Distributed LLM Serving on Consumer-Grade GPUs by Reconciling Computation and Communication

Lewei Jin, Kui Zhang, Yongqi Chen, Yifan Zhuo,
Renjie Li, Yi Gao, Bowei Yang, Zhengong Cai, Wei Dong
Zhejiang University
{jinlewei,kuizhang,yqcccchen}@zju.edu.cn

Abstract

Large language models are reshaping internet services. Serving these models is often costly, as it requires multiple high-end GPUs. Consumer-grade GPUs offer cheaper computational power, providing an opportunity for more cost-efficient LLM serving.

Prior efforts have explored distributed serving at scale, primarily focusing on model deployment strategies. However, communication efficiency has emerged as a challenge due to the imbalance in data transfer volumes between the two phases of inference: prefill and decode. Prefill requests can involve transmitting up to 1000 times more data than decode requests, leading to decode requests being delayed. Consequently, servers are underutilized while waiting for decode requests. In this paper, we present MoLink, an efficient distributed LLM serving system. It splits the prolonged transmission volume of prefill requests into smaller chunks and carefully scheduling their transmission. It consists of two parts: (i) a transmission scheduling algorithm that fairly determines whether to transmit prefill or decode requests, and (ii) a chunking determination algorithm that determines the transmit volume for prefill requests just-in-time. Our evaluation demonstrates that MoLink reduces TTFT, TPOT, and latency compared to the state-of-the-art distributed LLM serving system, with a maximum reduction of up to 46%.

1 Introduction

Large language models represent a groundbreaking shift in generative AI, reshaping existing Internet services, ranging from search engines to personal assistants. Yet, these advances come with a significant challenge: serving these models can be very expensive. For example, OPT-175B requires over 350 GB of accelerator memory for inference. As a result, even basic inference for these LLMs necessitates multiple high-end GPUs within a cluster.

In contrast to high-end GPUs, consumer-grade GPUs offer more affordable computation power. For instance, an RTX 4090 delivers 330 TFLOPS at FP16 precision, compared to the 312 TFLOPS of an A100, with over 4× lower hourly pricing in cloud markets [3]. It is reported that 101 million PC GPUs were shipped in Q4 2021 [11]. Although these consumer-grade GPUs are widely deployed, they remain underutilized. This presents the opportunity for more cost-efficient LLM serving.

Prior efforts have explored distributed computing at scale. For example, Folding@Home [22] sources upwards of 40,000 Nvidia and AMD GPUs. Petals [4] studies the fault-tolerance for serving LLMs on unsteady servers. HexGen [10] optimizes the deployment of LLMs in decentralized environments. Helix [14] discovers optimal LLM deployment and request scheduling under heterogeneous clusters. These studies effectively schedule LLM deployment for distributed serving.

In this paper, we focus on the communication optimization in bandwidth-constrained environments. This is primarily due to the imbalance in transferred data volume between the two phases of inference: *prefill* and *decode*. Prefill requests can involve transmitting up to 1000 times more data than decode requests, as their volume scales with the number of processed tokens. Ideally, many decode requests can be executed concurrently while prefill data is still being transmitted.

However, existing systems [28, 14, 4, 12] suffer from transmission competition between requests at different phases. For example, the decode request may be delayed by up to one second due to the prefill request occupying the available bandwidth during transmission—even though the decode request itself requires only a few milliseconds to transmit. Consequently, the server receiving the decode request remains idle during this period.

To address the issue, we present MoLink, an efficient distributed LLM serving system. It mitigates the transmission competition by splitting the prolonged transmission volume into smaller chunks and carefully scheduling these chunks to transmit. It consists of two part: (i) an transmission scheduling algorithm that determine when to transmit the prefill or the decode requests. It also guarantees that the transmission of prefill requests are not starved. (ii) an chunking determination algorithm that determine the transmit volume for prefill requests *just-in-time*, so that the decode request is not blocked by the prolonged transmission for prefill request.

We summary our contribution as follows:

- We identify a performance bottleneck in distributed LLM serving under bandwidth constrained environments—transmission competition among requests from different inference phases.
- We propose a novel transmission scheduling strategy—*chunk transmission*—which mitigates competition by splitting large transmission volumes into smaller chunks and carefully scheduling their transmission.
- We present **MoLink**, an efficient distributed LLM serving system that incorporates several optimizations. Our evaluation shows that it reduces TTFT, TPOT, and latency, with a maximum reduction of up to 46%.

2 Background

2.1 Distributed LLM Serving

LLM inference. Modern LLMs [16, 24] predict the next token given an input sequence. This prediction involves computing a hidden representation for each token within the sequence. An LLM can take a variable number of input tokens and compute their hidden representations in parallel, and its computation workload increases superlinearly with the number of tokens processed in parallel. The prefill step deals with a new sequence, often comprising many tokens, and processes these tokens concurrently. Unlike prefill, each decoding step only processes one new token generated by the previous step.

