JOURNAL OF INTERNATIONAL CRISIS AND RISK COMMUNICATION RESEARCH
ISSN: 2576-0017
2025, VOL 8, NO S8

Scalability And Resilience In Distributed LLM Training: A Survey Of Modern Techniques

Shreya Gupta

University of Southern California.

Abstract

This article scrutinizes the shifting terrain of distributed training systems for Large Language Models (LLMs), tackling the pivotal challenges of scalability and resilience. Modern LLMs' expanding size and intricacy expose fundamental constraints in traditional training approaches—namely memory limitations, computational demands, and communication bottlenecks. The article puts forward a methodical taxonomy built upon four interconnected pillars: parallelism approaches (data, tensor, pipeline, and sequence), memory optimization methods, fault tolerance mechanisms, and cluster management frameworks. The analysis reveals how these elements interact to facilitate effective training on volatile, preemptible cloud resources rather than dedicated supercomputing hardware. By highlighting the crucial interplay between these components, the article demonstrates how parallelism choices directly affect memory usage, communication dynamics, and resilience capabilities. Through detailed examination of systems engineered specifically for elastic training environments, the work spotlights innovations in checkpointing, recovery protocols, and dynamic reconfiguration. The conclusion identifies promising research avenues, including algorithm-system co-design for elasticity, automated parallelism strategy selection, standardized resilience benchmarking, and energy-conscious training methodologies.

Keywords: Distributed Training, Large Language Models, Memory Optimization, Fault Tolerance, Cloud-Native Computing.

1. Introduction

1.1 The Unprecedented Scale of Modern LLMs and the Resulting Systems Challenge

Artificial intelligence currently undergoes transformation propelled by large language model (LLM) scaling. GPT-3, PaLM, and Llama demonstrate that expanding parameter counts, training data volumes, and computational resources yields substantial—often emergent—capabilities in language understanding, generation, and reasoning.[1][2] This scaling phenomenon—the predictable performance gains achieved through scale—has sparked fierce competition toward increasingly massive models, with parameter counts surging from billions toward trillions.[3]

Such exponential expansion represents more than mere quantitative advancement; it has triggered a qualitative shift in required training infrastructure. Contemporary LLMs exceed the memory capacity of even cutting-edge single accelerators, rendering distributed training an absolute necessity rather than optional optimization.[1][5] Training itself constitutes a massive undertaking, demanding vast GPU clusters operating for weeks or months, consuming staggering amounts of energy and compute time.[2] This endeavor stretches existing hardware and software to breaking points, creating fundamental bottlenecks across three key dimensions:

- Memory: Storing model parameters, gradients, and optimizer states for trillion-parameter models demands tens of terabytes—far beyond any single device's capacity.[4][6]
- Computation: A single training run requires zettaFLOPs of operations, necessitating sustained, high-performance computation across thousands of devices.[8]
- Communication: Distributed settings require constant information exchange between devices, including gradients and activations. This communication frequently becomes the primary bottleneck, with some workloads spending over 90% of execution time on network operations, severely limiting scalability and efficiency.[2][3]

1.2 The Cloud-Native and Decentralized Imperative: Training on Dynamic and Distributed Resources

Traditionally, large-scale scientific computing relied on dedicated, on-premise supercomputers with specialized, custom interconnects. While LLM training benefits from such "hyperclusters," a marked paradigm shift toward elastic, cloud-native and increasingly decentralized infrastructure has emerged.[7][45] This transition is driven not only by economic forces—particularly the availability of discount "spot" virtual machines (VMs), which cost up to 5 times less than standard on-demand instances—but also by the fundamental goal of democratizing access to large-scale AI training.[7][48]

The democratization aspect is crucial: cloud-native and decentralized approaches enable a wider range of organizations, from academic institutions to startups and smaller companies, to participate in cutting-edge LLM research and development without needing to invest in prohibitively expensive dedicated infrastructure. This broader access fosters innovation and ensures that AI advancement isn't confined to only the largest technology companies with massive computing resources.

The emerging decentralized training paradigm takes this democratization even further by leveraging heterogeneous, geographically distributed resources. This approach allows training to occur across diverse computing environments—potentially spanning multiple cloud providers, private data centers, and edge devices—creating a more resilient and accessible training ecosystem. However, this new reality introduces fresh systems challenges:

- Resource Preemption: Spot instances face revocation with minimal warning, potentially derailing training runs spanning weeks or months.[7][37]
- Hardware Heterogeneity: Cloud and decentralized clusters typically comprise diverse GPU generations and architectures, causing performance variability and straggler issues.[47]
- Variable Network Performance: Unlike purpose-built, high-performance supercomputer networks, commodity cloud networks and especially geographically distributed networks exhibit higher latency and jitter, potentially crippling communication-intensive training algorithms.[8][47]
- Cross-Region Coordination: Decentralized training must overcome challenges in synchronizing work across different geographical regions with varying network capabilities and reliability.

This new landscape challenges the assumption that massive models require specialized, stable hardware.[6][45] It demands fundamental rethinking of system design, pivoting from raw performance on static resources toward resilience and elasticity on dynamic, heterogeneous, and geographically distributed resources.

1.3 A Unified Framework: The Interplay of Parallelism, Memory, and Resilience

The central argument advanced in this survey posits that distributed training's three pillars—scalability, memory efficiency, and resilience—function not as isolated concerns but as deeply interwoven considerations. Parallelism strategy selection, for instance, creates direct and cascading effects on memory consumption and communication patterns. These factors subsequently determine system vulnerability to failures and capacity for elastic adaptation.

Consider tensor parallelism versus pipeline parallelism. Tensor parallelism, which divides individual mathematical operations within model layers, effectively balances computational load but demands frequent, high-volume communication (e.g., All-Reduce) among participating devices. This creates acute sensitivity to network latency and jitter, making it most suitable for tightly-coupled GPUs within single server nodes connected via high-speed interconnects like NVLink.[5][15] Pipeline parallelism, conversely, partitions models by layers, requiring communication only of activations between sequential stages. This

substantially reduced communication volume better tolerates the higher latency typical of inter-node, commodity cloud networks.[9][13] Consequently, systems designed for resilience on preemptible cloud VMs might favor pipeline parallelism, despite tensor parallelism potentially offering superior theoretical performance on dedicated supercomputers. This interplay creates a complex, multi-dimensional optimization problem requiring careful navigation by system architects.

1.4 Contributions and Paper Organization

This survey makes several scholarly contributions:

- A systematic, comprehensive taxonomy of modern distributed LLM training techniques, organized into four logical pillars building upon one another, from foundational parallelism to cluster-level orchestration.
- A unified analytical framework clarifying critical trade-offs between computational performance, memory efficiency, communication overhead, and system robustness.
- In-depth analysis of systems specifically engineered for resilience and elasticity on dynamic, preemptible cloud resources—a critical and increasingly relevant research area.
- Identification of key underexplored research directions and a forward-looking agenda guiding future work toward more scalable, robust, and efficient training systems.

The paper continues as follows. Section 2 examines foundational parallelism strategies forming distributed training building blocks. Section 3 explores techniques for optimizing memory consumption at scale. Section 4 addresses critical mechanisms achieving fault tolerance and elasticity in dynamic environments. Section 5 investigates schedulers and cluster management systems orchestrating these complex workloads. Section 6 discusses open challenges and proposes future research directions. Section 7 concludes the survey.

2. Foundational Parallelism Strategies

Modern LLMs' immense scale necessitates the training process's distribution across multiple hardware accelerators. Distribution strategies have evolved through a clear causal chain, with each approach's limitations directly motivating subsequent innovations. This evolution reflects continuous efforts addressing the most pressing system bottlenecks at each model scaling stage. This section examines four foundational parallelism strategies—data, model (tensor and pipeline), sequence, and expert parallelism—tracing their development and analyzing respective trade-offs.

