different ways to split the tensors, and confirm those predictions empirically. Finally, we identify memory and computational pressure that arises from different redundancies in different parallel strategies.

Chapter 5

A Study of Checkpointing in Large Scale Training of Deep Neural Networks

In this chapter, we examine the checkpointing implementation of popular DL platforms. We perform experiments with three state-of-the-art DL frameworks common in HPC: Chainer, PyTorch, and TensorFlow. We evaluate the computational cost of checkpointing, file formats and file sizes, the impact of scale, and deterministic checkpointing. Our evaluation shows some critical differences in checkpoint mechanisms and exposes several bottlenecks in existing checkpointing implementations. We provide discussion points that can aid users in selecting a fault-tolerant framework to use in HPC. We also provide take-away points that framework developers can use to facilitate better checkpointing of DL workloads in HPC.

5.1 Introduction

While distributed training in HPC clusters can reduce weeks of training to days or even hours, these distributed architectures are susceptible to unrecoverable hardware and software failures that can ruin days or even weeks of training time. To mitigate these failures, most DL frameworks implement checkpointing as fault tolerance mechanisms to save the training state at a certain point. In case of failure, training is restored from the last valid checkpoint. However, checkpointing in DL is not just a defence mechanism against failures. Transfer learning, a common technique in DL relies on checkpointing. A model is trained on a different data set, saved, and finally fine-tuned with the target application data.

This significantly saves on training time and resources. Recently, gradient checkpointing has been used to overcome memory constraints by fitting large models in GPU memory and the cost of increased computational cost. Checkpointing is, therefore, a fundamental component of training DL models and even more important when DL such models are trained in HPC.

Despite being such a critical component of training DL models, checkpointing has largely been ignored by the DL community and checkpoint implementations of DL frameworks are not ideal for HPC. Related work (See section 5.2) shows that there is a need for advanced checkpointing mechanisms for DL workloads, especially in HPC. However, to improve existing mechanisms or create new ones, an in-depth study and evaluation of checkpointing in large scale training of DNNs is required. While checkpointing provides a way to restart applications on failure, it is important to ensure that on a restart, the behaviour of such applications is maintained. This ensures reproducibility and validation. Furthermore, it is important for researchers who want to analyze the behaviour of DL applications under particular circumstances [116].

In this chapter, we provide detailed explanations of how checkpoint mechanisms work to illustrate the design decisions involved. We then evaluate checkpointing using representative DL models with three state-of-the-art DL frameworks common in HPC: Chainer, PyTorch, and TensorFlow. We use Cifar [38] dataset because it provides an adequate number of images that allow DL training to finish within acceptable execution times therefore ideal for experimentation. In addition, this dataset allowed us to properly analyze the performance of distributed training when scaling on GPUs. We further study deterministic behaviours in these frameworks to validate reproducibility when checkpointing. In summary, the contributions of this chapter can be summarised as follows.

1. We explore, explain, and compare checkpoint mechanisms of distributed computing DL frameworks.

2. We perform a set of experiments to measure and evaluate checkpoint overhead at different scales.

3. We study the deterministic execution of DNN training. Such experiment is fundamental for reproducibility.

The remainder of this chapter is organised as follows. Section 5.2 gives a detailed background of distributed DL and different frameworks used in DL. Section 5.4 explains our experimentation methodology. Section 5.5 shows our evaluation and results. Section 5.6 discusses important points for future improvement and Section 5.7 concludes the chapter.

5.2 Background

In this section, we provide a background and related work in checkpointing for Deep Learning.

5.2.1 Checkpointing in DNN Training

As mentioned before, almost all frameworks provide some implementation of checkpointing. In most of the frameworks, the implementation checkpoint is straightforward. However, the implementation is not automatic and depending on the DL framework it may involve more modifications in the code to incorporate them. In the following subsections, we discribe these implementations from the frameworks used in this section.

Chainer Checkpoint implementation

Chainer implements checkpoints based on the extension `chainer.training.extensions.snapshot()`. This extension allows the serialization of a trainer object to save it to an output file. By using this extension, it is possible to resume training from a saved state. The extension uses as one of its parameters a trigger that tells Chainer when to perform the checkpoint, e.g. `trigger ̄(1, 'epoch')` indicating that the action is executed on each epoch. This extension must be added as an extension to the Chainer trainer through the `chainer.training.Trainer.extend()` function. In order to load a checkpoint, it is necessary to use the `chainer.serializers.load_npz()` function before the execution of the training begins.

For serialisation and deserialization processes, there are specific functions called `save_npz()` and `load_npz()` respectively. These functions can be considered of coordination as a prelude to the implementation of the serialization and deserialization processes. The serialisation process requires extra manipulation, because it is first necessary to pack the objects in a flat dictionary (`DictionarySerializer()`) and then serialize it (`numpy.savez()`). For the deserialization process, the `NpzDeserializer()` class can only be used to deserialise objects saved with the `save_npz()` function. The final storage process is handled by NumPy functions for serialisation (`numpy.savez(), numpy.savez_compressed()`) and deserialization (`numpy.load()`). The `numpy.savez_compressed()` function can be used if the compression of NPZ files is required. It is important to highlight that the NumPy serializers internally use the Pickle library, which could generate a security problem due to the fact that when loading pickled data, it is possible to execute arbitrary code. Therefore, `numpy.load()` restricts the loading of array objects stored in NPZ files. Chainer internally enables pickle load so that the checkpoint process can be carried out transparently.

PyTorch checkpoint implementation

At high level PyTorch provides the `torch.save()` and `torch.load()` functions to save the current state (training state checkpoint) and to load a saved checkpoint, respectively. The `torch.save()` function uses as a parameter a Python dictionary with the information to save. This information can vary according to needs and in this case the dictionary is loaded with information from the optimizer (`optimizer.state_dict()`) and the neural network state (`model.state_dict()`). In the same way, when a checkpoint is loaded, the function returns a dictionary with the previously saved data. It is necessary to carefully put the save and load functions in the code and ensure that only one process (rank 0) can execute the function. Furthermore, it is important to ensure that the checkpoint operation is performed after the all-reduce operation is executed. In our case, at the end of each epoch.

PyTorch implements serialization processes to convert the data and structures of the neural network to a byte stream that allows them to be stored and retrieved to their original structure. Within the function `save()` and `load()` there are two main functions that can be called according to the type of object to be serialized and deserialized, respectively. The `_save()` function can be used to serialize using zip files. However, in our experiments, the serialization is performed with the `_legacy_save()` function that creates a binary file without compression. The same argument applies to the `_load()` and `_legacy_load()` functions. However, it is important to note that the `load()` function first deserializes the data in the CPU, and then it is moved to the device from where it was saved. However, the data can be dynamically moved to another device using the `map_location` argument.

TensorFlow Checkpoint implementation

TensorFlow somehow shares a similarity with Chainer in performing the checkpoint process. This is because TensorFlow uses the callback mechanism to interact with the training process, similar to the extensions that Chainer uses for the same purpose. TensorFlow uses a callback called `tf.keras.callbacks.ModelCheckpoint()` to implement the checkpoints. Through this callback, it is possible to configure parameters such as the frequency of the checkpoints (`save_freq`) or if the complete model is stored or only its weights (`save_weights_only`). Furthermore, this callback has an interesting parameter that allows us to save the best model (`save_best_only`) by monitoring (`monitor`), either accuracy or loss.

The serialisation process can result in two types of file format, one native to TensorFlow and one HDF5. To get the files in HDF5 format, TensorFlow uses the H5PY library as part of the serialisation process. This differs from PyTorch in that the H5PY library is not implemented by default. TensorFlow, by default, saves files in their native format and to indicate that a file in HDF5 format is needed only requires setting the file extension of the output file to $.h5$. In case the callback input argument is `save_weights_only = false` TensorFlow will create a checkpoint of the entire model saving the model architecture (to allow to re-instantiate the model), the model weights, and the state of the optimiser to allow you to restart a training exactly where it left off. On the other hand, the deserialisation process does not require callbacks because the checkpoint is loaded before starting the training. TensorFlow can load only saved model weights with the `tf.keras.models.load_weights()` function or the entire saved model with the `tf.keras.models.load_model()` function.

5.2.2 Deterministic Behaviour of DNN Training

There is intrinsic randomness in the training results even with the same infrastructure (hardware, framework versions, etc.) and model configurations. This can raise reasonable questions about the reliability of the results after restarting a saved checkpoint. In order to validate the appropriate functioning of the checkpoints, we want to obtain a deterministic behaviour after a restart. This means that accuracy and loss values after a restart are precisely the same as those of a complete training (we call this *deterministic restart*).

To remove non-deterministic behaviour from the three DL frameworks, we added instructions that disable the randomness of the internal processes of the DL framework and of the libraries external to the DL framework (e.g. CuDNN, NumPy, or CuPy). Therefore, it is necessary to set the seed of the instructions that generate randomness to an equal and constant value when starting the DL applications. With this, it is possible to generate identical random sequences used by the DL frameworks' internal processes. Because distributed training use GPUs, some instructions to control randomness are related to CUDA and cuDNN [117] libraries. Also, it was impossible to implement deterministic restart in both Chainer and TensorFlow since training information is encapsulated into a single container. Such design decision makes it highly cumbersome to adapt either Chainer or TensorFlow for a deterministic restart (See Section 5.5.4).

