iJCEM International Journal of Core Engineering & Management

International Journal of Core Engineering & Management
Volume-7, Issue-12, 2024 ISSN No: 2348-9510

SCALING NEURAL NETWORK TRAINING: A REVIEW OF DISTRIBUTED GPU COMPUTING TECHNIQUES

Abhishek Shivanna,
abkshvn@gmail.com

Abstract

Training large-scale neural networks has become foundational to advancements in artificial intelligence, yet it presents significant computational challenges. Distributed GPU computing offers a scalable solution, enabling parallelized workloads across multiple GPUs and systems. This paper reviews the key paradigms of distributed training—data parallelism, model parallelism, and pipeline parallelism—highlighting their trade-offs and practical applications. It explores state-of-the-art frameworks such as PyTorch Distributed Data Parallel and NVIDIA NCCL, alongside optimization techniques like gradient accumulation and mixed precision training. Challenges including scalability bottlenecks, communication overhead, and energy efficiency are addressed, with insights drawn from case studies of models like GPT and PaLM. Finally, the paper identifies future opportunities in hardware and software innovations to further enhance distributed GPU training efficiency and scalability.

***Keywords** distributed gpu computing, neural network training, data parallelism, model parallelism, pipeline parallelism, hybrid training strategies.*

I. INTRODUCTION

The increasing complexity and scale of neural networks have been instrumental in driving advancements in artificial intelligence (AI). Models such as GPT, PaLM, and DALL-E have demonstrated remarkable capabilities [1], but their training demands often exceed the computational capacity of individual GPUs or single-node systems. These challenges are compounded by the exponential growth in data and the need for larger models to achieve state-of-the-art results across domains.

Distributed GPU computing has emerged as a critical enabler for training large-scale neural networks, offering a scalable solution to meet these demands. By leveraging parallelism across multiple GPUs and nodes, distributed training significantly accelerates computations, reduces memory constraints, and improves resource utilization [2]. However, it also introduces new complexities, such as communication overhead, synchronization issues, and scalability bottlenecks.

This paper provides a comprehensive review of distributed GPU computing for training large-scale neural networks. It explores the foundational paradigms of distributed training, including data parallelism, model parallelism, and pipeline parallelism, while highlighting hybrid approaches that combine these methods for optimal performance. The paper also examines prominent frameworks like PyTorch Distributed Data Parallel (DDP) and NVIDIA NCCL, as well as techniques to optimize training, such as mixed precision and gradient accumulation.

International Journal of Core Engineering & Management
Volume-7, Issue-12, 2024 ISSN No: 2348-9510

In addition to detailing these methods and tools, the paper addresses key challenges, such as scalability, energy efficiency, and communication efficiency. Through case studies of models like GPT and PaLM, the paper illustrates practical implementations and insights gained from real-world deployments. Finally, the review discusses emerging hardware and software innovations that promise to further advance the field.

By synthesizing these insights, this paper aims to provide a concise yet thorough resource for researchers and engineers navigating the challenges of distributed GPU computing in training next-generation neural networks.

II. CORE PARADIGMS IN DISTRIBUTED GPU TRAINING

Efficiently training large-scale neural networks requires leveraging the power of multiple GPUs, often across multiple machines. Distributed GPU training employs different paradigms to divide the computational workload, each with unique strengths, limitations, and use cases. This section reviews the three primary paradigms—data parallelism, model parallelism, and pipeline parallelism—along with hybrid approaches that combine these techniques to maximize scalability and efficiency.

- **Data parallelism** is the most commonly used paradigm, particularly for models that fit entirely within the memory of a single GPU. In this approach, the dataset is divided into smaller shards (see Fig 1), and each GPU processes a subset of the data independently. Gradients are then synchronized across GPUs to ensure that all replicas of the model remain consistent. The simplicity and robustness of data parallelism make it the default choice for many training tasks, especially when using frameworks like PyTorch Distributed Data Parallel (DDP) or Tensor Flow Multi Worker Mirrored Strategy. However, as the number of GPUs increases, the overhead associated with gradient synchronization—especially in large clusters—can reduce efficiency. This approach is less suitable for extremely large models that exceed the memory capacity of a single GPU, as it assumes the entire model can be replicated on each device.