2.1 Data Parallelism (DP): The Baseline and Its Memory Bottleneck

Data Parallelism (DP) represents the most straightforward, widely adopted distributed training method. Its core concept involves replicating entire models across multiple devices (workers). Each worker processes different training data slices (mini-batches) in parallel during forward and backward passes. Maintaining model replica synchronization requires gradient aggregation across all workers through collective communication operations, typically All-Reduce, before optimizer weight updates.[9][2]

DP's primary advantage lies in training throughput scalability through very large global batch sizes—the sum of worker mini-batch sizes. However, LLM training applications face severe constraints from a fundamental bottleneck: memory. Standard DP requires each worker to maintain complete model state copies, including parameters, gradients, and optimizer states (e.g., momentum and variance vectors for Adam optimizers).[5][6] Multi-billion parameter models easily exceed single GPU memory capacity, creating an insurmountable "memory wall" rendering vanilla DP impractical.[6]

2.2 Model Parallelism (MP): Decomposing the Model

Model Parallelism (MP) emerged directly addressing DP's memory wall. Rather than replicating models, MP partitions them, distributing layers and parameters across multiple devices.[8] This leverages aggregate cluster memory capacity to accommodate massive models. MP typically manifests in two distinct forms: tensor parallelism and pipeline parallelism.

2.2.1 Tensor Parallelism (TP): Intra-Layer Parallelism

Tensor Parallelism, also termed intra-layer model parallelism, partitions individual model layers. The key insight, pioneered by Megatron-LM, recognizes that Transformer models' most computation-intensive components involve large matrix multiplications (GEMMs) within Multi-Layer Perceptron (MLP) and

attention blocks.[5][10] TP parallelizes these GEMMs by splitting weight matrices along specific dimensions (e.g., column-wise for first matrices and row-wise for second matrices in MLP blocks) and distributing shards across GPU groups.[5][10]
During computation, each GPU performs matrix multiplication on local weight matrix shards. Collective communication operations ensure mathematical equivalence to the original, non-parallel operations. This typically requires two All-Reduce operations per Transformer layer (one during forward passes and one during backward passes) for partial result aggregation.[9]
TP design couples tightly with the underlying hardware topology. It's frequent, fine-grained communication creates high sensitivity to network latency and bandwidth. Consequently, TP works most effectively across GPUs within single server nodes connected by high-bandwidth, low-latency interconnects like NVIDIA's NVLink, which sustains the required communication volume with minimal overhead.[9] While TP delivers excellent computational load balancing and high device utilization, its dependence on high-speed interconnects limits its suitability for scaling across multiple server nodes over standard datacenter networks.

2.2.2 Pipeline Parallelism (PP): Inter-Layer Parallelism

Pipeline Parallelism, or inter-layer model parallelism, adopts an alternative partitioning approach. Rather than splitting individual layers, it divides entire models into sequential stages, each comprising contiguous layer blocks. These stages distribute across different devices, forming computational pipelines.[9] Data mini-batches break down into smaller micro-batches fed sequentially into pipelines. This enables different devices to process different micro-batches simultaneously, overlapping computation across stages.
Early, influential PP implementations include GPipe and PipeDream:

- GPipe introduced synchronous pipeline models using micro-batching to maximize device utilization.[9][13] During forward passes, activations pass from one stage to the next. During backward passes, gradients flow in reverse. Maintaining mathematical equivalence with serial execution requires accumulating gradients for all micro-batches before performing single, synchronous optimizer steps for entire mini-batches.[13] While ensuring correctness, this creates "pipeline bubbles"—periods at mini-batch processing beginnings and ends where some devices remain idle, waiting for pipelines to fill or drain. Bubble size proportionally increases with pipeline depth, potentially limiting efficiency.[14]
- PipeDream aimed to eliminate bubbles and achieve higher throughput. It employs asynchronous "1F1B" (one-forward, one-backward) scheduling, where devices alternate between performing forward passes for new micro-batches and backward passes for previous ones.[11][16] This keeps devices nearly fully utilized during steady-state operation. However, this efficiency introduces weight staleness. Since backward passes for given micro-batches might execute after model weights update from preceding micro-batches, gradients are computed using slightly older parameter versions. Ensuring correctness requires "weight stashing," storing multiple weight versions for each in-flight micro-batch, increasing memory requirements.[11] Subsequent work, including PipeDream-2BW and PipeMare, sought to improve asynchronous pipelining memory efficiency and convergence properties.[16][17]

Compared to TP, PP maintains substantially lower communication-to-computation ratios. It only requires passing activations and gradients between adjacent stages at partition boundaries, significantly smaller data volumes than TP's All-Reduce operations.[9] This creates far greater tolerance for network latency, making PP better suited for scaling across multiple nodes in commodity cloud environments.[9]

2.3 Sequence Parallelism (SP): Scaling to Long Contexts

As models grew larger and processing longer input sequences (contexts) became crucial, a new memory bottleneck emerged: activations. Even with TP and PP distributing model parameters, storing intermediate activations for each layer during forward passes requires prohibitive memory, particularly with sequence lengths reaching millions of tokens.[21][22]
Sequence Parallelism (SP) addresses this specific challenge. Its core concept involves partitioning tensors not along hidden dimensions (like TP) but along sequence dimensions.[20] For sequences of length L with TP degree N, each GPU stores activations for only L/N tokens.

The effectiveness of SP depends significantly on its integration with other technologies, particularly optimized attention implementations. Modern systems like FlashAttention provide crucial support for SP by enabling efficient memory access patterns and reducing redundant memory operations, which is especially important when dealing with partitioned sequences.[23] Without these optimized attention mechanisms, the benefits of SP would be substantially reduced due to the computational overhead of managing partitioned sequence dimensions.

Implementations in Megatron-LM and DeepSpeed-Ulysses display particular ingenuity. An All-Reduce operation mathematically equals a Reduce-Scatter operation followed by an All-Gather operation. SP modifies standard TP communication patterns by replacing the first forward pass All-Reduce operations with Reduce-Scatter, and the second operations with backward pass All-Gather. This ensures activations, replicated across TP ranks after the first All-Reduce in standard TP, remain partitioned in SP. This modification delivers substantial activation memory savings with zero additional communication overhead, as total transferred data volume remains unchanged.[20] DeepSpeed-Ulysses further optimizes this using efficient all-to-all collectives managing distributed attention computation.[23][24] SP's primary trade-off involves tight coupling with TP; it functions as a TP optimization and cannot apply independently.[20]

Different approaches to SP have emerged, with variations in how they handle attention computation across partitioned sequences. Some methods focus on reducing memory footprint at the cost of additional communication, while others prioritize minimizing communication overhead but require more sophisticated implementation. The choice between these approaches depends heavily on specific hardware configurations and sequence length requirements.

2.4 Expert Parallelism (EP): Scaling Mixture-of-Experts Models

An emerging parallelism strategy worth noting is Expert Parallelism (EP), designed specifically for Mixture-of-Experts (MoE) architectures. MoE models distribute computation across specialized "expert" networks that process only a subset of the input tokens based on a learned routing mechanism. This sparse activation pattern creates unique opportunities for parallelism that traditional approaches don't fully exploit.

In EP, different expert networks are distributed across devices, with each device responsible for a subset of the experts. This approach is particularly effective because only a small fraction of experts activate for any given input token, reducing both computation and communication requirements compared to dense models of equivalent parameter counts. Models like Switch Transformer, GShard, and more recent efforts like Mixtral demonstrate how EP can achieve remarkable parameter efficiency and training scalability.

EP introduces its own challenges, particularly in load balancing, as popular experts can create computational hotspots. Advanced implementations use techniques like auxiliary load balancing losses and dynamic expert allocation to mitigate these issues. As MoE architectures continue to gain prominence for their parameter efficiency, EP will likely become an increasingly important part of the distributed training ecosystem.

2.5 Hybrid Approaches and 3D Parallelism

In practice, these foundational strategies rarely apply in isolation. State-of-the-art training frameworks like DeepSpeed and Megatron-LM combine them into sophisticated hybrid approaches, often called "3D Parallelism".[25][30]

To illustrate how this works in practice, consider training a 175-billion parameter model (similar to GPT-3) on a cluster of 1024 GPUs. A common, highly effective multi-node cluster configuration might include:

- Tensor Parallelism (TP): 8-way parallelism within each server node, partitioning individual layers across 8 GPUs connected by NVLink
- Pipeline Parallelism (PP): 16-way parallelism across server nodes, with each group of 8 GPUs (one full node) handling a different segment of the model's layers
- Data Parallelism (DP): 8-way replication of the entire pipeline, allowing 8 different batches to be processed simultaneously
- Sequence Parallelism (SP): Integrated with the 8-way TP to efficiently handle long context windows

In this configuration, the full system processes 8 different batches simultaneously (DP=8), each batch flows through 16 pipeline stages (PP=16), and within each stage, computation is distributed across 8 GPUs (TP=8). This creates effective utilization of all 1024 GPUs (8×16×8=1024).
This hierarchical approach enables system designers to map appropriate parallelism strategies to corresponding hardware hierarchy levels, maximizing both scalability and efficiency. TP and SP operate within server nodes to leverage high-speed interconnects, while PP spans across nodes to accommodate higher-latency inter-node communication. DP then allows further scaling by replicating this entire pipeline structure.