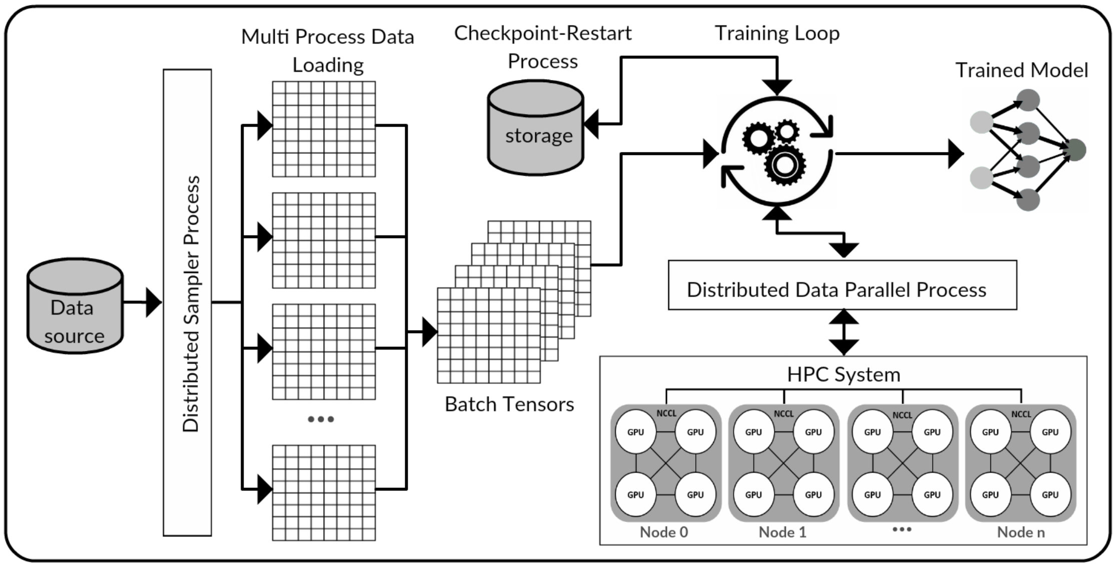

Multi Process Data Loading
Checkpoint-Restart Process
Training Loop
Trained Model
storage
Data source
Distributed Sampler Process
Batch Tensors
Distributed Data Parallel Process
HPC System
NCCL
GPU
Node 0
Node 1
Node n

Figure 5.1: Distributed Training Process.

5.3 Related Work

Over the years, researchers have focused on improving the performance of distributed training of DL models without taking into account aspects related to the effectiveness or optimization of existing checkpoint mechanisms. For example, there are studies like [118, 119, 120, 121] focused on optimizing storage and I/O performance because these become a bottleneck limiting the overall system scalability. Other studies such as [122, 123, 124] focused on optimization techniques related to SGD (Stochastic Gradient Descent) algorithm in multinode distributed environments with the improvement of not altering hyperparameters and without compressing data. Other researchers [125, 126, 127, 128] focused on the main bottlenecks that exist to scale distributed training, namely model distribution overheads. These studies were focused on the NCCL broadcast, the performance bottlenecks CUDA-Aware MPI runtime, and the limitation of the GPU approach in distributed training when the model does not fit in GPU memory.

On the other hand, some researchers [129, 130, 131, 132, 133] focused their studies on the fault-tolerant of the neural networks but considered faults degrade the performance of the DNN application and not catastrophic faults that spoil the execution of a DNN application completely. These investigations analyze the propagation of the soft errors from the hardware to the DNN application, the algorithmic error-resilience of the DNN, the Byzantine failures of distributed implementations, and ideas from coding theory to guarantee robustness during distributed training. Other investigations such as [134, 42] examine the fault tolerance and recovery characteristics of parallel distributed networks.

Though checkpointing in DL is not extensively studied, there is some increasing interest to optimise this process. Some investigations analyse different aspects in which the checkpoint is related both directly and indirectly to DL processes. In [135] they make use of a checkpoint library used in HPC applications to solve the limitations (low serialisation performance, blocking I/O, only one process performing the checkpoint process) of the simple checkpoint techniques applied in DL. The authors propose a checkpoint framework that acts as a bridge to advanced techniques used in multi-level HPC checkpointing, taking advantage of I/O patterns, layer-wise parallelism, and synchronous data-parallel training properties. The framework is implemented on top of the VeloC [136], a multi-level checkpointing tool for HPC systems. Qiao et al. [137] quantify the impact of faults on interactive convergent algorithms, and they proposed strategies based on checkpointing to tolerate faults in distributed training. They argue that it is possible to significantly reduce the rework cost due to partial failures with partial recovering combined with prioritising checkpoints to reduce the size of perturbations. As a result of this study, they showed decreased rework cost of failures by order of magnitude compared to traditional checkpoints.

Other studies use checkpoints not as fault tolerance mechanisms but rather as mechanisms to optimize performance. For example, in the case of Wei et al. [138] checkpoints are used as a mechanism to optimize training in Neural Machine Translation (NMT). The authors argue that there are two main problems in training: first, the training process becomes unstable and second that validation performance worsens after several epochs. So, to mitigate these issues, they periodically evaluate the models, which are saved as a checkpoint. Training ends when several consecutive checkpoints show no improvement and the checkpoint with the best evaluation is selected. In this case, the checkpoint mechanism is used to lead the training process. Also, in [139] checkpoints are used to decrease memory usage when training deep neural networks with the back-propagation algorithm. They use checkpoint strategies to determine checkpoints that should be kept in memory and those that should be calculated from stored checkpoints during the execution of the backward phase.

In this chapter, we also consider the deterministic behaviour of DNN as a way to validate the proper functioning of checkpoints. There are few studies about the use of deterministic elements in machine learning [140, 141, 142, 140, 143]. These studies use deterministic approaches to generate benchmarks or algorithms to increase machine learning-related applications' performance or reliability. Also, in [140] one of the main goals is the reproducibility of the results. Another study [144] is focused on the analysis of positive impacts of deterministic implementations in training. They identify the sources of nondeterminism in DL applications and some experimental conditions that form a gap between deterministic implementations and replicability.

5.4 Methodology

Fault tolerance in DL has been largely ignored by the community [135]. Perhaps due to the fact that, traditionally, most of the research was carried out on a single GPU or single node multi GPU environment. Furthermore, most of the DL frameworks are optimised for performance. However in HPC, fault tolerance is a fundamental component because faults and node failures are common with libraries such as FTI [145] built specifically to facilitate fault tolerance of applications in HPC.

Figure 5.1 shows the general structure of a multi-GPU training or distributed training with the implementation of the checkpoint process as a fault tolerance mechanism. In the diagram, the interaction between the checkpoint-restart mechanism and the training cycle is visible. It gives us an idea of the behaviour of the checkpoint-restart and its relationship with the performance of distributed training.

5.4.1 Evaluation Environment

We aim to evaluate existing support for fault tolerance in state-of-the-art DL frameworks and their readiness for HPC. This information is important for framework designers and users alike. For framework designers, our aim is to test, evaluate and validate existing implementations in HPC context. For users, our aim is to highlight key differences, similarities and trade offs in choosing frameworks vis-à-vis of fault tolerance. In a nutshell, we evaluate the following: $i)$ The computational cost of checkpointing, $ii)$ Efficiency of inbuilt checkpointing mechanisms in different frameworks, $iii)$ The effect of scale on checkpointing, and $iv)$ The deterministic behaviour in different frameworks. We perform this experiments using common machine learning models and datasets on two state-of-the-art HPC systems; Marenostrum and ABCI.

Power9 Cluster in the Marenostrum Supercomputer is a cluster of 52 nodes and each node is made up of 2 IBM Power9 processors and 4 NVIDIA V100 GPUs. Power9 processors can reach a frequency of 3 GHz, with 20 cores and 4 threads per core (160 threads per node). Each of the V100 GPUs has 16 GB HBM2 memory, with a performance of 7.8 teraFlops in double precision and 125 teraFlops with the tensor cores [146]. This cluster also has a high performance distributed file system (IBM GPFS), which allows access to data from all nodes in the cluster. Then, parallel applications can simultaneously access data from any node that has the file system. It is important to note that the Power9 cluster is an experimental system and many issues were still being evaluated by system administrators at the time of these experiments.

ABCI Supercomputer comprises 1,088 nodes of FUJITSU Server PRIMERGY CX2570 M4. Each compute node has two Intel Xeon Gold 6148 Processors and four NVIDIA Tesla V100 GPUs (16GB of memory per GPU). The GPUs are connected intra-node to the CPUs by PLX switches and PCIe Gen3 x16 links, and together by NVLink.

5.4.2 Experiments

Our experiments are largely divided into two groups. The first set of experiments is performed on Marenostrum, while the second on ABCI. Though these two systems share similar hardware characteristics, they are different on scale. As such, we are able to comprehensively evaluate both systems.

The first set of experiments evaluates the computational cost of checkpointing (Subsection 5.5.1), the checkpoint file size and format (Subsection 5.5.2), and deterministic behavior of the checkpoint-restart mechanisms of different frameworks (Subsection 5.5.4).We use the deterministic behavior to validate the restart functionality after saving a checkpoint and we determined the performance degradation this incurs. We use Horovod with PyTorch and TensorFlow to implement distributed training. These experiments were performed at Marenostrum4 supercomputer and up to 32 GPUs were used. Cifar10 was used as the dataset, ResNet50 with batch size of 64.