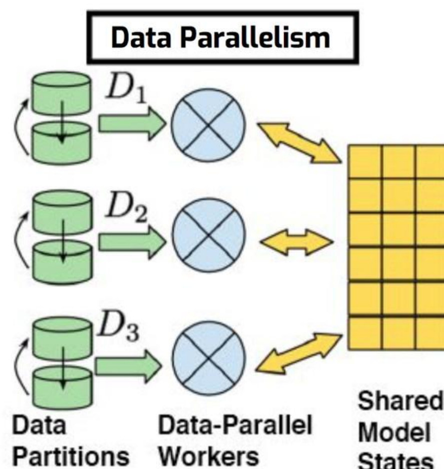

Fig 1: Data Parallelism

- **Model parallelism** addresses the limitations of data parallelism by dividing the model itself across multiple GPUs (see Fig 2). This can be done by splitting individual layers (horizontal parallelism) or assigning sequential layers to different GPUs (vertical parallelism). Model parallelism enables the training of massive architectures, such as GPT variants [3], which would otherwise be impossible to fit on a single GPU. Frameworks like Megatron-LM are specifically designed to support such use cases [4]. Despite its advantages, model parallelism introduces significant communication overhead, as activations and gradients must be transferred between GPUs during both forward and backward passes. Load balancing can also be a challenge, particularly when some parts of the model require more computation than others, leading to inefficiencies.

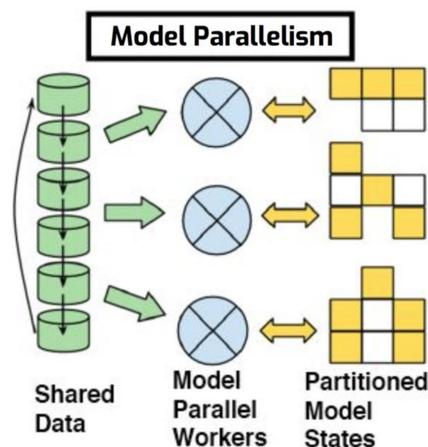

Fig 2: Model Parallelism

- **Pipeline parallelism** combines elements of data and model parallelism by partitioning the model into stages and distributing these stages across GPUs. Each GPU processes its assigned stage and passes intermediate results to the next GPU in the pipeline. This approach reduces memory usage by distributing computations across GPUs while maintaining a steady flow of data through the pipeline. Pipeline parallelism is particularly useful for sequential models and large architectures with clear stage boundaries. However, pipeline training can suffer from "pipeline bubbles" or stalls, where GPUs idle while waiting for dependencies from earlier stages. This makes scheduling and synchronization critical to maximizing efficiency.

In many cases, hybrid approaches that combine two or more paradigms are employed to balance their respective strengths and weaknesses. For instance, data and model parallelism are often combined for training large transformer models, where data parallelism handles input distribution and model parallelism addresses memory constraints. Similarly, data and pipeline parallelism can be used together to leverage the benefits of pipelined execution while maintaining efficient data distribution. Hybrid strategies require careful tuning and add complexity to implementation, but they can achieve exceptional scalability and resource utilization when designed correctly.

Choosing the appropriate paradigm depends on the characteristics of the model, the size of the dataset, and the available hardware. Data parallelism is well-suited for moderate-sized models and large datasets, while model parallelism is ideal for handling memory-constrained architectures. Pipeline parallelism provides a middle ground [11], but its efficiency depends on how well the model can be segmented into stages. Hybrid approaches are best for large-scale deployments

where maximizing efficiency across multiple GPUs and nodes is paramount. By understanding the trade-offs and use cases of these paradigms, researchers and engineers can design distributed training strategies that align with their specific requirements.

III. DISTRIBUTED TRAINING FRAMEWORKS AND TOOLS

Distributed training has been made feasible and efficient due to the evolution of several frameworks and libraries that manage the inherent complexities of parallelism, synchronization, and communication between GPUs. These tools not only abstract the lower-level technical details but also provide high-level interfaces for implementing sophisticated distributed training strategies. Understanding their architecture and capabilities is essential for designing scalable and efficient training pipelines.