Parallelism Strategy	Partitioning Strategy	Communication Pattern	Primary Benefit	Key Limitation / Cost	Prominent Implementation(s)
Data Parallelism	Mini-batch data	All-Reduce (gradients)	Scales throughput	Replicates model state, GPU memory wall	PyTorch DDP
Tensor Parallelism	Tensors within a layer	All-Reduce or All-Gather/Reduce-Scatter (activations/gradients)	Reduces model memory	High communication volume, sensitive to latency	Megatron-LM
Pipeline Parallelism	Layers of the model	Point-to-Point (activations)	Reduces model memory, low communication volume	Pipeline bubble (synchronous) or weight staleness (asynchronous)	GPipe, PipeDream
Sequence Parallelism	Sequence dimension of activations	All-Gather/Reduce-Scatter (activations)	Reduces activation memory for long sequences	Requires Tensor Parallelism	Megatron-LM, DeepSpeed-Ulysses

Table 1: Comparison of Parallelism Strategies

3. Memory Optimization at Scale
While parallelism strategies distribute models across multiple devices, per-device memory footprints remain critical constraints. Advanced optimization techniques reduce per-device memory consumption, enabling training larger models, using larger batch sizes for improved throughput, or training on hardware with limited VRAM. These techniques typically operate orthogonally to parallelism and combine with it for maximum effect. They generally involve fundamental trade-offs, exchanging abundant resources (e.g., compute cycles or CPU memory) for scarcer ones: GPU memory.
3.1 Activation Recomputation (Checkpointing)
During training backward passes, autograd engines require intermediate activations computed during forward passes to calculate gradients. Storing all activations in GPU memory consumes enormous resources, often becoming primary memory bottlenecks after accounting for model parameters and optimizer states.[26][27]
Activation recomputation, also called activation checkpointing or gradient checkpointing, directly addresses this by trading memory for computation. The approach works simply: instead of storing all intermediate activations for specified model regions (e.g., Transformer blocks), systems discard them after

forward passes. During backward passes, when these activations become necessary for gradient calculation, systems re-run forward passes for specific blocks to "rematerialize" activations on-the-fly.[27][28]
This approach dramatically reduces peak activation memory requirements, as activations no longer occupy memory throughout entire forward-backward cycles. Memory savings come at the cost of computational overhead from additional forward passes, potentially increasing total training time. However, for memory-bound workloads, this trade-off often proves highly favorable, potentially determining whether models fit in memory at all. Frameworks like DeepSpeed provide sophisticated activation checkpointing implementations, including features that partition recomputed activations across model-parallel ranks or even offload them to CPU memory, further extending memory efficiency boundaries.[29][30]

3.2 The Zero Redundancy Optimizer (ZeRO)

The Zero Redundancy Optimizer (ZeRO), developed by Microsoft within DeepSpeed, represents an optimization family fundamentally reimagining data parallelism memory usage.[4][31] Standard data parallelism wastes memory by replicating entire training states—model parameters, gradients, and optimizer states—on every GPU. ZeRO systematically eliminates this redundancy by partitioning states across data-parallel workers.

3.2.1 ZeRO Stages 1, 2, & 3: Partitioning Model States

ZeRO implements three progressive stages, each offering greater memory savings by partitioning more training state components:[31][34]

- ZeRO-Stage 1 (Optimizer State Partitioning): This stage addresses optimizer states (e.g., momentum and variance vectors in Adam optimizers), which consume substantial memory (e.g., 8 bytes per model parameter for Adam in mixed precision). Rather than replicating these states, ZeRO-1 partitions them across data-parallel processes. Each GPU maintains and updates only a slice of total optimizer states. After gradient reduction, each GPU updates its local parameter partition using its local optimizer state shard. A final All-Gather operation ensures all GPUs receive fully updated parameters. This stage alone reduces model state memory requirements up to 4x.[31]
- ZeRO-Stage 2 (Gradient and Optimizer State Partitioning): This stage extends the first by also partitioning gradients. Since each GPU only updates parameter partitions, it needs only gradients corresponding to those partitions. During backward passes, instead of All-Reduce, Reduce-Scatter operations send each GPU only final, averaged gradients for parameter slices they manage. This eliminates gradient redundancy and provides up to 8x memory reduction compared to standard data parallelism, while maintaining identical communication volumes.[31][36]
- ZeRO-Stage 3 (Parameter, Gradient, and Optimizer State Partitioning): This most aggressive stage partitions model parameters themselves alongside gradients and optimizer states.[31] Each data-parallel worker holds only model weight shards at any given time. During forward and backward passes, All-Gather collectives dynamically assemble complete layers just before computation needs them. Once computation completes, non-local parameter memory gets released. This powerful technique makes memory savings linear with data-parallel degrees. For example, 64-GPU systems using ZeRO-3 can reduce model state memory footprints 64-fold, at the cost of modest 50% communication volume increases compared to standard data parallelism.[31][34]

From a practical perspective, most production training setups find ZeRO-1 and ZeRO-2 offer an optimal balance of memory savings and performance overhead. These stages provide substantial memory reductions (4-8x) with minimal impact on training throughput, making them suitable for most large-scale training scenarios. ZeRO-3, while offering the most dramatic memory savings, introduces more significant communication overhead and is typically reserved for the most extreme cases where model size would otherwise be prohibitive. Organizations planning LLM training should carefully evaluate their specific hardware constraints and model sizes to determine which ZeRO stage best meets their needs.

3.2.2 ZeRO-Infinity: Breaking the GPU Memory Wall with CPU/NVMe Offloading

ZeRO memory partitioning logically extends beyond aggregate GPU cluster memory into deeper memory hierarchies. ZeRO-Offload and ZeRO-Infinity represent groundbreaking ZeRO-3 extensions doing precisely this.[32][35] These systems treat GPU memory, CPU RAM, and even fast NVMe solid-state drives as single, massive, virtual memory pools.

Partitioned model states (parameters, gradients, and optimizer states) offload from scarce, fast GPU memory to abundant but slower CPU or NVMe memory. Core innovations lie in sophisticated software orchestrating data movement across this hierarchy. Systems intelligently prefetch required parameter shards from CPU/NVMe to GPUs just before computation needs them and offload them once requirements end. By carefully overlapping data transfers with ongoing GPU computation, ZeRO-Infinity effectively hides slower memory tier latency, thereby "breaking the GPU memory wall".[35] This enables training models with tens or hundreds of trillions of parameters on existing hardware and democratizes billion-scale model training access by allowing it even on systems with limited GPU memory.[4][45]

3.3 Mixed-Precision and Quantized Training

Mixed-precision training reduces memory footprints and often accelerates computation. Instead of performing all calculations in standard 32-bit single-precision floating-point (FP32), it uses 16-bit half-precision formats like FP16 or BF16 for most forward and backward passes.[3][33] This immediately halves the memory required for storing activations and gradients.

Maintaining numerical stability and preventing information loss from small gradient values typically requires keeping master model weight copies in FP32. During optimizer steps, gradients are converted to FP32 before updating master copies. For FP16, which has limited dynamic range, dynamic loss scaling becomes critical. This involves scaling loss values upward before backward passes to bring small gradients into FP16 representable ranges, then scaling gradients back down before weight updates.[33][36] BF16 format, sharing FP32's exponent range, suffers from these issues less frequently and often requires no loss scaling. On hardware with specialized units like NVIDIA's Tensor Cores, 16-bit format operations can run significantly faster than 32-bit counterparts, providing dual benefits of memory savings and increased throughput.[33] Emerging research pushes boundaries further with 8-bit formats (FP8) for even greater efficiency.[36]

3.4 Memory-Balanced Parallelism Strategies

Beyond the standard memory optimization techniques, recent research has focused on memory-balanced approaches to parallelism. One notable example is memory-balanced pipeline parallelism, which addresses a key limitation of standard pipeline parallelism: uneven memory distribution across pipeline stages.

