Techniques for Efficient Parallelism in LLM Inference: A Survey

Hieu N. Nguyen*
Department of Computer Science and Engineering
Penn State University
State College, PA 16802
hnn5071@psu.edu

Abstract

The growth of Large Language Models (LLMs) to hundreds of billions of parameters has made distributed inference across multiple GPUs and nodes a fundamental requirement. This distribution, however, introduces a critical performance bottleneck: inter-accelerator communication. This survey provides a comprehensive analysis of the communication-bound challenges in distributed LLM inference and a structured taxonomy of techniques designed to mitigate them.

1 Introduction

Large Language Models (LLMs) have grown from hundreds of millions of parameters to hundreds of billions, with examples like GPT-3 at 175 billion [Brown et al., 2020], DeepSeek V2 at 236 billion parameters [Liu et al., 2024], and LLaMA-3 at 450 billion [Dubey et al., 2024]. Many open-source models like BLOOM-176B [Workshop et al., 2022] or OPT-175B requires a minimum of 350GB of memory just to store weights in half-precision (FP16) format. This scale has rendered single-GPU inference an impossibility for state-of-the-art models, which vastly exceed the memory capacity (e.g., 80GB on an NVIDIA A100 or H100) of any single accelerator. Moreover, this challenge is magnified by recent introduced of Large Reasoning Models such as o1 [Jaech et al., 2024] where the models generate a large number of tokens, making it difficult to fit the entire context to a single GPUs.

This physical constraint necessitates distributed inference, where the model's parameters, computation, and input context are partitioned across a cluster of GPUs or nodes or even commodity hardware over the Internet [Borzunov et al., 2023]. This partitioning is achieved through several primary parallelism strategies, which are often used in hybrid combinations such as Data Parallelism (DP) [Dean et al., 2012, Ben-Nun and Hoefler, 2019, Rajbhandari et al., 2020, Zhao et al., 2023], Tensor Parallelism (TP) , Pipeline Parallelism (PP) [Huang et al., 2019, Narayanan et al., 2021], Context Parallelism (CP) [Miao et al., 2025], and Expert Parallelism (EP) [Huang et al., 2024, Li et al., 2025].

While these strategies enable the execution of massive models, they introduce a new, dominant performance bottleneck: the communication overhead required to synchronize these distributed components. For example, in multi-GPU inference, the all-reduce collective operation required by Tensor Parallelism (TP) can contribute up to 30% of the total end-to-end latency [Facebook Engineering, 2025]. This bottleneck is exacerbated in multi-node deployments, where inter-accelerator bandwidth is lower and latency is higher.

In this survey, we aim to provide a systematic view of current literature on (1) different parallelism modes in LLM inference and the challenges/bottlenecks in each of them; (2) current techniques for increasing parallelism efficiency; and (3) discuss potential future works, such as the development of hardware-aware parallelism techniques.

*

Preprint. Under review.

2 Preliminaries

We first provide the background of LLM inference, including transformer architecture and the computation during the inference process. We also list key performance metrics for measuring the efficiency of LLM inference. Next, we discuss different modes of parallelism (e.g. Tensor Parallelism (TP), Pipeline Parallelism (PP), ...) that can be applied for these models. We also analyze the distinct communication patterns in different hardware.

2.1 LLM inference

LLM inference refers to the process of generating text or responses based on the large language model. In this study, we only consider language models that are built upon *a decoder-only transformer architecture*, meaning that the models generate text token by token or in an auto-regressive manner. A typical inference workflow starts with a request from user taking the form of a prompt (typically a text string with or without images). This request is then tokenized into a sequence of input tokens $\mathbf{x} := [x_t]_{t=1}^{P}$. The response is a completed sequence $\mathbf{y} := [y_t]_{t=1}^{T}$, where each x_t and y_t is a single token. Computing a single output token y_t requires one forward pass of the model over all the previous tokens $[\mathbf{x}|y_1, \ldots, y_{t-1}]$. To understand the possible bottlenecks during inference, one must first analyze the two distinct phases of the auto-regressive generation process common to decoder-only transformers :

1. **Prefill Phase (Input Processing):** In this phase, the model processes the entire input prompt $\mathbf{x}$ (which can be thousands of tokens long, especially in RAG use cases) in a single, large forward pass. This operation is compute-intensive, parallelizable, and can effectively saturate GPU compute units. From a communication perspective, it typically involves large, infrequent data transfers.
2. **Decode Phase (Token Generation):** In this phase, the model generates output tokens y_t one at a time, auto-regressively. Each new token generation is a matrix-vector operation, which is typically memory-bandwidth-bound, not compute-bound. The speed of this phase (measured as time-per-output-token, or TPOT) is what the user perceives as the "speed" of the LLM.