1. PyTorch Distributed Data Parallel (DDP)

PyTorch Distributed Data Parallel (DDP)[5] is among the most widely used frameworks for distributed training, thanks to its integration with the popular PyTorch deep learning library. DDP works by replicating the model across multiple GPUs and splitting the dataset into smaller shards, which are processed independently on each device. During the backward pass, it synchronizes gradients across GPUs to maintain consistency in the model's state. This synchronization relies on the NVIDIA Collective Communication Library (NCCL), a high-performance backend optimized for GPU-to-GPU communication. DDP's design prioritizes ease of use, seamlessly integrating with PyTorch's APIs, which allows researchers and practitioners to transition from single-GPU to multi-GPU training with minimal changes to their codebase. Additionally, DDP supports advanced features like mixed precision training, which reduces memory usage while maintaining numerical stability, making it ideal for training large-scale models on limited hardware resources.

2. TensorFlow MultiWorkerMirroredStrategy

TensorFlow's MultiWorkerMirroredStrategy provides a solution for distributed training that is both powerful and user-friendly. Unlike DDP, which is tightly coupled to PyTorch, this strategy is built into TensorFlow, offering a native method for leveraging multiple GPUs or nodes. It achieves parallelism by mirroring the model across devices and synchronizing updates to ensure all replicas remain consistent. The strategy is particularly notable for its robust fault-tolerant capabilities, allowing workers to recover from hardware or network failures without losing significant progress. This makes it well-suited for long-running training tasks in cloud environments, where failures are more common. MultiWorkerMirroredStrategy also includes optimizations such as gradient compression, which reduces the communication overhead associated with synchronizing weights during training, enabling more efficient scaling to larger clusters.

3. Horovod

While frameworks like DDP and MultiWorkerMirroredStrategy are tied to specific ecosystems, Horovod offers a more flexible approach to distributed training by supporting multiple deep learning libraries, including PyTorch, TensorFlow, and Keras. Originally developed by Uber, Horovod introduces an MPI-inspired interface [6] that abstracts communication operations, making it easier to implement distributed training without getting bogged down in the details. A standout feature of Horovod is its ring-all reduce algorithm, which efficiently synchronizes gradients [9] by organizing GPUs into a logical ring structure. This approach minimizes communication overhead, making it especially effective in large clusters with many GPUs.

International Journal of Core Engineering & Management
Volume-7, Issue-12, 2024 ISSN No: 2348-9510

Horovod's flexibility and efficiency have made it a popular choice for organizations that require cross-library compatibility or operate in heterogeneous environments.

4. Megatron-LM

For specialized use cases involving massive language models, NVIDIA's Megatron-LM provides a tailored solution. Unlike general-purpose frameworks, Megatron-LM focuses on model parallelism, enabling the training of architectures that exceed the memory capacity of a single GPU. It achieves this by dividing large transformer models into smaller components, which are distributed across multiple GPUs. The framework is tightly integrated with NCCL and other NVIDIA tools, ensuring optimal performance. Megatron-LM's primary use case is in training cutting-edge models like GPT-3, where the scale of computation and memory requirements necessitates advanced parallelization techniques. Although highly effective for its intended purpose, Megatron-LM's specialized nature limits its applicability to other types of neural networks or smaller-scale tasks.

5. DeepSpeed

DeepSpeed, developed by Microsoft, represents one of the most comprehensive frameworks for distributed training, incorporating multiple paradigms and advanced optimizations. It introduces the Zero Redundancy Optimizer (ZeRO), which addresses memory bottlenecks [7] by distributing optimizer states, gradients, and model parameters across GPUs. This innovation allows DeepSpeed to train models with tens or even hundreds of billions of parameters using fewer resources. In addition to ZeRO, DeepSpeed supports pipeline parallelism, gradient accumulation, and mixed precision training, making it one of the most versatile tools for distributed computing. Its design caters to the growing trend of hybrid parallelism, enabling users to combine data, model, and pipeline parallelism to achieve optimal scalability. DeepSpeed is particularly suited for training state-of-the-art transformer models and remains a key player in the ecosystem of distributed GPU computing.

