

Exploring Distributed Training Architectures: From Foundational Machine Learning to Large Language Models

MARCO FASOLI, University of Genoa, Italy

VALERIA CARDELLINI*, University of Rome Tor Vergata, Italy

In recent years, deep learning has seen significant growth, achieving state-of-the-art performance across diverse domains, such as computer vision, natural language processing, and medical imaging. However, because processing massive datasets and models with billions of parameters outpaces the capacity of single computing nodes, transitioning to distributed machine learning systems has become necessary. In this survey, we provide an overview of the current distributed machine learning (DML) landscape, by first describing the foundational concepts for distributed learning systems and then analyzing distributed architectures such as Parameter Server, All-Reduce and Gossip Learning. We then dig into the specific challenges of training and handling large language models (LLMs), describing the advanced techniques and optimizations that have been developed to address the challenges of dealing with huge models and datasets. Finally, we conclude by highlighting open research issues in the field of DML and outlining future directions for research and development.

ACM Reference Format:

Marco Fasoli and Valeria Cardellini. 2026. Exploring Distributed Training Architectures: From Foundational Machine Learning to Large Language Models. 1, 1 (October 2026), 33 pages.

1 Introduction

During the past decade, the field of Deep Learning (DL) has witnessed extraordinary empirical success, as a result of breakthroughs in natural language processing [49], computer vision [27], and autonomous systems [5]. This progress is driven by multiple factors, including the increasing number of model parameters [25], the volume of training datasets [19], and the computational power of the hardware infrastructure.

In the modern DL landscape, due to the scaling laws [25] of large language models (LLMs), there is a strong scaling relationship between parameters and data, in which the required data volume grows with the model size. This results in a data-intensive paradigm that must handle tens to hundreds of terabytes (and occasionally petabytes) of raw data. From the point of view of the models themselves, the transition into the current era of LLMs was accelerated by transformer-based architectures, with models such as BERT [13] and GPT-3 [6]. The latter consisted of 175 billion parameters, was trained on 300 billion tokens, and this scaling trend continues with the latest models.

As the memory footprint and computational power required to train and handle these models have surpassed the physical capacity of any individual computing node, designing an efficient distributed training system has become

*Orthogonal School's Mentor

Authors' Contact Information: Marco Fasoli, University of Genoa, Genoa, Italy, marcofasoli92@gmail.com; Valeria Cardellini, University of Rome Tor Vergata, Rome, Italy, cardellini@ing.uniroma2.it.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

fundamental [41]. This has led to the development of distributed machine learning (DML) architectures that address the challenges of scaling out distributed training [16, 18, 29].

1.1 Objectives and Structure

The purpose of this paper is to provide a comprehensive survey of a representative set of state-of-the-art (SOTA) architectures for DML and systems for training LLMs, with a particular focus on the challenges and solutions related to scaling up model size, optimizing communication efficiency, and managing resource constraints using clusters of nodes. The paper is organized as follows:

- **Section 2: Background on Distributed Machine Learning** provides an overview of the fundamental DML concepts and a high-level description of what an LLM is. In this section, we introduce the limits of the centralized training paradigm and the motivation for distributed approaches, highlighting the core parallelism strategies, the essential topologies for distributed training systems, and the different consistency models.
- **Section 3: Distributed Architectures** provides a description of the three main architectures for DML, analyzing their trade-offs in terms of scalability, communication efficiency, and fault tolerance. In the last part of this section, we discuss the most important frameworks for distributed training, highlighting their features and limitations.
- **Section 4: Distributed Training of LLMs** discusses the specific challenges of training LLMs, including the importance of the communication infrastructure, advanced parallelism techniques, and specific optimizations for training such models. Additionally, it introduces specific frameworks that have been developed to train LLMs.
- **Section 5: Conclusions** synthesizes the remaining open research issues in the field of DML and outlines future directions for research and development.

2 Background on Distributed Machine Learning

Before examining specific distributed training architectures, this section covers the core principles of DML and the trade-offs between model size, compute efficiency, and network overhead.

We first begin by reviewing the fundamentals of empirical risk minimization and stochastic gradient descent, which provide the basic mathematical framework for model training. We then introduce LLMs, which represent the main focus of this study, highlighting the key architectural components. Following this, we discuss why workloads are scaled horizontally. Next, we analyze the core principles of distributed training systems: partition strategies, communication topologies, and synchronization consistency models. Finally, we address the primary bottlenecks that limit efficiency when scaling out.

2.1 Machine Learning and Stochastic Gradient Descent

In supervised machine learning (ML), the goal is to find the parameters w that minimize the expected value of a given loss function $L(y, f(w; x))$. Here, $x \in X$ represents the input, $y \in Y$ denotes its corresponding true output label, and $f(w; x)$ is the model's prediction. In other words, the loss function $L(y, f(w; x))$ measures how close the predicted label is to the ground-truth. In this way, the model learns the pattern in the data and is able to make accurate predictions on new data by minimizing the loss function.

To estimate the expected risk, empirical risk minimization (ERM) is performed on a training dataset $D_{\text{train}} = \{(x_i, y_i)\}_{i=1}^{n_{\text{train}}}$ composed of n_{train} independent and identically distributed (IID) samples. Mathematically, the optimization

problem (Eq. 1) and the gradient descent algorithm (GD) (Eq. 2) are defined as:

$$w^* = \arg \min_w L(w; D_{\text{train}}) = \arg \min_w \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} L(y_i, f(w; x_i)) \quad (1)$$

$$w_t = w_{t-1} - \eta \cdot \nabla L(w_{t-1}; D_{\text{train}}) \quad (2)$$

where t denotes the training step, w_t represents the model parameters at step t , and η represents the learning rate hyperparameter. However, traditional GD computes the multi-variable derivative on all the training examples at each iteration, which makes it computationally expensive and often infeasible to apply on large datasets.

To alleviate this problem, Stochastic Gradient Descent (SGD) is introduced and computes gradients using only a single randomly selected data sample d_t (or a small mini-batch) at each timestamp t . The parameters are updated according to the following relation:

$$w_t = w_{t-1} - \eta \cdot \nabla L(w_{t-1}; d_t) \quad (3)$$

In this way, a trade-off between computational efficiency and convergence quality is achieved [45].

The shift from ML to DL is marked by systems that also learn the representation of the input data. This leads to additional parameters that allow the models to capture complex patterns and relationships in the data. For this reason, the number of parameters in DL models is usually very large, so the optimization problem becomes complex and requires sophisticated techniques to guarantee convergence. In particular, SGD is often combined with optimization techniques such as momentum, adaptive learning rates, and regularization methods to improve convergence speed and generalization performance. An example is the Adam optimizer [26], which combines first-moment (momentum) and second-moment estimates of the gradients to adaptively adjust the effective learning rate for each individual parameter.

In DL pipelines, the process of returning the model's predictions ($f(w; x)$) is called the *forward pass*, while we call *backward pass* the process of computing the gradients (and updating of the parameters). These forward and backward steps are executed repeatedly over the whole dataset, in cycles known as *epochs*, to obtain the final trained model.

2.2 Foundational Concepts of LLMs

LLMs are a class of deep neural networks optimized to work on natural language. Given a preceding text context, they are specifically trained to predict the next token in a text sequence in an auto-regressive manner.

The architecture of LLMs is typically based on the Transformer model [49], which uses self-attention mechanisms to capture dependencies in text data. The original transformer architecture consists of an encoder-decoder structure, while modern models tend to be decoder-only. In general, LLMs consist of multiple stacks of these blocks to capture more complex patterns.

The attention mechanism works by computing attention scores that determine how much each token should contribute to the representation of the current token being processed. The transformer architecture also consists of feed-forward neural networks, layer normalization, and residual connections to enhance stability and generalization.

Within each attention layer, input activations are projected into query (Q), key (K), and value vectors (V) using dense weight matrices. The canonical attention mechanism is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V$$

where d_k is the dimensionality of the key vectors.

However, these multi-head attention layers and Multi-Layer Perceptrons (MLPs) have billions of parameters, making LLMs significantly memory-intensive and computationally demanding.

2.3 Limits of the Centralized Training Paradigm

The process of training modern DL models typically requires computational power that cannot be handled by a single machine. This has resulted in the development of distributed training paradigms that partition the workload across multiple nodes in a cluster [10].

In this approach, instead of *scaling up* the infrastructure by adding more powerful hardware to a single machine, the standard is to *scale out* the system. This results in a distributed architecture where multiple nodes collaborate as a single unit. In particular, distributed systems have the following advantages:

- *Lower equipment cost*: it is often cheaper to deploy a cluster of commodity hardware nodes rather than to invest in a single high-end machine. Moreover, this approach allows for incremental scaling if needed.
- *Fault resilience*: the system no longer depends on a single point of failure. This is especially important for long-term training jobs [22] where failures are more common to happen.

Despite these advantages, horizontal scaling is not a silver bullet. In fact, the communication overhead and synchronization requirements of distributed training can introduce significant bottlenecks that can compromise the efficiency of this approach. For this reason, in this work we investigate the different approaches to overcome these challenges.

2.4 Partitioning Strategies

We can partition the training workload across a distributed system in several ways, and the selection of the partitioning strategy is critical, as these approaches significantly impact training efficiency. The main strategies for partitioning the workload in DML include data parallelism, model parallelism, pipeline parallelism, and hybrid approaches.

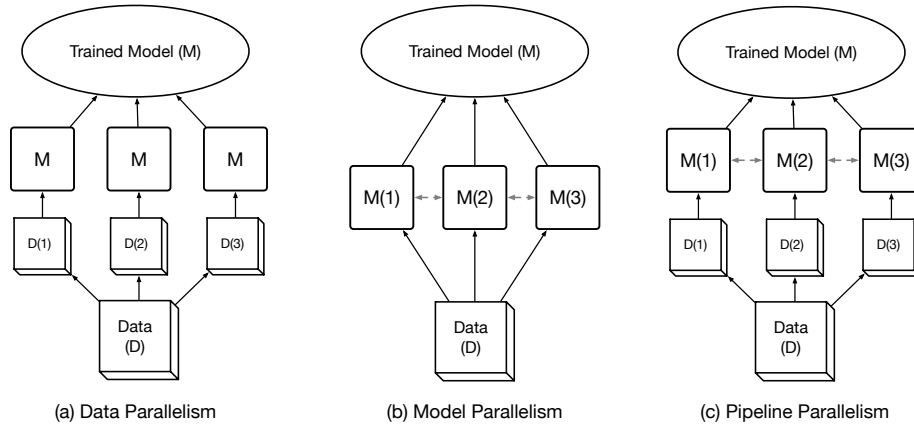


Fig. 1. Partitioning Strategies.

2.4.1 Data Parallelism. In data parallelism, a replica of the entire model is assigned to each node. The training dataset, on the other hand, is partitioned into distinct shards that are distributed across the nodes. Each worker computes

gradients on its assigned data shard and then synchronizes with the other nodes of the cluster to update the model weights, as illustrated in Figure 1 (a).

This approach is frequently used in practice due to its simplicity and general applicability to any deep learning architecture. However, it can encounter significant communication bottlenecks when synchronizing large models [24]. Moreover, data parallelism requires that the entire model fits within each node’s memory, a constraint that modern LLMs easily violate.

