LoongServe: Efficiently Serving Long-context Large Language Models with Elastic Sequence Parallelism

Bingyang Wu[1] Shengyu Liu[1] Yinmin Zhong[1] Peng Sun[2] Xuanzhe Liu[1] Xin Jin[1]
[1]*Peking University* [2]*Shanghai AI Lab*

Abstract

The context window of large language models (LLMs) is rapidly increasing, leading to a huge variance in resource usage between different requests as well as between different phases of the same request. Restricted by static parallelism strategies, existing LLM serving systems cannot efficiently utilize the underlying resources to serve variable-length requests in different phases. To address this problem, we propose a new parallelism paradigm, *elastic sequence parallelism* (ESP), to elastically adapt to the variance between different requests and phases. Based on ESP, we design and build LoongServe, an LLM serving system that (1) improves computation efficiency by elastically adjusting the degree of parallelism in real-time, (2) improves communication efficiency by reducing key-value cache migration overhead and overlapping partial decoding communication with computation, and (3) improves GPU memory efficiency by reducing key-value cache fragmentation across instances. Our evaluation under diverse real-world datasets shows that LoongServe improves the maximum throughput by up to 3.85× compared to the chunked prefill and 5.81× compared to the prefill-decoding disaggregation.

1 Introduction

The emergence of large language models (LLMs) has fundamentally pushed modern applications into a new era. These LLMs are trained on vast data to gain extraordinary capabilities in a wide range of areas, such as programming [37], chatting [1] and planning [16]. The context window is a key feature of LLMs. It is rapidly increasing to enable sophisticated reasoning about long documentation, relevant problem-solving with a large codebase, and customized generation based on long instructions [15]. Many organizations have released long-context LLMs, such as Anthropic's Claude-3 [10], Google's Gemini-1.5 [15], and UC Berkeley's Large World Model (LWM) [33], which all support a 1M context window.

Serving long-context LLMs poses significant challenges to LLM serving systems in both GPU computing and GPU memory. During the serving process, the GPU memory consumption of key-value caches grows linearly with the sequence length. When serving a *single* request with the input length of 1M tokens by a LWM model, the key-value cache alone can amount to 488GB, far exceeding the GPU memory capacity of the most advanced GPUs available today. As for the computational demand, the complexity of the attention mechanism in advanced long-context LLMs, such as LWM, is quadratic to the input sequence length, making the processing of long sequences more computationally intensive.

To accelerate the LLM computation, many works exploit different parallel dimensions. *Model parallelism* [44] partitions the model parameters across multiple GPUs to parallelize the computation. *Sequence parallelism* [25] partitions input sequences of requests across GPUs to achieve acceleration. Whether using one of them or a combination of both, existing practices decide the parallel configuration *statically* before launching the service.

However, the LLM inference workloads are highly dynamic. As the context window of LLMs increases, the variance of input lengths of requests becomes larger, leading to distinct computational demands for different requests. Furthermore, request processing is divided into two phases, the *prefill* phase and the *decoding* phase. The resource demand of the same request in different phases also varies significantly [7, 19, 20, 40, 52, 58]. Therefore, existing static parallelisms are not efficient for requests with varied input lengths, nor for the different phases of the same request.

One possible solution to mitigate these problems is to organize GPUs into multiple groups. Each group deploys an LLM instance and adopts a different parallel strategy to process sequences in a specific range of sequence length or a specific phase [20, 40, 58].

However, their static grouping strategy often mismatches with the resource demand of varied requests in different phases, because the resource demand dynamically changes at the granularity of iterations [55]. Additionally, these solutions [20, 40, 58] migrate key-value caches of *all* requests when transiting the phase, incurring extra communication overhead. Due to isolation among groups, the GPU memory across different groups cannot be utilized together to serve requests with long sequence lengths, leading to GPU memory fragmentation (§2.4).

To fundamentally address these problems, we propose *Elastic Sequence Parallelism* (ESP). Unlike existing static parallelisms, ESP dynamically decides the *Degree of Parallelism* (DoP) for requests in each iteration. For instance, during the prefill phase, ESP could set the DoP to the total number of GPUs, leveraging the entire cluster's resources to quickly process the request. Upon transiting to the relatively lightweight decoding phase, ESP can decrease the DoP to reduce communication overhead and release unnecessary resources to accelerate the processing of other requests.

Attn 机制的复杂度是 seq_len 的平方，更让 long seq 的处理变得 compute-intensive

静态 grouping 时 Iterative scheduling 导致每一个 iter 的 prefill/decode 比例动态变化，会产生 mismatch；且会导致额外通信开销

elastic scaling overhead 主要来自于 kv 迁移而非 model 重新实例化 (否则 inference latency violation)，因为 ESP 本质上是 sequence parallelism

However, fully unleashing the potential of ESP is challenging. First, if the elastic scaling overhead is excessive, it can negate the benefits of flexible resource allocation that ESP provides. For instance, if ESP involves the migration of a substantial amount of key-value caches across GPUs when scaling, it will incur significant communication overhead. Second, the dynamic loads of requests with variable sequence lengths in different phases form a complicated scheduling space, while decisions need to be made at the granularity of iterations, whose duration can be the scale of tens of milliseconds. How to find an efficient scheduling plan with such strict latency requirements remains a challenging problem.

To this end, we propose LoongServe, the first LLM inference serving system equipped with ESP to serve long-context LLMs. To realize efficient ESP, LoongServe adopts a set of novel elastic scaling mechanisms with no extra communication in both scale-up and scale-down scenarios. For the prefill phase, we propose a *proactive scaling-down* mechanism that combines the prefill phase and scaling down to reuse the communication of the prefill phase, thereby eliminating extra communication overhead. For the decoding phase, we propose a *multi-master decoding* mechanism that avoids migrating existing key-value caches and overlapping partial decoding communication with computation, thereby reducing the communication overhead. All these mechanisms manage tokens at the granularity of a single token across instances without any locality constraints, thereby eliminating GPU memory fragmentation. For online scheduling, LoongServe uses a scalable four-step scheduling algorithm that considers DoP setting, batching, key-value cache placement, and elastic scaling, to make decisions at the granularity of iterations with a polynomial complexity.

Prefill 阶段为啥要 scale down? 这和前面的说法矛盾

啥是 multi-master？为啥就不用迁移 kv 了？如何 overlap decode 的 comm-comp？

以 token 为粒度为什么就可以 reduce GPU frag?

4-step 算法里 DoP 设置和 elastic scaling 不是一个东西嘛?

Experiments on real-world datasets show that LoongServe improves the maximum throughput by up to 3.85× compared to the chunked prefill and 5.81× compared to the prefill-decoding disaggregation. Furthermore, experiments show that LoongServe improves the performance of the prefill phase and decoding phase simultaneously.

In summary, we make the following contributions:

- We identify the limitations of existing solutions in serving long-context LLMs and propose Elastic Sequence Parallelism (ESP) as a solution.
- We propose a set of efficient elastic scaling mechanisms and a scalable scheduling algorithm to unleash the potential of ESP.
- We evaluate LoongServe comprehensively to show its effectiveness compared to state-of-the-art solutions.

2 Background and Motivations

2.1 The Process of LLM Inference

Most of the popular LLMs [2, 10, 38, 49, 50] adopt the Transformer architecture [51]. The model consists of a stack of transformer layers, each containing an attention layer and a feed-forward network (FFN) layer. Attention layers make tokens in a request interact with other tokens, and FFN layers process requests token-wisely. In each iteration, given the tokens before it, the model predicts the next token. To avoid redundant computation, LLM serving systems cache the intermediate states of tokens, a.k.a *Key-Value Cache*, and reuse them for future token generation. This optimization divides the whole generation process into two phases: the *prefill phase* and the *decoding phase*. The prefill phase processes all the input tokens in a single iteration to build the key-value cache and generates the first output token, while the decoding phase only needs to compute the key-value cache for the newly generated output token. As a result, the prefill phase is much more compute-intensive than the decoding phase.

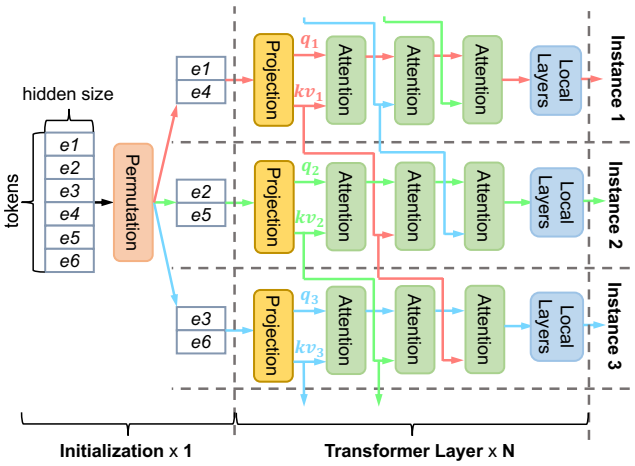


Figure 1. Sequence parallelism in the prefill phase.