The output length T depends **non-deterministically** on the textual contents of the string. There is no such thing as a typical output length, and the request processing cost is unknown prior.

Optimizing LLM inference typically targets three key performance metrics:

- **Resource efficiency:** Effectively utilizing GPU resources is essential for improving operational performance.
- **Throughput (queries/s):** Increasing the number of requests processed to serve more users.
- **Latency:** Minimizing response delays for a smooth user experience, including:
  - *Time-to-first-token (TTFT) for prefill:* The interval before the first token appears.
  - *Time-per-output-token (TPOT) for decoding:* The delay between consecutive tokens.

These metrics highlight the distinct computational characteristics of LLM inference: prefill operations are compute-intensive, while decoding relies heavily on memory bandwidth. To tackle these challenges and facilitate the deployment of large-scale models, advanced parallelism strategies are often employed in practice.

2.2 LLM Architecture

In this section, we discuss several aspects of the mainstream LLM architecture: Transformer-Style LLMs, that affect the design of parallel algorithms when serving these models.

Transformer LMs A typical Transformer architecture (Figure 1) consists of multiple stacked Transformer blocks. Each block is composed of a Multi-Head Self-Attention (MHSA) module, a Feed-Forward Network (FFN), and Layer Normalization (LN). The input to each block is the output of the preceding block, which is processed sequentially by its submodules to produce the block

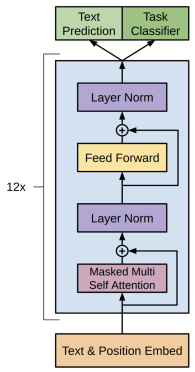


Figure 1: Transformer architecture. Figure from Radford et al. [2018]

output. Prior to the first Transformer block, a tokenizer converts the input sentence into a sequence of discrete tokens. These tokens are then mapped to continuous representations through an embedding layer, after which positional embeddings are added to encode the sequential order of the tokens.

Multi-Head Self-Attention. The MHSA module applies linear projections to the input feature matrix X to generate queries, keys, and values for each attention head:

$$Q_i = XW_i^Q, \quad K_i = XW_i^K, \quad V_i = XW_i^V,$$

where W_i^Q, W_i^K, and W_i^V denote the projection matrices for the i-th attention head. The self-attention operation is then computed as

$$Z_i = \text{Attention}(Q_i, K_i, V_i) = \text{Softmax}\left(\frac{Q_i K_i^\top}{\sqrt{d_k}}\right) V_i,$$

where d_k is the dimensionality of the queries and keys. Since the attention computation involves pairwise interactions between tokens, its computational complexity scales quadratically with the input sequence length. Finally, the outputs of all attention heads are concatenated and linearly projected to produce the MHSA output:

$$Z = \text{Concat}(Z_1, \ldots, Z_h)W^O,$$

where h is the number of attention heads and W^O is the output projection matrix.

Feed-Forward Network The Feed-Forward Network (FFN) is another core component of a Transformer block. It is typically applied after the MHSA module and is composed of two linear layers with a nonlinear activation in between. The FFN takes the output features X from the MHSA block and transforms them as

$$\text{FFN}(X) = W_2, \sigma(W_1 X),$$

where W_1 and W_2 are the weight matrices of the first and second linear layers, respectively, and $\sigma(\cdot)$ represents the activation function.

Mixture-of-Experts. Many studies focus on integrating Mixture-of-Experts (MoE) techniques into LLMs to improve performance while keeping computational costs manageable. Sparse Mixture-of-Experts (MoE) layers replace the FFN modules in standard Transformers with MoE blocks consisting of multiple expert FFNs (Figure 3(b,c)). A gating function dynamically selects the most appropriate experts for each token and routes tokens accordingly, typically activating one or two experts per token under top-1 or top-2 gating policies (Figure 3(a)). Unlike dense Transformers, where the computational cost per batch scales linearly with the total number of parameters, MoE models activate only a small subset of parameters for each token, substantially reducing the effective FLOPs. This property enables efficient training of extremely large models and has led to strong performance across vision, language, speech, and multi-task learning domains.

2.3 Hardware platforms

Several distributed AI systems exist in practice today, including NVIDIA HGX [NVIDIA Corporation, 2025], Google TPU [Zu et al., 2024], Amazon Trainium [Bshara, 2024], Cerebras CS-3 [Cerebras Systems, Inc., n.d.], and others. We also note that interconnects matter a lot (NVLink, PCIe, InfiniBand) in these systems, since communication cost depends strongly on hardware topology. For example, GPU networking uses all-to-all communication (upto 256 GPUs in DGX A100 and DGX H100) while TPU networking uses toroidal mesh.

