HyRAP: Hybrid Resource-Aware Parallelism for MoE Fine-Tuning on Heterogeneous Clusters

Zhiyu Cai[a*], Haowen Xu[b], Zhenxiang Pan[a], Tianfu Pang[a],
Jianxin Huang[a], Rongzhi Qi[a], Yingchi Mao[a*], and Jie Wu[c]
[a]College of Computer Science and Software Engineering, Hohai University, Nanjing, China
[b] NR Electric Co., Ltd., Nanjing, China
[c]China Telecom Cloud Computing Research Institute, Temple University, Philadelphia, USA

***Abstract*—Distributed fine-tuning of Mixture-of-Experts (MoE) models on heterogeneous clusters requires accurate resource profiling of both server nodes and MoE models. Comprehensive and reliable profiling is challenging due to heterogeneous device configurations and significant inter-layer differences in MoE models. Existing fine-tuning methods typically rely on static hardware characteristics, such as computational capability and GPU memory capacity, making it difficult to capture runtime performance. These methods also apply a single inter-layer parallelism strategy across MoE layers, which fails to accommodate the distinct characteristics of sparse and dense layers. In addition, mismatches between computational capability and GPU memory capacity often lead to idle GPU memory during fine-tuning. To address these challenges, we propose a Hybrid Resource-Aware Parallelism for MoE Fine-Tuning on Heterogeneous Clusters (HyRAP). HyRAP performs lightweight small-scale fine-tuning to construct runtime-aware resource profiles for both server nodes and MoE models. Based on these profiles, HyRAP constructs a profile-based parallel scheduler during the offline planning stage to determine efficient parallelism strategies. During the online fine-tuning stage, HyRAP builds a new inter-layer hybrid parallelism method that assigns different parallelism strategies to sparse and dense MoE layers, together with memory-aware activation caching, to improve resource utilization and training throughput. We compare HyRAP with six baselines on both real-world and simulated heterogeneous clusters. Experimental results show that HyRAP can achieve the highest training throughput, outperforming state-of-the-art methods by up to 36% on the most heterogeneous cluster while also maintaining strong robustness.**

***Index Terms*—LLMs, Mixture of Experts, Heterogeneous Computing, Distributed Fine-tuning, Memory Utilization.**

I. Introduction

Large Language Models (LLMs) [1], [2] have been widely adopted in domain-specific applications in recent years [3]. Despite their broad applicability, full-parameter fine-tuning introduces substantial computational and memory overhead [4], [5]. The capability of a single device node, such as an edge server with limited computational capability and GPU memory capacity, is often insufficient for large-scale fine-tuning. Recent studies [6], [7] therefore organize device nodes with heterogeneous hardware configurations into a cluster to enable distributed full-parameter fine-tuning.

Zhiyu Cai and Haowen Xu contributed equally to this work. Corresponding authors: Zhiyu Cai (260407010001@hhu.edu.cn) and Yingchi Mao (yingchi-mao@hhu.edu.cn).

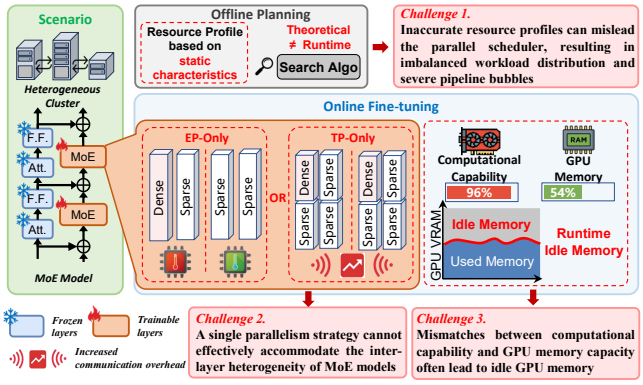

Fig. 1. Three key challenges in distributed MoE fine-tuning on heterogeneous clusters: (1) static hardware-based resource profiles may not accurately reflect runtime performance, leading to suboptimal parallel scheduling and severe pipeline bubbles; (2) a uniform parallelism strategy cannot accommodate the distinct computation and communication characteristics of dense and sparse layers; and (3) mismatches between computational capability and GPU memory capacity result in underutilized GPU memory.

Efficient distributed fine-tuning on heterogeneous clusters requires accurate resource profiles for both device nodes and MoE models. Existing systems, however, mainly rely on static or coarse-grained profiling strategies. For example, Alpa [6] builds theoretical resource profiles from static hardware characteristics, which may deviate from runtime performance under heterogeneous execution environments. Slapo [7] profiles models primarily according to parameter scale, making it difficult to capture fine-grained inter-layer differences in computation, communication, and memory demands. As illustrated by ***Challenge 1*** in Fig. 1, inaccurate resource profiles can mislead the parallel scheduler, resulting in imbalanced workload distribution and severe pipeline bubbles. This motivates a runtime-aware profiling mechanism that captures the runtime behavior of both heterogeneous device nodes and MoE layers.

Recent MoE-based fine-tuning methods have shown that activating only a subset of experts can reduce GPU resource consumption and improve performance on domain-specific tasks [8]–[12]. However, existing systems fail to account for the heterogeneous execution characteristics of different layer types, as they typically enforce a uniform parallelism strategy

across the entire MoE model. In particular, dense shared layers are computation-intensive and benefit from computation partitioning, whereas sparse expert layers are dominated by token routing and inter-expert communication. For example, GShard [10] mainly relies on Expert Parallelism (EP), which is effective for sparse expert layers but introduces frequent synchronization and excessive communication overhead when applied to dense layers. Conversely, applying Tensor Parallelism (TP) uniformly across MoE layers can fragment sparse expert computation and introduce unnecessary execution overhead. As illustrated by ***Challenge 2*** in Fig. 1, a single parallelism strategy cannot effectively accommodate the inter-layer heterogeneity of MoE models. This motivates a hybrid parallelism strategy that assigns different methods to dense and sparse layers to balance computation and communication during MoE fine-tuning.

In heterogeneous clusters, computational capacity and GPU memory capacity are often imbalanced. Since computation is saturated earlier than memory, a considerable amount of GPU memory may remain idle even when computation has become the performance bottleneck. As illustrated by ***Challenge 3*** in Fig. 1, jointly considering computational capability and GPU memory capacity does not necessarily guarantee high memory utilization. Existing memory optimization methods [13]–[15] mainly rely on pre-planned offloading or activation recomputation to reduce memory pressure. However, offloading incurs additional data transfer overhead, while recomputation increases computation cost. Moreover, these static strategies cannot effectively exploit idle GPU memory during fine-tuning. This limitation motivates a memory-aware caching mechanism that dynamically leverages idle GPU memory to reduce recomputation and improve memory efficiency.

