

Lars Sørli

Preparing the CERN Machine-Learned Particle-Flow Model for Exascale using Horovod: Experience and Performance Studies on the Flatiron and Jülich Supercomputers

Masteroppgave i MIDT

Veileder: Prof. Anne C. Elster

Medveileder: Dr. Maria Girone

November 2022

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

CERN-THESIS-2022-182
/ 2022



Forschungszentrum Jülich

Lars Sørli

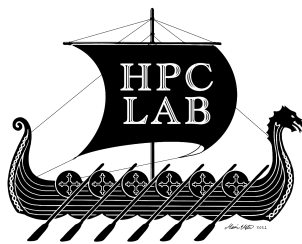
Preparing the CERN Machine-Learned Particle-Flow Model for Exascale using Horovod: Experience and Performance Studies on the Flatiron and Jülich Supercomputers

Masteroppgave i MIDT
Veileder: Prof. Anne C. Elster
Medveileder: Dr. Maria Girone
November 2022

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Lars Sørli

Preparing the CERN Machine-Learned Particle-Flow Model for Exascale using Horovod: Experience and Performance Studies on the Flatiron and Jülich Supercomputers



Master's thesis in Computer Science
Supervisor: Professor Anne C. Elster
Co-supervisor: Maria Girone
November 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science



Norwegian University of
Science and Technology

Abstract

There is an increase in interest of showing the importance of HPC for Artificial Intelligence and Artificial Intelligence for HPC. HPC centers are now boasting larger and larger compute power, with more centers reaching exascale. CERN is investigating the usage of Artificial Intelligence and Machine Learning to augment or replace some of the traditional workflows within the LHC experiments. Advantages of Machine Learning and Artificial Intelligence is their highly parallelizable nature on suitable hardware, such as GPUs.

MLPF, like every other large scale model, utilizes large compute resources in order to become efficient and accurate. Models and their datasets keep increasing in size, further expanding their need for more computer resources. The work in this thesis includes implementing a distributed version of the Graph Neural Network MLPF using the Horovod framework with the aim to scale the applications to exascale-class supercomputers. Horovod is a well-established framework for distributed workloads within the Artificial Intelligence field.

Our work uses the Horovod framework to distribute the work across up to 292 supercomputer nodes each with up to 4 GPUs, i.e. runs with up to over 1100 GPUs on the Jülich Juwels supercomputer. Our work focuses on experiments on the Nvidia Volta and Ampere architecture, as these were the best available to us during this thesis. During these scaling tests we observe that the performance scales well up to 24 nodes. In particular, we see a speedup of up to 20X and on 24 nodes, whereas the speedup reduces to 50X when going to 100 nodes.

The thesis also includes comparisons of scaling performances between the Volta and Ampere GPU architectures. Our results also show that the difference between using Volta and Ampere GPUs diminishes as one scales to many nodes, a tendency noticed after only 8 nodes a difference becomes only 36 seconds per epoch, whereas the timings on a single node is 485 seconds and 200 seconds, respectively for a single Volta and Ampere GPU. Some suggestions for future work are also included.

Sammendrag

Det er en økning i interessen for å vise viktigheten av HPC for kunstig intelligens og kunstig intelligens for HPC. HPC-sentre kan nå skryte av større og større datakraft, med flere sentre som når exascale. CERN undersøker bruken av kunstig intelligens og maskinlæring for å utvide eller erstatte noen av de tradisjonelle arbeidsflytene i LHC-eksperimentene. Fordeler med maskinlæring og kunstig intelligens er deres svært parallelliserbare natur på passende maskinvare, for eksempel GPUer.

MLPF, som alle andre storskala kunstig intelligens modeller, bruker store dataressurser for å bli effektiv og nøyaktig. Modellene og deres datasett fortsetter å øke i størrelse, noe som øker behovet for flere dataressurser ytterligere. Arbeidet i denne oppgaven inkluderer å implementere en distribuert versjon av Graph Neural Network MLPF ved å bruke Horovod-rammeverket for å skalere applikasjonene til superdatamaskiner i exascale-klassen. Horovod er et veletablert rammeverk for distribuert arbeidsmengde innenfor kunstig intelligens-feltet.

Arbeidet vårt bruker Horovod-rammeverket til å distribuere arbeidet på tvers av opp til 292 superdatamaskinnoder hver med opptil 4 GPUer, dvs. at det kjører med opptil over 1100 GPUer på Jülich Juwels superdatamaskinen. Det fokuseres på eksperimenter på Nvidia Volta- og Ampere-arkitekturen, da disse var de beste tilgjengelige for oss under denne oppgaven. Under disse skaleringstestene observerer vi at ytelsen skalerer godt opp til 24 noder. Spesielt ser vi en speedup på opptil 20X på 24 noder, mens speedup reduseres til 50X når man går til 100 noder.

Opgaven inkluderer også sammenligninger av skaleringsytelser mellom Volta- og Ampere GPU-arkitekturene. Resultatene våre viser også at forskjellen mellom å bruke Volta og Ampere GPUer avtar når man skalerer til mange noder, en tendens som merkes etter bare 8 noder blir en forskjell på bare 36 sekunder per epoke, mens timingen på en enkelt node er 485 sekunder og 200 sekunder, henholdsvis for en enkelt Volta og Ampere GPU. Noen forslag til fremtidig arbeid er også inkludert.

Acknowledgements

Firstly, I would like to especially thank Dr. Maria Girone, CTO of CERN open-lab, for being my co-advisor and having me onboard her team at CERN, through the CERN technical student program and to the Norwegian research council for providing partial-funding for my stay at CERN. Secondly, I want to give a thanks to all my colleagues at CERN especially the HPC group lead by Maria, including Eric Wulff, David Southwick and Eduard Cuba for all their help and support to make this work possible.

I would also like to thank my supervisor, Prof. Anne C. Elster for her support, insight and knowledge. I would not have managed this thesis or worked at CERN without her.

I would also like to thank the Center of Excellence RAISE for all their help, funding and support, including giving access to the Juelich systems which was the basis for a major part of this thesis. I would also like to thank Ian Fisk and the Simons Foundation for their access to the Flatiron system which was used to implement the Horovod framework on the MLPF model and benchmark the MLPF model on different GPU architectures.

Additionally, I would like to thank the HPC-Lab at NTNU for their ideas and conversations, especially like to thank Tor Andre Haugdahl and Mathias Chunnoo for their knowledge and discussions.

Lastly, I want to give thanks to my family and friends for their support during this project.

Project Description

All LHC experiments are investigating Artificial Intelligence (AI) and Machine Learning (ML) applications to augment or replace some traditional workflows. An advantage of deploying ML or AI algorithms for physics tasks is represented by the fast execution time of the inference, which is highly parallelizable on suitable hardware. Hence, this new generation of algorithms could result in a more efficient usage of the computing resources. The training of AI-based algorithms is an attractive use-case for HPC centers, including CERN as we look towards exascale computing.

This project will include the task of demonstrating the utility of HPC centers for High Energy Physics (HEP) ML or AI training and inference workflows, by re-designing a current ML-based LHC application to use a large distributed supercomputer-class system. The study should include an analysis of the ability of the re-designed applications to scale to hundreds, if not over 1000 GPUs, using distributed training techniques. A study of the training speed on at least two different GPU architectures should also be included as well as performance studies on two larger compute clusters, assuming availability.

Contents

Abstract	iii
Sammendrag	v
Acknowledgements	vii
Project Description	ix
Contents	xi
Figures	xiii
Tables	xv
Acronyms	xvii
1 Introduction	1
1.1 Contributions	2
1.2 Outline	2
2 Background and Related Work	3
2.1 NVIDIA GPU Frameworks	3
2.1.1 NVLink	4
2.1.2 Tensor Core	4
2.1.3 GPU Architectures	6
2.2 Parallelism	6
2.2.1 Model	6
2.2.2 Data	7
2.3 Message-Passing Interface	7
2.4 Deep Learning	9
2.4.1 Deep Neural Networks	10
2.4.2 Hyper-Parameter Optimization	11
2.4.3 Graph Neural Network	11
2.4.4 Machine-Learned Particle-Flow	12
2.5 TensorFlow	13
3 Methodology	15
3.1 Disitributed Training	15
3.2 Horovod	17
3.3 HPC Testbeds	18
3.3.1 Flatiron	18
3.3.2 Juelich	19
3.3.3 Installing required packages and the dataset	19
3.3.4 Processing the Results	20

3.4	Implementation and Testing	21
3.4.1	Flatiron	21
3.4.2	Flatiron vs Jureca	22
3.4.3	Flatiron MPIRUN vs SRUN	24
4	Results & Discussion	27
4.1	Juelich Results	27
4.2	Flatiron Results	29
4.3	Discussion	31
5	Conclusion & Future Work	35
5.1	Future Work	36
	Bibliography	37

Figures

2.1	FMA multiplication on newer NVIDIA GPUs [Source:NVIDIA]	5
2.2	Computations done in order to create TF32 with FP16 matrices [Source: NVIDIA]	6
2.3	Showing the result of a reduce routine Source: NVIDIA	8
2.4	Displaying showing the result of a AllGather routine [Source: NVIDIA]	9
2.5	Displaying the result of a ReduceScatter routine [Source: NVIDIA]	9
2.6	The structure of how a DNN is built up. Based on Figure 4 in Deep Neural network for Predicting Ore Pro- duction by Truck Haulage systems in Open Pit Mines [25]	11
3.1	Logical ring in which the GPUs communicate with each other. . . .	17
3.2	The software packages used for running the multinode training and the initialization command	20
3.3	Speedup compared to scaling of nodes on Flatiron	21
3.4	Average Time per Epoch for a set of amount of nodes	22
3.5	Speedup compared to scaling of nodes on both systems	22
3.6	Average Time per epoch for a set amount of nodes compared to Jureca	23
3.7	Speedup compared to scaling of nodes with GPUDirect RDMA en- abled	24
3.8	Average Time per epoch for a set of nodes with GPUDirect RDMA enabled	24
3.9	Speedup on the Flatiron system with SRUN launch command	25
3.10	Average Time per epoch on the Flatiron system with SRUN launch command	25
4.1	Scaling performances on Jureca	28
4.2	Scaling performances on Juwels & Jureca up to a hundred	28
4.3	Scaling performances on Juwels & Jureca up to 292 nodes	29
4.4	Average Epoch Time on Juelich	29

4.5	Amount of throughput in the form of steps iterated for all of the GPUs	30
4.6	Scaling performances Volta versus Ampere	30
4.7	Average time per node Volta versus Ampere	31

Tables

2.1	Transfer rates of different versions of NVLink [7–9]	4
2.2	Data for this table is found in the Volta- and Ampere architecture whitepapers [8, 9]	6
3.1	GPU specification on HPC testbeds	19
3.2	Software versions on the different clusters	20
4.1	The steps per epoch for a set number of nodes	32

Acronyms

AI	Artificial Intelligence.	1
ATLAS	A Toroidal LHC Apparatus.	1
CERN	European Organization for Nuclear Research.	1
CMS	Compact Muon Solenoid.	1
CUDA	Compute Unified Device Architecture.	3
cuDNN	CUDA Deep Neural Network.	3
DL	Deep Learning.	9
DNN	Deep Neural Network.	10
FMA	Fused Multiple-Add.	4
GNN	Graph Neural Network.	11
GPU	Graphics Processing Unit.	3
GT/s	Gigatransfers per second.	4
HEP	High Energy Physics.	ix
LHC	Large Hadron Collider.	1
ML	Machine Learning.	1
MLPF	Machine Learned Particle Flow.	12
MPI	Message-Passing Interface.	7
NCCL	NVIDIA Collective Communication Library.	3
NTNU	Norwegian University of Science and Technology.	1

PF Particle Flow. 12

PU Processing Unit. 1

SoC System on Chip. 3

SPMD Single Program Multiple Data. 7

TFLOPS Terra Floating Point Operations Per Second. 4

Chapter 1

Introduction

High-Performance Computing (HPC) is a field within Computer Science that focuses on the usage of supercomputers and computer clusters in order to solve advanced and computationally demanding problems, such as weather calculations, prime generation, and artificial intelligence. Supercomputers or computer clusters are different from traditional computers as these machines have multiple Processing Unit (PU), a set of N PU will create a node, a supercomputer/cluster is a vast amount of nodes combined together through interconnects, making it possible to work as one unit to solve problems. Artificial Intelligence (AI) is one of the newer fields that benefit from the throughput of such systems. At CERN they are investigating AI as an augmentation or replacement of some traditional workflow.

