IT & DATA SCIENCE

Parallelism Strategies for Distributed Training

Table of contents

Introduction

When it comes to training big models or handling large datasets, relying on a single node might not be sufficient and can lead to slow training processes. This is where distributed training comes to the rescue. There are several incentives for teams to transition from single-node to distributed training. Some common reasons include:

Faster Experimentation

In research and development, time is of the essence. Teams often need to accelerate the training process to obtain experimental results quickly. Employing multi-node training techniques, such as data parallelism, helps distribute the workload and leverage the collective processing power of multiple nodes, leading to faster training times.

Large Batch Sizes

When the batch size required by your model is too large to fit on a single machine, data parallelism becomes crucial. It involves duplicating the model across multiple GPUs, with each GPU processing a subset of the data simultaneously.

Large Models

In scenarios where the model itself is too large to fit on a single machine's memory, model parallelism is utilized. This approach involves splitting the model across multiple GPUs in various ways, with each GPU responsible for computing a portion of the model's operations. By dividing the model's parameters and computations, model parallelism enables training that was not possible before (e.g. training GPT-4) on machines with limited memory capacity.

Depending on your use case and technical setting, you will need to choose between different strategies, which will start your distributed training journey. In this blogpost, we will discuss some common and SotA strategies and evaluate in which scenarios you might want to consider them.

Parallelism Strategies

There are different parallelism strategies, each relevant for different use cases. Each approach has its own advantages and is suitable for different scenarios. You can either distribute the training data across multiple nodes or GPUs or divide the model itself in various ways across multiple nodes or GPUs. The first approach is particularly useful when the batch size used by your model is too large to fit on a single machine or when you aim to speed up the training process. The second strategy becomes handy if you want to train a big model on machines with limited memory capacity. Furthermore, both strategies can be combined to distribute data across multiple instances of the model, with each model instance running on multiple nodes or GPUs.

Side note: Parallelism techniques are still an active research topic in the field, continuously growing with each new technique and library for implementation purposes. In this blog post, we will cover some of the most common libraries and techniques in the current implementation space. Please feel free to reach out to us if you want to see different approaches and libraries in the future.

Data Parallelism

In data parallelism, you make copies of the model and distribute them to different processes or machines. It involves duplicating the model across multiple GPUs, where each GPU processes a subset of the data simultaneously. Once done, the results of the models are combined and training continues as normal. This approach is particularly useful when the batch size used by your model is too large to fit on a single machine, or when you aim to speed up the training process.

Implementations:

- Pytorch Data Parallel (DP): This implementation in PyTorch allows you to distribute the data across multiple GPUs on a single machine. It replicas the model in every forward pass and simplifies the process of utilizing multiple GPUs for training.
- Pytorch Distributed Data Parallel (DDP): DDP enables training models across multiple processes or machines. It handles the communication and synchronization between different replicas of the model, making it suitable for distributed training scenarios. It uses 'multi-process parallelism, and hence there is no GIL contention across model replicas'.
- Tensorflow MirroredStrategy: MirroredStrategy is a TensorFlow API that supports data parallelism on a single machine with multiple GPUs. It replicates the model on each GPU, performs parallel computations, and keeps the model replicas synchronized.
- Tensorflow MultiWorkerMirroredStrategy: This TensorFlow strategy extends MirroredStrategy to distribute training across multiple machines. It allows for synchronous training across multiple workers, where each worker has access to one or more GPUs.
- Tensorflow TPUStrategy: TPUStrategy is designed specifically for training models on Google's Tensor Processing Units (TPUs). It replicates the model on multiple TPUs and enables efficient parallel computations for accelerated training.

Side note: While DataParallel (DP) and DistributedDataParallel (DDP) are available in Pytorch, it is recommended to use DDP for its superior performance, as stated in the official PyTorch documentation. For a detailed overview of both settings, you can refer to the Pytorch documentation.

When to consider Data Parallelism

- Your model is fitting in a single GPU but you want to experiment faster.
- Your model is fitting in a single GPU but you want to experiment with bigger batch sizes.