The second set of experiments is to evaluate the effect of scale on checkpointing, the behaviour of different models and how each framework performs at scale. These set of experiments were performed on the ABCI supercomputer using up to 256 GPUs. Our goal is to evaluate the computation cost of checkpointing at such scales and the performance of different frameworks. We use ResNet50 and VGG16 models. These two models have different characteristics that make them interesting to study. VGG16 is a deep and dense model with 138 million parameters. ResNet50 is a residual network with skip connections and 25.5 million parameters. We train both models on the Cifar10 dataset with a batch size of 32 (Subsection 5.5.3). Both sets of experiments were run in all cases with 100 training epochs. When training with checkpoint, checkpoint frequency was set to 5 epochs.

5.5 Results and Evaluation

In this section we present our large scale evaluation and the results of our experiments.

5.5.1 Computational Cost of Checkpointing

Saving the state of a DNN consumes a significant amount of time, in particular if it is done at high frequency. Table 5.1 shows the total time to complete a training without checkpointing, with checkpointing, and the percentage of overhead represented by NoCkpt, Ckpt, and Ω, respectively. We performed this experiment for the three DL frameworks, Chainer, PyTorch, and TensorFlow. For each of the frameworks, distributed training runs were executed with different number of GPUs, ranging from 4 GPUs (1 node) to 32 GPUs (8 nodes). In addition, each of the training runs was repeated 5 times and average time is reported.

As the number of GPUs increases, the performance improves accordingly, both for training with and without checkpoint. PyTorch had the best performing training times with a speedup of 6.4 with 32 GPUs without checkpoint and 6.2 with checkpoint. PyTorch is the framework that presents the best performance when scaling on GPUs. It is also interesting to note that training times with and without checkpoint in TensorFlow had a speedup of 3.5 for both training with and without checkpoint. Chainer obtained the lowest performance in these small-scale experiments. However, Chainer's performance excels at large-scale tests (See Section 5.5.3).

Regarding the checkpoint overhead, the lowest overall average overhead is obtained by TensorFlow with 2.3%, followed by PyTorch with 2.5%. Chainer has the highest overall overhead of 14.27%. The best performance in TensorFlow can be attributed to the use of the HDF5 file format, because the serialization process of this file format is highly optimized. Furthermore, it can be seen that Chainer is the only one with a constant increase in overhead, which is proportional to the number of GPUs going from 3.8% with 4 GPUs to 22.1% with 32 GPUs. The other DL Frameworks maintain a relatively constant overhead fluctuating in most cases between 1% and 3% for PyTorch and 1% and 2% for TensorFlow.

Finally, checkpoint mechanisms do not seem to alter the performance profile of the DL frameworks when scaling in small-scale experiments. Although the checkpoint overhead is not negligible, it does not susbstantially affect training performance for PyTorch and TensorFlow. In the case of Chainer, it is necessary to carry out a more detailed review of the checkpoint operation to optimize the mechanisms for the checkpoint process. In this study, the performance of the checkpoint in Chainer was not optimized to be consistent with the other DL frameworks.

	Chainer			PyTorch			TensorFlow		
GPU	**NoCkpt**	**Ckpt**	Ω	**NoCkpt**	**Ckpt**	Ω	**NoCkpt**	**Ckpt**	Ω
4	3119	3240	3.8	1801	1826	1.3	1107	1124	1.5
8	1726	1869	8.2	885	896	1.2	633	648	2.3
12	1283	1451	13.1	601	623	3.6	497	504	1.5
16	1013	1153	13.7	454	465	2.5	412	420	1.9
20	862	1006	16.7	374	380	1.6	371	373	0.3
24	762	902	18.4	310	320	3.1	329	351	6.8
28	699	824	17.9	288	299	3.7	325	329	1.4
32	633	773	22.1	278	294	5.5	313	322	2.9

Table 5.1: Distributed training time(in seconds) for 100 epochs. Columns show the execution times for a training without checkpoint (NoCkpt), with checkpoint (Ckpt), and the percentage of checkpoint overhead (Ω).

	Model	**Chainer**	**PyTorch**	**Tensorflow**
Size(MB)	ResNet50	146	180	181
	VGG16	238	1025	257
Format		NPZ	Pickle	HDF5

Table 5.2: Size and format of the checkpoint for different frameworks.

ResNet50									
	Chainer			PyTorch			Tensorflow		
No GPUs	Without Ckpt	With Ckpt	Overhead	Without Ckpt	With Ckpt	Overhead	Without Ckpt	With Ckpt	Overhead
4	2162	2338	8.14	1738	1801	3.62	1856	1936	4.31
8	1143	1295	13.30	933	941	0.86	1080	1082	0.19
16	600	747	24.50	484	493	1.86	602	618	2.66
32	307	446	45.28	253	259	2.37	364	371	1.92
64	157	302	92.36	140	142	1.43	236	248	5.08
128	83	228	174.70	77	84	9.09	174	186	6.90
256	47	190	304.26	48	54	12.50	149	157	5.37

Table 5.3: Training times(in seconds) for ResNet50 for 100 epochs in ABCI while checkpointing at every 5 epochs

VGG16									
	Chainer			PyTorch			Tensorflow		
No GPUs	Without Ckpt	With Ckpt	Overhead	Without Ckpt	With Ckpt	Overhead	Without Ckpt	With Ckpt	Overhead
4	647	880	36.01	1106	1130	2.17	813	817	0.49
8	796	1023	28.52	1322	1353	2.34	615	622	1.14
16	355	584	64.51	721	750	4.02	385	395	2.60
32	180	415	130.56	370	402	8.65	235	244	3.83
64	95	336	253.68	198	231	16.67	150	159	6.00
128	52	291	459.62	113	141	24.78	114	128	12.28
256	31	270	770.97	70	98	40.00	99	110	11.11

Table 5.4: Training times(in seconds) for VGG16 for 100 epochs in ABCI while checkpointing at every 5 epochs

5.5.2 Checkpoint File Size and Format

With the increase in complexity of the DL models, the size of the checkpoint files also increases, so the size of the files should be considered as a key element within the optimization of DL frameworks.With each framework that is developed, the range of file formats expands. Table 5.2 shows the formats and sizes of the files saved in each checkpoint for each of the DL frameworks that we used in the experiments. In addition, the table is also divided according to the neural network model that was used. We wanted to determine if the neural network used can influence the size of the resulting checkpoint files.

Although in all frameworks the HDF5 format can be used, we decided to test the checkpoint in the formats that are specific to the implementation of each DL framework. The only one that implements a native file format without the use of third-party libraries is Tensor-

Flow. In the experiments with TensorFlow we used HDF5 because in the native file format it generates large files which could generate overhead, disadvantaging the performance of TensorFlow when it is compared to other DL frameworks. For reference, using ResNet50 as model, the checkpoint file in native TensorFlow format reached approximately 427 MB. On the other hand, PyTorch uses Pickle to serialize, so its file format is based on it. In the case of Chainer the resulting file format is NPZ and is based on the NumPy library. It is interesting to note that the NumPy library internally uses Pickle to implement serialisation.

If we compare the file sizes by neural network these do not vary considerably with ResNet50. PyTorch and TensorFlow maintain similar sizes, while with Chainer the file size is approximately 19% smaller. It is interesting how well optimized the serialization process is in Chainer, however the price of this optimization is paid with the degradation in performance. On the other hand, with the VGG16 model, all file sizes show a notable increase. This is to be expected as VGG16 has 138 million parameters [147] compared to ResNet50's 25 million [148]. Also, there is a notable increase in file size with PyTorch increasing by 469% and beating Chainer by 331% and TensorFlow by 299%. This large increase shows that PyTorch's serialization mechanisms may not be optimized for different neural network models, unlike Chainer and TensorFlow in which file sizes maintain proportionality increasing from ResNet50 to VGG16 by 63% in Chainer and 42% in TensorFlow.

If we compare the file sizes between the DL frameworks, we can see that there are notable differences that are attributable to the file formats used by each DL framework. Chainer and Pytorch file formats are based on the Pickle library which might suggest that the file sizes should be similar. However, these two DL frameworks perform serialization differently which can influence the size of the resulting file. Chainer uses Pickle through NumPy to serialize to NPZ format, and PyTorch uses pickle directly as part of its checkpoint implementation. Also, even though the HDF5 file format has optimized compression, it does not show a significant difference with the Chainer file size, so using NPZ-based formats can be a good decision. Taking into account the above argument and if there is no user concern for storage space, any of the 3 file formats performs its function adequately and attention should be directed to the performance of each of these. However, DL models and DL applications are growing in complexity and size, so in the short term it will be necessary to pay more attention to the serialization and compression processes of the files resulting from the checkpoints.

5.5.3 Checkpointing at Scale

This set of experiments was performed on the ABCI supercomputer. Our goal is to evaulate checkpointing overhead at scale. We train two models, VGG16 and ResNet50 on the Cifar10 dataset with checkpointing and without checkpointing. For each framework, we set checkpointing at every 5 epochs. This is not ideal, and checkpointing time is determined by several factors such as length of epoch or state of the model during training (i.e. checkpoint only when there is an increase in validation accuracy). For longer training such as ImageNet, one would checkpoint at every epoch though the frequency of checkpointing is dependent on several factors such as checkpointing overheads and the overall reliability of the system.

