Distributed Machine Learning

Yicheng Xiao
Computer Science and Technology
Southern University of Science and Technology
Shenzhen, China
xiaoyc2022@mail.sustech.edu.cn

***Abstract*—Nowadays, Machine Learning using Deep Neural Network is a hot topic to discuss. Transformer-based model and the development of GPU have accelerated research in relevant fields. However, as the size of the model grows larger and larger, it is harder and harder to load a single model in a single standalone GPU. Distributed Machine Learning is studied to evaluate the effect of training models across several GPUs. In this passage, we will start by introducing some of the cutting-edge distributed framework, then we will discuss what are the current research points in relevant fields and evaluate why these problems are worthy of research.**

***Index Terms*—Distributed Machine Learning, 3D Paralleism, Communication, Compression**

I. Background

Distributed System, to put in a straight way, is a system where multiple computers work collaboratively to accomplish tasks. These computers are interconnected via a network, sharing computational resources and data to enable efficient data processing and task distribution. The advantages and meanwhile the difficulties of distributed systems lie in their scalability and efficiency, allowing them to handle large-scale computational tasks and adapt to various complex scenarios.

Machine Learning is a significant branch of artificial intelligence, aiming to train models using data to equip computers with pattern recognition and predictive capabilities. In recent years, the rapid development of deep learning has driven widespread applications of machine learning technologies, such as natural language processing, image recognition, and recommendation systems. As the scale and complexity of models continue to grow, the demand for computational resources to train and deploy deep learning models has increased sharply.

Distributed machine learning has emerged as a solution, with the core idea of **dividing large-scale machine learning tasks across multiple computational nodes for parallel execution**. Its research focus lies in **evaluating how to improve training efficiency by leveraging multiple GPUs collaboratively**. In recent years, this field has become a research hotspot for several reasons:

1) First, distributed learning methods exhibit greater adaptability when processing massive datasets.
2) Second, the parameter size of deep learning models (e.g., GPT, Transformer) has grown rapidly, making it impossible for a single GPU to meet their computational requirements.
3) Third, distributed training can significantly reduce training time and improve efficiency.

The research on distributed machine learning not only facilitates the scalability of machine learning but also provides new possibilities for solving complex problems.

In this article, we will first introduce some common distributed parallel strategies, then explore the current research focuses in this field and analyze why these issues are worth in-depth investigation.

II. Concerned Problems

From the background introduced above, we can now perceive several essential aspects of distributed machine learning that require thorough research.

Currently, the deep learning neural network is heavily relied on backward propagation of gradients to find solution to the object function $J(\theta)$:

$$\begin{aligned} J(\theta) &= \frac{1}{2}(t_\theta(x) - y)^2 \\ \theta_{t+1} &\leftarrow \theta_t - \gamma \nabla_\theta J(\theta_t) \end{aligned} \tag{1}$$

A. How to manage large data?

When the data is growing larger and larger, relying on one machine to deal with all the data becomes unrealistic. We therefore require the data to be processed on several machines/ nodes.

Stochastic gradient descent provides theoretical support for distributing data to let machine handle them separately. We can divide the data samples to $\xi_i, i = 1, ..., N$

$$\begin{aligned} J_i(\theta) &= \frac{1}{2}(t_\theta(x_i) - y_i)^2 \\ \theta_{t+1} \leftarrow \theta_t - \gamma \nabla_\theta J(\theta_t) &= \theta_t - \frac{\gamma}{N} \sum_{i=1}^{N} \nabla_\theta J_i(\theta) \end{aligned} \tag{2}$$

Now, it is plausible for us to separately handle the data as long as we make sure all the parameters are updated with the gathered gradients.

B. How to manage the parameters?

The growth of GPU memory is less rapid than the growth of its computational capacity. Therefore, as the parameter size is growing, we need to find a proper strategy to tackle with the division of the model. The object function is the same, but when the model is divided, additional communication overhead might be introduced.