This thesis is a product of a collaboration between European Organization for Nuclear Research (CERN) Openlab and the Norwegian University of Science and Technology (NTNU). CERN is a collaborative research lab with researchers from all over the world, created in 1951 by the European Council to further the understanding of particle physics. CERN Openlab is a collaboration between industry and CERN, where CERN collaborates with leading IT companies and top universities in developing cutting-edge ICT solutions for the experiments within CERN, e.g Large Hadron Collider (LHC), Compact Muon Solenoid (CMS) and A Toroidal LHC Apparatus (ATLAS). Most of the data and research conducted in this thesis were done during a technical internship at CERN Openlab.

All LHC experiments are investigating AI and Machine Learning (ML) applications to augment or replace some traditional workflows. An advantage of deploying ML or AI algorithms for physics tasks is represented by the fast execution time of the inference, which is highly parallelizable on suitable hardware. Hence, this new generation of algorithms could result in a more efficient usage of computing resources. The training of AI-based algorithms is an attractive use case for HPC centers, including CERN. This project will include the task of demonstrating the utility of HPC centers for HEP ML/AI training and inference workflows, in particular for reconstruction workflows. This will include the ability to scale hundreds

of GPUs using distributed training techniques. A study of the training speed on different GPUs will also be included.

1.1 Contributions

The goals of this thesis is to:

- Implement and test a distributed training and inference method on the artificial intelligence model MLPF
- Perform scaling experiments towards exascale on HPC systems using the implemented distributed training method on MLPF
- Perform scaling experiments with multiple GPU architectures and compare scaling efficiency.
- Speedup the training process of the MLPF model

The main contributions of this thesis are:

- An successful implementation of a distributed method on the artificial intelligence model MLPF
- Scaling results of a graph neural network on multiple GPU architectures and HPC clusters
- An discussion of the importance of which HPC will have for AI and AI will have for HPC

1.2 Outline

The outline of the rest of the thesis is as follows:

- **Chapter 2** Description of relevant background information used in both the methodology and discussion
- **Chapter 3** Detailed description of the implementation of the distributed framework Horovod, how the experiments were performed, and the progression of the implementation of Horovod
- **Chapter 4** Presents the results from the various scaling tests performed and discussions surrounding the results. The results include scaling the MLPF model towards exascale, comparison of scaling with earlier GPU architectures, and discussion towards the Horovod framework scaling results
- **Chapter 5** Conclusion and future work of this project

Chapter 2

Background and Related Work

This chapter gives an overview of the most important and relevant background information related to the topic of this thesis. The following section will provide information about NVIDIA's deep learning frameworks, different GPU features, and architectures, deep learning and graph neural networks and end with detailing the MLPF model.

2.1 NVIDIA GPU Frameworks

NVIDIA is a multinational technology company focusing on Graphics Processing Unit (GPU) for use in both commercial and enterprise settings, including System on Chip (SoC) for cars and telephones. NVIDIA GPUs are preferred for Deep Learning, Section 2.4, due to their software toolset that is freely available. The most important tools for this thesis will be detailed below.

Compute Unified Device Architecture (CUDA) is a development library and tool-chain that eases the developer experience when programming for NVIDIA GPUs. It is the direct API connection between the hardware and the programming languages C/C++, and provides the most important features for intercommunication between nodes and GPUs in large clusters [1].

CUDA Deep Neural Network (cuDNN) is a library specific for GPU accelerated primitives for deep neural networks, providing highly tuned implementations of common deep neural network functions; convolution, activation layers, pooling, and others.[2, 3]

NVIDIA Collective Communication Library (NCCL) is a communication library, providing optimized communication primitives for NVIDIA GPUs and Networking. Providing NVIDIA optimized parallel communication routines: All-reduce, All-gather, Reduce, Broadcast, Reduce-scatter, and several forms of Send & Receive [4]. It is implemented and optimized for high bandwidth communication

on PCIe, NVLink, NVswitch TCP/IP and InfiniBand, and is scalable to an arbitrary amount of GPUs on a single or multiple nodes, either used with single- or multi-process applications [5].

2.1.1 NVLink

NVLink is a serial multi-lane near-range communication link developed first by Nvidia in 2014 [6]. NVLink offers a more scalable and faster interconnect in comparison to traditional PCIe interconnects. NVLink is supported for GPU communication, GPU - CPU and GPU - GPU. The first version of NVLink had a transfer rate of 20 Gigatransfers per second (GT/s). [7] In the newer releases NVIDIA has released new versions of NVLink which has further increased the transfer rate of the device, ref Table 2.2

Table 2.1: Transfer rates of different versions of NVLink [7–9]

NVLink comparison			
Name	Version	Transfer rate	Architecture
PCIe	4.0	16 GT/s	Standard
NVLINK	1.0	20 GT/s	Pascal
NVLINK	2.0	25 GT/s	Volta
NVLINK	3.0	50 GT/s	Ampere

2.1.2 Tensor Core

Tensor cores are special cores used in the newer NVIDIA GPUs, produced after 2016, and are specially made for Deep Learning. Tensor cores are mainly used and optimized for matrix manipulation. [8] The Turing architecture is not taken into consideration for this thesis, but did provide the following architectures; Deep Learning Super Sampling and the INT8 and INT4 precision [10].

Volta (1st Generation Tensor Core)

The Volta GPU architecture was first introduced in 2017 [8]. Tensor cores are special cores designed for Deep Learning and focus on matrix operations. A single core performs 64 x 16 bit floating point Fused Multiple-Add (FMA) operations per clock operation, providing a potential of 125 Terra Floating Point Operations Per Second (TFLOPS) for both training and inference. The tensor cores improve the calculation speed from the prior architecture, Pascal, by 12x on FP32 and 6x on FP16 [8].

Tensor cores works by doing FMA calculation, Equation 2.1, on 4x4 matrices, two input matrices with a FP16 input and a FP32/FP16 accumulation matrix, resulting

in either a FP32- or FP16 result matrix depending on the accumulation matrix.

$$D = A * B + C \quad (2.1)$$

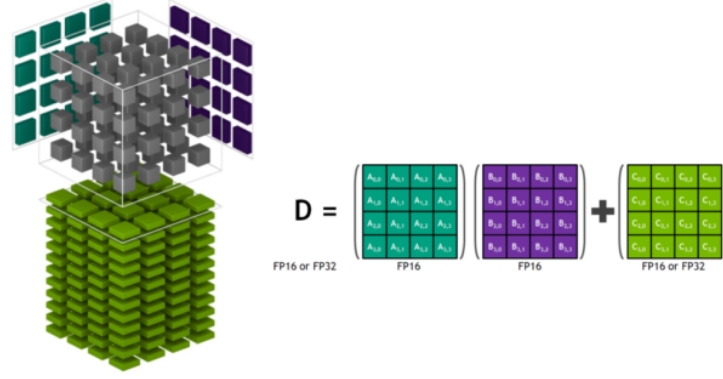


Figure 2.1: FMA multiplication on newer NVIDIA GPUs
[Source:NVIDIA]

NVIDIA Ampere (3rd Generation Tensor Core)

The Ampere architecture introduced the third generation Tensor cores, demonstrates overall performance increase for the existing data types supported and highlights new precision modes specific for AI-based computation, TF32 and BFloat16. TF32 is a precision mode that works with a 8-bit exponent (FP32) and 10-bit mantissa (FP16). It provides 10x the throughput of a V100 FP32 calculation while still maintaining close to the same accuracy. On the other hand, BFloat16 is an alternative to FP16, both (FP16 & BF16) have shown success in mixed-precision training. For this thesis, the mixed-precision is not within the scope and the model is focused on the FP32/TF32 accuracy. It also provides a speedup for sparse matrix calculations however this is beyond the scope of this thesis. [9]

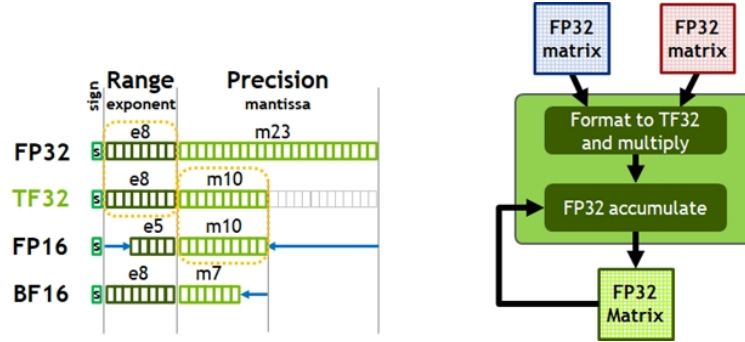


Figure 2.2: Computations done in order to create TF32 with FP16 matrices
[Source: NVIDIA]

2.1.3 GPU Architectures

The table below shows the improvement of the most important features from the different architectures compared in this thesis.

Table 2.2: Data for this table is found in the Volta- and Ampere architecture whitepapers [8, 9]

	Volta	Ampere	Difference
FP32 cores	5120	6912	1792 more
Tensor Cores	640	432	208 less
FP32	15.7 TFLOPS	19.5 TFLOPS	1.25x
A100 TF32 vs V100 FP32	15.7 TFLOPS	156 TFLOPS	10x

2.2 Parallelism

Parallelism is the job of splitting a workload between multiple machines in such a way that either each machine does a part of the job or processes a certain chunk of the data. With that there two main categorize of parallelism, Model and Data.

2.2.1 Model

Model parallelism is a technique that can be used for distributed neural network training, across both multiple devices and nodes. When applied, each device executes a unique part of the network, working in unity. Examples of this are large AI models, where different layers can be spread to different GPUs [11]. One side effect of distributed training is the increase of larger communication overhead [12]. Model parallelism solves this with the increase of devices, usually significantly increasing the start-up time of the network. In contrast to data parallelism, Section 2.2.2, model parallelism's reconfigurability can deeply improve on this overhead. Due to the difference in data transfer sizes and connection numbers,

utilizing model parallelism with the knowledge of the network architecture one can create more suitable dimensions for parallelism [11]. This form of parallelism can give solutions to problems where the model has grown too large to fit on a single device; by spreading the model across multiple devices the memory consumption per device decreases to an acceptable level. [13]

2.2.2 Data

A more common parallelization method is where the data is distributed over multiple devices, also called data parallelism. In artificial intelligence, the most used implementation of data parallelism is that the entire model is replicated across all devices. Each device will then be in charge of processing a part of the total input data, with both forward and backward passes. This results in parallelizing the computational load, providing equal load balancing between the devices; a result of executing the same amount of computation[14].