In traditional pipeline parallelism, the memory consumption can vary significantly between stages depending on the layers assigned to each. This creates inefficiency, as the pipeline depth is constrained by the most memory-intensive stage. Memory-balanced pipeline strategies dynamically allocate layers to stages based on their memory footprint rather than simply dividing the model into equal numbers of layers. This results in more uniform memory utilization across devices and enables deeper pipelines, further reducing per-device memory requirements.

Similar balancing approaches are being explored for other parallelism strategies. For example, adaptive tensor parallelism techniques dynamically adjust the degree of tensor parallelism for different model components based on their memory and computation profiles. These advanced approaches represent the cutting edge of memory optimization research, promising even more efficient utilization of available hardware resources.

Optimization Technique	Targeted Memory Component	Memory Savings	Computational Overhead	Communication Overhead
Activation Checkpointing	Activations	Trades compute for memory; can be >50%	One extra forward pass per checkpointed block	None
ZeRO-1	Optimizer States	~4x	Negligible	None
ZeRO-2	Gradients + Optimizer States	~8x	Negligible	None

ZeRO-3	Parameters + Gradients + Optimizer States	Linear with DP degree (Nd)	Negligible	+50% vs. standard DP
ZeRO-Offload/Infinity	All model states (moved to CPU/NVMe)	Enables model size > aggregate GPU memory	Dependent on PCIe/NVMe bandwidth; overlapped with compute	Significant GPU-CPU traffic
Mixed Precision (FP16/BF16)	Parameters, Gradients, Activations	~2x	None; often faster on compatible hardware	Halve the communication volume

Table 2: Memory Optimization Techniques

4. Fault Tolerance and Elasticity in Dynamic Environments

While parallelism and memory optimization enable scale, their effectiveness assumes stable, reliable hardware environments. This assumption breaks down in modern cloud-native settings, where resource preemption and transient failures represent norms, not exceptions.[11] Consequently, resilience—encompassing both fault tolerance and elasticity—has become a first-class concern in distributed training system design. Fault tolerance ensures training jobs survive hardware or software failures, while elasticity enables jobs to dynamically adapt to changing available resource quantities. These capabilities prove essential for cost-effective training on ephemeral resources like spot instances.

4.1 Checkpointing and Recovery Mechanisms

Checkpointing—periodically saving training states to persistent storage—forms the fundamental fault tolerance mechanism. When failures occur, training can resume from last valid checkpoints, potentially saving weeks or months of computation.

4.1.1 Synchronous vs. Asynchronous Checkpointing

Checkpointing methods significantly impact training efficiency.

- Synchronous Checkpointing: This traditional approach globally pauses entire training processes. All workers halt computation, save states (model parameters, optimizer states, etc.) to distributed file systems, and only then resume training.[37] While simple and robust, this method introduces substantial downtime, as GPUs—the most expensive resources—remain idle during potentially slow I/O operations writing terabytes of data to storage. As model sizes grow, this overhead becomes prohibitively expensive, creating painful trade-offs between checkpoint frequency (safety) and training throughput (speed).[38]
- Asynchronous Checkpointing: To mitigate this overhead, modern systems employ asynchronous checkpointing. This technique decouples I/O-bound saving processes from compute-bound training loops. Common implementations involve two-phase processes: first, rapid, synchronous training state copies from GPU memory to pinned CPU memory. Once complete, GPUs freely resume the next training iterations. The second phase, writing states from CPU memory to persistent storage, proceeds in the background on separate I/O threads without stalling GPUs.[39][40]

The impact of asynchronous checkpointing is substantial and quantifiable. For a 175-billion parameter model, a synchronous checkpoint might pause training for 10-15 minutes while terabytes of data are written to storage. With asynchronous checkpointing, the visible pause is reduced to just 3-5 seconds for the memory copy phase, with the remaining I/O operations proceeding in the background without affecting training throughput. This represents a 95-98% reduction in checkpoint-related downtime. For a training run saving checkpoints every 6 hours over 20 days, this can translate to recovering nearly a full day of otherwise lost training time, a significant efficiency gain for expensive GPU resources.[40]

4.1.2 Stateless vs. Stateful Recovery Models

Recovery processes following failures can be designed differently.

- Stateless Recovery: In this model, cluster managers or schedulers treat training jobs as stateless entities. When nodes fail, entire jobs terminate. Systems then attempt relaunching jobs from scratch, acquiring new resource sets and loading most recent global checkpoints from persistent storage. This approach offers simple implementation but proves highly inefficient, involving complete distributed job teardowns and setups, which consume substantial time.[41][42]
- Stateful Recovery: More sophisticated approaches involve stateful orchestrators maintaining awareness of live training jobs. If workers fail, systems gracefully handle departures, procure replacement workers, and have them rejoin existing training groups. New workers then restore states, either from shared checkpoints or by receiving them from peers. This model enables much faster, more surgical recovery from individual node failures, avoiding complete job restart costs.[43][44]

4.2 Elastic Training Systems

Elasticity extends fault tolerance concepts from merely surviving failures to actively adapting to resource availability changes. This key capability enables cost-effective preemptible spot instance usage, as training jobs dynamically shrink or grow worker pools responding to preemptions and new resource availability.

4.2.1 Byzantine Fault Tolerance in Decentralized Training

As distributed training increasingly moves toward decentralized environments spanning multiple organizations or untrusted participants, Byzantine fault tolerance (BFT) becomes increasingly relevant. BFT addresses scenarios where participating nodes might not just fail but actively behave maliciously or unpredictably—sending incorrect gradients, manipulating training data, or attempting to inject backdoors into models.

Traditional distributed training systems assume benign failures rather than adversarial behavior. However, in future trustless training environments where compute resources might be contributed by multiple entities (similar to blockchain networks), robust protection against Byzantine failures becomes critical. Emerging research explores BFT-inspired aggregation algorithms that can detect and filter out suspicious gradient updates, enabling reliable training even when a portion of the participating nodes behaves arbitrarily. These approaches typically involve gradient validation mechanisms, robust aggregation schemes like geometric median instead of average, and reputation systems for worker nodes.

While still in early research stages, Byzantine-resilient training will likely become increasingly important as training moves beyond the boundaries of single organizations and into decentralized, collaborative environments.

4.2.2 Dynamic Reconfiguration and Job Morphing

Varuna stands out as a system designed from first principles for elastic training on commodity cloud infrastructure.[6] Its core innovation, "job morphing," dynamically and automatically reconfigures job parallelism strategies responding to available node quantity changes.[6]

When spot instances face preemption, rigid systems fail. Varuna, however, detects resource reductions and triggers reconfiguration. It employs fast, micro-benchmark-driven simulators determining new optimal hybrid parallelism strategies—for example, potentially decreasing data parallelism degrees while increasing pipeline depths to best utilize remaining nodes. Jobs then seamlessly transition to new configurations, resuming from last checkpoints.[6] This allows uninterrupted training continuation, albeit potentially at different throughput rates, thus maximizing progress and minimizing wasted resources.

However, dynamic reconfiguration faces significant engineering challenges that should not be underestimated. The performance overhead during reconfiguration periods can be substantial—in some implementations, requiring complete pipeline draining, checkpoint restoration, and rebalancing, which may take minutes to complete for large models. There's also the risk of training instability, as changing batch sizes or parallelism strategies mid-training can disrupt optimizer dynamics and potentially affect convergence. Early implementations showed convergence rate degradation of up to 5-10% in some scenarios compared to static configurations. Recent systems have improved on these issues, but they remain important considerations when implementing elastic training.

4.2.3 Auto-scaling and Failure Handling in Orchestration Frameworks

General-purpose distributed computing frameworks like Ray provide robust elastic training support.[45][46] Ray's architecture inherently embraces dynamism, built around actors (stateful worker processes) added or removed at runtime.