2.2 Existing LLM Serving Systems

To accelerate LLM inference, existing solutions employ efficient GPU operator implementations, e.g., Flash Attention [14] and Flash Decoding [13] within a GPU, and exploit model parallelism, such as tensor parallelism [44], to partition parameters of LLM across GPUs to parallelize the computation. However, the degree of model parallelism needs to be determined before launching the system.

To mitigate the impact of the long context, chunked prefill [19, 40] splits the long context into chunks and processes them chunks by chunks with the decoding phase, but still incurs interference between two phases [58]. To avoid interference, prefill-decoding disaggregation [58] disaggregates two phases into different groups of GPUs. However, as mentioned in §1, it leads to high communication overhead due to frequent migration, GPU computation inefficiency due to resource mismatch, and GPU memory fragmentation due to isolation between groups.

2.3 Sequence Parallelism

To accelerate long-context LLM training, many LLM training systems propose sequence parallelism [11, 22, 26, 28, 30, 34].

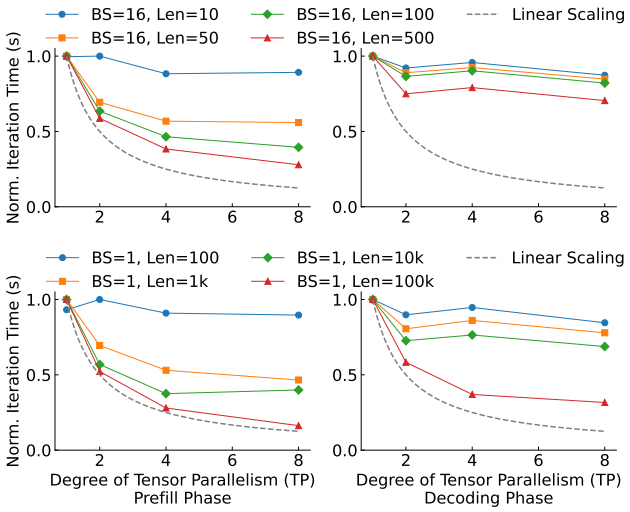


Figure 2. Scalability of requests with different lengths in the different phases.

In this work, we extend the Striped Attention [11] to the LLM serving scenario.

Figure 1 illustrates its basic idea. Before processing, the input sequence is permuted, divided into multiple segments, and then dispatched to different instances. Except for the attention layer, instances execute without communication. At the attention layer, each instance simultaneously (1) computes the attention with its local query (q_i) and key-value (kv_j) tensors and (2) sends the key-value tensors (kv_j) to its neighbor instance ($instance_{(i+1)\%n}$). After multiple rounds of this process, query tensors in each instance interact with all key-value tensors and then generate the final output to the next layer. This sequence parallelism is compatible with popular attention mechanisms, such as Multi-Head Attention (MHA) [51], Multi-Query Attention (MQA) [43], and Grouped-Query Attention (GQA) [8]. It also has the same computational complexity as tensor parallelism.

However, it cannot directly apply to the LLM serving scenario. First, it only supports the prefill phase. Second, it is used in the LLM training scenario where the degree of parallelism in each training stage is fixed. Conversely, the LLM serving systems need to handle highly dynamic inference traffic and support unique characteristics introduced by the decoding phase, including unique computational patterns of the decoding phase and key-value cache management.

2.4 Motivations and Challenges

Despite existing LLM serving systems supporting long-context LLMs (§2.2), their static nature inherently mismatches with dynamic LLM serving workloads. LLM serving workloads is particularly dynamic in two aspects. First, as the context window of LLMs increases, the resource demand during the prefill phase varies significantly across requests with different input lengths in both computation and GPU memory consumption. As shown in Figure 2, processing 100K input tokens is 105.97 times slower than processing 1K input tokens.

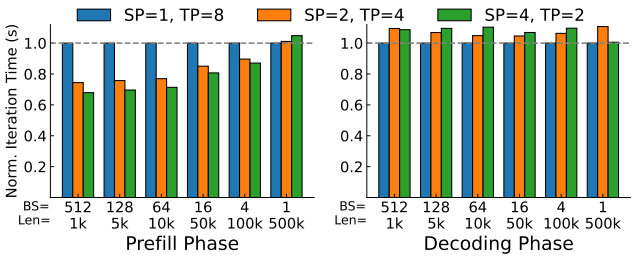


Figure 3. Comparison between fixed sequence parallelism and tensor parallelism.

The difference in GPU memory consumption across requests can be as high as 1,000,000 times when serving an LLM with a 1M context window because the size of the key-value cache is linear to the input length. Secondly, even for a single request, the resource demand of the prefill and decoding phase differs vastly. As shown in Figure 2, because the prefill phase is compute-intensive, increasing DoP can significantly accelerate it. Conversely, for the relatively compute-lightweight decoding phase, a larger DoP may lead to negligible performance improvement due to additional communication overhead. Therefore, static parallelism strategies cannot be efficient in all scenarios.

However, dynamically altering the parallelism strategy of LLM parameters, e.g., tensor parallelism, requires restarting the entire inference runtime, a process typically taking minutes and possibly longer for larger models, which is untenable for latency-sensitive serving scenarios.

Besides the inability to adapt to the dynamic workloads, they also cause GPU fragmentation issues. The reason is that the entire or most of the key-value cache of a request must reside in a single instance due to the locality constraint. This constraint leads to situations as depicted in Figure 4. Despite there being sufficient overall GPU memory (six slots), none of the instances can serve a request with six tokens due to the locality constraint.

To fundamentally address these issues, we propose elastic sequence parallelism (ESP). Our key insight is that LLM serving systems can extend SP to ESP by adding support to the decoding phase, flexibly managing the KV cache across phases, and dynamically distributing a request's input tokens across instances, allowing for an adjustable degree of parallelism (DoP) without re-partitioning LLM parameters. As shown in Figure 3, using SP with existing parallelism strategies, such as tensor parallelism, does not introduce additional overheads and may even achieve better performance for requests with different lengths in both context and decoding phases, indicating a potential to further LLM inference acceleration. Moreover, the Elastic Sequence Parallelism (ESP) facilitates the dynamic utilization of memory resources across multiple instances, thereby mitigating the problem of memory fragmentation.

While the dynamic adjustment capability of Elastic Sequence Parallelism (ESP) to meet the computational demands

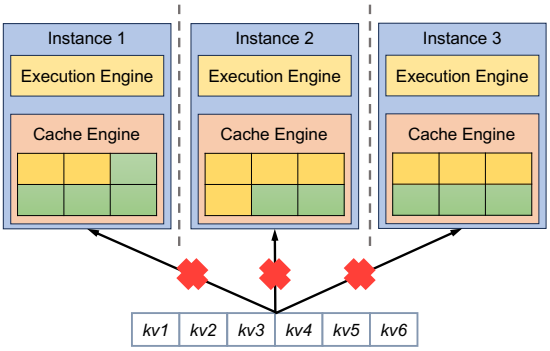


Figure 4. KV cache fragmentation of group based strategy.

of diverse requests is indeed promising, unleashing its full potential presents several challenges.

Elasticity Overhead. In the LLM inference scenario, altering the degree of sequence parallelism implies the redistribution of key-value caches among instances. When serving requests with long contexts, the size of the key-value cache can be substantial. If each adjustment in SP incurs significant communication costs, such as frequent key-value cache migration, the advantages of ESP could easily be overshadowed. Hence, there is a critical need for an efficient and cost-effective migration mechanism.

Scheduling complexity. In serving scenarios, the serving system faces hundreds of requests. It must decide on the grouping strategy, batching strategy, DoP of each batch, and request dispatching strategy for all requests in the system. Because the optimal scheduling is affected by the dynamic LLM inference workload, the scheduling decision must be made in this vast scheduling space within a latency constraint in the orders of tens of milliseconds.