Pipeline Parallelism

Scaling up the capacity of deep neural networks has proven to be an effective method for improving the quality of models in different machine learning tasks. However, in many cases, when we want to go beyond the memory limitations of a single accelerator, it becomes necessary to develop specialized algorithms or infrastructure.Here comes pipeline parallelism. Pipeline parallelism is a method where each layer (or multiple layers) are placed on each GPU (vertically or layer-level). If it is applied naively, the training process will suffer from severely low GPU utilization as it is shown in Figure 1(b). The figure shows a model consisting of 4 layers spread across 4 different GPUs (represented vertically). The horizontal axis represents the training process over time, and it demonstrates that only one GPU is used at a time. For more information about pipeline parallelism, refer to this paper.

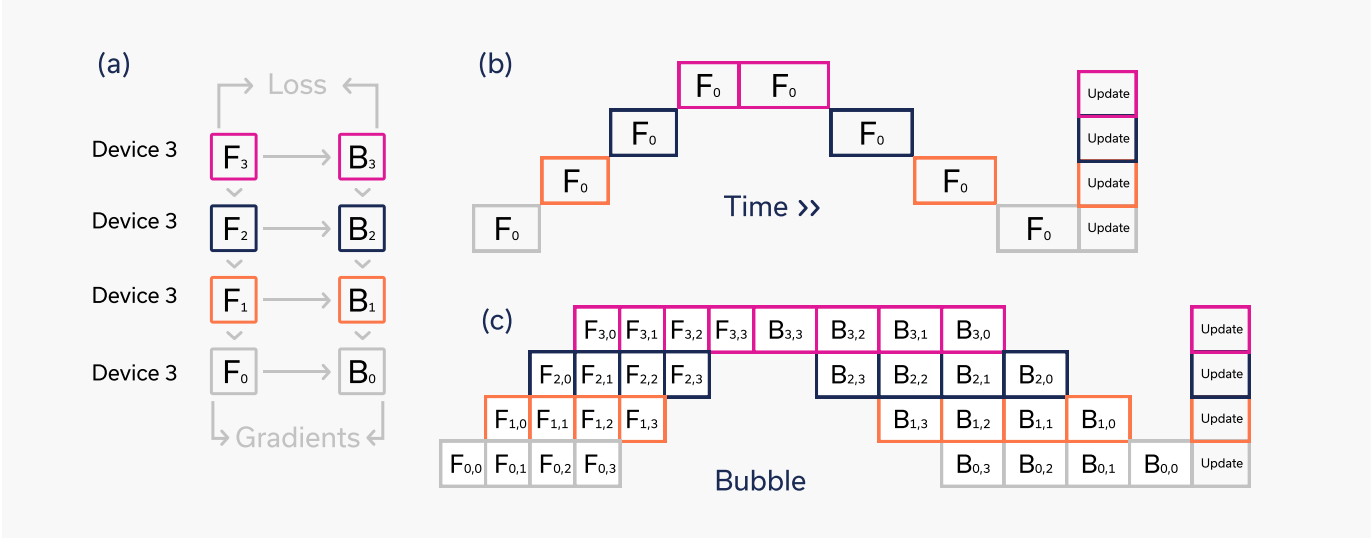

Figure 1: (a) An example neural network with sequential layers is partitioned across four accelerators. Fk is the composite forward computation function of the k-th cell. Bk is the back-propagation function, which depends on both Bk+1 from the upper layer and Fk. (b) The naive model parallelism strategy leads to severe under-utilization due to the sequential dependency of the network. (c) Pipeline parallelism divides the input mini-batch into smaller micro-batches, enabling different accelerators to work on different micro-batches simultaneously. Gradients are applied synchronously at the end.

Implementations:

- Pipe API's in Pytorch: Wraps nn.Sequential module. Uses synchronous pipeline parallelism.
- Fairscale: A Pytorch extension library by Meta for high performance and large scale training with SoTA techniques.
- Deepspeed: A deep learning optimization library by Microsoft that 'makes distributed training and inference easy, efficient, and effective'.
- Megatron-LM: only internal implementation is available
- Mesh Tensorflow: implemented as a layer over TensorFlow.

Side note: It is required to use Pytorch to leverage Deepspeed and Fairscale.

When to consider Pipeline Parallelism

You have a sequential model with many layers (e.g. neural networks, transformers), which does not fit into the memory of a single machine.