IV. OPTIMIZATION TECHNIQUES FOR DISTRIBUTED TRAINING

Effective distributed training involves more than simply leveraging multiple GPUs or nodes. It requires a suite of optimization techniques to ensure efficient utilization of resources, minimize communication overhead, and maintain the stability of training processes. These techniques address the unique challenges posed by distributed systems, such as memory limitations, communication latency, and synchronization inefficiencies. Below, we delve into several key optimization techniques, explaining their underlying principles and practical applications.

1. Gradient Accumulation

Gradient accumulation is a technique used to overcome the memory limitations of GPUs when training with large batch sizes. In deep learning, larger batch sizes often lead to more stable convergence and faster training; however, memory constraints can make this infeasible, especially for large models. Gradient accumulation addresses this by dividing a large batch into smaller micro-batches that fit within GPU memory. Instead of updating the model's parameters after every micro-batch, gradients are accumulated over several micro-batches. Once all gradients for the large batch are calculated, they are averaged, and a single update is applied. This allows the effective batch size to exceed the memory capacity of individual GPUs without requiring modifications to the model architecture. Gradient accumulation is particularly beneficial in scenarios where large-

iJCEM International Journal of Core Engineering & Management

International Journal of Core Engineering & Management

Volume-7, Issue-12, 2024 ISSN No: 2348-9510

scale models, such as GPT or PaLM, are trained on limited hardware resources.

2. Mixed Precision Training

Mixed precision training combines 16-bit (half-precision) and 32-bit (single-precision) floating-point computations to accelerate training while reducing memory usage. GPUs with specialized tensor cores, such as NVIDIA's Volta and Ampere architectures, are optimized for 16-bit operations, enabling faster computation without significantly compromising numerical accuracy. Mixed precision training employs dynamic loss scaling to prevent numerical underflow [8], ensuring that gradients remain large enough to be represented in 16-bit format. This approach is particularly effective in distributed environments, where reduced memory usage allows for larger models or batch sizes, and faster computation decreases the time spent on each iteration. By reducing the overhead of both computation and communication, mixed precision training has become a standard optimization technique for distributed systems, especially when training large transformer models.

3. Gradient Compression

Communication overhead is a major bottleneck in distributed training, particularly in data-parallel setups where gradients need to be synchronized across GPUs. Gradient compression reduces the size of gradient updates before transmission [10], addressing this bottleneck. Techniques such as quantization and sparsification are commonly used for this purpose. Quantization involves representing gradients with fewer bits, such as using 8-bit integers instead of 32-bit floats. Sparsification, on the other hand, focuses on transmitting only the most significant gradients while discarding smaller, less impactful values. Although these methods introduce some loss of precision, the trade-off is often worth it in scenarios where communication delays dominate the overall training time. Advanced implementations include error-feedback mechanisms, which compensate for the discarded information in subsequent iterations, maintaining convergence stability.

4. Overlapping Computation and Communication

In distributed systems, GPUs can often remain idle while waiting for communication operations, such as gradient synchronization, to complete. To address this inefficiency, overlapping computation with communication has become a key optimization strategy. This involves initiating communication tasks, such as all-reduce operations for gradient synchronization, while simultaneously performing computations for other parts of the training process. For instance, the backward pass of the neural network can be partitioned into segments, enabling gradients from earlier layers to be synchronized while gradients for later layers are still being computed. Frameworks like PyTorch DDP and Horovod incorporate this optimization, ensuring that GPUs spend less time idle and more time performing useful work. This approach is particularly beneficial in large-scale deployments where communication latency across nodes is significant.

5. Check pointing and Fault Tolerance Optimization

Long-running distributed training jobs are susceptible to failures caused by hardware malfunctions, network issues, or software bugs. Check pointing is an optimization technique that mitigates the impact of such failures by periodically saving the state of the model, optimizer, and training progress to persistent storage. In the event of a failure, training can resume from the latest checkpoint, avoiding the need to restart from scratch. Advanced checkpointing strategies minimize storage overhead and time spent saving checkpoints, such as saving only incremental

changes or using parallel I/O operations. This technique is critical in distributed systems, where the likelihood of failure increases with the number of GPUs and nodes involved.