3 LoongServe Overview

To address these problems, we propose a new parallelism strategy, *Elastic Sequence Parallelism* (ESP), and build a distributed LLM serving system, *LoongServe*, to fully unleash the potential of ESP. Figure 5 shows the architecture of LoongServe. LoongServe consists of a set of elastic instances and a global manager. These elastic instances can dynamically organize themselves into a set of disjoint ESP groups to process batches of requests in parallel with different configurable degrees of parallelism. They can also support efficient elastic scaling up and down without additional overhead to adapt to the varying resource demands of requests (§4). The GPU memory of LoongServe elastic instances also forms a unified distributed key-value cache pool, which can be flexibly used to store key-value tensors of requests at the granularity of a token across elastic instances to reduce GPU memory fragmentation (§4). The LoongServe global manager is responsible for managing requests, elastic instances, and the unified distributed key-value cache pool with the global view.

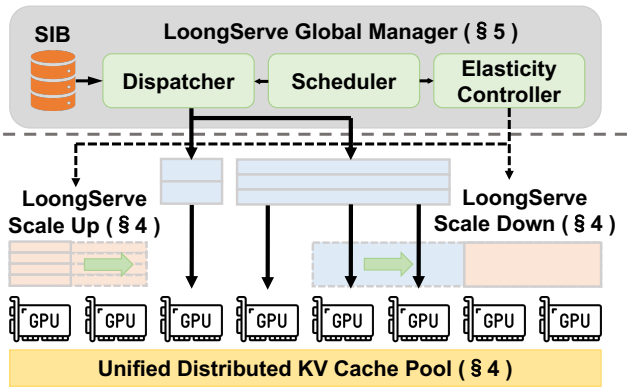


Figure 5. LoongServe system overview.

In each iteration, based on profiling results from the scaling information base (SIB), the global manager dynamically adjusts the degree of parallelism of existing batches, grouping strategy of elastic instances, batching and dispatching strategy of newly arrived requests, and placement strategy of key-value caches to improve the throughput and reduce the latency of requests in real-time (§5). In this process, the LoongServe dispatcher accompanied by the global manager dispatches newly arrived requests to a set of specific elastic instances as the global manager requires. At the same time, given the DoP and scaling plan generated by the global manager, the elasticity controller orders elastic instances to update their configuration to form the corresponding ESP groups and process requests in parallel. The key-value caches generated from LLM inference are placed at the position as specified in the scaling plan. The global manager monitors the progress of requests, the resource usage of elastic instances, and the key-value cache pool to update its decision.

4 LoongServe Elastic Instances

Elastic instances are the minimum independent execution units in LoongServe. Each maintains a replica of the model weights and employs a unified strategy of model parallelisms deployed on the equivalent number of GPUs (§5.4). Before each iteration, the global manager dynamically assigns elastic instances into multiple parallel groups, where each group handles the computation for a specific batch with different degrees of parallelism (DoP). The number of instances in a parallel group is its corresponding degree of sequence parallelism (DoSP). Elastic instances may also receive potential elastic scaling plans from the global manager, which entails the reassignment of instances' parallel groups. If so, upon completion of the iteration's computation, instances are required to realize elastic scaling to ensure that subsequent computations can proceed according to the newly defined DoP, meaning that the necessary key-value tensors are already in each instances's KV Cache pool.

Figure 6 shows the lifecycle of requests, where (B_i, I_j) indicates the j-th iteration of batch i. Because the computational complexity of the prefill phase is much higher than

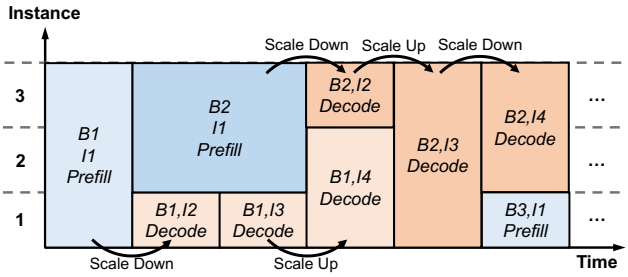


Figure 6. Lifecycle of requests.

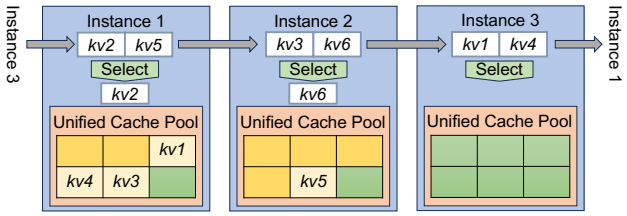


Figure 7. Elastic scale down in the prefill phase.

that of the decoding phase, it is always necessary to scale down the batch after the prefill phase ((B_1, I_1) and (B_2, I_1)). As the generated tokens become more and more, the computational complexity of the decoding phase increases and the GPU memory in a parallel group may be filled up with the KV Cache, leading to the necessity of scaling up the parallel group ((B_1, I_3) and (B_2, I_2)). It can also optionally scale down the decoding batch to leave more resources for the prefill batch ((B_2, I_3)).

To mitigate the overhead of scaling mechanisms, we have devised a set of zero-overhead ESP mechanisms, which allow for elastic scaling down in the prefill phase and scaling up in the decoding phase without any additional key-value tensor migrations. In this section, we elaborate on the design of these mechanisms. As for the optional scaling down in the decoding phase, the global manager only uses it when its benefits outweigh the overhead of scaling down (§5).

4.1 Elastic Scale-down

After a batch completes the prefill phase, its computational demand for its decoding phase decreases significantly. In this case, it is usually beneficial to scale down the size of its parallel group to release resources for other batches. Specifically, for a parallel group R with DoP d, the elastic scale-down mechanism needs to scale down the group to a new parallel group R' with DoP d', where $d' < d$. The primary challenge is to ensure that the entire key-value tensors of requests in the parallel group R are efficiently transferred to the new parallel group R'.

Existing solution: reactive migration. Existing practices [58] are reactive migration that migrates the key-value tensors from the parallel group R to the new parallel group R' after the prefill phase. However, the migration overhead is non-negligible and escalates linearly with the sequence length of requests. Specifically, for a request with an input length of 500K, the GPU consumption of the key-value tensors exceeds 2.2TB when serving a 175B LLM. Even if GPUs are connected by high-bandwidth interconnects, such as NVLINK and Infiniband, migrating a single request still spans several seconds, significantly longer than a decoding step.

Additionally, reactive migration suffers from the GPU fragmentation problem. It requires at least $O(blsh/d)$ unused GPU memory for key-value tensors in *each* instance of R to accommodate the newly generated key-value tensors before migration can proceed, where b, l, s, h are batch size, layer numbers, sequence length, hidden dimension, respectively. For instance, consider a request with an input length of 600K, and the DoP d is set to 3. When the unused slots in three instances are 100k, 200k, and 400k, respectively, the parallel group cannot handle the request, even if the total number of unused key-value slots suffice. The reason is that reactive migration first generates 200K key-value tensors on each instance, leading to an Out of Memory (OOM) error on the first instance.

A potential mitigation method is to distribute request tokens unevenly across instances based on their available unused key-value cache slots, rather than equally. However, this uneven distribution may lead to severe computational imbalances across instances, significantly delaying the entire batch's processing time, because the prefill phase is bottlecked by the slowest instance.

Our solution: proactive migration. To eliminate the migration overhead, we propose a new scale-down mechanism without additional communication overhead, called proactive migration. Our key observation is that during the prefill phase, sequence parallelism inherently circulates the key-value tensors among the parallel group. Instead of reactive migration after the prefill phase, we selectively retain key-value tensors in the KV cache pool of instances in the new parallel group R' during the prefill phase, thus achieving zero-overhead elastic scaling down.

Figure 7 shows an example. In this example, three instances form a parallel group R with DoP=3, and they are instructed to scale down to a new parallel group R' consisting of the first two instances, where the first four tokens stored in instance 1, and the rest tokens stored in instance 2. In this case, when receiving key-value tensors from the neighboring instance, besides computing the attention and sending them to the next instance, the first two instances also selectively save key-value tensors into their key-value cache pool as the instruction requires. After the computation of the prefill phase, the key-value tensors of requests are already in the KV cache pool of R'.

Compared to reactive migration, proactive migration incurs no additional migration overhead, because it reuses existing communication results. Furthermore, proactive migration eliminates memory constraints imposed by reactive

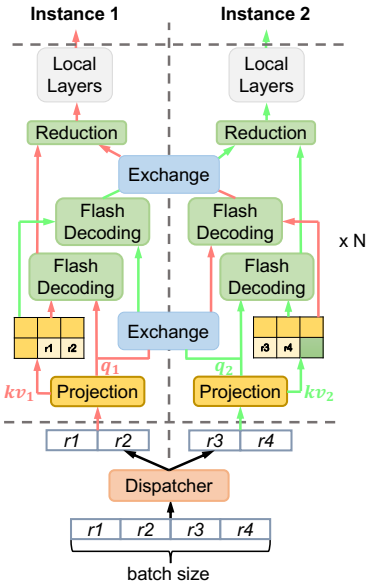


Figure 8. Elastic scale up in the decoding phase.

migration and allows for any token-level KV Cache allocation plan according to the memory availability of each instance without computational load imbalance. Besides, it also reuses existing buffer space in the sequence parallelism, avoiding additional GPU memory allocation. Because the computation is conducted layer-by-layer, this buffer only needs to temporarily stores a single layer's key-value tensors. The buffer size, i.e., $O(bsh/d)$, may be even smaller than that of the pure model parallelism, which requires $O(bsh)$.