Table 5.3 and 5.4 show the results of training on ResNet50 and VGG16 with and without checkpointing for Chainer, PyTorch and TensorFlow. As mentioned in section 5.4, VGG16 and ResNet model have different properties and this is seen in the results. It is worth noting that as expected, all frameworks scale well as we increase the number of GPUs. There is a notable difference in the training times for each framework with Tensorflow performing slightly faster than other frameworks when training at small scale but performing worse when running at large scale. This shows that TensorFlow is a framework that has been highly optimized to run in single, node executions, but it has not been optimized to run in a large scale distributed system. Chainer, on the other hand, is a framework that was conceived from the beginning to scale and this can be observed in the results.

However, Chainer has the highest checkpointing overhead and this overhead increases as the number of GPUs increases (ResNet model). This is due to the fact that in Chainer, checkpointing is a sequential operation performed by only one process while other processes wait. As such, the time to checkpoint will remain the same whether we use 1 GPU or 256 GPUs. When training the same model with checkpointing, the time for checkpointing is not large enough to be noticed when the number of GPUs is low. However at large scale, especially at 256 GPUs, the total time is double or more than the training time without checkpointing. On the contrary, Tensorflow performs slightly better than PyTorch and has the lowest checkpoint overhead. It is clear from these results that Chainer and TensorFlow have been optimized in very different ways, and this is demonstrated in these results.

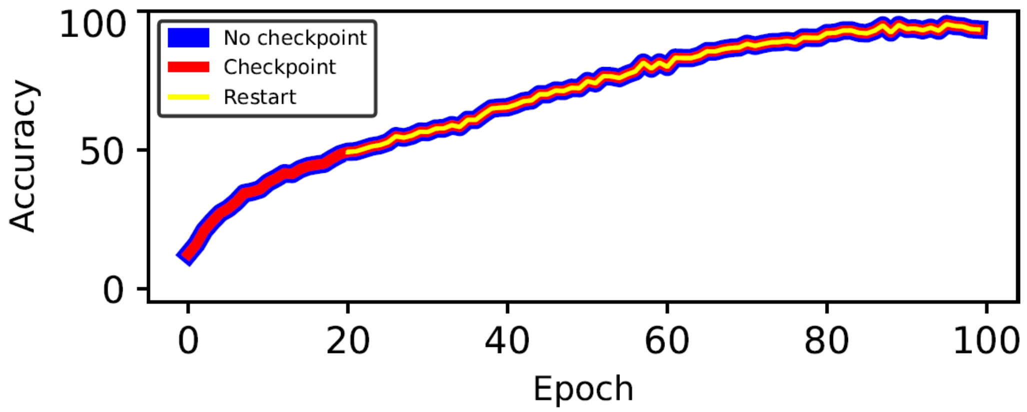
As mentioned early, VGG16 is a deep model with more parameters. Unlike ResNET, VGG16 has no skip connections and generally the model is known for being notoriously slow to train. However, since we are training the model on the Cifar10 dataset (smaller images), the training time is not too long. In this case, when training without checkpointing, Chainer shows the best performance and PyTorch is noticeably slow in comparison. As expected, VGG16 on average takes longer to checkpoint than ResNet. This is also related to the checkpoint sizes in Table 5.2. Chainer has a significant overhead that affects even the scaling behaviour. Even with just 4 GPUs, checkpoint overhead for Chainer is significantly high. Though not the same as Chainer, a significant checkpointing overhead is also observed in PyTorch especially as the number of GPUs increases. In the case of VGG16 as well, TensorFlow has the lowest checkpointing overhead.

5.5.4 Deterministic Checkpointing

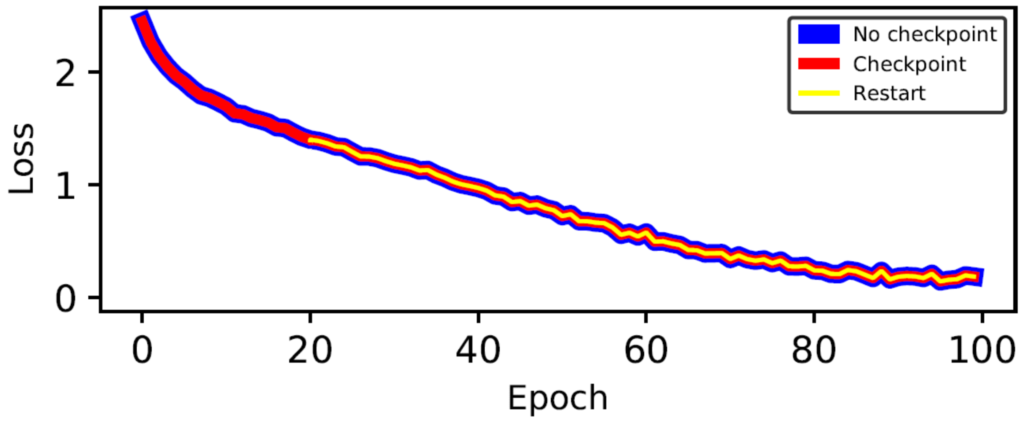
As mentioned in Section 5.4, we made the necessary modifications to the DL frameworks to obtain deterministic results. Figure 5.2 shows accuracy and loss during training. In blue the accuracy and loss are shown without performing a checkpoint, in red the accuracy and loss doing a checkpoint every 5 epochs and in yellow the restart from a checkpoint in epoch 20. Three types of deterministic results are shown. Figure 5.2a compares $i)$ the accuracy of training without checkpointing, $ii)$ the accuracy of training with checkpointing, and $iii)$ the accuracy of training after restart from epoch 20. The same description applies to Figure 5.2b but in regards to the loss.

With PyTorch it was possible to carry out a deterministic distributed training, both in a complete training cycle (100 epochs) and after restarting from a checkpoint (deterministic restart) which validates the operation of the checkpoint mechanism. We modified the checkpoint process to include data from the optimizer. Figures 5.2a and 5.2b show that from epoch 20 onwards (yellow line) values of accuracy and loss are exactly the same. That way, we eliminate the natural randomness generated by training results and the randomness that increases in distributed training, in which the synchronization of processes is not deterministic.

In the case of Chainer and TensorFLow, the accuracy and loss results were replicated, obtaining a deterministic behavior in a complete *failure-free* training run. However, we could not achieve a full deterministic restart. With Chainer, it was possible to generate an almost identical behavior pattern. Table 5.5 shows the accuracy and loss values every 10 epochs for a failure-free execution and compares it with an execution restarted from epoch 20. It can be seen that the values vary (e.g. epoch 20, accuracy is 0.740589 and after restart is 0.740552) and we do not get a deterministic restart.



(a) Accuracy.



(b) Loss.

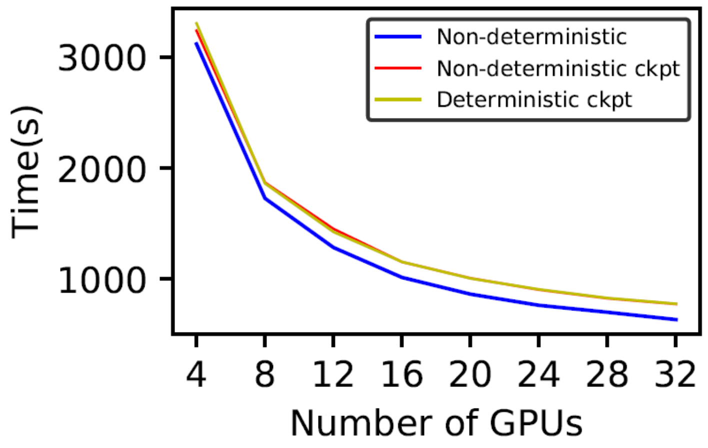
Figure 5.2: Deterministic PyTorch distributed training.

	Accuracy		**Loss**	
Epoch	**Training**	**After restart**	**Training**	**After restart**
10	0.248935	-	2.226590	-
20	0.740589	0.740552	0.743844	0.75368
30	0.789240	0.789595	0.615287	0.614189
40	0.810369	0.809659	0.549811	0.550377
50	0.832741	0.833629	0.478858	0.478444
60	0.854403	0.855291	0.430454	0.429604
70	0.861683	0.862216	0.389529	0.386645
80	0.878374	0.876953	0.353764	0.354646
90	0.887429	0.888139	0.323122	0.319450
100	0.902344	0.900923	0.283951	0.286143

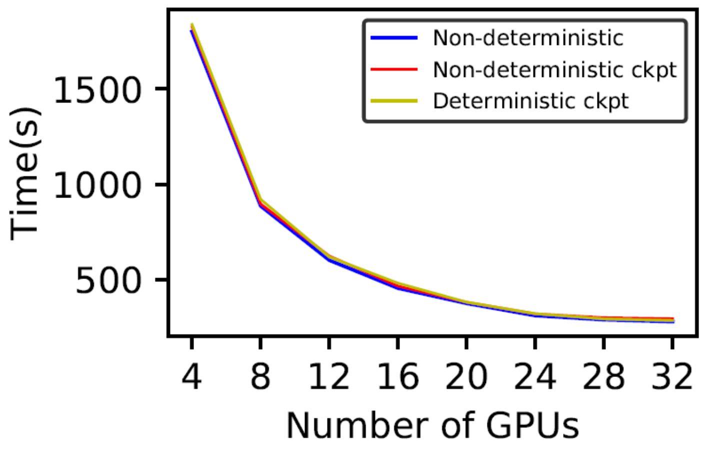
Table 5.5: Accuracy and loss values of a distributed training with Chainer.