3 Parallelism strategies and their bottlenecks

In this survey, we consider the following parallelism that can be applied for LLM inference:

3.1 Tensor Parallelism (TP)

Tensor parallelism divides each layer's tensors across multiple GPUs, allowing extremely large models (e.g., hundreds of billions of parameters) to be split among devices. Each GPU holds only a slice of each matrix, and the partial results are all-reduced to produce the correct output (Figure 2 (left)). This means TP requires frequent all-reduce synchronization points to merge computations across devices. These sync-points incur overhead determined by device interconnect bandwidth and latency [Li et al., 2024, Kim et al., 2025].

In practice, this overhead can become a primary bottleneck: for example, Meta reports that all-reduce communication can contribute up to 30% of end-to-end inference latency [Facebook Engineering, 2025]. Moreover, as models grow and more GPUs are added, the number of sync-points increases, further degrading latency. Recent studies have quantified this effect: dKim et al. [2025] measured the GPU kernel-induced all-reduce data transfer latency for LLaMA2-70B on an 8-GPU node under various settings. Their results (Figure 5) show that halving the sync-points (100% SPD) yields a 46% reduction in data-transfer latency. In summary, TP is powerful for scaling model size but is communication-bound; optimizing or reducing the sync-point overhead (e.g. via compression or dropping some synchronizations) is critical for high performance.

3.2 Pipeline Parallelism (PP)

In Pipeline Parallelism (PP), the model is partitioned sequentially, with contiguous blocks of layers placed on different GPUs or nodes (as illustrated in Figure 2 (right)). Each device (stage) processes its portion of the model and passes the intermediate results (activations) to the next stage.

PP can improve throughput, but it is limited by two main factors:

1. **Activation Transfers:** The activations (intermediate tensor outputs) can be very large for LLM layers. They must be communicated point-to-point between stages whenever a boundary is reached. These transfers are infrequent (once per micro-batch per stage) but heavy (full tensor). Thus the interconnect bandwidth becomes a bottleneck whenever GPUs hand off activations to the next stage.

2. **Pipeline "Bubbles":** Because stages operate sequentially on micro-batches, there are idle periods (bubbles) at the beginning and end of processing. Early on, the first stage runs alone until its output reaches the second stage; later, the last stage finishes before earlier stages start new work. If stages are unbalanced, some GPUs must wait for others to finish. In a steady pipeline, each GPU might not stay fully busy if one stage is slower. (For example, NVIDIA's NeMo documentation [NVIDIA Corporation, n.d.] notes that to minimize pipeline stalls one often splits each GPU's workload into multiple smaller "chunks" rather than one large block – implying that naive contiguous partitioning leads to bubbles.)

Perfectly balanced partitions and fast links are needed to mitigate these effects; otherwise, idle GPUs and large activation I/O can significantly slow inference.

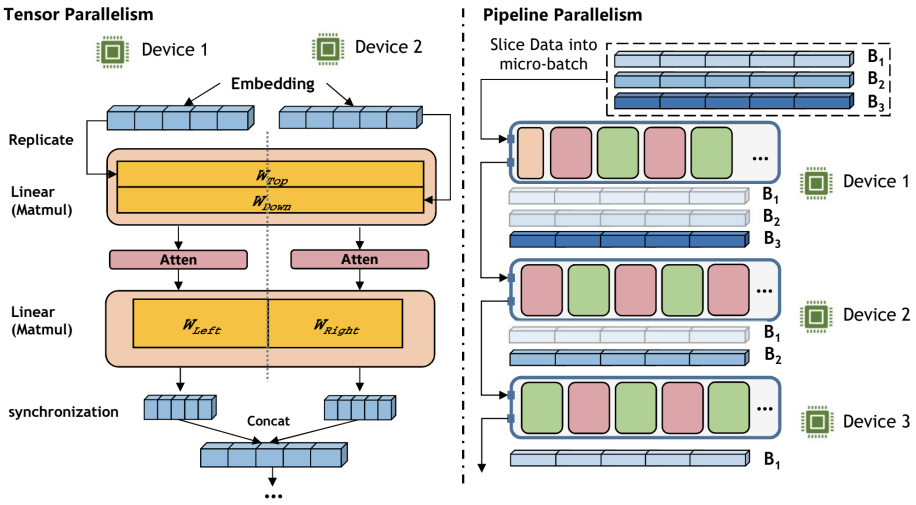


Figure 2: Tensor Parallelism (left) and Pipeline Parallelism (right). Figure from Ma et al. [2024]