Model parallelism. Open source LLMs now feature up to hundreds of billions of parameters, far exceeding the memory capacity of a single GPU. Consequently, serving an LLM requires multiple GPUs operating in parallel. Tensor Parallelism (TP) [23] partitions the weight of each operator among GPUs. However, it is highly sensitive to

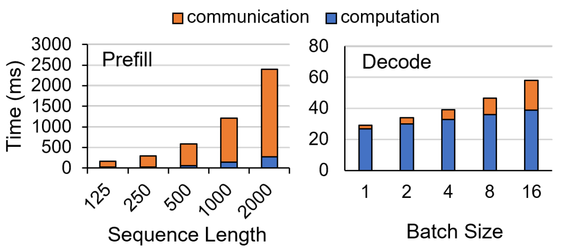


Figure 1: Computation and communication costs of prefill (left) and decode (right) with different input size for LLaMa-30B running on RTX 4090(s) servers linked with 100 mbps bandwidth.

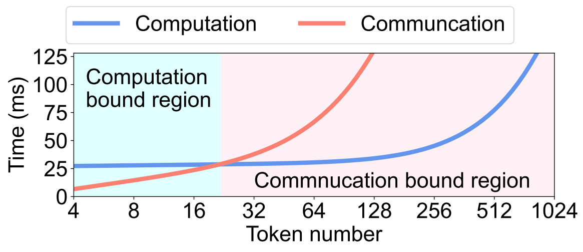


Figure 2: Computation and communication time as a function of the number of tokens. Decode tends to be in computation bound region while prefill tends to be in communication bound region.

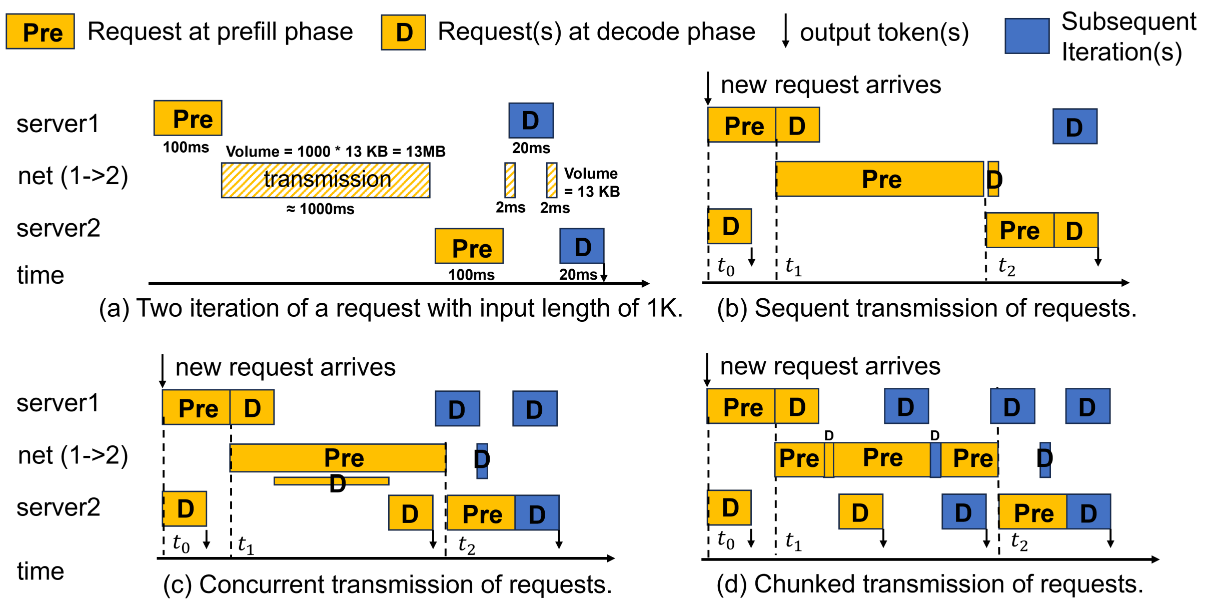
network conditions. Because it perform all-Reduce commnication. Conversely, Pipeline Parallelism (PP) [8] assigns multiple layers across GPUs to create pipeline stages. It splits inputs into multiple independent micro batches. These batches are running through the servers in the pipeline, which we referred as an *iteration* of inference. When employing pipeline parallelism for LLM serving on multiple servers, the activation tensor are transferred as the intermediate results between the servers.

3 Performance Analysis

3.1 Cost analysis of prefill and decode.

The prefill step processes a new sequence, often handling many tokens concurrently. In contrast, each decoding step processes only one new token generated by the previous step. This discrepancy leads to differences in computation and communication costs for requests at different phases.

