NTSM: OS-Level Memory Middleware for Scaling AI Without Boundaries

Anonymous Author(s)
Submission Id: 103

Abstract

The rapid scaling of large language models—from hundreds of billions to trillions of parameters—has rendered distributed memory management the critical bottleneck in modern AI systems. Existing frameworks manage cross-node memory entirely in user space, incurring serialization overhead that consumes up to 76.7% of data transfer latency, memory amplification reaching up to 6.7× from intermediate buffers, and escalating coordination costs that compound with cluster size. These overheads are fundamental: commodity operating systems lack abstractions for TB-scale tensors, cross-node sharing, and the sequential, layer-granular access patterns characteristic of transformer workloads.

We present NTSM, a kernel-space middleware that recasts distributed memory management as a first-class OS primitive. Implemented as a Linux kernel module with a minimal kernel modification (one exported symbol on x86), NTSM exposes a symmetric virtual address space across cluster nodes where applications access remote data through ordinary memory operations. Page faults trigger zero-copy RDMA transfers with automatic coherence enforcement via a directory-based M/S/I protocol operating at 32KB–2MB Virtual Memory Region granularity—matching the natural access grain of transformer layers while amortizing fault overhead. By executing in kernel space, NTSM directly manipulates page tables, pins memory for DMA, and manages TLB state, eliminating the serialization, buffer copies, and user-kernel transitions inherent to user-space approaches.

NTSM integrates transparently with existing frameworks: 152 lines modified for PyTorch FSDP, 89 lines for Ray Parameter Server. Evaluation on a 34-node RDMA cluster with models up to 175B parameters demonstrates up to 6.5× speedup in distributed training memory operations, 4.4× reduction in time-to-first-token for inference, and 47% reduction in parameter server iteration time—establishing kernel-level memory management as a viable foundation for next-generation distributed AI infrastructure.

CCS Concepts

• **Computer systems organization** → **Distributed architectures**; **Distributed systems organizing principles**; • **Computing methodologies** → *Neural networks.*

Keywords

distributed shared memory, kernel-space optimization, machine learning systems, distributed training, distributed inference

1 Introduction

The explosive growth of large language models (LLMs)—from GPT-3's 175 billion parameters [6] to DeepSeek-V3's 671 billion [10] and trillion-parameter Mixture-of-Experts architectures [13]—has created a fundamental mismatch between model memory requirements and available hardware resources. A single 175B model requires 350GB–1TB for training; emerging trillion-parameter models exceed 3TB. No single accelerator, GPU or otherwise, can accommodate these memory footprints. Consequently, distributed memory management has become the critical bottleneck for both training and inference at scale.

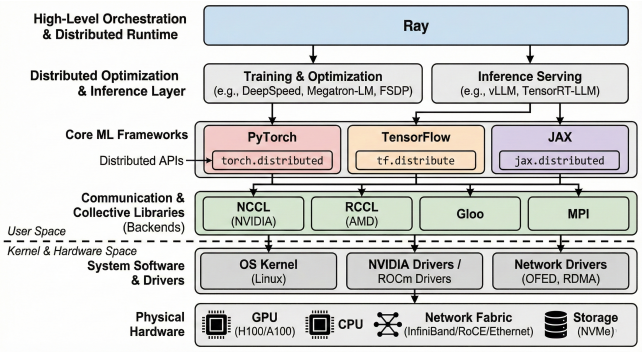


Figure 1: Software Stack for Distributed AI

Modern distributed AI systems employ a tiered memory hierarchy. Tier 1 comprises high-bandwidth accelerator memory (GPU HBM, TPU SRAM), while Tier 2 encompasses host CPU memory pooled across cluster nodes. As illustrated in Figure 1, existing frameworks—DeepSpeed [1], PyTorch FSDP [45], Megatron-LM [35], and Ray [25]—manage Tier 2 memory entirely in user space, relying on collective APIs (AllGather, AllReduce) and explicit serialization for cross-node data movement. This design imposes substantial overhead: serialization consumes up to 76.7% of data transfer latency, memory amplification reaches up to 6.7× due to intermediate buffers, and scalability degrades as coordination costs compound with cluster size.

The root cause of these inefficiencies lies deeper than framework design: *the operating system is fundamentally blind to AI workloads.* Commodity OSes manage memory through generic abstractions—4KB pages, LRU eviction, demand paging optimized for random access [19, 38]—that bear no relationship to how AI systems actually consume memory. Transformer models access parameters sequentially at layer granularity (400MB–2GB per layer), exhibit predictable read-heavy patterns during inference, and require TB-scale shared state across nodes. The OS provides none of the primitives these patterns demand: no native support for large contiguous allocations, no awareness of cross-node sharing, no coherence for distributed memory, and no optimization for RDMA-based data movement.

Forced to work around an oblivious OS, AI frameworks have constructed elaborate user-space memory managers. DeepSpeed ZeRO-3 and FSDP partition model state across CPU memory when GPU capacity is exhausted, implementing their own sharding, prefetching, and collective communication. Ray builds an entire distributed object store (Plasma) with custom serialization and reference counting. Each framework duplicates this infrastructure independently, yet all share the same fundamental constraint: user-space code cannot manipulate page tables, pin memory for DMA, or control TLB state. Every cross-node access requires serialization, buffer copies, system calls, and user-kernel transitions—overhead that scales with cluster size and model complexity.

This raises a fundamental question: **What new OS primitives and mechanisms are needed to efficiently support the memory demands of modern AI workloads?** What if the OS exposed interfaces that recognize AI workloads access memory in large, predictable chunks? What if distributed memory required no more copies than local memory? What if cross-node sharing were as transparent as accessing local DRAM?

We present NTSM, an in-kernel distributed shared memory system that answers these questions. NTSM operates primarily as a loadable Linux kernel module, requiring only a single-line kernel modification to export one TLB management function on x86. It interposes between AI frameworks and cluster hardware, exposing a symmetric shared address space where applications access remote data through ordinary memory operations; page faults trigger zero-copy RDMA transfers with automatic coherence enforcement. By moving memory management into the kernel, NTSM eliminates serialization entirely, removes user-kernel transition overhead, and replaces explicit collective communication with demand-driven data movement.

NTSM bridges the gap between OS capabilities and AI requirements through four key mechanisms. First, kernel-space execution enables direct page table manipulation, DMA pinning, and TLB management—operations that user-space frameworks must approximate through costly system calls. Second, symmetric virtual addressing preserves pointer validity across nodes, enabling pointer-rich data structures (optimizer states, routing tables) to function without modification. Third, Virtual Memory Regions (VMRs) at 32KB–2MB granularity match AI access patterns, amortizing fault overhead while bounding transfer size. Fourth, an M/S/I directory protocol provides automatic coherence with minimal overhead for read-dominated inference workloads.

NTSM is designed as infrastructure—a Tier 2 memory substrate that complements rather than replaces existing frameworks. It integrates with PyTorch FSDP (152 lines modified) and Ray Parameter Server (89 lines modified), transparently accelerating their distributed memory operations without requiring applications to adopt new APIs or rewrite communication logic. Whether the primary computation occurs on GPUs with CPU memory offloading, on CPUs with matrix extensions, or in hybrid configurations, NTSM provides efficient cross-node memory sharing below the framework layer.

Our contributions include:

(1) We provide the first comprehensive analysis of memory management overhead in distributed AI frameworks, identifying serialization, memory amplification, and user-kernel transitions as fundamental bottlenecks that scale with cluster size.
(2) We design and implement NTSM, a kernel-level distributed shared memory system that provides transparent memory pooling through demand paging, zero-copy RDMA, and automatic coherence at AI-appropriate granularity (32KB–2MB VMRs).
(3) We demonstrate that NTSM integrates with existing frameworks (PyTorch FSDP, Ray) with minimal modification (<250 lines total), achieving up to 6.5× speedup in distributed training and 4.4× reduction in time-to-first-token.