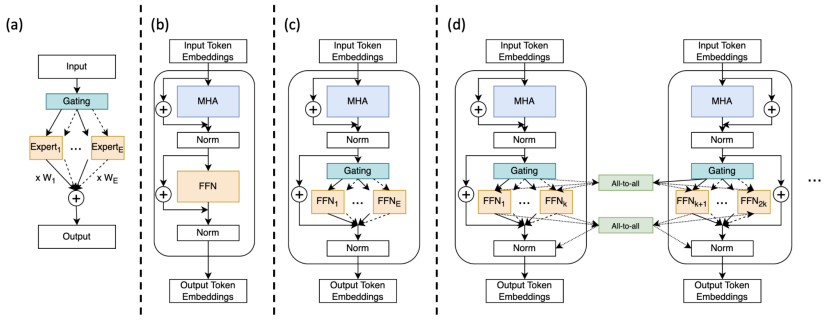


Figure 3: Illustration of MoE Models. Figure from Huang et al. [2024].

3.3 Context Parallelism (CP)

The requirement of processing long context inputs arises in many applications, including RAG, deep research, or reasoning models. Long-context LLM inference faces unique challenges that CP aims to address:

- **Computation**: Dense self-attention FLOPs scale quadratically with context length, so attention computation quickly dominates as the sequence grows.
- **Memory**: The KV-cache (keys and values for past tokens) grows linearly with context length. Managing and reading this large KV-cache on each inference step taxes memory bandwidth.

Context parallel techniques split the input tokens across devices and exchange attention tensors so that tokens in different chunks can still attend to each other. However, naive approaches face the communication challenge: when parallelizing context across multiple GPUs or hosts, tokens must attend to tokens on other ranks. This requires exchanging parts of the query/key/value tensors across devices, incurring extra cross-host latency. In long-context regimes, this inter-device communication can become significant.

3.4 Expert Parallelism (EP)

MoE models present an important trade-off: they reduce computation while incurring higher memory usage compared to traditional Transformers of comparable capacity. This trade-off necessitates

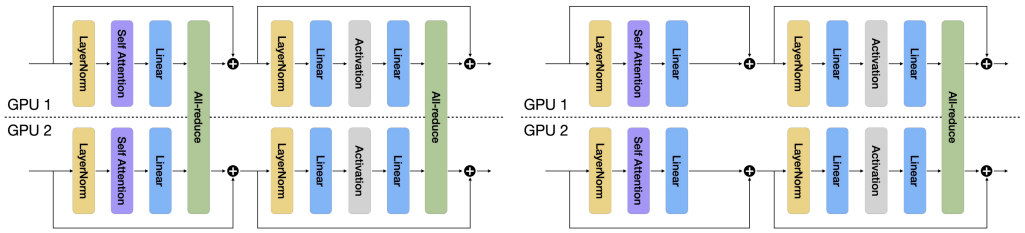
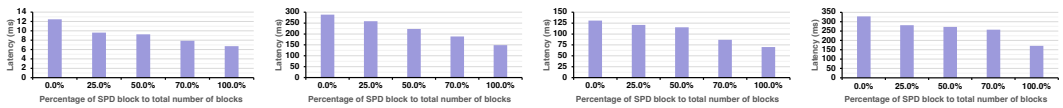


Figure 4: Sync-Point Drop (SPD) reduces the number of sync-points (from 2 to 1 per transformer decoder block). Figure from Kim et al. [2025]



(a) 1-node×8-GPUs, HBW (b) 1-node×8-GPUs, LBW (c) 2-node×4-GPUs, HBW (d) 2-node×4-GPUs, LBW

Figure 5: Data transfer latency of LLaMA2-70B distributed inference with SPD on different system settings of NVIDIA A100-80G GPU node. 'HBW' represents high bandwidth setting and 'LBW' represents low bandwidth setting for device interconnect. Input consist of batch size of 1 and sequence length of 128 is used. (Figures from Kim et al. [2025])

expert parallelism (EP) strategies to enable scalable deployment, as the large number of experts makes it impractical to place the entire model on a single host (Figure 3.d). During EP-based inference, tokens are exchanged between data-parallel and expert-parallel ranks via a two-phase all-to-all communication pattern driven by routing decisions. This communication can become a major bottleneck, especially in large-scale MoE inference.

There are several critical sources of inefficiency in MoE inference:

- **Latency:** Many EP methods exhibit much higher latencies than the dense models and can be up to 15x slower for language models (LM) and 3x slower for machine translation (MT) compared to FLOP-equivalent dense models [Huang et al., 2024]. Li et al. [2025] analyzes the DeepSpeed-MoE framework and shows that EP communication overhead can account for 47.1% to 59.2% of the total forward-pass latency, even on high-speed interconnects. This bottleneck becomes even more severe on slower hardware like PCIe or Ethernet.
- **Load Imbalance and Static Gating Inefficiency:** Many implementations use a "static gating" policy that assumes balanced expert loads. However, token distribution is often highly skewed; a few "hot" experts receive most tokens while many others remain inactive, leading to oversubscribed GPUs or idle resources [Huang et al., 2024]. When loads are imbalanced, experts must process "placeholders" (zeros) to fill capacity, wasting up to 12.8x computation in LM and 64x in MT [Huang et al., 2024].
- **Memory Pressure:** MoE models have significantly larger memory footprints due to expanded parameters and high dynamic memory use during gating/reordering [Huang et al., 2024].