To demonstrate this, we analyze the cost of a single inference iteration for LLaMA-30B across two RTX 4090-equipped servers connected via a 100 Mbps link. The costs are dissected into computation and communication components. Results are shown in Figure 1. We can see that the prefill phase for a single request as the prompt length increases. The results reveal that communication time dominates computation time. In contrast, the



Pre Request at prefill phase
D Request(s) at decode phase
output token(s)
Subsequent Iteration(s)
server1
net (1->2)
server2
time
Volume = 1000 * 13 KB = 13MB
transmission
≈ 1000ms
100ms
20ms
2ms
Volume = 13 KB
new request arrives
(a) Two iteration of a request with input length of 1K.
(b) Sequent transmission of requests.
(c) Concurrent transmission of requests.
(d) Chunked transmission of requests.

Figure 3: A case of transmission competition between prefill and decode for LLaMa-30B running on RTX 4090(s) linked with 100 mbps bandwidth. The prompt length is 1000. The batch size is 4.

decode phase for a batch of requests as the batch size grows, where computation time far exceeds communication time.

Figure 2 plots the trend of communication and computation time as a function of the number of tokens. We can see that the computation overhead of decode is more than the communication overhead (the number of tokens ranges from 1 to 32), which we refer to as a *computation bound task*. On the other hand, although the number of tokens for prefill can easily exceed the compute saturation point t (i.e., 256), the communication overhead is much greater than the computation overhead, which we refer to as a *communication bound task*. This suggests that interleaving the two phases offers an opportunity to overlap communication with computation and thus improve overall throughput.

3.2 Transmission competition of prefill and decode.

It is non-trivial to achieve high transmission and computation overlap. When multiple micro-batches are processed concurrently, competition between decode and prefill phases occurs. Figure 3a shows two iteration of a request with input length of 1000 under two-degree pipeline parallelism under the bandwidth of 100 mbps. For the prefill, it takes 1000ms to completely transfer the activation, while it takes only about 2ms to transfer the volume for decode.

Transmission competition. The competition often happens when sequentially processing a new arriving request in prefill phase and an existing batch of requests in decode phase. Figure 3b shows a naive transmission schedule, which we referred as serial transmission. It serially transfer the activation volume of both prefill and decode. We can see that the activation volume of prefill can be large and takes a long time to transfer, during which the transmission of decode is blocked and the server2 idles at the time between t_1 and t_2.

To address the issue, existing distributed serving systems [12, 14, 4] employ a concurrent transmission schedule, which asynchronously sends the activations of both prefill and decode requests. Figure 3c shows an example. We can see that although the volume of decode is sent once the computation is finished at t_2, the transmission of decode is delayed by several milliseconds since a large part of the bandwidth is allocated to prefill. In real practice, we observe that the delay of decode can be hundreds of times greater compared to the transmission without competition.

Opportunity. To solve this problem, our idea is to transfer the activation generated in prefill phase in chunks, which we referred as *chunking transmission*. Figure 3(d) shows an option of chunking and transmitting the activation of prefill requests. We can see that when prefill request finishes its execution, it start to transmit only a chunk of the activation generated. While this transmission is always finished before the transmission of decode, the decode are not delayed during its transmission. As a results, server2 finishes more iterations.

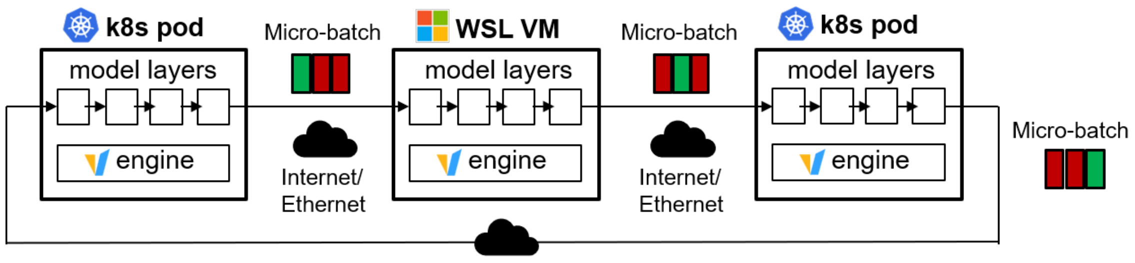


Figure 4: The architecture of MoLink. Different part of model layers are deployed on workers with the pipeline parallelism strategy. It supports accessing both Linux servers and Windows PCs.

4 MoLink

4.1 Architecture

Figure 4 shows the architecture of MoLink. Different part of model layers are deployed on workers with the pipeline parallelism strategy. We further improve the performance of MoLink by (i) scheduling the transmission of prefill and decode requests (Section 4.2). (ii) scheduling the number of micro-batches (Section 4.3).

Platform support. We present MoLink as a platform designed for serving large language models (LLMs) using distributed consumer-grade GPUs. MoLink supports both Windows PCs and Linux servers. For Linux servers, we use Kubernetes to manage Docker containers. For Windows PCs and containerized environments such as AutoDL [3], we implement lightweight Kubernetes-like functionality to manage resources and deployments.