Ray Train, the framework's distributed deep learning library, leverages this foundation. Training jobs launch with specified worker ranges (e.g., minimums and maximums). Ray autoscalers manage underlying cluster resources, adding or removing nodes based on demand. If nodes fail or face preemption, Ray Train detects worker losses and continues training with remaining sets. When new resources become available, new workers launch and integrate into training jobs on-the-fly.[46][47] This provides flexible, resilient foundations for building elastic training applications.[48]

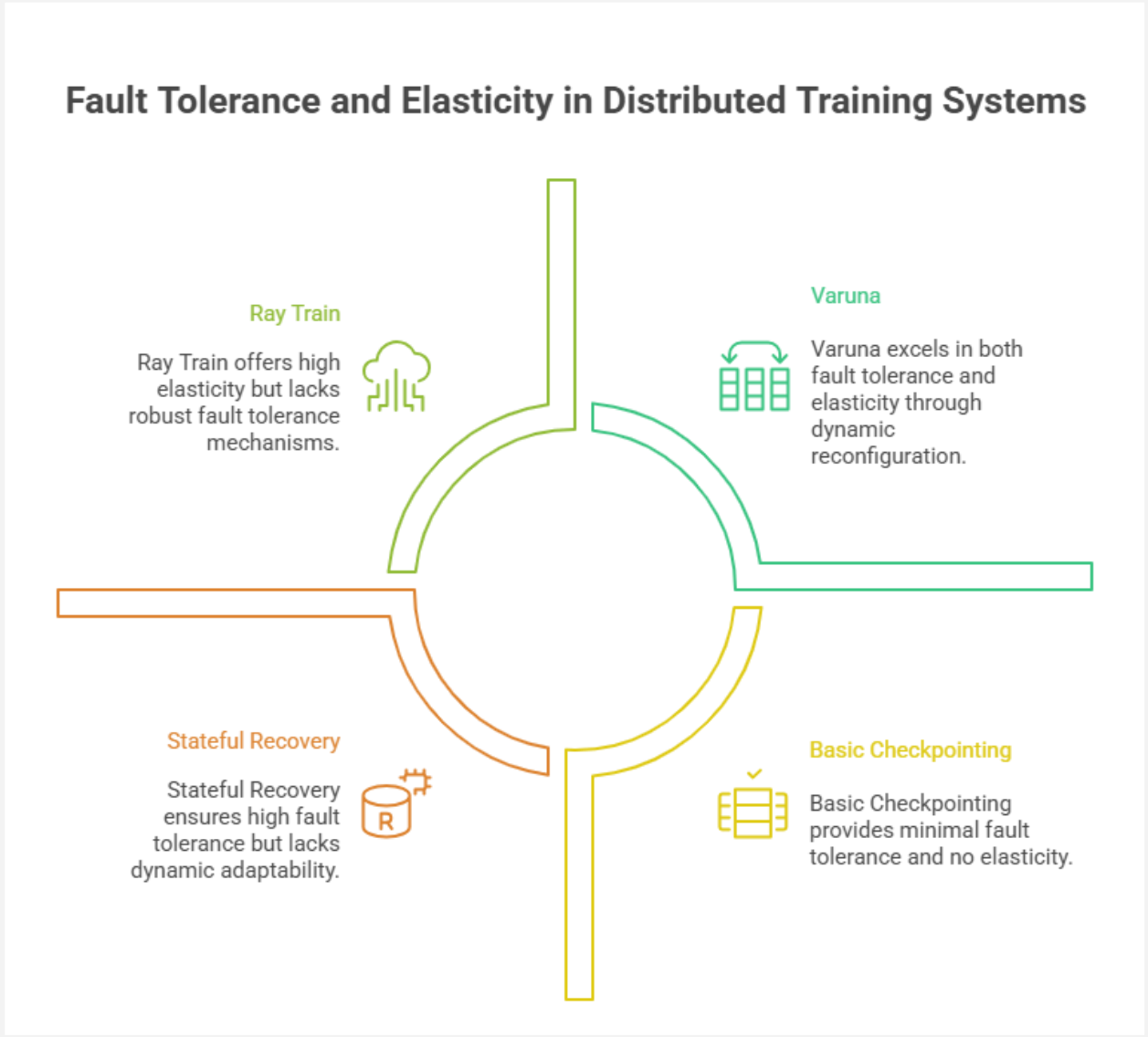

Fig 1: Fault Tolerance and Elasticity in Distributed LLM Training

5. Schedulers and Cluster Management

Moving up system stacks, cluster schedulers and management layers play crucial roles in orchestrating large-scale LLM training jobs. These systems handle resource allocation, job placement, and policy enforcement across multi-tenant clusters. Tightly-coupled, communication-intensive distributed training's unique requirements expose general-purpose scheduler limitations and drive specialized solution

development. Modern AI/ML workload schedulers must solve complex, multi-objective optimization problems, balancing job atomicity, network topology awareness, resource efficiency, and fairness.

5.1 Gang Scheduling for Tightly-Coupled Workloads

Distributed training frameworks like PyTorch DDP require all worker processes to be simultaneously active and communicating to make progress.[47] This "all-or-nothing" requirement fundamentally conflicts with default container orchestrator scheduling behaviors like Kubernetes. Native Kubernetes schedulers evaluate and place pods individually. If a distributed jobs request N pods but clusters only have N-1 pod resources, default schedulers place those N-1 pods. These pods start, consume valuable resources, but remain idle, indefinitely awaiting the Nth pod that cannot be scheduled. This wastes resources and, in worst cases, creates cluster-wide deadlocks.[49][50]

Gang scheduling addresses this by treating pod groups constituting single jobs as atomic units. Schedulers only commit to placing any pods if and only if sufficient resources exist for placing all pods simultaneously.[49] This prevents deadlocks and ensures resources avoid waste on partially-scheduled jobs. This functionality, absent from default Kubernetes schedulers, comes from specialized, pluggable schedulers designed for batch and HPC workloads.

Several widely-used industry tools implement gang scheduling in Kubernetes environments:

- Volcano: A CNCF incubating project that introduces PodGroup custom resources defining "gangs" of pods requiring co-scheduling. Volcano offers features specifically tailored for ML workloads, including fair-sharing, resource reservation, and queue management.[51][52]
- Kueue: Google's lightweight batch scheduling framework for Kubernetes which supports gang scheduling while focusing on simplicity and integration with standard Kubernetes resources.[53]
- YuniKorn: Apache's scheduler that provides fine-grained resource management with gang scheduling capabilities, often used in Hadoop/Spark environments transitioning to Kubernetes.[54]
- KubeDL: A Kubernetes-native job scheduling framework specifically designed for deep learning workloads with built-in gang scheduling support.[54]

Organizations deploying large-scale LLM training typically select one of these specialized schedulers to ensure efficient resource utilization and avoid the inefficiencies of partial job placement.

5.2 Topology-Aware Placement for Communication Optimization

For communication-intensive LLM training, pod placement locations matter as much as whether they get placed. Collective communication operation performance depends heavily on datacenter physical network topologies. Data transfers between GPUs in the same servers over NVLink run orders of magnitude faster than between GPUs in different racks connected through multiple network switches.[55]

Standard schedulers remain topology-agnostic; they see flat resource pools and may scatter single job pods across distant racks, severely degrading communication performance and overall training throughput. Topology-aware scheduling addresses this by endowing schedulers with physical network hierarchy knowledge (nodes, racks, spines, etc.). When placing jobs, schedulers use this information to co-locate pods requiring frequent communication, such as those within the same tensor-parallel or data-parallel groups, onto physically proximate nodes.[55][56]

To illustrate the critical importance of topology-aware placement, consider a concrete example: an 8-way tensor-parallel job running on 8 GPUs. When all 8 GPUs are located within the same server node connected via NVLink (which provides ~600 GB/s bidirectional bandwidth), the frequent All-Reduce operations required for tensor parallelism might take just 5-10 milliseconds. However, if the scheduler naively places these 8 GPUs across different server racks connected via standard datacenter networking (10-100 Gb/s), these same operations could take 500-1000 milliseconds—a 100x performance degradation. In a training run with thousands of iterations, this poor placement could extend training time from days to months, or make training practically infeasible.

Google's Topology Aware Scheduling (TAS), often paired with Kueue schedulers, strategically places workers minimizing network hops between them, reducing contention and optimizing bandwidth utilization.[56][57] This intelligent placement significantly improves training times and efficiency. Some advanced systems even incorporate rack-level awareness, prioritizing placement within the same power domain to minimize vulnerability to power-related failures that might affect entire racks simultaneously.