4.2 Elastic Scale-up

Due to the decoding phase's lightweight computation pattern, the decoding batch often executes with a small DoP for overall efficiency. However, as the decoding progresses, the size of the generated key-value tensors may exceed the capacity of the KV cache pool in the parallel group, necessitating adding new instances to expand the capacity. Moreover, if the batch size is sufficiently large, the computation in the decoding phase may become compute-bound. In such cases, employing more GPUs can reduce latency, but also require an efficient scale-up operation. When scaling up the parallel group, the primary challenge is to ensure that the newly added instances can efficiently participate in the ongoing computation without incurring additional overhead.

Existing solution: entire requests migration. Previous works such as tensor-parallelism [44] (TP) and FlashDecoding [13] only support distributed decoding computations across multiple GPUs within a *single* instance. When resources of an instance, such as GPU memory, are insufficient, they have to migrate some requests in the batch to another instance, and then parallelly process them in different instances. However, migrating entire key-value caches of a request to another instance incurs significant migration overhead, even higher than a decoding step itself. Additionally, it requires the entire or most of the key-value tensors of a request must be stored in a single instance, leading to memory fragmentation issues.

Our solution: multi-master distributed decoding. We first extend sequence parallelism to the decoding phase, called single-master distributed decoding, allowing multiple instances to participate in the decoding computation of a single batch. In a parallel group, an instance is designated as the master instance, responsible for driving the computation process. At each attention layer, it first computes query and key-value tensors, saves key-value tensors into its local KV-Cache pool, and sends query tensors to other instances in the parallel group. All instances execute the local attention computation in parallel and send results back to the master instance. After that, the master instance proceeds to compute other local layers, such as the FFN layer, and starts to execute the next attention layer. As long as a single instance has enough memory to store newly generated key-value tensors, the entire batch can be processed across multiple instances by designating it as the master instance. When scaling up the parallel group, the global manager only needs to add new instances to the parallel group and instruct them to execute without migrating existing key-value tensors.

However, this approach has its limitations when the batch size is large. In terms of memory management, it requires that the unused slots in the master instance are sufficient to store key-value tensors generated in the next iteration, leading to memory fragmentation issues. In terms of computation, since the local layers like FFN are all performed on the master instance, when the decoding phase becomes compute-bound, its performance is restricted by the computation resources in the master instance.

To address these issues, we further extend it to multi-master distributed decoding. As shown in Figure 8, a parallel group contains multiple master instances. Different master instances are responsible for different requests in a batch. They save corresponding key-value tensors into their local KV cache pools to break the memory fragmentation issue, and parallelize local layer computations across multiple master instances to improve computational efficiency. When master instances exchange query tensors, the communication between them can further overlap with the local attention computation of its mastered requests.

5 LoongServe Global Manager

With efficient elastic scaling mechanisms to change the DoP, the global manager is responsible for using them to schedule requests and key-value tensors across elastic instances efficiently. There may be newly arrived requests in the pending queue $P = \{r_1, r_2, ..., r_{n_P}\}$, a set of batches of requests in the decoding phase $B = \{B_1, B_2, ..., B_{n_B}\}$, where each decoding batch B_i is associated with an existing parallel group

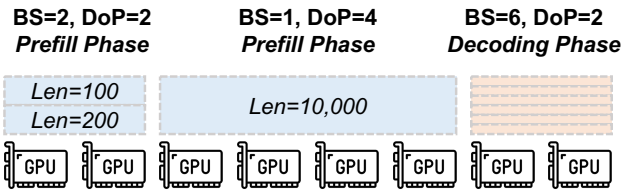


Figure 9. Illustration of the flexible scheduling space.

G_i. Elastic instances are either idle due to scale-down operation in the last iteration or executing a decoding batch. In each iteration, the global manager needs to dispatch some requests from P to execute in the prefill phase, allocate elastic instances to them, decide the batching strategy, and change the states of existing decoding batches. As shown in Figure 9, there are lots of requests with different sequence lengths in the different phases. All these aspects, including batching strategy, the DoP of each batch, and placement of key-value tensors, are configurable and may be different across iterations. It forms a scheduling space exponential to the number of requests and elastic instances.

The primary challenge for the global manager is to generate an efficient scheduling plan from this complex scheduling space in real time. The real-time requirement is due to the dynamic nature of the LLM inference workload. The efficiency of a scheduling plan is highly dependent on the current workload. Different workload requires different scheduling plans. For example, when serving a request with sub-linear scalability under the light load, as long as scaling-up is beneficial, it is better to serve this request with a high DoP to use more idle GPUs for GPU utilization improvement. Conversely, if under heavy load, due to the request's poor scalability, it is better to serve this request with a low DoP to leave more resources for other requests for GPU utilization improvement. It indicates that the global manager needs to make decisions in real-time based on the real-time state. However, the LLM inference is super fast. The duration of an iteration can be as low as tens of milliseconds. The scheduling time is restricted by a limited time budget.

To address these issues, we propose a *scalable four-step scheduling algorithm*. The key insight is that we decouple this scheduling problem into four sub-problems: dispatching, elastic instance allocation, batching, and elastic scaling plan generation. In each step, the LoongServe global manager generates an efficient plan for a respective aspect in polynomial time. Then, the LoongServe global manager combines these plans to form the final scheduling plan.

5.1 Dispatching

The dispatching step is to choose a subset of requests R_p from the pending queue P to execute the prefill phase in the current iteration. In this step, the global manager considers resource availability in both GPU computing and GPU memory. As in prior works [27, 55], the global manager scans through P in the first-come-first-serve (FCFS) order.

GPU memory constraints. As for GPU memory, GPU memory consumption of key-value tensors is the primary concern. The global manager does not add a request into R_p unless there are sufficient unused slots for this request. Moreover, because evicting a long sequence request incurs significant recomputation overhead, the global manager also tries to avoid future eviction and recomputation due to insufficient GPU memory. As in prior works [52], the global manager estimates it by considering the maximum future key-value cache consumption based on the current state and the maximum sequence length of the request given by users. The global manager does not add it into R_p if it may trigger eviction.

GPU computing constraints. As for GPU computing, the primary concern is the efficiency of dispatching R_p and their impact on existing requests in the decoding phase B. The global manager uses an analytical model and profiling results from SIB (§5.5) to measure them in terms of iteration time $T(R_p, E_p)$ on the occupied instances E_p. E_p is initialized as the subset of all idle instances.

First, there is a tipping point that R_p transitions from memory bound to compute bound. Before that point, adding more requests into R_p improves the efficiency of GPU computing. After that point, it only extends the execution time with negligible efficiency improvement. We estimate this point by profiling the upper bound of the iteration time that a prefill batch is memory bound. The global manager stops adding more requests into R_p when the iteration time of R_p exceeds this tipping point.

Second, adding more requests into R_p might interfere with some existing batches $B_p = \{B_{p,1}, B_{p,2}, ...\} \subseteq B$ if new requests utilize some unused key-value cache slots in instances occupied by B. To simplify this problem, we conservatively consider it in the worst case that R_p may preempt B_p. In this case, for each $B_{p,i}$, unused key-value slots of instances in its parallel group $G_{p,i}$ can be used to add an additional subset of new requests $R'_{p,i}$ into R_p. The global manager analyzes the performance gain of executing $R'_{p,i}$ and the cost of preempting $B_{p,i}$ to decide whether to add $R'_{p,i}$ into R_p and expand E_p. The performance cost, i.e., the impact of preemption on the output token latency, is formulated as follows:

$$\text{Cost} = \sum_{r \in B_{p,i}} \frac{T(R_p \cup R'_{p,i}, E_p \cup G_{p,i})}{r.\text{output\_len}} \tag{1}$$

As for the performance gain, it is formulated as the impact on the input token latency of $R'_{p,i}$:

$$\text{Gain} = \sum_{r \in R'_{p,i}} \frac{(\text{AvgLat}_\text{d} - \min(B_{p,i}.\text{exec\_time}))^+}{r.\text{input\_len}} \tag{2}$$

In this equation, AvgLat_d is the average execution time of finished requests in the decoding phase, and $\min(B_{p,i}.\text{exec\_time})$ is the executed time of requests in the decoding phase. The subtraction between them estimates how long $R'_{p,i}$ have to

wait for the decoding batch in the worst case. The global manager adds $R'_{p,i}$ into R_p and adds $G_{p,i}$ into E_p if the gain is greater than the cost. The complexity of this step is $O(n_B)$.