4.2 Transmission Scheduling

To mitigate the transmission competition between prefill and decode requests, our solution is to chunk the prolonged activation of prefill request and transmit them without occupying the bandwidth for decode request. It is designed to maximize the throughput of our serving system. It should also guarantee that the activation transmission for prefill requests is not starved in the extreme situation.

Weighted-priority transmission scheduling. Algorithm 1 shows our schedule policy of transmission. We continuously check whether there are requests finished in the server, and put their volume to transfer in the queue. The volume is divided into two type of queue depending on the phase of the requests (line 5-9). We priories activation transmission for decode requests. Therefore, we choose to transmit a decode request even there exits prefill requests needing transmission (line 12-14). Then, we set a waiting weight W to guarantee that the activation transmission of prefill request will not be starved. When a decode request takes precedence over the prefill request transmission, W is counted as incremented by 1 (Line 10-11). Only if the waiting weight W are exceeding a threshold N or there is no decode requests in queue. We calculate the volume of the transmission and start the transmission for prefill requests. The waiting weight W is reset to 0 after this transmission (line 15-23).

Just-in-time chunk determination. We propose an adaptive chunking determination algorithm that performs a *just-in-time* determination of the chunk volume of activations for the next transmission. Our goal is to *predict the available time interval for chunk transmission when a transmission is*

Algorithm 1 Transmission schedule (a server)

```
1:  Initialize volume queue $vq_1, vq_2 \leftarrow \emptyset$
2:  Initialize waiting weight $W$ = 0
3:  Initialize max waiting weight $N$ = 30
4:  **While** True **do**
5:    **While** $v_{new}$ = get_next_volume() **do**
6:      **if** $v_{new}$ in phase.decode:
7:        add $v_{new}$ to $vq_1$
8:      **else**
9:        add $v_{new}$ to $vq_2$
10:     **if** $vq_1 \neq \emptyset$ and $vq_2 \neq \emptyset$
11:         $W = W + 1$
12:     **if** $vq_1 \neq \emptyset$ and $W <$ N
13:       send($vq_1[0]$)
14:       pop($vq_1$)
15:     **elif** $vq_2 \neq \emptyset$
16:       **if** $W$ >= N
17:         $v_t = vq_2[0].left$
18:       **else**
19:         $v_t$ = chunk($vq_2[0].left$)
20:       $vq_2[0].left = vq_2[0].left - v_t$
21:       **if** $vq_2[0].left$ == 0
22:         pop($vq_2$)
23:       send($v_t$)
24:       $W$ = 0
```

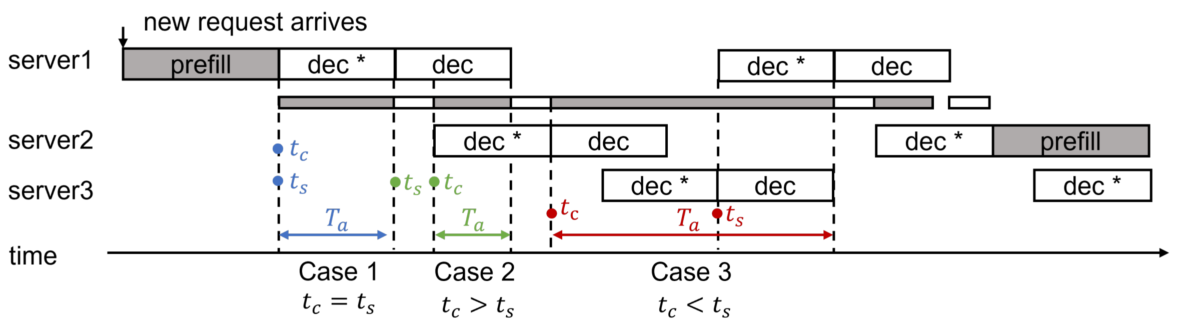


Figure 5: Determining the volume of a chunk. T_a indicates the duration of chunk transmission. The calculation of T_a has three cases, which depend on the relationship between t_c and t_s. t_c indicates the start time of the transmission. t_s indicates the start time of the current (or next) decode execution. dec* denotes the first batch of the iteration.

Variable	Description
t_c	The start time of prefill transmission.
t_s	The start time of the current (or next) decode execution.
t_f	The finish time of the current (or next) decode execution.
t_p	The finish time of the first decode batch execution in latest iteration.
T_d	The duration of the current (or next) decode execution.
T_m	The transmission overhead for first decode execution in latest iteration.
T_o	The computing overhead for first decode execution in latest iteration.
T_a	The duration of the prefill transmission.

Table 1: Notations of variables.

to start. We define variables to address the issue, whose details are shown in Table 1.