C. How to balance the overhead?

We can know that both communication and computation overhead will occur once we introduce distributed methods. We require `all-reduce` communication on gradients when dividing the data and require `all-gather` communication on parameters when dividing the model itself. We can use **Amdahl's law** to evaluate the scaling efficiency:

$$S = \frac{1}{(1-p) + \frac{p}{N}} \tag{3}$$

where S is the theoretical speedup after parallelizing the entire task, p is the proportion of the task that can be parallelized relative to the entire task and N is the number of processors.

Amdahl's Law states that the acceleration gained by increasing processing power is limited by the non-parallelizable portion of a task, with the maximum acceleration being $\frac{1}{1-p}$. This emphasizes the importance of maximizing the parallelizable part of a model for better efficiency. For example, attention models have replaced LSTMs in natural language processing due to their greater parallelizability.

Also, **Gustafson's law** gives the speedup in the execution time of a task that theoretically gains from parallel computing.

$$S = (1-p) + p \times N \tag{4}$$

where S, p, N represents the same thing as the Amdahl's law.

Compared to the pessimistic speedup limit of Amdahl's Law, Gustafson's Law allows the scale of the computational problem to increase with the increase in processing power, thus avoiding the limitation of speedup. This aligns better with many machine learning problems. For example, given a certain computational capacity, the scale of the problem can be adjusted to match the processing power through model structure, data size, hyperparameters, and other factors.

D. More Problems

In order to better optimize around the above three problems, researchers have found several fields to dive into, for example:

- Compression, which intends to reduce the communication overhead
- Error feedback, which aims to ensures the convergence after compression
- Federated learning, which aims to utilize resources from different data center to train the same model.

Therefore, more problems are introduced and yet to be solved.

III. Common Distributed Training Method

A. Data Parallelism

As we have mentioned in Section II.A, data parallelism is usually utilized to handle large dataset.

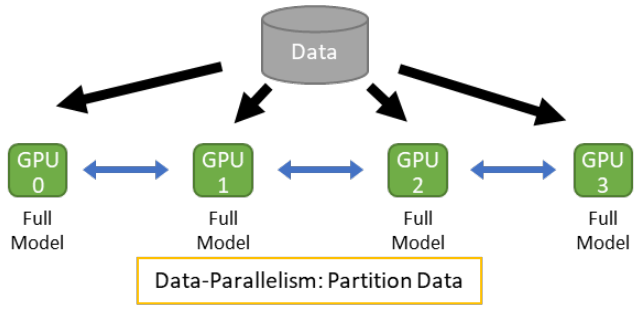


Fig. 1: Data Parallelism Overview

There are two trend of applying data parallelism: Parameter Sever based methods and All-reduce based methods.

a) *Parameter Server:*

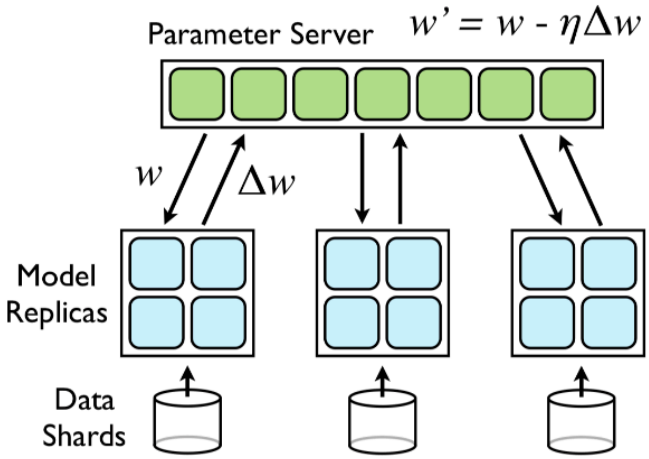


Fig. 2: DownpourSGD, a parameter server based methods to apply data parallelism [1]

Parameter Server based methods heavily rely on a server or multiple servers to control and manage global gradients. Each rank will have a full replica of the model, updating itself only when the parameter server has gathered all the gradients calculated on each machine and sent back the reduced gradient to each rank.

b) *All-reduce Based (Distributed Data Parallelism):*

As the advancement of technology, communication bandwidth has increased, therefore we no longer require a server to manage all the gradients as some libraries like `nccl` have implemented highly efficient `all-reduce` operations.