2.4.2 Model Parallelism. In model parallelism, the model itself is partitioned across multiple nodes, with each node responsible for computing a subset of the model’s layers or parameters, as shown in Figure 1 (b). In this case, the training dataset is not partitioned. This approach allows for training models that are too large to fit into the memory of a single node. However, it introduces additional complexity in terms of communication and synchronization, as nodes must exchange intermediate activations and gradients during the forward and backward passes.

A main challenge in model parallelism is designing an efficient partition of the model that minimizes communication overhead while balancing the computational load across nodes [33]. This often requires careful consideration of the model architecture and dependencies between layers.

2.4.3 Pipeline Parallelism. This approach applies inter-layer model parallelism while splitting each global batch into smaller micro-batches. In pipeline parallelism, the model is divided into stages and the dataset into shards. Each stage of the model is assigned to a different node. When the first node completes the forward pass of its assigned stage, it sends the intermediate activations to the next node in the pipeline, which continues the forward pass, as shown in Figure 1 (c). Instead of waiting for the entire batch to be processed, the first node can start processing the next micro-batch of data, creating a pipeline of computations across the nodes. This approach can improve resource utilization and throughput, but it also introduces challenges in terms of load balancing and managing the dependencies between stages.

2.4.4 Hybrid Approaches. DL models often consist of heterogeneous sub-architectures and this requires a combination of different parallelization strategies. With hybrid approaches, the system can use different paradigms of parallelism for different parts of the model, optimizing for both memory usage and communication overhead. For example, convolutional neural networks (CNNs) that consist of both convolutional and fully connected layers may benefit from a hybrid approach, where we can train convolutional layers using data parallelism, while fully connected layers are trained using model parallelism [23]. This allows for more efficient utilization of computational resources and can lead to faster convergence.

2.5 Centralized vs. Decentralized Topologies

Having examined the various partitioning approaches for DML, it is clear that carefully designing a distributed training architecture is fundamental to realizing the practical benefits of scaling out. This requires selecting an appropriate communication topology that outlines the specific role assigned to each node in the training system. In particular, distributed networks are fundamentally divided into centralized and decentralized architectures.

2.5.1 Centralized Architectures. This architectural design structures the cluster of nodes in a hierarchical layout, where part of the nodes are dedicated to hosting model parameters, while the remaining nodes are responsible for computing local gradients over assigned data shards. In particular, the nodes responsible for computing the local gradients are called *worker nodes* and they send their computations to the *server nodes* which update the weights according to the

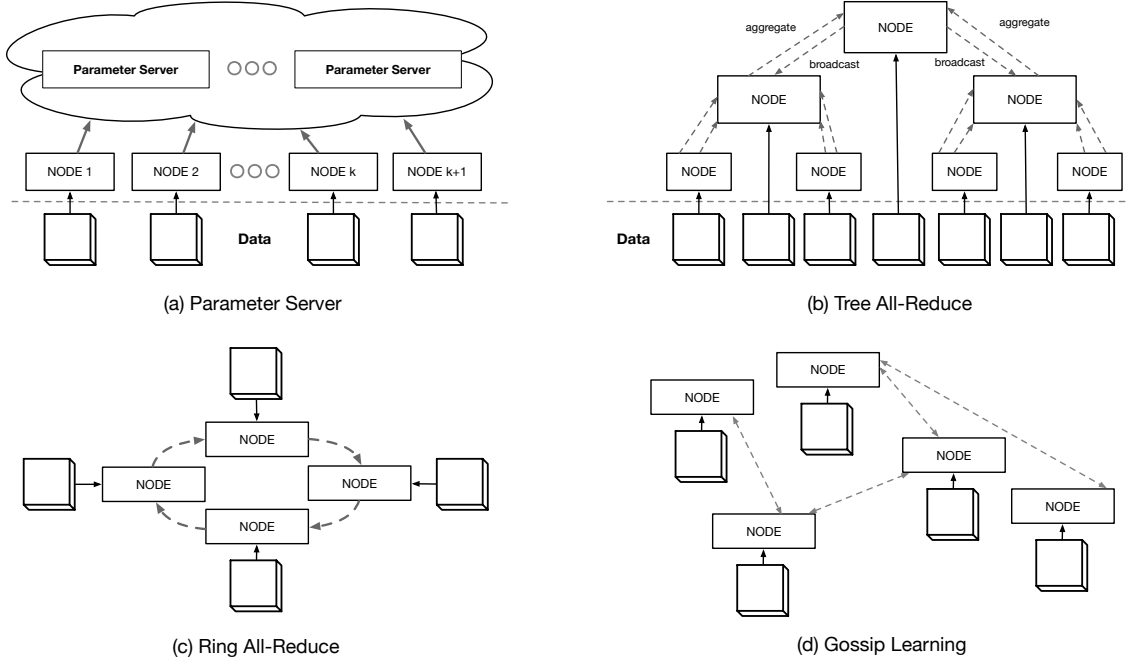


Fig. 2. Distributed Architectures.

received gradients, as can be seen in Figure 2 (a). This design is most prominently implemented via the Parameter Server (PS) [29] framework, which played a pivotal role in the early development of distributed deep learning systems.

2.5.2 Decentralized Architectures. Instead of having different roles assigned to different nodes, in a decentralized architecture all the nodes are identical (i.e., peers), and they all store a complete copy of the model parameters. Each node computes local gradients on its assigned data shard and then synchronizes with its neighbors via collective communication operations, typically implemented through an All-Reduce [16] primitive. Although these architectures do not suffer from the bottleneck of a centralized server, the topology of the communication network can significantly impact the overall training efficiency. The most common decentralized topologies are Ring All-Reduce [16], where each node communicates only with its two immediate neighbors in a logical ring structure (as shown in Figure 2 (c)), and Tree All-Reduce [2], where nodes are organized in a tree structure to perform hierarchical communication (as shown in Figure 2 (b)).

2.5.3 Fully Distributed Architectures. Both centralized and decentralized architectures typically assume a controlled environment and a balanced distribution of data. In contrast, fully distributed architectures allow multiple clients to collaboratively train a model without sharing their raw data. In this setup, each client computes local updates based on its own data. Depending on the coordination structure, these updates are aggregated via a central server, as in Federated Averaging [34], or propagated across the network through peer-to-peer exchanges using gossip protocols, as implemented in Gossip Learning [18]. This approach is particularly useful in scenarios where data privacy is a concern, such as healthcare or finance, where sensitive information cannot be shared among nodes. An example of a fully distributed architecture using gossip is shown in Figure 2 (d).

2.6 The Synchronization Problem and Consistency Models

A core problem to address in data-parallel systems is determining *when* gradient updates must be synchronized by different nodes. In DML landscape, the three primary consistency models implemented are: synchronous, asynchronous, and bounded asynchronous.

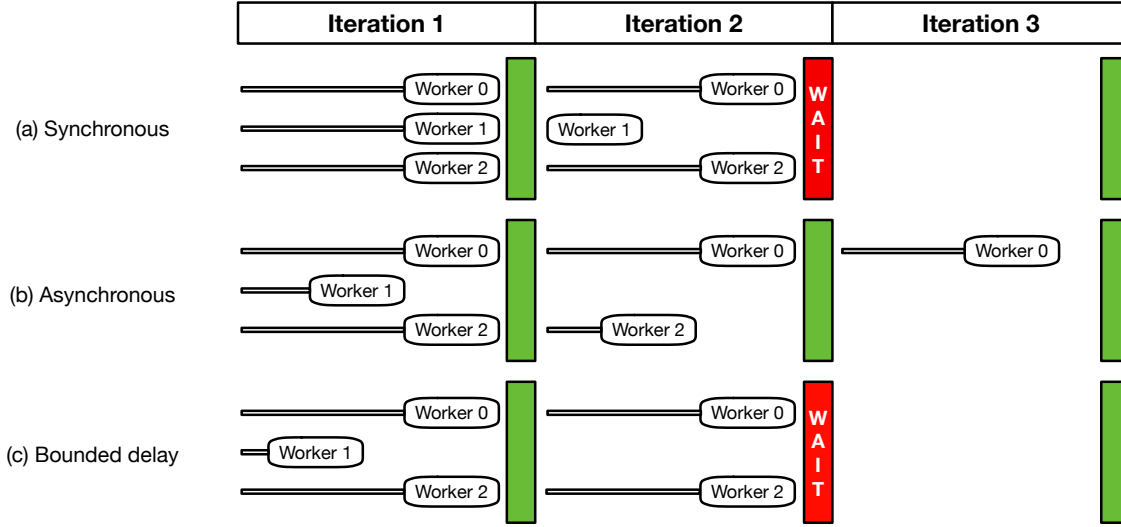


Fig. 3. Synchronization Models in Distributed Training.

2.6.1 Bulk Synchronous Parallel. In the Bulk Synchronous Parallel (BSP) framework, all worker nodes complete an iteration on assigned data mini-batches and then synchronize their parameter updates with the other nodes. This means that every worker must wait for all the others to complete their computations and communicate their gradient updates before proceeding to the next iteration, as shown in Figure 3 (a). This strict synchronization ensures that all workers operate on a consistent view of the model parameters. However, it makes the system vulnerable to the *straggler problem* [8], where slower nodes can delay the entire training process.

2.6.2 Asynchronous Training. In contrast, asynchronous training allows worker nodes to compute and apply updates to the model parameters independently, without waiting for other nodes, as shown in Figure 3 (b). In particular, when a worker node computes the gradients, it immediately applies the update to the global model state and shares it with the rest of the cluster. This can lead to a faster overall training time, as it eliminates the bottleneck caused by stragglers. However, since there is no check for staleness, there is the risk that *stale gradients* are applied to the global model, leading to divergence. Hogwild [43] is an asynchronous implementation of distributed SGD, which allows multiple nodes (or threads) to update shared parameters without locks, demonstrating that as long as the updates of the single worker only modify small parts of the model, the system can still converge effectively.

2.6.3 Bounded Asynchrony. In order to balance the trade-offs between strict synchronization and complete asynchrony, bounded asynchronous models have been proposed. These models allow for some degree of staleness in the updates, but set a limit on how outdated the gradients can be. This approach aims to mitigate the straggler problem while still maintaining a level of consistency that supports effective convergence.

In particular, the Stale Synchronous Parallel (SSP) [8] model relaxes the synchronization barrier by allowing faster worker nodes to advance ahead of slower worker nodes, but it enforces a user-defined maximum iteration staleness bound, denoted as τ . If the fastest worker in the cluster outpaces the slowest worker by more than τ iterations, the system automatically blocks the fastest node until the straggler catches up and commits its updates. An example of a bounded asynchronous model is shown in Figure 3 (c) with $\tau = 1$.

This approach tackles the straggler problem to a significant extent, but if the physical performance variance across cluster machines is excessively high or the staleness bound τ is configured too loosely, the accumulation of out-of-date updates can still slow the convergence.

2.7 Challenges of Scaling Out Distributed Training

In the previous sections, we have discussed the issue of handling a cluster of nodes for distributed training, which introduces systemic problems that need to be addressed in order to achieve the desired speedup. These challenges span across network communication, fault tolerance, and dynamic cluster scaling.

2.7.1 The Network Communication Bottleneck. The main challenge in scaling out distributed training is the network communication overhead. In fact, the communication of large amounts of parameter updates among nodes can become a significant bottleneck, leading to congestion and under-utilization of computational resources.