As the data is spread across the devices, a synchronization method is needed in order to synchronize the devices and average the results. This synchronization technique is either centralized or decentralized [14]. In the centralized case, a parameter server is responsible for the updates for the workers and can either be a designated CPU node or simply the device with rank 0. The parameter server is responsible for all updates on the trainable parameters, and the devices only send and fetch parameters from the server. With decentralized synchronization every device is responsible to send and receive the parameters to each other, ensuring that all devices are updated with the same parameters. [15]

Both these methods provide significant communication overhead as the amount of communication required to synchronize all the devices scales linearly with the number of devices, reducing the overall utilization of devices due to the blocked state waiting for synchronization to finish. Due to this overhead, networks with a large computational load or sparsely connected networks are preferred for data parallelism, as the speed-up in computations is more likely to outweigh the overhead from the communication[15].

2.3 Message-Passing Interface

Message-Passing Interface (MPI) is a library specification defined to primarily address the message-passing parallel programming model, where data is moved between processes through various operations on each process [16]. The specification defines the names, calling sequences, and results of cooperative operations, meaning one process cannot force a message on another without the recipient actively listening. Programs utilizing an MPI implementation follow the model Single Program Multiple Data (SPMD) with the same code executed in all the processes. However, the data operated upon is different for each process [17]. There are

multiple MPI implementations, where each parallel computer vendor offers an implementation for their machines, along with other free publicly available implementations such as OpenMPI[18] and MPICH[19].

The most commonly used communication routines are:

- AllReduce
- AllGather
- Reduce
- Gather
- Broadcast
- ReduceScatter
- Send
- Recieve
- SendRecieve

Reduce is one of the basic communication routines and is used as a standalone routine, but also in combination routines such as AllReduce & ReduceScatter. The reduce routine works similarly to the functional programming method reduce, involving reducing a set of data into a smaller set of data through a mathematical function. For every rank you want to reduce, you receive the data and instead of locally storing it you perform a function to a local variable in this case you sum the values, shown in the figure below[20].

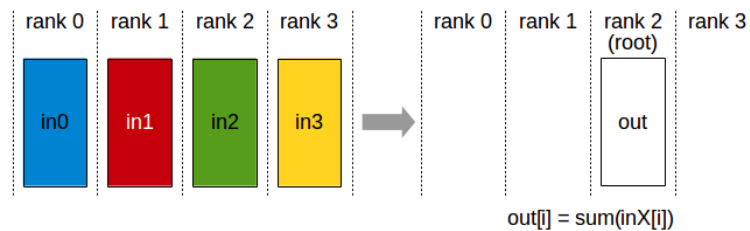


Figure 2.3: Showing the result of a reduce routine
Source: NVIDIA

AllReduce is a routine that combines the two basic routines, reduce and broadcast. The routine starts with the root rank initiating a Reduce between all ranks, the result of the reduce is then broadcasted to every rank. Each rank will then have the same results[21].

AllGather routine is similar to the Gather routine, but instead of only the root rank receiving all the results they are received by every rank. It works as follows: after each rank has completed its computations, they will all have a piece of the full result. Each rank will then share its result with every other rank, creating the full result on each rank. Each rank K shares its N values with the other ranks, creating a result of size $N * K$ chunks on every rank[21].

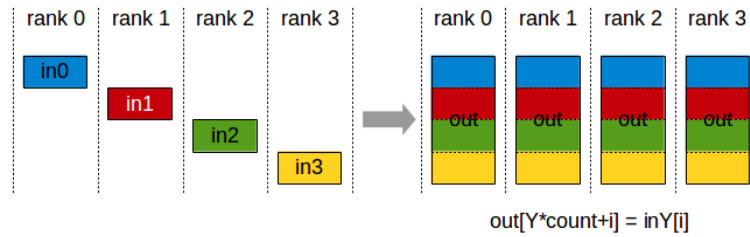


Figure 2.4: Displaying showing the result of a AllGather routine
[Source: NVIDIA]

ReduceScatter is a routine combining the Reduce- and Scatter operation; reducing the values of all inputs and then scattering the results in equal chunks to all the ranks [22].

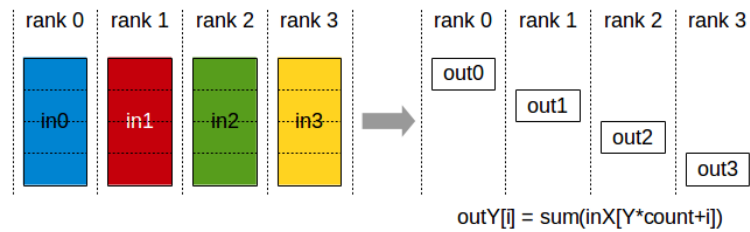


Figure 2.5: Displaying the result of a ReduceScatter routine
[Source: NVIDIA]

2.4 Deep Learning

Deep Learning (DL) is a neural network with three or more layers. It is a network trying to simulate the human brain's behavior in specific tasks, learning from large amounts of data. In general, there are numerous tasks; classification, denoising, regression, anomaly detection, and so on [23]. Deep learning models are characterized by how well it performs on specific tasks by estimating the accuracy of the predictions on data not in the training dataset and in some cases the loss. The loss is a value created from a loss function, which is a method to evaluate how well the model models the dataset. It is a measurement of how well the model predicts correctly. In DL or ML one can categorize the training of the model as two main algorithms, supervised and unsupervised, depending on how the dataset looks and what the model is going to learn [24].

Unsupervised learning uses datasets where the features of the data are many and tries to learn patterns/properties of the structure of the dataset. These algorithms can be used to group/cluster large amounts of data into more manageable sets or more generally denoising, synthesis, and so on. More generally it involves observing several examples of a random vector $\mathbf{x}$ and attempt to figure out the probability distribution $p(\mathbf{x})$ alongside other interesting properties [23].

Supervised learning is where the dataset is built up differently, still containing a set of features, however, every example in the dataset is associated with a label. The goal is to infer the function $f : X \rightarrow Y$ for the given dataset. It does this by observing several examples of a vector $\mathbf{x}$ and the associated value $\mathbf{y}$, then inferring a function $f(\mathbf{x}) = \mathbf{y}$. This is usually done by estimating the probability of *if $\mathbf{y}$ then $\mathbf{x}$* or $p(\mathbf{y}|\mathbf{x})$ [24].

2.4.1 Deep Neural Networks

Deep Neural Network (DNN) is the model and/or system for the algorithms to learn. It is built up of a multitude of perceptrons in a layer, the most basic unit for a deep neural network. A model consists of multiple layers, an input, an output, and N hidden layers. The occurrence of the layers are input, N x hidden, and finally the output layer. Each layer consists of numerous perceptrons, and the number of perceptrons per layer may vary[24].

The input layer is the one receiving the data and spreads it to the first hidden layer. The hidden layer is where the computation happens, ref equation 2.2. The output layer is where the result is found. The perceptron will take an arbitrary input $\mathbf{x}$ from the previous layer, e.g a combination of all the valid results from the previous layer. They will multiply this value with their pre-calculated unique weight, $\mathbf{W}$, add a bias/threshold value, $\mathbf{b}$, and finally apply the layers activation function, σ . The activation function is the one determining if the value calculated, Equation 2.2, is sent on to the next layer. The activation function for each layer can vary, with the most used functions being linear, ReLU, and Sigmoid. The use of an activation function determining if the value is sent to the next layer is the basis for forward propagation, which is one of the most used methods for training. Another method is backward propagation which travels back through the layers after an iteration and adjusts the weights to minimize large divergence [23].

$$f(\mathbf{x}) = \sigma(\mathbf{x} * \mathbf{W} + \mathbf{b}) \quad (2.2)$$

The output layer shows the probabilistic results for the data being one of the pre-determined result categories or types. An example of a model is one used to determine if there is a car in the image, the output layer takes chunks of pixels/-plain pixels and spreads those to the first layer. Each of the perceptrons determine on a different basis from each other, e.g a perceptron might determine if there is a side door window in the data, and in the next layer a perceptron might determine if there then is a door with a side door window. At the end in the output layer, you get a statistical representation of whether there is a car in the image, or not.

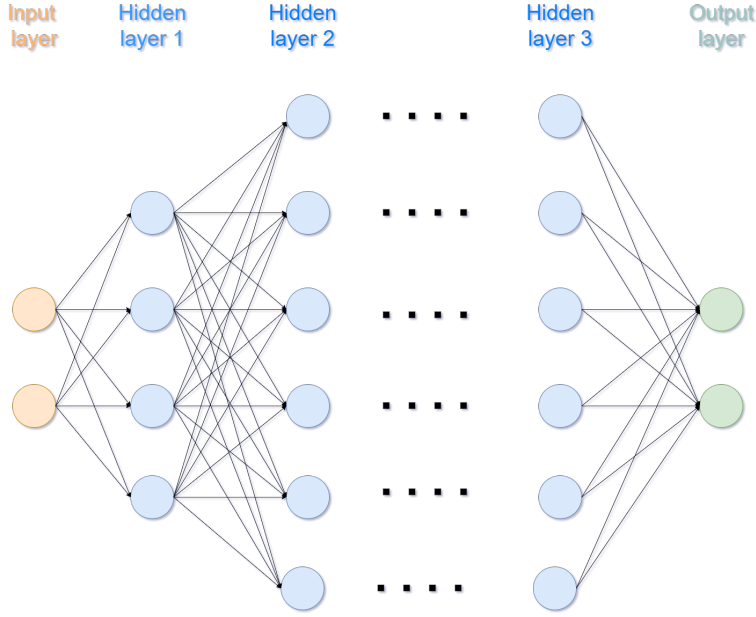


Figure 2.6: The structure of how a DNN is built up.
 Based on Figure 4 in Deep Neural network for Predicting Ore Production by Truck
 Haulage systems in Open Pit Mines [25]

2.4.2 Hyper-Parameter Optimization

Hyper optimization or tuning is in Deep Learning or Machine learning the problem of choosing the optimal set of hyper-parameters for the learning algorithm. A hyper-parameter is a parameter which is controlled by the user before starting training session. These parameters should not be differentiated with model parameters which are the parameters that are changed during the learning process [26].

2.4.3 Graph Neural Network

Graph Neural Network (GNN) is a neural network built around the graph data-type [27], and is especially useful for non-euclidean data or data bound by nature [28] but is not restricted only to this type. Such data can be social networks, molecules, and physics. For euclidean types, it can be images and text[29–31], for it to work on euclidean data, the data needs to be transformed into its respective graph format. The data is associated with the goal of either supervised or unsupervised learning, where a function t maps a graph G and one of the nodes n to a vector of reals or truths: $t(G, n) \in \mathbb{R}^m$. The domain focuses on three cases, graph, node, or edge[32].