5.3 Integrated Orchestration Systems

The next cluster management frontier involves moving beyond separate scheduling and execution systems toward tightly integrated frameworks automating entire processes, from model compilation to parallel execution.

- Pathways (Google): Pathways represents a new large-scale orchestration architecture. Built on asynchronous distributed dataflow models, computations appear as operator graphs consuming and producing futures.[59][60] Its single-controller design provides global cluster views, enabling efficient gang-scheduling of complex, heterogeneous parallelism patterns across thousands of accelerators. Pathways orchestrates computations spanning multiple TPU "pods," managing not just computation placement but also data transfer coordination over dedicated interconnects, enabling novel parallelism scheme research.[59][60]
- Alpa (Berkeley): Alpa approaches problems from a compiler perspective. This system automates optimal parallelization strategy discovery for given models and clusters.[61][62] Alpa introduces hierarchical parallelism views, distinguishing between inter-operator parallelism (like pipelining, partitioning operator graphs) and intra-operator parallelism (like tensor parallelism, partitioning operators themselves). It combines dynamic programming and integer linear programming (ILP) solvers searching vast, hierarchical spaces to find optimal model partition methods and device mesh mappings, effectively unifying and automating complex hybrid parallelism strategy design tasks.[61][62]

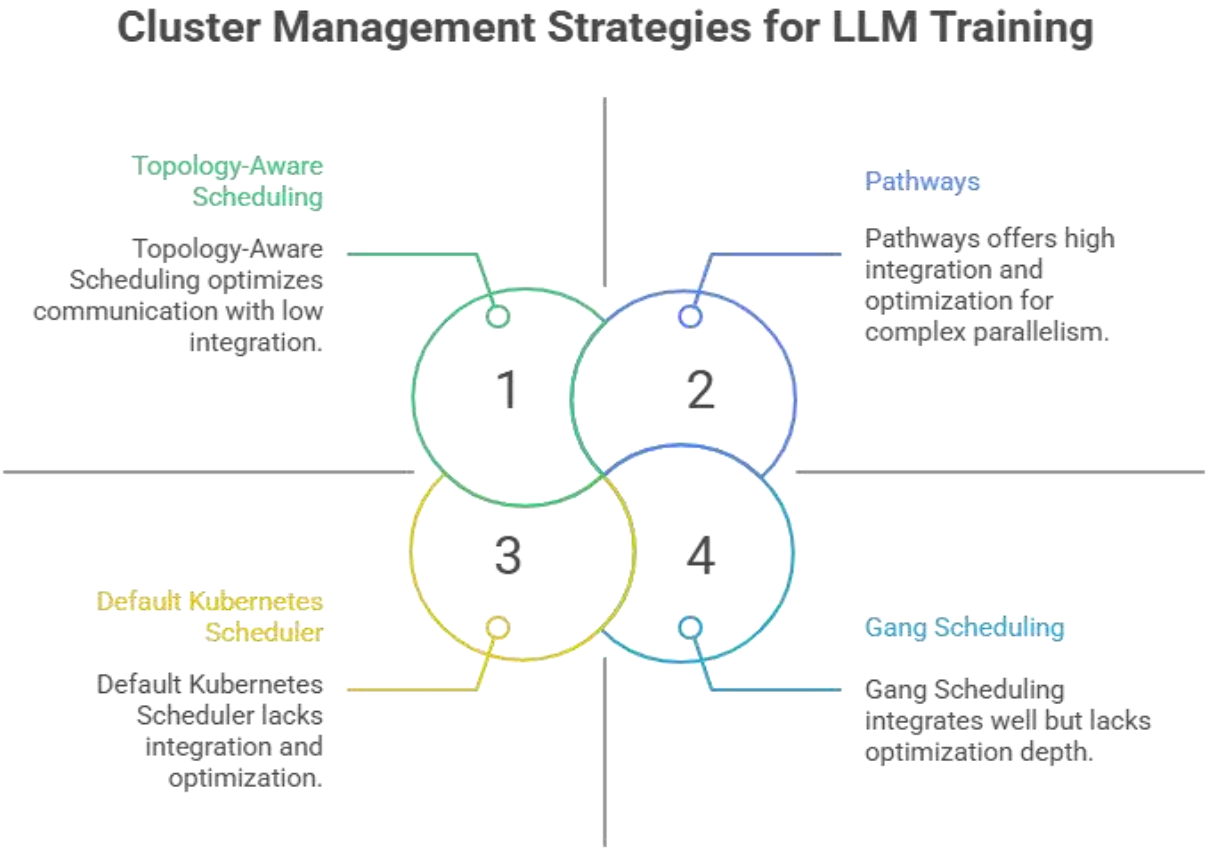

Fig 2:Cluster Management Strategies for LLM Training

6. Challenges and Future Research Directions

Despite remarkable scalable and resilient LLM training progress, significant challenges persist, indicating fertile ground for future research. Solutions to these problems will prove critical for democratizing large-scale training access and advancing AI frontiers sustainably and efficiently.

6.1 Co-design of Algorithms, Systems, and Schedulers for Elasticity

Most current systems treat machine learning algorithms as fixed workloads for execution. Resilience happens at system levels through checkpoint/restart mechanisms and dynamic reconfiguration, but training algorithms themselves remain largely unaware of changing resource landscapes. A promising future direction involves deep algorithm-system co-design. This might include:

- Gradient-staleness-robust optimizers: Developing optimization algorithms specifically designed to handle the gradient staleness that occurs during asynchronous recovery and elastic reconfiguration. These could incorporate techniques like staleness-aware learning rate adjustment, where the optimizer automatically reduces learning rates proportionally to detected staleness levels.
- Resource-aware adaptive batch sizing: Creating algorithms that automatically adjust batch sizes in response to resource changes while preserving convergence properties through careful learning rate scaling and gradient accumulation techniques.
- Elastic consistency models: Developing training frameworks that can dynamically shift between synchronous and asynchronous training paradigms based on available resources and network conditions, with theoretical guarantees on convergence behavior.
- Failure-anticipating pre-emptive checkpointing: Building predictive models that can anticipate failures before they occur (using system telemetry) and trigger targeted checkpoints only for endangered model components.

Such approaches would shift from reactive resilience (recovering from changes) toward proactive adaptation (algorithmically anticipating and adjusting to changes). The key breakthrough would be training algorithms that treat elasticity and resource variability as first-class concerns rather than exceptional conditions.

6.2 Automating the Parallelism Strategy Search Space

Hybrid parallelism strategy design spaces grow combinatorially vast. Choosing optimal data, tensor, pipeline, and sequence parallelism combinations, along with corresponding parallelism degrees and device mesh configurations, presents daunting tasks highly dependent on both model architectures and specific hardware clusters. While compiler-based systems like Alpa have significantly advanced this search automation, developing more scalable, efficient, and generalizable search algorithms remains a critical open problem.

Several promising approaches for navigating this complex search space are emerging:

- Reinforcement Learning (RL): Using RL agents to explore the parallelism strategy space by learning from execution traces and performance feedback. Systems like AutoPipe are beginning to demonstrate how RL can efficiently discover parallelism strategies that outperform hand-tuned configurations, particularly for novel model architectures.
- Performance Modeling: Developing analytical models that can predict distributed training performance without actually executing the full workload. These models combine theoretical communication and computation cost analysis with empirical measurements from small-scale profiling runs to rapidly evaluate thousands of potential configurations.
- Genetic Algorithms and Evolutionary Approaches: Using population-based optimization techniques that "evolve" parallelism strategies through mutation and selection based on actual performance measurements or model predictions.
- Transfer Learning for Configuration: Leveraging knowledge from previously optimized models to accelerate the search for new, similar architectures—effectively "warm starting" the optimization process rather than beginning from scratch.

Future systems must quickly find near-optimal plans for novel model architectures and highly heterogeneous hardware environments, perhaps using machine learning-based performance models guiding searches.[61][62] The ultimate goal is a system that can automatically adapt its parallelization strategy as models evolve during development and as available hardware resources change over time.