The literature categorizes communication mitigation techniques into three major strategies:

- **Model precision reduction:** instead of transmitting full-precision parameters, systems can compress the numerical data types to lower-bit alternatives (for instance, using 16-bit half-precision floating-point numbers), reducing the amount of data that needs to be sent over the network.
- **Compression:** techniques like gradient quantization and sparsification can be applied to reduce the size of updates. For example, only the most significant gradients can be transmitted, while smaller gradients are either dropped or approximated [29].
- **Communication scheduling:** by carefully scheduling when and how updates are communicated, systems can overlap communication with computation [51].

2.7.2 Fault Tolerance and Dynamic Scaling. Distributed architectures must also be designed to handle failures and dynamic changes in the cluster configuration. As the number of nodes increases, the likelihood of hardware failures also increases, making fault tolerance a critical aspect of distributed training systems. Additionally, the ability to dynamically scale the cluster by adding or removing nodes during training can provide flexibility and cost savings.

Ensuring fault tolerance in distributed training involves implementing mechanisms for detecting and recovering from node failures, such as checkpointing or replication. These solutions are also crucial for allowing dynamic scaling of resources during training without interrupting or restarting the training process.

3 Distributed Architectures

In this section, we provide an overview of the SOTA distributed architectures for training large models in clusters of nodes, analyzing their trade-offs in terms of communication efficiency, synchronization choices, and fault tolerance. For each architecture, there is also an overview of the different frameworks that implement these paradigms.

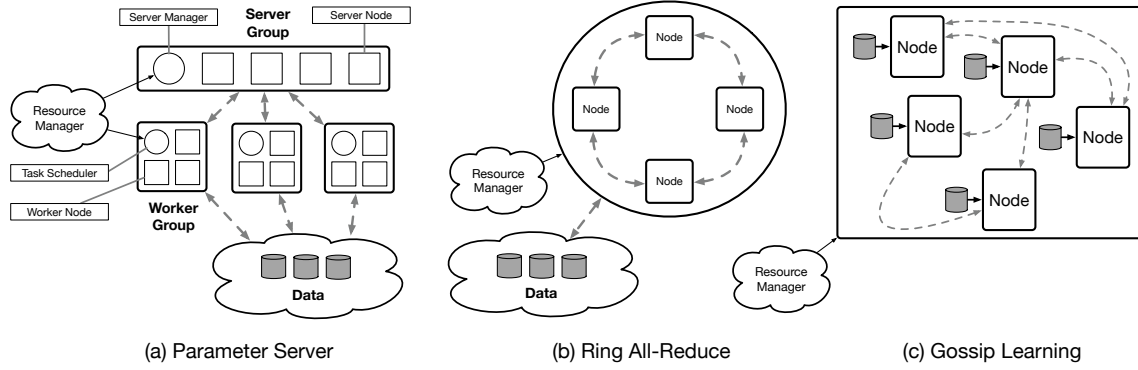


Fig. 4. Distributed Architectures.

3.1 Parameter Server

The third generation of Parameter Server (PS) [29] is a centralized architecture designed to address the challenges of ensuring efficient communication, fault tolerance, durability, and elastic scalability while being easy to use. This architecture was introduced in 2014 and played a fundamental role in allowing the training of models on a scale that was previously unfeasible.

The Parameter Server architecture is based on a master-worker paradigm, where the master nodes (i.e., the parameter servers) are responsible for storing and updating the model parameters, while the worker nodes perform the computation of gradients and send them back to the servers. In particular, the server nodes are grouped into a cluster managed by the server manager. Each of these nodes is in charge of a shard of the model parameters, implementing model parallelism (also called *sharding*). On the other hand, worker nodes perform the computation of gradients on their shard of the training data, assigned by the resource manager in a data-parallel fashion. Worker nodes do not need to have a complete copy of the model parameters, as they can request the necessary parameters with respect to the data they are working with. A practical example of how the PS architecture works can be found in Appendix A, while a visualization is shown in Figure 4 (a).

3.1.1 Communication Efficiency. Due to the centralized nature of the PS architecture, the authors of the original paper [29] proposed a set of techniques to improve communication efficiency. In particular, the model is represented as a key-value dictionary. This allows for the use of a push-pull communication model, where workers ask for the parameters they need by listing the keys of the desired parameters. After the computation of the gradients, the workers send the updates to the servers using a push command. This approach is called the *range push and pull* and allows for a significant reduction in communication overhead, as workers only need to communicate the parameters that are relevant to their computation.

Using this representation of the model parameters, the messages sent between workers and servers can be quite redundant, as workers often work on the same parameters. This means that parameter keys are sent multiple times, which leads to significant communication overhead. To address this issue, the authors proposed a technique called *key caching*, where servers cache the keys associated to a specific worker and from that point on they can send the value of the parameters and the hash of the keys, substantially reducing the communication overhead.

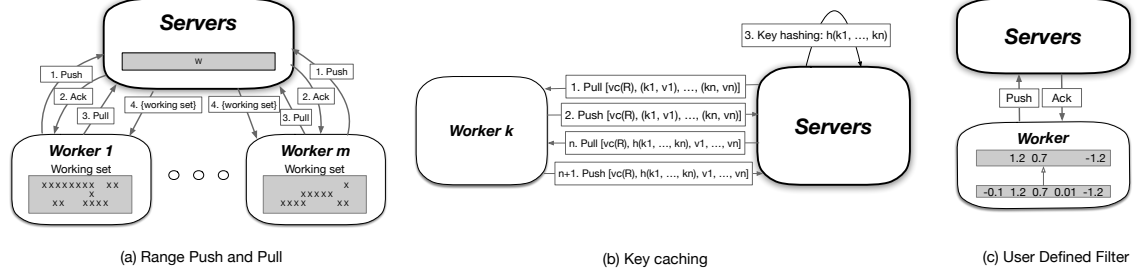


Fig. 5. Parameter Server Communication Optimizations.

Other techniques to improve communication efficiency include the use of User Defined Filters (UDFs) and compression algorithms. UDFs allow the user to define a filter in worker nodes to send only the most significant gradients, while compression algorithms are implemented in the server nodes to drop the parameters with zero values, further reducing the communication overhead.

3.1.2 Synchronization. The PS architecture allows for both synchronous, asynchronous and bounded asynchronous training. Indeed, each task (which may be a pull request or a computation of gradients) is computed by the workers asynchronously, but it is possible to define dependencies between tasks so that the workers can be forced to wait for the completion of a task before starting another one.

This allows for different levels of consistency of the model parameters, depending on the synchronization strategy used. Depending on the specific use case, it may be more beneficial to use a synchronous approach or an asynchronous one. For example, in a topic modeling task, the authors found that the asynchronous approach was more efficient, while in the case of fitting a sparse logistic regression model, the bounded asynchronous approach was more suitable.

3.1.3 Fault Tolerance. Since the PS architecture is based on a centralized model in which nodes have different roles, it is important to handle failure with respect to the different types of nodes. In particular, the failure of a server node can lead to the loss of model parameters, which is a major concern, while the failure of a worker node can lead to the loss of gradients.

In order to handle server nodes failure, the authors proposed to partition the parameters as a conventional distributed hash table [47]. Specifically, parameter keys and server node IDs are inserted into the hash ring, and each server node is responsible for the range of keys from its ID to the next server node's ID in counterclockwise direction. This is crucial in case of failure, as the next server node in the ring can take over the responsibility of the failed node and continue to serve the parameters.

Using this approach, chain replication can be implemented, where each server node is responsible for its range of keys and also of the one of the next server node in the ring. This allows for a high level of fault tolerance, as the failure of a server node does not lead to the loss of model parameters, as they are replicated. During the communication of the parameters, range vector clocks are used in order to ensure that the parameters are correctly updated, without losing any updates or adding the same update multiple times.

While the failure of a server node is critical and it always needs to be handled through consistent hashing, the failure of a worker node is less critical. In this case, recovery of the worker is optional and the system can continue to operate without it. However, if the worker needs to be recovered (for instance, in case of a small number of nodes in the cluster),

the resource manager can re-assign a shard of the training data to the worker node and it can re-join the training process.

Both the consistent hashing for server nodes and the handling of worker nodes can also be used to scale the system, where nodes can be added or removed from the cluster. In this case, the consistent hashing allows for a smooth transition of the model parameters to the new server nodes, while the resource manager can re-assign the training data to the new worker nodes.

3.1.4 Pros and Cons. The PS architecture has several advantages, including the ability to handle large models and datasets through the sharding of both parameters and data. It also supports flexible consistency of the model parameters and fault tolerance. It is best suited for training large models in which the gradient updates are sparse, as it allows for efficient communication. Additionally, than to the different synchronization strategies, it is suited to handle clusters of heterogeneous nodes effectively.

A main shortcoming is poor performance with dense gradients, as the communication overhead can become a bottleneck due to the system centralized nature.

3.1.5 Frameworks. The PS architecture has evolved through multiple generations to address the scaling limits of massive deep learning workloads. The first generation lacked specialized optimizations for distributed training, while the second generation introduced custom update primitives and a more efficient communication protocol. Prominent examples of this generation include YahooLDA and Google’s DistBelief [10]. The third generation, which is the focus of this section, introduced a more robust and scalable architecture with improved communication efficiency, fault tolerance, and elastic scalability. Notable frameworks that implement the third generation of PS include MXNet [7] and TensorFlow [1]. These frameworks provide the necessary tools and abstractions to efficiently train large models in distributed environments, making them suitable for a wide range of applications in deep learning workloads. To support high-performance hardware acceleration, specialized variants have been developed, such as GeePS [9], which is designed to leverage GPU clusters to train large models.

This architecture has been adopted and extended by frameworks such as Apache SINGA¹, and it became a foundational component in recommendation systems such as ByteDance’s Monolith².

3.2 All-Reduce

The All-Reduce architecture is based on a decentralized paradigm, where all nodes play the same role and communicate in a peer-to-peer fashion. Derived from the High-Performance Computing (HPC) domain³, All-Reduce [16] was adapted for deep learning workloads to address the challenges of scaling out distributed training, avoiding the structural bottlenecks of centralized architectures like the Parameter Server.

In a standard data-parallel fashion, every node maintains a complete local copy of the model parameters and computes gradients on its respective shard of the training data. After computing the gradients, the nodes collectively execute an All-Reduce operation to aggregate the gradients across the entire cluster. This ensures that every node receives the identical set of updated model parameters, maintaining consistency across the distributed system. A visualization of the All-Reduce architecture is shown in Figure 4 (b).

¹<https://singa.apache.org/>

²<https://github.com/bytedance/monolith>

³AllReduce is a standard collective communication primitive defined in the Message Passing Interface specification.

3.2.1 Communication Efficiency. The communication efficiency of the All-Reduce approach relies on two main collective primitives: Reduce-Scatter, which aggregates gradients across all nodes and distributes chunks of the combined result to each worker, and All-Gather, which then broadcasts those chunks so that every node receives a complete copy of the fully updated gradients. In order to optimize data transfer, there exist two different designs for this architecture: *ring-based* and *tree-based* All-Reduce.

In the ring-based All-Reduce, nodes are organized into a logical ring where nodes communicate with their immediate neighbor. Instead of sending all the gradients at once, the gradient vector is divided into N equal segments (where N is the number of nodes in the cluster). Then, the two primitives are executed in the following way:

- **Reduce-Scatter phase:** each node concurrently sends a gradient segment to its successor, while receiving a different segment from its predecessor. When a node receives a segment, it performs a sum operation with its local segment and forwards the result to the next node. At the end of this phase, each individual node holds a unique segment of the aggregated gradients.
- **All-Gather phase:** to ensure that all nodes have the complete set of aggregated gradients, the completed segments are exchanged among the nodes in a similar manner. During this phase, nodes receive the fully summed segments and overwrite their local values.