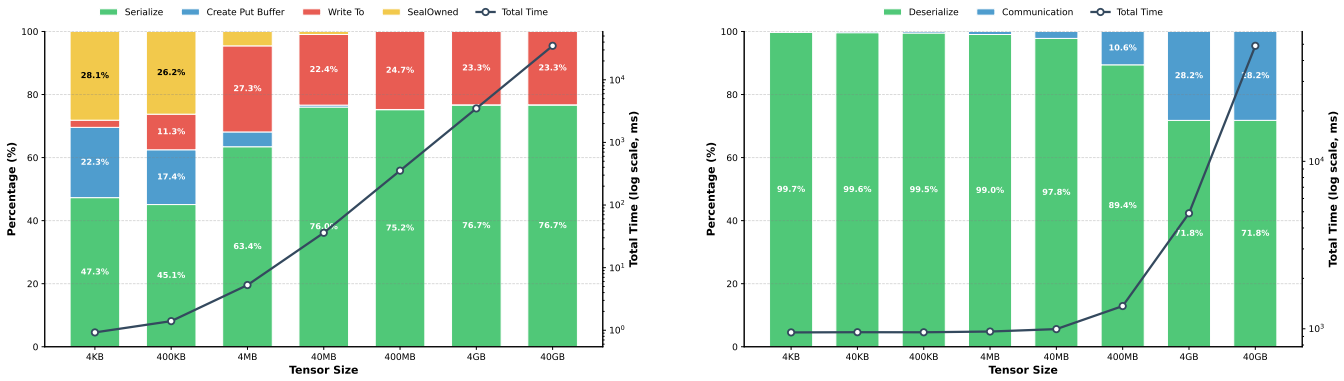
The remainder of this paper is organized as follows: Section 2 provides background and analyzes memory management overhead in existing frameworks. Section 3 presents NTSM's architecture. Section 4 details our implementation and framework integration. Section 5 quantifies NTSM's performance on large-scale distributed AI workloads. Section 6 discusses related work. Section 7 concludes the paper.

2 Background and Motivation

The distributed nature of deep learning workloads presents unique challenges compared to traditional distributed computing. During training, workers must frequently synchronize large volumes of data – model parameters, gradients, and activation states – while maintaining strict computational dependencies between forward and backward passes. Unlike traditional distributed applications where computation and communication patterns are often predictable, deep learning workloads exhibit dynamic memory access patterns and varying communication requirements across different phases of training. Moreover, the performance impact of these communication patterns becomes more pronounced as model sizes grow, since both the volume of data movement and the frequency of synchronization increase with model scale. This has spurred the development of several parallel training strategies, each attempting to optimize different aspects of the distributed training process.

2.1 Parallel Training & Inference Strategies

Modern distributed deep learning systems employ several parallelization strategies, each with distinct trade-offs in memory efficiency, communication overhead, and implementation complexity. **Data Parallelism** [23] replicates the entire model across workers while partitioning input data, requiring parameter synchronization only during gradient averaging. While this approach minimizes communication frequency and simplifies implementation, it maintains full model state replication, limiting its applicability when model size exceeds single-device memory. **Model Parallelism** [35] addresses this limitation by partitioning model parameters across devices, but introduces frequent cross-device communication for activations and gradients during both forward and backward passes. **Pipeline Parallelism** [15] attempts to balance these trade-offs by dividing the model into sequential stages, enabling concurrent computation across stages but potentially introducing pipeline bubbles and complexity in managing activation checkpoints. More recently,



(a) Put operation **(b) Get operation**

Figure 2: Ray's operation breakdown

Tensor Parallelism [35] has emerged to handle large individual layers by distributing tensor operations across devices, though this requires careful orchestration of partial results and can increase implementation complexity.

2.2 Distributed Training & Inference Frameworks

Distributed deep learning systems operate across multiple software layers, each addressing different aspects of distribution. At the core ML framework level, PyTorch's Distributed Data Parallel (DDP) [23] implements data parallelism using an AllReduce-based approach where each worker maintains a complete model copy. To address DDP's memory limitations, PyTorch introduced Fully Sharded Data Parallel (FSDP) [45], which shards model parameters across processes and dynamically reconstructs them using All-Gather operations during computation. While these optimizations improve memory efficiency, they still operate entirely in user space, requiring data serialization and memory copies between user and kernel space. At a higher level, distributed computing frameworks like Ray [25] provide cluster management and resource orchestration. Ray's architecture includes a distributed object store (Plasma) for sharing tensors across nodes and a task scheduling system for distributed computation. However, its user-space implementation necessitates additional serialization and memory management overhead, which is particularly challenging for large-scale DL workloads. Ray can also support Parameter Server (PS) architecture for memory-intensive processes. In this architecture, computation is handled by worker nodes, while storage nodes (referred to as parameter servers) are responsible for storing and updating the model parameters, which are partitioned across the servers and work as the shared memory space in the system [21] [20]. PS allows elastic scalability and address the issues of node failure and data loss.

2.3 Memory Management Overhead

The rapid growth in deep learning model size has created a fundamental challenge—memory capacity on a single node is no longer sufficient to hold modern models. Dense models such as GPT-3 [6], BLOOM [5], Gopher [31], LLaMA 3.1 [11], Megatron-Turing NLG [36], and PaLM [9] range from 175B to 540B parameters, requiring 350GB to 1.08TB in FP16 format just for parameters, with training demanding 2–3× additional memory for gradients and optimizer states (1–3.2TB total). Sparse Mixture-of-Experts (MoE) models push these limits further. DeepSeek-V3 contains 671 billion total parameters requiring 1.34TB [10], Switch Transformer scales to 1.6 trillion parameters requiring 3.2TB [13], and emerging trillion-parameter models like PanGu-Σ [33] exceed 2TB for parameters alone.

This memory pressure affects both training and inference, pushing the community towards distributed solutions that span multiple nodes. However, current distributed memory solutions are built entirely in user space, incurring substantial overhead from serialization, memory copies, and kernel transitions. Meanwhile, commodity operating systems remain oblivious to AI workloads—they lack native abstractions for TB-scale tensors, cross-node memory sharing, and the access patterns characteristic of modern deep learning.

2.3.1 The Cost of Distributed Memory Management. To understand the performance implications of distributed memory management, we analyze Ray's distributed object store and PyTorch's Fully Sharded Data Parallel (FSDP). We select these two frameworks because they represent the dominant paradigms in distributed AI: Ray exemplifies general-purpose distributed computing with explicit object management, while FSDP represents deep learning-specific solutions with automatic sharding. Other frameworks (DeepSpeed, Megatron-LM, Horovod) share similar architectural patterns—user-space memory management, serialization for communication, and explicit data movement—and thus exhibit comparable overheads.

Challenge C1: Data transformation overhead. Modern transformer architectures access memory at layer granularity, where each layer contains $\sim 12h^2$ parameters (h = hidden dimension). For LLaMA models, this yields $\sim$*total_params*/*num_layers* per layer: 400MB–2GB for 7B–70B models (32–80 layers) [11], 3.6GB per layer for GPT-3 175B (96 layers) [6], and up to 8GB per layer for PaLM 540B (118 layers) [9]. Both Ray and FSDP require expensive data transformations at this scale. As shown in Figure 2a, serialization consumes 76.7% of `put` latency for a 4GB tensor (2.67s out of 3.48s), with deserialization exhibiting similar overhead (71.8% of `get` time).

In FSDP, equivalent overhead arises from parameter flattening and state dict transformations. These transformations are fundamental to user-space memory management, where data must be converted between application format and the communication layer's format on every access.

Challenge C2: Memory amplification. User-space memory management creates multiple buffer copies at each software layer. Table 1 quantifies this for Ray, where a 2GB tensor requires 10.95GB across two nodes—a 5.48× amplification from serialization buffers (1.1–1.2×), Plasma store alignment and metadata (1.5×), and transfer buffers (0.5× per node).

FSDP exhibits similar amplification through different mechanisms. During forward/backward passes, all-gather operations require buffers to reconstruct full parameters from shards. The summon_full_params API materializes complete parameters alongside existing sharded copies. Prefetch buffers pre-allocate memory for upcoming layers. During checkpointing, full_optim_state_dict gathers optimizer state to rank 0, holding both sharded and full copies simultaneously. *What if distributed memory required no more copies than local memory?*

Tensor Size	Total Memory	Amplification
500MB	3.30GB	6.7x
1GB	5.04GB	5.04x
2GB	10.95GB	5.48x
4GB	21.88GB	5.47x
8GB	43.94GB	5.49x
16GB	88.13GB	5.51x

Table 1: Memory amplification in Ray's distributed object store for a two-node put/get scenario.