6. Dynamic Load Balancing

In distributed training, particularly with model or pipeline parallelism, ensuring balanced workloads across GPUs is crucial for maintaining efficiency. Imbalanced workloads, where some GPUs finish their tasks earlier than others, lead to idle time and reduced overall throughput. Dynamic load balancing addresses this by dynamically redistributing tasks to underutilized GPUs. This can be achieved through profiling the training process and adjusting the assignment of layers, data shards, or pipeline stages accordingly. Advanced implementations use runtime monitoring and adaptive scheduling algorithms to react to changes in workload characteristics during training. While this adds complexity to the system, the gains in utilization often justify the additional effort.

7. Energy-Aware Training

As the scale of distributed training grows, so does its energy consumption, making energy efficiency an important consideration. Energy-aware training techniques focus on optimizing resource utilization to reduce power consumption without compromising performance. Strategies include selectively activating only the necessary GPU cores (e.g., sparsity-aware computation), minimizing idle time through scheduling optimizations, and dynamically adjusting precision or batch size based on power usage. Some frameworks also integrate energy monitoring tools, providing real-time feedback to guide further optimizations. Given the increasing emphasis on sustainable AI, energy-aware training is an emerging area of focus in distributed training research.

V. CHALLENGES IN DISTRIBUTED TRAINING

While distributed training has revolutionized the field of machine learning by enabling the training of large-scale neural networks, it comes with a set of challenges that must be addressed to achieve efficient, scalable, and reliable performance. These challenges span hardware, software, and algorithmic domains, and understanding them is crucial for designing robust distributed systems. Below, we explore these challenges in depth, explaining their origins and discussing approaches to mitigate them.

1. Scalability Bottlenecks

One of the most significant challenges in distributed training is achieving linear scalability. Ideally, doubling the number of GPUs should halve the training time, but this is rarely the case due to diminishing returns. As the number of GPUs increases, communication overhead and synchronization delays grow, reducing the efficiency of the system. For instance, in data parallelism, the process of synchronizing gradients across GPUs becomes a bottleneck as the cluster size grows. This is particularly problematic in large-scale clusters where inter-node communication latency can outweigh the benefits of parallelism.

To address scalability bottlenecks, researchers have explored hierarchical communication strategies, such as dividing GPUs into subgroups for local synchronization before global synchronization. Another approach is gradient compression, which reduces the size of data exchanged between GPUs. Additionally, advanced algorithms, such as asynchronous updates and stale gradient methods, trade some accuracy for improved scalability. However, these solutions often require careful tuning and can introduce complexity into the training process.

iJCEM
International Journal of Core Engineering & Management

International Journal of Core Engineering & Management
Volume-7, Issue-12, 2024 ISSN No: 2348-9510

2. Communication Overhead

Communication overhead arises from the need to exchange information between GPUs during distributed training. This includes synchronizing model parameters in data parallelism, exchanging activations and gradients in model parallelism, and transmitting intermediate outputs in pipeline parallelism. The cost of communication increases with the number of GPUs and the size of the model, leading to significant inefficiencies in large-scale setups.

High-performance interconnects, such as NVIDIA NVLink and Infiniband, play a critical role in mitigating communication latency by providing high bandwidth and low latency between GPUs. Libraries like NVIDIA NCCL optimize collective communication operations, such as all-reduce, to make data exchange more efficient. Another promising area of research is overlapping communication with computation, where communication tasks are initiated while computations for other parts of the model are still ongoing. Despite these optimizations, communication overhead remains a fundamental challenge, particularly when scaling across multiple nodes.

3. Load Balancing and Resource Utilization

Distributed training systems often face imbalances in workload distribution, particularly in model and pipeline parallelism. Imbalanced workloads occur when certain GPUs process more computationally intensive tasks or larger data shards, leading to idle time for other GPUs. This reduces overall system efficiency and elongates training time.

Load balancing is a complex challenge, as it requires dynamically redistributing tasks in response to workload variations during training. For example, in model parallelism, different layers of a neural network may have varying computational demands, and assigning these layers to GPUs must be carefully managed. In pipeline parallelism, "pipeline bubbles" occur when downstream GPUs remain idle while waiting for upstream GPUs to finish their tasks. Strategies such as fine-grained task partitioning, runtime profiling, and adaptive scheduling algorithms help alleviate these issues but require additional overhead and careful implementation.

