Distributed Machine Learning Systems : A survey

Sizhe Wang [*] **Pinyi Zhao** [*] **Henghui Bao** [*]
University of Southern California
`{sizhewan, pinyizha, henghuib}@usc.edu`

Abstract

The growing computational cost of training machine learning models has made distributed processing essential in modern artificial intelligence research and applications. This proposal explores the categorization of distributed machine learning (ML) frameworks based on communication topologies: iterative MapReduce/AllReduce, parameter servers, and data flow-based systems. We provide a comparative analysis of representative frameworks, such as Spark MLlib, TensorFlow, and Microsoft Multiverso, emphasizing their design principles, strengths, and challenges. The findings aim to guide researchers and practitioners in selecting suitable frameworks for various ML tasks.

1 Introduction

The exponential growth in machine learning (ML) data and model complexity has necessitated distributed computing to meet scalability requirements. Distributed ML systems leverage parallelism to handle the computational and data-intensive nature of modern ML. Communication topology, a critical design aspect, dictates how information is shared and synchronized across distributed nodes, influencing performance, scalability, and fault tolerance. This paper categorizes ML frameworks based on their communication topologies and evaluates their impact on distributed ML.

2 Mainstream frameworks in Distributed ML

2.1 Iterative MapReduce/AllReduce Topology

Frameworks using the iterative MapReduce or AllReduce topology rely on aggregating intermediate results across distributed nodes. Popular examples include Apache Spark(Zaharia et al., 2016) and MPI-based systems. Understanding the principles of these frameworks requires delving into their core design, strengths, challenges, and specific implementations, such as Spark MLlib's iterative processing for machine learning tasks.

- **Design:** Iterative MapReduce frameworks like Spark are built on the principle of dividing tasks into smaller, parallelized units of computation. These units are executed on distributed nodes, and their results are aggregated in successive stages. Spark, as a distributed computing platform, operates in an environment where nodes do not share memory, and all data exchanges occur over a network. Typically, Spark is deployed on a cluster of inexpensive compute nodes, such as commodity servers or virtual containers, distinguishing it from high-performance architectures like GPU-accelerated systems or shared-memory servers.

  The core execution model involves splitting a program into a Directed Acyclic Graph (DAG) of tasks, as shown in Figure 2. Each DAG identifies parallelizable tasks (e.g., `map`, `filter`) and synchronization points (e.g., `groupByKey`, `join`), where data shuffling and reducing are necessary. Tasks that require global data synchronization, such as shuffling, define the boundaries of execution stages, introducing bottlenecks.

  For iterative machine learning algorithms, Spark MLlib leverages a mini-batch gradient descent approach. This involves broadcasting model parameters to each node, performing local computations to calculate gradients, and aggregating the results via tree-based reduction (e.g., `treeAggregate`). These iterative stages allow distributed systems to refine models across multiple updates.

- **Strengths:** Iterative MapReduce frameworks excel in tasks involving batch processing

[*] Equal Constributions.

Topology	Strengths	Weaknesses	Frameworks	Use Cases
MapReduce/AllReduce	Simple aggregation, fault tolerance	Communication overhead, static workflows	Spark MLlib, MPI	Batch learning, iterative tasks
Parameter Server	Scalable, flexible update mechanisms	Synchronization bottlenecks, fault recovery	Petuum, Multiverso	Deep learning, large-scale ML
Data Flow	Expressiveness, hardware acceleration	Scheduling overhead, debugging complexity	TensorFlow, PyTorch	Dynamic tasks, complex models

Table 1: Comparative Analysis of Communication Topologies in Distributed ML Systems