Challenge C3: Scalability overhead. Scaling to multiple nodes amplifies memory management overhead in both frameworks. Figure 3a illustrates this for PyTorch FSDP as nodes increase from 1 to 128, where memory operations—copying, movement, concatenation, parameter (un)flattening, prefetch buffer management, and gradient synchronization—consume an increasing fraction of execution time. Figure 3b quantifies the underlying cause, showing that scaling from 1 to 16 nodes increases cache misses by 2.91× while context switches surge by 22.27×, reflecting the cost of repeated user-kernel transitions for memory allocation, collective communication setup, and synchronization barriers. Architectural bottlenecks compound these issues. Ray's Global Control Store (GCS) centralizes address lookup on the head node, creating a single point of failure. FSDP's sync_module_states broadcasts parameters from rank 0, serializing initialization. Collective operations (AllReduce, AllGather) require balanced sharding across all ranks, conflicting with the irregular data distributions common in AI workloads like Mixture-of-Experts. *What if memory management overhead vanished as we scale, rather than multiplied?*

2.3.2 User-Space Memory Management Overhead. The overheads above are inherent to user-space memory management. Beyond C1–C3, this design creates additional challenges:

Challenge C4: Pointer invalidation and software fragmentation. When an object moves between nodes, its memory address changes, breaking pointer-based data structures—adjacency lists in Graph Neural Networks [41], optimizer state dictionaries, and MoE routing tables [13]—and forcing developers to explicitly reconstruct references after every transfer. This problem is compounded by software fragmentation: each framework (Ray, PyTorch Distributed, DeepSpeed, Megatron-LM) maintains its own memory management layer, serialization format, and communication protocols. When new hardware or optimization techniques emerge, each framework must independently implement support, duplicating effort and delaying adoption.

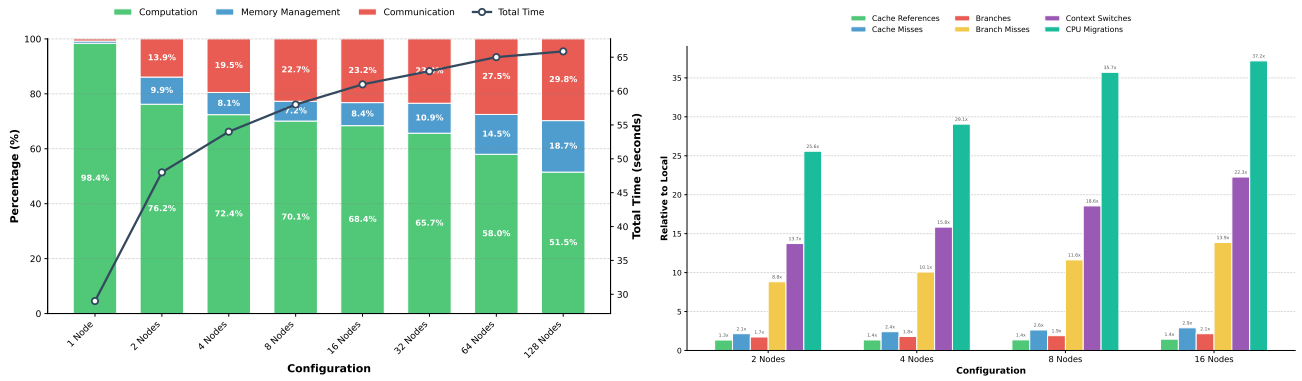
2.3.3 The Case for Unified Memory. Challenges C1–C4 share a common root cause: user-space frameworks must explicitly manage data placement and movement across a fragmented memory hierarchy. As illustrated in Figure 4, *memory pooling*—retrieving data from a neighbor's DRAM over RDMA at 12.5–50 GB/s (100–400 Gb/s NICs) [28]—is faster than traditional *offloading* to NVMe at 3–7 GB/s [34]. DeepSpeed ZeRO-3 [32] and FSDP [23] implement pooling at the framework level, but still require explicit all-gather collectives and user-space serialization.

2.3.4 Challenges for Distributed Inference. **Challenge C5: Dynamic and unpredictable memory requirements.** Unlike training, inference memory varies with user inputs. The KV cache grows linearly with sequence length—for a 70B model, from a few GB to over 40GB [18]. Static partitioning cannot accommodate this, where peak provisioning wastes resources while underprovisioning causes failures. Dynamic batching in vLLM [18] compounds this by processing requests with varying lengths simultaneously. The cost of memory management in inference is substantial. In our preliminary experiments with Llama-3.1-8B (§5), we observed approximately 4 million page faults during inference. For the typical range of 200–300 generated tokens per user request [46], an average of 20% of inference latency is spent on page fault handling alone. This overhead directly impacts user-perceived latency and reduces serving throughput. Current systems (Megatron-LM [35], DeepSpeed-Inference [1]) use static partitioning—pre-determined shard placement that cannot adapt to dynamic memory growth. *What if memory could grow and shrink transparently via demand paging?*

3 System Design

Figure 5 illustrates the fundamental architectural shift that NTSM introduces: moving distributed memory management from user space into the kernel. In traditional systems (left), ML frameworks, communication backends, and orchestration layers each maintain their own memory abstractions at the user level, requiring explicit data movement and serialization for every cross-node access. NTSM (right) interposes between applications and hardware, consolidating distributed memory into a kernel-managed shared region where parameters, gradients, and optimizer states can be accessed transparently via demand paging and zero-copy RDMA.

NTSM provides a transparent distributed shared memory abstraction for AI workloads. Applications issue ordinary loads and stores; on a fault, the kernel fetches remote data via RDMA, enforces coherence, and maps pages—replacing explicit "serialize → copy → send → deserialize" paths with a fault-driven data path.



(a) LLAMA3P1-8B FSDP time breakdown

(b) LLAMA3P1-8B FSDP performance statistics

Figure 3: Analysis of PyTorch FSDP memory management overhead during distributed training of LLAMA3P1-8B.

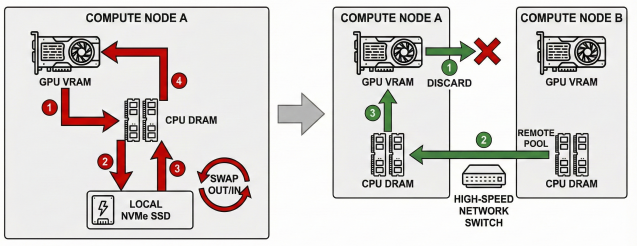


Figure 4: Remote memory vs. conventional offloading.

This section describes NTSM's architecture and the mechanisms that enable this transparency.

3.1 Design Principles

NTSM's design follows from the challenges identified in Section 2.3. We summarize the key decisions here before detailing their realization.

To eliminate serialization and user-kernel crossing overhead (C1–C3), NTSM runs in kernel space—primarily as a loadable module, with a single exported symbol required on x86—and directly manipulates page tables, TLBs, and DMA mappings. Data moves in its native in-memory representation via RDMA, with no intermediate bounce buffers. The kernel handles faults, updates PTE permissions, and pins pages for DMA—operations that cannot be performed efficiently from user space.

To preserve pointer validity and avoid software fragmentation (C4), all nodes map the shared region at identical virtual addresses. A pointer value on one node refers to the same logical location on every other node, enabling pointer-rich structures like optimizer state dictionaries and MoE routing tables to work without modification. By providing memory management below the framework layer, NTSM benefits all frameworks automatically.

To address the scale mismatch between classical DSM and AI workloads, NTSM introduces Virtual Memory Regions (VMRs) as the coherence unit. Rather than tracking millions of 4KB pages, NTSM operates at 32KB–2MB granularity, matching the natural access grain of transformer layers. A fault on any page fetches the entire VMR, amortizing fault overhead and RDMA setup cost. NTSM uses a directory-based protocol with M/S/I states at VMR granularity, with explicit point-to-point invalidations and acknowledgments. Coherence activates only on write faults; read-only access incurs no invalidations, matching AI's read-heavy forward pass.

Finally, to handle dynamic memory requirements in inference (C5), NTSM populates VMRs on first access and reclaims them under memory pressure using LRU eviction. This supports varying KV cache sizes and workload shifts without framework-specific memory managers.

3.2 System Architecture

Figure 6 illustrates NTSM's architecture and the complete path of a read fault. Each node is structured into three layers: user space (AI application with tensors), kernel space (NTSM module), and hardware (CPU, DRAM, RDMA NIC). A symmetric shared region spanning 1TB is mapped at identical virtual addresses on all nodes, initially with no access permissions.

The NTSM kernel module contains five interacting components. The **Fault Manager** intercepts page faults and coordinates the response. The **Memory Manager** allocates physical pages, tracks VMR metadata (32KB–2MB granularity), and implements the M/S/I coherence protocol. The **Cache Manager** maintains CPU cache coherence after NIC-initiated DMA writes. The **Page Table Manager** manipulates PTEs to enforce coherence states and handles TLB flushes. The **Message Manager** handles RDMA communication, including work request submission and completion queue processing.