In summary, expert parallelism enables deploying MoE models at scale, but the two-phase all-to-all shuffle and imbalance introduce serious overheads in latency and resource usage.

4 Techniques for reducing bottlenecks

In this section, we discuss many state-of-the-art approaches that aim to reduce bottlenecks and improve the efficiency of different parallelism modes in serving LLMs.

4.1 Tensor Parallelism

Sync-Point Drop (SPD) [Kim et al., 2025] addresses the communication overheads in TP due to sync-points by dropping synchronization in less critical parts to reduce overhead. Simply removing

sync-points to improve speed leads to numerical disparity across parallel devices. The paper introduces Sync-Point Drop (SPD), an optimization technique that selectively removes synchronization after attention outputs to improve latency while using specialized block designs and sensitivity-based strategies to minimize accuracy loss (Figure 4). The architecture is also modified to ensure that the FFN continues to receive meaningful information.

SPD categorize all blocks in an LLM into three types: **In-Sensitive Blocks (ISB)**, which exhibit minimal quality degradation under SPD and can be handled via **zero-shot dropping** by removing the sync-point without additional training; **Sensitive Blocks (SB)**, which suffer more pronounced quality degradation than ISBs and are recovered using **block-to-block distillation**, a low-cost fine-tuning approach where a student SPD block is trained to mimic a teacher TP block using an MSE loss; and **Extremely Sensitive Blocks (ESB)**, which experience sharp quality degradation when sync-points are removed and therefore require **head grouping initialization**, in which attention heads are reordered based on their functionality (attention scores) prior to performing block-to-block distillation.

The results demonstrate that these techniques are highly effective in practice: for LLaMA2-70B, up to 70% of the blocks can be converted to SPD while incurring less than 1% accuracy loss, yielding an overall latency reduction of approximately 20%. Moreover, larger models tend to contain a higher proportion of In-Sensitive Blocks, for example, 84% in OPT-66B, making them even more suitable candidates for this optimization.

Direct-data-access (DDA) algorithms Meta employs Direct-Data-Access (DDA) algorithms to bypass the latency of conventional all-reduce operations by allowing each rank to directly load data from others and perform local reductions, thereby reducing communication latency [Facebook Engineering, 2025]. Specifically, the **DDA flat algorithm** improves all-reduce performance for small messages by enabling direct memory reads across ranks and local reduction, reducing latency from $O(N)$ to $O(1)$ at the cost of increasing data movement from $O(n)$ to $O(n^2)$, while the **DDA tree algorithm** decomposes all-reduce into a reduce-scatter followed by an all-gather, leveraging direct data access at each stage to transfer the same data volume as the ring algorithm but with lower constant-factor latency, making it well suited for moderately sized messages. In practice, DDA significantly outperforms established communication libraries such as NCCL and RCCL: on AMD MI300X, it achieves overall performance comparable to NVIDIA H100, exceeding the RCCL baseline by 10–50% during decoding (small messages) and by 10–30% during prefill, which translates to an approximate 10% reduction in TPOT.

4.2 Pipeline Parallelism

In pipeline parallelism (PP), processing long sequences in a single pass can overload hardware resources, while splitting inference into very short sequences often results in poor utilization of computational capacity. To address this inefficiency, HPipe [Ma et al., 2024] targets practical deployment settings characterized by three key challenges: *communication discrepancy*, where fast intra-host links (e.g., PCIe) coexist with slow inter-host networks (on the order of $\sim$1 GB/s), limiting the effectiveness of communication-heavy parallelism strategies such as tensor parallelism; *device heterogeneity*, in which uneven compute capabilities across devices make balanced workload allocation difficult; and *extended context windows*, where long input sequences increase arithmetic intensity and render traditional micro-batch-based pipelines inefficient due to their coarse execution granularity.