5.2 Elastic Instance Allocation

After generating exact R_p to execute, the global manager needs to decide actual elastic instance allocation for them in this step. In this step, the primary concern is to mitigate the interference between the prefill phase, i.e., R_p, and the decoding phase, i.e., B, while maximizing the efficiency of GPU computing.

The global manager first allocates idle instances to R_p. If the unused key-value cache slots of idle instances are not enough, R_p can preempt a few instances with the most unused key-value cache slots to obtain sufficient key-value cache slots. To avoid preemption, the global manager tries to migrate existing key-value tensors in preempted instances to other active instances if possible. As a result, R_p can obtain elastic instances E_p with sufficient key-value cache slots.

However, it is still possible to achieve better performance by allocating more elastic instances to the compute-intensive prefill phase. To this end, the global manager repeatedly considers whether to allocate elastic instances with the fewest used key-value cache slots, e_{min}, to R_p. The constraint is that the batch using e_{min} can migrate its key-value tensors to other instances in the decoding phase. In this case, the input token latency is reduced as follows:

$$\text{Gain} = \sum_{r \in R_p} \frac{T(R_p, E_p) - T(R_p, E_p \cup e_{min})}{r.\text{input\_len}} \tag{3}$$

But it also incurs the migration overhead as follows:

$$\text{Cost} = \sum_{r \in R_p} \frac{V(e_{min})}{\text{avg\_bandwidth} \cdot r.\text{input\_len}} \tag{4}$$

In this equation, $V(e_{min})$ is the volume of existing key-value tensors in e_{min}, and the avg_bandwidth is the average bandwidth between e_{min} and target instances. Target instances are always instances with the most unused key-value cache slots. The global manager repeatedly allocates e_{min} to R_p until the gain is less than the cost. The complexity of this step is $O(m)$, where m is the number of elastic instances.

5.3 Batching

After deciding R_p and E_p, this step optimizes the batching strategy of R_p on E_p to further minimize the latency of the prefill phase. In this step, the primary concern is to assign different DoPs to requests with different sequence lengths.

To address this issue, we formulate this batching problem as a dynamic programming (DP) problem. The optimization goal is to minimize the input latency of R_p on E_p. Our key insight is that requests with similar sequence lengths have similar characteristics and should be batched together. Therefore, the global manager first sorts requests based on their sequence lengths in descending order. The allocated elastic instances are also sorted based on their locations and the number of unused key-value cache slots in ascending order. Let $f[i][k]$ be the minimum input latency of the first i requests when using the first k elastic instances. The DP equation can be formulated as follows:

$$f[i][k] = \min_{\substack{0<j\le i, 0<l\le k, \\ D[j,i]\le V[l,k]}} (f[j][l] + T(R[j,i], E[l,k])) \tag{5}$$

The $D[j,i]$ in the equation is the number of tokens of requests from j to i, and the $V[l,k]$ is the number of unused key-value cache slots of elastic instances from l to k. These sums of values in an interval can be calculated by maintaining a prefix sum array in advance. The $T(R[j,i], E[l,k])$ is the sum of input latency of requests from j to i when using elastic instances from l to k. The minimum input latency sum of all requests, i.e., $\min_{0<j\le m} f[n][j]$, can be found in polynomial time. When updating $f[i][k]$, the global manager records the last split point of requests $split_{req}[i][k]$, i.e., j, and the last split point of elastic instances $split_{ins}[i][k]$, i.e., l. The global manager uses them to backtrack to generate the batching plan and the corresponding DoP for each batch.

If naively update $f[i][k]$ for all $0 < i \le n$ and $0 < j \le m$, the time complexity of this DP algorithm is $O(|R_p|^2 \cdot |E_p|^2)$. However, we notice that $split_{req}[i][k]$ and $split_{ins}[i][k]$ have following properties:

$$\begin{aligned} split_{req}[i][k-1] \le split_{req}[i][k] \le split_{req}[i][k+1], \\ split_{ins}[i-1][k] \le split_{ins}[i][k] \le split_{ins}[i+1][k]. \end{aligned} \tag{6}$$

Therefore, this problem can be optimized to $O((|R_p| + |E_p|)^2)$ by using *Quadrangle Inequality Properties* [54]. Although it can further optimize the time complexity, it is efficient enough in practice.

5.4 Elastic Scaling Plan Generation

As described in §4, the global manager also needs to generate elastic scaling plans for proactive scaling down and up.

For proactive scaling down, the key insight is that the decoding phase scales poorly. As shown in Figure 2, in most cases, the minimum best DoP of requests in the decoding phase are similar and are smaller than that in other cases. Therefore, we set the degree of model parallelism as the minimum best DoP at launch time. At run time, the global manager only needs to scale down the DoP to the minimum DoP that the key-value tensors of requests can fit in the corresponding elastic instances. It is optimal for most requests in the decoding phase. Even for requests in the decoding phase with larger best DoPs, it is still near-optimal, because leaving more elastic instances to the compute-intensive prefill phase with longer duration is more beneficial.

For scaling up, the global manager scales up when GPU computing or GPU memory is insufficient. Insufficient GPU

computing refers to the decoding phase becoming compute-bound. Because FFN layers first become the computation bottleneck and their complexity is related to the batch size, the global manager uses a batch size threshold to detect it. We profile this threshold in advance. The multi-master decoding is used as long as it can reduce memory fragmentation or execution time. The number of newly key-value tensors generated by each master is set to as uniform as possible.

5.5 Optimizations

Analytical Model based on SIB. The iteration time of the prefill phase under different scenarios guides the decision-making. Ideally, we can record them into SIB in advance and retrieve them at run time. However, there are innumerable combinations of requests with different input lengths executing at different DoPs. It is impossible to cover all the cases only by profiling. Therefore, we propose an analytical model to estimate them, which is formulated as follows:

$$\mathrm{T}_p(R) = \alpha_p + \beta_p \cdot \sum_{r \in R} r.\text{input\_len} + \gamma_p \cdot \sum_{r \in R} r.\text{input\_len}^2 \quad (7)$$

In this equation, α_p, β_p, and γ_p are the coefficients capturing the constant overhead, linear computation (e.g., FFN layers), and quadratic computation (e.g. attention layers), respectively. They are trained by the least square method based on a few profiling results. For different parallelism strategies, we train different coefficients.

Other optimizations. LoongServe also employs other optimizations to improve the efficiency by filling small bubbles in the scheduling plan. Please refer to the Appendix A for more details.

6 Implementation

LoongServe is implemented in approximately 15K lines of code based on C++, CUDA, Python, and OpenAI Triton [48], and reuses some components from LightLLM [46] and vLLM [27]. The front end of LoongServe is the same as LightLLM and is similar to OpenAI API [38]. Users can send requests to LoongServe without modifications if they have used LightLLM previously.

The LoongServe global manager is mainly implemented in Python, but some core logic, such as the batching algorithm, is implemented in C++ to accelerate looped functions. To manage multiple batches at the same time, the global manager assigns each batch to a Python coroutine.

The global manager uses Ray [36] to communicate with elastic instances. Because ESP introduces extra RPC parameters, RPC parameters is carefully designed to reduce extra serialization overhead. Elastic instances also cache active ESP metadata. Similar to vLLM [27], when tensor parallelism is enabled, the global manager mainly sends information to a single rank in an elastic instance, and this rank uses NCCL [3] to broadcast the information to other ranks to further reduce serialization overhead.

For each elastic instance, it manages its corresponding key-value cache pool by using PagedAttention [27] at the granularity of a single token. Although StripedAttention performs pretty well in long sequences, it causes redundant computation in short sequences due to its special causal attention mask. We tune the tile size to skip most redundant computations in short sequences. We also reduce the lifecycle of extra parameters introduced by StripedAttention in shared memory to improve the occupancy of streaming multiprocessors (SMs). For the decoding phase, we implemented a customized version of Flash-Decoding [13] with extra parameters to support ESP. All the above optimizations are compatible with MHA, MQA, and GQA, and have the same accuracy as the original implementations.

The communication between elastic instances is based on NCCL with dedicated CUDA streams. Tensor parallelism and sequence parallelism use different NCCL communicators. To support multiple dynamic parallel groups at the iteration level, we use NCCL group functions to merge multiple point-to-point operations to form collective operations in selected NCCL ranks.