The numbered arrows in Figure 6 trace a read fault where Node 0 fetches data from Node 1:

(1) The application issues a load to address `0x7f...` within the shared region.
(2) The CPU raises a page fault (no valid PTE); the Fault Manager intercepts it.
(3) The Fault Manager computes the containing VMR and locates the home node via the Memory Manager.

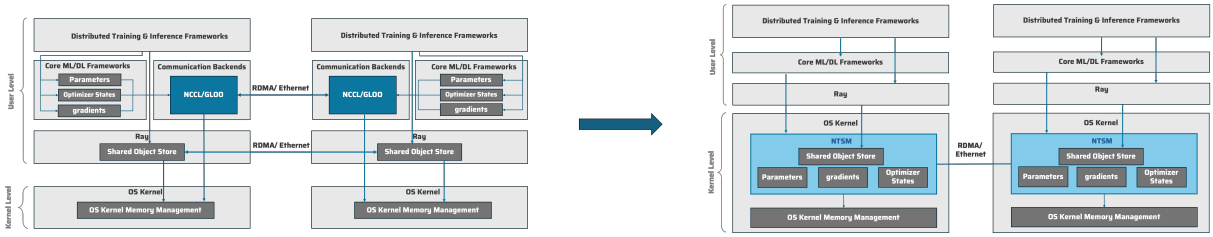


Figure 5: NTSM-enabled vs. traditional distributed AI stack

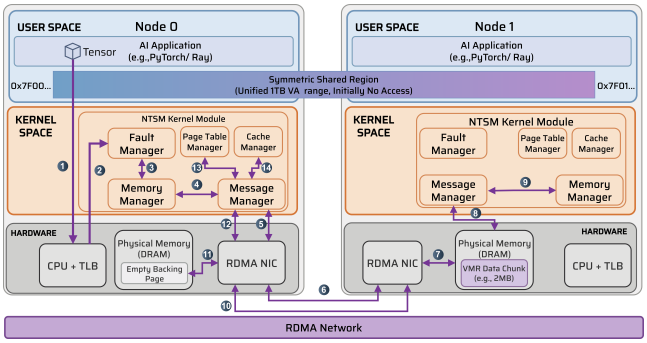


Figure 6: NTSM system architecture

(4) The Message Manager prepares a one-sided RDMA Read request for the remote VMR data.
(5) The request is submitted to the local RDMA NIC.
(6) The NIC transmits the work request to Node 1's RDMA NIC.
(7) Node 1's NIC accesses physical memory directly and sends the data back.
(8) A completion queue entry (CQE) notifies Node 1's Message Manager to update coherence state.
(9) Node 1 updates the VMR state to Shared and records Node 0 in the sharer set.
(10) Node 1 sends the VMR pages to Node 0 via RDMA Write.
(11) Node 0's RDMA NIC writes data directly into pre-allocated physical pages.
(12) A CQE notifies Node 0's Message Manager that the transfer completed.
(13) The Page Table Manager updates PTEs to valid/read-only and flushes the TLB.
(14) The CPU cache is flushed to ensure coherence with the NIC-written data.

This entire sequence is transparent to the application—it simply retries the faulting load and finds valid data. No serialization, no intermediate buffers, and no explicit communication API is required.

3.3 Symmetric Virtual Address Space

NTSM reserves a contiguous virtual address range on every node—for example, 1TB starting at a fixed base address. At initialization, nodes coordinate to agree on the shared region bounds. Physical memory is not allocated at reservation time; the region is mapped with no access permissions.

When an application first accesses any address in the shared region:

(1) The CPU raises a page fault because no valid PTE exists.
(2) The Fault Manager computes the VMR containing the faulting address.
(3) If this is the first cluster-wide access, the faulting node becomes the initial owner and allocates local pages. Otherwise, the Fault Manager queries the Memory Manager to locate the current owner and fetches data via RDMA.
(4) PTEs are installed with appropriate permissions, and the fault resolves.

This design preserves pointer identity: a pointer `0x7f...` refers to the same logical location on every node. Applications can pass pointers across nodes, store them in data structures, and dereference them transparently—similar to the symmetric address space approach in Popcorn Linux [4]. NTSM uses a fixed `mmap` base address for the shared region, bypassing Address Space Layout Randomization (ASLR) for this specific range while leaving the rest of the address space randomized. The kernel handles all relocation of backing pages.

3.4 Virtual Memory Regions

A VMR is the fundamental unit of coherence and data transfer. Each VMR is defined by its base address (aligned within the shared region), its size (configured at module load and fixed for the run), and its coherence state. The Memory Manager tracks the manager node (responsible for serializing coherence transitions), the owner (for Modified VMRs), and the sharer set (for Shared VMRs).

VMR size is configurable from 4KB to 1GB. The choice reflects a trade-off: larger VMRs amortize fault and metadata overhead but may fetch more data than needed; smaller VMRs reduce over-fetch but increase fault frequency. For transformer models, where layers range from 400MB to 1.6GB and are accessed sequentially, 32KB–2MB provides effective amortization while bounding transfer size. Section 5.3.1 evaluates this trade-off empirically.

Unlike Linux Transparent Huge Pages, VMRs do not require physically contiguous memory. THP must find or create contiguous 2MB regions, leading to fragmentation over long-running jobs and unpredictable promotion latency. VMRs are purely a software abstraction: any set of 4KB pages can back a VMR, with the Page Table Manager handling the mapping. This avoids THP's fragmentation problems while still providing coarse-grained coherence.

3.5 Directory Coherence Protocol

NTSM implements a write-invalidate protocol with M/S/I states, coordinated by the Memory Manager using a directory-based approach. Each VMR is assigned a manager node deterministically—for example, by hashing the VMR base address modulo the node count. The manager's Memory Manager maintains authoritative metadata: whether the VMR is uncached, shared, or modified; the owner if modified; and the set of sharers if shared. Because responsibility is partitioned across nodes, there is no central coordinator; every node manages roughly 1/N of the VMRs.

A read fault (Invalid → Shared) proceeds as follows:

(1) The faulting node sends a read request to the VMR's manager.
(2) The manager checks coherence state: if uncached, it allocates pages and becomes the owner; if shared, it forwards to any sharer; if modified, it forwards to the current owner.
(3) The data source sends VMR contents via RDMA write.
(4) The requester maps pages read-only; the manager records it in the sharer set.

A write fault (Shared → Modified) requires invalidating other copies:

(1) The CPU raises a protection fault because pages are mapped read-only.
(2) The Fault Manager sends an upgrade request to the manager.
(3) The manager sends point-to-point invalidations to all sharers.
(4) Each sharer unmaps its copy and acknowledges.
(5) After all acknowledgments, the manager grants exclusive ownership.
(6) The requester remaps the VMR read-write.

If a node writes to a VMR in Invalid state, it sends a read-exclusive request. The manager invalidates any sharers, retrieves data from the current owner if the VMR was modified, and grants exclusive ownership to the requester in a single round-trip where possible.

This protocol does not assume network ordering. Correctness relies on the manager serializing all transitions for a given VMR and on explicit invalidation acknowledgments before granting write permission. This is the same fundamental approach used by prior directory-based DSM systems, extended here to VMR granularity.

3.6 RDMA Transport

NTSM uses RDMA for both control messages and bulk data transfer. Coherence lookups, invalidations, and acknowledgments are small and latency-sensitive; the Message Manager sends these using two-sided Send/Recv operations on pre-established reliable connections. Each node maintains one queue pair per peer.

VMR data transfers use one-sided RDMA Write operations. Critically, the receiving node prepares the destination *before* any data arrives: it allocates physical pages, pins them to prevent swapping, and registers them with the NIC for DMA. Only then does it provide the DMA-mapped addresses to the sender, which performs the RDMA Write directly into those pages. Completion is signaled via Write-with-Immediate, which posts an entry to the receiver's completion queue. The Fault Manager polls this queue, and upon completion, installs PTEs mapping the same physical pages into the application's address space.

This path is truly zero-copy: because destination pages are allocated, pinned, and DMA-mapped before the transfer, data lands directly into application-backing pages with no intermediate buffer, no serialization, and no CPU-mediated copy. The physical pages used as the DMA target become the backing store for user-space mappings—no kernel buffers intervene at any point.

3.7 Eviction and Dynamic Working Sets

AI inference exhibits dynamic memory behavior. KV cache size grows with sequence length, batch sizes vary across requests, and different workloads have different memory footprints. NTSM handles this through demand paging and eviction.