HPipe addresses these challenges through a two-phase optimization workflow consisting of a *Prepare Phase* for scheduling decisions and a *Runtime Phase* for execution. First, it introduces *token-dimension pipelining*, which pipelines execution along the token dimension rather than across micro-batches, enabling finer-grained parallelism that is particularly effective for long-context tasks where sequences can be segmented without violating sequential dependencies. Second, to mitigate hardware heterogeneity, HPipe employs a dynamic programming–based *balanced distribution* strategy that determines optimal layer-level cut points, assigning fewer layers to devices with weaker compute or higher communication overhead to equalize pipeline stage latencies. Finally, HPipe performs *optimal sequence slicing* by dynamically adjusting slice lengths: since token execution time grows linearly with token position, the scheduler assigns longer slices earlier in the sequence and shorter slices later, maximizing GPU FLOPs utilization while minimizing pipeline bubbles.

Evaluated on LLaMA-7B and GPT3-2B using a heterogeneous cluster of P100 and RTX3090 GPUs, HPipe demonstrates that jointly optimizing model partitioning and sequence scheduling enables efficient scaling of LLM inference even on sub-optimal and heterogeneous hardware platforms.

4.3 Context Parallelism

Meta [Facebook Engineering, 2025] introduced two variants of context parallelism in the attention module, commonly referred to as *ring attention*, to enable efficient scaling to extremely long context lengths. In this design, the input sequence is partitioned across multiple context-parallel (CP) ranks while preserving full-context attention through selective tensor exchange. The **Pass-KV** variant assigns each rank a subset of input tokens and computes local query, key, and value tensors, after which key and value tensors are exchanged among ranks to allow attention over the entire context. The **Pass-Q** variant follows a similar procedure but instead exchanges query tensors across ranks, offering an alternative communication pattern with different bandwidth and latency trade-offs. When combined with an optimized fast-attention kernel, these context-parallelism techniques deliver strong performance for long-context inference. Meta reports processing one million tokens in under one minute on a single H100 host, and fewer than one minute for ten million tokens using distributed inference across multiple H100 hosts (e.g., 32 H100 nodes). Moreover, for LLaMA3405B, context parallelism exhibits near-linear scaling, achieving a 128K-token prefill in 3.8 seconds using 16 nodes and a 1M-token prefill in 77 seconds.

4.4 Expert Parallelism

While Mixture-of-Experts (MoE) models achieve state-of-the-art performance by decoupling model capacity from training cost, their inference deployment remains challenging due to communication overheads, memory pressure, and load imbalance across experts.

To address these issues, Huang et al. [2024] propose three complementary optimization techniques that significantly improve Expert Parallelism (EP) efficiency without degrading model quality. First, **Dynamic Gating** replaces static expert capacity allocation with an adaptive mechanism that tunes expert capacity based on actual token load. This approach simplifies token dispatch using indexing operations instead of costly batch matrix multiplications and reduces communication overhead via a lightweight "size all-to-all" notification phase, allowing experts to avoid allocating memory for empty token slots. Second, **Expert Buffering** introduces a caching strategy tailored to inference scenarios where many experts are inactive, particularly in machine translation decoders. Frequently used ("hot") experts are kept in GPU memory, while inactive experts are buffered in CPU memory using a Last-In-First-Out (LIFO) eviction policy, which empirically matches the sequential execution pattern of MoE experts. Third, **Expert Load Balancing** mitigates out-of-memory risks and performance skew by redistributing experts across GPUs based on runtime activation statistics. This includes a greedy balancing strategy to equalize expected load and an anti-correlation strategy for MT decoders that places experts with negatively correlated activation patterns on the same device. Evaluations on two production clusters (Apple and Pear) demonstrate that dynamic gating improves maximum throughput by 6.21–11.55$\times$ for language modeling and 2.58–10.98$\times$ for machine translation compared to Fairseq, while reducing activation memory by up to 79.6%. Expert buffering further reduces static GPU memory allocation by 1.47$\times$, enabling large MoE models to run on memory-constrained hardware at the cost of modest latency increases due to CPU–GPU transfers.

Complementary to these system-level optimizations, the speculative MoE (s-MoE) framework proposed by Li et al. [2025] focuses on reducing communication overhead during MoE inference through speculative execution. The framework introduces **Speculative Token Reshuffling (s-TS)**, an online mechanism that predicts a token's expert routing path using a conditional probabilistic model before the gating layer, enabling early communication planning. It further proposes **Speculative Expert Pre-grouping (s-EG)**, an offline optimization that clusters experts frequently activated by the same semantic groups of tokens onto the same device to reduce cross-device communication. In addition, s-MoE incorporates several **system-level optimizations**, including kernel fusion, communication coalescing, dispatch deduplication, and cached lookup tables to eliminate redundant data transfers when multiple selected experts reside on the same GPU. Evaluated on DeepSeek-V2 and Mixtral-8×7B under both high-bandwidth (900 GB/s) and low-bandwidth PCIe/UPI interconnects, s-TS alone improves inference throughput by up to 2.34$\times$ under TTFT and TPOT constraints and up to 5.98$\times$ under p90-TBT constraints compared to DeepSpeed-MoE. When combined with s-EG, s-MoE

achieves end-to-end throughput gains of up to 6.54×, with additional engineering optimizations contributing a further ∼7% performance improvement.