An example of how the Ring All-Reduce architecture works can be found in Appendix B. This ring-based architecture optimizes the bandwidth usage. However, in order to handle massive clusters, the tree-based All-Reduce [2] is sometimes preferred since it lowers the communication steps. In this design, nodes are organized into a hierarchical tree structure, where each node communicates with its parent and child nodes. In this case, the primitives change slightly:

- **Up-Tree Reduce phase:** leaf nodes pass their local gradient up to their designated parent node, which aggregates the gradients from its children and forwards the result up the tree. This process continues until the root node has the complete aggregated gradient.
- **Down-Tree Broadcast phase:** once the root node has the complete aggregated gradient, it broadcasts the result down the tree to all nodes.

3.2.2 Synchronization. The All-Reduce architecture implements the BSP synchronization paradigm, forcing a strict synchronous execution flow. Workers independently compute their local gradients and then they must wait until all other nodes have completed their computations before executing the communication primitives.

While this approach simplifies model convergence, it introduces the straggler problem, meaning that the overall training time is determined by the slowest node in the cluster, which can lead to inefficiencies and under-utilization of resources.

3.2.3 Fault Tolerance. Fault tolerance in the All-Reduce architecture is more challenging compared to the PS architecture, due to the fact that each node represents a critical component of the communication topology. In fact, the failure of a single node breaks the communication chain and stops the entire training process.

This issue is typically managed through checkpointing, which means that the model state and optimizer variables are periodically saved to a shared file system, such as HDFS. When a node fails, the entire cluster must be restarted from the last checkpoint, which can lead to significant overhead and increased training time.

3.2.4 Pros and Cons. The All-Reduce architecture achieves bandwidth utilization that is independent of cluster size via ring primitives or provides scalable step latency via tree topology, allowing practitioners to choose the approach

best suited to the cluster size and network characteristics. Furthermore, it is well suited for training models with dense gradients, as it avoids the communication bottleneck of the centralized Parameter Server.

However, its performance degrades significantly in the presence of stragglers, meaning that it is suited for homogeneous clusters where nodes have similar computational power and memory capacity. Additionally, fault tolerance is more challenging compared to the PS architecture, as the failure of a single node can halt the entire training process.

As demonstrated by Lian et al. [32], decentralized architectures can outperform centralized architectures when the communication network is slow. This was validated in a synchronous setting against a PS architecture with a single server node. Furthermore, Iandola et al. [21] showed that a tree-structured all-reduce is preferred to a single PS to optimize throughput and scaling.

3.2.5 Frameworks. The adoption of All-Reduce in DL was driven by Baidu’s introduction in 2017, inspiring the development of Horovod [46], a framework that implements the All-Reduce architecture and provides a simple interface for distributed training.

Today, the most important implementations are PyTorch Distributed Data Parallel (DDP) [31] and NVIDIA’s Collective Communications Library (NCCL) [37], which provide optimized All-Reduce operations for GPU clusters. These frameworks have become the standard for distributed training, enabling efficient scaling of large models across multiple nodes and GPUs.

For training extremely large models, systems like Microsoft DeepSpeed [11] and NVIDIA’s Megatron-LM [36] have been developed, which combine the All-Reduce architecture with model parallelism techniques to handle models that exceed the memory capacity of a single GPU.

3.3 Gossip Learning

Gossip Learning (GL) [18] is a fully decentralized architecture based on a peer-to-peer communication model. Introduced in the early 2010s, it is designed to train models while retaining privacy and security of local data without central coordination.

In this architecture, every node operates as an independent entity, maintaining its own local copy of the model parameters and training on its local data. Periodically, each node selects random peers from the network to exchange and aggregate model parameters, allowing for the diffusion of information across the network, as shown in Figure 4 (c).

3.3.1 Communication Efficiency. The communication efficiency of GL relies on peer-to-peer communication. Instead of forcing all nodes to broadcast the gradients simultaneously, the architecture relies on a Random Peer Sampling (RPS) strategy, where each node selects a random subset of peers to communicate with. Once this subset of peers is selected, the communication is driven by two primitives:

- **Push:** each node sends its local model parameters to the selected peers.
- **Pull:** each node requests the model parameters from the selected peers.

As a last step, *aggregation* is computed when each node aggregates the received model parameters with its local model parameters, using a predefined aggregation function (e.g. averaging). Since the DL models contain a large number of parameters, some implementations use Segmented GL, where the model parameters are divided into segments and each segment is exchanged with a different subset of peers. This allows for a more efficient use of network bandwidth and reduces the communication overhead. A practical example of how the Gossip Learning architecture works can be found in Appendix C.

3.3.2 Synchronization. GL uses an asynchronous execution flow, where nodes independently compute their local gradients and communicate with their peers, without any synchronization barrier. This eliminates the straggler problem; however, it results in weaker consistency of the model parameters, which can lead to slower convergence.

3.3.3 Fault Tolerance. Fault tolerance is a structural advantage of the GL architecture, as it is immune to single points of failure and requires no complex recovery protocols. Indeed, the entire architecture is built to handle node churn, where nodes can join and leave the network at any time. If a node fails or leaves the network, its neighbors bypass it during the next RPS cycle, meaning that the failure of a single node results only in a loss of compute power and specific data updates.

3.3.4 Pros and Cons. GL is well suited for training models in environments where privacy and security are a priority, as it allows training on local data without the need to share sensitive information. It is highly resilient to failures and scales well in large networks, as the communication overhead is distributed across the nodes.

However, it suffers from slower convergence due to weak consistency of the model parameters and it struggles significantly with non-IID data distributions, where the local data of each node is not representative of the overall data distribution.

3.3.5 Frameworks. It is worth noting that GL is not yet widely adopted in the deep learning community, and there are limited frameworks that implement this architecture. Nevertheless, the Gossip Learning Framework (GoLF) [39] provides a benchmarking suite to evaluate P2P ML algorithms. For applied work, P2PFL is a library that embeds gossip communication protocols into PyTorch and TensorFlow pipelines.

3.4 Additional Frameworks

In the previous sections, we have discussed the main frameworks that implement the three distributed architectures. However, there are other frameworks that implement specific parallelism techniques that are not tied to a specific architecture and in this section we discuss a relevant, non-exhaustive selection of them.

3.4.1 GPipe. Developed by Google, GPipe [20] is a framework that introduces a pipeline parallelism approach to training deep neural networks. It uses a synchronous SGD algorithm and uses activation recomputation to reduce memory usage during training, making it suitable for training very deep models. However, the synchronous nature of GPipe introduces pipeline bubbles, which can lead to under-utilization of resources and increased training time. Moreover, its heavy reliance on activation recomputation introduces computational overhead, which can slow down the training process.

3.4.2 PipeDream. Developed at the same time as GPipe, PipeDream [35] takes the opposite trade-off. In fact, it overlaps computation with communication by allowing multiple mini-batches to execute concurrently across devices in a 1F1B (one forward, one backward) manner. Additionally, it incorporates an automated planner that determines a partitioning that minimizes its estimated pipeline cost across devices, minimizing pipeline bubbles and improving resource utilization. However, the asynchronous nature of PipeDream can lead to stale gradients, which can affect the convergence and accuracy of the model. For this reason, PipeDream maintains multiple weight versions in memory simultaneously, which significantly increases memory overhead.

3.4.3 DAPPLE. DAPPLE [14] is a synchronous training framework that explores hybrid parallelism strategies, combining data parallelism and pipeline parallelism. It introduces an automated parallelization strategy planner that partitions

model layers into pipeline stages and distributes training data across nodes based on cluster topology. Unlike PipeDream, DAPPLE couples this hybrid approach with a synchronous 1F1B execution strategy. However, its limitations include the search overhead of the planner, and the performance bottlenecks introduced by strict synchronization in heterogeneous clusters.

3.4.4 Summary. In this section, we have discussed the main distributed architectures for training a ML model in a cluster of nodes, analyzing their trade-offs in terms of communication efficiency, synchronization choices, and fault tolerance. In Table 1, we provide a summary of the most important frameworks that implement these architectures, highlighting their features and limitations.

Table 1. Comparison of Selected Distributed Deep Learning Frameworks.

Framework	Parallelism	Comm. Primitives	Synchronization	Pros	Cons
MXNet [7]	Data, Model	Key-Value Push/Pull	BSP, ASP, Bounded	Elastic scaling, key caching	Server bottleneck, dense overhead
TensorFlow [1]	Data, Model	Key-Value Push/Pull	BSP, ASP, Bounded	Wide adoption, flexible sync	Centralized bottleneck, complex setup
GeePS [9]	Data, Model	Key-Value Push/Pull	BSP, ASP, Bounded	GPU optimization, memory management	Server bottleneck, legacy design
Horovod [46]	Data	Ring All-Reduce	Synchronous (BSP)	Bandwidth-efficient, simple integration	Straggler sensitive
PyTorch DDP [31]	Data	Ring / Tree All-Reduce	Synchronous (BSP)	Native performance, high throughput	Straggler sensitive
GoLF [39]	Data	Peer Push/Pull (RPS)	Asynchronous	Churn resilient, privacy preserving	Slow convergence, non-IID sensitive
GPipe [20]	Pipeline	Point-to-Point	Synchronous (Micro-batches)	Exact SGD, activation saving	Pipeline bubbles, re-computation cost
PipeDream [35]	Pipeline	Point-to-Point	Asynchronous (1F1B)	Minimal bubbles, auto partitioning	Weight staleness, memory footprint
DAPPLE [14]	Hybrid	P2P, All-Reduce	Synchronous (1F1B)	Stage replication, zero staleness	Complex setup, planner overhead

4 Distributed Training of LLMs

In the previous sections, we have provided an overview of the challenges and techniques involved in distributed training of general DL models, also discussing the trade-offs of specific approaches. The training of more modern LLMs presents additional challenges that require a more specialized approach, due to the scale of the models and the complexity of the training process. In this section, we dive into the specific aspects of distributed LLM training and discuss the main techniques developed to address these challenges.

4.1 The Parameter Barrier

Modern LLMs, such as Kimi K3 [48], consist of trillions of parameters and require massive amounts of computational resources to train. The paradigms that we presented in the previous sections introduced some techniques to tackle the challenges of distributed training, but they are insufficient to handle the scale of LLMs. In fact, traditional distributed DL architectures fail under this setting:

- The centralized Parameter Server suffers from network communication bottlenecks and high synchronization latency. Since LLM backpropagation produces dense gradients, the centralized communication model slows down the training process.
- The standard data parallelism (implemented, for instance, in Ring All-Reduce) cannot be applied to train LLMs because the model is too large to fit in the memory of a single GPU. Even if the parameters could fit in memory, for each computation step a single node should also store the activations and optimizer states, which increases the memory requirements even further. This makes it almost impossible to train LLMs using standard data parallelism.

Consequently, training LLMs requires specialized optimization paradigms, which we discuss in the following sections. In particular, we focus on communication hierarchies and data transfer in GPU clusters, advanced parallelism techniques, and memory-centric optimizations.