New accesses simply fault in VMRs as needed; no pre-allocation is required. Under memory pressure, the Memory Manager selects victim VMRs based on LRU order. For a Shared victim, the node unmaps locally and notifies the manager to remove it from the sharer set. For a Modified victim, the node writes back to the manager or transfers ownership to another sharer before unmapping. Evicted VMRs transition to Invalid locally; the next access re-faults and fetches current data.

This mechanism is transparent to applications. Memory grows as tensors are accessed, shrinks as cold regions are evicted, and frequently accessed VMRs remain resident. For AI's repetitive layer-by-layer access pattern, LRU provides a good approximation of optimal eviction.

4 Implementation

We implemented NTSM as a kernel module for Linux, tested on kernel versions 5.15 and 6.11. On x86, a one-line kernel patch is required to export `flush_tlb_range`; the module loads unmodified on arm64. The user-space integration uses PyTorch 2.5 and Ray 2.37. The implementation consists of three components: the kernel module (2,923 lines of C), a Python user-space library (30 lines), and targeted modifications to PyTorch FSDP (152 lines) and Ray Parameter Server (89 lines).

It is important to note that NTSM performs *no ML computation in kernel space*. The kernel module handles only memory management: allocation, page table manipulation, coherence protocol, and RDMA transfers. All tensor operations, neural network computations, and gradient calculations remain in user space, executed by PyTorch or other frameworks. This design sidesteps the well-known challenges of kernel-space floating-point operations and keeps the kernel module focused on what the OS does best: memory and I/O management.

4.1 Kernel Module

Applications initiate shared memory by calling `mmap` with `MAP_FIXED`, which creates a Virtual Memory Area (VMA) at the same base address on all nodes. No physical pages are allocated at this point; the region is mapped with no access permissions. When the application accesses any address in this region, a page fault occurs and the Fault Manager handles it on demand.

VMR management. A VMR is the unit of coherence and data transfer. Each VMR tracks its base address, size, coherence state

(M/S/I), home node, and an array of PTE pointers for all constituent pages. VMR size is configured at module load time:

```
insmod ntsm.ko vmr_size=2097152  # 2MB VMRs
```

The size remains fixed during execution so all nodes maintain consistent boundaries. When a fault occurs, the Fault Manager computes the containing VMR as `vmr_base = fault_addr & ~(vmr_size - 1)`, fetches all pages via a single RDMA transfer, allocates physical pages, and populates the PTE array. The PTE array enables O(1) state transitions: on invalidation, we iterate through the array and modify permissions directly, then batch TLB flushes via `flush_tlb_range()`.

We evaluated VMR sizes from 4KB to 1GB (Section 5.3.1). For dense sequential workloads like parameter tensors, larger VMRs (1–2MB) minimize fault overhead. For sparse workloads like MoE routing, smaller VMRs (64–128KB) reduce over-fetch. The default is 512KB.

Page table manipulation. The M/S/I protocol requires controlling page permissions: Modified allows read and write, Shared allows read only, and Invalid triggers faults on any access. We implement this by performing manual page walks to locate the PTE for each page, then setting permission bits accordingly. After modifying PTEs, TLB flushes are required. The `flush_tlb_range` function is exported on arm64 but not on x86, where it is defined but marked internal. Our x86 deployment adds a single `EXPORT_SYMBOL` directive (one line) to expose this function to modules. This is the sole kernel modification required; all other NTSM functionality operates as a standard loadable module. We have submitted this export for upstream inclusion, as the function is already stable ABI on arm64 and the asymmetry appears unintentional. Since TLB flushing must occur in atomic context, we enqueue it via workqueue.

When a page transitions to Invalid, we free its physical memory by clearing the page refcount. This allows batch deallocation rather than blocking on each free. For eviction, we maintain an LRU queue of page indices ordered by fault time, enabling identification of cold pages.

Multi-process and thread safety. For multi-process support, we track all process IDs and apply the M/S/I protocol between processes locally. For multi-threaded access, each VMR has a mutex lock to manage contention. We chose mutexes over spinlocks because they consume fewer CPU cycles, leaving more resources for the AI application.

DMA and cache coherence. To enable RDMA, we allocate DMA addresses from physical pages and pin them by modifying PTEs to prevent swapout during NIC access. After the NIC completes a write, we flush CPU caches using `flush_cache_page` before mapping pages to user space.

Memory regions. We use `IB_MR_TYPE_DMA` for RDMA memory regions because it allows placing DMA addresses directly in work requests without translation. Unlike other memory region types that require registering physical memory segments, this eliminates frequent registration overhead—important since our design constantly allocates and deallocates pages. The trade-off is that remote DMA addresses must be propagated explicitly: we include them in control messages and copy them into subsequent data transfer work requests.

Message types. Control messages (invalidations, acknowledgments) are latency-sensitive. We send them using `IB_SEND_INLINE` to eliminate one DMA operation, and use two-sided RDMA Send/Recv for notification semantics. Data messages (page transfers) are throughput-sensitive. We use one-sided RDMA Write to transfer pages directly into the receiver's memory. The preceding control message carries the receiver's DMA address, which we embed in the Write work request. We use `IB_WR_WRITE_WITH_IMM` with the page index in the immediate field to notify the receiver of completion without an additional message.

Connection setup. All Queue Pairs (QPs) are Reliable Connection (RC) type for guaranteed delivery. Each node runs both server and client roles: the server connects to all remote clients and processes incoming messages, while local clients connect to remote servers for outbound messages. This separation reduces contention between threads.

Completion handling. We use interrupt-based completion rather than polling. While polling offers lower latency, it consumes CPU cycles that the AI application needs. Interrupt-based handling trades slightly higher latency for better CPU availability.

4.2 User-Space Library

The Python library provides two functions: `ntsm.init(size)` maps the shared region via the kernel module's character device, and `ntsm.allocate_tensor(shape, numel)` returns a PyTorch tensor backed by shared memory. The returned tensor is standard—all PyTorch operations work unchanged.

4.3 Framework Integration

Integrating NTSM with an existing framework requires three steps: (1) replace parameter/gradient allocation with `allocate_tensor`, (2) remove explicit collective communication for data that is now shared, and (3) ensure all nodes initialize the shared region before model construction.

PyTorch FSDP.. FSDP shards model parameters across ranks and uses AllGather to reconstruct full parameters before each forward pass. With NTSM, parameters reside in shared memory and are accessed via page faults—no AllGather is needed. We modified `torch/distributed/fsdp/_flat_param.py` (152 lines changed):

Before *FSDP allocates local tensors*

```
flat_param = nn.Parameter(torch.empty(
    total_size, dtype=dtype))
```

After *NTSM allocates from shared region*

```
flat_param = nn.Parameter(ntsm.
    allocate_tensor((total_size,), dtype))
```

We also removed `_all_gather()` in `unshard()` and `_reduce_scatter()` in `reshard()`—when a rank accesses a remote shard, NTSM fetches it automatically via page fault.

Ray Parameter Server. Ray's Parameter Server stores model parameters on dedicated server actors. Workers fetch parameters via `ray.get()` and push gradients via `ray.put()`, both serializing through Plasma. We modified the server (89 lines changed):

Before *Allocates local tensors, requires ray.get()/put()*

```
self.params = {k: torch.zeros_like(v) for k,
    v in model.items()}
```

After *Shared memory, direct access without serialization*

```
self.params = {k: ntsm.allocate_tensor(v.
    shape, v.numel())
                for k, v in model.items()}
```

Workers access `self.params` directly—the first access faults in the remote data, eliminating Ray's 76.7% serialization overhead (Figure 2a).

Integrating other frameworks. The same pattern applies to any distributed AI framework: (1) identify where parameters/gradients are allocated, (2) replace with `ntsm.allocate_tensor()`, (3) remove explicit collectives or transfers, and (4) add `ntsm.init()` before allocation. For closed-source frameworks, users can allocate tensors from the shared region at the application level and pass them to the framework.

5 Evaluation

We evaluate NTSM on a 34-node cluster provided by CloudLab, where each node contains a 24-core AMD EPYC 7402P processor (2.80GHz) and 128GB DDR4-3200 ECC memory. The nodes are equipped with dual 480GB SATA SSDs and Mellanox ConnectX-5 NICs with dual 100Gbps ports over PCIe 4.0. A non-blocking Ethernet fabric interconnects the nodes, ensuring full bisection bandwidth across the cluster. All nodes run Linux kernel version 5.15 with our NTSM kernel module. Given our cluster size of 34 nodes, we conducted our evaluation experiments up to 32 nodes to maintain balanced configurations for distributed training and inference scenarios.