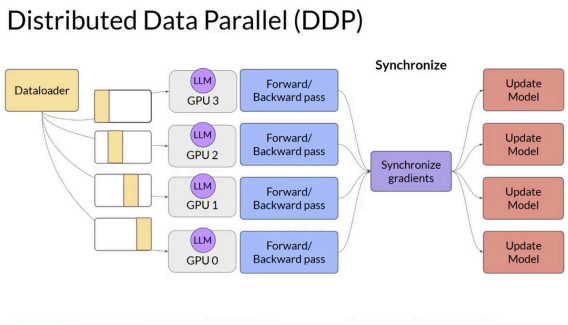


Fig. 3: Distributed Data Parallelism Implementation

As we can see from the Fig. 3, the synchronized process is no longer handled by parameter server, which is inefficient and memory-consuming. Currently, utilizing `nccl` backend provided by nvidia and DDP package provided by torch are a trending way in data parallelism.

B. Model Parallelism: Pipeline Parallelism

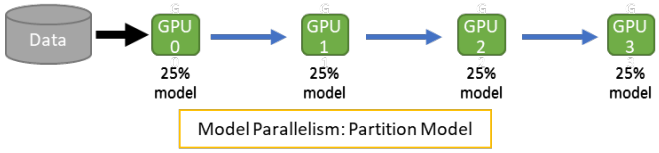


Fig. 4: Pipeline Parallelism Overview

Pipeline parallelism is initially proposed by GPipe [2]. It is a model parallelism strategy where we divide the model by its layers.

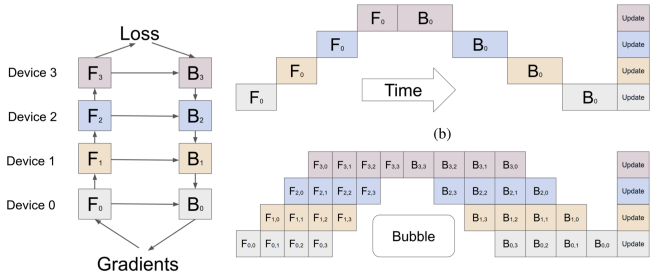


Fig. 5: GPipe Implementation

From Fig. 5, we can see that a major problem induced by pipeline parallelism is bubble. Through batching and asynchronize communication we can reduce bubble to a certain extent.

C. Model Parallelism: Tensor Parallelism

If we refer to pipeline parallelism as dividing the model horizontally, then we can refer to tensor parallelism as dividing the model vertically. In Megatron-LM, we can see how this is realized.

