Revista Colombiana de Computación
Vol. 25, No. 1. January – June 2024, pp. 48-59
e-ISSN: 2539-2115, https://doi.org/10.29375/25392115.5056
Selected paper previously presented at the Latin America High Performance Computing Conference (CARLA 2023), an event held in Cartagena de Indias, Colombia, September 18-22, 2023.

REVISTA COLOMBIANA DE COMPUTACIÓN

A Study of Pipeline Parallelism in Deep Neural Networks

Gabriel Núñez[1], Hairol Romero-Sandí[1], Elvis Rojas[1,3], and Esteban Meneses[2,3]

[1] Universidad Nacional, Pérez Zeledón, Costa Rica
[2] Costa Rica Institute of Technology, Cartago, Costa Rica
[3] National High Technology Center, San José, Costa Rica
gnunez@una.ac.cr, hromero@una.ac.cr, erojas@una.ac.cr, emeneses@cenat.ac.cr

(Received: 6 February 2024; accepted: 18 June 2024, Published online: 30 June 2024)

Abstract. The current popularity in the application of artificial intelligence to solve complex problems is growing. The appearance of chats based on artificial intelligence or natural language processing has generated the creation of increasingly large and sophisticated neural network models, which are the basis of current developments in artificial intelligence. These neural networks can be composed of billions of parameters and their training is not feasible without the application of approaches based on parallelism. This paper focuses on studying pipeline parallelism, which is one of the most important types of parallelism used to train neural network models in deep learning. In this study we offer a look at the most important concepts related to the topic and we present a detailed analysis of 3 pipeline parallelism libraries: Torchgpipe, FairScale, and DeepSpeed. We analyze important aspects of these libraries such as their implementation and features. In addition, we evaluated them experimentally, carrying out parallel trainings and taking into account aspects such as the number of stages in the training pipeline and the type of balance.

Keywords: Deep learning, parallelism, artificial neural networks, distributed training.

1 Introduction

In the era of Artificial Intelligence, Distributed Neural Networks (DNN) have become a fundamental tool for processing large volumes of data and performing complex tasks (Alshamrani & Ma, 2022). Speech recognition for transcription to text, natural language recognition used by chatbots, computer vision for object detection and classification, content generation in music are some of the areas in which neural networks have demonstrated their effectiveness (Russakovsky et al., 2015; TensorFlow: Overview, 2023). These applications require intensive data processing and training of neural network models that can have trillions of parameters.

Deep Learning (DL) has been revolutionary, as it enables models to be trained and complex problems to be addressed more efficiently. However, as neural networks become larger and more complex, significant computational challenges arise (Chilimbi et al., 2014). A critical aspect in distributed training is the parallelism strategy used to speed up the training process. Existing parallelism techniques include data parallelism and model parallelism. Model parallelism has been shown to be effective by breaking a model into smaller parts and distributing its training across different devices or nodes (Takisawa et al., 2020).

In this article, a comparative analysis of Model Parallelism (MP) in DNN will be carried out. There are two main objectives: First, to describe the most important elements related to MP. Second, evaluate and compare the performance of different training libraries. In addition, the differences in features of the evaluated libraries are described and analyzed. The libraries used are Torchgpipe (TGP), FairScale (FSC) and DeepSpeed (DSP), using the Python framework called PyTorch.

To perform this comparison, a series of experiments will be carried out using the same dataset and common reference models. Different metrics such as training time and training loss will be measured in order to determine the effectiveness and efficiency of each library with the same parameter settings.

By analyzing and comparing these MP techniques in distributed neural networks, we hope to gain a deeper understanding of their strengths and limitations. Furthermore, it is expected to provide to the

research community with a practical guide on which libraries may be more suitable for their specific needs in terms of distributed training of DL models.

2 Background

2.1 Artificial Neural Networks

Neural networks are computational models inspired by the biological concept of the functioning of the brain of living beings. They are an approach to DL that uses algorithms and data structures to process information and perform pattern recognition tasks. Neural networks are represented as a graph composed of nodes and edges. The interconnected nodes or neurons form layers, where each neuron takes inputs, performs calculations, and generates an output that is transmitted to the neurons in the next layer. These connections between neurons are assigned weights, which are adjusted during the network training process to optimize its performance. The ability of neural networks to learn and adapt from data and information provided during training is what makes them effective in pattern recognition and classification tasks in complex data (Farkas et al., 2020; Deep Learning, 2020).

2.2 Deep learning frameworks

The frameworks used for DL are sets of tools and software libraries designed to facilitate the development, training, and deployment of learning models. These frameworks provide a high-level programming interface that allows developers to build and work with neural networks and other DL models more efficiently by hiding low-level implementation details. DL frameworks offer a variety of features and utilities, such as predefined layers, optimization algorithms, training methods, visualization tools, and support for parallel and distributed processing.

In addition, many frameworks also include integrations with numerical computing libraries and hardware accelerators to make the most of available computing power (Rojas et al., 2021).

The DL landscape is constantly evolving, therefore there are several popular frameworks that are designed to make it easy to deploy and train DL models. These frameworks offer optimized operations for handling tensors and the efficient implementation of DL algorithms for CPUs, GPUs, and TPUs. Among some of the frameworks that can be found are TensorFlow (Abadi et al., 2016; TensorFlow: Overview, 2023), MXNet (2023) and PlaidML (2023), which support the popular Keras (2023) high-level API. On the other hand, another framework like Caffe (Jia et al., 2014) developed by Berkeley AI Research and with contributions from the community, it is written in C++, with CUDA used for GPU computing and with support for using Python/Numpy and MATLAB. Likewise, Deeplearning4j (DL4J) (2023) is a suite of tools to run DL on JVM (Java Virtual Machine) and, as indicated by its creators, it is the only framework that allows training models from Java while interacting with the Python ecosystem. Finally, PyTorch (2023), which is the framework that has been selected to run the experiments performed in this paper by applying the parallelism technique called Pipeline Parallelism (PP).