5.1 Experimental Setup

We designed our experiments to maximize memory resource utilization as the number of nodes in the cluster increases. With more nodes, the total memory capacity grows, allowing us to distribute larger models more effectively. This scaling helps us stress the memory subsystem and test NTSM's efficiency under more demanding conditions. Specifically, we evaluate three large-scale language models across different node configurations to observe how well NTSM manages memory in distributed settings:

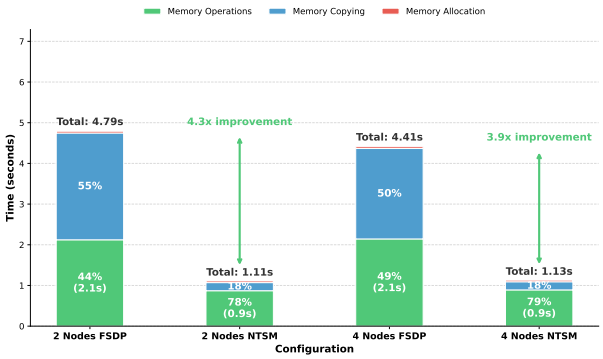


Figure 7: Memory management speedup in training: NTSM vs FSDP (2-4 nodes).

- LLaMA-3.1-8B [11] (8B parameters, 32GB model size) on 2-4 nodes
- LLaMA-3.1-70B [11] (70B parameters, 280GB model size) on 8-16 nodes
- OPT-175B [43] (175B parameters, 700GB model size) on 32 nodes

Model configurations are evaluated across three distinct distributed computing scenarios: distributed training, distributed inference and parameter server architecture. In each scenario, we place key data structures that are shared among nodes, such as model weights, into NTSM shared memory to maximize the benefits of transparent memory sharing.

We compare against PyTorch FSDP and Ray+PyTorch because: (1) FSDP is the state-of-art for distributed training on CPUs with automatic sharding, (2) Ray is the leading framework for distributed inference with CPU memory management, and (3) both represent user-space approaches that NTSM aims to improve upon.

5.2 Performance Breakdown Analysis

5.2.1 Training. NTSM achieves 4.3–6.5× speedup in memory operations by eliminating all-gather collectives and reducing memory copying overhead. To understand the source of these gains, we break down memory management into three components: allocation/initialization, copying, and operations. For FSDP, these operations primarily involve sharding via `aten::copy_`, `aten::clone`, and `aten::view`, while NTSM manages page tables and internal memory components directly at the kernel level.

Figures 7, 8, and 9 present the breakdown across different cluster sizes. At small scale (2-4 nodes, Figure 7), NTSM achieves a 4.3× reduction in total execution time. The most dramatic improvement appears in memory copying, where NTSM completes in 0.2s compared to FSDP's 2.62s—a 13× reduction. Memory operations similarly improve from 2.12s to 0.87s. These gains stem from NTSM's ability to avoid the serialize-transfer-deserialize cycle that FSDP requires for every parameter shard.

As we scale to 8-16 nodes (Figure 8), NTSM's advantages become more pronounced. At 16 nodes, NTSM achieves its peak improvement of 6.5× in total execution time (1.35s vs 8.71s). Memory

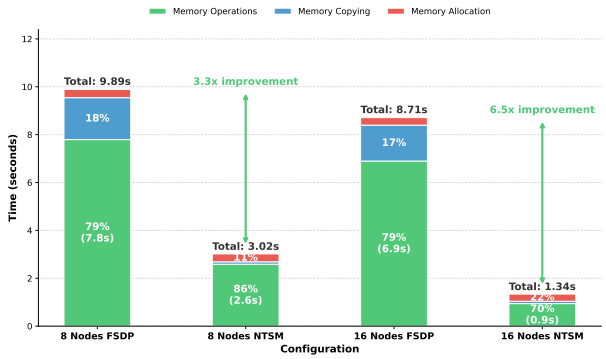


Figure 8: Memory management speedup in training: NTSM vs FSDP (8-16 nodes).

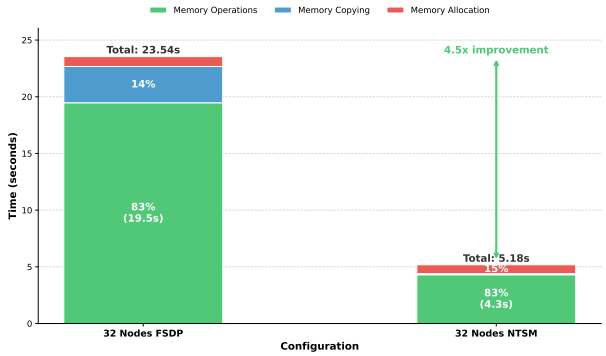


Figure 9: Memory management speedup in training: NTSM vs FSDP (32 nodes).

copying drops to just 0.1s compared to FSDP's 1.5s, while memory operations improve from 6.89s to 0.945s. This scaling behavior reflects a fundamental difference in how the two systems handle distributed memory: FSDP's all-gather collectives grow in cost with cluster size, while NTSM's on-demand paging incurs overhead only for actually-accessed data.

At large scale (32 nodes, Figure 9), NTSM maintains strong performance with a 4.5× improvement in total execution time (5.18s vs 23.54s). The memory copying advantage reaches 32× (0.1s vs 3.2s), and memory operations show a 7.3× improvement (4.3s vs 19.47s). While the overall speedup slightly decreases from the peak at 16 nodes, this is expected: at 32 nodes with the OPT-175B model, the working set size and coordination complexity increase substantially. Nevertheless, NTSM's fine-grained memory access at 32KB granularity—versus FSDP's layer-level restrictions—continues to provide significant benefits by enabling better communication-computation overlap.

5.2.2 Inference. Figure 10 breaks down inference memory-management overhead (excluding compute time) between NTSM and Ray+PyTorch. NTSM reduces this overhead by 3.3–6.5× through zero-copy RDMA and on-demand paging: 3.9–4.3× at 2–4 nodes, up to 6.5× at 16

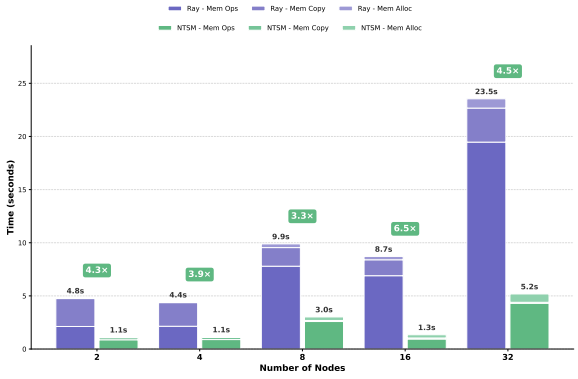


Figure 10: Inference memory-management speedup: NTSM vs Ray+PyTorch.

nodes, and 4.5× at 32 nodes. The gains come from eliminating redundant copies and shifting data movement to fault-driven RDMA transfers.

5.3 VMR Size Trade-offs

5.3.1 VMR Size Sensitivity. A key design parameter in NTSM is the size of Virtual Memory Regions (VMRs)—the granularity at which pages are managed and transferred between nodes. This choice involves a fundamental trade-off between OS overhead and network transfer efficiency.

Figure 11 shows the impact of VMR size on inference latency for Llama-3.1-8B, breaking down the time into page fault handling and RDMA transfer components. As VMR size increases from 4KB to 1GB, we observe two opposing effects:

VMR is a readahead window, not a tensor size. VMR size should be interpreted like a *block size* or *readahead window* for fault-driven RDMA, not the size of an AI object such as a layer or the KV cache. A naive approach might set VMR size equal to layer size to minimize fault count—a 100MB layer would incur just one fault. However, this ignores network physics: transferring 100MB in a single VMR takes ∼4ms of blocking time. In contrast, 512KB VMRs decompose the same layer into ∼200 transfers of ∼23μs each, enabling aggressive pipelining where one transfer completes while the CPU processes previously-fetched data.

The optimal VMR size balances OS overhead (fewer faults → larger VMRs) against transfer blocking time (shorter per-fault latency → smaller VMRs). For Llama-3.1-8B inference, 32KB–2MB is optimal; beyond 2MB, reduced page fault overhead is offset by increased transfer time and over-fetching. Smaller VMRs may be preferable for sparse access patterns such as Mixture-of-Experts architectures.