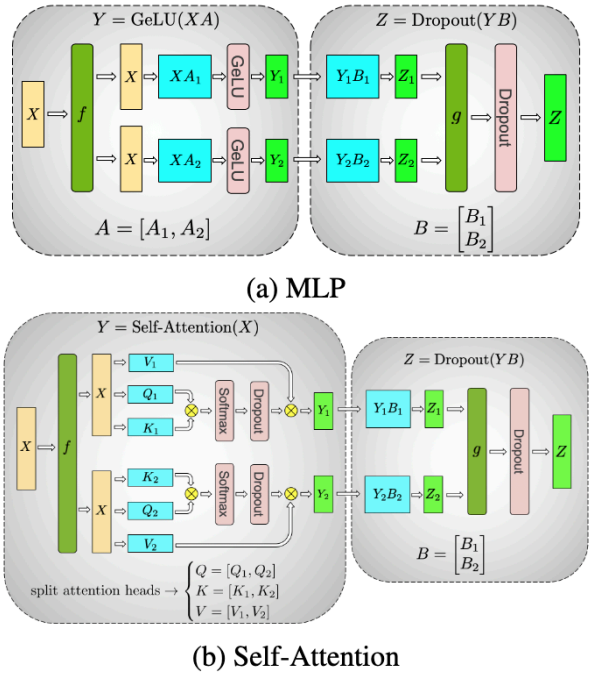


Fig. 6: Model parallelism for MLP blocks and Self-Attention blocks [3]

Using model parallelism, we will introduce an all-reduce on Z through the backward pass and an all-reduce on gradients through the forward pass. Therefore, an efficient communication library is the heart of implementing tensor parallelism.

D. Model Parallelism: Expert Parallelism

The expert parallelism is a parallelism method applied on Mixture-of-Expert model (MoE). A MoE model can be regarded as a layer in the whole model.

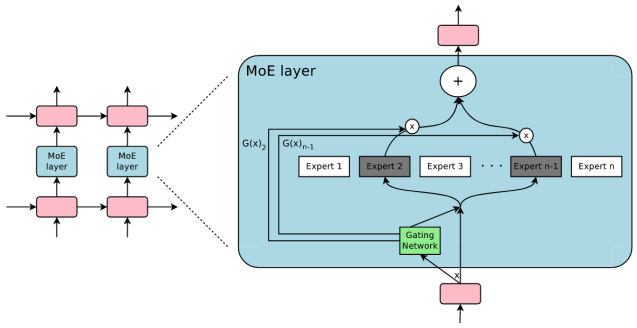


Fig. 7: A Mixture of Experts (MoE) layer embedded within a recurrent language model. In this case, the sparse gating function selects two experts to perform computations. [4]

Using MoE can help model to separate different features and increase the accuracy, but it also induces problems like increasing parameters and biased routing problems. Expert parallelism divides different experts to different rank and use the gating network to decide on which rank should the input be sent to, which greatly decreases the load on one single machine. [4]

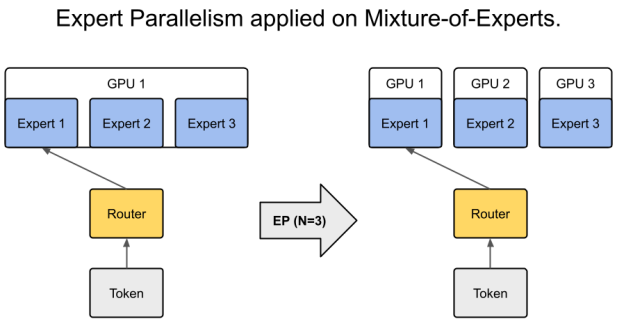


Fig. 8: Expert Parallelism, which can save the memory usage on single device by introducing extra communication overhead.

E. Sequence Parallelism

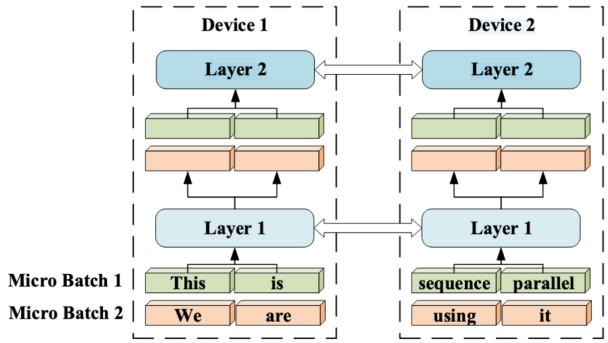
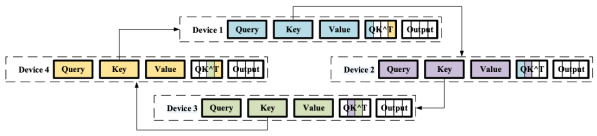
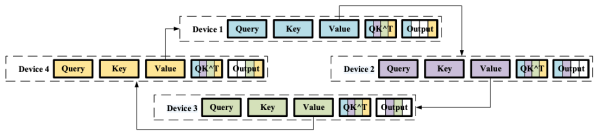