Graph focused case, the focus is to predict the property of the entire graph. The function t is therefore independent of the node n and classifies/regresses based

upon the entire graph, $t(G)$ [33]. An example of this is how you can teach a network to determine if a protein is an enzyme or not in bioinformatics.

Node focused case, the focus is to predict the task or identity of each node n within the graph G . The result of the function t is pendent on the node n , and the result for each node depends on the properties of the given node, which can also include the adjacent nodes [33]. An example could be to predict the subject of a given paper based upon the network of words and citation

Edge focused case, the focus is to predict the role or relationship between two nodes[33]. An example here is in a social network you could determine the relationship between two friends or a potential relationship between two individuals.

2.4.4 Machine-Learned Particle-Flow

Machine Learned Particle Flow (MLPF) is an AI model constructed using the neural network class graph neural network (GNN). The particle-flow reconstruction process, which is often laborious and computationally intensive, is eased by using the MLPF model. Such event reconstruction methods are crucial for providing a comprehensive, precisely calibrated physics interpretation of the collision event and are utilized in many experiments at CERN, including CELLO, DELPHI, CMS, ATLAS, and others.

One of these reconstruction approaches is Particle Flow (PF). The approach is used to recreate a "particle-level interpretation of high multiplicity hadron collision events using individual detector components from a multi-layered, heterogeneous, irregular-geometry detector, such as calorimeter clusters and tracks. In order to reconstruct charged and neutral hadron candidates as well as photons, electrons, and muons with optimized efficiency and resolution, the PF algorithm typically correlates tracks and calorimeter clusters from detector layers such as the electromagnetic calorimeter (ECAL), hadron calorimeter (HCAL), and others." Since the features and shape of the detectors are essential for optimal physics performance, simulations have been used to fine-tune each version of the algorithm for each experiment.

DELPHES

MLPF supports a variety of datasets, but for the sake of this thesis, the Delphes dataset—which was developed as a benchmark for the MLPF model—will be the main emphasis. Delphes 3 was used to model the response of a multi-functional detector, while Pythia 8 was used to generate high-energy physics collision events. With 50K top quark-antiquark events from proton-proton collisions at 14 TeV overlaid with minimal bias events, which have an average PU of 200, it is a training dataset for particles. It also includes 5,000 created events used for validation,

which are evaluated using quantum chromodynamic multijet events in a separate physics domain. The inputs are detector hits, the ground truth is generated particles, and DELPHES reconstructed particles are used for extra validation.

2.5 TensorFlow

TensorFlow is an open-source library created by Google's Brain team as an end-to-end platform for Machine learning, developed with the goal to improve and standardize the machine learning development.[34]. Done through the usage of high-level APIs, the process of creating and training neural networks is simplified. TensorFlow is created through a collective of larger APIs for a number of different languages, Python, C++, Java and Go. of these, the most used language is Python [35]. TensorFlow represents computation, shared state and mutate operations as data flow graphs. By having these represented as data flow graphs it enables TensorFlow to map the information across multiple machines, in clusters and architectures. This architecture improves on the imperfect designs of the "parameter server" where the management of the shared state is built in the system.[36]

A package complementing TensorFlow is Keras, which is a high-level API built upon TensorFlow 2, it is an industry-strength framework that easily can run at exascale due to its scalability to large GPU clusters and even entire TPU pods. The purpose of Keras is to mediate quicker and easier creation and experimentation of new networks. It offers a well-documented API with a wide range of standardized neural network functionality and high flexibility and mutability to implement custom modules.[37] One of the features in which Keras is well suited for is the callback functionality; a set of functions that can be specified to run at certain steps or epochs during training [38]. Such callbacks can be integrated callbacks created from Keras or custom self-implemented callbacks. [37]

Chapter 3

Methodology

This chapter will give an introduction to the thought process for the implementation of the multi-node configuration of the MLPF model and the reasoning for the choices, including some information on how the experiments were run. Section 3.1 will detail the distributed training options within TensorFlow and why these were not chosen. Section 3.2 gives an introduction to the selected framework for the multi-node task, and gives an explanation of how the framework works and its benefits. Section 3.3 will detail the differences between the systems used and how the benchmark was done. Section 3.4 explains the thought process and iteration of how the Horovod framework was implemented and the results following the implementation tests

3.1 Disitributed Training

Within TensorFlow, Section 2.5, there are strategies implemented for multiple GPUs - single node and multiple GPUs - multiple nodes, both of which support a multitude of different configurations and systems, E.g. SLURM, GCE, AWS, and more. Most notable are the three strategies:

- MirroredStrategy for multiple GPU - single node
- MultiWorkerMirroredStrategy for multiple GPU - multiple nodes
- ParameterServerStrategy for multiple GPU - multiple nodes

These strategies are detailed below, with an explanation for why they were or were not used for the final result.

MirroredStrategy

This is a generic strategy for synchronous distributed training on one machine with multiple GPUs. It is also a data parallelism strategy that works by creating a replica per GPU, where all variables in the model are mirrored. These variables are kept in sync by applying identical updates. These updates are communicated through the usage of all-reduce algorithms, detailed in chapter 2.3, aggregating tensors across

all devices and adding them up. By default, the NCCL all-reduce algorithm is used [39]. This strategy was already implemented on the MLPF model before the thesis started, and is the backbone for where the thesis has reached. The results from this were not interesting as one were looking at the bigger picture, scaling the model towards exascale.

ParameterServerStrategy

ParameterServerStrategy is a strategy for distributed training over multiple nodes. It works by having a cluster of both workers- and parameter servers, where the parameter servers are where the workers read and update their parameters, weights/gradients. After the training is done the values of the parameter servers are processed, often averaged, and the result is presented. This method is defined as asynchronous as the workers never communicate with each other, they only read and update the variables in the parameter server independently. This method required a bigger overhead in the configuration aspect, especially when switching between different clusters [40].

MultiWorkerMirroredStrategy

This is a similar distributed strategy to MirroredStrategy. Implementing synchronous distributed training across multiple nodes, with the potential of scaling training on CPUs and GPUs. Mirroring copies of variables in the model across all workers. Different from the MirroredStrategy is that when updating variables between the nodes multiple strategies can be selected, depending on the problem and system. Most used is the parameter-server strategy which is explained in a later section of this thesis, and the central storage strategy, where the variables are not mirrored across devices only the operations. Between the nodes, only the variables are mirrored and they are stored as close to the CPU as possible [41].

Initially, this strategy was the selected implementation for multi-node scaling, as this implementation supported SLURM configurations out of the box, meaning just by looking at the SLURM environment variables it could determine everything it needed to know in order to connect to every node in the system. This was thought to be the solution in order to scale the model, but with testing the strategy a problem surfaced with the connection between the nodes. When running the model with two nodes the strategy would not connect. When the A100 cards were used, the system would go into an endless loop trying to find all the nodes. Doing the exact same test with the V100 cards surprisingly resulted in a full connection, and the model managed to train/run for 1 epoch before crashing. After doing further research into the problem, it was determined that this was a "black box" problem, as the debug information from both MPI, NCCL, and TensorFlow did not give any indication as to why the problem persisted on the A100 but not the V100 cards. This was tested on both systems. The solution to scaling the problem ended up being a third-party framework called Horovod.

3.2 Horovod

Horovod, a distributed deep learning framework, originally created by UBER as a substitution to the Parameter Server, Section 3.1, solution provided by TensorFlow in 2018. Developed with MPI, it gives less pragmatic changes to the code base, in contrast to using the built-in distributed configuration provided by TensorFlow in 2018. TensorFlow has since implemented new standards for distributed training with the strategy system discussed above; this requires less code change in models developed in TensorFlow. Since the creation of the Horovod framework, it has been made publicly available and is now an open-source project funded by the LF AI & Data Foundation, helping cut down training time and making distributed training easier and more accessible. Horovod is configured to run with the Ring-AllReduce, a method minimizing the cost of communication between the different workers/GPUs when sharing results [42, 43].

Ring-AllReduce

Ring-AllReduce is a communication method introduced by researchers from Florida State University for handling large-sized data communication in cluster environments [44]. It is a collective communication method for optimal usage of the bandwidth. The method is only optimal in the bandwidth region, however, the latency is proportional to the amount of processes. The Ring-AllReduce works by using the MPI operations scatter-reduce and allgather. The GPUs are "linked" in a logical ring where each GPU has a specific neighbor for sending and receiving, ref Figure 3.1. All the GPUs scatter-reduce their results to their receiving neighbor, and each GPU ends then up with a chunk of the final result. Next, the GPUs perform allgather to exchange those chunks until all the GPUs have received the complete final result.

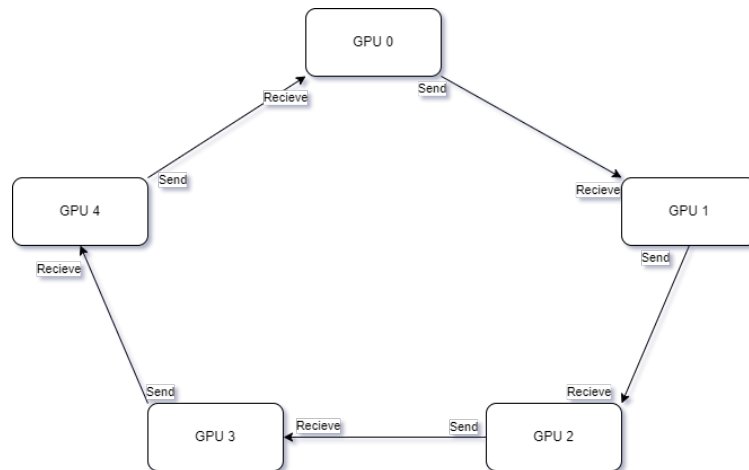


Figure 3.1: Logical ring in which the GPUs communicate with each other.

Why Horovod?

Horovod became the selected implementation on the MLPF model as this is one of the largest contributed distributed frameworks. It is used with a lot of different networks and AI models, and it solves the biggest problem with implementation, that being that it is easily run on multiple clusters. The framework is not system dependent meaning it can run on any cluster without extra configuration, so all it needs is to be installed with the desired communication packages available, E.g. MPI, NCCL, GLOO etc. For the configuration of the Horovod used in the experiments in this thesis, the packages NCCL, OpenMPI, TensorFlow and Keras were selected. This was all installed through the python package manager, *pip*, following the instruction written in the Horovod documents.

```
HOROVOD_WITH_TENSORFLOW=1 HOROVOD_WITH_MPI=1 HOROVOD_GPU_OPERATIONS=NCCL  
pip install horovod[tensorflow,keras]
```

As there is no simple way of showing how Horovod was implemented one can take a look at the pull request on GitHub. Available here:
<https://github.com/jpata/particleflow/pull/104>

3.3 HPC Testbeds

Throughout this thesis, two HPC centers were used for performing the scaling performance benchmarks, Juelich and Flatiron. In terms of hardware spec, these systems are similar, both boasting a large quantity of high-end enterprise NVIDIA GPUs. For the benchmark, the centers were each used for different experiments. Most of the preliminary work around the implementation of distributed workload with Horovod was done on Flatiron, and the result of this research can be found seen later in this section. This thesis focuses on three main points; scaling efficiency towards exascale, scaling performance differences between Volta and Ampere and the importance of HPC for AI in the coming years. The investigation around the scaling performance of the model towards exascale Juelich was preferred, due to its larger quantity of nodes accessible on multiple clusters, the result of this will also be compared to the results shown in the Horovod paper. Flatiron was chosen for the comparison between different architectures, as the centres have clusters with both cards.