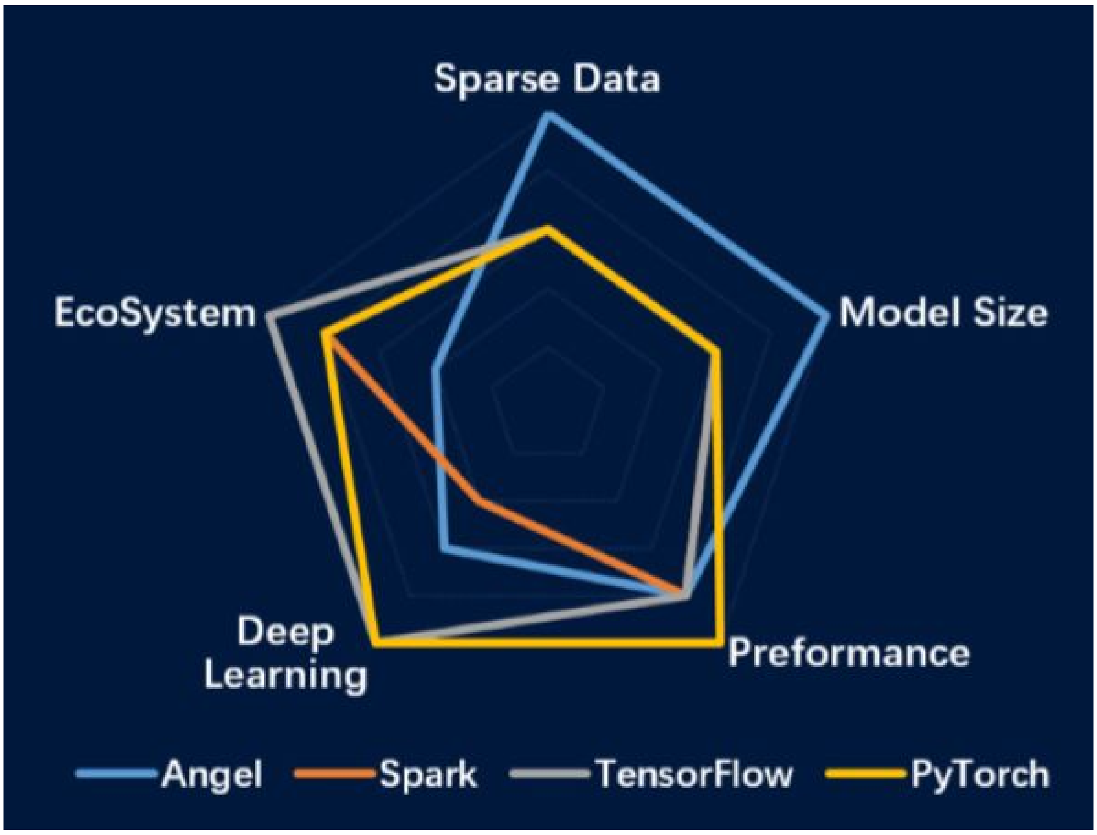

Figure 1: **Comparison of different frameworks** Angel represents the Parameter Server, Spark exemplifies MapReduce, and TensorFlow and PyTorch embody the Data Flow.

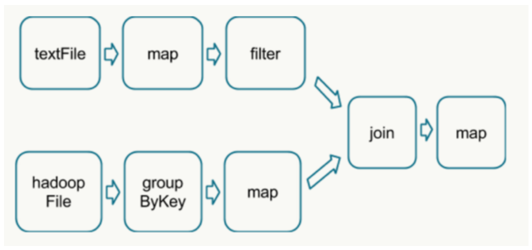

Figure 2: **DAG**

and iterative computations. Their design inherently supports fault tolerance by checkpointing intermediate states, enabling recovery from node failures. Spark, for instance, uses Resilient Distributed Datasets (RDDs) as immutable data structures that allow efficient parallel computations while ensuring resilience. Integration with distributed storage systems, such as Hadoop Distributed File System (HDFS), enables seamless handling of large datasets.

Another key strength lies in the simplicity and scalability of Spark's data-parallel approach. By decoupling computation and communication, the framework scales to massive datasets without requiring sophisticated parameter-sharing mechanisms. Machine learning algorithms, such as K-Means clustering, benefit from Spark's ability to cache intermediate results in memory, significantly reducing I/O overhead and accelerating iterative computations.

- **Challenges:** Despite its strengths, the iterative MapReduce topology faces several challenges. Communication overhead is a significant limitation, especially during synchronization-intensive operations like shuffling and reducing. High-frequency synchronization introduces latency, which becomes a bottleneck in networks with limited bandwidth.

Additionally, Spark's reliance on global parameter broadcasting and blocking synchronization hinders its performance for complex models with large parameter spaces. In mini-batch gradient descent, the slowest node determines the pace of progress, resulting in inefficiencies in heterogeneous environments. The static nature of Spark's architecture also limits support for dynamic workflows and real-time processing. These challenges necessitate optimizations in communication protocols and adaptive execution strategies to improve efficiency.

Furthermore, Spark MLlib has notable limitations in handling complex neural networks. Its standard library primarily supports basic architectures, such as logistic regression and simple multi-layer perceptrons, with limited flexibility in activation functions or hyperparameter tuning. As a result, Spark struggles with deep learning workloads, often requiring external platforms like TensorFlow or PyTorch for more advanced tasks.

- **Example:** Spark MLlib provides an illustrative example of iterative processing for machine learning. The platform divides datasets into partitions distributed across worker nodes. For mini-batch gradient descent, Spark performs three key steps in each iteration:

  1. Broadcasting the current model parameters to all partitions.
  2. Computing local gradients on mini-batches within each partition.
  3. Aggregating gradients using `treeAggregate`, a hierarchical reduction process that minimizes communication overhead by summarizing data in a tree-like structure.