2.3 Distributed training

Distributed training is used in the field of DL to train models simultaneously on multiple computing nodes. Instead of training the model on a single node, distributed training takes advantage of the computing power of multiple devices to speed up the training process and handle larger datasets. To speed up the training time, the data is divided into partitions and distributed among the nodes. Each node has a copy of the model and processes its own data partition, this in case of using the Data Parallel approach. Alternatively, when the models are very large, the Model Parallel distributed training approach is used. The latter will be the one that will be developed in this article (Huang et al., 2019).

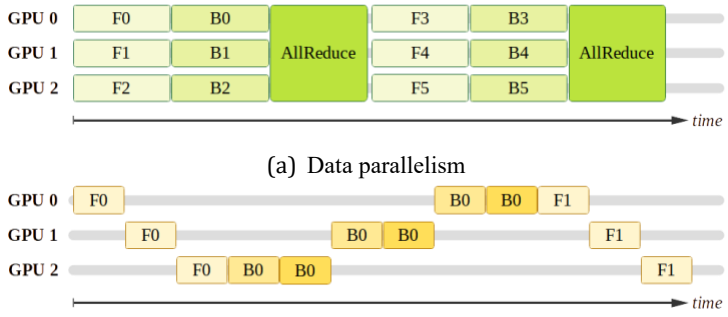


Figure 1. Parallelism techniques.

2.4 Parallelism techniques

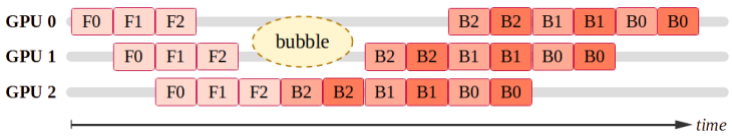
Data Parallelism (DP) is a technique used in DL to speed up the execution of algorithms by dividing data into smaller pieces and processing it simultaneously on multiple computing devices or nodes, same as shown in the figure 1a. According to Mofrad et al. (2020) the DP breaks the input data into small horizontal partitions where each partition can be processed separately. In Chen (2023) it is indicated that in DP, each computing node has a complete copy of the model and works with a portion of the data. Each copy of the model performs independent calculations on its own subset of data, and the results are then combined to obtain the final result (AllReduce function). This makes it possible to process large volumes of data more efficiently and reduce the execution time of the algorithms.

Model Parallelism (MP) is used to accelerate the training and execution of learning models, this by dividing the model into partitions and these are distributed in different computing accelerators, similar to the figure 1b. In Mofrad et al. (2020) it is indicated that in MP the DNN layers are broken into vertical partitions where these partitions are usually processed following a Bulk Synchronous Parallel (BSP) share-memory communication scheme. That is, each accelerator is in charge of processing a specific part of the model, be it a layer or a set of layers. Each part of the model performs independent calculations in its own accelerator, it communicates with the next accelerator that processes the next layer to send it the computed results (Krizhevsky, 2014).

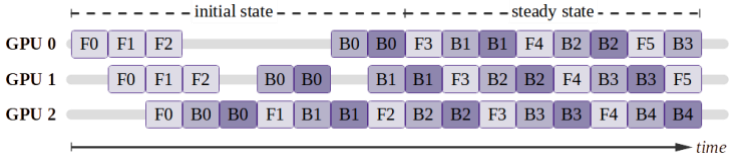
As indicated by Harlap et al. (2018) the MP is useful when working with large-scale models that do not fully fit into the memory of a single accelerator. By splitting the model into smaller parts and processing them in parallel, the MP achieves faster training times than the DP, allowing it to make efficient use of available computing resources.

On the other hand, as pointed out by Dean et al. (2012) the performance benefits of distributing a DNN across multiple machines depend on the connectivity structure and computing needs of the model. Models with a large number of parameters or high computational demands typically benefit from access to more CPU and memory, to the point where communication costs dominate. Furthermore, according to Takisawa et al. (2020) learning performance degrades due to time spent communication between the nodes, and there may be an overload.

Pipeline Parallelism (PP). The transfer of data of heterogeneous sizes in a cluster or between multiple clusters causes an inefficient use of the available network bandwidth, as evidenced by the experiments carried out by Yildirim et al. (2016). Therefore, pipelining, parallelism and concurrency are very effective in removing these bottlenecks, especially when used together and in the right combinations. In Padua (2011) defines pipelining as a parallel processing strategy in which an operation or calculation is partitioned into unconnected stages. As evidenced by Huang et al. (2019) and Kim et al. (2020) when using the MP on multiple GPUs, there will be certainty that various parts of the model will be on different GPUs. If the model training is done sequentially, the training process for each GPU will be active one at a time, similar to what is shown in figure 1b. This will cause a waste of GPU resources and will generate the so-called pipeline bubble phenomenon. Moreover, Rajbhandari et al. (2020) indicate that to hide the pipeline bubble in PP the input mini-batch is divided into several micro-batches and pipes the execution of these micro-batches in several GPUs.