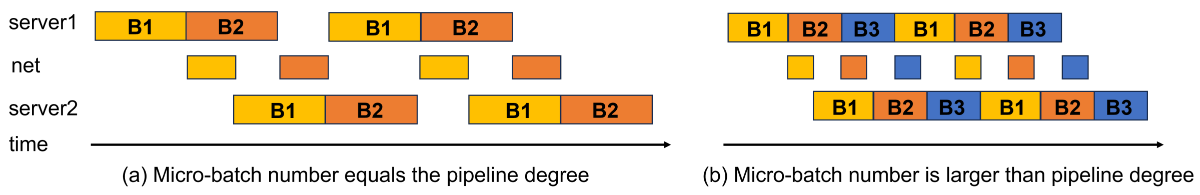
Since the available time interval is dependent on the transmission for decode requests, we show three methods for calculating the chunk volume in relation to the decode transmission. Considering the decode transmission start time t_c, and the current (or next) execution time, denoted as t_s, as illustrated in Figure 5. Our goal is to predict the available time T_a, which can be calculated as: $T_a = t_s + T_d - t_c$ where T_d represents the duration of the current (or next) decode execution.

Case 1: The start time of transmission t_c equals the start time of the current execution t_s. This situation typically occurs with the first chunk of prefill requests, where the transmission of the prefill and the execution of the subsequent decode batch begin simultaneously. In this case, the finishing time of the decode execution t_f, can be calculated as: $t_f = t_s + T_d$. Since the execution time of a decode batch primarily depends on the number of tokens processed, we express the execution duration as a function of token count: $T_d(x)$, where x denotes the number of tokens. This function is derived from profiling data collected on our real system. Therefore, the available time interval T_a can be computed as: $T_a = t_f - t_c = T_d$.

Case 2: The start time of transmission t_c is later than the start time of the current execution t_s. This situation occurs when the transmission of a chunk (e.g., from a prefill stage) begins after the completion of one of multiple serial decode batches. In this scenario, the system has already started executing a decode batch when the transmission begins. As in Case 1, the finishing time of the decode execution t_f, can be calculated as: $t_f = t_s + T_d$. However, since $t_c > t_s$, the available time interval T_a is reduced and can be computed as: $T_a = t_f - t_c = T_d + t_s - t_c$. This value is smaller than in Case 1, due to the delayed start of transmission.

Case 3: The start time of transmission t_c is earlier than the start time of the current execution t_s. This situation arises when the transmission of a chunk (e.g., from a prefill stage) begins after the completion of the final decode batch in a sequence of multiple serial decode executions. In this case, there is no decode batch currently executing in the system. Therefore, we consider the finishing time t_f of the next decode execution, which will occur in the upcoming iteration of the system.

Since LLMs follow the auto-regressive property, the execution of the next iteration can be predicted based on the execution trace of the previous iteration. Let $\{B_1, B_2, \ldots, B_N\}$ denote the sequence of decode batches in the latest iteration. The batch B_1 is expected to be the next to execute, as indicated by dec* in Figure 5. Hence, we focus on the start time (i.e., arrival time) of the next execution of B_1, denoted as t_f.



server1
net
server2
time
B1
B2
B3
(a) Micro-batch number equals the pipeline degree
(b) Micro-batch number is larger than pipeline degree

Figure 6: The impact of micro-batch number in pipeline.

We now model both the transmission overhead T_m and the computation overhead T_o required for B_1 to complete its iteration across servers.

The computation overhead T_o is calculated as:

$$T_o = \sum_{i=1}^{M} T_{d_i}(n),$$

where $T_{d_i}(n)$ represents the execution time on server i for n tokens, and is a function derived from profiling data collected per server.

The transmission overhead T_m is given by:

$$T_m = \sum_{i=1}^{M} \text{Lat}_i + \sum_{i=1}^{M-1} \frac{\text{act\_sz} \times n}{\text{Band}_i} + \frac{\text{tok\_sz} \times n}{\text{Band}_M},$$

where M is the pipeline degree (i.e., number of servers), Lat_i is the latency between server i and server $i+1$ for $i \leq M-1$, and between server M and server 1 when $i = M$, Band_i denotes the corresponding bandwidth, n is the number of tokens processed in this batch, act_sz and tok_sz are the activation and token sizes, respectively (e.g., 13312B and 2B for LLaMa-30B).

The estimated finishing time of the decode execution is: $t_f = t_p + T_o + T_m$, where t_p denotes the completion time of the first decode batch in the latest iteration. Thus, the available time interval T_a is: $T_a = t_f - t_c = t_p + T_o + T_m - t_c$. This value is typically larger than in Case 1 due to the delayed start of the next decode execution.

4.3 Extending the number of micro-batch

To fully utilize GPUs, existing systems often set the number of micro-batches equal to the degree of pipeline parallelism [12, 28]. This approach assumes that the transmission overhead between servers is negligible, given the relatively small volume of activations. However, when servers are connected via limited bandwidth, the transmission overhead for activations can increase significantly. This can lead to pipeline bubbles, reducing the overall efficiency of the system.