4.2 Communication Hierarchy and Data Transfer in GPU Clusters

The computational throughput of a distributed training system is bounded by its network efficiency, and the performance scales according to the following model:

$$\text{Total Training Speed} = \text{Single GPU Speed} \times N_{\text{chips}} \times R_{\text{speed_up}}$$

where N_{chips} represents the number of GPUs in the cluster and $R_{\text{speed_up}}$ is the speedup efficiency ratio. The latter is the main variable that a system designer controls, and the goal is to prevent $R_{\text{speed_up}}$ from degrading as N_{chips} grows. Every GPU idling for data is a GPU not computing, and that is what drives the ratio down; the duration of such stalls depends directly on the communication links data must traverse. To minimize this wait, modern architectures map workloads across a communication hierarchy structured as follows:

- (1) **High-bandwidth intra-node interconnects:** traditional server architectures use Peripheral Component Interconnect Express (PCIe) bus to send data between devices. In this setup, every GPU-to-GPU data transfer must cross the PCIe slot and be orchestrated by the host CPU's PCIe controller, introducing high arbitration latency and severe bandwidth saturation. To eliminate this bottleneck, NVIDIA developed NVLink [38], which is a high-bandwidth, low-latency interconnect that allows GPUs to communicate directly with each other without going through the CPU. This is extended by NVSwitch [38], which is a high-speed switch that allows multiple

GPUs to communicate with each other simultaneously. In this way, GPUs exchange data bypassing the host CPU, PCIe controller, and system RAM.

- (2) **Low-latency inter-node fabrics:** when the system scales to multiple nodes, data must be transferred across the network. Using TCP/IP stacks for inter-node communication introduces significant latency and bandwidth limitations because the host CPU must handle packet processing and routing. In order to address this, distributed clusters deploy Remote Direct Memory Access (RDMA) protocols, which allow GPUs to directly access the memory of remote nodes without involving the host CPU. This is implemented through high-speed interconnects such as InfiniBand or Ethernet with RDMA support, which provide low-latency and high-bandwidth communication between nodes.
- (3) **Hardware-tuned collective communication primitives:** to further optimize communication, modern distributed training architectures implement optimized communication libraries that are specifically tuned for the underlying hardware. Examples of these libraries include NVIDIA Collective Communications Library (NCCL) [37] and Huawei Collective Communication Library (HCCL). These software libraries detect the layout of the cluster and construct communication schedules matched to the cluster topology to execute collective communication tasks such as All-Reduce, Reduce-Scatter and All-Gather. In this way, these libraries act as a traffic control system that maximizes overall bandwidth utilization and minimizes latency across the entire cluster.

4.3 Advanced Parallelism Techniques

We discussed that when the model size exceeds the memory capacity of a single GPU, model parallelism is used to split the model across multiple GPUs. Within this context, model parallelism can be categorized into two primary paradigms based on partitioning granularity: general model parallelism and tensor parallelism. The fundamental difference between these two strategies lies in whether the model is segmented inter-layer or intra-layer. While inter-layer partitioning is sufficient for most DL models, LLMs require a more fine-grained approach to model parallelism. The size of the model parameters and the complexity of the computations involved in LLMs require the model to be split across multiple GPUs, even within a single layer, which is addressed by tensor parallelism.

4.3.1 1D Tensor Parallelism. Popularized by Megatron-LM [36], 1D tensor parallelism is a technique that partitions linear transformation operators across multiple GPUs. A standard linear transformation is defined as:

$$Y = XA$$

with A being the weight matrix and X the activation matrix. In 1D tensor parallelism, the weight matrix A is split along its columns:

$$A = \begin{bmatrix} A_1 & A_2 & \cdots & A_P \end{bmatrix}$$

where P is the number of GPUs. Each GPU computes a partial output $Y_i = XA_i$ and the final output is obtained by concatenating the partial outputs horizontally:

$$Y = \begin{bmatrix} Y_1 & Y_2 & \cdots & Y_P \end{bmatrix} = \begin{bmatrix} XA_1 & XA_2 & \cdots & XA_P \end{bmatrix}$$

When a second linear transformation $Z = YB$ is applied to the output Y , the weight matrix B is split along its rows:

$$B = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_P \end{bmatrix}$$

In order to compute $Z = \begin{bmatrix} Y_1 & \dots & Y_P \end{bmatrix} \times \begin{bmatrix} B_1 \\ \vdots \\ B_P \end{bmatrix}$, each GPU computes a partial output $Z_i = Y_i B_i$. The final output is obtained

by an All-Reduce operation that sums the partial outputs across all GPUs:

$$Z = \sum_{i=1}^P Z_i = \sum_{i=1}^P Y_i B_i$$

The problem with this approach is that this parallelization strategy is constrained by the dimensionality of the matrices, which limits the maximum number of GPUs that can be used for training. Specifically, the target dimensions of the weight matrices must be divisible by the number of GPUs (i.e., P). In LLMs, matrices are sharded along the Attention Head dimension. Consequently, if an LLM has 32 attention heads, it can be trained using 1D tensor parallelism on 1, 2, 4, 8, 16 or 32 GPUs. Furthermore, this approach does not partition the (intermediate) activations, which can lead to significant memory consumption.

4.3.2 2D Tensor Parallelism. To address the issue of partitioning also the activations, 2D tensor parallelism [52] was introduced. It enables sharding both model parameters and activations across a processor grid of $P = q \times q$ GPUs, based on the SUMMA matrix multiplication algorithm. Given $q = 2$, the activation matrix X and the weight matrix A are partitioned as follows:

$$X = \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix}, \quad A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$$

where each GPU (i, j) holds exactly X_{ij} and A_{ij} . The goal of each GPU (i, j) is to compute its sub-block of the final output matrix Y :

$$Y_{ij} = \sum_{k=0}^{q-1} X_{ik} A_{kj}$$

Since each GPU initially holds only X_{ij} and A_{ij} , it lacks the other blocks along its row and column to complete this summation. To resolve this without replicating the full matrices, the execution runs through q synchronized communication and computation steps:

At step $t = 1$, X_{i0} is broadcast horizontally along the rows of the processor grid, while A_{0j} is broadcast vertically along the columns. Then, X_{i0} is multiplied with A_{0j} on each GPU as follows:

$$\begin{bmatrix} X_{00}A_{00} & X_{00}A_{01} \\ X_{10}A_{00} & X_{10}A_{01} \end{bmatrix}$$

Then at step $t = 2$, X_{i1} is broadcast horizontally along the rows of the processor grid, while A_{1j} is broadcast vertically along the columns. Then, X_{i1} is multiplied with A_{1j} on each GPU as follows:

$$\begin{bmatrix} X_{01}A_{10} & X_{01}A_{11} \\ X_{11}A_{10} & X_{11}A_{11} \end{bmatrix}$$

The final output is obtained by summing the two partial outputs:

$$Y = XA = \begin{bmatrix} X_{00}A_{00} + X_{01}A_{10} & X_{00}A_{01} + X_{01}A_{11} \\ X_{10}A_{00} + X_{11}A_{10} & X_{10}A_{01} + X_{11}A_{11} \end{bmatrix}$$

In this way, the activation memory usage per GPU drops by a factor of q^2 , allowing training LLMs on larger number of GPUs. Because activations scale aggressively with sequence length and batch size, this approach is often the only way to train LLMs on large context windows. However, it requires multiple communication steps during every layer's execution loop, and the total GPU number P must be a perfect square.

4.3.3 2.5D Tensor Parallelism. To mitigate the communication overhead of 2D tensor parallelism, 2.5D tensor parallelism [50] adds a third dimension to the processor grid, creating a 3D grid of $P = q \times q \times d$ GPUs. Assuming $q = d = 2$, the matrix operations are distributed as follows:

- The activation matrix (X) is partitioned along its row dimension into d macro-layers. Each layer slice is then divided into a grid of $d \times q$ rows and q columns:

$$X = \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \\ X_{20} & X_{21} \\ X_{30} & X_{31} \end{bmatrix}$$

- The weight matrix (A) is partitioned into a standard 2D grid of $q \times q$ blocks, and then each block is replicated d times across the third dimension:

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$$

Because parameter weight matrices are replicated across the third dimension, the grid splits the global matrix multiplication task into d independent sub-tasks, each of which can be computed in parallel:

$$Y = XA = \text{concat} \left(\begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix} \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}, \begin{bmatrix} X_{20} & X_{21} \\ X_{30} & X_{31} \end{bmatrix} \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \right)$$

This approach reduces network communication costs to $O(1/\sqrt{d})$ relative to traditional 2D tensor parallelism. In an 8-GPU configuration, two groups of 4 GPUs each execute 2D tensor parallelism to compute independent sub-tasks, with the final output obtained by concatenating the group results.

4.3.4 3D Tensor Parallelism. This last approach extends the 2.5D tensor parallelism in order to further reduce communication costs across massive multi-node clusters.

The 3D tensor parallelism [4] technique is based on a processor grid of $P = q \times q \times q$ GPUs. In this setup, both the activation matrix X and the weight matrix A are sharded simultaneously along all three primary matrix dimensions (M, N, K). Consequently, each individual device (i, j, l) in the processor cube stores only a single localized tensor

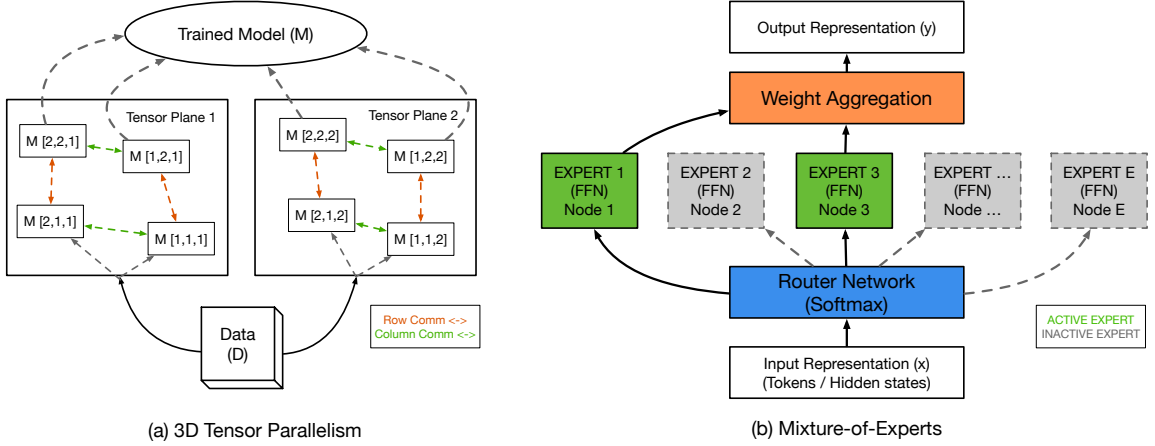


Fig. 6. Advanced Parallelism Techniques.

fragment, X_{ijl} and A_{ijl} . To compute the linear transformation $Y = XA$, the execution is run through a synchronized three-step process:

- **All-Gather phase:** the partial fragments of X and A are collected orthogonally across the rows and columns of the processor grid to form intermediate sub-blocks on each device.
- **Local computation:** each individual GPU multiplies its newly aggregated sub-blocks to compute a localized partial product.
- **Reduce-Scatter phase:** the partial products are summed and scattered across the remaining reduction axis of the cube, leaving each GPU with its correct shard of the final output activation matrix Y .