(a) Intra-batch pipeline parallelism



(b) Inter-batch pipeline parallelism

Figure 2. Pipeline parallelism approaches.

In Narayanan et al. (2019) and Takisawa et al. (2020) they indicate 2 types of approaches applied to the PP. The first approach called intra-batch PP, which divides a mini-batch into micro-batches, an example of this is shown in figure 2a. The second approach called interbatch PP shown in figure 2b, indicates that training can be done in two stages: initial stage and steady stage. The initial stage is to enter as many mini-batches as necessary to keep the pipeline full. In the steady stage, the forward and backward propagation processes alternate. This second approach reduces the pipeline bubble. PP is especially useful when working with large-scale models that have complex architecture and require a significant amount of computation, similar to MP. Importantly, unlike MP, the forward and backward propagation phases of different input data overlap in a pipelined manner accelerating DNN training.

2.5 Related Work

Distributed training of DL models has been the subject of extensive research, and various approaches have been proposed to improve the performance and efficiency of this process. Huang et al. presented pioneering research on model training with MP. In their work, GPipe (Huang et al., 2019; Kim et al., 2020) proposed a PP-based approach. In the work carried out by Rojas et al. (2022), the performance of distributed training is analyzed when carrying out experiments using various parallelization mechanisms, such as PyTorch DDP, Horovod, DSP and FSC. The experiments run in this study were developed on the PyTorch framework applying DP-based parallelism techniques. In another study (Chatelain et al., 2022) they experiment training large models using the sharded data parallel strategy implemented in FSC and PyTorch. In this study they experiment with the possibilities of cheating the scaling laws with spurious parameters to save on training costs. Another study (Liang & Alsmadi, 2022) evaluated the performance of DSP in classification tasks through seven neural network architectures.

PipeDream (Narayanan et al., 2019), Xpipe (Guan et al., 2020), HetPipe (Park et al., 2020), Dapple (Fan et al., 2021), Chimera (Li & Hoefler, 2021), AvgPipe (Chen et al., 2023), AutoPipe (Liu et al., 2022), start from the PP approach to improve training performance, through the proposal of architectures and algorithms looking for ways to mitigate the pipeline bubble problem. In a recent article (Zhang et al., 2023) they perform extensive experiments with variable configurations to evaluate the factors that can affect the performance of GPipe. In the study Luo et al. (2022) addresses the issue of training performance and classifies the tools according to the type of pipelining, either asynchronous pipelining or synchronous pipelining. Asynchronous pipelining offers flexibility and speed, but with possible data inconsistency. Synchronous pipelining provides stability and consistency, although there may be a bottleneck. In Yang et al. (2022) a new pipe scheme called WPipe is proposed. In this study, perform a comparison between WPipe against the GPipe, PipeDream and PipeDream2BW libraries in training with natural language models.

Our work is inspired by the aforementioned research and seeks to contribute to the DL field by performing a comparative analysis of the TGP, FSC and DSP libraries on the PyTorch framework. This research aims to provide valuable information in this field by evaluating the performance and efficiency of these libraries using the same approach of MP on DNN using PP.

3 Model parallel libraries

Torchgpipe (TGP). It has the implementation of GPipe in PyTorch. GPipe is a library that implements a PP approach in which model segments are trained sequentially in stages. That is, once the layer sequences in the network are defined in terms of the model parameters (direct calculation function, and the cost estimation function), GPipe divides the network into cells and places each cell in its respective accelerator. During the forward step, GPipe first splits each mini-batch into micro-batches, which are piped through the accelerators. During the backward step, the gradients for each micro-batch are calculated based on the same model parameters used for the forward step. To allow data transfer between neighboring partitions, communication primitives are automatically inserted into partition boundaries. Consequently, the initial segments are trained before the final segments. This allows better utilization of computing resources and speeds up the training process (Huang et al., 2019; Kim et al., 2020).

FairScale (FSC). It is an extension library for large-scale, high-performance training in PyTorch. FSC makes the latest distributed training techniques available in the form of composable modules and easy-to-use APIs that attempt to scale models with limited resources. With FSC, models can scale by layer parallelism and tensor parallelism. Also, by using the layered fragmentation model, memory utilization is reduced and computational calculations become more efficient. On the other hand, it applies techniques that try to optimize the use of memory and training performance, regardless of the scale of the model. The PP in FSC is an implementation as described in GPipe. In FSC, the implementation of fairscale.nn.Pipe was adopted from TGP (FairScale, 2021).

DeepSpeed (DSP). Microsoft is the developer behind this library, which focuses on improving the efficiency and scalability of training DL models in high-performance systems. DSP is governed by three pillars of innovation, which are, (1) training through systems such as ZeRO, 3D-Parallelism, DeepSpeed-MoE, ZeRO-Infinity, (2) inference by reducing latency and cost, this through inference-customized kernels and (3) compression through a library specifically designed to facilitate model compression.

The PP in DSP improves both memory and computational training efficiency by dividing the layers of a model into stages that can be processed in parallel. DSP uses gradient accumulation to extract PP. Training data is divided into micro-batches for parallel processing in the pipeline. The stages communicate the activations and gradients with each other. The local gradients are accumulated and reduced in parallel, followed by updating the weights by the optimizer. DSP provides hybrid data and PP, can be combined with MP. For the implementation of the DSP PP in PyTorch it is required that the model be expressed in a sequence of layers (torch.nn.Sequential) (Aminabadi et al., 2022; DeepSpeed, 2023; Rasley et al., 2020). DSP is compatible with several DL frameworks, such as PyTorch, which is the one we will be using in this article, as it can be used on a wide range of hardware architectures and multi-GPU systems.