To address these challenges, we propose HyRAP, a Hybrid Resource-Aware Parallelism for MoE Fine-Tuning on Heterogeneous Clusters. HyRAP first conducts small-scale fine-tuning to obtain runtime-aware resource profiles of both device nodes and MoE models. Based on these profiles, HyRAP constructs a profile-based scheduler that selects efficient parallelism strategies. To handle the inter-layer heterogeneity of MoE models, HyRAP introduces an inter-layer hybrid parallelism strategy, assigning different parallelism methods to dense and sparse layers. HyRAP also incorporates a memory-aware activation caching mechanism, which dynamically utilizes idle GPU memory to reduce recomputation and improve memory efficiency during fine-tuning.

The main contributions of this paper are summarized as follows:

- **Runtime-aware resource profiling.** We introduce a lightweight profiling mechanism that uses small-scale fine-tuning to obtain runtime-aware profiles of compute nodes and MoE models.
- **Inter-layer hybrid parallelism.** We propose an inter-layer hybrid parallelism method tailored to MoE fine-tuning, assigning tensor parallelism to dense layers and expert parallelism to sparse layers.
- **Memory-aware activation caching.** We develop a memory-aware activation caching mechanism to adaptively utilize idle GPU memory, thereby improving memory utilization and reducing recomputation overhead.

The remainder of this paper is organized as follows. Section II reviews related work. Section III introduces the proposed HyRAP framework. Section IV details the design of HyRAP. Section V presents the experimental evaluation and analysis. Finally, Section VI concludes the paper.

II. Related Work

A. Parallel Schedulers for Heterogeneous Clusters

Prior works on a heterogeneous cluster typically employ parallel schedulers to explore optimal parallelism strategies [16]. For example, Alpa [6] constructs a hierarchical search space and employs integer linear programming to derive inter- and intra-operator parallelism strategies. Slapo [7] unifies the search space across multiple parallelism dimensions and utilizes simulation to estimate execution costs. Metis [17] adopts a heterogeneity-aware search technique that prunes suboptimal strategies within a large configuration space. Colossal-AI [18] provides a PyTorch-compatible infrastructure to support hybrid parallelism, while SpeedLoader [19] reduces inefficiencies by minimizing redundant PCIe transfers through computation rescheduling. However, existing parallel schedulers still largely rely on theoretical profiles constructed from static hardware characteristics, which makes it difficult to capture dynamic runtime characteristics. As a result, the generated parallelism strategies may introduce pipeline bubbles and lead to inefficient GPU memory utilization. [20]

B. Inter-layer Parallelism for Distributed Training

Early distributed training frameworks, such as Megatron-LM [21] and DeepSpeed-MoE [22], established the 3D parallelism paradigm for homogeneous clusters. These frameworks introduce Data Parallelism (DP), Tensor Parallelism (TP), and Pipeline Parallelism (PP). Traditional DP replicates model parameters across nodes to enable parallel gradient computation. ZeRO [23] and Fully Sharded Data Parallel (FSDP) [24] further reduce memory overhead through parameter and optimizer-state sharding. TP partitions weight matrices to reduce memory consumption, while PP divides the model into sequential stages, where micro-batch scheduling strategies (e.g., 1F1B [25]) help mitigate pipeline bubbles. To improve the efficiency of fine-tuning sparse layers, GShard [10] and Switch Transformer [26] introduce Expert Parallelism (EP) based on All-to-All token routing. Subsequent systems, such as Tutel [27] and MegaBlocks [28], further improve execution efficiency through dynamic batching and block-sparse computation. Most parallelism strategies apply a single parallelism strategy across all layers, neglecting the substantial inter-layer differences in MoE models and often causing an imbalance between computation and communication.

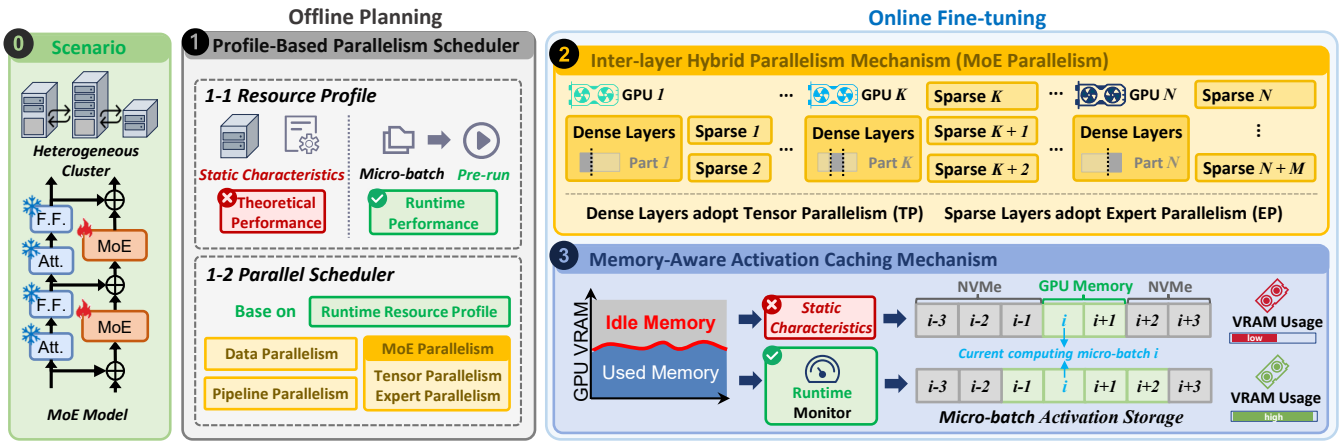

Fig. 2. The framework of HyRAP. HyRAP operates in two stages: an offline planning stage that employs the profile-based Parallel scheduler to generate a hybrid parallelism strategy based on runtime resource profiles, and an online fine-tuning stage that integrates the Inter-layer Hybrid Parallelism Mechanism with the Memory-Aware Activation Caching Mechanism to improve heterogeneous resource utilization and distributed training efficiency.

C. Memory Optimization for Distributed Fine-Tuning

GPU memory remains a critical bottleneck in distributed fine-tuning. Prior works primarily adopt memory optimization strategies based on offloading and heterogeneous execution. [29] For example, ZeRO-Offload [13] reduces memory overhead by offloading optimizer states and activations to CPU memory or NVMe storage. Teraio [14] minimizes redundant data movement through tensor lifecycle analysis. HeteGen [15] leverages heterogeneous parallelism across CPUs and GPUs to distribute memory and computational workloads. In contrast to dynamic runtime behavior, memory optimization methods are largely based on static designs and therefore cannot effectively utilize transiently available GPU memory during training.