Figure 6a shows a example when we use a micro-batch number equals to the pipeline degree. We can see that the transmission process of intermediate activations takes times and delays the execution

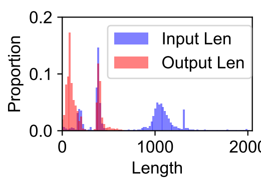


Figure 7: Distribution of input and output length.

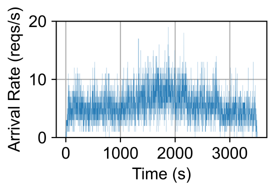


Figure 8: Arrival rate

of micro-batches (i.e., B1 and B2) in the target server. As a result, the server idles when finishing one of the micro-batch (i.e., B2) since the arrival of another micro-batch (i.e., B1) is always delayed. This under-utilization of servers naturally exists when the number of micro-batch is equals to the pipeline degree, since there is no more micro-batch fitting in the idle time of servers.

To address the issue, we extend the micro-batch to a larger number than the pipeline degree. Figure 6b shows a example. By adding a new micro-batch (i.e., B3) that sequentially executing after the B2, we can see that the transmission time of B1 are overlapped by the execution of B3. Therefore reduce the idle time of servers. The optimal number of micro-batch can be different according to both the hardware and network conditions. We set the search space of micro batch number as N to $2N$, where N indicate the degree of pipeline. We emulate the number of micro-batches to find a optimal value, since the search space of micro-batch number is limited.

5 Evaluation

Models. We evaluate MoLink on Qwen, a representative and popular open-source Transformer model family. Specifically, we use Qwen 7B [19] to study the system performance on models. We run model inference with half-precision (FP16).

Cluster setup. The distributed cluster setup has clusters that contain 3 servers, each server is equipped with a RTX 4090 GPU. Inter-node communication has an average bandwidth of 100 Mps and an average latency of 30 ms (as our profiling results on lab servers). These servers are configured to use pipeline parallelism for LLM serving.

Table 2: The comparison results at different request rate. The bandwidth is 100 mbps, and the delay is 30 ms. The percentages displayed after each value represent the comparison with the corresponding value of vLLM.

Rate	TTFT (s)			TPOT (s)			End-to-end Latency (s)		
(req/s)	vLLM	Helix	MoLink	vLLM	Helix	MoLink	vLLM	Helix	MoLink
0.7	17.3	14.2 (82%)	9.29 (54%)	3.49	3.42 (98%)	3.01 (86%)	509	497 (98%)	449 (88%)
0.3	6.20	5.66 (91%)	5.23 (84%)	1.69	1.65 (98%)	1.30 (77%)	306	299 (98%)	256 (83%)
0.2	3.74	3.42 (91%)	3.30 (88%)	0.50	0.47 (94%)	0.41 (82%)	111	106 (96%)	93 (83%)
0.1	3.39	3.27 (96%)	3.23 (95%)	0.28	0.29 (104%)	0.26 (93%)	74	75 (102%)	69 (94%)

Table 3: The comparison results at different bandwidths. The request rate is 0.3 req/s, and the delay is 30 ms. The percentages displayed after each value represent the comparison with the corresponding value of vLLM.

bandwid-	TTFT (s)			TPOT (s)			End-to-end Latency (s)		
th (mbps)	vLLM	Helix	MoLink	vLLM	Helix	MoLink	vLLM	Helix	MoLink
60	8.07	7.34 (91%)	7.32 (91%)	1.24	1.21 (97%)	1.07 (86%)	229	223 (97%)	213 (93%)
100	3.74	3.42 (92%)	3.30 (88%)	0.5	0.47 (95%)	0.41 (82%)	111	106 (96%)	93 (83%)
200	2.02	1.86 (92%)	1.83 (91%)	0.29	0.28 (97%)	0.25 (86%)	68	64 (94%)	58 (85%)
400	1.45	1.35 (93%)	1.26 (87%)	0.25	0.23 (95%)	0.21 (86%)	56	53 (95%)	47 (85%)

We also study the impact of bandwidth, latency and pipeline degree on serving metrics.

Trace. We use Azure Conversation [2] to simulate the arrival of requests. It is a representative trace of LLM inference invocations with input and output tokens. Fig. 7 shows the length distribution of the datasets. Fig. 8 shows the arrival rate of the datasets. We remove requests with input lengths larger than 2048 or output lengths larger than 1024, and scale the frequency of requests arrival to fit in the GPUs we use, which we indicate as the arrival rate in our experiment. We warm up the cluster for 1 minute and test for 30 minutes.

Metrics. We measure the latency of the LLM service using two key metrics: *Time to First Token (TTFT)*, which indicates the duration of the prefill phase, and *Time per Output Token (TPOT)*, which represents the average time taken to generate each token after the first one. Additionally, we report the *end-to-end latency* for each request. These metrics are averaged over the serving duration to provide a comprehensive view of the service's performance.

Baselines. We compare MoLink against two state-of-the-art distributed LLM serving systems.