4 Evaluating Parallelism Mechanisms

4.1 Library Features

TGP (Kim et al., 2020) performs training on a model by implementing GPipe, the procedure involves simply wrapping the model with the torchgpipe.GPipe function. It is important to note that the model must be structured as a nn.Sequential, since GPipe will automatically segment the model into partitions. Each partition represents a set of consecutive layers that run together on a single device. GPipe optimizes training efficiency by using CUDA, automatically managing the transfer of each partition between different devices.

Determining the optimal balance for a model can be challenging. Especially when designing or adjusting an evolving model, the architecture of the model can change over time. In this situation, the use of the `torchgpipe.balance` function is recommended to achieve automatic balancing. Although this may not result in a perfect optimum balance, it does ensure a good enough level of balance. Among the available balance tools are `balance_by_time()` and `balance_by_size()`, both based on layer profiles. It is important to mention that the checkpoints, in a process known as recalculation, rerun forward propagation during backpropagation. This assumes that both runs are identical, which in turn requires that all layers be referentially transparent in forward propagation to ensure process consistency.

FSC (FairScale, 2021) introduces advanced efficient memory and performance management strategies for model training. One of these features is Efficient Memory Management, which is based on ZeRO algorithms to balance training in parallel with DP and MP, resolving the conflict between memory, computation, and communication. This is accomplished by implementations such as Optimizer State Sharding, Sharded Data Parallel, and Fully Sharded Data Parallel. Additionally, techniques like OffloadModel take advantage of the CPU to store key model elements and gradients, improving training efficiency by moving selected layers to the GPU as needed. Other solutions include Adascale for training large batches without loss of precision, Enhanced Activation Checkpointing to reduce GPU memory usage, and SlowMo Distributed Data Parallel to address slowdowns in distributed training.

Furthermore, FSC also extends the functionality of PyTorch with features like `fairscale.nn.checkpoint.checkpoint_wrapper`, which makes it easier to manipulate arguments in the forward step, transfer intermediate data to the CPU, and handle non-tensor results. These combined strategies and tools seek to address critical memory, communication, and efficiency problems in model training, offering practical solutions and significantly improving the performance and scalability of the training process.

DSP (DeepSpeed, 2023) offers advanced strategies for efficient model training, highlighting the PP as a crucial tool. This technique accommodates various forms of parallelism, including the combination of DP, MP, and PP, achieving scalability in models of up to a trillion parameters through 3D parallelism, and training acceleration up to 7 times on low-bandwidth clusters. The Zero Redundancy Optimizer (ZeRO) is a pillar in DSP that enables the training of massive models. With ZeRO enabled, it is possible to train models of more than 13 billion parameters without MP and up to 200 billion with MP. In addition, techniques such as Activation Partitioning optimize memory in ZeRO by reducing the activation memory footprint proportional to the degree of MP.

ZeRO-Offload, meanwhile, pushes model size limits efficiently by leveraging both GPU and CPU resources. Smart Gradient Accumulation enables larger batches with limited memory by splitting them into sequential micro-batches, and Communication Overlapping overlays communication on backpropagation, improving performance even with modest batch sizes. Additionally, DSP simplifies data loading by automatically handling batch creation from PyTorch data sets. These combined strategies offer complete solutions for efficient and scalable training of DL models.

4.2 Libraries implementation level

The initialization of pipeline training varies depending on the library. In the code listing in figure 3, you can see part of the initialization and execution code for TGP, FSC, DSP. DSP is a highly configurable library, up-to-date and with many optimization options, for this reason it shows more complex code.

```
#GPipe
sample = torch.rand(batch_size,3,32,32)
balance_time = balance_by_time(partitions,model,sample)
model = GPipe(model=model,balance=balance_time,chunks,checkpoint="always")

#FairScale
model = fairscale.nn.Pipe(model=model, balance, chunks)

#DeepSpeed
model = PipelineModule(layers=join_layers(model),
                       loss_fn=torch.nn.CrossEntropyLoss(),
                       num_stages=microbatches,
                       partition_method="uniform",
                       activation_checkpoint_interval=1)

engine,_,_,_ = deepspeed.initialize(args=args,
                                    model=model,
                                    optimizer=optimizer,
                                    model_parameters=[p for p in
                                                      model.parameters()
                                                      if p.requires_grad],
                                    training_data=training_data)
```

Figure 3. Initialization and execution code of the PP libraries.

The first lines of code that are observed belong to TGP (lines 4, 5, and 6). In these lines of code the initialization of TGP is performed. TGP optimizes training using CUDA and automatically moves each partition of the neural network model to different devices. By default, TGP assigns in order the GPUs starting at cuda:0 for each partition. Line 4 is used to set a sample input into the model to then calculate the

balance of the partitions on the GPUs, either time-based or size-based (line 5). In line 6 TGP is executed loading the neural network model, balance, chunks (micro-batch) and checkpoint. The latter is used to checkpoint all micro-batches and reduce memory usage. Other options are "except last" to apply checkpoint except to the last micro-batch and "never" to not apply checkpointing. Other important training instructions are added in the training code.

FSC is a library based on GPipe and TGP. However, it lacks some features like auto balancing. Line 9 shows how the FSC execution is. It is observed that this instruction, as in TGP, also receives the network model, the balance and the chunks. In the case of balancing, this is provided manually as a list of layer numbers per GPU (balance = [2,3,2,3,2,2,2,3]). Like TGP, FSC requires adding other instructions in the training to successfully execute parallelization.