III. The Framework of HyRAP

HyRAP consists of three major components as illustrated in Fig. 2: the Profile-Based Parallel Scheduler, the Inter-layer Hybrid Parallelism Mechanism, and the Memory-Aware Activation Caching Mechanism.

① **Profile-Based Parallel Scheduler.** HyRAP performs a lightweight small-scale fine-tuning to construct runtime-aware resource profiles for heterogeneous clusters. Instead of relying solely on theoretical hardware parameters, HyRAP measures runtime computational throughput, communication bandwidth, and model execution characteristics to accurately capture the actual runtime behavior of both the cluster and the MoE model. Based on the collected runtime resource and MoE model profiles, HyRAP employs a profile-based parallel scheduler to generate optimized hybrid parallelism strategies. The scheduler jointly coordinates DP and PP to achieve efficient workload distribution across heterogeneous nodes. MoE-specific parallelism is further handled by the Inter-layer Hybrid Parallelism Mechanism.

② **Inter-layer Hybrid Parallelism Mechanism.** Within each PP group, HyRAP employs an inter-layer hybrid parallelism to decouple the heterogeneous characteristics of MoE layers. Dense layers are partitioned across multiple GPUs using TP to accelerate computation-intensive operations. Sparse layers are distributed using EP, where experts are assigned to different GPUs without partitioning, allowing the system to reduce communication overhead and improve efficiency.

③ **Memory-Aware Activation Caching Mechanism** A runtime memory monitor continuously tracks actual GPU memory usage during fine-tuning. Instead of relying on static memory allocation strategies, HyRAP dynamically utilizes idle GPU memory to cache micro-batch activations. Activations for future micro-batches are proactively migrated from NVMe storage to GPU memory, thereby improving memory utilization efficiency and reducing runtime data transfer overhead.

IV. Design Details of HyRAP

A. Profile-Based Parallel Scheduler

1) Resource Profiles: HyRAP models a heterogeneous cluster as a set of N compute nodes. Each node n_i contains d_i GPUs, where devices within the same node are homogeneous and share identical computational capability and memory capacity. Different nodes may exhibit varying computational capabilities C_i and memory capacities M_i. All nodes are interconnected through a unified local-area network with homogeneous inter-node bandwidth, denoted as BW_{net}. Node metadata is collected using the Ray distributed framework [30].

HyRAP decomposes the MoE model into a frozen backbone $\mathcal{F}_{\text{backbone}}$ and a trainable side network $\mathcal{F}_{\text{side}}$. The backbone follows a stacked Transformer architecture parameterized by (L, h, s, V), where L, h, s, and V denote the number of layers, hidden dimension, sequence length, and vocabulary size, respectively. The side network adopts a hierarchical MoE architecture for task-specific adaptation, parameterized by $(L_{\text{side}}, h, N_{\text{moe}}, N_{\text{expert}}, r)$. Here, L_{side} denotes the number of side-network layers, which is typically aligned with the backbone depth L. The hidden dimension h represents both input and output dimensions and is shared with the backbone

architecture. N_{moe} denotes the total number of MoE layers, while N_{expert} denotes the number of sparse expert layers within each MoE model. The bottleneck dimension is represented by r. Following the standard expert architecture, each expert consists of a down-projection matrix $W_{\text{down}} \in \mathbb{R}^{h \times r}$ and an up-projection matrix $W_{\text{up}} \in \mathbb{R}^{r \times h}$.

After initializing the MoE model and system configurations, HyRAP performs a lightweight small-scale fine-tuning to construct runtime resource profiles. Two types of profiles are collected: (1) a node profile that captures the actual performance characteristics of each node, including computational throughput and I/O bandwidth, and (2) a model profile that characterizes the layer-wise computational and memory requirements of the MoE model.

To capture the discrepancy between theoretical peak performance and actual runtime behavior, the node profile is constructed through empirical benchmarking. For each node n_i, its resource configuration is represented as $R_i = (C_i, M_i)$, where C_i denotes the effective computational capability and M_i denotes the available memory capacity.

The computational capability C_i measures the effective throughput of node n_i under multi-GPU execution. C_i is estimated by executing a warm-up batch of size b_{warm}. HyRAP measures both the forward-pass execution time T_{fwd} and the backward-pass execution time T_{bwd}. The computational capability of node n_i is defined as

$$C_i = \frac{b_{\text{warm}}}{T_{\text{fwd}} + T_{\text{bwd}}}, \tag{1}$$

where C_i reflects not only the raw GPU computational capability but also the impact of intra-node communication overhead and synchronization latency.

HyRAP is aware of the available memory capacity M_i of node n_i through the small-scale fine-tuning profiling process. The memory consumption during MoE fine-tuning consists of three components: parameter memory, optimizer and gradient memory, and activation memory. For an L-layer model, the memory demand of layer l is defined as

$$M_{\text{total}}^{(l)} = M_{\text{param}}^{(l)} + M_{\text{optim}}^{(l)} + M_{\text{act}}^{(l)}, \tag{2}$$

where M_{param} denotes the memory required to store the trainable parameters of layer l.

The trainable parameter memory $M_{\text{param}}^{(l)}$ consists of dense parameters $\Theta_{\text{dense}}^{(l)}$ and sparse expert parameters $\Theta_{\text{sparse}}^{(l)}$. The parameter memory consumption is defined as

$$M_{\text{param}}^{(l)} = \Phi\Big(|\Theta_{\text{dense}}^{(l)}|\Big) + \Phi\Big(|\Theta_{\text{sparse}}^{(l)}|\Big), \tag{3}$$

where $\Phi(\cdot)$ maps the number of parameters to memory consumption according to the adopted numerical precision. Although only a subset of experts is activated for each token during execution, all expert parameters must reside in GPU memory throughout fine-tuning. Therefore, $|\Theta_{\text{sparse}}^{(l)}|$ corresponds to the complete set of expert parameters.

The optimizer memory $M_{\text{optim}}^{(l)}$ at layer l consists of gradient tensors and optimizer states associated with all trainable parameters. For a standard optimizer, the optimizer memory consumption is defined as

$$M_{\text{optim}}^{(l)} = \Phi\Big(|\Theta_{\text{grad}}^{(l)}|\Big) + \Phi\Big(|\Theta_{\text{opt}}^{(l)}|\Big), \tag{4}$$

where $|\Theta_{\text{grad}}^{(l)}|$ denotes the total number of gradient parameters, and $|\Theta_{\text{opt}}^{(l)}|$ denotes the optimizer state variables.