5.4 End-to-End Performance

5.4.1 Distributed Training Scalability. Figure 12 shows training throughput scalability against PyTorch FSDP as nodes increase from 2 to 32. NTSM consistently achieves approximately 14% higher throughput by: (1) enabling on-demand memory access through

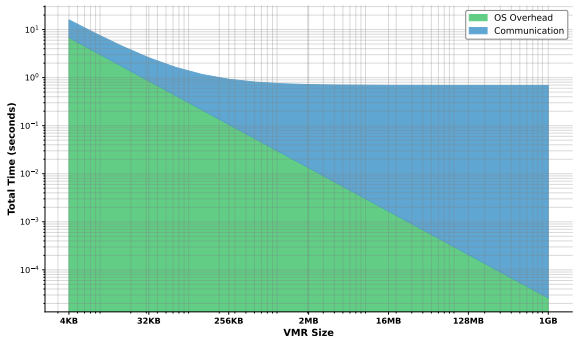


Figure 11: VMR size impact on inference latency (Llama-3.1-8B).

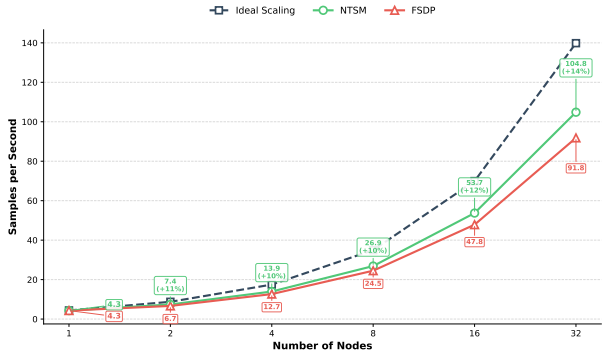


Figure 12: Training throughput scalability: NTSM vs FSDP.

page faults rather than FSDP's all-gather synchronization barriers, and (2) eliminating user-kernel transitions and serialization overhead through direct page table manipulation.

5.4.2 *Distributed Inference Latency.*

Time to First Token (TTFT) Analysis. TTFT captures cold-start prefill latency, where the model must be fetched over the network before generating the first token. We measure cold start explicitly with no model data cached in DRAM. Critically, generating the *first* token requires only the critical execution path—approximately 10–15% of the model.

As shown in Figure 13, Ray+PyTorch increases steeply from 850ms at 2 nodes to 3.8s at 32 nodes, while NTSM scales from 195ms to 890ms—a consistent 3.9–4.4× speedup. Ray+PyTorch must serialize and distribute all parameters to each worker *before* the first forward pass can begin. NTSM avoids this synchronization barrier through demand paging: nodes begin execution immediately and fetch parameters on-demand via RDMA, overlapping data transfer with computation. The jumps at 8 and 32 nodes reflect transitions to larger models (70B and 175B).

5.4.3 *Parameter Server Iteration Time.*

We evaluate NTSM against Ray+PyTorch in parameter server architectures by storing model weights and gradients in NTSM shared memory. As shown in Figure 14, NTSM achieves 28% improvement at 2 nodes (74.4s vs 103.4s),

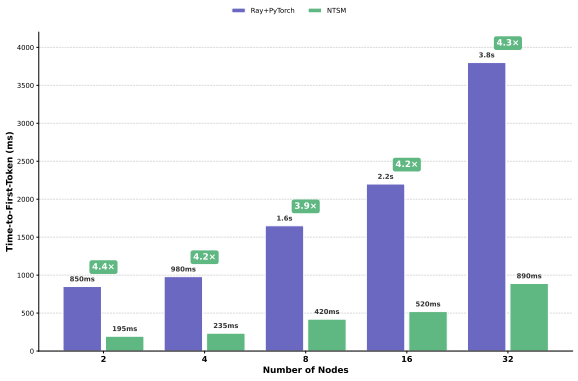


Figure 13: Time to First Token (TTFT): NTSM vs Ray+PyTorch.

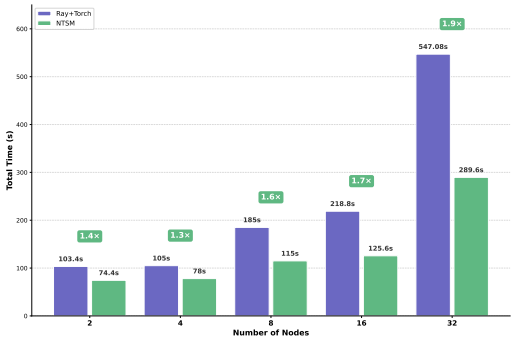


Figure 14: Parameter server iteration time: NTSM vs Ray+PyTorch.

43% at 16 nodes (125.6s vs 218.83s), and 47% at 32 nodes (289.6s vs 547.08s). These gains stem from eliminating redundant copying and serialization in Ray+PyTorch's RPC framework.

5.5 Limitations

NTSM is designed for datacenter conditions with large-scale distributed workloads. First, NTSM provides little benefit for small models fitting within a single node's memory or at small cluster scales (2–4 nodes), where the overhead of page fault handling, RDMA setup, and coherence tracking exceeds potential gains. For training workloads fitting entirely in GPU memory, NTSM introduces unnecessary overhead.

Second, NTSM is optimized for read-heavy workloads. Write-intensive workloads require constant cache invalidation across nodes, which can saturate network bandwidth—making NTSM better suited for inference than for workloads with frequent parameter updates.

Finally, VMR size selection requires workload-specific tuning. As shown in Section 5.3.1, inappropriate VMR sizes lead to either excessive page fault overhead or internal fragmentation.

6 Related Work

Table 2 positions NTSM against prior systems across key dimensions.

Key observations. No prior system combines: (i) kernel-space implementation for direct page table control, (ii) PGAS design with selective sharing, (iii) zero-copy RDMA, (iv) automatic coherence, (v) coherence granularity >4KB, and (vi) AI workload awareness. Classical DSM (IVY [19], TreadMarks [17], Munin [8]) lacks RDMA and targets GB-scale workloads—a 175B model spans 684M pages at 4KB granularity. PGAS systems (UPC++ [3], Global Arrays [27]) lack automatic coherence. RDMA-based systems operate in user space: Grappa [26] uses delegation without caching; GAM [7] uses 512B cache lines. Kernel systems provide partial solutions: Popcorn [4] uses TCP over the entire address space; LITE [39] lacks coherence; Infiniswap [14] provides capacity expansion without shared-memory semantics. AI frameworks (FSDP [45], ZeRO [32], Ray [25]) require explicit collectives with serialization overhead.

Distributed AI frameworks. FSDP and ZeRO shard model state and reconstruct via AllGather/ReduceScatter; Ray uses serialization-based object stores. Key differences from NTSM: (i) explicit collectives vs. transparent memory access, (ii) serialization overhead vs. native representation, (iii) user-space buffer management vs. kernel-level zero-copy, (iv) no coherence vs. automatic M/S/I protocol.

Complementary systems. KV cache optimizations (vLLM [18], LMCache [24], Mell [40], InfiniStore [44], ShadowKV [37]), prefill/decode disaggregation (Mooncake [30], DistServe [47], Splitwise [29], KVDirect [2]), and MoE expert offloading (MoE-Infinity [42], HOBBIT [22], Fiddler [16]) solve orthogonal problems at higher abstraction levels. NTSM could serve as the underlying memory substrate for such systems.

7 Conclusion

Current distributed AI frameworks manage memory in user space, incurring overhead from serialization (up to 76.7% of latency), memory amplification (up to 6.7×), and escalating coordination costs—because commodity OSes lack primitives for TB-scale tensors and cross-node sharing. NTSM addresses this by moving distributed memory management into the kernel, providing a symmetric shared address space where applications access remote data through ordinary memory operations. Page faults trigger zero-copy RDMA transfers with automatic M/S/I coherence, while Virtual Memory Regions at 32KB–2MB granularity match transformer layer access patterns.

With minimal framework modifications (152 lines in PyTorch FSDP, 89 lines in Ray), NTSM achieves up to 6.5× speedup in distributed training, 4.4× reduction in time-to-first-token, and 47% reduction in parameter server iteration time on a 34-node cluster with models up to 175B parameters. As models continue scaling beyond individual node capacity, OS-level abstractions that understand AI workload characteristics will become essential infrastructure for efficient distributed execution.

References

[1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 46, 15 pages.

[2] Ho Yin Au, Wei Lin Chen, Yuhang Xiong, Wei Jiang, Zeyuan Zhou, Ion Stoica, Xiyue Pu, and Ying Sheng. 2025. KVDirect: Distributed Disaggregated LLM Inference with Proactive KV Cache Transfer. *arXiv preprint arXiv:2503.00721* (2025).