3.3.1 Flatiron

A computing center funded by the Simons foundation, currently in two locations, New York (Rusty) and one on the west coast (Popeye). Within this thesis, the Rusty cluster was used. Both Rusty and Popeye are built with the work scheduler SLURM, but with Rusty's selection of GPUs, it was the preferred choice. Rusty is the largest compute center available within Flatiron, boasting a vast number of

nodes with A100 and V100 cards. During this thesis, multiple experiments were performed at this cluster, and the main takeaways from this cluster will be the experiments comparing the different architectures.

3.3.2 Juelich

A computing center in Germany, it boasts the two clusters currently in the HPC TOP500 list, Jureca and Juwels. Jureca was used for the fine granularity runs up to and including 24 nodes when inspecting the scaling performance. For nodes greater than 24 I used the Juwels system. The clusters are both built with the work scheduler SLURM so the configuration stays the same. The only difference between the clusters is in the hardware, more specifically the CPU, which is negligible for the performance compared here. Some of the packages used will have minor differences in patch version.

Table 3.1: GPU specification on HPC testbeds

GPU Compute nodes	Memory module	CPU	GPU	System
192	512GB 3200MHz	2 x AMD 7742	4 x A100	Jureca
936	512GB 3200MHz	2x AMD 7402	4 x A100	Juwels
36	1024GB	AMD	4 x A100	Flatiron
31	768GB	AMD	4 x A100	Flatiron

3.3.3 Installing required packages and the dataset

Installing and running the MLPF model on the two systems was an easy procedure as the model is built on python, which is supported by most systems. The installation was done by following the README in the MLPF repository which required installing the packages in the *requirement.txt*. In order to install Horovod I used the command written in Section 3.2. After the packages were installed, the dataset had to be downloaded, which was best done through running the model on a single node. This download method is a prebuilt method within the model, it could also be downloaded manually but its recommended through the model in order to create the proper folder structure. Once the python packages was installed and the dataset downloaded, one can start performing experiments.

Since all the clusters use SLURM as their work scheduler, it was easy to create a generic slurm script that could be used by all of them. The MLPF repository already had examples of a SLURM script which could be used to launch a single node experiment on flatiron, *mlpf/flatiron/pipeline_train_4GPU.sbatch*; this was used as a baseline. It was configured such that a single node would just launch the model with the python launch command while in the multi node you would launch through the slurm command *srun*, ref Listing 3.2.

In the Horovod documentation it is recommended to use *mpirun* or *Horovodrun*, but while testing configurations on the Jureca systems, it was not possible to use any *mpirun* based commands and *srun* was the only viable option, it was decided to use the *srun* command for both systems. Comparing the initial tests on flatiron with both *srun* and *mpirun* one could see a slight speedup using *srun* over *mpirun*, however, this was not investigated further in this thesis. As a result, the different systems were launched using Figure 3.2, there is a minor difference in runtime as the two systems had slightly different packages available, however, all experiments were run with the same major versions of all software. The software used for the different systems can be seen in Table 3.2.

```
module --force purge; module load modules/2.0
module load slurm gcc cmake cudnn cuda openmpi/cuda

SINGLENODE: CUDA_VISIBLE_DEVICES=0,1,2,3 python3 mlpf/pipeline.py train -c $1 -p $2

MULTINODE: srun --cpu-bind=none python3 mlpf/pipeline.py train -c $1 -p $2
```

Figure 3.2: The software packages used for running the multinode training and the initialization command

Table 3.2: Software versions on the different clusters

Software	Juelich	Flatiron
GCC	10.3.2	10.2.1
CMake	3.18.0	3.22.3
CUDA	2.12.7	2.11.4
cuDNN	11.5	11.4.4
NCCL	8.3.1.22	8.4.2.15
OpenMPI	4.1.2	4.0.7

3.3.4 Processing the Results

To compare the efficiency of the different number of nodes, the output from the TensorFlow framework is processed. The framework presents for every epoch, every full iteration through the dataset, and the metrics of the model. These metrics consist of interesting factors for the scientists of the model, however, within this thesis we will only focus on the time. These metrics can be found in the output logs for every given run, each run is given an experiment name and an auto-generated id in the slurm script. The initial epoch of every run is ignored as the metrics from this iteration contains a lot of false positives, given that the metrics are too closely connected to the random initialization of the weights. The raw data from all the logs are then processed by a python script presenting all the data.

3.4 Implementation and Testing

During the thesis there were multiple experiments building up to the final implementation, this is a collection of different runs on the Flatiron and Juelich systems, showing preliminary results that helped introduce optimization and find bottlenecks within the implementation.

3.4.1 Flatiron

The implementation was started on the Flatiron cluster in New York, on A100 nodes. The experiments were run with *mpirun* as per the official documentation of Horovod, giving the result shown in Figure 3.3. This experiment was conducted 3 times with 10 epochs every run up to 8 nodes. This was an initial scaling test so there was no need for scaling further than 8 nodes. Initially, we assumed this

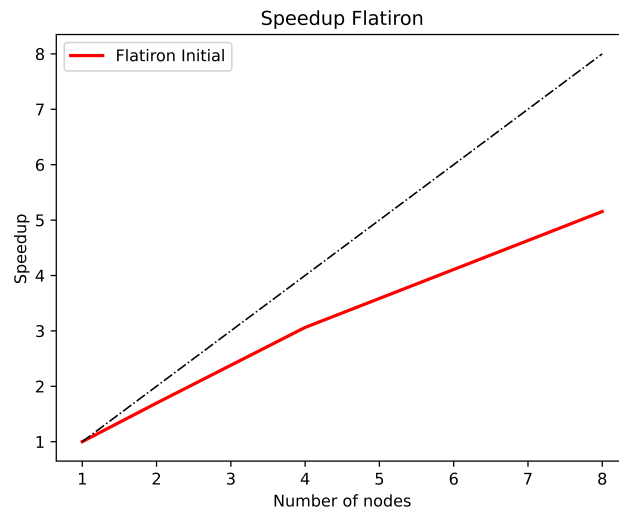


Figure 3.3: Speedup compared to scaling of nodes on Flatiron

result was alright as it showed ok scaling and the dataset wasn't that large. It was not until this result was shown for the CoE RAISE members that the scaling did not look correct for the size of the dataset and the number of nodes. This led to testing the implementation on the Jureca system.

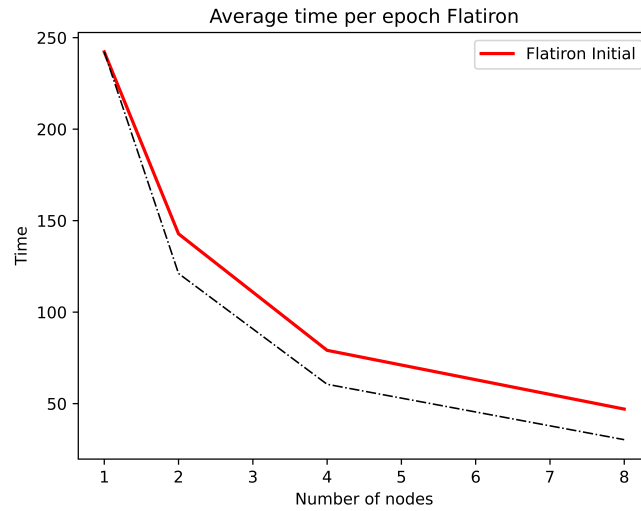


Figure 3.4: Average Time per Epoch for a set of amount of nodes

3.4.2 Flatiron vs Jureca

The RAISE work package which I was a part had access to the Juelich system, which in return gave me access to the Jureca cluster. The results of scaling the system on Jureca can be seen in Figure 3.5, greatly improving on what was seen on the Flatiron cluster. This was great news as it showed the implementation

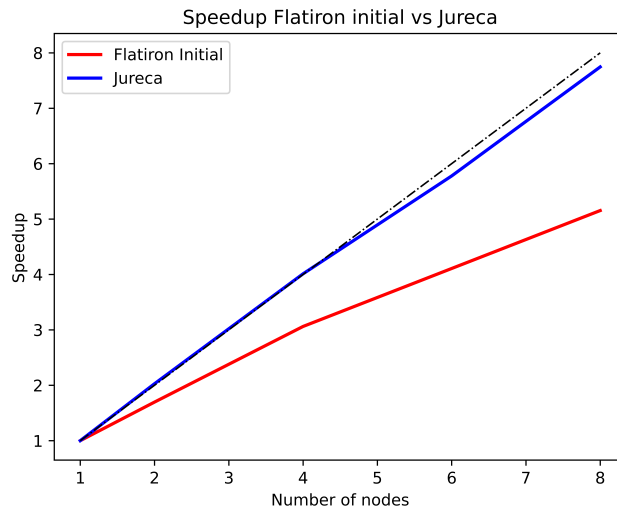


Figure 3.5: Speedup compared to scaling of nodes on both systems

of the distributed framework worked and that it was a cluster issue. In order to run on the Juelich system one used the command `srun` to launch instead of the

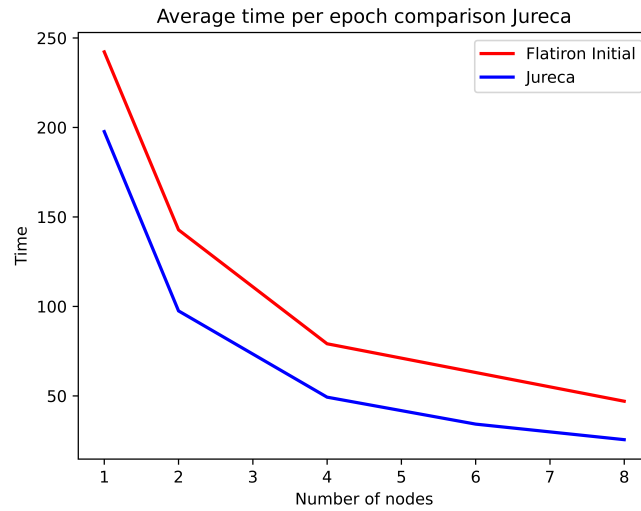


Figure 3.6: Average Time per epoch for a set amount of nodes compared to Jureca

mpirun which was preferred from the Horovod docs. This was due to the system configuration not allowing the usage of **mpirun**.

While testing on the Juelich system I was still being updated on the Flatiron system, as this system was also important for different scaling tests further along on the thesis.

After the preliminary tests, there were done more scaling tests on the Juelich system, such as scaling up to 24 nodes and upwards to 292 nodes. During these tests on the Juelich system, the administrators of Flatiron installed new updates on multiple packages. One of the features available on the GPUs now was the GPUDirect RDMA feature. This feature enables the GPUs to directly access each other's memory. This was a reason to rerun the initial experiment to see if the difference was that feature. These changes gave a much better scaling results on the Flatiron system, but time per epoch the Jureca system still outperformed.