6.3 Standardizing Resilience Benchmarking

Currently, no widely accepted benchmarks or metric sets exist evaluating distributed training system resilience and elasticity. Papers typically report throughput on fixed GPU quantities, failing to capture performance in dynamic, real-world cloud environments. Standardized benchmarking suites allowing fair, rigorous system comparisons are needed.

The article propose the development of a composite "Resilience Score" (RS) that would combine multiple metrics into a single, comparable value. This score would incorporate:

- Recovery Point Objective (RPO): Maximum work lost upon failure, determined by checkpointing frequency and overhead.[37][39] (Measured in training iterations or wall-clock time)
- Recovery Time Objective (RTO): Time required to detect failures, acquire new resources, and resume training.[40] (Measured in seconds or minutes)
- Elasticity Overhead: Performance penalties or time required to reconfigure jobs when scaling up or down.[6][47] (Measured as a percentage of normal training throughput)
- Performance under Jitter: Throughput degradation when subjected to simulated network latency and variability.[8][9] (Measured as a percentage of ideal throughput)
- Preemption Resilience: The ability to maintain training progress under various preemption rates and patterns. (Measured through specialized benchmark scenarios)

A weighted formula combining these metrics could produce a single RS value, where higher scores indicate more resilient systems. For example:

$RS = w_1(1/RPO) + w_2(1/RTO) + w_3(1/ElasticityOverhead) + w_4(PerformanceUnderJitter) + w_5(PreemptionResilience)$

Where w_1 through w_5 are weighting factors that can be adjusted based on specific deployment priorities.

This standardized approach would allow researchers and practitioners to make informed decisions about which distributed training systems best meet their resilience requirements and enable meaningful comparisons across different implementations.

6.4 Energy-Aware and Sustainable Training

The environmental and economic costs of training state-of-the-art LLMs raise growing concerns. Single training run energy consumption can equal hundreds of households' annual usage. To date, system design has overwhelmingly prioritized minimizing time-to-solution. A critical future direction involves incorporating energy efficiency as a first-class optimization objective. This might manifest in several ways:

- Energy-Aware Schedulers: Schedulers making placement decisions based not only on performance but also power consumption, potentially consolidating workloads, allowing idle node power-down.[50][53]
- Power-Capping-Aware Training: Systems operating efficiently under dynamic power caps, adjusting computation and communication, staying within given power budgets.[37][38]
- Algorithmic Efficiency: Developing fundamentally more compute-efficient model architectures and training approaches:
    - Sparse Models: Leveraging techniques like conditional computation, mixture-of-experts, and dynamic network pruning to activate only relevant parts of models during training and inference
    - Efficient Architectures: Designing inherently more parameter-efficient architectures that achieve comparable quality with fewer FLOPs
    - Sample-Efficient Training: Creating algorithms that learn from fewer examples, reducing the total computation needed for convergence
    - Knowledge Distillation: Training smaller, efficient models by transferring knowledge from larger pre-trained models
- Carbon-Aware Scheduling: Systems that schedule computation based on carbon intensity of electricity grids, prioritizing training during periods of abundant renewable energy availability.
- Hardware-Software Co-Optimization: Designing specialized hardware accelerators in concert with algorithms that exploit their efficiency characteristics.

The focus on sustainability extends beyond just reducing power consumption—it encompasses the entire lifecycle of model development, from initial training through deployment and ongoing maintenance. Truly sustainable AI will require innovations across all these dimensions, with particular emphasis on the often-overlooked algorithmic efficiency that can provide order-of-magnitude improvements in energy utilization.

Conclusion

The journey to train ever-larger language models has catalyzed a rapid and profound evolution in distributed computing. This survey has charted this evolution, moving from the initial challenge of pure computational scale to the more nuanced, multi-faceted problem of achieving scalable, efficient, and resilient training in dynamic, cloud-native environments. The article has structured this landscape into four interdependent pillars—parallelism, memory optimization, fault tolerance, and cluster management—demonstrating how innovations in each area have been driven by the limitations of the last. The overarching trend is a move away from static, brute-force scaling and towards intelligent, adaptive, and automated systems. The future of LLM training lies not in simply adding more hardware but in the co-design of algorithms and systems that can navigate the complex trade-offs between performance, memory, cost, and resilience. The open challenges identified—from automated parallelism search to standardized resilience benchmarking and energy-aware training—highlight that this field remains a vibrant and critical area of research. The continued progress in this domain will be essential for unlocking the next generation of artificial intelligence in a scalable, robust, and sustainable manner.

References

[1] Rajesh Kumar, Isabelle Laurent, David Müller, and Klaus Elli, "Multimodal and Distributed LLMs: Bridging Scalability and Cross-Modal Reasoning," Preprints, 2025. [Online]. Available: https://www.preprints.org/manuscript/202505.1156/v1

[2] Jiangfei Duan et al., "Efficient Training of Large Language Models on Distributed Infrastructures: A Survey," arXiv:2407.20018v1, 2024. [Online]. Available: https://arxiv.org/html/2407.20018v1

[3] Zhongwei Wan et al., "Efficient Large Language Models: A Survey," arXiv:2312.03863v4. [Online]. Available: https://arxiv.org/html/2312.03863v4

[4] Samyam Rajbhandari et al., "ZeRO: Memory Optimization Towards Training A Trillion Parameter Models," ResearchGate, 2019. [Online]. Available: https://www.researchgate.net/publication/336304157_ZeRO_Memory_Optimization_Towards_Training_A_Trillion_Parameter_Models

[5] Mohammad Shoeybi et al., "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," arXiv:1909.08053, 2020. [Online]. Available: https://arxiv.org/abs/1909.08053

[6] Sanjith Athlur et al., "Varuna: Scalable, Low-cost Training of Massive Deep Learning Models," Microsoft Research, 2022. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2022/03/varuna-eurosys22.pdf

[7] Shashank Prasanna, "Train Deep Learning Models on GPUs using Amazon EC2 Spot Instances," Amazon Web Services, 2019. [Online]. Available: https://aws.amazon.com/blogs/machine-learning/train-deep-learning-models-on-gpus-using-amazon-ec2-spot-instances/

[8] Haotian Dong et al., "Beyond A Single AI Cluster: A Survey of Decentralized LLM Training," arXiv:2503.11023v1, 2025. [Online]. Available: https://arxiv.org/html/2503.11023v1

[9] Yanping Huang et al., "GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism," arXiv:1811.06965v5, 2019. [Online]. Available: https://arxiv.org/pdf/1811.06965

[10] Hugging Face, "Megatron-LM,". [Online]. Available: https://huggingface.co/docs/accelerate/usage_guides/megatron_lm

[11] Deepak Narayanan et al., "PipeDream: Generalized Pipeline Parallelism for DNN Training," 2019. [Online]. Available: https://people.eecs.berkeley.edu/~matei/papers/2019/sosp_pipedream.pdf

[12] Papers With Code, "GitHub Repository,". [Online]. Available: https://github.com/paperswithcode

[13] Yanping Huang et al., "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," 33rd Conference on Neural Information Processing Systems, 2019. [Online]. Available: https://fid3024.github.io/papers/2019%20-%20GPipe:%20Efficient%20Training%20of%20Giant%20Neural%20Networks%20using%20Pipeline%20Parallelism.pdf