LoongServe also provides tools to generate profiling results under different scenarios. These profiling results are stored in a SQLite database. Each time training analytical models just needs to select the corresponding profiling results from the database.

7 Evaluation

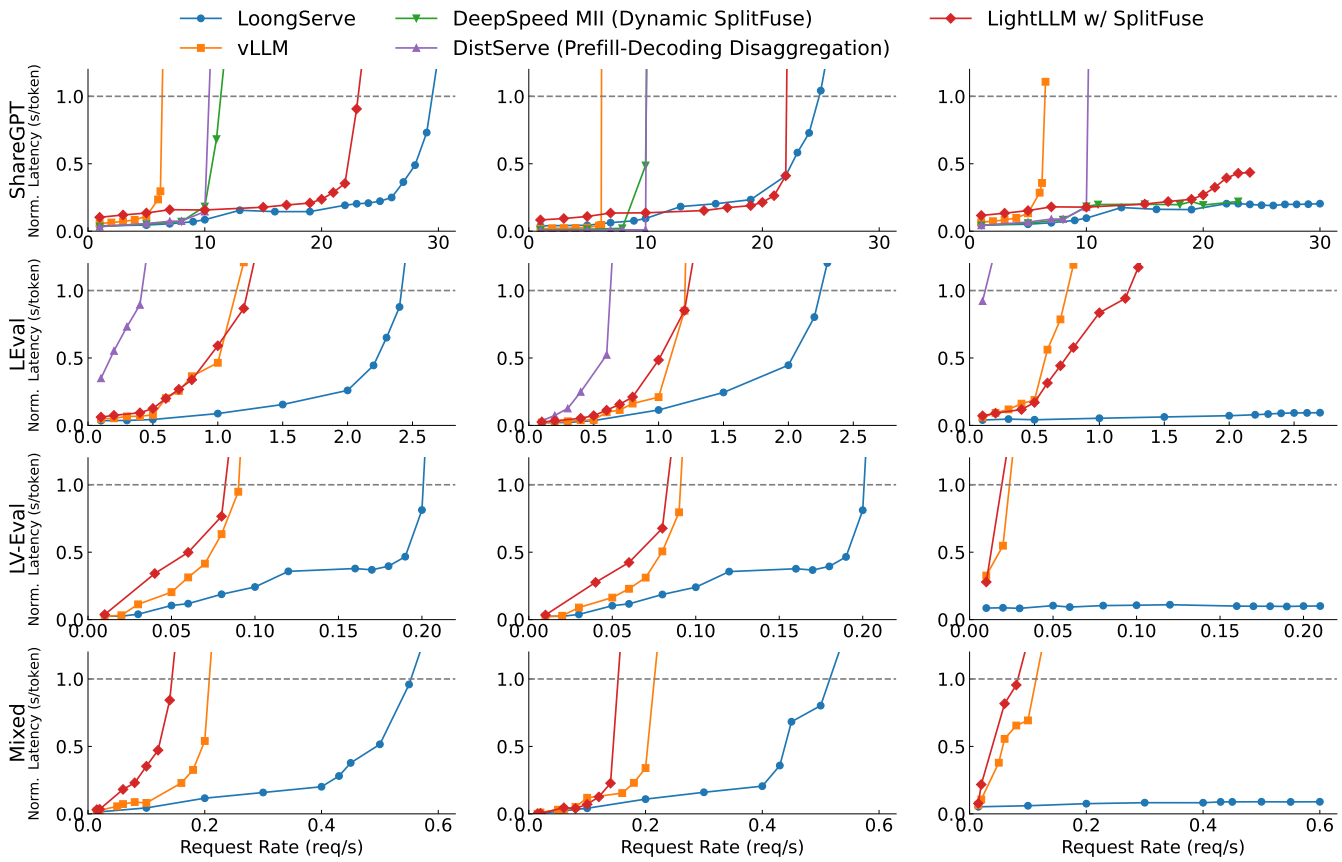
In this section, we evaluate the performance of LoongServe with state-of-the-art solutions on different real-world workloads and show the effectiveness of its components.

7.1 Experimental Setup

Model. We use the LWM-1M-Text model [33] as the long-context LLM model in our evaluation. It is the open-sourced pre-trained LLM with the largest context window size (1M tokens) when we started experiments. Besides, it uses the same model architecture as Llama-2-7B [50], which is widely used in practice.

Testbed. We evaluate LoongServe on servers each with eight NVIDIA A800 80GB GPUs, 128 CPUs, 2048 GB of host memory, and four 200 Gbps InfiniBand NICs. The NVLink bandwidth between two GPUs is 400 GB/s. We use PyTorch 2.0.0, CUDA 12.2, OpenAI Triton 2.1.0, and HuggingFace tokenizers 0.15.2 for our evaluation. Most experiments are conducted on a single server, and we also evaluate the multi-node performance of LoongServe on two servers.

Workloads. Similar to prior work [27, 31, 58], the arrival pattern of requests is generated by a Poisson process. The



Avg. per-token latency. Avg. input token latency. Avg. output token latency.

Figure 10. Average latency of different LLM serving systems with the LWM-1M-Text (Llama-2-7B) on real-world workloads.

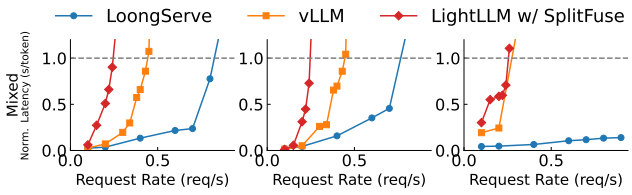
input lengths and output lengths of requests are sampled from the following real-world datasets:

- **ShareGPT** [4]: It is collected from real-world conversations with ChatGPT and is widely used in prior work [27, 52, 58]. Due to the limited context window of ChatGPT-3.5, the range of sequence length in this dataset is 4 - 2.3K tokens.
- **L-Eval** [9]: It contains human-labeled query-response pairs on diverse tasks, such as summarization and question answering. It is used to evaluate the long-context capability of Qwen 1.5 [47]. The range of sequence length in this dataset is 2.7K - 210.5K tokens.
- **LV-Eval** [56]: It is the dataset containing the longest requests when we started experiments. It mainly contains many long-context question-answer tasks. When converting words into tokens, the range of sequence length in this dataset is 15.1K - 497.3K tokens.
- **Mixed**: Because the above datasets only cover a small range of sequence lengths, we also mix them to evaluate the performance of LoongServe on a more diverse workload. The sampling probability of each dataset is the same.

Baselines. We compare LoongServe with the following state-of-the-art LLM serving systems:

- **vLLM** [27][1]: It is one of the most popular LLM serving systems. To fully leverage all GPUs and serve requests with long context, we set the tensor parallelism to 8.
- **DeepSpeed-MII** [5][2]: It proposes Dynamic SplitFuse [19] to decompose long input sequences into small chunks to prevent performance degradation of the decoding phase when serving both phases. However, when serving requests with long context larger than 32K tokens, we encounter the "illegal memory access" error, so we only evaluate it on ShareGPT. The tensor parallelism is set to 8.
- **LightLLM w/ SplitFuse** [46][3]: To evaluate the performance of SplitFuse for long sequences, we also evaluate SplitFuse provided by LightLLM. Similar to SARATHI [7], it needs to set the chunk size. We set the chunk size as the ideal "P:D ratio" described in SARATHI [7] by calculating it for each dataset before experiments, although it is unknown in practice. The tensor parallelism is set to 8.
- **DistServe** [58]: It proposes prefill-decode disaggregation to reduce the impact of the prefill phase on the decoding phase. We contact the authors to get the source code and set parallelism strategies for it. Because its parallelism

[1] vLLM 0.3.0, commit hash: 1af090b57d0e23d268e79941f8084bf0a8ad8621
[2] commit hash: 773b735d6294a98dd842d82ef024d0d9b050f66aa
[3] commit hash: e2b5168a50f24d960ead314b0649428e35381f80



Per-token latency. Input latency. Output latency.

Figure 11. Multi-node Performance.

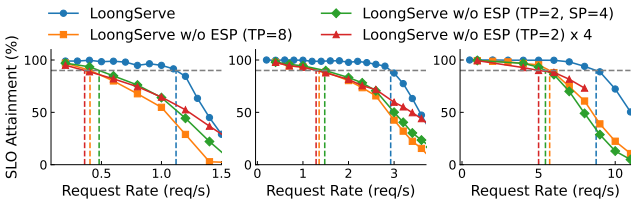
strategies are restricted by the head of the model and requests with long context need large GPU memory, we use four GPUs for the prefill phase and four GPUs for the decoding phase. The DoP for each phase is set to 4.