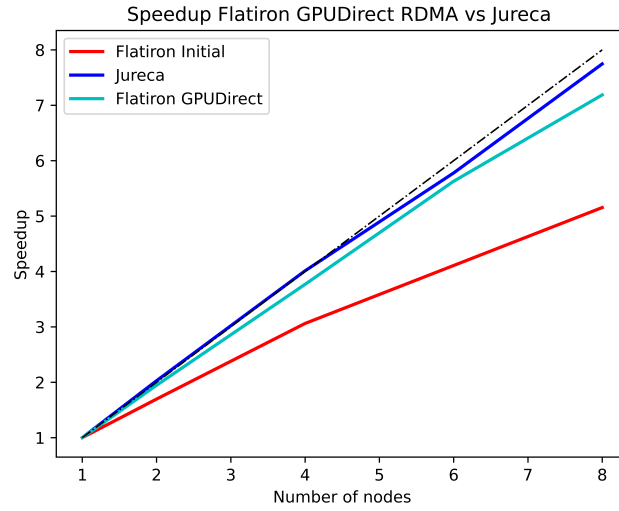


Figure 3.7: Speedup compared to scaling of nodes with GPUDirect RDMA enabled

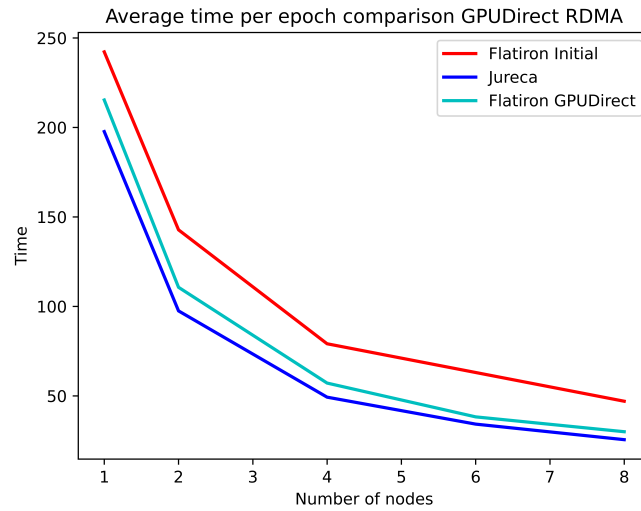


Figure 3.8: Average Time per epoch for a set of nodes with GPUDirect RDMA enabled

3.4.3 Flatiron MPIRUN vs SRUN

Both systems have very similar architecture and hardware, software packages are only differing in the minor versions for the different systems. Still, there was a time per epoch gap of 20 seconds on a single node. Looking at the differences between the two systems I could see that the single major difference was the way each experiment was launched. In the Jureca implementation we used **SRUN** whilst on

the experiments on Flatiron we used **MPIRUN**, changing the launch method gave the results shown in Figure 3.9 and Figure 3.10. Results are finally on par with the results from Juelich system.

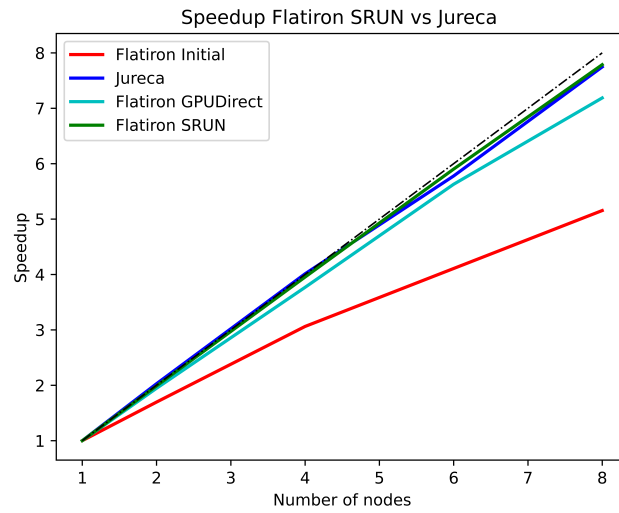


Figure 3.9: Speedup on the Flatiron system with SRUN launch command

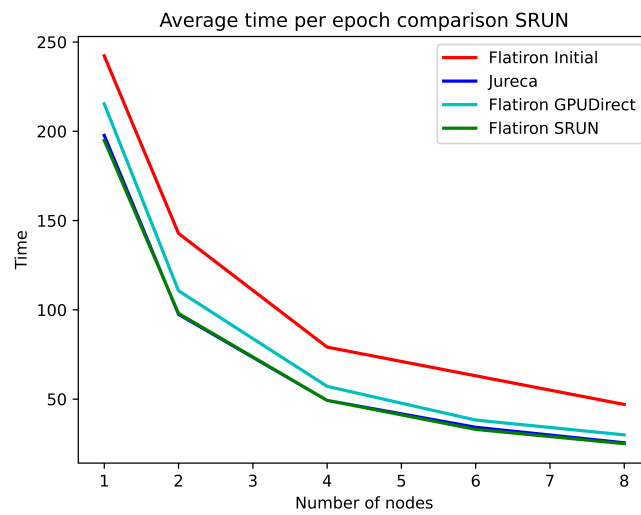


Figure 3.10: Average Time per epoch on the Flatiron system with SRUN launch command

Chapter 4

Results & Discussion

In this chapter, we will present and discuss the results from the different experiments performed on the systems at Flatiron and Juelich. Each system provided data for specific experiments. Section 4.1 takes a look at the results of the scaling benchmarks on the Juelich system. Section 4.2 looks at the scaling performances between the different architectures using the Flatiron cluster. Section 4.3 discusses the results.

4.1 Juelich Results

The results in the figures below are based on the Juelich systems. The results are based on experiments with 1 node up to 24 nodes with an increase of 2 nodes for every experiment on the Jureca cluster. After 24 nodes it was done with an increment of 16 starting from 32 nodes on the Juwels cluster. The change of cluster is due to the resource limitation on Jureca and the increment was changed to 16 due to wait time in the resource queue, giving a more fine result for the scaling with nodes in the sequence 1-24, where every even number of nodes are benchmarked. The change to the Juwels system came with a cost as I did not have enough resources to do small jumps between the node counts. It forced me to do larger jumps of 16 as a investigational experiment to see the potential of the scaling results of the MLPF model with the DELPHES dataset.

The runs performed at Jureca were performed in a set of 2 days with the end result of 90 valid epochs per node count. The reason for only 90, and not 100, is due to the initialization phase of TensorFlow. The initial epoch for every run will have misrepresentative values and is therefore disregarded. The runs of Juwels were performed over a full month due to queue time, but instead of running multiple runs on the Juwels system for the different node counts, the epoch value was increased to 100 in order to mitigate the difference; resulting in 99 valid epochs per node count. These results are based on the benchmarking dataset from the MLPF creators, the given result doesn't show scaling limitation of the model itself but rather the dataset.

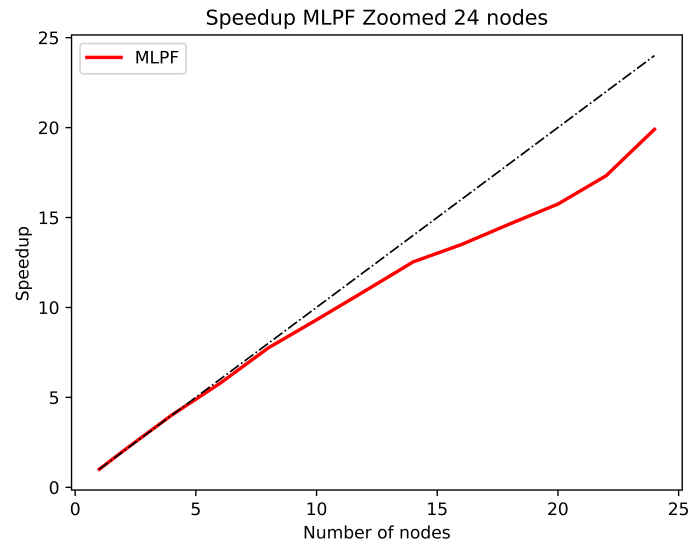


Figure 4.1: Scaling performances on Jureca

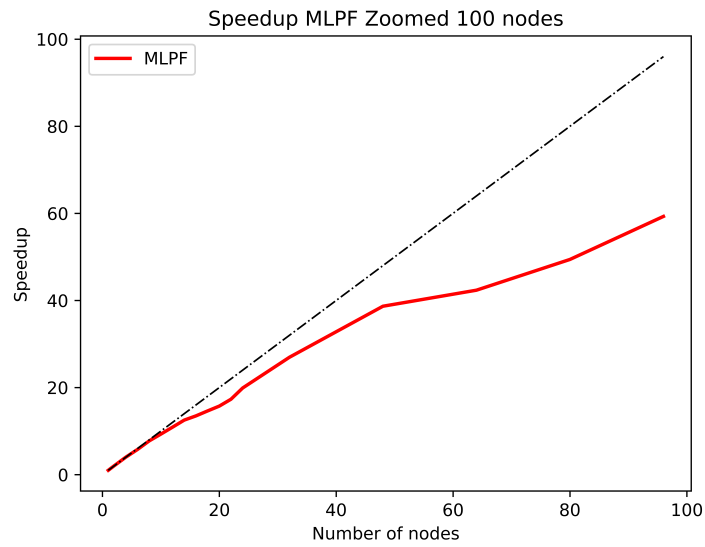


Figure 4.2: Scaling performances on Juwels & Jureca up to a hundred

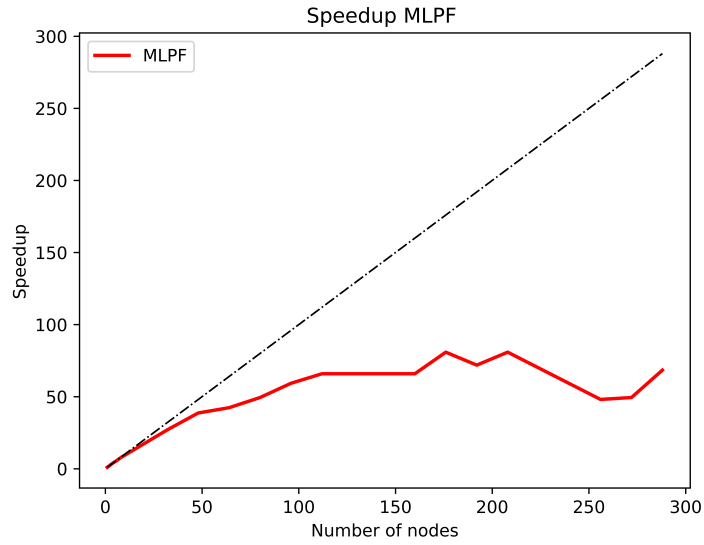


Figure 4.3: Scaling performances on Juwels & Jureca up to 292 nodes

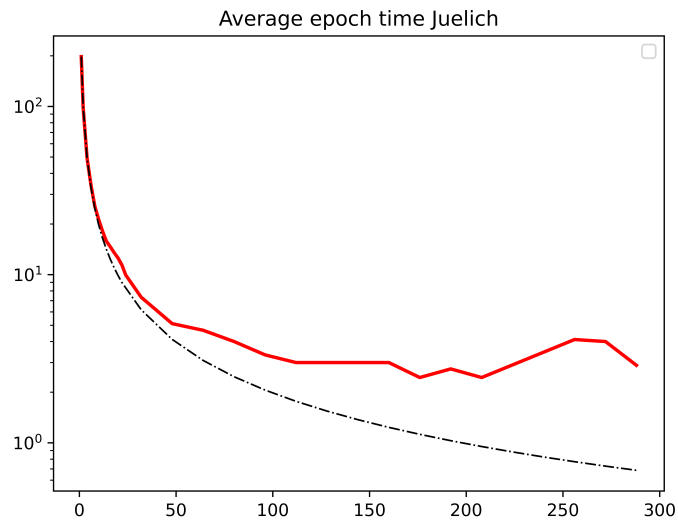


Figure 4.4: Average Epoch Time on Juelich

4.2 Flatiron Results

The experiments conducted here were done on the Flatiron system, with each experiment conducted on different days. The format for the experiment was that each run was conducted 10 times; with each run consisting of 10 epochs. For the experiments the Volta card's memory best utilization was at a batch size of 16. This