Not being able to obtain the same deterministic results in Chainer and TensorFlow after restart is not related to a malfunction of the checkpoint mechanism. Rather, it is the result of not having enough and reliable elements in the framework to modify the non-deterministic state, the difficulty that comes with a deeper manipulation of the framework and the impossibility of manipulating optimised third-party libraries that are used especially in distributed training.

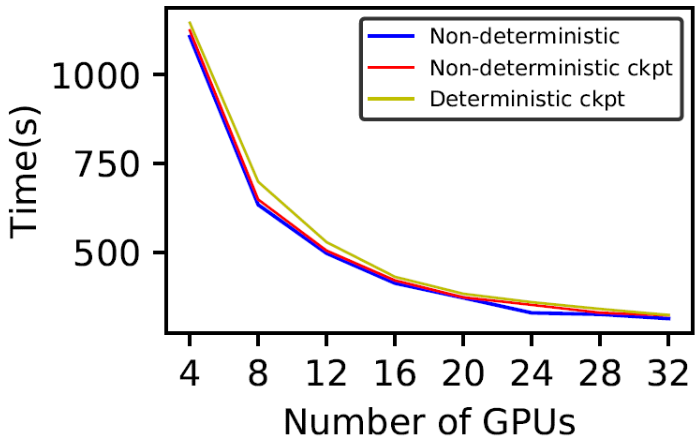
Previously, the performance of non-deterministic training with and without a checkpoint was analysed. Now, we analyse the performance of deterministic training with the execution of checkpoint. Figure 5.3 shows the performance of each of the DL frameworks when running distributed training. Figures show the time it takes to complete a full training (100 epochs) according to the number of GPUs. With these, it is possible to compare the performance of non-deterministic training without checkpoint, non-deterministic training with checkpoint, and deterministic training with checkpoint for each DL framework.



(a) Chainer.



(b) PyTorch.



(c) TensorFlow.

Figure 5.3: Performance of distributed training by each of the DL frameworks.

In the 3 figures, the execution of deterministic training with checkpoint follows the same behaviour as non-deterministic training with checkpoint, presenting an almost imperceptible difference in performance (red line and yellow line). If we calculate the average time difference between non-deterministic checkpoint and deterministic checkpoint, for the entire range, we get 14.1 seconds for Chainer, 19.8 seconds for PyTorch, and 16.7 for TensorFlow. These time differences are relatively low, if we take into account that the calculation was made based on the training times carried out with amounts of GPUs ranging from 4 GPUs to 32 GPUs.

Time variations can be attributed to latencies generated by the parallel file systems or the network. Of course, the completion time of a checkpointed training increases depending on the number of checkpoints that need to be performed during the training. The results show us that deterministic behavior does not influence the performance of distributed training with the execution of checkpoint.

5.6 Discussion

Checkpointing is an integral part of DL workloads in HPC. Our study has unveiled several insights that may help both framework developers and users when training DL models. We discuss those observations in the following subsections.

5.6.1 Checkpoint file format

Each DL framework has its own file format for saving checkpoints. This is expected as the checkpoint mechanism is not standard. There are clear differences between file sizes due to the choice of file formats and compression mechanisms. For instance, Chainer is able to significantly compress checkpoint files. To facilitate using existing checkpointing libraries for DL workloads, the file format should be addressed. For instance, a framework agnostic checkpointing library should be able to checkpoint in a file format that frameworks understand and are able to resume from. Such developments could take the approach used by ONNX [149] which allows Framework Interoperability and inference in multiple hardwares.

5.6.2 Checkpoint implementations

All DL frameworks explored in this study have the ability to checkpoint during training in a native way. Unfortunately, DL frameworks have evolved to adapt to new HPC systems that allow distributed training without taking checkpointing into account. There are checkpointing libraries specifically designed for HPC applications [145, 136]. Those libraries are optimized for better performance under distributed processing conditions. There is already an effort to adapt a checkpoint library developed for HPC environments to DL applications [135]. The idea is to offer DL applications advanced features such as asynchronous multi-level checkpointing, hiding the complexity of heterogeneous storage, or providing efficient serialisation on local storage, which are some of the fundamental elements to obtain the maximum performance. All these elements are not present in the current DL frameworks.

5.6.3 Checkpoint scalability

Increasing the number of GPUs significantly improves training time of DL models. However, using multiple nodes does not improve the checkpointing time as only one node is in charge of checkpointing. Though Chainer has some sort of multi-node checkpoint implementation, our study showed that this simply produces redundant copies of the application's state and does not improve the time to checkpoint. There is therefore an opportunity for external libraries typically used in HPC to improve checkpointing in DL workloads.

5.6.4 Deterministic behavior in DNN

Determinism in DL models is a secondary consideration. That might be expected since ANNs have a random nature that seeks to model the behavior of neurons in a biological brain. However, the randomness of the results can raise reasonable doubts about their veracity as they are not reproducible, especially when DL models grow in complexity and are susceptible to failures. This is particularly important for critical-mission applications such as self-driving cars, among others. We realised that the options offered by DL frameworks to generate 100% deterministic results in training and when restarting a training (deterministic restart) are not always reliable to generate reproducible results. It is imperative for DL frameworks to provide clear and easy-to-implement mechanisms that allow the reproducibility of results and therefore research environments that allow validation of the experiments.

5.6.5 Data parallel vs model parallel

Though this study primarily focused on data parallelism, it is also important to consider model parallelism especially now that it is becoming popular due to increase in model sizes. In the data parallelism paradigm, there is a replica of the model on each GPU. In model parallelism, the model is split between processes and each process trains a part of the model. Existing checkpointing implementations in DL frameworks do not do partial checkpoints. The challenge is that all processes have a replica of the model and all perform an all-reduce operation at the end of an iteration. To scale checkpointing up, the model has to be broken up, so that each process checkpoints a small part of it.

	PyTorch	Tensorflow	Chainer
Checkpoint Implementation	✓	✓	✓
Multi-Node Checkpointing	-	-	✓
Deterministic Training	✓	✓	✓
Model Parallel Checkpointing	-	-	-

Table 5.6: Checkpointing Features of DL Frameworks.

A summary of the findings discussed in this section and the checkpointing properties of the different frameworks is given in Table 5.6.

5.7 Conclusion

Checkpointing is a fundamental component when training DL models in HPC due to the lengthy training times and high probability of hardware faults in HPC systems. In this chapter, we have evaluated through a series of experiments the checkpointing implementations of three state-of-the-art DL frameworks. We have evaluated factors such as computational cost of checkpointing, file sizes and formats, the effect of scale and deterministic behaviour. All frameworks have a form of checkpointing support that is considered sufficient. However, our evaluation has shown that at scale, this checkpoint implementations come with a significant overhead as many GPUs remain idle during checkpointing, particularly for Chainer and Pytorch. The file size of the checkpoint changes significantly with the model in Chainer unlike Tensorflow and Pytorch. The insights provided in this chapter can help users and framework developers to guide future developments of advanced checkpointing libraries for DL workloads in HPC.

Chapter 6

Conclusion

There is no doubt that Machine Learning will continue to drive revolutions in Artificial Intelligence. It is also clear that HPC will be at the core of these revolutions as models and data sizes get larger. Though the work carried out in this thesis is not conclusive, it has led to key findings and tools that could substantially improve Machine Learning in HPC. In this chapter, we first summarise the chapters in this thesis and conclusions from each chapter. We then provide key takeaways and lessons from this thesis and finally present ideas that couldn't be explored due to time limitations but will be pursued as future work.

6.1 Summary of Achievements

In chapter 1 highlighted the journey of Machine Learning and the role AI will play in future scientific discoveries and other applications. We provide the motivations for studying ML in HPC systems. These motivations included the need for larger models to drive innovation as well as the increasing use of scientific datasets. We then used this motivation to give the challenges that ML applications face in HPC systems. Such challenges include traditional HPC challenges such as network congestion as well as complex challanges specific to ML such as framework limitations and the grand challenge of performance modelling. We also shared a summary of the objectives and contributions of this thesis.

In chapter 2 we gave a background of Machine Learning. We also explained how distributed ML works and the different frameworks used in this thesis to carry out experiments. We further discussed HPC systems used in this thesis, i.e. MareNustrum supercomputer and ABCI supercomputer. Though related work is provided in each chapter, we have a short related work for Machine Learning in HPC.

Chapter 3 is the first research chapter of this thesis and presents the first research work carried out. This chapter introduces PyCOMPSs, a framework that aims to ease the development and execution of Python parallel applications for distributed infrastructures, such as Clusters and Clouds. We then present a Hyperparameter scheme built on top of PyCOMPSs that can significantly accelerate the process of searching for the correct Hyperparameters in HPC. This work is based on the premise that each experiment in a hyperparameter search is essentially a task. By exploring task parallelism, we can launch as many tasks as there are computing resources and terminate a task as soon as the desired parameters are found or using a predefined algorithm. We demonstrate the usefulness of this tool by conducting a wide range of experiments using state of the art ML models.

Chapter 4 is the core chapter of this thesis. This chapter's work was conducted over two years, and hence it's the longest chapter. This chapter first introduced challenges in distributed ML which build the foundation for this work. We then introduce and define the strategies for distributed training. We produce paraDL, an oracle to guide the right strategy for distributed training. We first demonstrate this tool's accuracy by conducting a myriad of experiments up to thousands of GPUs. We then use this tool to show limitations in scaling ML such as communication, memory capacity and computation. Finally, we provide a summary of the analysis.