The activation memory $M_{\text{act}}^{(l)}$ at layer l is defined as

$$M_{\text{act}}^{(l)} = B_{\text{micro}} \times s \times h \times \delta_{\text{dtype}} \times \alpha^{(l)}, \tag{5}$$

where B_{micro} denotes the micro-batch size, while s and h denote the sequence length and hidden dimension, respectively. δ_{dtype} denotes the byte width of the activation data type. The factor $\alpha^{(l)}$ represents the number of intermediate activations retained for backward computation within layer l.

2) Parallel Scheduler: The overall training throughput is determined by the slowest DP group. Therefore, maximizing throughput is equivalent to minimizing the maximum per-iteration latency among all groups $\mathcal{G} = \{g_1, \ldots, g_D\}$:

$$\min_{\mathcal{S}} \max_{g_k \in \mathcal{G}} T_{\text{pipe}}^{k}, \tag{6}$$

where T_{pipe}^{k} denotes the iteration latency of the k-th PP group.

The scheduler is subject to two constraints. First, the structural constraint requires that each PP group g_k must encompass the complete model:

$$\sum_{n_i \in g_k} l_i = L, \quad \forall\, g_k \in \mathcal{G}. \tag{7}$$

Algorithm 1 Profile-Based Parallel Scheduler

Require: $\mathcal{N}$, L, $\{C, M\}$, BW_{net}, m, Ω
Ensure: $\mathcal{S}^*$

Initialize $T_{\min} \leftarrow \infty$, $\mathcal{S}^* \leftarrow \emptyset$, $\mathcal{N} \leftarrow \text{Sort}_{\downarrow}(\mathcal{N}, C)$
for $D \leftarrow |\mathcal{N}|$ **to** 1 **do** ▷ *Prioritize high DP*
  $\mathcal{G} \leftarrow \text{GREEDYGROUPPARTITION}(\mathcal{N}, D, C)$
  $T_{iter}^{max} \leftarrow 0, \mathcal{S}_{curr} \leftarrow \emptyset, valid \leftarrow \text{True}$
  for *group* $g_k \in \mathcal{G}$ **do**
    $C_{group}^{k} \leftarrow \sum_{n \in g_k} C_n, \mathcal{L}_g \leftarrow \emptyset$
    for $n_i \in g_k$ **do**
      $l_i \leftarrow \text{Round}(L \times (C_i / C_{group}^{k}))$
      $\mathcal{L}_g \leftarrow \mathcal{L}_g \cup \{l_i\}$
      if $M_{\text{resident}}(l_i) + M_{\text{act}}^{\min} > M_i$ **then**
        $valid \leftarrow \text{False}$; **break** ▷ *Memory pruning*
    if not $valid$ **then**
      break
    $p \leftarrow \text{List}(g_k)$
    $T_{pipe}^{k} \leftarrow \text{PIPELINELATENCY}(p, \mathcal{L}_g, C, BW_{\text{net}}, m, \Omega)$
    $T_{iter}^{max} \leftarrow \max(T_{iter}^{max}, T_{pipe}^{k})$
    $\mathcal{S}_{curr} \leftarrow \mathcal{S}_{curr} \cup (g_k, p, \mathcal{L}_g)$
  if $valid$ **and** $T_{iter}^{max} < T_{\min}$ **then**
    $T_{\min} \leftarrow T_{iter}^{max}$
    $\mathcal{S}^* \leftarrow \mathcal{S}_{curr}$
return $\mathcal{S}^*$

Second, the memory constraint requires that the total resident memory on each node does not exceed its available memory capacity M_i, as defined in the node profile:

$$M_{\text{resident}}(l_i) + M_{\text{act}}^{\min} \leq M_i, \quad \forall\, n_i \in \mathcal{N}, \tag{8}$$

where $M_{\text{resident}}(l_i)$ denotes resident model memory of layer l_i, and $M_{\text{act}}^{\min}$ denotes the minimum activation buffer requirement.

To search for an effective parallelism strategy, the scheduler follows three principles. First, a larger DP degree D is preferred, since DP incurs lower communication overhead than PP, reducing pipeline bubbles and network latency. Second, given a fixed DP degree D, nodes are partitioned into PP groups using a compute-aware greedy strategy to balance computational capability across groups and mitigate throughput degradation caused by stragglers. Third, model layers are allocated proportionally to node computational capability within each PP group (i.e., $l_i \propto C_i$).

As shown in Algorithm 1, the scheduler explores candidate DP degrees D from $|\mathcal{N}|$ to 1. For each D, node grouping and layer allocation are performed under the memory constraint in Eq. (8), and invalid configurations are pruned if any node exceeds its memory capacity. Decreasing D increases PP depth and reduces per-node layer allocation l_i.

After node groups are formed, the inner loop of Algorithm 1 (lines 5–20) performs PP group assignment. To mitigate workload imbalance caused by node heterogeneity, HyRAP adopts a compute-aware capacity allocation strategy. Specifically, node n_i within group g_k is assigned approximately $l_i \approx L \times (C_i / C_{\text{group}}^k)$ layers.

The communication latency within each PP group is estimated under a uniform interconnect bandwidth BW_{net} and a linear pipeline topology, where the per-stage latency is Ω / BW_{net}. The activation volume is defined as $\Omega = B_{\text{micro}} \cdot s \cdot h \cdot \delta_{\text{dtype}}$. Pipeline latency is then computed using Algorithm 2.

Algorithm 2 PipelineLatency

Require: p, $\mathcal{L}$, C, BW_{net}, m, Ω
Ensure: T_{pipe}

```
P ← |p|, T_stage^max ← 0
for k ← 1 to P do
    n ← p[k], l ← L[n], T_comp ← l / C_n
    if k < P then
        n_next ← p[k + 1]
        T_comm ← Ω / BW_net
    else
        T_comm ← 0
    T_stage ← T_comp + T_comm
    T_stage^max ← max(T_stage^max, T_stage)
T_pipe ← (m + P − 1) × T_stage^max
return T_pipe                    ▷ m: number of micro-batches
```

B. Inter-layer Hybrid Parallelism Mechanism

MoE models comprise both computation-intensive dense layers and communication-intensive sparse expert layers. A single parallelism strategy cannot effectively capture such inter-layer heterogeneity. Therefore, HyRAP employs a hybrid parallelism mechanism at the inter-layer granularity.