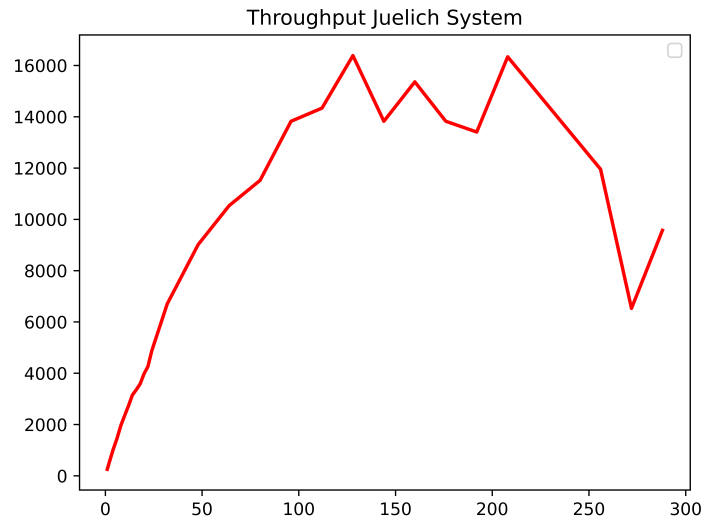


Figure 4.5: Amount of throughput in the form of steps iterated for all of the GPUs

was also then used as a batch size on the Ampere nodes with also an experiment with max batch size for their memory capacity. The experiments were conducted on an increasing node count from 1 to 12, with an increment of 2; looking at scaling performances, time, and throughput.

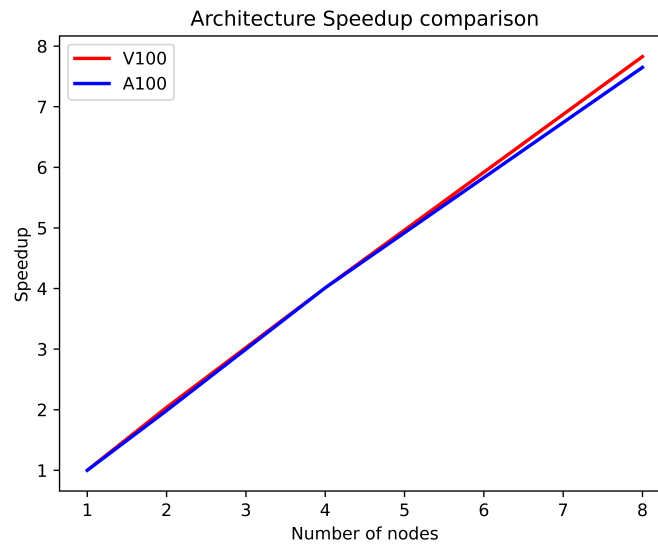


Figure 4.6: Scaling performances Volta versus Ampere

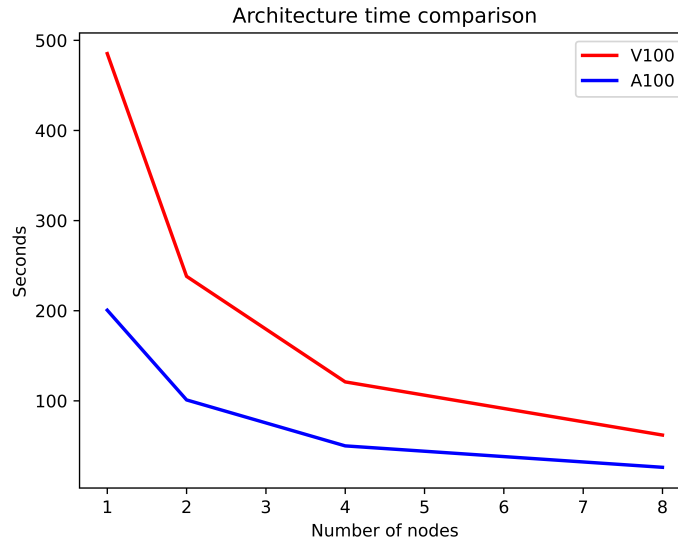


Figure 4.7: Average time per node Volta versus Ampere

4.3 Discussion

Looking at Figure 4.1, we see a close to linear scaling up to 8 nodes, after this it starts to diverge from the linear scaling, and at 24 nodes we see a scaling of 19.76. This result itself is somewhat expected with the size of the dataset. The result is still close enough to linear scaling to further investigate scaling at a higher node count. Figure 4.2 shows the result of the exploratory investigations of large scaling on the MLPF model with the DELPHES dataset. We see that after 50 nodes the divergence from the linear scaling just further increases and the gain from using more nodes become increasingly smaller. These results can also be seen from Figure 4.5; here we achieve a continuous speed-up throughout the scaling up to 130 nodes; it then starts to flatten and the throughput stays more constant, matching what we see in the other figures.

Why is it that the speedup starts to flatten at 100 nodes? One factor is the dataset, more specifically the dataset size; this is why the scaling of the model flattens after 100 nodes. The dataset of 32GB can be seen as a large dataset when it comes to smaller models, but when the model becomes larger and does more precise estimations, bigger datasets are preferred due to their wider range of data coverage giving a better estimate. The size of the dataset is the primary reason why the scaling is flattening. Alongside size, communication overhead needs to be taken into account, the large number of nodes, the larger the communication overhead is.

Figure 4.4 shows the average time per epoch for each node amount, as this is

the basis for the speedup calculation. This can be seen translating quite well to the speedup. Flattening after 130 nodes, we can see the time getting closer to a static value at around 3 seconds. This can be interpreted as the potential communication cost for a large number of nodes. One can argue for that being the case with larger number of nodes. The computation per node will be smaller than with a lower number of nodes. At 130 nodes, the computation might have reached such a small step time that the cost of sharing the weights between the nodes are larger than the actual computation. On the other side, one can also discuss the case of what the communication cost would be for the lower number of nodes, obviously, it will be smaller as the smaller number of nodes only needs to share the weights between fewer nodes so in that sense it is more negligible. It plays a bigger factor when the data processed is becoming smaller, and at the current benchmark, the data can be interpreted to be quite small at 130 nodes. A way to check is to calculate the number of steps, each of which is done at the different levels. The number steps per epoch can be calculated as following:

$$\text{Step per epoch} = \text{Dataset events} / (\text{Number of nodes} * \text{GPUs per node} * \text{Batch size})$$

Table 4.1: The steps per epoch for a set number of nodes

Nodes	Steps per epoch
N	$50000/(N*4*24)$ steps per epoch
1	$50000/(1*4*24) = 50000/96 \approx 520$ steps per epoch
10	$50000/(10*4*24) = 50000/960 \approx 52$ steps per epoch
50	$50000/(50*4*24) = 50000/4800 \approx 11$ steps per epoch
100	$50000/(100*4*24) = 50000/9600 \approx 5$ steps per epoch
130	$50000/(130*4*24) = 50000/12480 \approx 4$ steps per epoch
200	$50000/(200*4*24) = 50000/19200 \approx 2.6$ steps per epoch

Looking at the calculations above, one can see more indications as to why the time flattens after 100 nodes. With the dataset being a constant size and the number of workers increasing, the sizing of the problem per worker will slowly decrease to when the amount per worker equals 1. This supports the theory of the dataset being too small for further increase, with the model only properly benefiting from multinode training up to a certain node amount.

Scaling the model also has a downside regarding the usage of computational resources; running the model on multiple nodes in order to train it is just a part of the job. With AI models, there are many initialization parameters that need to be optimized together in order to produce the best result. One of these parameters is the amount of compute resources, e.g., nodes or GPUs. This parameter is more special than the others as the higher count of compute resources it is going to use for training the hyper optimization needs to use the same, in order to produce the

correct optimal optimization result. In a simpler explanation with hyper parameter optimization for one node, it is parallelized as you can launch multiple instances of the same optimization for different parameters, but the optimization scales n^2 . If one finds out the optimal number of nodes for the given dataset one can focus the hyperparameter optimization for that node amount and save resources. Even if training converges faster due to the scaling of nodes, it is only a partial speedup of the process of making a complete model.

Taking a different look at the results, could the framework be a limitation in the speedup gained per node? Did we manage to configure the Horovod setup correctly to get the best speedup possible? In the published paper of Horovod, there is data showing the performance result of the model on other neural networks of similar size and data size [42]. This data shows the result of scaling multiple different models on the pascal architecture, going from 1 GPU up to 256 GPUS (1/4 node to 64 nodes). None of these models are based on the Graph Neural Network, so there is no current paper showing scaling results for well-known GNN models. But the results show similar signs when scaling as the results showing on the Figure 4.3. This is a highly unlikely scenario as seen with the results, the scaling proceeds well up to a sufficient scaling level that one would expect of a dataset this size. Even the models in the paper show signs of losing speedup to a certain degree after scaling reaches a certain number of nodes and the different models perform differently.

The Horovod paper is based on the data from using Pascal-based GPUs. In this thesis, there were not any available Pascal GPUs to see the performance of the model on that architecture, but there have been papers taking a look at scaling effects of GNN networks on the Volta architecture showing similar results [45]. To further this documentation, we are presenting the result of scaling multinode on both Volta and Ampere to show the scaling effects on both architectures and the difference of throughput. This is shown in Figure 4.6 and Figure 4.7, demonstrating the results that we were expecting with the Ampere architecture being two times faster overall with the same batch size. Increasing the batch size shows an even larger speedup by utilizing the larger memory within the Ampere cards. There has not been any research showing results of scaling GNN networks on the Ampere architecture, so Section 4.2 helps to provide some information on how well a GNN network scales.

Chapter 5

Conclusion & Future Work

Over the years, HPC clusters have become progressively faster, with more clusters breaking the exa scale barrier [46, 47]. It is important to have workloads which can fully utilize the vast amount of computer resources available. One of the fields which is highly applicable to use these resources in is AI, due to their high scalability and the size of their workload. CERN's experiments create and store an absurd amount of data. All of this data serves multiple purposes, one of which is as datasets for AI models. Through this thesis we, have taken a look at the scaling performances of the Particle Flow model MLPF with the benchmark dataset DELPHES and the potential need of HPC resources for AI. The work of the thesis mainly focuses on the scaling performance of the MLPF model towards exascale with the enterprise GPU architecture NVIDIA Ampere, scaling performance between GPU generations and the importance of HPC for AI. This thesis also focused predominantly on the implementation of distributed training for the model and large scale experiments on multiple GPU architectures on multiple clusters.