HPC clusters are susceptible to faults and errors that can crash the entire system or a HPC node. As training large models takes weeks or even months, it's crucial to ensure that such efforts are not wasted should an error happen. For this reason, checkpointing is a fundamental part of ML. However, due to ML frameworks' historical single node development, checkpointing in ML is not adapted to HPC. Chapter 5 is a detailed study of checkpointing in HPC. Though we don't develop a checkpointing tool (This is left for future work), we study the details of checkpointing of ML in HPC and highlight important things that should be considered when developing a checkpointing framework for ML applications in HPC.

6.2 Notable Observations

In the introduction we embarked on a journey to study scaling ML workflows in HPC. In this section, we highlight some of the key observations findings from this thesis.

Task-based programming model in Machine Learning. Task based model programming allows us to divide computation into tasks. This model has potential advantages and use cases especially in hyperparameter optimisation. While it is technically possible, to submit all evaluations in independent job. But there are two main drawbacks: first, all jobs will have to wait and compete with the other users of the queue, while with PyCOMPSs are in the same reservation (even enabling to reuse memory objects from one task to the next if they use the same object), and second, PyCOMPSs also enables you to design more complex workflows without the user need to do it by hand (as it should be done using directly with slurm), that in the application of the paper maybe it is not needed, but when you include a merging process, or a more complex structure, then becomes very difficult doing it with a queuing system.

Performance Modelling: Performance models are instrument to understand scaling behaviour of parallel applications . Using a performance models, we are able to understand the how performance changes as we manipulate different parameters. From this work we realised that a purely analytical model is impossible and unrealistic. Because of this we combined some empirical measurements in our model to enhance prediction. This empirical measurements are easy to acquire for any system and therefore dont limit our model.

Low Focus on Fault Tolerance: In the literature study leading to the final chapter that focuses on fault tolerance, we noticed how little exists in the literature on check-pointing for ML workloads. As mentioned in the chapter, this is probably due to the fact that existing implementations have so far been enough. However, with the increasing large models and complex datasets, this area of research is likely to gain some traction. We have highlighted in chapter 5 so key areas that can be improved in existing frameworks to support more complex checkpointing such as those common in HPC. We note that with increasing use of model parallelism, better checkpointing approaches will be required.

6.3 Behind the Scenes

This thesis is the result of several years of research and over a decade of building the foundational knowledge required to perform such research. Over those years, I have tried countless things and learnt invaluable lessons that I share here. This section highlights some of the things that went on behind the scenes and that eventually led to the results published in this thesis.

In the initial days of this research, most of the work was carried out in Tensorflow. Though Tensorflow is a fully developed and well-maintained framework, several challenges arose due to the complexity of modifying the framework for research purposes. For the work carried out in chapter 4, we opted for the Chainer framework that allowed more flexibility. PyTorch framework is also used in chapter 5. Experimenting with different frameworks allowed us to understand each framework's strengths and weaknesses and use a specific framework for specific tasks. This work also helped me gain a very deep understanding of different frameworks. Furthermore, we developed a chainer extension to support different parallelsim strategies and publicly released this extension under the name ***ChainerMNX***

One of the most challenging problems to solve during the implementation of different parallelism strategies was ensuring correctness. For instance, splitting a model across multiple devices should not yield different results from training the model on a single device. A lot of effort was put into ensuring reproducibility and this consistency across different parallelism strategies. To achieve this, we had to implement several strategies to debug cases that yielded different results. A significant engineering effort was required for this. For most of the parallelism strategies, we guarantee bit-wise reproducibility. This was possible due to the flexibility that the Chainer framework allows.

I also had a short collaboration with the Earth Sciences Department at BSC, where we explored opportunities in Deep Learning for Hurricane season analysis. This project was exciting due to the nature of hurricane data available, which is limited to 100 years. Furthermore, hurricane data contains multiple channels and dimensions, including a time dimension. This is interesting from a scalability point of view as there are many dimensions to scale, unlike image data and requires novel methods because the data set is very small. Even though the Earth Science Department soon hired someone specifically to work on that project, my work was instrumental as it formed the building block of the bigger project. Apart from working with leading climate scientists, this work also gave me insight into Earth Sciences and hurricanes and hurricane season prediction. Part of the results of this short collaboration was published in a workshop at ISC-2018.

6.4 Future Outlooks

There is only so much that can be done in three years. Several choices had to be made regarding what to prioritise and the extent to which research was carried out. Therefore, this thesis's end is not the end of research on the issues raised in this chapter. Several ideas have just been highlighted and not fully developed. This will be carried out in the future. For instance, in chapter 3, we presented a HPO scheme. However, we did not present the full tool that can be used for this. Future work will develop the tool and avail it for public use. In chapter 5, we carried out an extensive study of checkpointing for ML in HPC and highlighted the need for a checkpointing library that factors in ML applications' unique nature. This tool will be developed in the future.

Appendix A

Publications and Dissemination

A.1 Publications

1. **Kahira, A.N.** , Nguyen, T, Wahib, M., Gomez, L. B., Badia, R. M., Takano, R. *An Oracle for Guiding Large-Scale Model/Hybrid Parallel Training of Convolutional Neural Networks* In The ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2021)

2. Rojas, E., **Kahira, A. N.**, Meneses, E., Gomez, L. B., & Badia, R. M. (2020). *A Study of Checkpointing in Large Scale Training of Deep Neural Networks.* In The 2020 International Conference on High Performance Computing & Simulation(HPCS 2020)

3. **Kahira, A. N.**, Gomez, L. B., Aktas, H. Yalcin, G. Unsal, O. & Cristal, A. *Effects of Hardware Faults on Artificial Neural Networks* In 6. Ulusal Yüksek Başarımlı Hesaplama Konferansı. (Basarım 2020 HPC)

4. **Kahira, A. N.**, Gomez, L. B., Conejero, J., & Badia, R. M. (2019, August). Accelerating hyperparameter optimisation with PyCOMPSs. In Proceedings of the 48th International Conference on Parallel Processing: Workshops (pp. 1-8). `https://doi.org/10.1145/3339186.3339200`

5. **Kahira, A.N.** , Gomez, L. B., & Badia, R. M. (2018, June). *Training deep neural networks with low precision input data: a hurricane prediction case study.* In International Conference on High Performance Computing (pp. 562-569). Springer, Cham.
`https://doi.org/10.1007/978-3-030-02465-9_40`

6. **Kahira, A.**, Bautista Gomez, L., & Badia Sala, R. M. (2018, April). A machine learning workflow for hurricane prediction. In Book of abstracts (pp. 72-73). Barcelona Supercomputing Center.

A.2 Workshops, Summer Schools and Talks

1. **Deep Learning Indaba - Stellenbosch, South Africa**, September 2018
   A series of master classes that brings advanced knowledge of Machine Learning and Deep Learning with the aim of strengthening Machine Learning in Africa
   Presented a poster and won a poster award

2. **International HPC Summer School - Ostrava, Czech Republic**, July 2018
   Among the 80 PhD students and Post-docs selected from Europe,Japan and America Presented a poster and a flash talk

3. **International Supercomputing Conference (ISC) - Frankfurt, Germany**, June 2018
   Presented a poster and flash talk at the PhD Forum Presented at workshop paper at the 1st International Workshop on Approximate and Transprecision Computing on Emerging Technologies

4. **BSC International Doctoral Symposium - Barcelona, Spain**, April 2018
   Presented a poster and extended abstract.

A.3 Research Collaboration

1. **AIST-Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory , Tokyo, Japan**
   A three months research stay at Tokyo Institute of Technology working on novel architectures for model parallelism on the ABCI supercomputer.

A.4 Grants

1. **Eurolab4HPC Short Term Collaboration Grant**
   Grant worth 5000 Euros for a research stay at Tokyo Institute of Technology to study Scaling of Large Machine Learning Models in HPC

2. **DFG SPPEXA Travel Grant to ISC 2018**
   Grant worth 600 Euros to cover Travel expenses to ISC

3. **INPhINIT La Caixa Fellowship for PhD in Spanish Centres of Excellence**
   European Union's Horizon 2020 research and innovation programme under the Marie Sklowdowska-Curie grant agreement No. 713673

Bibliography

[1] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[2] W. G. Hatcher and W. Yu, “A survey of deep learning: platforms, applications and emerging research trends,” *IEEE Access*, vol. 6, pp. 24411–24432, 2018.

[3] S. Dong, P. Wang, and K. Abbas, “A survey on deep learning and its applications,” *Computer Science Review*, vol. 40, p. 100379, 2021.

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

[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[6] R. Raina, A. Madhavan, and A. Y. Ng, “Large-scale deep unsupervised learning using graphics processors,” in *Proceedings of the 26th annual international conference on machine learning*, pp. 873–880, 2009.

[7] A. Halevy, P. Norvig, and F. Pereira, “The unreasonable effectiveness of data,” *IEEE Intelligent Systems*, vol. 24, no. 2, pp. 8–12, 2009.

[8] Y. Shen *et al.*, “Learning semantic representations using convolutional neural networks for web search,” WWW ’14 Companion, (New York, NY, USA), p. 373–374, Association for Computing Machinery, 2014.

[9] A. Kahira *et al.*, “Training deep neural networks with low precision input data: A hurricane prediction case study,” in *High Performance Computing*, (Cham), pp. 562–569, Springer International Publishing, 2018.

[10] I. Wallach *et al.*, “Atomnet: A deep convolutional neural network for bioactivity prediction in structure-based drug discovery,” 2015.