Fig. 9: Sequence Parallelism. Input sequences are split into multiple chunks and the sub-sequences are fed to different corresponding devices.[5]

At first look, sequence parallelism is much like data parallelism. However, while the data used in data parallelism shares no dependence with each other except fro probabilistic distribution, the sequence parallelism has to handle the case where two sequences might have high attention score with each other. Therefore, accompanied with Sequence Parallelism, Ring Self-Attention is proposed.



(a) Transmitting key embeddings among devices to calculate attention scores



(b) Transmitting value embeddings among devices to calculate the output of attention layers

Fig. 10: Ring Self-Attention.[5]

Through this, Sequence Parallelism can break the length limitation of Transformer model training. (Usually the transformer only takes in the sequence that is less than the maximum sequence length it supports.)

IV. Difficulties and Research Targets

In distributed machine learning, one of the primary challenges is efficiently **managing the trade-off between computation and communication** while ensuring accuracy. As models grow larger and data becomes more complex, the communication overhead between distributed workers or devices becomes a bottleneck that can limit performance. Furthermore, the increased complexity of model architectures, such as those involving hybrid parallelism (e.g., combining data and tensor parallelism), complicates the task of ensuring efficient load balancing and synchronization.

5 methods of distributed machine learning methods are introduced as above. Each of them has their own problems to fix.

Type	Difficulties
Data Parallelism	• Gradient synchronization • Communication overhead • Network latency impacts efficiency • Overlapping computation and communication
Pipeline Parallelism	• Bubbles due to imbalanced computation • Managing dependencies between stages • Inefficient resource usage from imbalanced stages
Tensor Parallelism	• Complexity of distributing tensors across devices • High communication costs • Synchronization issues between devices
Expert Parallelism	• Load balancing between experts • Biasness of distribution of tensors • Dynamic partitioning of work • Communication overhead for expert coordination
Sequence Parallelism	• Maintaining sequential dependencies during parallelization • Inefficiency from long sequences and order constraints

Apart from these, other fields in Machine Learning are also studied under distributed settings: Efficient Optimizer, Stochastic Gradient Descent, Error Feedback, Compression Kernel, etc.

For all the methods, the following difficulties cannot be ignored.

Fault tolerance. In distributed systems, failures are inevitable, and the system must be robust enough to handle node failures, network issues, and data inconsistencies without halting the training process. This requires advanced techniques for model recovery and checkpointing.

Scalability. As the number of devices or workers increases, maintaining scalability while keeping the system efficient and avoiding issues like network congestion, uneven workload

distribution, or memory limitations becomes increasingly difficult.

Designing algorithms and frameworks that can adapt to the varying capacities of **different hardware** (e.g., GPUs, TPUs) and **network configurations** adds an extra layer of complexity. The research targets in this domain are focused on improving communication efficiency, enhancing fault tolerance, and developing more adaptive, scalable architectures for distributed machine learning systems.

V. Current Breakthrough

A. Megatron

Megatron-LM is a large-scale language model training framework developed by **NVIDIA**. It is designed to efficiently scale the training of transformer-based models across multiple GPUs and nodes. Megatron-LM leverages model parallelism and mixed precision to maximize training speed and memory efficiency, enabling the development of large models like GPT and BERT at scale. We have already introduced Megatron in Tensor Parallelism as Megatron-LM is currently the most adequate and the most stable framework for supporting tensor parallelism. [3]