Dense shared layers involve dense computations over all tokens and are primarily dominated by large-scale matrix multiplications. To accelerate these computation-intensive operations, HyRAP employs TP to exploit intra-layer parallelism. Given an input activation $X \in \mathbb{R}^{B_{\text{micro}} \times h}$ and projection matrices $W_{\text{down}} \in \mathbb{R}^{h \times r}$ and $W_{\text{up}} \in \mathbb{R}^{r \times h}$, the weight matrices are partitioned across N_{tp} devices, where W_{down} is partitioned column-wise and W_{up} is partitioned row-wise. The local computation on device i is defined as

$$Y^{(i)} = \text{GeLU}\left(X W_{\text{down}}^{(i)}\right) W_{\text{up}}^{(i)}. \tag{9}$$

The final output is aggregated by an All-Reduce operation:

$$Y = \sum_{i=1}^{N_{\text{tp}}} Y^{(i)}. \tag{10}$$

The communication cost is primarily determined by the volume of transmitted data. Since dense layers are computation-intensive, the communication overhead can be effectively overlapped with computation, thereby enabling efficient parallel execution. For sparse MoE layers, the primary challenge lies in their inherent sparsity and relatively small parameter scale. Since the expert modules within each sparse layer contain only a limited number of parameters, forcibly applying TP would partition the computation into excessively fine-grained matrix operations. In such cases, the proportion of fixed execution overhead becomes significantly larger, reducing the achievable parallel efficiency. Therefore, HyRAP adopts EP for task-specific MoE layers. Different experts are assigned to different devices, while the weight matrix of each expert remains intact.

At the communication level, EP relies on the All-to-All communication primitive. Tokens within a batch are routed to different experts according to their task IDs. The system first performs an All-to-All Dispatch operation, in which tokens are transferred to the devices hosting their corresponding experts. After expert computation is completed, an All-to-All Combine operation is executed to return the outputs to their original devices. Although the communication pattern of All-to-All is more complex than that of All-Reduce, it avoids partitioning small matrices in sparse layers. For bottleneck layers with small hidden dimensions, TP introduces frequent fine-grained communications, where each communication step is constrained by the latency of hardware interconnects. Although EP transfers a larger volume of data, it is primarily bandwidth-sensitive rather than latency-sensitive. In heterogeneous clusters, assigning EP workloads to high-bandwidth nodes can effectively mitigate the latency accumulation caused by small-packet communications.

TP is predominantly latency-sensitive due to the frequent transmission of small messages, whereas EP is primarily bandwidth-sensitive because of the large-volume data exchange introduced by token routing and expert aggregation. By applying TP to dense layers and EP to sparse layers, HyRAP effectively aligns different parallelism strategies with

the communication characteristics of individual layer types. This design reduces unnecessary communication overhead, alleviates performance degradation caused by fine-grained synchronization, and improves inter-device communication efficiency. Consequently, HyRAP enhances the overall efficiency of distributed fine-tuning in heterogeneous clusters.

C. Memory-Aware Activation Caching Mechanism

MoE fine-tuning commonly replaces repeated backbone computation with activation reuse to reduce training overhead. However, the activation cache itself can still occupy a considerable amount of GPU memory, especially on a heterogeneous cluster where devices have different memory capacities. Static cache allocation often leaves a portion of GPU memory underutilized, while aggressive caching may trigger out-of-memory (OOM) errors. To address the above issues, HyRAP introduces a memory-aware dynamic activation caching mechanism that adaptively exploits idle GPU memory.

HyRAP treats GPU memory as the primary cache space and host storage (e.g., CPU memory or NVMe) as the auxiliary cache. A sliding-window caching strategy is employed to retain recently or imminently accessed activation tensors in GPU memory, while less frequently used tensors are asynchronously offloaded to host storage. Unlike static cache allocation, the cache capacity in HyRAP is dynamically adjusted according to runtime memory availability.

For a node n_i with memory capacity M_i, let $M_{\text{resident}}(l_i)$ denote the resident model memory of layer group l_i, and let $M_{\text{temp}}(t)$ denote the temporary runtime memory overhead at time t, including intermediate tensors and communication buffers. The available cache budget at runtime is defined as:

$$M_{\text{cache}}^{(i)}(t) = \alpha_i(t)\left(M_i - M_{\text{resident}}(l_i) - M_{\text{temp}}(t)\right), \quad (11)$$

where $\alpha_i(t) \in (0,1)$ is a dynamic safety coefficient used to prevent OOM errors while maximizing cache utilization.

Instead of using a fixed empirical value, HyRAP adaptively determines $\alpha_i(t)$ according to the runtime memory fluctuation of each node. The framework continuously monitors memory allocation variance and communication-induced memory spikes during pipeline execution. Let $\Delta M_i(t)$ denote the observed peak memory fluctuation within a recent monitoring window. The safety coefficient is estimated as:

$$\alpha_i(t) = 1 - \frac{\Delta M_i(t) + \epsilon}{M_i - M_{\text{resident}}(l_i)}, \quad (12)$$

where ϵ is a reserved safety margin for unpredictable runtime overhead. In this way, nodes with stable memory usage can safely allocate a larger cache region, while nodes experiencing significant runtime fluctuation automatically reduce cache occupancy to avoid OOM risks.

During execution, the cache manager asynchronously monitors pipeline progress and dynamically manages activation placement. For the $(j+1)$-th micro-batch, required activations are prefetched in advance. The prefetch operation is triggered under the following constraint:

$$T_{\text{prefetch}} = \frac{\text{Size}\left(\text{Cache}_{j+1}\right)}{BW_{\text{I/O}}} < T_{\text{comp}}^{(j)}, \quad (13)$$

where $BW_{\text{I/O}}$ denotes the effective bandwidth between host storage and GPU memory, and $T_{\text{comp}}^{(j)}$ denotes the computation time of the current micro-batch. The resulting overlap enables activation migration to be hidden behind computation.

To efficiently utilize the dynamic cache budget $M_{\text{cache}}^{(i)}(t)$, HyRAP adopts a greedy eviction policy. Activations required by near-future micro-batches are preferentially retained in GPU memory, while activations associated with distant micro-batches are asynchronously migrated to host storage. As GPU memory becomes available, subsequent activations can be migrated back into GPU memory to improve memory utilization.

TABLE I
SIMULATED CLUSTER TOPOLOGY

Cluster	Topology	Total Nodes
Cluster A	$\langle L_4, L_4, L_4 \rangle$	3
Cluster B	$\langle L_2, L_2, M_2, M_2 \rangle$	4
Cluster C	$\langle L_2, M_2, H_2 \rangle$	3
Cluster D	$\langle L_1, L_2, M_1, M_2, H_1 \rangle$	5

Note: In the topology tuples, L, M, and H denote the $vGPU_L, vGPU_M$, and $vGPU_H$ tiers, respectively. The notation X_k indicates a node provisioned with k instances of $vGPU_X$.