Finally, DSP requires a slightly more complex process to initialize and run the workouts. From lines 12 to 16 the pipeline module is executed. This module receives several important parameters. The first parameter "layers" is assigned the result of the `join_layers` function, which is responsible for taking a neural network model from the `torchvision.models` package to make it sequential. Other parameters are for the calculation of the loss function, the number of stages in which the model will be divided, the balance method (partition method) and the checkpoint interval. Subsequently, DSP initialization is executed to load the model generated by the pipeline module, the optimizer, and the training dataset. Training execution can be simplified by using the `engine.train_batch()` command that encapsulates typical training instructions.

Balance. A relevant element to take into account is the criteria by which the MP libraries carry out load balancing in the GPUs when dividing the neural network model. This balance requires allocating a certain number of layers for each available GPU. For example, for a neural network with 19 layers and 8 GPUs the balance can be: balance = (2,3,2,3,2,2,2,3).

DSP uses two criteria, one called "parameters" in which the number of trainable parameters on each pipeline stage is used and another called "uniform" that balances the number of layers per stage. There is another criterion based on regular expressions, but it will not be taken into account in this study. In the case of TGP, it also allows two balancing criteria. One based on time (balance_by_time) and the other based on size (balance_by_size). By time, it is taken the elapsed time of each layer and by size it is detected the CUDA memory usage of each layer. FSC is a simpler library and does not provide automatic balancing mechanisms, so it requires manual balancing assignment. That is, establish how many layers should be executed per GPU.

4.3 Experimental Evaluation

Experimental Setup. The following experiments aim to evaluate the performance of the three previously described libraries (TGP, FSC, and DSP). We use PP as a starting point, however we also test for pure MP when the library allows it.

It is important to mention that MP is used with neural network models that do not fit in the memory of a GPU, since this type of parallelism is based on dividing the artificial neural network into multiple partitions, depending on the number of GPUs available or other special criteria. In the following experiments, a small neural network with 138,000,000 parameters is used, which is considered small compared to other trillion-parameter neural networks. However, for the type of experiments that we are going to carry out, it is sufficient and allows us to carry out DL training in reasonable times. Our intention is to experiment with the types of libraries, to determine their differences and to lay the foundation for future studies.

The experiments are carried out using the CIFAR100 dataset and the VGG19 neural network. A sequential version of VGG19 was used as this is a requirement of some MP libraries. A sequential version is required so that the neural network can be partitioned, assigned to GPUs, and computed correctly. In most cases the transformation of a typical neural network model to its sequential version is not automatic. The results obtained are from the execution of 25 epochs. In addition, 10 repetitions of each training were performed to obtain statistically correct results. Regarding the use of hardware, the parallel trainings were executed scaling up to 8 GPUs.

In these experiments we evaluated the PP depending on the type of library. In all cases we use a batch size of 128. In the case of TGP and FSC to enable PP we use a micro-batch (chunks) of 8 and 16. This means that the batch size is divided into 8 or 16 depending on the experiment. We also experimented with a microbatch of 1 (doesn't split the batch) by disabling PP and running the trainings in MP. Due to the

differences between the libraries, the DSP configuration is different due to the hybrid parallelism that it implements and the configurations that it supports. With DSP we maintain a micro-batch of 16 and it was not possible to implement pure MP.

Balance is an important element in training. So, we also use it as an evaluation criterion. For both TGP and DSP the aforementioned criteria were implemented. FSC requires manual assignment of the balance. So, we obtained the balance of layers that TGP performed for each criteria and for each experiment, and it was replicated for FSC. Table 1 shows a summary of the most important configurations that were implemented in the trainings.

Table 1. Experimental setup.

Element name	Implemented element
Deep learning framework	PyTorch
Parallelism type	Model, pipeline
Parallelism library	Torchgpipe, FairScale, DeepSpeed
Network model	VGG19
Optimizer	SGD
Batch size	128
Micro-Batch	8, 16
Image dataset	CIFAR100
GPUs	8

Hardware Configuration. The experiments to be developed in this article were executed in the high-performance system called ThetaGPU, which is located in the Argonne Leadership Computing Facility. Each ThetaGPU node integrates 8 NVIDIA DGX A100 GPUs, along with 2 AMD EPYC 7742 processors. Likewise, it is made up of 24 nodes, with 26 TB of DDR4 memory and 8320 GB of GPU memory. Theta performance is 11.7 petaflops and ThetaGPU is 3.9 petaflops.

Table 2. PP with different types of balance and using a micro-batch of 16.

	TGP				FSC				DSP			
	Size		Time		Size		Time		Uniform		Parameters	
# GPU	Loss	Time	Loss	Time	Loss	Time	Loss	Time	Loss	Time	Loss	Time
2	0.86	2424	0.85	2436	0.86	2360	0.83	2386	0.93	2964	0.82	2934
4	0.87	1691	0.86	1925	0.87	1697	0.86	1826	0.88	1842	0.85	1823
6	0.84	1909	0.88	1896	0.86	2027	0.88	2005	0.84	1433	0.95	1445
8	0.91	2214	0.87	2197	0.89	2331	0.88	2235	0.91	1316	0.94	1306

Experimental Results. The results of the experiments can be seen in the tables 2 and 3. Table 2 shows the data from training with the three PP libraries. In all cases, the training time in seconds and the loss are reported. For each library, trainings were performed using two types of balance. In the case of TGP and FSC, balance by time and size was used. DSP implemented "uniform" and "parameter" type balancing. An exact comparison could not be made as DSP does not support the balance types of the other two libraries.