5 Challenges & Open Problems

While Tensor, Pipeline, Context, and Expert Parallelism individually provide mechanisms to scale LLM inference, there are several promising avenues for future research to further improve efficiency and scalability.

- **Hybrid Parallelism:** One natural direction is to combine multiple parallelism strategies to leverage their complementary strengths. For example, Tensor + Pipeline parallelism (TP+PP) is already commonly used in large-scale training frameworks such as Megatron-LM [Shoeybi et al., 2019]. Integrating Context Parallelism (CP) into this mix could enable efficient inference for extremely long sequences, while Expert Parallelism (EP) can be layered on top to support MoE models. Research is needed to determine optimal partitioning strategies, scheduling policies, and load balancing techniques that minimize communication overhead while maximizing GPU utilization.
- **Hardware-Aware Optimization:** Modern GPU and interconnect architectures offer opportunities for designing parallelism strategies that are aware of hardware characteristics. For instance, minimizing cross-node communication for slow links (e.g., PCIe or Ethernet) or exploiting high-bandwidth links (NVLink/HB) for critical synchronization points can significantly reduce end-to-end latency. Additionally, future designs may co-optimize memory layouts, kernel implementations (e.g., FlashAttention [Dao et al., 2022, Dao, 2024]), and communication patterns jointly with parallelism strategies.
- **End-to-End Scheduling and Co-Design:** Integrating TP, PP, CP, and EP into a unified inference engine presents challenges in scheduling computation and communication. Future systems may benefit from end-to-end co-design of software and hardware, where model partitioning, tensor sharding, pipeline scheduling, and interconnect usage are jointly optimized to minimize latency and maximize throughput.

While current parallelism strategies allow scaling LLM inference to very large models and contexts, the next generation of efficient LLM inference will likely rely on intelligent combinations of parallel modes, adaptive scheduling, and hardware-aware optimization to address the communication, memory, and computation bottlenecks that persist in current systems.

6 Related Work

Survey on Serving LLMs. Recent surveys have examined methodologies for efficient LLM serving from a machine learning systems perspective. For example, Miao et al. [2025] provide a comprehensive overview of inference techniques, including various parallelism strategies, memory management, and system-level optimizations. However, they do not focus on parallel computation and techniques for increasing efficiency.

Survey on Model Compression for Large Language Models. Model compression techniques, such as quantization, pruning, and distillation, have been extensively surveyed by Zhu et al. [2024]. These methods primarily aim to reduce model size and computational requirements while preserving accuracy. Although compression typically targets FLOPs and memory usage, some approaches also overlap with communication optimization. For instance, quantization reduces the size of tensors exchanged between devices during distributed inference, indirectly alleviating communication bottlenecks. Thus, insights from model compression research can complement parallelism strategies in achieving efficient LLM inference.

7 Conclusion

In this survey, we have reviewed the primary parallelism strategies used to scale large language model (LLM) inference: Tensor Parallelism (TP), Pipeline Parallelism (PP), Context Parallelism (CP), and

Table 1: Summary of LLM inference parallelism strategies with bottleneck mitigation techniques.

Mode	Short Description	Primary Bottlenecks	Techniques for Reducing Bottlenecks
TP	Splits model layers/matrices across multiple GPUs to handle very large models.	Frequent all-reduce sync-points; communication-bound; limited scalability.	Sync-Point Drop (SPD) [Kim et al., 2025]; Direct-Data-Access (DDA) algorithms [Facebook Engineering, 2025].
PP	Divides model into sequential blocks (stages) on different GPUs; activations passed between stages.	Pipeline bubbles; large activation transfers; load imbalance across stages.	**HPipe** [Ma et al., 2024].
CP	Splits input token sequences across devices; exchanges attention tensors (KV) for long-context.	KV-cache memory usage; quadratic attention FLOPs; cross-device tensor communication.	**Ring-Attention** with Pass-KV/Q variants, with optimized fast-attention kernels [Facebook Engineering, 2025].
EP	Distributes MoE experts across GPUs; tokens routed to subset of experts.	All-to-all token shuffle; load imbalance; static gating; high memory footprint.	Dynamic Gating; Expert Buffering; [Huang et al., 2024] Speculative MoE (s-MoE) [Li et al., 2025].

Expert Parallelism (EP). Each approach addresses specific challenges posed by increasing model size, long context lengths, or sparse expert architectures, but also introduces distinct bottlenecks in computation, memory, and communication. We next discuss many techniques to reduce these bottlenecks. An overview is provided in Table 1.