  The iterative training process terminates when the model converges or the maximum number of iterations is reached. While this approach is straightforward and effective for certain models, it reveals inefficiencies when applied to large-scale deep learning tasks due to the high cost of parameter broadcasting and synchronization.

  For example, the `join` operation in a Spark DAG requires matching all data across partitions, resulting in a shuffle stage that consumes significant resources. Conversely, operations like `map` and `filter` operate independently on each partition, allowing for high parallelism without inter-node communication. The balance between parallel computation and synchronization defines Spark's performance.

- **Limitations:** While Spark MLlib's data-parallel approach is simple and intuitive, it comes with several drawbacks:

  – The global broadcasting of model parameters in each iteration is resource-intensive, especially for large models, leading to bandwidth bottlenecks and memory overhead on worker nodes.
  – The blocking nature of gradient aggregation causes delays when certain nodes take longer to compute, due to data skew or hardware heterogeneity.
  – Limited support for advanced neural network architectures, such as RNNs or LSTMs, and hyperparameter customization restricts its applicability to deep learning tasks.

  These limitations highlight the need for alternative platforms, such as Parameter Servers for distributed training or TensorFlow/PyTorch for flexible and efficient deep learning.

2.2 Parameter Server for Distributed Training

Frameworks employing the Parameter Server (PS) architecture have become foundational for distributed machine learning, integrated into mainstream frameworks like TensorFlow and MXNet. These systems offer a scalable solution to parallelize gradient descent computations for model training.

- **Design:** The Distributed Parameter Server(Li et al., 2013) framework employs a master-worker architecture consisting of two primary node types: **server nodes** and **worker nodes**. Server nodes store model parameters, aggregate gradients from workers, and update the global model. Worker nodes handle subsets of training data, compute local gradients, and communicate with server nodes through *push* and *pull* operations.

  As shown in Figure 3, the PS system divides its functionality into two major groups: **server**

groups and **worker groups**. Each server node within a server group manages a portion of the model parameters, while server managers coordinate resource allocation. Worker nodes in each worker group compute gradients independently on local data and interact only with the server nodes, not with other workers. This design minimizes inter-worker communication overhead and ensures scalability for large-scale training tasks.

The training process using the PS architecture follows an iterative workflow

1. Each worker node loads a partition of the training data.
2. Worker nodes *pull* the latest model parameters from the server nodes.
3. Workers compute gradients locally on their data partitions.
4. Gradients are *pushed* to server nodes for aggregation and parameter updates.
5. The process repeats until the model converges or a specified number of iterations is completed.

PS supports both synchronous and asynchronous gradient descent. In synchronous training, all workers must complete their gradient computation before parameter updates proceed, ensuring consistency but introducing potential delays. In contrast, asynchronous training allows workers to proceed with computations even if some gradients are outdated, significantly improving efficiency but trading off strict consistency.

- **Strengths:** Parameter Server frameworks excel in handling large-scale distributed training due to their high scalability and adaptability. Key strengths include:
  - **Flexibility:** PS supports a wide range of training paradigms, including synchronous and asynchronous gradient descent, making it suitable for diverse workloads.
  - **Efficient resource utilization:** By partitioning model parameters across server nodes and distributing data among worker nodes, PS minimizes communication bottlenecks and scales effectively with the number of nodes.
  - **Integration:** PS is integrated into popular deep learning frameworks like TensorFlow and MXNet, providing users with seamless support for distributed training.
- **Challenges:** Despite its advantages, the Parameter Server architecture faces challenges in achieving an optimal balance between consistency and parallel efficiency:
  - **Communication overhead:** Frequent push and pull operations between workers and servers can strain network bandwidth, especially for models with large parameter spaces.
  - **Stale gradients:** Asynchronous training may use outdated gradients, leading to slower convergence or suboptimal model performance.
  - **Fault tolerance:** While server nodes are often designed to be fault-tolerant, failures in worker nodes can disrupt training, particularly in synchronous setups.
- **Example:** To illustrate, consider the training of a machine learning model with a loss function $F(w)$ that includes a regularization term. The goal is to minimize $F(w)$ using distributed gradient descent:

  1. Workers compute local gradients $\nabla F_i(w)$ for their data partitions.
  2. Gradients are *pushed* to server nodes, which aggregate them to update global parameters w.
  3. Updated parameters are *pulled* by workers for the next iteration.

  This process enables efficient parallel computation of gradients while leveraging the PS architecture's scalability. For example, TensorFlow's distributed training uses PS to divide workloads across clusters, with workers executing forward and backward passes and servers managing gradient aggregation.
- **Limitations:** The Parameter Server architecture has inherent limitations:
  - **Consistency-efficiency tradeoff:** Synchronous training ensures consistent updates but can lead to idle workers waiting for slower nodes. Asynchronous training mitigates this but sacrifices consistency.