In table 2 you can see similar loss values in all the results. This result was expected due to the type of neural network used. It was not difficult to find optimized hyperparameters to obtain loss values appropriate to the type of training. For this study, loss is an important metric. However, it is not an important indicator to observe differences between libraries. The previously described behavior with the loss is also replicated in the results of table 3.

Regarding the training times, we can see that DSP with 8 GPUs has a superior performance of 1316 and 1306 seconds in the "uniform" and "parameter" balancing modes, respectively. The above with respect to TGP and FSC that reported higher times between 2197 and 2235 seconds. On the other hand, in the case of TGP and FSC there is an erratic behavior in the performance. With 8 GPUs, similar training times were generated to those generated with 2 GPUs. This behavior does not occur with DSP where scaling on GPUs slightly increases performance with both balancing methods. The poor performance of TGP and FSC when scaling is attributed to the wait times generated between GPUs, which are increased by using a micro-batch

of 16. That is, with a large number of microbatch, the pipeline is divided into smaller stages, decreasing performance. We can verify this with the results of the table 3 in which micro-batches of 1 and 8 were used. In the case of DSP this behavior is not reflected. We attribute this to the fact that DSP, in addition to PP, also automatically implements a type of hybrid parallelism (Akintoye et al., 2022; DeepSpeed, 2023; Zeng et al., 2021), so it can take advantage of the GPUs at its disposal and not suffer from the effects of a large number of stages in the pipeline.

Table 3 shows results with micro-batches of 1 (Mbs1) and 8 (Mbs8). When using a micro-batch of 1, the pipeline is disabled and the training runs with pure MP. The trainings were executed for the two types of balance. With DSP, this type of training was not carried out due to the configuration parameters, it was not possible to implement it. One of the first observations that we can make is the increase in performance reflected in all cases when using a micro-batch of 1. In our experiments this behavior is presented by the type of neural network used, but with large neural networks that do not fit in the GPU memory the pipeline is an important mechanism to improve performance. Despite this, there will always be a trade-off between GPU utilization and micro-batch size. Without a pipeline, the performance is good, but similar times are presented in trainings with 2 and 8 GPUs. This is an indicator that our choice of micro-batch must be modified until acceptable values are found for the type of hardware, neural network model, and dataset. Training with micro-batch of 8 also shows an increase in performance over training with micro-batch of 16. This behavior was described earlier in this section.

Finally, if we compare the balance types for each parallelism library, no significant differences are reflected among themselves at this scale. In other words, with a small neural network and few GPUs. In complex execution environments, this type of optimization can make a difference in performance when training neural network models.

Table 3. PP with different types of balance and using a micro-batch of 8 (Mbs8) and micro-batch 1 (Mbs1, pure model parallelism).

	Torchgpipe								FairScale							
	Size				Time				Size				Time			
#	Mbs1		Mbs8		Mbs1		Mbs8		Mbs1		Mbs8		Mbs1		Mbs8	
GPU	Loss	Time	Loss	Time	Loss	Time	Loss	Time	Loss	Time	Loss	Time	Loss	Time	Loss	Time
2	0.85	383	0.71	1356	0.86	378	0.84	1387	0.87	386	0.83	1343	0.88	392	0.87	1372
4	0.81	237	0.83	880	0.82	247	0.81	933	0.82	295	0.85	916	0.82	286	0.84	972
6	0.79	262	0.83	1026	0.81	253	0.87	1024	0.81	307	0.83	1074	0.82	266	0.84	1062
8	0.81	308	0.87	1175	0.84	294	0.89	1176	0.82	427	0.81	1251	0.81	376	0.83	1219

5 Concluding Remarks and Future Work

Acceleration of neural network training through parallelism is an increasingly important element in DL research. The efforts of many researchers are reflected in a large number of studies in which new libraries, algorithms, and parallelism approaches are proposed. They seek to take advantage of the hardware power of high-performance computing systems. This study focused on Pipeline Parallelism, which is a subset of Model Parallelism. The main concepts of this type of parallelism were described. In addition, 3 libraries that implement Pipeline parallelism were described in detail and experiments were carried out to measure the performance based on certain criteria of the type of parallelism.

As future work, it is necessary to carry out training with large neural network models, which allow to more accurately evaluate the performance of pipeline libraries and algorithms. In addition, we believe it is necessary to implement more libraries based on model parallelism. We have noticed differences in the libraries related to the features and configurations that they offer to carry out the trainings. An example is DSP, which offers a very wide number of configurations, not only to implement pipeline parallelism, but also to perform memory and IO optimizations. Based on the above, we also have as future work to delve into the optimizations and techniques that DSP uses to perform parallelism.

Acknowledgments

This research used resources of the Argonne Leadership Computing Facility (ALCF), which is a DOE Office of Science User Facility supported under Contract DE-AC0206CH11357.