While 3D tensor parallelism communicates the least data of the four schemes, it introduces substantial implementation complexity due to the coordination of 3D collective operations. Additionally, if the global training batch size is not sufficiently large, this extreme fine-grained sharding can reduce matrix block sizes too much, leading to poor GPU compute utilization. A visualization of the 3D tensor parallelism approach is shown in Figure 6 (a).

4.3.5 3D Parallelism. Due to the massive scale of LLMs and the cardinality of the training data, these tensor sharding techniques are combined with data parallelism and pipeline parallelism to form a 3D parallelism strategy [28]. In particular, 3D parallelism combines three orthogonal dimensions of scaling simultaneously:

- **Data Parallelism dimension:** the training data is partitioned across multiple GPU, allowing each GPU to compute gradients based on its local data shard.
- **Pipeline Parallelism dimension:** the model is partitioned across multiple GPU, allowing each GPU to compute its assigned layer(s) in a pipelined fashion.
- **Tensor Parallelism dimension:** within each layer, the model parameters and activations are further partitioned across multiple GPUs.

By nesting tensor sharding within macro-level pipelines, 3D parallelism allows data, model parameters, and activation states to scale together.

However, scaling dense models using 3D parallelism requires that every token must pass through every parameter in the network. To expand model capacity further without prohibitive computational costs, architectures shifted towards sparse models.

4.3.6 Expert Parallelism and Mixture-of-Experts (MoE). Sparse Mixture-of-Experts (MoE) [15] models decouple total model capacity from the compute required per token. Instead of executing all parameters, the standard dense Feed-Forward Network (FFN) layers are replaced with an MoE layer containing a router and E independent FFN experts. For an input token x , the router computes selection probabilities across all E experts and routes the token to a sparse subset of $k \ll E$:

$$\text{Output } y = \sum_{i \in \text{Top-}k} G(x)_i \cdot E_i(x)$$

where $G(x)_i$ is the routing softmax weight for the i -th expert and $E_i(x)$ is the output of the i -th expert, as shown in Figure 6 (b). To host the experts, Expert Parallelism (EP) shards these E sub-networks across P_{ep} GPU, with each GPU hosting E/P_{ep} experts. This parallelization strategy introduces the following operational characteristics:

- **All-to-All communication primitives:** it consists of a two-phase communication pattern. In the first phase (the dispatch phase), each GPU sends its input tokens to the GPUs that host the corresponding experts. In the second phase (the combine phase), the GPUs hosting the experts send back the computed outputs to the original GPUs.
- **Load imbalance:** the router may not distribute tokens evenly across the experts, leaving some GPUs being overloaded while others remain underutilized. For this reason, MoE systems introduce load-balancing techniques, such as auxiliary loss functions and expert capacity limits, to ensure that the tokens are distributed more evenly across the experts.

4.3.7 5D Parallelism. To scale modern ultra-large sparse models, EP is combined with 3D parallelism and Sequence Parallelism (SP) to form a 5D parallelism strategy. In particular, SP is a technique that shards intermediate activation tensors along the sequence dimensions during non-linear operations (e.g. LayerNorm) that were not previously sharded by 3D parallelism. Together, these strategies form a 5D Parallelism paradigm that coordinates five orthogonal dimensions of scaling simultaneously. By nesting these five layers of parallelism, clusters can scale context lengths, model parameters, and training data together.

However, this approach requires a complex configuration and relies heavily on expensive hardware connections to keep the GPUs in sync. This bottleneck introduced a shift in the research community towards memory-centric optimizations, which aim to reduce the memory footprint of training states without altering the model’s physical structure. Instead of slicing layers and operators across different devices, these techniques preserve standard data-parallel setups but eliminate redundancy by sharding optimizer states, gradients, and parameters across the cluster’s collective memory.

4.4 Memory-Centric Optimizations

In this section, we discuss the memory-centric optimizations that have been developed to address the challenges of training LLMs. To scale data-parallel training past the memory capacity of a single GPU, techniques such as ZeRO (and its variants) and FSDP were introduced.

4.4.1 Zero Redundancy Optimizer (ZeRO). ZeRO [41] addresses the memory bottleneck of data-parallel training by systematically sharding specific training states across data-parallel processes while preserving high computational efficiency and low communication overhead.

In order to optimize memory usage, ZeRO classifies GPU memory consumption into two main categories:

- **Model states:** this category includes the persistent mathematical components of the model, such as parameters, optimizer states, and gradients. In modern training setups, these optimizer states dominate memory consumption, often requiring up to six times the memory of the model parameters themselves.
- **Activation states:** this category includes the intermediate activations generated during the forward pass of the model.

In a standard data-parallel setup, each GPU maintains a complete copy of the model states, leading to significant memory redundancy. For this reason, ZeRO-powered Data Parallelism (ZeRO-DP) was introduced and it is based on the idea of partitioning the model states across data-parallel processes. In particular, ZeRO-DP divides the model states into three distinct stages:

- **ZeRO-Stage 1 (optimizer state partitioning):** this stage targets the largest memory consumer, which is the optimizer state. Instead of replicating the optimizer states across all the GPUs, Stage 1 shards the static optimizer states (e.g. momentum, variance) across data-parallel processes, while model parameters and gradients are still replicated. At the end of the optimization step, each process updates only its own parameter slice, and an All-Gather operation is executed to reconstruct the full updated parameter set on every GPU.
- **ZeRO-Stage 2 (gradient partitioning):** in addition to partitioning the optimizer states, stage 2 also shards the gradients across data-parallel processes. As the backward pass flows through the neural network layers, each GPU computes gradients but only retains the gradient tensors that correspond to its optimizer slice. Then a Reduce-Scatter operation is executed to sum and distribute these gradients directly to their respective owners.
- **ZeRO-Stage 3 (model parameter partitioning):** built upon the previous stages, stage 3 extends the partitioning to include model parameters. During the forward and backward passes, each GPU does not hold the full model parameters. Instead, right before a layer is executed, an All-Gather primitive is triggered to reconstruct the necessary parameters for that layer. After the layer’s computation is completed, the additional parameters are discarded to free up memory. This stage achieves a linear memory reduction relative to the cluster size, while increasing the network data transfer by approximately 50%. There also exist some variants of this stage which handle MoE architectures.

This approach tackles the model states memory consumption, but it does not address the activation states memory consumption. To address this problem, ZeRO-R was introduced, which optimizes the following aspects of the training process:

- **Activation partitioning:** ZeRO-R shards the activation checkpoints across the nodes of the cluster. When a layer triggers its backward pass, the system runs an All-Gather operation to temporarily rebuild the necessary activation for the gradient calculation.
- **Active memory defragmentation:** continuous allocation and deallocation of memory during the training process can lead to fragmentation. This can cause training crashes due to out-of-memory errors even when the total free memory is sufficient, as the memory is split into non-contiguous blocks. ZeRO-R manages this problem by implementing a memory defragmentation strategy that actively reorganizes memory allocations to form larger contiguous blocks of free memory.

However, ZeRO has some limitations, such as the increased communication overhead due to the sharding of model states and the frequent synchronization of gradients, parameters, and optimizer states across multiple GPUs.

4.4.2 ZeRO-Offload. ZeRO-Offload [44] is a ZeRO variant that allows the user to complete the training of LLMs even when hardware resources are limited. Under this layout, forward and backward propagation loops remain on the GPU, while gradient accumulation, optimizer states, and optimizer update steps are offloaded to the CPU. This strategy enables the training of LLMs up to 13 billion parameters on a single GPU.

However, this approach introduces additional communication overhead between the CPU and GPU, which can lead to performance degradation.

4.4.3 ZeRO-Infinity. ZeRO-Infinity [42] implements an integrated storage pipeline that aggregates GPU memory, host CPU RAM, and NVMe storage into a unified memory pool. It was introduced as a solution to address the memory constraints of extreme-scale training and is based on two main system innovations:

- **The Infinity Offload Engine:** this is a routing system that dynamically balances split model states across CPU RAM or NVMe storage, based on the current memory pressure and the access patterns of the model states.
- **Memory-Centric Tiling:** it is an optimization layer that divides tensor operators into smaller tiles.

4.4.4 Fully Sharded Data Parallelism. While the ZeRO ecosystem successfully breaks the memory wall for extreme-scale configurations, its implementation relies on specific library dependencies. To standardize these state-sharding methodologies within ML runtimes, Fully Sharded Data Parallelism (FSDP) was introduced as a native alternative inside the PyTorch ecosystem. FSDP [54] is heavily inspired by the state-sharding techniques of ZeRO-Stage 3, but is implemented as a native PyTorch module. In fact, FSDP uniformly shards the model parameters, gradients, and adaptive optimizer states across data-parallel processes. To maintain a minimal memory footprint without altering the correctness of the training process, FSDP coordinates the parameter movement via precise communication primitives. In particular, the execution loop is divided as follows:

- **Forward pass:** before a specific layer is executed, FSDP blocks the forward pass to launch an intra-node or inter-node All-Gather communication primitive. This operation retrieves the necessary parameter shards from other processes, ensuring that each process has access to the required parameters for its local computation. After the forward pass is completed, the parameters are discarded to free up memory, leaving only the primary sharded parameters.
- **Backward pass:** during the backward pass, the system operates in a similar manner. An All-Gather operation is triggered to rebuild the full parameters required for gradient computation.
- **Gradient sharding and reduction:** instead of using a standard All-Reduce operation, FSDP employs a Reduce-Scatter primitive to shard the gradients across processes. This operation simultaneously sums the calculated gradients across all nodes and scatters the resulting tensors so that each process receives only its corresponding shard of the gradients.

In order to prevent continuous parameter gathering and discarding from introducing performance degradation, FSDP integrates several optimization mechanisms:

- **Nested auto-wrapping policies:** instead of treating an entire LLM as a single block, FSDP relies on auto-wrapping policies that partition the network into independent submodules. This allows All-Gather, compute,

and memory discard operations to be executed sequentially layer by layer, reducing the peak memory usage and improving the overall efficiency of the training process.

- **Prefetching:** FSDP implements forward and backward prefetching schedules, in order to overlap communication and computation. For instance, through *BackwardPrefetch*, FSDP anticipates the upcoming step and asynchronously gathers the parameters for the next layer while the current layer is still computing its gradients. This overlap between communication and computation helps hide the latency of parameter gathering, thereby improving training throughput.
- **Mixed precision and memory management:** FSDP supports mixed precision training. For instance, parameters can be stored in standard 32-bit floating point format, but communicated and computed using lower precision Float16. Communicating tensors in a lower-precision format can reduce the amount of data transferred over the network. Moreover, FSDP supports optional offloading to CPU when GPU memory is insufficient, allowing for a more flexible memory management strategy.

FSDP enables a more efficient and scalable training process for LLMs, while maintaining training correctness. However, FSDP has structural trade-offs that can affect scaling performance across modern clusters, such as its heavy dependency on network bandwidth and the complexity of its wrapping policies. Since parameters must be gathered and discarded continuously across both forward and backward loops, FSDP’s scaling is highly sensitive to cluster interconnect speeds. On networks lacking intra-node NVLink or inter-node InfiniBand, the communication overhead can become a significant bottleneck, limiting the overall training throughput. Additionally, the complexity of the auto-wrapping policies can introduce challenges in terms of model architecture compatibility and ease of use, requiring careful consideration when designing and implementing LLMs with FSDP.

4.5 Frameworks for Distributed Training of LLMs

In this section, we provide an overview of the most popular frameworks for distributed training of LLMs.