[3] John Bachan, Scott B. Baden, Dan Bonachea, Paul H. Hargrove, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Brian van Straalen, and Yili Yan. 2019. UPC++: A High-Performance Communication Framework for Asynchronous Computation. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 963–973. https://doi.org/10.1109/IPDPS.2019.00104

[4] Antonio Barbalace, Binoy Ravindran, and David Katz. 2014. Popcorn: a Replicated-kernel OS Based on Linux. 123–138. Linux Symposium 2014, OLS 2014 ; Conference date: 14-07-2014 Through 16-07-2014.

[5] BigScience Workshop. 2023. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. *arXiv preprint arXiv:2211.05100* (2023).

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[7] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient distributed memory management with RDMA and caching. *Proc. VLDB Endow.* 11, 11 (July 2018), 1604–1617. https://doi.org/10.14778/3236187.3236209

[8] John B. Carter, John K. Bennett, and Willy Zwaenepoel. 1991. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 152–164.

[9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sashank Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2024. PaLM: scaling language modeling with pathways. *J. Mach. Learn. Res.* 24, 1, Article 240 (March 2024), 113 pages.

[10] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437* (2024).

[11] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024).

[12] Jiarui Fang, Zilin Zhu, Shenggui Li, Hui Su, Yang Yu, Jie Zhou, and Yang You. 2023. Parallel Training of Pre-Trained Models via Chunk-Based Dynamic Memory Management. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (2023), 304–315. https://doi.org/10.1109/TPDS.2022.3219819

[13] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: scaling to trillion parameter models with simple and efficient sparsity. *J. Mach. Learn. Res.* 23, 1, Article 120 (Jan. 2022), 39 pages.

[14] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 649–667.

[15] Yanping Huang, Cheng Yonglong, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc Le, and Zhifeng Chen. 2018. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *arXiv:1811.06965 arXiv preprint* (11 2018). https://doi.org/10.48550/arXiv.1811.06965

[16] Keisuke Kamahori, Yile Gu, Kan Zhu, and Baris Can Guo. 2024. Fiddler: CPU-GPU Orchestration for Fast Inference of Mixture-of-Experts Models. *arXiv preprint arXiv:2402.07033* (2024).

[17] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*. USENIX Association, 115–131.

[18] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

Table 2: Comparison of NTSM with related systems. Bold indicates NTSM's unique advantages. K=Kernel, U=User-space, P=Partial, ✓=Yes, –=No, N/A=Not Applicable.

System	Impl	PGAS	RDMA	Zero-copy	Coherence	Coh. Unit	Sym. VA	Transparent	AI-Aware	Scale
Classical Distributed Shared Memory										
IVY [19]	K	–	–	–	Auto	1KB	✓	✓	–	GB
TreadMarks [17]	U	–	–	–	Auto	4KB	✓	✓	–	GB
Munin [8]	U	–	–	–	Auto	4KB	✓	✓	–	GB
Grappa [26]	U	P	✓	–	Deleg.	N/A	P	✓	–	GB
GAM [7]	U	–	✓	–	Auto	512B	–	P	–	GB
PGAS Systems										
UPC++ [3]	U	✓	✓	–	None	Object	✓	–	–	TB
Global Arrays [27]	U	✓	P	–	None	Block	✓	–	–	TB
Kernel-Level Memory Systems										
Popcorn [4]	K	–	–	–	Auto	4KB	✓	✓	–	GB
LITE [39]	K	N/A	✓	✓	**None**	N/A	–	–	–	GB
Infiniswap [14]	K	N/A	✓	–	N/A	4KB	–	✓	–	TB
Distributed AI Frameworks										
PyTorch FSDP [45]	U	–	P	–	None	Param	–	–	✓	TB
DeepSpeed ZeRO [32]	U	–	P	–	None	Param	–	–	✓	TB
Ray [25]	U	–	–	–	None	Object	–	–	P	TB
Megatron-LM [35]	U	–	P	–	None	Param	–	–	✓	TB
PatrickStar [12]	U	–	–	–	None	Chunk	–	–	✓	TB
NTSM	**K**	✓	✓	✓	**Auto**	**32KB–2MB**†	✓	✓	✓	**TB**

†Optimal range; configurable from 4KB to 1GB.

[19] Kai Li and Paul Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems* 7, 4 (1989), 321–359. https://doi.org/10.1145/75104.75105

[20] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 583–598.

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

[22] Peng Li, Ming Zhang, Yang Li, Hui Sun, and Kexin Sheng. 2024. HOBBIT: Mixed Precision Expert Offloading for Fast MoE Inference. *arXiv preprint arXiv:2411.01433* (2024).

[23] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch distributed: experiences on accelerating data parallel training. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3005–3018. https://doi.org/10.14778/3415478.3415530

[24] Yuhan Liu, Hanchen Li, Yihua Du, Shang Yao, Jiayi Ruan, Junwei Huang, Suyi An, Peng Hong, Lun Lou, Chuanxiong Zhong, Yao Gu, Hao Zhang, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, Ying Sheng, and Lianmin Zheng. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*. ACM, 38–56.

[25] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[26] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 291–305. https://www.usenix.org/conference/atc15/technical-session/presentation/nelson

[27] Jarek Nieplocha, Robert J. Harrison, and Richard J. Littlefield. 1996. Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing* 10, 2, 169–189. https://doi.org/10.1007/BF00130708

[28] Amir Noohi, Mostafa Derispour, and Antonio Barbalace. 2025. RMAI: Rethinking Memory for AI (Inference): In-Kernel Remote Shared Memory as a Software Alternative to CXL. In *Proceedings of the 5th Workshop on Machine Learning and Systems* (World Trade Center, Rotterdam, Netherlands) *(EuroMLSys '25)*. Association for Computing Machinery, New York, NY, USA, 122–131. https://doi.org/10.1145/3721146.3721954

[29] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative LLM inference using phase splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

[30] Ruoyu Qin, Zheming Li, Weiyang He, Mingxing Zhang, Yongwei Wu, Weimin Zhu, and Haibo Chen. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbots. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association.

[31] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. 2022. Scaling Language Models: Methods, Analysis & Insights from Training Gopher. *arXiv preprint arXiv:2112.11446* (2022).

[32] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 20, 16 pages.

[33] Xiaozhe Ren, Pingyi Zhou, Xinfan Meng, Xinjing Huang, Yadao Wang, Weichao Wang, Pengfei Li, Xiaoda Zhang, Alexander Podolskiy, Grigory Arshinov, et al. 2023. PanGu-Σ: Towards Trillion Parameter Language Model with Sparse Heterogeneous Computing. *arXiv preprint arXiv:2303.10845* (2023).

[34] Samsung Electronics. 2022. Samsung 990 PRO NVMe SSD Specifications. https://semiconductor.samsung.com/consumer-storage/internal-ssd/990-pro/ Sequential read up to 7,450 MB/s, sequential write up to 6,900 MB/s (PCIe 4.0).

[35] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *ArXiv* abs/1909.08053 (2019). https://api.semanticscholar.org/CorpusID:202660670

[36] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, and Shrimai Prabhumoye. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *arXiv preprint arXiv:2201.11990* (2022).

[37] Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Bai, and Beidi Chen. 2024. ShadowKV: KV Cache in Shadows for High-Throughput Long-Context LLM Inference. *arXiv preprint arXiv:2410.21465* (2024).

[38] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4th ed.). Pearson.

[39] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE: Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA,

306–324. https://doi.org/10.1145/3132747.3132762

[40] Yao Wu, Jie Li, Hong Jiang, and Ke Zheng. 2024. Mell: Memory-efficient Large Language Model Serving via Multi-GPU KV Cache Management. *arXiv preprint arXiv:2405.04970* (2024).

[41] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[42] Leyang Xue, Yao Fu, Zhan Zhan, Luo Luo, et al. 2024. MoE-Infinity: Activation-Aware Expert Offloading for Efficient MoE Serving. In *Proceedings of Machine Learning and Systems (MLSys)*.

[43] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. OPT: Open Pre-trained Transformer Language Models. *arXiv preprint arXiv:2205.01068* (2022).

[44] Zhe Zhang and Yiying Zhang. 2025. InfiniStore: Distributed Disaggregated Memory for Long-Context LLM Serving. *arXiv preprint arXiv:2501.04972* (2025).

[45] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3848–3860. https://doi.org/10.14778/3611540.3611569

[46] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P Xing, Joseph E Gonzalez, Ion Stoica, and Hao Zhang. 2024. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. *arXiv preprint arXiv:2309.11998* (2024).

[47] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianxin Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 193–210.