ORCID iD

Gabriel Núñez https://orcid.org/0000-0002-6907-533X
Hairol Romero-Sandí https://orcid.org/0000-0002-3199-1244
Elvis Rojas https://orcid.org/0000-0002-4238-0908
Esteban Meneses https://orcid.org/0000-0002-4307-6000

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., . . . Zheng, X. (2016). TensorFlow: A System for Large-Scale Machine Learning. *e Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16). November 2–4* (pp. 264-283). Savannah, GA, USA: USENIX Association. https://doi.org/10.48550/arXiv.1605.08695

Akintoye, S., Han, L., Zhang, X., Chen, H., & Zhang, D. (2022). A Hybrid Parallelization Approach for Distributed and Scalable Deep Learning. *IEEE Access, 10*, 77950-77961. https://doi.org/10.1109/ACCESS.2022.3193690

Alshamrani, R., & Ma, X. (2022). Deep Learning. In C. L. McNeely, & L. A. Schintler (Eds.), *Encyclopedia of Big Data* (pp. 373-377). Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-32010-6_5

Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., . . . He, Y. (2022). DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 1-15). Dallas, TX, USA: IEEE. https://doi.org/10.1109/SC41404.2022.00051

Chatelain, A., Djeghri, A., Hesslow, D., & Launay, J. (2022). Is the Number of Trainable Parameters All That Actually Matters? In M. F. Pradier, A. Schein, S. Hyland, F. J. Ruiz, & J. Z. Forde (Ed.), *Proceedings on "I (Still) Can't Believe It's Not Better!" at NeurIPS 2021 Workshops. 163*, pp. 27-32. PMLR. https://proceedings.mlr.press/v163/chatelain22a.html

Chen, M. (2023). Analysis of Data Parallelism Methods with Deep Neural Network. *EITCE '22: Proceedings of the 2022 6th International Conference on Electronic Information Technology and Computer Engineering, October 21 - 23* (pp. 1857 - 1861). Xiamen, China: Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3573428.3573755

Chen, Z., Xu, C., Qian, W., & Zhou, A. (2023). Elastic Averaging for Efficient Pipelined DNN Training. *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP '23* (pp. 380-391). Montreal, QC, Canada: Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3572848.3577484

Chilimbi, T., Suzue, Y., Apacible, J., & Kalyanaraman, K. (2014). Project Adam: Building an Efficient and Scalable Deep Learning Training System. *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14). October 6–8* (pp. 570-582). Broomfield, CO: USENIX Association. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-chilimbi.pdf

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q. V., . . . Ng, A. Y. (2012). Large Scale Distributed Deep Networks. In F. Pereira, C. J. Burges, L. Bottou, & K. Q. Weinberger (Ed.), *Advances in Neural Information Processing Systems (NIPS 2012). 25*, pp. 1223-1231. Curran Associates. https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf

Deep Learning. (2020). In A. Tatnall (Ed.), *Encyclopedia of Education and Information Technologies* (First ed., p. 558). Springer Cham. https://doi.org/10.1007/978-3-030-10576-1_300164

Deeplearning4j: Deeplearning4j Suite Overview. (2023, July). https://www.deepspeed.ai/

DeepSpeed authors: Deepspeed (overview and features). (2023, July). (Microsoft) https://www.deepspeed.ai/

FairScale authors. (2021). *Fairscale: A general purpose modular pytorch library for high performance and large scale training.* https://github.com/facebookresearch/fairscale

Fan, S., Rong, Y., Meng, C., Cao, Z., Wang, S., Zheng, Z., . . . Lin, W. (2021). DAPPLE: a pipelined data parallel approach for training large models. *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (pp. 431-445). Virtual Event, Republic of Korea: Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3437801.3441593

Farkas, A., Kertész, G., & Lovas, R. (2020). Parallel and Distributed Training of Deep Neural Networks: A brief overview. *2020 IEEE 24th International Conference on Intelligent Engineering Systems (INES)* (pp. 165-170). Reykjavík, Iceland: IEEE. https://doi.org/10.1109/INES49302.2020.9147123

Guan, L., Yin, W., Li, D., & Lu, X. (2020, November 9). XPipe: Efficient Pipeline Model Parallelism for Multi-GPU DNN Training. *arXiv:1911.04610v3 [cs.LG]*. https://doi.org/10.48550/arXiv.1911.04610

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., & Gibbons, P. (2018, June 18). PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *arXiv:1806.03377v1 [cs.DC]*. https://doi.org/10.48550/arXiv.1806.03377

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., . . . Chen, Z. (2019, July 25). GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *arXiv:1811.06965v5 [cs.CV]*, 1-11. https://doi.org/10.48550/arXiv.1811.06965

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., . . . Darrell, T. (2014, June 20). Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv:1408.5093v1 [cs.CV]*, 1-4.

Keras: Keras api references. (2023, July). https://keras.io/api/

Kim, C., Lee, H., Jeong, M., Baek, W., Yoon, B., Kim, I., . . . Kim, S. (2020, April 21). torchgpipe: On-the-fly Pipeline Parallelism for Training Giant Models. *arXiv:2004.09910v1 [cs.DC]*, 1-10. https://doi.org/10.48550/arXiv.2004.09910

Krizhevsky, A. (2014, April 26). One weird trick for parallelizing convolutional neural networks. *arXiv:1404.5997v2 [cs.NE]*, 1-7. https://doi.org/10.48550/arXiv.1404.5997

Li, S., & Hoefler, T. (2021). Chimera: efficiently training large-scale neural networks with bidirectional pipelines. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Article No. 27*, pp. 1-14. St. Louis, Missouri, USA: Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3458817.3476145

Liang, G., & Alsmadi, I. (2022, February 12). Benchmark Assessment for DeepSpeed Optimization Library. *arXiv:2202.12831v1 [cs.LG]*, 1-8. https://doi.org/10.48550/arXiv.2202.12831