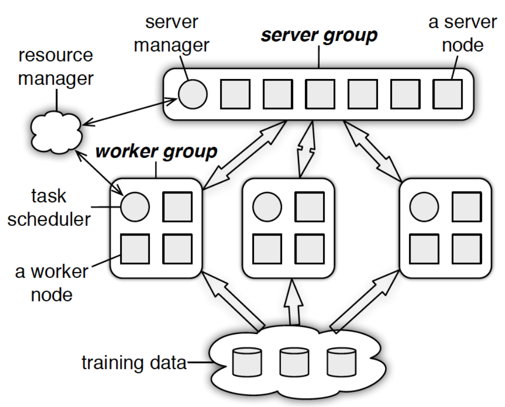

Figure 3: **Parameter Server Architecture**

- **Scalability bottlenecks:** Communication delays increase as the number of workers or parameter size grows, requiring advanced techniques like hierarchical aggregation to improve efficiency.
- **Applicability:** While PS is highly effective for traditional machine learning models, it faces challenges in training complex neural networks, where specialized frameworks like Horovod often outperform it.

• **Optimization Strategies:** Addressing these limitations involves strategies such as:

  - **Gradient compression:** Reducing the size of gradients sent during push operations to lower bandwidth requirements.
  - **Hierarchical aggregation:** Using tree-based reduction for gradient updates to minimize communication overhead.
  - **Hybrid approaches:** Combining PS with other distributed training paradigms, such as AllReduce, to leverage the strengths of both methods.

2.3 Data Flow Topology

Data flow-based systems represent computations as directed acyclic graphs (DAGs) of operations. Google's TensorFlow is the most prominent example in this category.

• **Design:** Data flow-based systems conceptualize machine learning computations as DAGs, where each node corresponds to an operation (e.g., matrix multiplication, activation functions) and edges denote the data dependencies between these operations. This structure enables parallel execution by scheduling independent operations concurrently while adhering to the constraints of the data flow. The dynamic construction of computation graphs allows systems to adapt to different workloads and model structures efficiently. Furthermore, runtime optimization strategies, such as operation fusion and memory reuse, enhance the performance of DAG-based computations, making them suitable for large-scale and heterogeneous workloads.

• **Strengths:** One of the primary advantages of the data flow topology is its flexibility in representing and executing complex machine learning models, ranging from deep neural networks to reinforcement learning algorithms. The abstraction of computations as graphs simplifies integration with various hardware accelerators like GPUs, TPUs, and FPGAs, enabling significant performance boosts. Additionally, this topology supports dynamic task scheduling, which is essential for workloads with unpredictable data dependencies. The modularity of DAGs allows researchers to experiment with new architectures by seamlessly adding, removing, or modifying components without affecting the overall system integrity.

• **Challenges:** Despite its strengths, the data flow topology introduces notable challenges. The overhead associated with scheduling DAG operations, especially for large and intricate models, can lead to increased latency. Debugging and profiling computations in DAG-based systems can be complex, as the indirect representation of operations often obscures the root causes of errors. Furthermore, optimizing memory usage and execution order within the DAG requires sophisticated algorithms, which can introduce additional computational overhead and implementation complexity.

• **Example:** TensorFlow exemplifies the data flow topology by providing a comprehensive framework for building and executing DAGs. Its support for automatic differentiation simplifies the training of machine learning models, as gradients are computed directly from the graph representation. Tensor-

Flow seamlessly integrates with GPUs and TPUs, leveraging hardware accelerators for high-performance training and inference. The framework also includes tools like TensorBoard for visualizing computation graphs and monitoring training processes, addressing some of the challenges associated with debugging and performance profiling in DAG-based systems.

3 Future Directions

The evolution of distributed ML frameworks focuses on overcoming current limitations. Promising areas include:

- **Decentralized Topologies:** Reducing reliance on central servers to improve fault tolerance and scalability.
- **Federated Learning:** Enhancing privacy and efficiency by training models locally on edge devices.
- **Optimized Hardware Utilization:** Leveraging specialized hardware like GPUs, TPUs, and FPGAs for accelerated training.
- **Hybrid Approaches:** Combining strengths of various topologies, such as integrating parameter servers with data flow mechanisms.

4 Expected Results

Broader Impacts: Beyond technical contributions, this research aims to inform the development of more accessible and efficient distributed ML frameworks. By providing a comprehensive analysis of communication topologies. The findings could democratize ML capabilities for resource-constrained environments.

References

Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. 2013. Parameter server for distributed machine learning. In *Big learning NIPS workshop*, volume 6. Lake Tahoe, CA.

Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65.