Parameter Server Paradigm

Scaling up the capacity of deep neural networks has proven to be an effective method for improving the quality of models in different machine learning tasks. However, in many cases, when we want to go beyond the memory limitations of a single accelerator, it becomes necessary to develop specialized algorithms or infrastructure.Here comes pipeline parallelism. Pipeline parallelism is a method where each layer (or multiple layers) are placed on each GPU (vertically or layer-level). If it is applied naively, the training process will suffer from severely low GPU utilization as it is shown in Figure 1(b). The figure shows a model consisting of 4 layers spread across 4 different GPUs (represented vertically). The horizontal axis represents the training process over time, and it demonstrates that only one GPU is used at a time. For more information about pipeline parallelism, refer to this paper.

Implementations:

- Pytorch RPC-Based Distributed Training (RPC): RPC facilitates communication between the workers and the parameter server, enabling efficient synchronization of the model parameters during training.
- Tensorflow ParameterServerStrategy: In TensorFlow 2, pParameterServerStrategy distributes the training steps to a cluster that scales up to thousands of workers (accompanied by parameter servers). This configuration is known as asynchronous training.

When to consider Parameter Server

- You have a federated learning use-case
- You want to train your model with multiple workers asynchronously using spot instances. In this case, if some workers experience failures, it does not hinder the cluster from carrying out its tasks. This enables the cluster to train using instances that may occasionally become unavailable, such as preemptible or spot instances.

Tensor Parallelism

Tensor parallelism is a technique that involves dividing the model horizontally. It assigns each chunk of the tensor to a designated GPU. During processing, each GPU independently works on its assigned chunk, allowing for parallel computation across multiple GPUs. This approach is often referred to as horizontal parallelism, as the splitting of the tensor occurs at a horizontal level. The results from each GPU are then synchronized at the end of the computation step, combining the individual outputs to form the final result. Tensor parallelism enables efficient utilization of multiple GPUs and can significantly accelerate the processing of large tensors in deep learning and scientific computing tasks.

Implementations:

- Pytorch Tensor Parallel: Tensor Parallelism (TP) is built on top of DistributedTensor (DTensor) and provides several Parallelism styles: Rowwise, Colwise and Pairwise Parallelism.
- DeepSpeed Tensor Parallelism for Inference of HuggingFace Models
- Megatron-LM: only internal implementation is available

Special considerations from HuggingFace documentation: Tensor Parallelism (TP) requires a very fast network, and therefore it's not advisable to do TP across more than one node. Practically, if a node has 4 GPUs, the highest TP degree is therefore 4. If you need a TP degree of 8, you need to use nodes that have at least 8 GPUs.

Combination of Parallelism Techniques

Sometimes, a data science task may require a combination of different training paradigms to achieve optimal performance. For instance, you might want to leverage two or more methods that we covered earlier to take advantage of their respective strengths. There are many possible combinations of these techniques. However, we will cover only 2 in this section, which are state-of-the-art. If you want to train a gigantic model with billions of parameters, you should consider one of these techniques:

The Zero Redundancy Optimizer (ZeRO)

ZeRO (Zero Redundancy Optimizer) is a memory optimization technology for large-scale distributed deep learning that greatly reduces the resources needed for model and data parallelism while enabling the training of models with billions to trillions of parameters. It eliminates memory redundancies by partitioning the model states, including parameters, gradients, and optimizer states, across data-parallel processes instead of replicating them. ZeRO consists of three main optimization stages: Optimizer State Partitioning (Pos or Stage 1), Add Gradient Partitioning (Pos+g or Stage 2), and Add Parameter Partitioning (Pos+g+p or Stage 3). Each stage progressively reduces memory requirements while maintaining similar communication volumes as data parallelism. For more information about ZeRO, please refer to this paper.

Or watch a video here: https://www.microsoft.com/en-us/research/blog/zero-deepspeed-new-system-optimizations-enable-training-models-with-over-100-billion-parameters/

Implementations:

- Pytorch: Stage 1 implemented as ZeRO-1. Further stages are beeing implemented.
- Fairscale: All 3 stages are implemented.
- DeepSpeed: All 3 stages are implemented. Using ZeRO in a DeepSpeed model is quick and easy because all you need is to change a few configurations in the DeepSpeed configuration JSON. No code changes are needed.