4.5.1 DeepSpeed. DeepSpeed [11] was developed by Microsoft and it is an open-source optimization library that provides a suite of tools and techniques specifically for training LLMs. This framework allows the use of each of the ZeRO stages, but also supports full 3D parallelism. Moreover, it includes memory scaling variants such as ZeRO-Offload and ZeRO-Infinity to exploit host CPU and NVMe memory tiers.

Configuring DeepSpeed requires complex hyperparameter tuning and its performance is highly dependent on the underlying hardware and network infrastructure.

4.5.2 Megatron-LM. This framework [36] was developed by NVIDIA, with the goal to maximize hardware throughput when training dense autoregressive models. Megatron-LM provides a highly optimized implementation of 1D tensor parallelism and inter-layer pipeline parallelism. Later versions introduced sequence and expert parallelism alongside distributed optimizers and shared data parallelism. To achieve high GPU utilization, it implements a custom CUDA kernel and strict communication-computation overlap in order to hide synchronization latency.

Even in this case, setting up Megatron-LM requires specialized system expertise in CUDA configuration and distributed training paradigms. In addition, the framework is primarily designed for large-scale pre-training pipelines rather than downstream fine-tuning tasks.

4.5.3 Colossal-AI. While previous frameworks focus on optimizing specific aspects of LLM training, Colossal-AI [30] aims to provide a unified framework that integrates multiple parallelism strategies. It supports 1D, 2D, 2.5D, and

3D tensor parallelism, as well as pipeline and data parallelism. In addition, Colossal-AI includes advanced memory optimization techniques (ZeRO) but also a large set of optimizations such as automatic mixed precision, gradient checkpointing and host CPU offloading. It is open-source and well-documented, making it accessible to a wider range of researchers and practitioners.

Some of the optimizations in Colossal-AI are explicitly designed for specific NVIDIA microarchitectures, which can limit the portability of the framework across different hardware platforms.

4.5.4 *Alpa*. Alpa [55] is a framework that automates 3D parallelism without requiring manual configuration. It uses a compiler-based approach to automatically partition the model and data across multiple GPUs, optimizing both memory usage and communication efficiency. It structures the search space into two levels: intra-operator (tensor and data parallelism via integer linear programming) and inter-operator (pipeline parallelism via dynamic programming).

A main limitation of this approach is its poor compatibility with the PyTorch ecosystem, which has become the de-facto standard for deep learning research and deployment.

4.5.5 *Megatron-DeepSpeed*. This framework [12] is a hybrid of the two aforementioned frameworks, combining the strengths of Megatron-LM and DeepSpeed. It integrates Megatron-LM’s highly optimized tensor parallelism and pipeline parallelism with DeepSpeed’s ZeRO memory optimization techniques. It also provides highly optimized support for the NCCL communication library, resulting in low synchronization overhead and high throughput.

However, the complexity of managing this framework is significant, as it requires expertise in both Megatron-LM and DeepSpeed configurations. Additionally, the system is dependent on CUDA and NCCL, which can limit its portability to non-NVIDIA hardware platforms, and it imposes strict physical network requirements, such as high-speed inter-device links and low-latency interconnects.

4.5.6 *DeepSpeed-MoE*. DeepSpeed-MoE [40] extends DeepSpeed to optimize sparse MoE architectures. It combines EP with ZeRO and TP, introducing optimized All-to-All communication patterns and structural variants (such as Residual MoE) to reduce parameter and communication overhead.

4.5.7 *SmartMoE*. SmartMoE [53] is a framework tailored for MoE models, that overcomes static compilers by accounting for dynamic token routing. It pairs offline search space construction with online search algorithm to improve load balancing and overall training efficiency.

4.5.8 *MindSpeed*. MindSpeed [3] is a framework developed by Huawei and has been designed to optimize the training of LLMs on Ascend AI processors. It integrates data parallelism, model parallelism, and pipeline parallelism, along with memory optimization techniques such as ZeRO.

4.5.9 *Hugging Face Accelerate*. Hugging Face Accelerate [17] is a lightweight library developed by the Hugging Face team, designed to tackle the challenges of distributed training in a user-friendly manner. It provides popular PyTorch strategies such as DDP, FSDP, and DeepSpeed, and allows users to easily configure the system without dealing with complex initialization and synchronization logic.

However, it has some limitations related to model parallelism and tensor parallelism. Moreover, it is not well-suited for parallelizing the optimization of LLMs since it may struggle to optimize the communication patterns and memory usage for large-scale models.

4.5.10 Summary. In this section, we have discussed the advanced techniques and frameworks that have been developed to address the challenges of training LLMs. In Table 2, we provide a summary of the most popular frameworks for distributed training of LLMs, highlighting their parallelism strategies, communication and synchronization methods, as well as their pros and cons.

Table 2. Comparison of Distributed Frameworks for LLM Training.

Framework	Parallelism Strategy	Communication and Synchronization	Pros	Cons
DeepSpeed [11]	3D (Data, Pipeline, Tensor), ZeRO	All-Reduce, All-Gather (<i>Synchronous BSP</i>)	High memory efficiency, multi-tier offloading	Complex hyperparameter tuning, hardware sensitivity
Megatron-LM [36]	Tensor, Sequence, Pipeline, Expert, FSDP/DDP	Custom CUDA, All-Reduce (<i>Synchronous BSP</i>)	High throughput, high GPU utilization	High CUDA expertise, limited fine-tuning focus
Colossal-AI [30]	Hybrid (1D/2D/2.5D/3D Tensor, Pipeline, ZeRO)	All-Reduce, CPU Offload (<i>Synchronous BSP</i>)	Unified parallelism ecosystem, well documented	Limited non-NVIDIA portability, hardware lock-in
Alpa [55]	Automated 3D (Intra/Inter-operator)	All-Reduce, P2P (<i>Synchronous BSP</i>)	Automated compiler search, minimal manual config	Poor PyTorch compatibility, compiler overhead
Megatron-DeepSpeed [12]	Hybrid (Tensor, Pipeline, ZeRO)	All-Reduce, NCCL (<i>Synchronous BSP</i>)	High throughput, strong memory scaling	High management complexity, steep interconnect demands
DeepSpeed-MoE [40]	Expert, ZeRO, Tensor	All-to-All, All-Reduce (<i>Synchronous BSP</i>)	Optimized MoE communication, flexible sparse scaling	High network sensitivity, communication overhead
SmartMoE [53]	Expert, Dynamic Routing	All-to-All (<i>Synchronous BSP</i>)	Dynamic load balancing	Search space overhead, complex runtime setup
MindSpeed [3]	Hybrid (Data, Model, Pipeline, ZeRO)	Custom HCCL Primitives (<i>Synchronous BSP</i>)	Ascend processor optimization, high hardware speed	Platform dependency
Hugging Face Accelerate [17]	Data, Hybrid (DDP, FSDP, DeepSpeed)	High-Level NCCL Wrappers (<i>Synchronous BSP</i>)	User-friendly interface, simple configuration	Limited model parallelism, inefficient for massive LLMs

5 Conclusions

In this work, we have shown the critical importance of distributed machine learning in addressing the growing computational and memory demands of modern deep learning models, particularly large language models. By exploring foundational partitioning strategies, communication topologies, and distributed architectures, it is evident that the only way to enable unprecedented model growth and training efficiency is through the adoption of distributed systems.

Although memory-centric optimizations and advanced parallelization strategies have significantly mitigated the parameter memory wall, several open research issues remain. Future work must prioritize the development of more efficient communication protocols, the design of adaptive and fault-tolerant distributed architectures, and the exploration of novel optimization techniques that can further enhance convergence rates and model performance. Furthermore, extending distributed training to edge environments introduces significant hardware heterogeneity, with devices differing significantly in computational power, memory capacity, and energy constraints. This requires dynamic scheduling, straggler mitigation, and energy-aware optimization. Alongside these technical demands, embedding carbon- and water-awareness into workload placement and cluster management will be important to mitigate the

growing environmental footprint of large-scale LLM training. Finally, lowering the barrier to entry through cost-efficient performance training, alongside ensuring robust data privacy and security, will be crucial for the global widespread and accessibility of distributed DL.

References

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283. <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>
- [2] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. 2014. A Reliable Effective Terascale Linear Learning System. *J. Mach. Learn. Res.* 15, 1 (2014), 1111–1133. doi:10.5555/2627435.2638571
- [3] Ascend Contributors. 2023. *MindSpeed-LLM: Distributed Large Language Model Acceleration Framework for Ascend NPUs*. Open Source Software. Huawei Ascend Community. <https://gitee.com/ascend/MindSpeed-LLM>
- [4] Zhengda Bian, Qifan Xu, Boxiang Wang, and Yang You. 2021. Maximizing Parallelism in Distributed Training for Huge Neural Networks. *CoRR* abs/2105.14450 (2021). arXiv:2105.14450 <https://arxiv.org/abs/2105.14450>
- [5] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Praseoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. *CoRR* abs/1604.07316 (2016). arXiv:1604.07316 <http://arxiv.org/abs/1604.07316>
- [6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). <https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfc4967418bf8ac142f64a-Abstract.html>
- [7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 <http://arxiv.org/abs/1512.01274>
- [8] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric P. Xing. 2013. Solving the Straggler Problem with Bounded Staleness. In *14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013*, Petros Maniatis (Ed.). USENIX Association. <https://www.usenix.org/conference/hotos13/session/cipar>
- [9] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. 2016. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-Specialized Parameter Server. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues (Eds.). ACM, 4:1–4:16. doi:10.1145/2901318.2901323
- [10] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger (Eds.). 1232–1240. <https://proceedings.neurips.cc/paper/2012/hash/6aca97005c68f1206823815f66102863-Abstract.html>
- [11] DeepSpeed Team. 2020. DeepSpeed: A Deep Learning Optimization Library. <https://github.com/deepspeedai/DeepSpeed>.
- [12] DeepSpeed Team. 2021. *Megatron-DeepSpeed: Ongoing Research Training Transformer Language Models at Scale*. Open Source Software. Microsoft DeepSpeed. <https://github.com/deepspeedai/Megatron-DeepSpeed>
- [13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. doi:10.18653/V1/N19-1423
- [14] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, Jaejin Lee and Erez Petrank (Eds.). ACM, 431–445. doi:10.1145/3437801.3441593
- [15] Wensheng Gan, Zhenyao Ning, Zhenlian Qi, and Philip S. Yu. 2026. Mixture of Experts (MoE): A Big Data Perspective. *Inf. Fusion* 127 (2026), 103664. doi:10.1016/J.INFFUS.2025.103664