For LoongServe, we set tensor parallelism to 2 and ESP to 4. For all the systems, the number of key-value cache slots is set as much as possible to improve the throughput.

Metrics. We focus on the serving throughput. For each request rate, we measure the *normalized per-token latency*, i.e., the mean of requests' end-to-end latency divided by their sequence lengths, *normalized input latency*, i.e., the mean of prefill phase time divided by the input lengths, and the *normalized output latency*, i.e., the mean of decoding phase time divided by the output lengths, for each system. To compare different systems, we set a latency service level objective (SLO) and compare the maximum throughput under this SLO. Similar to prior work [52], we set SLO to 25× of the latency under the light load.

7.2 End-to-End Performance

We compare LoongServe with four baselines on four real-world workloads in three key metrics. Figure 10 shows the results. Because LoongServe often uses different elastic instances to execute different phases, the decoding phase is well protected from the impact of the prefill phase. As a result, the output latency of LoongServe achieves low latency and is significantly better than other baselines (nearly infinite output throughput improvement). For the prefill phase, LoongServe accelerates them by setting the appropriate DoP to avoid blocking pending prefill phases. In contrast, vLLM consistently treats different requests in the same way. On the ShareGPT, vLLM wastes resources on short requests with poor scalability. On other datasets, it causes severe interference between the decoding phase and the prefill phase. LoongServe improves the total throughput and input throughput by up to 4.64× and 4.00×. DeepSpeed MII and LightLLM w/ SplitFuse try to split the input sequence into chunks to protect the decoding phase. However, decomposing the input sequence makes the prefill phase inefficient. Furthermore, for long sequences in L-Eval and LV-Eval, the prefill phase still causes severe interference with the decoding phase, because the "P:D" ratio is high. Compared to them, LoongServe improves the total throughput and input throughput by up to 3.85× and 3.37×. DistServe disaggregates the prefill phase and the decoding phase to reduce the interference between them. However, it means that each phase can only use half of the GPUs (four GPUs). On the ShareGPT, four GPUs are not enough to serve lots of requests in the decoding phase. On the L-Eval, four GPUs are not enough to serve lots of requests in the prefill phase. On the LV-Eval and Mixed, four GPUs for each phase even do not have enough memory to serve requests with long context and thus trigger the Out-of-Memory (OOM) error, so it is not shown in these two rows of figures. Compared to DistServe, LoongServe improves the total throughput and input throughput by up to 5.81× and 3.58×.



(a) Zipf=1.0. (b) Zipf=1.2. (c) Zipf=1.4.

Figure 12. P90 goodput under different sequence length distributions.

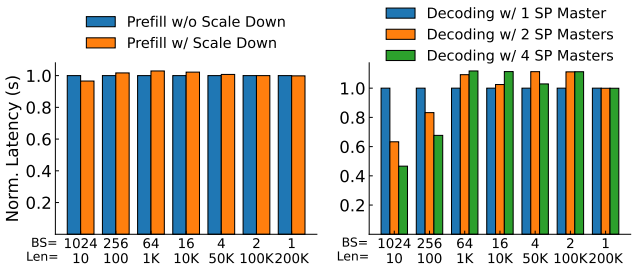
7.3 Multi-Node Performance

We also evaluate the multi-node performance of LoongServe on a 16 GPU cluster. In this experiment, we deploy baselines in each server and use the same parallelism strategies as the single-node evaluation. LoongServe also uses the same model parallelism strategy and extends the ESP to 8. We also use the same Mixed dataset. Figure 11 shows the results. In the multi-node setting, LoongServe scales well and achieves the best performance in all metrics by setting the appropriate DoP for each request. Therefore, LoongServe avoids unnecessary communication overhead for short requests and enlarges the parallelism for long requests. As a result, LoongServe improves the total throughput and input throughput by up to 1.86× and 1.72× compared to vLLM, 3.37× and 3.11× compared to LightLLM w/ SplitFuse, while significantly reducing the output latency under all request rates.

7.4 Ablation Study

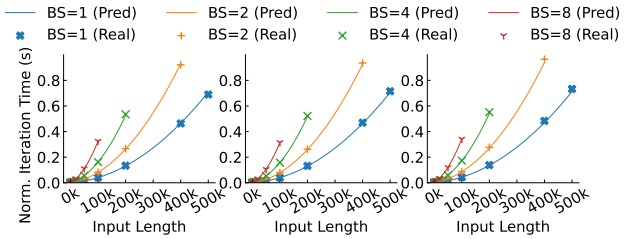
7.4.1 Effectiveness of Elastic Sequence Parallelism.

To show the effectiveness of elastic sequence parallelism, we conduct an ablation study on the P90 goodput (throughput of requests under the SLO) [58] of different parallelism strategies under different sequence length distributions. Baselines include traditional tensor parallelism, i.e., LoongServe w/o ESP (TP=8), static hybrid parallelism, i.e., LoongServe w/o ESP (TP=2, SP=4), and parallelism with replication, i.e., LoongServe w/o ESP (TP=2) × 4. To simulate different scenarios, we sample the sequence lengths from the Mixed dataset with different Zipf parameters. Because the baseline with replication is not able to serve requests with long context, we limit the maximum length of requests to 200K tokens.



(a) Scale down overhead. (b) Scale up overhead.

Figure 13. Overhead of elastic scaling mechanisms.



(a) SP2TP4. (b) SP4TP2. (c) SP8TP1.

Figure 14. Accuracy of LoongServe analytical model.

Figure 12 shows the results. As shown in the figure, only introducing sequence parallelism is not enough to achieve good performance. Neither static hybrid parallelism nor parallelism with replication handles the dynamic inference workload. In contrast, LoongServe dynamically adjusts strategies for the dynamic workload and consistently achieves P90 goodput improvement by 2.33×, 1.98×, and 1.53× under different sequence distributions. P90 goodput also indicates that ESP is beneficial for most requests.

7.4.2 Scaling Overhead. We evaluate the overhead of scaling down and scaling up under different batch sizes and input lengths by timing the time to forward a batch with and without scaling, respectively. Figure 13 shows the results. For scaling down, we only introduce a negligible overhead (less than 2%) for all batch sizes and prompt lengths. For scaling up, on large batch sizes, since the compute-memory ratio is high for matrix multiplication operations (including projection for query, key, and value, as well as matrix multiplication for the FFN), distributing the computation to more instances can significantly reduce the computation time, leading to a 2× improvement in per-iteration latency. On small batch sizes, the overhead of scaling up is higher because of the overhead introduced by communication and synchronization. However, the overhead is still acceptable (less than 10%) and LoongServe dynamically uses the best one.

7.4.3 Accuracy of Analytical Model. At last, we evaluate the accuracy of analytical models in different parallelism strategies. As shown in Figure 14, the analytical model of LoongServe achieves high accuracy (less than 10% deviation) for different batches of requests with different sequence lengths in different parallelism strategies. It is reliable to guide the LoongServe global manager.

8 Related Work

Existing LLM Serving Systems. Existing LLM serving systems adopt efficient GPU operators for the long sequence requests, such as Flash Attention [14] and Flash-Decoding [13]. They are orthogonal to our work and are integrated into LoongServe. To mitigate the impact of the long context, SARATHI [7] and DeepSpeed-FastGen [19] split the long context into chunks and process them chunks by chunks, but they still incur interference between two phases [58]. SplitWise [40], DistServe [58], and TetriInfer [20] disaggregate two phases into different groups of GPUs to avoid interference, but their static parallelism and partition strategies are not flexible to handle the dynamic workload. InfiniteLLM [32] tries to alleviate the GPU fragmentation across instances. However, it still needs periodic key-value tensor migration to maintain the locality and does not consider elastic resource demand between different requests or different phases. LoongServe proposes a novel elastic scaling mechanism without additional overhead and ESP to handle dynamic workload based on requests' resource demand without locality constraints.

Sequence Parallelism. Many works [11, 22, 26, 28, 30, 34] are proposed to accelerate long-context LLM training. One class of work still uses TP for the attention layers and suffers from the same problems as TP. Another class of work like Striped Attention [11] parallelizes the attention mechanism in the sequence dimension. All these works focus on LLM training and their DoP are fixed. Our work targets the serving scenario, supporting the decoding phase, dynamic DoP, and efficient key-value cache management.