Fully Sharded Data Parallel

Fully Sharded Data Parallel (FSDP) by the FairScale team at Meta is essentially a mix of tensor parallelism and data parallelism that aims to accelerate the training of large models by sharding the model's parameters, gradients, and optimizer states across data-parallel workers. Unlike traditional data parallelism, where each GPU holds a copy of the entire model, FSDP distributes these components among the workers. This distribution allows for more efficient use of computing resources and enables training with larger batch sizes and models. Additionally, FSDP provides the option to offload the sharded model parameters to CPUs when they are not actively involved in computations. By utilizing FSDP, researchers and developers can scale and optimize the training of their models with simple APIs, enabling more efficient training of extremely large models. For more information and implementation on Pytorch, please refer to this blog.

Implementations:

- Pytorch: In the next releases, Pytorch is planning to make it easy to switch between DDP, ZeRO1, ZeRO2 and FSDP flavors of data parallelism in the new API. To further improve FSDP performance, memory fragmentation reduction and communication efficiency improvements are also planned.
- Fairscale: The version of FSDP on their Github repository is for historical references as well as for experimenting with new ideas in research of scaling techniques.

Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning

Alpa is a framework that automates the complex process of parallelizing deep learning models for distributed training. It focuses on two types of parallelism: inter-operator parallelism (e.g. device placement, pipeline parallelism and their variants) and intra-operator parallelism (e.g. data parallelism, Megatron-LM's tensor model parallelism). Inter-operator parallelism assigns different operators in the model to different devices, reducing communication bandwidth requirements but suffering from device underutilization. Intra-operator parallelism partitions individual operators and executes them on multiple devices, requiring heavier communication but avoiding data dependency issues. Alpa uses a compiler-based approach to automatically analyze the computational graph and device cluster, finding optimal parallelization strategies for both inter- and intra-operator parallelism. It generates a static plan for execution, allowing the distributed model to be efficiently trained on a user-provided device cluster.

Traditional methods of parallelism, such as device placement and pipeline parallelism, require manual design and optimization for specific models and compute clusters. Alpa simplifies this process by automatically determining the best parallelization plan for a given model, making it easier for ML researchers to scale up their models without extensive expertise in system optimization. It achieves this by leveraging heterogeneous mapping and conducting passes to slice the computational graph, partition tensors, and formulate an Integer-Linear Programming problem to optimize intra-operator parallelism. The inter-operator pass minimizes execution latency using a Dynamic Programming algorithm.

Finally, the runtime orchestration generates execution instructions for each device submesh, allowing for efficient distributed computation. For more information, please refer to the documentation.

Implementations:

- Alpa: Built on top of a tensor computation framework Jax. Alpa can automatically parallelize jax functions and run them on a distributed cluster. Alpa analyzes the computational graph and generates a distributed execution plan tailored for the computational graph and target cluster. The generated execution plan can combine state-of-the-art distributed training techniques including data parallelism, operator parallelism, and pipeline parallelism.The framework's code is also available as open-source. For more information about the strategy, please refer to this blogpost and this presentation too.

Side note: The combined strategies are SoTA approaches, which need further investigation for various use cases. In the time of writing this blog, there has not been much comparison material available, possibly due to high costs of training such models and the approaches being new.

Conclusion

In conclusion, distributed training is a powerful solution for handling large datasets and training big models. It offers several advantages, such as faster experimentation, the ability to work with large batch sizes, and the opportunity to train models that are too large to fit in a single machine's memory. Depending on your specific use case, you can choose from different parallelism strategies.

It's important to note that the field of parallelism techniques is still evolving, with new techniques and libraries being developed. While we have covered some of the most common and state-of-the-art strategies in this blog post, there may be other approaches and libraries worth exploring. The choice of parallelism technique depends on your specific use case, technical setting, and available resources.

References & Further Reads

1. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism
2. Pytorch documentation about Pipeline Parallelism
3. Implementing a Parameter Server Using Distributed RPC Framework
4. Scaling Distributed Machine Learning with the Parameter Server
5. Tensor Parallelism in Pytorch
6. Fully Sharded Data Parallel: faster AI training with fewer GPUs
7. Model Parallelism by HuggingFace
8. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM
9. Fit More and Train Faster With ZeRO via DeepSpeed and FairScale
10. ZeRO & DeepSpeed: New system optimizations enable training models with over 100 billion parameters

About Run:ai

Run:ai is an AI management platform for MLOps, Data Science, and DevOps teams. In addition to helping these teams access and utilize their GPU resources more effectively, it also has a powerful set of features that can abstract infrastructure complexities and simplify the process of training and deploying models. With or without a GPU shortage, Run:ai enables data scientists to focus on innovation without having to worry about resource limitations.