- [16] Andrew Gibiansky. 2017. *Bringing HPC Techniques to Deep Learning*. Technical Report. Baidu Research. <https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/>
- [17] Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, and Sourab Mangrulkar. 2022. *Accelerate: Training and Inference at Scale Made Simple, Efficient and Adaptable*. Open Source Software. Hugging Face. <https://github.com/huggingface/accelerate>
- [18] István Hegedűs, Gábor Danner, and Márk Jelasity. 2019. Gossip Learning as a Decentralized Alternative to Federated Learning. In *Distributed Applications and Interoperable Systems - 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11534)*, José Pereira and Laura Ricci (Eds.). Springer, 74–90. doi:10.1007/978-3-030-22496-7_5
- [19] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory F. Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. 2017. Deep Learning Scaling is Predictable, Empirically. *CoRR* abs/1712.00409 (2017). arXiv:1712.00409 <http://arxiv.org/abs/1712.00409>
- [20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, Hyukjoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 103–112. <https://proceedings.neurips.cc/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html>
- [21] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, and Kurt Keutzer. 2016. FireCaffe: Near-Linear Acceleration of Deep Neural Network Training on Compute Clusters. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2592–2600. doi:10.1109/CVPR.2016.284
- [22] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafir (Eds.). USENIX Association, 947–960. <https://www.usenix.org/conference/atc19/presentation/jeon>
- [23] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 2279–2288. <http://proceedings.mlr.press/v80/jia18a.html>
- [24] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the Second Conference on Machine Learning and Systems, SysML 2019, Stanford, CA, USA, March 31 - April 2, 2019*, Amee Talwalkar, Virginia Smith, and Matei Zaharia (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2019/hash/b422680f3db0986ddd7f8f126baaf0fa-Abstract.html
- [25] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *CoRR* abs/2001.08361 (2020). arXiv:2001.08361 <https://arxiv.org/abs/2001.08361>
- [26] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). <http://arxiv.org/abs/1412.6980>
- [27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger (Eds.). 1106–1114. <https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html>
- [28] Zhiqian Lai, Shengwei Li, Xudong Tang, Keshi Ge, Weijie Liu, Yabo Duan, Linbo Qiao, and Dongsheng Li. 2023. Merak: An Efficient Distributed DNN Training Framework With Automated 3D Parallelism for Giant Foundation Models. *IEEE Trans. Parallel Distributed Syst.* 34, 5 (2023), 1466–1478. doi:10.1109/TPDS.2023.3247001
- [29] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 583–598. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu
- [30] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. 2023. Colossal-AI: A Unified Deep Learning System For Large-Scale Parallel Training. In *Proceedings of the 52nd International Conference on Parallel Processing, ICPP 2023, Salt Lake City, UT, USA, August 7-10, 2023*. ACM, 766–775. doi:10.1145/3605573.3605613
- [31] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (2020), 3005–3018. doi:10.14778/3415478.3415530
- [32] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. 2017. Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5330–5340. <https://proceedings.neurips.cc/paper/2017/hash/f75526659f31040afeb61cb7133e4e6d-Abstract.html>
- [33] Ruben Mayer, Christian Mayer, and Larissa Laich. 2017. The TensorFlow Partitioning and Scheduling Problem: It's the Critical Path! *CoRR* abs/1711.01912 (2017). arXiv:1711.01912 <http://arxiv.org/abs/1711.01912>

- [34] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA (Proceedings of Machine Learning Research, Vol. 54)*, Aarti Singh and Xiaojin (Jerry) Zhu (Eds.). PMLR, 1273–1282. <http://proceedings.mlr.press/v54/mcmahan17a.html>
- [35] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 1–15. doi:10.1145/3341301.3359646
- [36] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prithvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters using Megatron-LM. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 58. doi:10.1145/3458817.3476209
- [37] NVIDIA Corporation. 2026. NVIDIA Collective Communications Library (NCCL). <https://developer.nvidia.com/nccl>. Accessed: 2026-07-04.
- [38] NVIDIA Corporation. 2026. NVIDIA NVLink and NVLink Switch. <https://www.nvidia.com/en-us/data-center/nvlink/>. Accessed: 2026-07-04.
- [39] Róbert Ormándi, István Hegedűs, and Márk Jelasity. 2013. *Gossip Learning Framework (GoLF): Benchmarking Framework for Distributed P2P Machine Learning*. Open Source Software. University of Szeged. <https://github.com/ormandi/Gossip-Learning-Framework>
- [40] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato (Eds.). PMLR, 18332–18346. <https://proceedings.mlr.press/v162/rajbhandari22a.html>
- [41] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 20. doi:10.1109/SC41405.2020.00024
- [42] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 59. doi:10.1145/3458817.3476205
- [43] Benjamin Recht, Christopher Ré, Stephen J. Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando C. N. Pereira, and Kilian Q. Weinberger (Eds.), 693–701. <https://proceedings.neurips.cc/paper/2011/hash/218a0aef1d1a4be65601cc6ddc1520e-Abstract.html>
- [44] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 551–564. <https://www.usenix.org/conference/atc21/presentation/ren-jie>
- [45] Lorenzo Rosasco. 2018. *Introductory Machine Learning Notes*. Lecture Notes. University of Genoa. <https://lcs.unige.it/courses/ml/1718/MLNotes.pdf>
- [46] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). <http://arxiv.org/abs/1802.05799>
- [47] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA*, Rene L. Cruz and George Varghese (Eds.). ACM, 149–160. doi:10.1145/383059.383071
- [48] Kimi Team. 2026. Kimi K3: Open Frontier Intelligence. *CoRR* abs/2607.24653 (2026). [arXiv:2607.24653](https://arxiv.org/abs/2607.24653) doi:10.48550/ARXIV.2607.24653
- [49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.), 5998–6008. <https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html>
- [50] Boxiang Wang, Qifan Xu, Zhengda Bian, and Yang You. 2022. Tesseract: Parallelize the Tensor Parallelism Efficiently. In *Proceedings of the 51st International Conference on Parallel Processing, ICPP 2022, Bordeaux, France, 29 August 2022 - 1 September 2022*. ACM, 12:1–12:11. doi:10.1145/3545008.3545087
- [51] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2015. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, Shahram Ghandeharizadeh, Sumita Barahmand, Magdalena Balazinska, and Michael J. Freedman (Eds.). ACM, 381–394. doi:10.1145/2806777.2806778
- [52] Qifan Xu and Yang You. 2023. An Efficient 2D Method for Training Super-Large Deep Learning Models. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023, St. Petersburg, FL, USA, May 15-19, 2023*. IEEE, 222–232. doi:10.1109/IPDPS54959.2023.00031
- [53] Mingshu Zhai, Jiaao He, Zixuan Ma, Zan Zong, Runqing Zhang, and Jidong Zhai. 2023. SmartMoE: Efficiently Training Sparsely-Activated Models through Combining Offline and Online Parallelization. In *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, Julia Lawall and Dan Williams (Eds.). USENIX Association, 961–975. <https://www.usenix.org/conference/atc23/presentation/zhai>

- [54] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (2023), 3848–3860. doi:10.14778/3611540.3611569
- [55] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 559–578. <https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin>

A Parameter Server in practice

As we discussed in Section 3.1, the Parameter Server uses a centralized architecture to manage the model parameters and coordinate the training process across multiple worker nodes. This appendix section will provide a practical example on how the computation and communication flow occurs in a PS-based distributed training system, using a general machine learning algorithm as a reference.

Algorithm 1 Distributed Subgradient Descent

Task Scheduler:

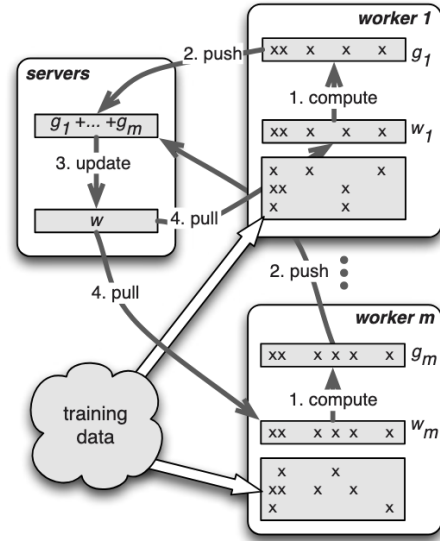
- 1: issue `LOADDATA()` to all workers
- 2: **for** iteration $t = 0, \dots, T$ **do**
- 3: issue `WORKERITERATE( $t$ )` to all workers
- 4: **end for**

Worker $r = 1, \dots, m$:

- 5: **function** `LOADDATA`
- 6: load a part of training data $\{y_{i_k}, x_{i_k}\}_{k=1}^{n_r}$
- 7: pull the working set $w_r^{(0)}$ from servers
- 8: **end function**
- 9: **function** `WORKERITERATE( $t$ )`
- 10: gradient $g_r^{(t)} \leftarrow \sum_{k=1}^{n_r} \partial l(x_{i_k}, y_{i_k}, w_r^{(t)})$
- 11: push $g_r^{(t)}$ to servers
- 12: pull $w_r^{(t+1)}$ from servers
- 13: **end function**

Servers:

- 14: **function** `SERVERITERATE( $t$ )`
 - 15: aggregate $g^{(t)} \leftarrow \sum_{r=1}^m g_r^{(t)}$
 - 16: $w^{(t+1)} \leftarrow w^{(t)} - \eta (g^{(t)} + \partial \Omega(w^{(t)}))$
 - 17: **end function**
-



In this toy example, the task scheduler orchestrates the training process by issuing commands to the worker nodes. Each worker node loads its portion of the training data and pulls the initial working set of model parameters from the servers. During each iteration, workers compute gradients based on their local data and push these gradients to the servers. The servers aggregate the gradients, update the global model parameters, and send the updated parameters back to the workers for the next iteration. This process continues until convergence or until a predefined number of iterations is reached.

B Ring All-Reduce in practice

As described in Section 3.2, the Ring All-Reduce architecture utilizes a decentralized, peer-to-peer topology to aggregate gradients without relying on a central server. This appendix section provides a practical example of the computation and communication flow in a Ring All-Reduce distributed training system.

Algorithm 3 Gossip Learning Distributed SGD

Worker $r \in \{0, \dots, N - 1\}$ (**Asynchronous Loop**):

- 1: **function** WORKERITERATE(t)
- 2: Compute local gradient $g_r^{(t)} \leftarrow \sum_{k=1}^{n_r} \partial l(x_{i_k}, y_{i_k}, w_r^{(t)})$
- 3: Update local parameters $w_r^{(t+\frac{1}{2})} \leftarrow w_r^{(t)} - \eta g_r^{(t)}$

▷ *Random Peer Sampling (RPS)*

- 4: Select a random subset of peers $P \subset \{0, \dots, N - 1\} \setminus \{r\}$

▷ *Push & Pull Phase*

- 5: **for** each peer $p \in P$ **do**
- 6: **Push:** Send local parameters $w_r^{(t+\frac{1}{2})}$ to peer p
- 7: **Pull:** Request parameters $w_p^{(t+\frac{1}{2})}$ from peer p
- 8: **end for**

▷ *Aggregation Phase*

- 9: Receive parameters from responsive peers $P_{resp} \subseteq P$
- 10: **Aggregate:** $w_r^{(t+1)} \leftarrow \frac{1}{|P_{resp}|+1} \left(w_r^{(t+\frac{1}{2})} + \sum_{p \in P_{resp}} w_p^{(t+\frac{1}{2})} \right)$
- 11: **end function**

In this architecture, every node operates as an independent entity, maintaining its own local copy of the model parameters and training strictly on its local data. Unlike the synchronized steps of Ring All-Reduce, Gossip Learning relies on an asynchronous execution flow. Periodically, each node utilizes a Random Peer Sampling (RPS) strategy to select a random subset of peers from the network. The communication is driven by three primary primitives: nodes *Push* their local model parameters to the selected peers, *Pull* the remote parameters from those peers, and *Aggregate* the received data (typically by averaging) with their own local model. This design entirely eliminates the straggler problem and is immune to single points of failure, naturally handling node churn as peers can join or leave the network without halting the overall training process.