B. ZeRO

ZeRO (Zero Redundancy Optimizer) is an optimization technique for training large-scale deep learning models, developed by **Microsoft**. It reduces memory usage by partitioning the model states (like gradients, optimizer states, and parameters) across multiple GPUs, rather than replicating them. This enables training of extremely large models while maintaining high performance and reducing memory bottlenecks. ZeRO has in total 3 stages which incrementally shards optimizer states, gradients and parameters. [6]

C. DeepSpeed

DeepSpeed is a deep learning optimization library from **Microsoft** that aims to improve the scalability, efficiency, and training speed of large models. It integrates techniques like ZeRO, mixed precision, and model parallelism to enhance model training across multiple GPUs and nodes. DeepSpeed can utilize 3D parallelism, which stands for the mixture of using several parallelism strategies, enabling itself the capability to train models with billions of parameters efficiently. Currently, the DeepSpeed strategy has integrated into PyTorch.

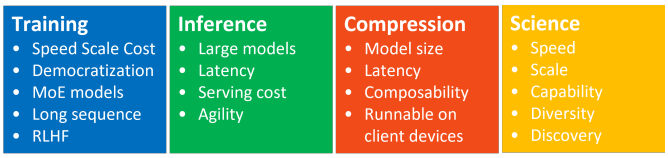


Fig. 11: DeepSpeed's four innovation pillars

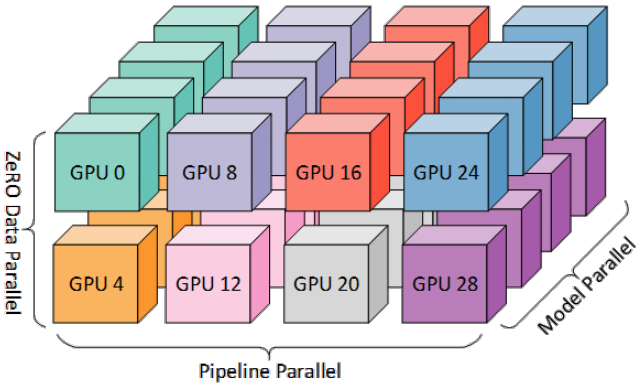


Fig. 12: 3D Parallelism used in Deepspeed.

D. Alpa

Alpa is a distributed deep learning framework designed to optimize large-scale model training on multiple devices. It focuses on **automatic parallelization** of model training using both data and model parallelism techniques. Alpa is optimized for ease of use, enabling seamless scaling of models. [7]

VI. Future Trend and Conclusion

- **Federated Learning** under heterogeneous data center setting. As the models become larger and larger, implementing methods that can train model over slow networks can help realize the training with less demand and reliance on data center and computation center.
- **Automatic Strategy Selection** like Alpa, to help ease the process of utilizing devices, methods that center around how to efficiently utilize the limited resources will be studied.
- **New Models** that adapt the distributed setting. Currently, we are using techniques to shard or partition given models to adapt the model to distributed settings. New models originally designed for distributed settings are yet to be designed.

References

[1] J. Dean *et al.*, "Large Scale Distributed Deep Networks."

[2] Y. Huang *et al.*, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2019. Accessed: Sep. 05, 2024. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html

[3] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism." Accessed: Sep. 05, 2024. [Online]. Available: http://arxiv.org/abs/1909.08053

[4] N. Shazeer *et al.*, "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer." Accessed: Sep. 05, 2024. [Online]. Available: https://arxiv.org/abs/1701.06538v1

[5] S. Li, F. Xue, C. Baranwal, Y. Li, and Y. You, "Sequence Parallelism: Long Sequence Training from System Perspective." Accessed: Sep. 05, 2024. [Online]. Available: http://arxiv.org/abs/2105.13120

[6] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models." Accessed: Sep. 05, 2024. [Online]. Available: http://arxiv.org/abs/1910.02054

[7] L. Zheng *et al.*, "Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning." Accessed: Oct. 26, 2024. [Online]. Available: http://arxiv.org/abs/2201.12023