4. Fault Tolerance

Distributed training systems are inherently more susceptible to failures than single-GPU systems due to their scale and complexity. Hardware failures, network disruptions, or software bugs can interrupt training, leading to lost progress and wasted computational resources. The risk of failure increases with the number of GPUs and nodes involved, making fault tolerance a critical requirement for distributed systems.

Check pointing is the most commonly used technique to address this challenge. By periodically saving the model's state, optimizer state, and training progress, checkpointing allows training to resume from the last saved point in the event of a failure. However, frequent checkpointing can introduce overhead, especially for large models, so optimizing checkpoint frequency and storage mechanisms is essential. Some frameworks, like TensorFlow and Horovod, include built-in support for fault tolerance, such as elastic training, which allows nodes to dynamically join or leave the cluster without disrupting the overall process.

5. Memory Constraints and Scaling Models

Training large-scale models often exceeds the memory capacity of individual GPUs, even in distributed setups. While model parallelism and pipeline parallelism can distribute the model across multiple GPUs, they introduce additional challenges such as increased communication overhead and complexity in managing dependencies between segments of the model. Techniques

like gradient checkpointing, where intermediate results are recomputed instead of stored, help reduce memory usage but at the cost of additional computation.

Recent advancements, such as ZeRO (Zero Redundancy Optimizer) in Microsoft's DeepSpeed, address this issue by partitioning optimizer states, gradients, and model parameters across GPUs. This approach minimizes memory redundancy and enables training of models with tens or hundreds of billions of parameters. However, these solutions require careful orchestration and may not generalize well to all architectures.

6. Debugging and Reproducibility

Distributed training systems are notoriously difficult to debug due to their inherent complexity. Issues such as deadlocks, race conditions, and inconsistent results can arise from synchronization errors, hardware variability, or software bugs. These problems are further exacerbated in multi-node setups, where network delays or hardware heterogeneity can introduce non-deterministic behaviour.

To improve debugging and reproducibility, researchers rely on tools like Tensor Board and PyTorch's Profiler for monitoring and visualizing training performance [12]. Logging and checkpointing are also essential for tracing errors and analysing their causes. However, achieving true reproducibility in distributed environments remains challenging, as even small variations in floating-point computations can lead to different outcomes. This has led to increased interest in deterministic algorithms and robust logging frameworks.

VI. FUTURE DIRECTIONS IN DISTRIBUTED TRAINING

The field of distributed training is rapidly evolving, driven by the increasing demand for scalable and efficient solutions to train ever-larger neural networks. As new challenges arise with advancements in model architecture and data complexity, so do opportunities for innovation. This section explores promising future directions that aim to redefine the landscape of distributed training.

1. Hardware Innovations

The design and deployment of specialized hardware accelerators are crucial to meeting the growing demands of distributed training. While GPUs remain the primary workhorse, new architectures like tensor processing units (TPUs), custom AI accelerators[13], and domain-specific hardware are gaining traction. These devices are optimized for the massive parallelism and reduced precision requirements of deep learning workloads. Emerging trends in hardware design focus on integrating high-bandwidth memory (HBM), reducing memory bottlenecks, and improving energy efficiency.

Another exciting area of development is advanced interconnect technologies. Technologies like NVIDIA's NVLink and Mellanox's Infiniband have already improved communication within GPU clusters, but newer solutions, such as silicon photonics, aim to push the boundaries of inter-node bandwidth while minimizing latency. These advancements promise to alleviate communication bottlenecks, one of the most significant limitations of current distributed systems.

2. Advanced Communication Protocols

Efficient communication is a cornerstone of distributed training, and innovations in this domain are expected to play a pivotal role in improving scalability. Protocols like Remote Direct Memory Access (RDMA) and hardware-accelerated collective operations [14] are already being leveraged to

minimize communication latency. Future efforts are likely to focus on adaptive communication strategies that dynamically optimize data exchange based on workload characteristics.
Decentralized communication models represent another area of interest. Traditional centralized communication strategies, such as parameter servers, can become bottlenecks as the number of devices increases. Decentralized approaches, such as peer-to-peer communication networks, offer a scalable alternative by distributing the communication load across the system. These models require sophisticated algorithms to ensure synchronization and fault tolerance, making them an active area of research.