Implementation of the distributed training workload turned out to be more difficult than initially anticipated, encountering both framework issues and cluster issues. Firstly the implementation with the TensorFlow's own distributed training strategy was preferred, but this implementation wasn't usable between clusters and dependent on specific cluster configurations. Horovod resulted into being the better choice for implementation for the model with is system independent usage. Secondly the implementation with Horovod gave varying results when performing the same experiment on a similar clusters. This was due to software differences available between the clusters, CUDA packages with or without RDMA and launch commands.

Scaling the MLPF model shows great scaling when scaling upwards to 24 nodes with a scaling efficiency close to 20x, the scaling beyond shows adequate scaling with a efficiency of 50x when reaching 100 nodes. Showing a weak scaling when the model scales further up to 292 nodes; i.e 1168 GPUs. This has to do with the dataset being so small, it emphasizes the need of HPC for AI. A single node of

4x A100 GPUs spent hours in order to converge on the DELPHES dataset, whilst using multiple nodes sped up the calculation times ten folds, reaching the same results within minutes. This gives the developer or researcher more time to understand the problem and the result, speeding up the process to further optimize the models to perfection. Still with the hyper optimization and extra iterations the overall gain from scaling models in clusters overshadows everything. Hyper-optimizations is always performed to find the optimal parameters for the given configuration, which might mitigate most of the extra iterations.

The Ampere cards gave great results when scaling the model and will be an important factor for training when new larger datasets appear for the MLPF model. The Volta cards however also showed to be a great substitution for Ampere cards for scaling models, especially MLPF if the resources at clusters at centers are busy. The initial experiments shows the scaling efficiency between both architectures to be exceptionally close, providing similar results. This is great news as the old architectures can still be viable for training and optimization looking into the future, when smaller models need resources. When scaling to a vast amount of nodes on such a small dataset the efficiency difference between the cards start to minimize and both the cards are viable options.

HPC will be one of the most important beneficiaries of AI, as AI will be a large beneficiary of the continuous growth of HPC clusters. The larger the computer power the world has viable the larger the models that can be trained, and the faster models can be optimized and used for the benefit of society, further pushing our world into its new technological phase.

5.1 Future Work

Finally, we would like to present some future directions of research:

- Perform optimizations on the Horovod configuration. The Horovod framework has support for multiple different softwares and configurations, the experiments performed in the thesis was performed with the default settings of the framework. There can be settings and mai packages which can give different and/or better results
- Investigate the scaling performance of the model with newer generations of GPU architectures, all architectures further towards exascale.
- Perform hyper optimizations on the model with the new parameter deciding the amount of nodes.

Bibliography

- [1] *Cuda toolkit documentation v11.8.0*. [Online]. Available: <https://docs.nvidia.com/cuda/>.
- [2] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro and E. Shelhamer, 'Cudnn: Efficient primitives for deep learning,' *CoRR*, vol. abs/1410.0759, 2014. arXiv: 1410.0759. [Online]. Available: <http://arxiv.org/abs/1410.0759>.
- [3] *Nvidia cudnn*. [Online]. Available: <https://docs.nvidia.com/deeplearning/cudnn/index.html>.
- [4] *Nvidia collective communications library (nccl)*, Feb. 2022. [Online]. Available: <https://developer.nvidia.com/nccl>.
- [5] *Nvidia collective communication library (nccl) documentation*. [Online]. Available: <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html>.
- [6] *Nvlink*. [Online]. Available: <https://docs.nvidia.com/networking/display/SGXH100/Introduction>.
- [7] 1st ed. NVIDIA Corporation, 2016. [Online]. Available: <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>.
- [8] 1st ed. NVIDIA Corporation, 2018. [Online]. Available: <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>.
- [9] 1st ed. NVIDIA Corporation, 2020. [Online]. Available: <https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf>.
- [10] *Nvidia turing architecture in-depth*, Aug. 2022. [Online]. Available: <https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/>.

- [11] A. Castelló, M. F. Dolz, E. S. Quintana-Ortí and J. Duato, 'Analysis of model parallelism for distributed neural networks,' in *Proceedings of the 26th European MPI Users' Group Meeting*, ser. EuroMPI '19, Zürich, Switzerland: Association for Computing Machinery, 2019, ISBN: 9781450371759. DOI: 10.1145/3343211.3343218. [Online]. Available: <https://doi.org/10.1145/3343211.3343218>.
- [12] Y. Zhong, C. Xie, S. Zheng and H. Lin, *Compressed communication for distributed training: Adaptive methods and system*, 2021. DOI: 10.48550/ARXIV.2105.07829. [Online]. Available: <https://arxiv.org/abs/2105.07829>.
- [13] A. Krizhevsky, 'One weird trick for parallelizing convolutional neural networks,' *CoRR*, vol. abs/1404.5997, 2014. arXiv: 1404.5997. [Online]. Available: <http://arxiv.org/abs/1404.5997>.
- [14] I. Merelli, E. Mosca and L. Milanese, 'Parallel computing, data parallelism,' in *Encyclopedia of Systems Biology*, W. Dubitzky, O. Wolkenhauer, K.-H. Cho and H. Yokota, Eds. New York, NY: Springer New York, 2013, pp. 1624–1624, ISBN: 978-1-4419-9863-7. DOI: 10.1007/978-1-4419-9863-7_1028. [Online]. Available: https://doi.org/10.1007/978-1-4419-9863-7_1028.
- [15] C. J. Shallue, J. Lee, J. M. Antognini, J. Sohl-Dickstein, R. Frostig and G. E. Dahl, 'Measuring the effects of data parallelism on neural network training,' *CoRR*, vol. abs/1811.03600, 2018. arXiv: 1811.03600. [Online]. Available: <http://arxiv.org/abs/1811.03600>.
- [16] *Message passing interface*. [Online]. Available: <https://hpc.nmsu.edu/discovery/mpi/introduction/>.
- [17] M. M. Strout, B. Kreaseck and P. D. Hovland, 'Data-flow analysis for mpi programs,' in *2006 International Conference on Parallel Processing (ICPP'06)*, 2006, pp. 175–184. DOI: 10.1109/ICPP.2006.32.
- [18] R. L. Graham, T. S. Woodall and J. M. Squyres, 'Open mpi: A flexible high performance mpi,' in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, N. Meyer and J. Waśniewski, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 228–239, ISBN: 978-3-540-34142-0.
- [19] W. Gropp, E. Lusk, N. Doss and A. Skjellum, 'A high-performance, portable implementation of the mpi message passing interface standard,' *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996, ISSN: 0167-8191. DOI: [https://doi.org/10.1016/0167-8191\(96\)00024-5](https://doi.org/10.1016/0167-8191(96)00024-5). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0167819196000245>.
- [20] *Mpi_rreduce*. [Online]. Available: https://www.open-mpi.org/doc/v4.1/man3/MPI_Reduce.3.php.
- [21] *Mpi_allreduce*. [Online]. Available: https://www.open-mpi.org/doc/v3.0/man3/MPI_Allreduce.3.php.

- [22] *Mpi_reduce_scatter*. [Online]. Available: https://www.open-mpi.org/doc/v4.1/man3/MPI_Reduce_scatter.3.php.
- [23] I. Goodfellow, Y. Bengio and A. Courville, *Deep learning*. MIT press, 2016.
- [24] Y. LeCun, Y. Bengio and G. Hinton, ‘Deep learning,’ *nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [25] J. Baek and Y. Choi, ‘Deep neural network for predicting ore production by truck-haulage systems in open-pit mines,’ *Applied Sciences*, vol. 10, p. 1657, Mar. 2020. DOI: 10.3390/app10051657.
- [26] T. Yu and H. Zhu, ‘Hyper-parameter optimization: A review of algorithms and applications,’ *arXiv preprint arXiv:2003.05689*, 2020.
- [27] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner and G. Monfardini, ‘The graph neural network model,’ *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009. DOI: 10.1109/TNN.2008.2005605.
- [28] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam and P. Vandergheynst, ‘Geometric deep learning: Going beyond euclidean data,’ *CoRR*, vol. abs/1611.08097, 2016. arXiv: 1611.08097. [Online]. Available: <http://arxiv.org/abs/1611.08097>.
- [29] K. Han, Y. Wang, J. Guo, Y. Tang and E. Wu, *Vision gnn: An image is worth graph of nodes*, 2022. DOI: 10.48550/ARXIV.2206.00272. [Online]. Available: <https://arxiv.org/abs/2206.00272>.
- [30] L. Huang, D. Ma, S. Li, X. Zhang and H. Wang, ‘Text level graph neural network for text classification,’ *CoRR*, vol. abs/1910.02356, 2019. arXiv: 1910.02356. [Online]. Available: <http://arxiv.org/abs/1910.02356>.
- [31] P. Pradhyumna, G. P. Shreya and Mohana, ‘Graph neural network (gnn) in image and video understanding using deep learning for computer vision applications,’ pp. 1183–1189, 2021. DOI: 10.1109/ICESC51422.2021.9532631.
- [32] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner and G. Monfardini, ‘The graph neural network model,’ *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009. DOI: 10.1109/TNN.2008.2005605.
- [33] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner and G. Monfardini, ‘The graph neural network model,’ *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009. DOI: 10.1109/TNN.2008.2005605.
- [34] *Tensorflow*. [Online]. Available: <https://www.tensorflow.org/>.
- [35] Tensorflow, *Tensorflow/tensorflow: An open source machine learning framework for everyone*. [Online]. Available: <https://github.com/tensorflow/tensorflow>.

- [36] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu and X. Zheng, ‘TensorFlow: A system for Large-Scale machine learning,’ pp. 265–283, Nov. 2016. [Online]. Available: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>.
- [37] K. Team, *Simple. flexible. powerful.* [Online]. Available: <https://keras.io/>.
- [38] Keras-Team, *Keras-team/keras: Deep learning for humans.* [Online]. Available: <https://github.com/keras-team/keras>.
- [39] *Mirroredstrategy.* [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/distribute/MirroredStrategy.
- [40] *Parameterserverstrategy.* [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/ParameterServerStrategy.
- [41] *Multiworkermirroredstrategy.* [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/distribute/MultiWorkerMirroredStrategy.
- [42] A. Sergeev and M. D. Balso, ‘Horovod: Fast and easy distributed deep learning in TensorFlow,’ *arXiv preprint arXiv:1802.05799*, 2018.
- [43] *Horovod docs.* [Online]. Available: https://horovod.readthedocs.io/en/stable/summary_include.html#why-horovod.
- [44] A. Gibiansky, *Bringing hpc techniques to deep learning.* [Online]. Available: <https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/>.
- [45] T. Baruah, K. Shivdikar, S. Dong, Y. Sun, S. A. Mojumder, K. Jung, J. L. Abellán, Y. Ukidave, A. Joshi, J. Kim and D. Kaeli, ‘Gnnmark: A benchmark suite to characterize graph neural network training on gpus,’ pp. 13–23, 2021. DOI: 10.1109/ISPASS51385.2021.00013.
- [46] *Fugaku.* [Online]. Available: <https://www.fujitsu.com/global/about/innovation/fugaku/>.
- [47] *Direction of discovery.* [Online]. Available: <https://www.olcf.ornl.gov/frontier/>.