Liu, W., Lai, Z., Li, S., Duan, Y., Ge, K., & Li, D. (2022). AutoPipe: A Fast Pipeline Parallelism Approach with Balanced Partitioning and Micro-batch Slicing. *2022 IEEE International Conference on Cluster Computing (CLUSTER)* (pp. 301-312). Heidelberg, Germany: IEEE. https://doi.org/10.1109/CLUSTER51413.2022.00042

Luo, Z., Yi, X., Long, G., Fan, S., Wu, C., Yang, J., & Lin, W. (2022). Efficient Pipeline Planning for Expedited Distributed DNN Training. *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications* (pp. 340-349). IEEE. https://doi.org/INFOCOM48880.2022.9796787

Mofrad, M. H., Melhem, R., Ahmad, Y., & Hammoud, M. (2020). Studying the Effects of Hashing of Sparse Deep Neural Networks on Data and Model Parallelisms. *2020 IEEE High Performance Extreme Computing Conference (HPEC)* (pp. 1-7). Waltham, MA, USA: IEEE. https://doi.org/10.1109/HPEC43674.2020.9286195

MXNet: Mxnet api docs. (2023, July). https://mxnet.apache.org/versions/1.9.1

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Gang, G. R., . . . Zaharia, M. (2019). PipeDream: generalized pipeline parallelism for DNN training. (pp. 1-15). Huntsville, Ontario, Canada: Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3341301.3359646

Padua, D. (2011). Pipelining. In D. Padua (Ed.), *Encyclopedia of Parallel Computing* (pp. 1562–1563). Boston, MA, USA: Springer. https://doi.org/10.1007/978-0-387-09766-4_335

Park, J. H., Yun, G., Yi, C. M., Nguyen, N. T., Lee, S., Choi, J., . . . Choi, Y.-r. (2020). HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (pp. 307-321). USENIX Association. https://www.usenix.org/conference/atc20/presentation/park

PlaidML: Plaidml api docs. (2023, July). https://github.com/plaidml/plaidml

Pytorch: Pytorch documentation. (2023, July). https://pytorch.org/

Rajbhandari, S., Rasley, J., Ruwase, O., & He, Y. (2020, May 13). ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. *arXiv:1910.02054v3 [cs.LG]*, 1-24. https://doi.org/10.48550/arXiv.1910.02054

Rasley, J., Rajbhandari, S., Ruwase, O., & He, Y. (2020). DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. *KDD '20: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. Virtual Event. July 6 - 10.* CA, USA: Association for Computing Machinery. https://doi.org/10.1145/3394486.3406703

Rojas, E., Pérez, D., Calhoun, J. C., Bautista Gomez, L., Jones, T., & Meneses, E. (2021). Understanding Soft Error Sensitivity of Deep Learning Models and Frameworks through Checkpoint Alteration. *2021 IEEE International Conference on Cluster Computing (CLUSTER)* (pp. 492-503). Portland, OR, USA: IEEE. https://doi.org/10.1109/Cluster48925.2021.00045

Rojas, E., Quirós-Corella, F., Jones, T., & Meneses, E. (2022). Large-Scale Distributed Deep Learning: A Study of Mechanisms and Trade-Offs with PyTorch. In I. Gitler, C. Barrios Hernández, & E. Meneses (Ed.), *High Performance Computing. CARLA 2021. Communications in Computer and Information Science. 8th Latin American Conference, CARLA 2021, October 6–8, 2021, Revised Selected Papers. 1540*, pp. 177-192. Guadalajara, Mexico: Springer, Cham. https://doi.org/10.1007/978-3-031-04209-6_13

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., . . . Fei-Fei, L. (2015, January 30). ImageNet Large Scale Visual Recognition Challenge. *arXiv:1409.0575v3 [cs.CV]*. https://doi.org/10.48550/arXiv.1409.0575

Takisawa, N., Yazaki, S., & Ishihata, H. (2020). Distributed Deep Learning of ResNet50 and VGG16 with Pipeline Parallelism. *2020 Eighth International Symposium on Computing and Networking Workshops (CANDARW)* (pp. 130-136). Naha, Japan: IEEE. https://doi.org/10.1109/CANDARW51189.2020.00036

TensorFlow: Overview. (2023, July). https://www.tensorflow.org/

Yang, P., Zhang, X., Zhang, W., Yang, M., & Wei, H. (2022). Group-based Interleaved Pipeline Parallelism for Large-scale DNN Training. *International Conference on Learning Representations.* https://openreview.net/forum?id=cw-EmNq5zfD

Yildirim, E., Arslan, E., Kim, J., & Kosar, T. (2016). Application-Level Optimization of Big Data Transfers through Pipelining, Parallelism and Concurrency. *IEEE Transactions on Cloud Computing, 4*(1), 63 - 75. https://doi.org/10.1109/TCC.2015.2415804

Zeng, Z., Liu, C., Tang, Z., Chang, W., & Li, K. (2021). Training Acceleration for Deep Neural Networks: A Hybrid Parallelization Strategy. *2021 58th ACM/IEEE Design Automation Conference (DAC)* (pp. 1165-1170). Francisco, CA, USA: IEEE. https://doi.org/10.1109/DAC18074.2021.9586300

Zhang, P., Lee, B., & Qiao, Y. (2023, October). Experimental evaluation of the performance of Gpipe parallelism. *Future Generation Computer Systems, 147*, 107-118. https://doi.org/10.1016/j.future.2023.04.033