V. PERFORMANCE EVALUATION

A. Experimental Settings

Real-world Cluster. Our experiments are conducted on a three-node real-world cluster. Node-0 contains two NVIDIA A100 GPUs, and each GPU provides 80GB memory. Node-1 and Node-2 each contain two NVIDIA RTX 3090 GPUs, and each GPU provides 24GB memory. Intra-node data transfer uses PCIe 4.0 x16 interconnects with 32 GB/s bandwidth, while inter-node communication uses a 1GbE network.

Simulated Cluster. Our experiments are also conducted on the simulated clusters constructed from NVIDIA MIG partitions of A100 GPUs. NVIDIA MIG partitions are configured into three bandwidth-limited vGPU tiers: $vGPU_L$ (14 SMs, 10 GB, 4 GB/s), $vGPU_M$ (28 SMs, 20 GB, 4 GB/s), and $vGPU_H$ (42 SMs, 40 GB, 8 GB/s). We combine these $vGPU$s to build four simulated clusters (A–D) with progressively increasing heterogeneity, as shown in Table I.

Software. All evaluations are executed within isolated Docker environments (`cuda:11.8.0` on Ubuntu 20.04). We leverage Ray v2.9.0 for heterogeneous resource discovery and distributed task orchestration.

Models Setup. We choose Meta Llama-3-8B [31] as the backbone model in all experiments. To adapt the model under heterogeneous and memory-constrained settings, we adopt LoRAMoE [32] as the fine-tuning method. Unless otherwise specified, all reported results are obtained by fine-tuning Meta Llama-3-8B with LoRAMoE.

Baselines. We compare HyRAP with representative state-of-the-art systems from three categories. First, we include distributed fine-tuning frameworks that support search parallelism and workload balancing, including Alpa [6] and Slapo [7]. Second, we evaluate against MoE training systems,

TABLE II
THROUGHPUT COMPARISON (TFLOPS)

Method	Real-world Cluster	Simulated Clusters			
		Cluster A	Cluster B	Cluster C	Cluster D
Alpa	48.62	28.40	22.74	16.55	12.96
GShard	45.33	28.16	21.68	15.25	11.93
Teraio	34.15	22.39	18.67	15.04	12.69
HeteGen	36.88	24.18	20.42	17.52	15.36
MegaBlocks	61.24	30.22	27.18	22.35	20.03
Slapo	63.50	30.92	28.37	24.33	21.36
HyRAP (Ours)	**68.15**	**33.45**	**32.02**	**30.69**	**28.96**

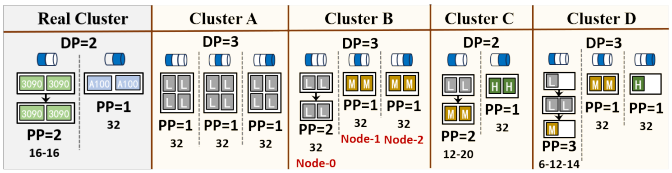

Fig. 3. This figure illustrates the logical pipeline topologies and layer assignment strategies generated by the HyRAP framework across various heterogeneous clusters. In Cluster B: two L_2 nodes are aggregated into a single pipeline group (Node-0, PP=2), while two M_2 nodes operate independently as individual replicas (Node-1 and Node-2, PP=1).

including GShard [10] and MegaBlocks [28]. Finally, to assess performance under memory-constrained heterogeneous environments, we compare with advanced memory optimization methods, including Teraio [14] and HeteGen [15].

Metrics. We choose end-to-end training throughput (TFLOPS) as the primary metric for evaluating system efficiency. TFLOPS is chosen because it jointly captures the impact of heterogeneous memory, computation, and communication resources. Throughput is defined as

$$\text{Throughput} = \frac{B_{\text{global}} \times s \times \mathcal{F}}{T_{\text{step}} \times 10^{12}}, \tag{14}$$

where B_{global} is the global batch size, s is the sequence length, $\mathcal{F}$ is the floating-point operations per token, and T_{step} is the per-iteration training time.

B. End-to-End Throughput Analysis

We evaluate the end-to-end training throughput of HyRAP and the baselines on the real-world cluster and four simulated heterogeneous clusters. The results are shown in Table II.

HyRAP achieves the highest throughput across all clusters. On the real-world cluster, it reaches 68.15 TFLOPS, demonstrating that HyRAP can utilize heterogeneous cluster resources more effectively. The real-world cluster consists of both A100 and RTX 3090 GPUs, which exhibit substantial differences in computational capability, while inter-device communication is constrained by limited PCIe bandwidth. HyRAP explicitly captures such physical heterogeneity through runtime-aware profiling and adaptive scheduling. It further combines fine-grained pipeline scheduling with heterogeneity-aware expert placement, reducing the impact of slow communication links and improving the overlap between computation and communication.

As shown in Fig. 3, HyRAP dynamically adjusts the DP and PP configurations to match different cluster topologies. Based on the runtime resource profiles, HyRAP generates a hybrid parallelism strategy that balances computational load across heterogeneous nodes. This design mitigates straggler bottlenecks and improves overall training throughput.

Alpa and GShard exhibit a clear throughput decline as cluster heterogeneity increases. In the relatively homogeneous Cluster A, both methods still maintain competitive throughput. In the highly heterogeneous Cluster D, their throughput drops to 12.96 TFLOPS and 11.93 TFLOPS, respectively. This trend is closely related to their underlying design assumptions. Both methods are primarily designed for homogeneous clusters and tend to partition model states or workloads in a relatively uniform manner. As cluster heterogeneity increases, the overall execution speed becomes increasingly constrained by the slowest nodes, leading to severe straggler effects. For GShard, the problem is further exacerbated by its reliance on All-to-All communication. Under the uneven bandwidth conditions of Cluster D, this communication pattern introduces substantial network congestion and synchronization overhead, thereby further reducing training efficiency.

Teraio and HeteGen reduce GPU memory pressure through CPU–GPU offloading, but their throughput remains limited. On the real-world cluster, Teraio reaches 34.15 TFLOPS. Its performance is constrained by PCIe bandwidth, since frequent tensor transfers reduce GPU utilization. HeteGen is more robust in highly heterogeneous settings such as Cluster D, because flexible offloading helps avoid global stalls caused by insufficient memory on individual GPUs.

MegaBlocks and Slapo maintain relatively high throughput across all clusters. MegaBlocks achieves 61.24 TFLOPS on the real-world cluster by reducing sparse execution fragmentation. However, it lacks global scheduling support for heterogeneous resources, which limits its effectiveness in the highly imbalanced Cluster D. Slapo employs integer linear programming to achieve more precise workload balancing and delivers the second-best performance across the simulated clusters.

C. Throughput Sensitivity to Node Resource Degradation

We study throughput robustness under single-node resource perturbations on Cluster B. In this experiment, the original parallelism strategy remains fixed after the perturbation, and no rescheduling is applied. This setting allows us to examine how local bottlenecks propagate through the existing strategy.