Looking forward, the convergence of these parallelism strategies into hybrid and hardware-aware approaches appears to be a promising direction. Combining multiple parallel modes can leverage their complementary advantages, while dynamic scheduling and hardware-conscious design can reduce latency and memory pressure. Further innovations in memory management, communication optimization, and adaptive routing will be critical for supporting the next generation of LLMs with hundreds of billions of parameters and extremely long contexts.

Overall, advancing efficient parallelism strategies remains essential for practical, large-scale LLM inference and for unlocking the full potential of future models in research and real-world applications.

References

T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

A. Borzunov, M. Ryabinin, A. Chumachenko, D. Baranchuk, T. Dettmers, Y. Belkada, P. Samygin, and C. Raffel. Distributed inference and fine-tuning of large language models over the internet. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL `https://openreview.net/forum?id=XmN7ZNbUAe`.

T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf`.

N. Bshara. Aws trainium: the journey for designing and optimization full stack ml hardware. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 4–4, 2024.

Cerebras Systems, Inc. Cs-3 system. https://www.cerebras.ai/system, n.d. accessed: 2025-11-12.

T. Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, et al. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407, 2024.

Facebook Engineering. Scaling llm inference: Innovations in tensor parallelism, context parallelism, and expert parallelism. https://engineering.fb.com/2025/10/17/ai-research/scaling-llm-inference-innovations-tensor-parallelism-context-parallelism-expert-parallelism/, 2025. accessed: 2025-11-16.

H. Huang, N. Ardalani, A. Sun, L. Ke, H.-H. S. Lee, S. Bhosale, C.-J. Wu, and B. Lee. Toward efficient inference for mixture of experts. *Advances in Neural Information Processing Systems*, 37: 84033–84059, 2024.

Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

A. Jaech, A. Kalai, A. Lerer, A. Richardson, A. El-Kishky, A. Low, A. Helyar, A. Madry, A. Beutel, A. Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

H.-B. Kim, D. Hoang, A. Kundu, M. Samragh, and M. Cho. Spd: Sync-point drop for efficient tensor parallelism of large language models. *arXiv preprint arXiv:2502.20727*, 2025.

Q. Li, B. Zhang, L. Ye, Y. Zhang, W. Wu, Y. Sun, L. Ma, and Y. Xie. Flash communication: Reducing tensor parallelization bottleneck for fast large language model inference, 2024. URL https://arxiv.org/abs/2412.04964.

Y. Li, P. Zheng, S. Chen, Z. Xu, Y. Lai, Y. Du, and Z. Wang. Speculative moe: Communication efficient parallel moe inference with speculative token and expert pre-scheduling. *arXiv preprint arXiv:2503.04398*, 2025.

A. Liu, B. Feng, B. Wang, B. Wang, B. Liu, C. Zhao, C. Dengr, C. Ruan, D. Dai, D. Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

R. Ma, X. Yang, J. Wang, Q. Qi, H. Sun, J. Wang, Z. Zhuang, and J. Liao. Hpipe: Large language model pipeline parallelism for long context on heterogeneous cost-effective devices. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, pages 1–9, 2024.

X. Miao, G. Oliaro, Z. Zhang, X. Cheng, H. Jin, T. Chen, and Z. Jia. Towards efficient generative large language model serving: A survey from algorithms to systems. *ACM Comput. Surv.*, 58(1), Sept. 2025. ISSN 0360-0300. doi: 10.1145/3754448. URL https://doi.org/10.1145/3754448.

D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–15, 2021.

NVIDIA Corporation. Nvidia hgx platform. https://www.nvidia.com/en-us/data-center/hgx/, 2025. Accessed: 2025-11-12.

NVIDIA Corporation. Parallelisms — nvidia nemo framework user guide. https://docs.nvidia.com/nemo-framework/user-guide/latest/nemotoolkit/features/parallelisms.html, n.d. accessed: 2025-12-19.

A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al. Improving language understanding by generative pre-training, 2018. URL https://api.semanticscholar.org/CorpusID:49313245.

S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

B. Workshop, T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

X. Zhu, J. Li, Y. Liu, C. Ma, and W. Wang. A survey on model compression for large language models. *Transactions of the Association for Computational Linguistics*, 12:1556–1577, 2024. doi: 10.1162/tacl_a_00704. URL https://aclanthology.org/2024.tacl-1.85/.

Y. Zu, A. Ghaffarkhah, H.-V. Dang, B. Towles, S. Hand, S. Huda, A. Bello, A. Kolbasov, A. Rezaei, D. Du, et al. Resiliency at scale: Managing {Google's}{TPUv4} machine learning supercomputer. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 761–774, 2024.