- vLLM [12]. It is a representative LLM serving system widely used in both academia and industry. It uses concurrent transmission schedule. It asynchronously sends the prefill or decode volumes using sockets. We use version v0.7.2 [25]. It is the same version as the basic implementation of MoLink.
- Helix [14]. It is a high-throughput serving system for distributed clusters. It sequentially sends the prefill or decode volume into the ZeroMQ message queue. The underlying ZeroMQ asynchronously sends messages.

5.1 Main results

Table 2 shows the average TTFT (Time to First Token), TPOT (Tokens Per Second), and end-to-end latency for state-of-the-art systems and MoLink at different request rates. We set the rate from 0.1 to 0.7 req/s, which typically represents low to high workloads. For example, when the rate is 0.7 req/s, the TPOT of the serving systems has exceeded 3s, which is unacceptable for a serving scenario. From the table, we can see that MoLink achieves the smallest values for TTFT, TPOT, and end-to-end latency at all request rates, with a maximum reduction of 46%. Since both vLLM and Helix asynchronously send activation volumes of requests without awareness of prefill and decode, they suffer from transmission competition between prefill and decode. This can be most severe when comparable prefill and decode requests are concurrently processed, which typically happens at an arrival rate of 0.3 req/s. This is the point where MoLink achieves the most reduction in TTFT, TPOT, and end-to-end latency compared to vLLM and Helix. On the contrary, the benefits of MoLink decrease with high workloads (e.g., 0.7 req/s) and low workloads (e.g., 0.1 req/s), since a type of request (e.g., prefill or decode) dominates the execution.

Table 3 shows the average TTFT (Time to First Token), TPOT (Tokens Per Second), and end-to-end latency for MoLink at different bandwidth levels. We can see that MoLink benefits from the proposed optimization techniques across a wide range of bandwidth. Although the communication overhead can decrease with the increase of bandwidth (e.g., from 100 Mbps to 400 Mbps), resulting in the impact of transmission competition

Table 4: The comparison results at different delay. The request rate is 0.3 req/s, and the bandwidth is 100 mbps. The percentages displayed after each value represent the comparison with the corresponding value of vLLM.

Delay	TTFT (s)			TPOT (s)			End-to-end Latency (s)		
(ms)	vLLM	Helix	MoLink	vLLM	Helix	MoLink	vLLM	Helix	MoLink
10	3.25	3.13 (96%)	3.12 (96%)	0.31	0.31 (100%)	0.27 (87%)	72	72 (97%)	63 (87%)
20	3.46	3.28 (95%)	3.19 (92%)	0.41	0.40 (98%)	0.34 (83%)	94	92 (98%)	79 (84%)
30	3.74	3.42 (92%)	3.3 (88%)	0.5	0.47 (95%)	0.41 (82%)	111	106 (96%)	93 (83%)
50	3.81	3.73 (98%)	3.7 (97%)	0.67	0.65 (97%)	0.59 (88%)	143	139 (97%)	127 (89%)

Table 5: Ablation study of proposed techniques. The request rate is 0.3 req/s, the bandwidth is 100 mbps and the delay is 30ms.

	TTFT (s)	TPOT (s)	Latency (s)
All opimizations	3.30	0.41	93.07
w/ chunk transmission	3.37	0.44	99.25
w/ micro-batch extending	3.60	0.43	97.23
No opmizations	3.56	0.48	107.45

being mitigated, the communication volume is still non-negligible. This is because the prompts of prefill requests can be large in long-context scenarios, such as summarization tasks.

Table 3 shows the average TTFT (Time to First Token), TPOT (Tokens Per Second), and end-to-end latency for MoLink at different network delays. As the delay increases, the benefits of MoLink decrease. This is because pipeline bubbles become more frequent due to the more delayed arrival of micro-batches, as illustrated in Figure 6. In this experiment, we used a fixed number (5) of micro-batches. Increasing the number of micro-batches could mitigate the impact of delay to some extent.

5.2 Ablation Study

We then isolate the improvement brought by each individual technique in MoLink through an ablation study. Table 5 lists the TTFT, TPOT, and end-to-end latency that MoLink achieves when enabling each technique individually. We set the number of micro-batches to 5. We can see that, although individual techniques contribute to the reduction of TTFT, TPOT, and end-to-end latency, their combined effect is more significant. For example, chunk transmission reduces TPOT by 8.4%, while micro-batch extension reduces TPOT by 10.5%. When both techniques are combined, they reduce TPOT by 14.6%, demonstrating the efficiency of these optimizations.

6 Related Work

Machine Learning Model Serving. Many recent LLM-specific systems tackle the unpredictable execution time and high memory consumption in LLM serving. Orca [28] proposed iteration level scheduling to release resources once a request is finished. vLLM [12] introduced PageAttention to reduce the memory consumption by allocating the exact number of pages a request requires. Speculative Inference [13, 15] applies a small model to predict multiple output tokens. Splitwise [18] and DistServe [29] disaggregates the prompt and decode phase of requests. All the above works are orthogonal to our work. Sarathi [1] introduced chunked prefill, which allocates a budget to the prompt phase. However, it does not optimize the the transmission of prefill requests by chunking the volume.