[11] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, S. J. Pennycook, *et al.*, “Cosmoflow: Using deep learning to learn the universe at scale,” in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 819–829, IEEE, 2018.

[12] E. Callaway, “’it will change everything’: Deepmind’s ai makes gigantic leap in solving protein structures.,” *Nature*, 2020.

[13] T. Ben-Nun *et al.*, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” *CoRR*, vol. abs/1802.09941, 2018.

[14] Y. You *et al.*, “ImageNet Training in Minutes,” in *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pp. 1:1–1:10, 2018.

[15] Y. Huang *et al.*, “GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism,” *CoRR*, vol. abs/1811.06965, 2018.

[16] T. Kurth *et al.*, “Exascale Deep Learning for Climate Analytics,” SC ’18, pp. 51:1–51:12, 2018.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[18] M. Copik *et al.*, “Extracting clean performance models from tainted programs,” 2020.

[19] F. Cappello, G. Al, W. Gropp, S. Kale, B. Kramer, and M. Snir, “Toward exascale resilience: 2014 update,” *Supercomputing Frontiers and Innovations: an International Journal*, vol. 1, no. 1, pp. 5–28, 2014.

[20] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[21] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[22] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, “Mastering the game of go with deep neural networks and tree search,” *nature*, vol. 529, no. 7587, p. 484, 2016.

[23] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, “Pipedream: Fast and efficient pipeline parallel dnn training,” 2018.

[24] C.-C. Chen, C.-L. Yang, and H.-Y. Cheng, “Efficient and robust parallel dnn training through model parallelism on multi-gpu platform,” 2018.

[25] S. Tokui, R. Okuta, T. Akiba, Y. Niitani, T. Ogawa, S. Saito, S. Suzuki, K. Uenishi, B. Vogel, and H. Y. Vincent, “Chainer: A deep learning framework for accelerating the research cycle,” 2019.

[26] A. Paszke and et al., “Pytorch: An imperative style, high-performance deep learning library,” 2019.

[27] M. Abadi and et al., “Tensorflow: A system for large-scale machine learning,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (Savannah, GA), pp. 265–283, USENIX Association, Nov. 2016.

[28] M. Abadi and et al., “Tensorflow: Large-scale machine learning on heterogeneous systems,” 2015. Software available from tensorflow.org.

[29] S. Tokui and K. Oono, “Chainer:a next-generation open source framework for deep learning,” 2015.

[30] i. Preferred Networks and i. Preferred Infrastructure, “Cupy – numpy-like api accelerated with cuda,” 2015.

[31] T. Akiba, K. Fukuda, and S. Suzuki, “Chainermn: Scalable distributed deep learning framework,” 2017.

[32] R. Collobert, K. Kavukcuoglu, and C. Farabet, “Torch7: A matlab-like environment for machine learning,” 2011.

[33] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, “Large scale distributed deep networks,” in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1223–1231, Curran Associates, Inc., 2012.

[34] F. Chollet, “Keras,” 2020.

[35] A. Gibiansky, “Bringing hpc techniques to deep learning,” feb 2017.

[36] N. Corporation, “Nvidia collective communications library (nccl),” mar 2020.

[37] L. Deng, “The mnist database of handwritten digit images for machine learning research,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[38] A. Krizhevsky and G. Hinton, “Learning multiple layers of features from tiny images,” tech. rep., Citeseer, 2009.

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A Large-Scale Hierarchical Image Database,” in *CVPR09*, 2009.

[40] National Energy Research Scientific Computing Center, “CosmoFlow datasets.” `https://portal.nersc.gov/project/m3363/`. [15 January 2020].

[41] S.-X. Zou, C.-Y. Chen, J.-L. Wu, C.-N. Chou, C.-C. Tsao, K.-C. Tung, T.-W. Lin, C.-L. Sung, and E. Y. Chang, “Distributed training large-scale deep architectures,” in *International Conference on Advanced Data Mining and Applications*, pp. 18–32, Springer, 2017.

[42] V. Amatya, A. Vishnu, C. Siegel, and J. Daily, “What does fault tolerant deep learning need from mpi?,” in *Proceedings of the 24th European MPI Users’ Group Meeting*, pp. 1–11, 2017.

[43] H. Asaadi and B. Chapman, “Comparative study of deep learning framework in hpc environments,” in *2017 New York Scientific Data Summit (NYSDS)*, pp. 1–7, IEEE, 2017.

[44] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, “Performance modeling and scalability optimization of distributed deep learning systems,” in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1355–1364, 2015.

[45] R. Mayer and H.-A. Jacobsen, “Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–37, 2020.

[46] T. Ben-Nun and T. Hoefler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[47] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015. Software available from tensorflow.org.

[48] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[49] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[50] "PyCOMPSs: Parallel computational workflows in Python," *International Journal of High Performance Computing Applications*, 2017.

[51] J. Bergstra JAMESBERGSTRA and U. Yoshua Bengio YOSHUABENGIO, "Random Search for HyperParameter Optimization," *Journal of Machine Learning Research*, 2012.

[52] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for Hyper-Parameter Optimization," in *NIPS Proceedings*, 2011.

[53] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," tech. rep.

[54] F. Madrigal, C. Maurice, and F. Lerasle, "Hyper-parameter optimization tools comparison for multiple object tracking applications," *Machine Vision and Applications*, vol. 30, pp. 269–289, mar 2018.

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[56] L. Hertel, J. Collado, P. Sadowski, J. Collado, and P. Baldi, "Sherpa : Hyperparameter Optimization for Machine Learning Models," no. Nips, 2018.

[57] "SHADHO: Massively scalable hardware-aware distributed hyperparameter optimization," in *Proceedings - 2018 IEEE Winter Conference on Applications of Computer Vision, WACV 2018*, 2018.

[58] J. Bergstra, D. Yamins, and D. D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pp. I–115–I–123, JMLR.org, 2013.

[59] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A Research Platform for Distributed Model Selection and Training," jul 2018.

[60] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A Distributed Framework for Emerging AI Applications," dec 2017.

[61] P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, and I. Sutskever, "Deep double descent: Where bigger models and more data hurt," *arXiv preprint arXiv:1912.02292*, 2019.

[62] "Paraver: a flexible performance analysis tool."

[63] J. Dean *et al.*, "Large scale distributed deep networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, pp. 1223–1231, 2012.

[64] S. Rajbhandari *et al.*, "ZeRO: Memory Optimization Towards Training A Trillion Parameter Models," *ArXiv*, vol. abs/1910.02054, 2019.

[65] T. B. Brown *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[66] A. Mathuriya *et al.*, "Cosmoflow: Using deep learning to learn the universe at scale," SC '18, pp. 65:1–65:11, 2018.

[67] X. Jia *et al.*, "Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes," *CoRR*, vol. abs/1807.11205, 2018.

[68] M. Yamazaki *et al.*, "Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds," *CoRR*, vol. abs/1903.12650, 2019.

[69] M. Bayatpour *et al.*, "Scalable reduction collectives with data partitioning-based multi-leader design," SC '17, pp. 64:1–64:11, 2017.

[70] T. T. Nguyen *et al.*, "Hierarchical Distributed-Memory Multi-Leader MPI-Allreduce for Deep Learning Workloads," CANDAR18, pp. 216–222, 2018.

[71] F. Seide *et al.*, "1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs," Interspeech 2014, September 2014.

[72] W. Wen *et al.*, "TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning," in *Advances in Neural Information Processing Systems 30*, pp. 1509–1519, Curran Associates, Inc., 2017.

[73] N. Strom, “Scalable distributed DNN training using commodity GPU cloud computing,” Sixteenth Annual Conference of the International Speech Communication Association, 2015.

[74] F. Sattler, S. Wiedemann, K. Müller, and W. Samek, “Sparse binary compression: Towards distributed deep learning with minimal communication,” *CoRR*, vol. abs/1805.08768, 2018.

[75] C. Renggli *et al.*, “SparCML: High-Performance Sparse Communication for Machine Learning,” SC ’19, 2019.

[76] N. Dryden *et al.*, “Channel and Filter Parallelism for Large-Scale CNN Training,” SC ’19, pp. 46:1–46:13, 2019.

[77] F. Seide *et al.*, “On parallelizability of stochastic gradient descent for speech DNNS,” in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 235–239, May 2014.

[78] P. Goyal *et al.*, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour,” *CoRR*, vol. abs/1706.02677, 2017.

[79] S. L. Smith *et al.*, “Don’t Decay the Learning Rate, Increase the Batch Size,” in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Conference Track Proceedings*, 2018.

[80] T. Akiba *et al.*, “Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes,” *CoRR*, vol. abs/1711.04325, 2017.

[81] O. Russakovsky *et al.*, “Imagenet large scale visual recognition challenge,” *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[82] B. Ginsburg *et al.*, “Large batch training of convolutional networks with layer-wise adaptive rate scaling,” 2018. [01 April 2020].

[83] Y. You *et al.*, “Large Batch Optimization for Deep Learning: Training BERT in 76 minutes,” 2020. [01 April 2020].

[84] J. G. Pauloski *et al.*, “Convolutional neural network training with distributed K-FAC,” *arXiv preprint arXiv:2007.00784*, 2020.

[85] H. Zhang *et al.*, “Context encoding for semantic segmentation,” in *CVPR2018*, June 2018.