Cluster B is a representative heterogeneous setting with three DP groups. One group adopts a two-stage pipeline, while the other two groups execute without a pipeline. Based on the topology, we inject resource constraints into Node-2, which serves as an independent replica, and Node-0, which serves as the pipeline head. We then measure the resulting change in end-to-end throughput, as reported in Table III.

Cases I and II evaluate robustness under computational throttling. In Case I, reducing the compute capability of Node-2 by 30% leads to a throughput drop of 24.6%. Since synchronous DP requires all replicas to progress together, the

TABLE III
ROBUSTNESS RESULTS UNDER SINGLE-NODE RESOURCE DEGRADATION

Case	Constrained Node	Resource	TFLOPS	Percentage Decrease
Baseline	-	100% SMs/Mem	32.02	-
Case I	Node-2 (M_2)	70% SMs	24.15	↓ 24.6%
Case II	Node-0 (L_2)	70% SMs	23.88	↓ 25.4%
Case III	Node-2 (M_2)	80% Mem	30.86	↓ 3.6%
Case IV	Node-2 (M_2)	60% Mem	18.54	↓ 42.1%
Case V	Node-2 (M_2)	40% Mem	**OOM**	-

slower replica delays the global synchronization point and forces other branches to wait. In Case II, the perturbation is applied to Node-0. As the head stage of the pipeline, its slowdown propagates directly to downstream stages and introduces pipeline bubbles at Node-1. The delayed stage execution then extends to gradient synchronization across replicas. These two cases show that throughput degradation remains close to the scale of local compute loss, which indicates that the original plan already utilizes the assigned node capacities with limited slack on critical paths. Cases III–V evaluate robustness under memory capacity reduction. In Case III, GPU memory is reduced to 80% of the original capacity, and throughput decreases by only 3.6%. This result shows that the dynamic activation caching mechanism preserves a safety margin for runtime memory fluctuation. Under this condition, the cache budget remains sufficient for activation prefetch, and most data migration can still be overlapped with computation. In Case IV, GPU memory is further reduced to 60%, and throughput drops by 42.1%. At this point, the available cache space becomes too limited for stable prefetching on critical nodes. As a result, activation migration is exposed more frequently on the execution path, and the overlap between computation and communication is weakened. In Case V, GPU memory is reduced to 40%. The remaining memory can no longer accommodate the resident parameters and runtime buffers required for normal execution, and training cannot proceed.

Overall, the results reveal two properties of HyRAP under node-level perturbations without runtime rescheduling. First, the throughput loss under isolated compute degradation is propagated to the global execution with a similar magnitude. This behavior indicates that the parallelism strategy closely matches the available device capacity under normal operation and leaves little idle slack. Second, under memory pressure, HyRAP retains a clear tolerance range before throughput degrades sharply. This benefit comes from its hierarchical storage mechanism, which reserves safety space for runtime memory fluctuation when sufficient cache space is available.

D. Ablation Study

To evaluate the effectiveness of the key mechanisms in HyRAP, we conduct an ablation study on the real-world cluster. Specifically, we analyze the contributions of the Inter-layer Hybrid Parallelism Mechanism and the Memory-Aware Activation Caching Mechanism to end-to-end training throughput.

(1) Inter-layer Hybrid Parallelism Mechanism. We compare HyRAP against TP-Only and EP-Only baselines. As

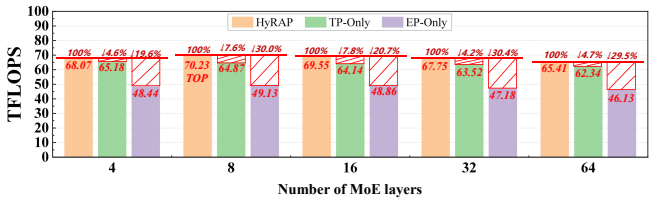

Fig. 4. Throughput Comparison of Different Parallelism Strategies Under Varying Numbers of MoE layers.

TABLE IV
THROUGHPUT AND UTILIZATION UNDER DIFFERENT MEMORY QUOTAS

Method	Quota Ratio	Batch Size = 16		Batch Size = 32	
		TFLOPS	Mem Util.	TFLOPS	Mem Util.
Static-Quota	5%	59.81	56.2%	58.15	56.5%
Static-Quota	10%	62.94	61.4%	62.12	61.8%
Static-Quota	20%	64.30	71.5%	64.53	71.9%
Static-Quota	25%	-	OOM	-	OOM
HyRAP (Ours)	-	**67.53**	**94.1%**	**67.76**	**95.2%**

shown in Fig. 4, we report the end-to-end throughput under different numbers of sparse MoE layers (N_{moe}).

HyRAP achieves the highest throughput on the heterogeneous cluster, reaching 70.23 TFLOPS. The TP-Only and EP-Only baselines experience throughput reductions of 7.6% (64.87 TFLOPS) and 30.0% (49.13 TFLOPS), respectively. These empirical results clearly demonstrate the effectiveness of the Inter-layer Hybrid Parallelism Mechanism.

Although TP-Only delivers competitive throughput, it still falls behind HyRAP by 4.7%–7.6%, primarily because synchronization overhead becomes significant under mixed micro-batches. In TP-Only, token routing requires multiple partitioned computations and All-Reduce operations, which further increases PCIe communication latency. In contrast, HyRAP employs EP together with All-to-All communication primitives. This design reduces token exchange overhead and improves the overlap between computation and communication.

EP-Only consistently delivers the lowest throughput and falls behind HyRAP by approximately 30% on average. The performance degradation of EP-Only primarily originates from dense layers. EP-Only places fully activated and computation-intensive dense layers on a single node, resulting in severe resource underutilization and global workload imbalance. These results highlight the necessity of applying TP to partition dense layers across the heterogeneous cluster.

HyRAP combines multiple parallelism paradigms by applying TP to dense layers to avoid single-node bottlenecks and employing EP for highly sparse layers to reduce communication overhead. This hybrid mapping adapts significantly better to heterogeneous clusters. HyRAP still maintains a high throughput of 65.41 TFLOPS even when the number of sparse MoE layers reaches 16 ($N_{\text{moe}} = 16$).

(2) Memory-Aware Activation Caching Mechanism. We compare HyRAP with a Static-Quota baseline that reserves a fixed portion of GPU memory for activation caching. Table IV

reports the throughput and memory utilization.

In the Static-Quota baseline, throughput improves as the cache quota increases from 5% to 20%. A larger cache quota retains more activations in GPU memory and reduces data transfer overhead. Static allocation quickly reaches a hard limit: when the quota is increased to 25%, both configurations encounter OOM errors. This result indicates that static reservation cannot safely exploit transient idle memory.