Distributed LLM Serving. Several methods explored the potential of utilizing distributed GPUs for ML tasks. Some of them [9, 17] co-design the model partition and placement on a heterogeneous cluster. Learninghome [21] and DeDLOC [5] studied network-aware routing on a decentralized cluster. SWARM [20] optimize the pipeline communication in a heterogeneous network. There are also several efforts on using approximations to reduce network communication [26] or synchronization [7]. SkyPilot [27] and Mélange [6] select the best type of GPUs for a request. Petals [4] studies a decentralized pipeline parallel setup. It designs a greedy model allocation and request scheduling for a dynamical device group, losing optimization opportunities for a fixed device group. They are only focus on the model placement and request scheduling.

Limitations

At this stage,we concentrate on two limitations of this work, aiming to inspire future potential research directions.

Network Fluctuation Adaptation. Currently, we assume the network is static in our design. However, networks can be highly dynamic. Future work could explore adaptive mechanisms to handle network fluctuations, ensuring consistent performance under varying network conditions.

Fault tolerance. Currently, we assume the devices are reliable in our design. However, in real-world scenarios, devices can fail or experience faults. Future work could investigate fault-tolerance mechanisms to ensure system robustness and reliability, even in the presence of device failures.

7 Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant 62272407, in part by the “Pioneer" and “Leading Goose” RD Program of Zhejiang under Grant 2023C01033.

References

[1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. 2023. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*.

[2] Amazon. 2024. Azure conversation. `https://github.com/Azure/AzurePublicDataset/blob/master/data/AzureLLMInferenceTrace_conv.csv`.

[3] AutoDL. 2024. Autodl. `https://www.autodl.com/`.

[4] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. 2022. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188*.

[5] Michael Diskin, Alexey Bukhtiyarov, Max Ryabinin, Lucile Saulnier, Anton Sinitsin, Dmitry Popov, Dmitry V Pyrkin, Maxim Kashirin, Alexander Borzunov, Albert Villanova del Moral, and 1 others. 2021. Distributed deep learning in open collaborations. *Advances in Neural Information Processing Systems*, 34:7879–7897.

[6] Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. 2024. M\'elange: Cost efficient large language model serving by exploiting gpu heterogeneity. *arXiv preprint arXiv:2404.14527*.

[7] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. 2017. Gaia:{Geo-Distributed} machine learning approaching {LAN} speeds. In *14th USENIX symposium on networked systems design and implementation (NSDI 17)*, pages 629–647.

[8] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and 1 others. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32.

[9] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, and 1 others. 2022. Whale: Efficient giant model training over heterogeneous {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 673–688.

[10] Youhe Jiang, Ran Yan, Xiaozhe Yao, Yang Zhou, Beidi Chen, and Binhang Yuan. 2023. Hexgen: Generative inference of large language model over heterogeneous environment. *arXiv preprint arXiv:2311.11514*.

[11] jonpeddie. 2024. Q4’21 sees a nominal rise in gpu and pc shipments quarter-to-quarter.

[12] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626.

[13] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR.

[14] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. 2025. Helix: Serving large language models over heterogeneous gpus and network via max-flow. ASPLOS ’25. Association for Computing Machinery.

[15] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2023. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 1(2):4.

[16] OpenAI. 2023. Gpt-4 technical report.

[17] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. 2020. {HetPipe}: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 307–321.

[18] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium*

on Computer Architecture (ISCA), pages 118–132. IEEE.

[19] Qwen. 2024. Qwen2-7b-instruct. https://huggingface.co/Qwen/Qwen2-7B-Instruct.

[20] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. 2023. Swarm parallelism: Training large models can be surprisingly communication-efficient. In *International Conference on Machine Learning*, pages 29416–29440. PMLR.

[21] Max Ryabinin and Anton Gusev. 2020. Towards crowdsourced training of large neural networks using decentralized mixture-of-experts. *Advances in Neural Information Processing Systems*, 33:3659–3672.

[22] Michael Shirts and Vijay S Pande. 2000. Screen savers of the world unite! *Science*.

[23] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.

[24] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, and 1 others. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

[25] vllm project. 2025. vllm v0.7.2. https://github.com/vllm-project/vllm/releases/tag/v0.7.2.

[26] Jue Wang, Yucheng Lu, Binhang Yuan, Beidi Chen, Percy Liang, Christopher De Sa, Christopher Re, and Ce Zhang. 2023. Cocktailsgd: Fine-tuning foundation models over 500mbps networks. In *International Conference on Machine Learning*, pages 36058–36076. PMLR.

[27] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and 1 others. 2023. {SkyPilot}: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455.

[28] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538.

[29] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210.