3. Exploring New Algorithms and Architectures

Finally, the evolution of distributed training will be driven by advances in algorithms and model architectures themselves. Techniques such as sparsity-aware training, neural architecture search [15], and modular models offer opportunities to reduce resource requirements while maintaining state-of-the-art performance. Distributed training systems will need to adapt to these emerging trends, incorporating new paradigms and optimizing for novel workloads.

VII. CONCLUSION

The increasing scale and complexity of neural networks have made distributed GPU computing indispensable for training modern AI models. This paper reviewed the core paradigms of distributed training—data parallelism, model parallelism, pipeline parallelism, and their hybrid combinations—emphasizing their advantages, challenges, and applicability. These paradigms form the foundation for scaling training workloads across multiple GPUs and nodes, enabling the development of models with billions of parameters, such as GPT and PaLM.

To operationalize these paradigms, frameworks like PyTorch Distributed Data Parallel and TensorFlow MultiWorkerMirroredStrategy provide user-friendly interfaces for managing distributed training workflows, while technologies like NVIDIA NCCL optimize the critical communication layer. Specialized tools, such as Megatron-LM, cater to the unique demands of massive language models, pushing the boundaries of what is computationally feasible. Together, these tools and technologies simplify the implementation of distributed training while addressing common bottlenecks.

Optimization techniques, such as mixed precision training, gradient accumulation, and communication-efficient algorithms, further enhance the performance and scalability of distributed systems. However, challenges remain, including communication overhead, energy efficiency, and scalability bottlenecks as cluster sizes grow. Addressing these challenges requires a thoughtful selection of paradigms, frameworks, and hardware configurations tailored to the specific needs of a given training task.

Looking ahead, innovations in hardware, such as tensor cores and high-bandwidth interconnects, alongside advancements in distributed training algorithms, will further expand the horizons of what distributed GPU computing can achieve. As AI models continue to grow in size and complexity, distributed training will remain a cornerstone of scalable and efficient neural network development.

This review provides a comprehensive foundation for understanding the current state of

iJCEM International Journal of Core Engineering & Management

International Journal of Core Engineering & Management
Volume-7, Issue-12, 2024 ISSN No: 2348-9510

distributed GPU training. By synthesizing the key paradigms, frameworks, and optimizations, it equips researchers and engineers with the knowledge needed to design efficient and scalable training strategies for the next generation of AI models.

REFERENCES

1. Brown, T. B. et al. (2020). Language Models are Few-Shot Learners. arXiv preprint arXiv:2005.14165.
2. Rajbhandari, S., Rasley, J., Ruwase, O., & He, Y. (2020). ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-16).
3. Narayanan, D. et al. (2021). Efficient Large-Scale Language Model Training on GPU Clusters. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.
4. Shoeybi, M. et al. (2019). Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv preprint arXiv:1909.08053.
5. Paszke, A. et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. Advances in Neural Information Processing Systems, 32.
6. Sergeev, A., & Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint arXiv:1802.05799.
7. Rajbhandari, S. et al. (2021). ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.
8. Micikevicius, P. et al. (2018). Mixed Precision Training. International Conference on Learning Representations.
9. Thakur, R., Rabenseifner, R., & Gropp, W. (2005). Optimization of Collective Communication Operations in MPICH. The International Journal of High Performance Computing Applications, 19(1), 49-66.
10. Lin, Y. et al. (2018). Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. International Conference on Learning Representations.
11. Huang, Y. et al. (2019). GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. Advances in Neural Information Processing Systems, 32.
12. Cai, H., Gan, C., & Han, S. (2019). Once for All: Train One Network and Specialize it for Efficient Deployment. International Conference on Learning Representations.
13. Jouppi, N. P. et al. (2017). In-Datacenter Performance Analysis of a Tensor Processing Unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture (pp. 1-12).
14. Liu, J., Wu, J., & Panda, D. K. (2004). High Performance RDMA-Based MPI Implementation over InfiniBand. International Journal of Parallel Programming, 32(3), 167-198.
15. Gale, T., Elsen, E., & Hooker, S. (2019). The State of Sparsity in Deep Neural Networks. arXiv preprint arXiv:1902.09574.