Instead, HyRAP dynamically adjusts cache occupancy according to runtime memory availability. This design increases memory utilization to 95.2% without triggering OOM. HyRAP further reduces data transfer overhead and improves execution efficiency. These results confirm the effectiveness of the Memory-Aware Activation Caching Mechanism in both memory utilization and end-to-end training performance.

VI. Conclusion

To address the low resource utilization and limited system throughput of distributed MoE fine-tuning, we propose a Hybrid Resource-Aware Parallelism for MoE Fine-Tuning on Heterogeneous Clusters (HyRAP). HyRAP effectively adapts hybrid DP and PP strategies to heterogeneous scenarios by combining actual resource profiles with profile-guided scheduling. Furthermore, HyRAP designs the inter-layer hybrid parallelism mechanism to achieve the balance between computation and communication and the memory-aware activation caching mechanism to enhance GPU memory utilization without introducing OOM risks. Experiments on both real-world and simulated heterogeneous clusters demonstrate that HyRAP can consistently achieve higher throughput and better robustness than six baselines on the heterogeneous cluster. These results indicate that HyRAP is a promising approach for improving the resource utilization and system throughput of MoE fine-tuning in heterogeneous distributed scenarios.

References

[1] H. Touvron *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[2] OpenAI, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[3] J. ZHU *et al.*, "Short-term load forecasting method based on Attention-LSTM and multi-model integration," *Electric Power Engineering Technology*, vol. 42, no. 5, pp. 138–147, 2023.

[4] L. Ouyang *et al.*, "Training language models to follow instructions with human feedback," in *Adv. Neural Inf. Process. Syst.*, 2022.

[5] Z. Song *et al.*, "Injecting domain-specific knowledge into large language models: A comprehensive survey," in *Findings Assoc. Comput. Linguistics: EMNLP*, 2025, pp. 25 297–25 311.

[6] L. Zheng *et al.*, "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," in *16th USENIX Symp. Oper. Syst. Des. Implement. (OSDI)*, 2022, pp. 559–578.

[7] H. Chen *et al.*, "Slapo: A schedule language for progressive optimization of large deep learning model training," in *Proc. 29th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2024, pp. 1095–1111.

[8] T. Zadouri *et al.*, "Pushing mixture of experts to the limit: Extremely parameter efficient moe for instruction tuning," in *Proc. 12th Int. Conf. Learn. Represent. (ICLR)*, 2024.

[9] Z. Li *et al.*, "Sparser mixture-of-adapters with cross-layer generalization," in *Proc. Conf. Nations Amer. Chapter Assoc. Comput. Linguistics: Human Lang. Technol. (NAACL-HLT)*, 2025, pp. 3988–4002.

[10] D. Lepikhin *et al.*, "Gshard: Scaling giant models with conditional computation and automatic sharding," in *Proc. 9th Int. Conf. Learn. Represent. (ICLR)*, 2021.

[11] X. Pan *et al.*, "FSMoE: A flexible and scalable training system for sparse mixture-of-experts models," in *Proc. 30th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2025, pp. 524–539.

[12] Y. Yuan *et al.*, "X-MoE: Enabling scalable training for emerging mixture-of-experts architectures on HPC platforms," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2025, pp. 1315–1331.

[13] J. Ren *et al.*, "Zero-offload: Democratizing billion-scale model training," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2021, pp. 551–564.

[14] Z. Yuan *et al.*, "Cost-efficient LLM training with lifetime-aware tensor offloading via gpudirect storage," *arXiv preprint arXiv:2506.06472*, 2025.

[15] X. Zhao *et al.*, "Hetegen: Efficient heterogeneous parallel inference for large language models on resource-constrained devices," in *Proc. 7th Annu. Conf. Mach. Learn. Syst. (MLSys)*, 2024.

[16] W. Cai *et al.*, "Shortcut-connected expert parallelism for accelerating mixture of experts," in *Proc. 42nd Int. Conf. Mach. Learn. (ICML)*, 2025, pp. 6211–6228.

[17] T. Um *et al.*, "Metis: Fast automatic distributed training on heterogeneous gpus," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2024, pp. 563–578.

[18] S. Li *et al.*, "Colossal-ai: A unified deep learning system for large-scale parallel training," in *Proc. 52nd Int. Conf. Parallel Process. (ICPP)*, 2023, pp. 766–775.

[19] Y. Zhang *et al.*, "Speedloader: An I/O efficient scheme for heterogeneous and distributed LLM operation," in *Adv. Neural Inf. Process. Syst.*, 2024.

[20] Y. LIU *et al.*, "Collaborative task allocation method for protection and control intelligent terminal in distribution networks considering elastic allocation of resources," *Electric Power Engineering Technology*, vol. 43, no. 5, pp. 100–111, 2024.

[21] M. Shoeybi *et al.*, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[22] J. Rasley *et al.*, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discov. Data Min. (KDD)*, 2020, pp. 3505–3506.

[23] S. Rajbhandari *et al.*, "Zero: memory optimizations toward training trillion parameter models," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2020, p. 20.

[24] Y. Zhao *et al.*, "Pytorch FSDP: experiences on scaling fully sharded data parallel," *Proc. VLDB Endow.*, vol. 16, no. 12, pp. 3848–3860, 2023.

[25] S. Fan *et al.*, "DAPPLE: a pipelined data parallel approach for training large models," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*, 2021, pp. 431–445.

[26] W. Fedus *et al.*, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *J. Mach. Learn. Res.*, vol. 23, pp. 120:1–120:39, 2022.

[27] C. Hwang *et al.*, "Tutel: Adaptive mixture-of-experts at scale," in *Proc. 6th Conf. Mach. Learn. Syst. (MLSys)*, 2023.

[28] T. Gale *et al.*, "Megablocks: Efficient sparse training with mixture-of-experts," in *Proc. 6th Conf. Mach. Learn. Syst. (MLSys)*, 2023.

[29] Z. HU *et al.*, "Transformer state detection and assessment method based on voiceprint compression and cost-sensitive techniques," *Electric Power Engineering Technology*, vol. 43, no. 3, pp. 209–216, 2024.

[30] P. Moritz *et al.*, "Ray: A distributed framework for emerging AI applications," in *Proc. 13th USENIX Symp. Oper. Syst. Des. Implement. (OSDI)*, 2018, pp. 561–577.

[31] A. Grattafiori *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[32] S. Dou *et al.*, "Loramoe: Alleviating world knowledge forgetting in large language models via moe-style plugin," in *Proc. 62nd Annu. Meeting Assoc. Comput. Linguistics (ACL)*, 2024, pp. 1932–1945.