Long-context LLM with Accuracy Loss. Another type of Long-context LLM trades off accuracy for efficiency. Some works [12, 39, 45] change the attention mechanism to reduce computation. Recent works [6, 53, 57] also try to prune the key-value cache to reduce the memory footprint. All these works incur accuracy loss and are not widely used. LoongServe does not affect the accuracy of the original LLM and provides a more efficient serving mechanism. Besides, LoongServe is compatible with MQA [43], GQA [8], and MoE [24] to reduce the memory footprint and computational complexity.

Elastic Training. Many works focus on elastic training for deep neural networks (DNNs) [17, 18, 21, 23, 29, 35, 41, 42]. Compared to them, LoongServe proposes a new parallelism strategy, ESP, rather than using data parallelism as in the training phase. LoongServe also considers more unique factors in the LLM serving, such as key-value tensor management and request batching, under more strict latency constraints.

9 Conclusion

To serve long-context LLM under the dynamic workload, we propose elastic sequence parallelism and build LoongServe, which provides a set of elastic scaling mechanisms without additional overhead and a scalable scheduling algorithm for ESP. Evaluation across diverse datasets shows that compared to SOTA solutions, LoongServe significantly improves the total throughput, input token throughput, and output token latency simultaneously.

References

[1] 2022. Introducing ChatGPT. https://openai.com/blog/chatgpt. (2022).

[2] 2023. Bard, an experiment by Google. https://bard.google.com/. (2023).

[3] 2023. Optimized primitives for collective multi-GPU communicatio Resources. https://github.com/NVIDIA/nccl. (2023).

[4] 2023. ShareGPT Teams. https://sharegpt.com/. (2023).

[5] 2024. DeepSpeed Model Implementations for Inference (MII). https://github.com/microsoft/DeepSpeed-MII. (2024).

[6] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant J Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: KV Cache Reduction through Key Tokens Selection for Efficient Generative Inference. *Conference on Machine Learning and Systems* (2024).

[7] 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* (2023).

[8] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *The Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[9] Chenxin An, Shansan Gong, Ming Zhong, Xingjian Zhao, Mukai Li, Jun Zhang, Lingpeng Kong, and Xipeng Qiu. 2023. L-Eval: Instituting Standardized Evaluation for Long Context Language Models. *arXiv* (2023).

[10] Anthropic. 2024. Introducing the next generation of Claude. https://www.anthropic.com/news/claude-3-family. (2024).

[11] William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. 2023. Striped Attention: Faster Ring Attention for Causal Transformers. *arXiv* (2023).

[12] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating Long Sequences with Sparse Transformers. *arXiv* (2019).

[13] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. *arXiv* (2023).

[14] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Neural Information Processing Systems*.

[15] Google. 2024. Our next-generation model: Gemini 1.5. https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/. (2024).

[16] Significant Gravitas. 2023. AutoGPT. (2023). https://github.com/Significant-Gravitas/AutoGPT

[17] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. ElasticFlow: An Elastic Serverless Training Platform for Distributed Deep Learning. In *ACM ASPLOS*.

[18] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning.. In *USENIX NSDI*.

[19] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. 2024. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. *arXiv* (2024).

[20] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. *arXiv* (2024).

[21] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *USENIX NSDI*.

[22] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Leon Song, Samyam Rajbhandari, and Yuxiong He. 2023. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv* (2023).

[23] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *ACM SOSP*.

[24] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. *arXiv* (2024).

[25] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing Activation Recomputation in Large Transformer Models. *arXiv* (2022).

[26] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Conference on Machine Learning and Systems* (2023).

[27] 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 *ACM SOSP*.

[28] Dacheng Li, Rulin Shao, Anze Xie, Eric P Xing, Joseph E Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. 2023. Lightseq: Sequence level parallelism for distributed training of long context transformers. *arXiv* (2023).

[29] Mingzhen Li, Wencong Xiao, Biao Sun, Hanyu Zhao, Hailong Yang, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, Wei Lin, and Depei Qian. 2023. EasyScale: Accuracy-consistent Elastic Training for Deep Learning. In *ACM SC*.

[30] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. 2023. Sequence Parallelism: Long Sequence Training from System Perspective. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.

[31] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *USENIX OSDI*.

[32] Bin Lin, Tao Peng, Chen Zhang, Minmin Sun, Lanbo Li, Hanyu Zhao, Wencong Xiao, Qi Xu, Xiafei Qiu, Shen Li, Zhigang Ji, Yong Li, and Wei Lin. 2024. Infinite-LLM: Efficient LLM Service for Long Context with DistAttention and Distributed KVCache. *arXiv* (2024).

[33] Hao Liu, Wilson Yan, Matei Zaharia, and Pieter Abbeel. 2024. World Model on Million-Length Video and Language with RingAttention. *arXiv* (2024).

[34] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context. *arXiv* (2023).

[35] Yujing Ma, Florin Rusu, Kesheng Wu, and Alexander Sim. 2021. Adaptive Elastic Training for Sparse Deep Learning on Heterogeneous Multi-GPU Servers. *arXiv* (2021).

[36] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *USENIX OSDI*.

[37] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *International Conference on Learning Representations (ICLR)*.

[38] OpenAI. 2023. GPT-4 Technical Report. (2023).

[39] Matteo Pagliardini, Daniele Paliotta, Martin Jaggi, and François Fleuret. 2023. Faster Causal Attention Over Large Sequences Through Sparse Flash Attention. *arXiv* (2023).
[40] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. 2023. Splitwise: Efficient generative LLM inference using phase splitting. *arXiv* (2023).
[41] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*.
[42] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *USENIX OSDI*.
[43] Noam Shazeer. 2019. Fast Transformer Decoding: One Write-Head is All You Need. *arXiv* (2019).
[44] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. (2020).
[45] Weigao Sun, Zhen Qin, Dong Li, Xuyang Shen, Yu Qiao, and Yiran Zhong. 2024. Linear Attention Sequence Parallelism. *arXiv* (2024).
[46] LightLLM Team. 2023. LightLLM: A Light and Fast Inference Service for LLM. https://github.com/ModelTC/lightllm. (2023).
[47] Qwen Team. 2024. Introducing Qwen1.5. https://qwenlm.github.io/blog/qwen1.5/. (2024).
[48] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*.
[49] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. (2023).
[50] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Neural Information Processing Systems* (2017).
[52] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast Distributed Inference Serving for Large Language Models. *arXiv* (2023).
[53] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient Streaming Language Models with Attention Sinks. In *International Conference on Learning Representations (ICLR)*.
[54] F. Frances Yao. 1980. Efficient dynamic programming using quadrangle inequalities. In *ACM Symposium on Theory of Computing*.
[55] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-BasedGenerative Models. In *USENIX OSDI*.
[56] Tao Yuan, Xuefei Ning, Dong Zhou, Zhijie Yang, Shiyao Li, Minghui Zhuang, Zheyue Tan, Zhuyu Yao, Dahua Lin, Boxun Li, Guohao Dai, Shengen Yan, and Yu Wang. 2024. LV-Eval: A Balanced Long-Context Benchmark with 5 Length Levels Up to 256K. *arXiv* (2024).
[57] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2024. H2o: Heavy-hitter oracle for efficient generative inference of large language models. In *Neural Information Processing Systems*.
[58] 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. *arXiv* (2024).

A Other Optimizations

Relaxed FCFS Dispatching. Although FCFS is simple and efficient enough for the short-context LLMs, it is not always optimal for the long-context LLMs. For requests with short input lengths, even if they can execute, previous requests acquiring more resources may block them, leading to long queuing time. For requests with long input lengths, even if they can execute, there may be few elastic instances available for them, leading to long execution time.

To address these issues, our key insight is to relax FCFS to improve the overall efficiency of LLM serving. First, LoongServe allows *out-of-order execution* (OOE). When the oldest requests cannot execute due to resource constraints, the global manager tries to execute younger requests if possible. To avoid starvation, LoongServe provides a threshold *max_num_ooe*. It indicates that after *max_num_ooe* iterations of dispatching, the global manager must dispatch the oldest requests first.

Second, LoongServe allows *delay execution*. Similar to §5.1, the global manager roughly estimates the waiting time to use all instances, i.e., $\text{AvgLat}_\text{d} - \min(B.\text{exec\_time})$, and the reduction in iteration time when using all elastic instances rather than available ones. If the performance gain is greater than the cost, the global manager delays it into future iterations.

Greedy Execution. In each iteration, different batches may have different execution times. For example, the decoding phase is typically much faster than the prefill phase. If these independent batches need to wait for the slowest one, there will be much idle time. In this case, the global manager allows faster batches to continue executing more iterations called *greedy iterations*, as long as the slowest batch does not finish. If a faster batch finishes, the global manager will schedule pending requests to fill the idle time as long as they finish before that of the slowest normal iteration. As a result, it does not cause starvation.