[86] Y. Wu and K. He, “Group normalization,” in *Proceedings of the European conference on computer vision (ECCV)*, pp. 3–19, 2018.

[87] A. Kolesnikov *et al.*, “Big Transfer (BiT): General Visual Representation Learning,” *arXiv preprint arXiv:1912.11370*, 2019.

[88] S. Lym *et al.*, “PruneTrain: Fast Neural Network Training by Dynamic Sparse Model Reconfiguration,” SC ’19, 2019.

[89] T. Chen *et al.*, “Training Deep Nets with Sublinear Memory Cost,” *ArXiv*, vol. abs/1604.06174, 2016.

[90] M. Wahib *et al.*, “Scaling Distributed Deep Learning Workloads beyond the Memory Capacity with KARMA,” *arXiv preprint arXiv:2008.11421*, 2020.

[91] J. Domke *et al.*, “HyperX Topology: First at-Scale Implementation and Comparison to the Fat-Tree,” SC ’19, 2019.

[92] J. Dong *et al.*, “EFLOPS: Algorithm and System Co-Design for a High Performance Distributed Training Platform,” in *HPCA*, pp. 610–622, 2020.

[93] A. Castelló *et al.*, “Analysis of Model Parallelism for Distributed Neural Networks,” EuroMPI ’19, pp. 7:1–7:10, 2019.

[94] A. Gholami *et al.*, “Integrated model, batch and domain parallelism in training neural networks,” *arXiv preprint arXiv:1712.04432*, 2017.

[95] Z. Jia *et al.*, “Exploring hidden dimensions in parallelizing convolutional neural networks,” *arXiv preprint arXiv:1802.04924*, 2018.

[96] J. Zhihao *et al.*, “Beyond Data and Model Parallelism for Deep Neural Networks,” *CoRR*, vol. abs/1807.05358, 2018.

[97] Y. Oyama, N. Maruyama, N. Dryden, E. McCarthy, P. Harrington, J. Balewski, S. Matsuoka, P. Nugent, and B. Van Essen, “The case for strong scaling in deep learning: Training large 3d cnns with hybrid parallelism,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1641–1652, 2021.

[98] N. Dryden *et al.*, “Improving Strong-Scaling of CNN Training by Exploiting Finer-Grained Parallelism,” in *IPDPS 2019*, pp. 210–220.

[99] A. Krizhevsky, “One weird trick for parallelizing convolutional neural networks,” *arXiv preprint arXiv:1404.5997*, 2014.

[100] M. Shoeybi *et al.*, “Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism,” *ArXiv*, vol. abs/1909.08053, 2019.

[101] “Benanza: Automatic μbenchmark generation to compute ”lower-bound” latency and inform optimizations of deep learning models on gpus,”

[102] H. Jin *et al.*, “Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures,” *ACM Trans. Archit. Code Optim.*, vol. 15, Sept. 2018.

[103] J. A. Rico-Gallego *et al.*, “A survey of communication performance models for high-performance computing,” *ACM Comput. Surv.*, vol. 51, Jan. 2019.

[104] P. Sanders *et al.*, “Two-tree algorithms for full bandwidth broadcast, reduction and scan,” *Parallel Computing*, vol. 35, no. 12, pp. 581 – 594, 2009.

[105] S. C. Kim *et al.*, “Measurement and prediction of communication delays in myrinet networks,” *Journal of Parallel and Distributed Computing*, vol. 61, no. 11, pp. 1692 – 1704, 2001.

[106] M. Martinasso *et al.*, “A contention-aware performance model for hpc-based networks: A case study of the infiniband network,” in *Euro-Par 2011 Parallel Processing*, 2011.

[107] S. Chunduri *et al.*, “Gpcnet: Designing a benchmark suite for inducing and measuring contention in hpc networks,” SC ’19, 2019.

[108] “GPUDirect.” `https://developer.nvidia.com/gpudirect`. [21 April 2020].

[109] A. Li *et al.*, “Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect,” *CoRR*, vol. abs/1903.04611, 2019.

[110] I. Sergey *et al.*, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” *CoRR*, vol. abs/1502.03167, 2015.

[111] K. Simonyan *et al.*, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” in *ICLR 2015*, 2015.

[112] C. Kim *et al.*, “torchgpipe: On-the-fly pipeline parallelism for training giant models,” *arXiv preprint arXiv:2004.09910*, 2020.

[113] D. Narayanan *et al.*, “PipeDream: Generalized Pipeline Parallelism for DNN Training,” SOSP ’19, p. 1–15, 2019.

[114] Y. Xu *et al.*, “Automatic Cross-Replica Sharding of Weight Update in Data-Parallel Training,” *arXiv preprint arXiv:2004.13336*, 2020.

[115] G. Dong *et al.*, “Fully convolutional spatio-temporal models for representation learning in plasma science,” *arXiv preprint arXiv:2007.10468*, 2020.

[116] O. E. Gundersen and S. Kjensmo, “State of the art: Reproducibility in artificial intelligence,” 2018.

[117] N. Corporation, “Nvidia cudnn,” 2020.

[118] Z. Zhang, L. Huang, U. Manor, L. Fang, G. Merlo, C. Michoski, J. Cazes, and N. Gaffney, “Fanstore: Enabling efficient and scalable i/o for distributed deep learning,” 2018.

[119] S. Pumma, M. Si, W. Feng, and P. Balaji, “Parallel i/o optimizations for scalable deep learning,” in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 720–729, 2017.

[120] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, “Entropy-aware i/o pipelining for large-scale deep learning on hpc systems,” in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 145–156, 2018.

[121] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury, “Efficient user-level storage disaggregation for deep learning,” in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–12, 2019.

[122] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, “Distributed deep learning using synchronous stochastic gradient descent,” 2016.

[123] S. Alqahtani and M. Demirbas, “Performance analysis and comparison of distributed machine learning systems,” 2019.

[124] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, “Revisiting distributed synchronous sgd,” 2016.

[125] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, and et al., “Large scale distributed deep networks,” in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS’12, (Red Hook, NY, USA), p. 1223–1231, Curran Associates Inc., 2012.

[126] J. Keuper and F. Preundt, "Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability," in *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pp. 19–26, Nov 2016.

[127] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. Panda, "Efficient large message broadcast using nccl and cuda-aware mpi for deep learning," 09 2016.

[128] V. Campos, F. Sastre, M. Yagües, M. Bellver, X. G. i Nieto, and J. Torres, "Distributed training strategies for a computer vision deep learning algorithm on a distributed gpu cluster," *Procedia Computer Science*, vol. 108, pp. 315 – 324, 2017. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[129] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," 2017.

[130] T. Liu, W. Wen, L. Jiang, Y. Wang, C. Yang, and G. Quan, "A fault-tolerant neural network architecture," 2019.

[131] P. Blanchard, E. M. E. Mhamdi, R. Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent," 2017.

[132] Y. Chen, L. Su, and J. Xu, "Distributed statistical machine learning in adversarial settings: Byzantine gradient descent," 2017.

[133] L. Chen, H. Wang, Z. Charles, and D. Papailiopoulos, "Draco: Byzantine-resilient distributed training via redundant gradients," 2018.

[134] A. Kulakov, M. Zwolinski, and J. Reeve, "Fault tolerance in distributed neural computing," 09 2015.

[135] B. Nicolae, J. Li, J. M. Wozniak, G. Bosilca, M. Dorier, and F. Cappello, "Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models," in *2020 20th IEEE/ACMCCGRID*, pp. 172–181, 2020.

[136] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "Veloc: Towards high performance adaptive asynchronous checkpointing at large scale," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 911–920, 2019.

[137] A. Qiao, B. Aragam, B. Zhang, and E. Xing, "Fault tolerance in iterative-convergent machine learning," in *Proceedings of the 36th International Conference on Machine*

Learning (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, (Long Beach, California, USA), pp. 5220–5230, PMLR, 09–15 Jun 2019.

[138] H.-R. Wei, S. Huang, R. Wang, X.-y. Dai, and J. Chen, "Online distilling from checkpoints for neural machine translation," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, (Minneapolis, Minnesota), pp. 1932–1941, Association for Computational Linguistics, June 2019.

[139] O. Beaumont, J. Herrmann, L. Eyraud-Dubois, J. Hermann, A. Joly, and A. Shilova, "Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory," 2019.

[140] R. Islam, P. Henderson, M. Gomrokchi, and D. Precup, "Reproducibility of benchmarked deep reinforcement learning tasks for continuous control," 2017.

[141] I. Icke and J. C. Bongard, "Improving genetic programming based symbolic regression using deterministic machine learning," in *2013 IEEE Congress on Evolutionary Computation*, pp. 1763–1770, 2013.

[142] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, p. I–387–I–395, JMLR.org, 2014.

[143] I. Ilievski, T. Akhtar, J. Feng, and C. A. Shoemaker, "Efficient hyperparameter optimization for deep learning algorithms using deterministic rbf surrogates," 2016.

[144] P. Nagarajan, G. Warnell, and P. Stone, "Deterministic implementations for reproducibility in deep reinforcement learning," 2018.

[145] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: High performance fault tolerance interface for hybrid systems," in *SC '11*, pp. 1–12, 2011.

[146] N. Corporation, "Nvidia tesla v100," 2020.

[147] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014.

[148] S. Zagoruyko and N. Komodakis, "Wide residual networks," 2016.

[149] O. R. developers, “Onnx runtime.” `https://www.onnxruntime.ai`, 2021. Version: x.y.z.