[14] Houming Wu, Ling Chen, and Wenjie Yu, "BitPipe: Bidirectional Interleaved Pipeline Para," arXiv:2410.19367v1, 2024. [Online]. Available: https://arxiv.org/pdf/2410.19367
[15] Changjiang Gou, "A review of Pipeline Parallel Training of Large-scale Neural Network," SlideShare, 2022. [Online]. Available: https://www.slideshare.net/slideshow/a-review-of-pipeline-parallel-training-of-largescale-neural-networkpdf/255335332
[16] Deepak Narayanan et al., "Memory-Efficient Pipeline-Parallel DNN Training," in Proceedings of the 38th International Conference on Machine
Learning, 2021. [Online]. Available: https://proceedings.mlr.press/v139/narayanan21a/narayanan21a.pdf
[17] SERP AI, "PipeMare,". [Online]. Available: https://serp.ai/posts/pipemare/
[18] Haosheng Zou et al., "360-LLaMA-Factory: Plug & Play Sequence Parallelism for Long Post-Training," arXiv:2505.22296v1, 2025. [Online]. Available: https://arxiv.org/pdf/2505.22296
[19] Haosheng Zou et al., "360-LLaMA-Factory: Plug & Play Sequence Parallelism for Long Post-Training," arXiv:2505.22296v1, 2025. [Online]. Available: https://arxiv.org/html/2505.22296v1
[20] Yujie Wang et al., "Data-Centric and Heterogeneity-Adaptive Sequence Parallelism for Efficient LLM Training," arXiv:2412.01523v1, 2024. [Online]. Available: https://arxiv.org/html/2412.01523v1
[21] Jiarui Fang et al., "A Unified Sequence Parallelism Approach for Long Context Generative AI," arXiv:2405.07719v3, 2024. [Online]. Available: https://arxiv.org/html/2405.07719v3
[22] Vijay Korthikanti et al., "Reducing Activation Recomputation in Large Transformer Models," arXiv:2205.05198, 2022. [Online]. Available: https://arxiv.org/abs/2205.05198
[23] DeepSpeed, "Getting Started with DeepSpeed-Ulysses for Training Transformer Models with Extreme Long Sequences," July 30, 2025. [Online]. Available: https://www.deepspeed.ai/tutorials/ds-sequence/
[24] DeepSpeed, "Latest News,". [Online]. Available: https://www.deepspeed.ai/
[25] DeepSpeed, "Training Overview and Features,". [Online]. Available: https://www.deepspeed.ai/training/
[26] Hey Amit, "PyTorch Activation Checkpointing: Complete Guide," Medium, 2024. [Online]. Available: https://medium.com/@heyamit10/pytorch-activation-checkpointing-complete-guide-58d4f3b15a3d
[27] PyTorch, "Current and New Activation Checkpointing Techniques in PyTorch," 2025. [Online]. Available: https://pytorch.org/blog/activation-checkpointing-techniques/
[28] DeepSpeed, "Activation Checkpointing,". [Online]. Available:
https://deepspeed.readthedocs.io/en/latest/activation-checkpointing.html
[29] Github Repository, "DeepSpeed,". [Online]. Available: https://github.com/deepspeedai/DeepSpeed
[30] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "DeepSpeed: System Optimizations Enable Training Deep-Learning Models with Over 100 billion Parameters," NVIDIA. [Online]. Available: https://www.nvidia.com/en-us/on-demand/session/gtcsj20-s22646/
[31] Jie Ren et al., "ZeRO-Offload: Democratizing Billion-Scale Model Training," arXiv:2101.06840, 2021. [Online]. Available: https://arxiv.org/abs/2101.06840
[32] Samyam Rajbhandari et al., "ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning," arXiv, 2021. [Online]. Available: https://www.microsoft.com/en-us/research/publication/zero-infinity-breaking-the-gpu-memory-wall-for-extreme-scale-deep-learning/
[33] TutorialsPoint, "DeepSpeed Mixed Precision Training,". [Online]. Available:
https://www.tutorialspoint.com/deepspeed/deepspeed-mixed-precision-training.htm
[34] DeepSpeed Team, "Getting Started with DeepSpeed," GitHub Documentation. [Online]. Available: https://github.com/deepspeedai/DeepSpeed/blob/master/docs/_tutorials/getting-started.md
[35] Lionel Dong, "Deepspeed GPT training tutorial: Mastering large language model optimization," BytePlus Technical Articles, 2025. [Online]. Available: https://www.byteplus.com/en/topic/499160?title=deepspeed-gpt-training-tutorial-mastering-large-language-model-optimization
[36] DeepSpeed, "Mixed Precision ZeRO++," 2025. [Online]. Available:
https://www.deepspeed.ai/tutorials/mixed_precision_zeropp/
[37] Grant Wilkins et al., “Analyzing the Energy Consumption of Synchronous
and Asynchronous Checkpointing Strategies,”https://par.nsf.gov/servlets/purl/10404060
[38] Tonmoy Dey et al., “Optimizing Asynchronous Multi-Level Checkpoint/Restart Configurations with Machine Learning,”. https://www.osti.gov/servlets/purl/1770373
[39] PyTorch, “Asynchronous Saving with Distributed Checkpoint (DCP),” 2024.
https://docs.pytorch.org/tutorials/recipes/distributed_async_checkpoint_recipe.html
[40] Lucas Pasqualin et al., “Reducing Model Checkpointing Times by Over 10x with PyTorch Distributed Asynchronous Checkpointing,” PyTorch, 2024. https://docs.pytorch.org/blog/reducing-checkpointing-times/
[41] Sebastian Raschka, “Machine Learning FAQ,”. https://sebastianraschka.com/faq/docs/stateless-stateful.html

[42] System Design School, “Understanding Stateful vs Stateless in Software Engineering,”. https://systemdesignschool.io/blog/stateful-vs-stateless
[43] GeeksforGeeks, “Stateless and Stateful Systems in System Design,” 2025. https://www.geeksforgeeks.org/system-design/stateless-and-stateful-systems-in-system-design/
[44] Pure Storage, “Stateful vs. Stateless Applications: What’s the Difference?” 2024. https://blog.purestorage.com/purely-educational/stateful-vs-stateless-applications-whats-the-difference/
[45] Matthew Deng, Amog Kamsetty, Richard Liaw, and Will Drevo, “Distributed deep learning with Ray Train is now in Beta,” Anyscale, 2022. https://www.anyscale.com/blog/distributed-deep-learning-with-ray-train-is-now-in-beta
[46] Anyscale, “Ray Train and RayTurbo Train,”. https://docs.anyscale.com/rayturbo/rayturbo-train/
[47] Haoran Lin et al., “Ray-based Elastic Distributed Data Parallel Framework with Distributed Data Cache,” ResearchGate, 2023. https://www.researchgate.net/publication/372931330_Ray-based_Elastic_Distributed_Data_Parallel_Framework_with_Distributed_Data_Cache
[48] Uber, “Elastic Distributed Training with XGBoost on Ray,” 2021. https://www.uber.com/en-IN/blog/elastic-xgboost-ray/
[49] Bibin Wilson, “Gang Scheduling in Kubernetes,” Techiescamp, 2024. https://blog.techiescamp.com/gang-scheduling-in-kubernetes/
[50] Run.AI, “Problem Aware, MLOps - Kubernetes Scheduling for AI,”. https://pages.run.ai/hubfs/PDFs/White%20Papers/Kubernetes%20Scheduling%20for%20AI.pdf
[51] AlibabaCloud, “Work with gang scheduling,” 2024. https://www.alibabacloud.com/help/en/ack/ack-managed-and-ack-dedicated/user-guide/work-with-gang-scheduling
[52] Volcano, “Plugins,”. https://volcano.sh/en/docs/plugins/
[53] Rahul Kadam, “Batch Scheduling on Kubernetes: Comparing Apache YuniKorn, Volcano.sh, and Kueue,” Infracloud, 2025. https://www.infracloud.io/blogs/batch-scheduling-on-kubernetes/
[54] KubeDL, “Gang Scheduling,”. https://kubedl.io/docs/training/gangscheduling/
[55] OpenReview, “Efficient Pre-Training of LLMs via Topology-Aware Communication Alignment on More Than 9600 GPUs,”. https://openreview.net/pdf?id=oOGD6GwafB
[56] Google Cloud, “Schedule GKE workloads with Topology Aware Scheduling,”. https://cloud.google.com/ai-hypercomputer/docs/workloads/schedule-gke-workloads-tas
[57] Kueue, “Topology Aware Scheduling,”. https://kueue.sigs.k8s.io/docs/concepts/topology_aware_scheduling/
[58] Volcano, “Network Topology Aware Scheduling,” 2025. https://volcano.sh/en/docs/network_topology_aware_scheduling/
[59] Paul Barham et al., “Pathways: Asynchronous Distributed Dataflow For ML,” arXiv:2203.12533v1, 2022. https://arxiv.org/pdf/2203.12533
[60] TensorFlow, “Architecture,”. https://www.tensorflow.org/tfx/serving/architecture
[61] Lianmin Zheng et al., “Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning,” ResearchGate, 2022. https://www.researchgate.net/publication/358233150_Alpa_Automating_Inter-_and_Intra-Operator_Parallelism_for_Distributed_Deep_Learning
[62] Lianmin Zheng et al., "Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning," OSDI, 2022. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin