Achieving Locality and Fairness in LLM Serving

Paper #130 (12 pages)

Abstract

Large language model (LLM) inference workload dominates a wide variety of modern AI applications, ranging from multi-turn conversation to document analysis. Balancing fairness and efficiency is critical for managing diverse client workloads with varying prefix patterns. Unfortunately, existing fair scheduling algorithms for LLM serving, such as Virtual Token Counter (VTC), fail to take prefix caching locality into consideration and thus suffer from poor performance. On the other hand, prefix-aware scheduling algorithms in existing LLM serving frameworks tend to maximize the prefix cache hit rate without considering fair sharing among clients.

This paper introduces the *first* prefix-aware fair scheduling algorithm, Deficit Longest Prefix Match (DLPM), which can maintain a high degree of prefix locality with a fairness guarantee. We also introduce a novel algorithm, DoubleQ, extending DLPM for distributed setup that can find a balance point among fairness, locality, and load-balancing. Our extensive evaluation demonstrates the superior performance of DLPM and DoubleQ in ensuring fairness while maintaining high throughput (up to 2.87× higher than VTC) and low per-client (up to 7.18× lower than state-of-the-art distributed LLM serving system) latency.

1 Introduction

Online inference workloads for large language models (LLMs) are rapidly becoming widespread, driven by their general-purpose capabilities and versatility across a wide range of tasks such as search engines [2], coding assistant [12], autonomous agents [30, 34, 48], and tool calling [35, 39]. The release of OpenAI's o1 model has further highlighted the *test-time scaling* phenomenon [4, 8, 32, 43], where the allocation of increased computational resources during inference via techniques such as Monte Carlo Tree Search (MCTS) [37, 54], Best-of-N sampling [43] and Self-refine [25], can substantially improve the quality of LLM-generated answers across various tasks. The increasingly complex test-time compute requirements underscore the growing prominence of inference workloads in the LLM landscape.

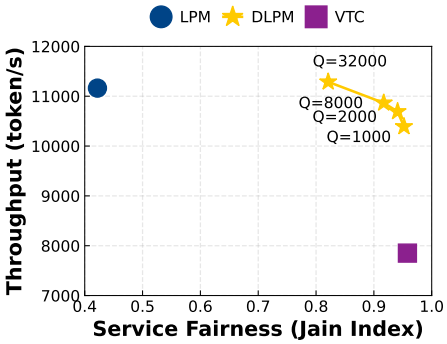


Figure 1: DLPM achieves a *new Pareto frontier* considering locality and fairness in LLM serving. Q is a hyper-parameter in DLPM, indicating how much we relax the fairness bound of DLPM. Results are obtained in a single A10 GPU.

Despite the advance in LLM generation quality, efficiently scaling online LLM inference services remains challenging, posing substantial barriers to their broad adoption. On the one hand, service providers need to provide isolation between concurrent tasks to ensure stable and predictable performance for all clients [31]: a client's experience should not be negatively impacted by a dominant or malicious client. On the other hand, service providers want to maximize system efficiency to improve throughput and reduce cost.

Unfortunately, existing scheduling algorithms [21, 41, 44, 57] for LLM serving fall short of achieving these dual goals effectively, as shown in Fig. 1. Although fair scheduling algorithms such as Virtual Token Counter (VTC) [41], are work-conserving – ensuring the system is fully utilized as long as there are requests in the system – they are not locality-aware. Locality awareness is essential for enhancing memory and computational efficiency, particularly through mechanisms such as prefix sharing [57]. Reusing the prefix's key-value (KV) tensors across multiple requests allows multiple requests sharing the same prefix to retain only one copy of the prefix's KV tensors in GPU memory. Moreover, it reduces redundant

computation of the prefix's KV tensors. Conversely, algorithms such as Longest Prefix Match (LPM) [57] enhance the system efficiency by prioritizing prefix locality: reordering the requests to maximize the prefix cache hit rate, yet they fail to guarantee effective isolation among clients – a malicious client can monopolize shared resources by sending a large volume of requests with long identical prefix, significantly degrading the performance experienced by other clients.

Achieving both fairness and prefix locality in LLM inference scheduling is challenging, as these two goals inherently conflict with each other. Prefix sharing, for instance, may require reordering requests to group those with identical prefixes together. In contrast, fair scheduling algorithms prioritize serving requests in a specific order to ensure isolation and prevent any single client from dominating resources. This necessary ordering can interfere with the efficiency gains from prefix sharing, as it restricts the flexibility to reorder requests for optimal resource utilization. This challenge is exacerbated in a distributed setting, where the algorithm must decide not only the order in which the requests are dispatched, but also to which GPU they are dispatched to achieve load balancing and prefix locality. This dual consideration of dispatch order and location significantly complicates achieving efficient and fair resource allocation across multiple GPUs.

In this paper, we introduce the first prefix-aware fair scheduling algorithm **Deficit Longest Prefix Match (DLPM)** for LLM serving which relaxes the dispatch order required by VTC to better preserve prefix locality while still bounding the allocation fairness. As illustrated in Fig. 1, DLPM can achieve throughput comparable to that of LPM while maintaining a degree of fairness close to that provided by VTC. We further propose a novel *distributed* scheduling algorithm **Double Quantum (DoubleQ)** that builds on top of DLPM to preserve high per-GPU prefix locality with a global fairness guarantee in a distributed setting.

In summary, this paper makes the following contributions:

- We introduce the *first* prefix-aware fair scheduling algorithm DLPM and its distributed version DoubleQ for LLM serving, which can achieve up to 2.87× higher throughput than VTC and up to 7.18× lower latency than the state-of-the-art locality-aware scheduling algorithm [44, 57].
- We provide rigorous theoretical bounds on DLPM and DoubleQ's fairness property, including service bound and latency bound between various types of clients.
- We conduct extensive evaluations on our proposed algorithms and demonstrate their superiority in achieving high system throughput while preserving fairness guarantees.

2 Background and Motivation

In this section, we first briefly introduce the basics of LLM inference, prefix caching, and fairness in LLM serving (§2.1). We then discuss key issues with existing LLM serving scheduling algorithms and the challenges they pose (§2.2).

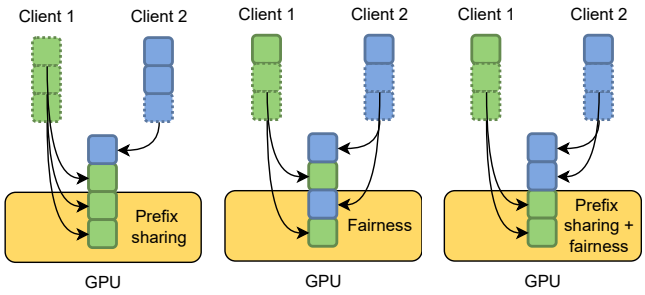
2.1 Transformer-Based LLM Inference

LLM Inference Modern transformer-based LLM inference consists of *prefill* and *decode* phases. The prefill phase takes input prompt, computes internal embedding vectors through the attention mechanism [47], and generates the first output token. These embedding vectors are normally stored inside the GPU memory as the *KV cache* to avoid recomputation. In the decode phase, new tokens are generated auto-regressively until an End-Of-Sequence (EOS) token is encountered or the pre-defined maximum token length is reached. During each iteration of token generation, the key-value (KV) cache of all previous tokens will be needed and the key-value tensors of the newly generated token will be appended to the KV cache. Such auto-regressive generation can lead to sub-optimal device utilization and decreased serving throughput [36]. To enhance GPU utilization, [52] proposed *continuous batching*. However, limited memory capacity emerged as a critical bottleneck, restricting batch sizes and thus reducing GPU efficiency. To address this issue, [21] developed PagedAttention, which mitigates memory fragmentation inherent in continuous batching and significantly enhances memory efficiency.

Prefix Caching and Locality To further improve the memory and computation efficiency, SGLang [57] introduced RadixAttention to facilitate the reuse of the KV cache of the shared prefix across multiple different LLM calls. By exploiting the prefix locality, memory usage for the KV cache is reduced, allowing for larger batch sizes and improved GPU utilization. Additionally, it eliminates redundant computations for the shared KV cache,

This technique is increasingly crucial for emerging multi-call LLM workloads such as Tree-of-Thought [51], Skeleton-of-thought [29], MCTS [54], and Self-refine [25], where there are substantial opportunities for prefix sharing. For instance, in a Tree-of-Thought program, all branches originating from the same node share the entire prefix up to the root. As the tree expands, the number of requests sharing the same prefix grows, and as the tree deepens, the length of the shared prefix increases.

LLM Serving Fairness Achieving efficient online LLM inference with Service Level Objective (SLO) guarantees necessitates isolation among different clients [41]. This need arises because clients share the same GPU accelerators and compete for these GPU resources. Without isolation, there is a risk that one client might monopolize resources, leading to the starvation of others. Moreover, to optimize resource utilization, it is crucial to reallocate unused resources from one client to another rather than merely imposing a rate limit [31] on each client for isolation purpose. Rate limit simply disallows clients to send requests beyond a certain rate which harms the resource utilization as shown in [41]. Formally, our goal is to achieve the classic max-min fairness [49], where the fair scheduling ensures each client receives at least $1/n$ of the resources, with n representing the total number of clients. If



(a) LPM. **(b)** VTC. **(c)** DLPM (ours).

Figure 2: Requests from the same client share prefixes with each other. In LPM locality-aware scheduling, the system schedules the GPUs to process all requests from Client 1 to maximize prefix sharing while starving Client 2. In VTC fair scheduling, the system processes requests in turn to maximize fairness, while ignoring the prefix sharing opportunity. Our DLPM scheduling achieves the best of two worlds through a novel quantum mechanism (§4) to guarantee locality while not sacrificing fairness.

some clients do not fully utilize their allocated share, these resources can be redistributed to others.

VTC [41] proposes the first fair scheduling algorithm called Virtual Token Counter targeting the continuous batching mechanism in online LLM serving. It tracks the tokens serviced for each client as the virtual counter and prioritizes clients with the lowest counters in each batching iteration. By tracking token-level resource usage, VTC achieves fair scheduling even when the output length of the request is unknown in advance.

2.2 The Trade-offs

Locality vs. Fairness Achieving both strong fairness and high locality for efficient online LLM serving is inherently challenging, since these two are usually at odds with each other, as illustrated in Fig. 2. On the one hand, locality-aware scheduling (Fig. 2a) reorders requests to group those with similar prefixes – often originating from the same client – to the same GPU to optimize for prefix locality. On the other hand, the VTC fair scheduler (Fig. 2b) adheres to a strict order based on per-client resource usage counters to dispatch requests, ensuring no client continuously dominates the GPU usage; such an order compromises locality as it intersperses the requests of the same client with requests from other clients. Fig. 1 also demonstrates the vastly different prioritizations of these two techniques, highlighting the trade-off between fairness and prefix locality.

Locality vs. Load-Balancing The challenge intensifies in distributed settings, where model replicas are served on multiple workers, each managed by its own local scheduler, with a global scheduler coordinating all these local workers. In this scenario, the scheduling algorithm on the global scheduler must balance a trade-off between locality and load balancing.

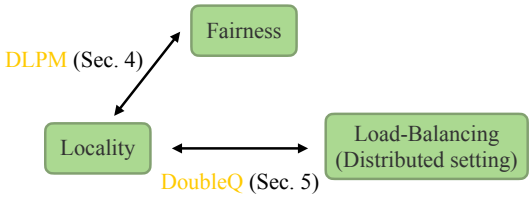


Figure 3: This paper addresses the conflict between fairness and locality through the DLPM mechanism (§4). It further addresses the conflict between locality and load balancing in distributed settings with the DoubleQ mechanism (§5).

For instance, simply distributing requests equally across the cluster is suboptimal due to the high prefix recompute overhead. Similarly, always dispatching requests with the same prefix to a single GPU can lead to workload imbalance.

Design Goals The main goal of this paper is to provide a principled way of navigating the trade-off between strong fairness and high locality in online LLM serving, as well as between locality and load-balancing in distributed settings. Our methodology ensures that the algorithms for single and distributed settings can be combined to maintain global fairness effectively. In the remainder of the paper, we begin by discussing preliminary concepts related to fairness in LLM serving (§3), then we introduce our fair scheduling design for a single worker (§4), and finally, we expand this approach to distributed fair scheduling (§5).

3 Preliminaries

In this section, we first formally define the properties a fair scheduling algorithm needs to meet for LLM serving, following those described in VTC [41]. We then discuss the cost function we adopt for service measurement.

Definition 3.1 (Backlog)**.** A client u is backlogged if dispatching additional requests cannot further increase throughput and can only incur additional queueing delay. In distributed settings, depending on the load-balancing policy, a backlogged client may have requests in queues of certain workers or all workers.

Fairness Properties Similar to VTC, our goal is to achieve *approximate max-min fairness* [49] on the service received by each client; **different from** VTC, we also want to preserve prefix cache locality. More formally, an LLM serving system that can achieve *approximate* max-min fairness should satisfy the following three properties [41]:

1. During any time interval $[t_1, t_2)$, if two clients f and g are continuously backlogged, they should receive a similar level of service, i.e. $|W_f(t_1,t_2) - W_g(t_1,t_2)| \leq \delta$, where δ is a constant value independent of $t_2 - t_1$.

Table 1: The upper half includes notations for service measurement. The lower half includes notations for the DLPM and DoubleQ algorithm and their analysis. *The extend tokens are the input tokens excluding prefix tokens.

Notation	Explanation
$W_f(t_1,t_2)$	service received by f during interval $[t_1,t_2)$
n_e	number of processed extend tokens*
n_q	number of processed output tokens
w_e	weight of extend tokens in the cost function
w_q	weight of output tokens in the cost function
Q^u	the quantum assigned to each client in DLPM
q_i	the deficit counter of client i in DLPM
Q^w	the quantum assigned to each worker in DoubleQ
$q_{i,w}$	the deficit counter of worker w for client i in DoubleQ
L_{input}	maximum number of input tokens in a request
L_{output}	maximum number of output tokens in a request
M	maximum number of tokens that can be fitted in a running batch
U	maximum number of counter that a single request can consume $w_e \cdot L_{input} + w_q \cdot M$
D	data parallelism degrees

2. A client f that is continuously backlogged during a time interval should not receive less service than another client g that is not continuously backlogged during the same time interval, i.e. $W_g(t_1,t_2) - W_f(t_1,t_2) \leq \delta$, where δ is a constant value.
3. The scheduling policy should be work-conserving: no worker should be idle if there are requests in the queue.

The first property states that a client sending at a high request rate is guaranteed to not receive more than their fair share of service and will not impact other normal-behaved clients. The second property prevents clients from accumulating unused service by first sending at a low request rate and later monopolizing the system. The third property guarantees that no resources are wasted in order to enforce fairness.

Measurement of Service Another important aspect in designing a fair scheduling algorithm for LLM serving is how the service should be measured. In VTC, the cost function is defined as a weighted sum of the number of input tokens and the number of output tokens. To incorporate the impact of prefix sharing (i.e., reduced memory and computations), we introduce a slightly different measure. Intuitively, with prefix sharing, the prefix tokens' cost should only be counted once when it is first calculated and stored in the GPU memory. Our prefix-aware version of the cost function is then defined as $W(t_1,t_2) = w_e \cdot n_e(t_1,t_2) + w_q \cdot n_q(t_1,t_2)$. The notations are explained in Tab. 1. Here w_e and w_q are set to be 1 and 2, inspired by OpenAI's pricing for GPT4[1]

[1] https://openai.com/api/pricing/

4 Deficit Longest Prefix Match (DLPM)

In this section, we present our algorithm DLPM for the single worker in §4.1 and the proved fairness guarantees in §4.2.

4.1 Algorithm Design

In the Longest Prefix Match (LPM) algorithm [57], at each continuous batching step, the scheduler first sorts current requests in the waiting queue based on their matched prefix length and then adds them to the new batch until the memory pool is full. LPM efficiently utilizes memory by grouping requests that can share a common prefix, thus maximizing the decoding batch size, which in turn leads to better operational intensity and throughput for the decoding phase.

To maintain the cache hit rate while introducing a fairness guarantee, it is essential not to disrupt the LPM order of the requests excessively. To achieve this, we incorporate a quantum mechanism inspired by the deficit round robin (DRR) approach [42]. This mechanism compels the scheduler to occasionally prioritize requests from less-served clients over those with the longest matching prefixes. Intuitively, this mechanism is effective because it preserves the local ordering inherent to the LPM. As a result, the system continues to benefit significantly from the memory savings brought by the shared prefixes, while the additional cost of prefix recomputation is incurred only when switching to serve less-served clients. This balanced approach allows DLPM to uphold the core efficiencies of the original LPM algorithm while enhancing fairness across client requests, ensuring that no clients monopolize the batching process to the detriment of others.

The core algorithm of DLPM is presented in Algorithm 1. Initially, the algorithm initializes all clients' deficit counter q_i to zero, with Q^u representing the service quantum replenished to each client in a cycle. At each continuous batching step, DLPM performs the following steps: 1) It sorts the requests in the waiting queue by their matched prefix length and then tries to add them to the currently running batch (B) until the memory pool is full. 2) The request will be added to B if the corresponding client has a positive deficit counter ($q_i > 0$). Otherwise, the request will be skipped. When all the active clients have $q \leq 0$, they will be replenished by Q^u at Line 7. 3) After each request is added to B, the corresponding client's deficit counter will deduct the amount of service invoked by the extend tokens. 4) The new batch B then goes through one model forward step. After each decoding step, the service invoked by the output tokens will be deducted from the client's deficit counter accordingly.

4.2 Fairness Guarantees of DLPM

In this section, we provide the theoretical fairness guarantees of DLPM that correspond to the three properties we introduced in §3. The full proofs are provided in Appendix A.1.

Algorithm 1 Deficit Longest Prefix Match (DLPM)

```
1: let l denotes the client list
2: let B denotes current running batch
3: function CheckRefill(l, Queue)
4:     for all i ∈ {client(r) | r ∈ Queue} do
5:         if q_i > 0 then return
6:     for all i ∈ l do
7:         if q_i ≤ 0 then q_i ← q_i + Q^u
8: end function
9: ▷ with monitoring stream:
10: while True do
11:     if new request r from client i arrived then
12:         if i ∉ l then q_i ← 0, l ← l + u
13:         Queue ← Queue + r
14: ▷ with execution stream 1:
15: while True do
16:     Queue ← SortByPrefix(Queue)
17:     while not Queue.empty() do
18:         for each r ∈ Queue do
19:             i ← client(r)
20:             if q_i ≤ 0 then CheckRefill(l, Queue)
21:             if q_i > 0 then
22:                 if CanAdd(r) then
23:                     B ← B + r
24:                     q_i ← q_i − w_e · extend_length(r)
25:                     Queue ← Queue − r
26:     ForwardStep(B)
27:     q_i ← q_i − w_q · |{r|client(r) = i, r ∈ B}|
28:     B ← filter_finished_requests(B)
```

Theorem 4.1 (**Service bound between backlogged clients**). *Under the DLPM scheme: for any time interval $[t_1,t_2)$, if two clients f and g are continuously backlogged. Then the difference in their received service are bounded: $|W_f(t_1,t_2) - W_g(t_1,t_2)| \le 2\cdot(U+Q^u)$, where $U = w_e \cdot L_{input} + w_q \cdot M$.*

Proof. Let the client with maximum service be f, and the client with minimum service be g. Consider t_1 and t_2.

- At t_2, since both clients f and g are backlogged and are in client list l, both client f and client g have been replenished the *same* k number of times in Line 7 since t_1. f and g are backlogged, Line 5 ensures that both clients have negative q_i before reaching Line 7 and be replenished.
- Since t_1, client f at t_2 has received service $W_f(t_1,t_2) = q_f(t_1) + k\cdot Q^u - q_f(t_2)$. client g at t_2 has received service $W_g(t_1,t_2) = q_g(t_1) + k\cdot Q^u - q_g(t_2)$.
- $|W_f(t_1,t_2) - W_g(t_1,t_2)| = |q_f(t_1) - q_f(t_2) - q_g(t_1) + q_g(t_2)| \le |q_f(t_1) - q_f(t_2)| + |q_g(t_2) - q_g(t_1)| \le 2\cdot(U+Q^u)$, according to Theorem A.1.

□

Theorem 4.2 (**Service bound between backlogged and non-backlogged clients**). *Under the DLPM scheme: Client f that is continuously backlogged during time interval $[t_1,t_2)$ should not receive less service than another client, g, that is not continuously backlogged during the same time interval, that is $W_f(t_1,t_2) \ge W_g(t_1,t_2) - 2U - 2Q^u$.*

Proof.

- Consider client f and client g. f is *continuously* backlogged and g is not *continuously* backlogged.
- If g is not backlogged during the entire duration from t_1 to t_2, $W_g(t_1,t_2) \le U$, with no new request arrival.
- Let client f be replenished k_f^t at time t in Line 7.
- Since f is continuously backlogged from t_1 to t_2, $k_f^{t_2} - k_f^{t_1} \ge k_g^{t_2} - k_g^{t_1}$. A backlogged client will be replenished for the same time as another backlogged client, from Theorem 4.1. A non-backlogged client will be replenished less as it is not in the active client list (Line 5).
- $W_g(t_1,t_2) - W_f(t_1,t_2) = (q_g(t_1) + k_g^{t_2}Q^u - q_g(t_2) - k_g^{t_1}Q^u) - (q_f(t_1) + k_f^{t_2}Q^u - q_f(t_2) + k_f^{t_1}Q^u) \le 2(U+Q^u) - Q^u\cdot(k_f^{t_2} - k_f^{t_1} - k_g^{t_2} + k_g^{t_1}) \le 2(U+Q^u)$, since $Q^u\cdot(k_f^{t_2} - k_f^{t_1} - k_g^{t_2} + k_g^{t_1}) > 0$.

□

The DLPM algorithm is work-conserving since it only manipulates the dispatch order and does not reject a request if it fits into the running batch.

Theorem 4.1 and Theorem 4.2 reflect the first and second properties introduced in §3. Illustrative examples for Theorem 4.1 can be found in Fig. 8 and Fig. 12 in §7.1, where within any time interval, the difference of the received service of two continuously backlogged clients is bounded.

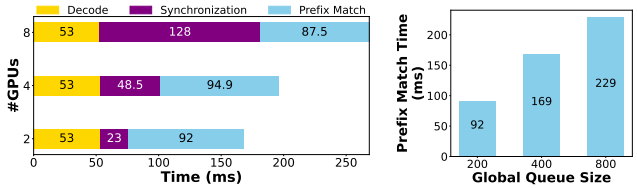
5 Applying DLPM to Distributed Scheduling

In this section, we first present the strawman solution of centralized DLPM for distributed scheduling that ignores the scheduling overhead (§5.1). We then proposed a decentralized DLPM solution that hides this overhead while preserving fairness property (§5.2).

5.1 Strawman: Centralized DLPM

The DLPM algorithm works perfectly when there is no scheduling overhead such that the DLPM scheduler could immediately make decisions based on freshest GPU states. Unfortunately, in real-world distributed scenarios, scheduling overhead happens significantly because of concurrent request handling and synchronization, prefix tree traversing and maintenance, and more. Recent work has also shown that the CPU scheduling overhead occupies nearly half of the inference time for two popular LLM inference engines [45].

Global-local States Synchronization To enable global DLPM for fair scheduling in distributed setups, we need to synchronize local and global prefix caching information. This



(a) Scheduler Overhead Breakdown. The global queue size is 200. Decode batch size is 25.

(b) Prefix match overhead w.r.t global queue size.

Figure 4: Global scheduler overhead breakdown w.r.t data parallelism degree and global queue size. The time for one decode step with bs=25 is also reported for reference. Existing serving engines such as vLLM [21] and SGLang [57] normally perform a continuous batching step after multiple (e.g., 10 in SGLang) decoding steps.

synchronization ensures that the global scheduler can replicate the decision-making process typical of a single worker. Using the token RadixTree from SGLang [57] as an example, to construct an accurate global RadixTree at time t_i (assume the last time the global scheduler dispatches the requests at time t_{i-1}), updates from each worker s are encapsulated as $\Delta Tree_s$, defined as:

$$\Delta Tree_s(t_{i-1}, t_i) = (N_{\text{inserted}}, N_{\text{evicted}}, M_{\text{KV}})_s$$

where N_{inserted} and N_{evicted} are sets of nodes that have been inserted to or evicted from the RadixTree, between the last dispatch time t_{i-1} and the current time t_i. M_{KV} indicates the current available KV cache memory.

Upon sending these updates, the worker enters a blocked state, awaiting new requests from the global scheduler. The global scheduler then updates the RadixTree accordingly and dispatches new requests to the local worker following the DLPM algorithm. Such a synchronous approach guarantees the effectiveness and correctness of DLPM in the distributed setup; however, it incurs significant overhead due to the need to block workers while awaiting new requests, and the race conditions on the global waiting queue across workers.

Overhead Analysis The global scheduler's overhead primarily stems from synchronization overhead, algorithmic overhead (e.g., the frequent tree-matching overhead for the global waiting queue), and metadata updates overhead. Among these, the metadata updates overhead per worker remains relatively constant as the system scales. However, the synchronization and algorithmic overhead increase dramatically as the data parallelism degree (D)[2] and global queue size increases, as shown in Fig. 4. The prefix matching process (algorithmic overhead) involves matching all incoming requests in the global waiting queue against each worker's radix tree and sorting them based on prefix length to determine the dispatch order. The "Prefix Match" time (blue) increases significantly as the global queue size increases (Fig. 4b), which is normally the case when the data parallelism degree grows.

[2]Here data parallelism degree refers to the number of model replicas in the distributed settings.

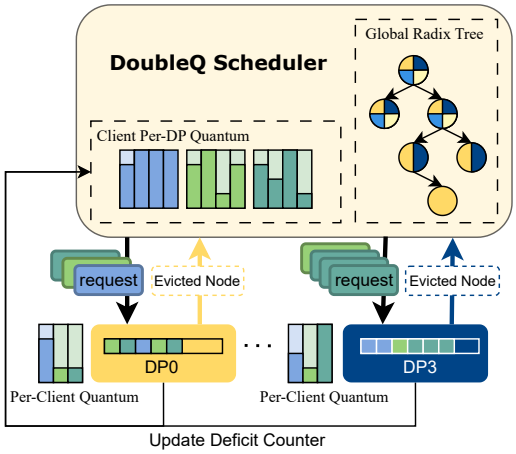


Figure 5: An overview of the DoubleQ scheduler. The global scheduler tracks the deficit counters for each client per worker to control the "stickiness" of a client to a worker. The local schedulers maintain the deficit counters for each client to enforce the fair sharing of the local GPU resources.

Overall, Fig. 4 demonstrates how synchronization and algorithmic overheads dominate as the data parallelism degree increases, particularly for higher degrees ($D = 8$) – they add to around 40% decoding overhead in the demonstrated case. This analysis underscores the challenges of designing scalable global schedulers to mitigate synchronization and algorithmic bottlenecks as the system scales.

Besides the significant scheduling overhead, the Global DLPM scheduler also requires extensive modification of the local worker to enable local-global information synchronization and the blocking operation to wait for the global scheduler dispatching requests.

5.2 Our Solution: Decentralized DLPM

To mitigate *the global scheduling overhead and tight coupling* between the global scheduler and the local worker, we resort to decentralized scheduling: dispatching the requests directly to local workers and queueing them at the local worker instead of the global scheduler. Most of the existing distributed schedulers for LLM serving (e.g., Preble [44] and SGLang [56]) follow this design.

In such a decentralized design, the local worker can directly run a fair scheduling algorithm (e.g., DLPM); as long as the global scheduler can balance the per-client service on all the local workers, we could achieve global fairness guarantee [3]. Previous works in CPU scheduling [1] and wireless LANs bandwidth sharing [3] also demonstrate the effectiveness of such design. Therefore, the challenge now becomes how to strike a good trade-off between load balancing and locality.

Double Quantum (DoubleQ) Our key insight is to prioritize

Algorithm 2 DoubleQ Scheduling

1: let s_w denotes the current queue size of worker w.
2: $W \leftarrow$ GETWORKERS(), $R \leftarrow$ INITRADIXTREE($|W|$)
3: **function** SELECTWORKER(G, i)
4: $G_{avail} \leftarrow \{w \mid q_{i,w} > 0\}$
5: **while** $G_{avail} == \emptyset$ **do**
6: **for all** $w \in W$ **do** $q_{i,w} \leftarrow q_{i,w} + Q^w$
7: $G_{cand} \leftarrow G \cap G_{avail}$
8: **if** $G_{cand} == \emptyset$ **then return** $\arg\min_{w \in G_{avail}} s_w$
return $\arg\min_{w \in G_{cand}} s_w$
9: **end function**
10: ▷ `with concurrent stream 1:`
11: **while** True **do**
12: **if** new request r from client i arrived **then**
13: $G \leftarrow$ R.LONGESTMATCHWORKERS(r)
14: $w \leftarrow$ SELECTWORKER(G, $client(r)$)
15: DISPATCH(w, r)
16: $q_{i,w} \leftarrow q_{i,w} - w_e \cdot r$.input_tokens
17: $s_w \leftarrow s_w + 1$
18: R.INSERT(r.input_tokens, w)
19: ▷ `with concurrent stream 2:`
20: **while** True **do**
21: **if** request r from client i has finished at worker w **then**
22: $q_{i,w} \leftarrow q_{i,w}$ - $w_q \cdot r$.output_tokens
23: $s_w \leftarrow s_w - 1$
24: ▷ `with concurrent stream 3:`
25: **while** True **do**
26: **if** prefix P has been evicted at worker w **then**
27: R.EVICT(P, w)

locality first until certain limits are met: we use the quantum mechanism again to avoid a client becoming too sticky to a single worker due to the prefix cache locality by assigning quantum to each worker for each client. As demonstrated in Fig. 5 and Algorithm 2, for each new request, the global scheduler first matches it with the global radix tree and get the workers G that have its longest-matched prefix (Line 13). Then in the SELECTWORKER function (Line 3), if any wokers in G has deficit counter larger than 0 (G_{avail}), the worker with minimum queue size in $G \cap G_{avail}$ will be chosen. Otherwise, the worker with minimum queue size in G_{avail} will be selected. After each request is dispatched, the request's input tokens will be inserted into the global radix tree and the corresponding deficit counter will be updated (Line 17). The global scheduler will periodically update corresponding deficit counter when there are requests finished (Line 23) as well as prune the global radix tree with collected local workers' eviction information (Line 27). Note that unlike the centralized DLPM where the eviction information needed to be passed to the global scheduler synchronously, in DoubleQ this happens asynchronously with negligible overhead.

We note that our DoubleQ scheduling (with local workers running DLPM)[3] provides global fairness guarantees corresponding to the properties introduced in §3 through the following theorems.

Theorem 5.1 (Service bound between backlogged clients). *At any time interval* $[t_1, t_2)$, $\max_i W_i(t_1,t_2) - \min_i W_i(t_1,t_2) \leq 2 \cdot |W| \cdot (U + Q^u)$. *The difference between the maximum service among all backlogged clients and the minimum service among all backlogged clients is bounded by* $2 \cdot |W| \cdot (U + Q^u)$, *where* $|W|$ *is the number of workers.*

Theorem 5.2 (Service bound between backlogged and non-backlogged clients). *Consider any execution of the DoubleQ scheme. Client* f *that is continuously backlogged during time interval* $[t_1, t_2)$ *should not receive less service than another client,* g, *that is not continuously backlogged during the same time interval, where* $W_g(t_1,t_2) - W_f(t_1,t_2) \leq 2 \cdot (U + Q^u) \cdot |W|$.

Since there are no requests rejected to enforce fairness, DoubleQ scheduling is work-conserving.

6 Evaluation

6.1 Setup

Implementation We implement our DLPM and DoubleQ schedulers with 1000 LoC in Python on top of SGLang [57], a fast industry-standard LLM inference system.

Models and Hardware Our evaluation is conducted on the widely-used model Llama-3.1-8B and Llama-3.2-3B-Instruct [10]. Other transformer-based LLMs such as Qwen [50], DeepSeek [7], and Mistral [18] share a similar backbone architecture and are also compatible with our system. For hardware, we test on NVIDIA A100 80GB and A10G GPUs.

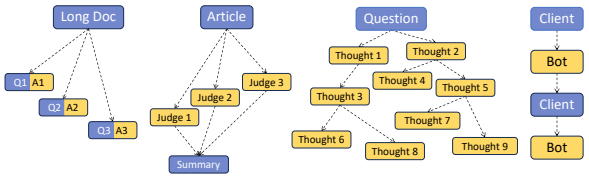
Table 2: Workload configurations.

Workload	Dataset	Avg Prefix Len.	Avg Output Len.
Long-context QA	LooGLE [23]	21449	15
Tree of Thoughts	GSM8K [6]	546	256
LLM-as-a-Judge	Synthetic articles [57]	2701	256
Real multi-turn	Chatbot Arena [56]	56	142

Workloads and Datasets We evaluate the efficiency and effectiveness of the schedulers on 4 diverse LLM-based workloads, each characterized by its unique execution graph structures (Fig. 6) and variations in prefix and output length distributions. as detailed in Tab. 2. Specifically, we evaluate long document understanding using the LooGLE [23] dataset. We implement the Tree-of-Thought [51] program for solving GSM8K [6] problems (with a tree height of 4), and the

[3] Generally, in DoubleQ, the local worker can run any other fair scheduling algorithms such as VTC. In this paper, DoubleQ specifically refers to the implementation using DLPM at the local workers.

LLM-as-a-Judge [57] program, which utilizes the branch-solve-merge technique to evaluate synthetic articles. We also conduct experiments on real-world multi-turn conversation traces from Chatbot Arena [56].



(a) Long-Context QA (b) LLM-as-a-Judge (c) Tree of Thoughts (d) Multi-Turn Chat

Figure 6: Illustration of the execution graphs of different workloads in our benchmark.

Synthetic Traces For Long-context QA, Tree-of-Thoughts, and LLM-as-a-Judge, we generate synthetic client request traces following the Gamma process, as done in [24, 40, 41], with the request rate increasing as the number of GPUs scales.

For these three workloads, we evaluate two distinct types of misbehaving patterns, as detailed in Tab. 3. The first type (S1) involves a misbehaving client sending more requests than well-behaved clients. Specifically, although all clients send programs at the same request rate, the misbehaving client submits programs with a more complex execution graph (e.g., more branches in Tree-of-thought). The second type (S2) features a misbehaving client sending programs with the same structural complexity and at the same request rate as well-behaved clients, but with the input altered to increase the prefix length. These workloads are evaluated with the Llama-3.1-8B model served on NVIDIA A100 GPUs. The related results are reported in §6.2.

Table 3: Synthetic Workload Configurations. 👎 stands for misbehaving client and 👍 denotes well-behaved clients.

Workload	Detailed Behavior
S1: More Requests	
Long-context QA	👎: Higher req rate
Tree-of-thought	👎: Trees of 4 branches (340 req per tree) 👍: Trees of 2 branches (30 req per tree)
LLM-as-Judge	👎: Evaluation with 16 dimensions 👍: Evaluation with 2 dimensions
S2: Longer Prefix	
Long-context QA	👎: 2× longer input documents
Tree-of-thought	👎: 10× longer input questions
LLM-as-Judge	👎: Extra 600 tokens before each article

Real-world Traces For real-world multi-turn conversation, we re-scale the request time stamps provided in the dataset[4] and aggregate multiple clients' requests to closely mimic high-demand scenarios. This workload is evaluated with the Llama-3.2-3B-Instruct model served on NVIDIA A10G GPUs. The related results are reported in §6.3.

Baselines We compare DLPM and DoubleQ with three baseline scheduling algorithms.

- **DoubleQ**: The local worker adopts DLPM, and the global scheduler runs the DoubleQ algorithm when Data Parallelism Degree $D > 1$.
- **RR + LPM**: The local scheduler runs LPM, and the global scheduler uses the round-robin (RR) algorithm when $D > 1$. It is the default distributed scheduling algorithm in SGLang [57] without fairness guarantees.
- **Preble** [44]: Preble is a state-of-the-art distributed LLM serving system that aims to provide high serving throughput by balancing load distribution and locality, yet without fairness guarantee. Specifically, it dispatches requests based on a pre-defined prefix-matching ratio to decide whether to explore a new GPU or exploit locality.
- **VTC**: The local scheduler runs VTC [41], and the global scheduler applies a per-client round-robin strategy when $D > 1$. Extending VTC with round-robin scheduling is the straightforward approach to ensuring fairness in distributed settings, with fairness bound proven in Appendix A.3.

Metrics To measure the system efficiency and fairness achieved by different scheduling algorithms, we use the following three metrics:

- **Service Rate**: We measure the clients' service as a weighted sum of the number of input tokens and the number of output tokens, following VTC [41][5]. As discussed in §3, the weight for input token is 1 and the weight for output token is 2.
- **Jain's Fairness Index** [17] is a widely-used metric for evaluating the fairness of resource allocation in networked systems [22]. The index is mathematically defined as:

$$J(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2},$$

  where x_i represents the allocation for the i^{th} client, and n is the total number of clients. The value of J ranges from $\frac{1}{n}$ (minimum fairness, when one client monopolizes all resources) to 1 (maximum fairness, when resources are equally distributed). In our context, we compute the Jain's Fairness Index by letting x_i denote the service rate of client i. The calculation is based on the time interval during which all clients are active, ensuring an accurate representation of fairness across the system.
- **P50 and P99 Latency**: We assess the scheduler's effectiveness in maintaining service quality for well-behaved clients by measuring their P50 and P99 latency. We measure latency using the end-to-end completion time for program evaluation. We use the TTFT (Time to First Token) latency[6] metric for long-context QA tasks.

[4] https://huggingface.co/datasets/lmsys/chatbot_arena_conversations

[5] Note that this service rate is from clients' perspective. From the system's perspective, the actual service is measured by the cost function using the number of extend tokens.

[6] For QA tasks, a shorter TTFT contributes to improved client experiences.

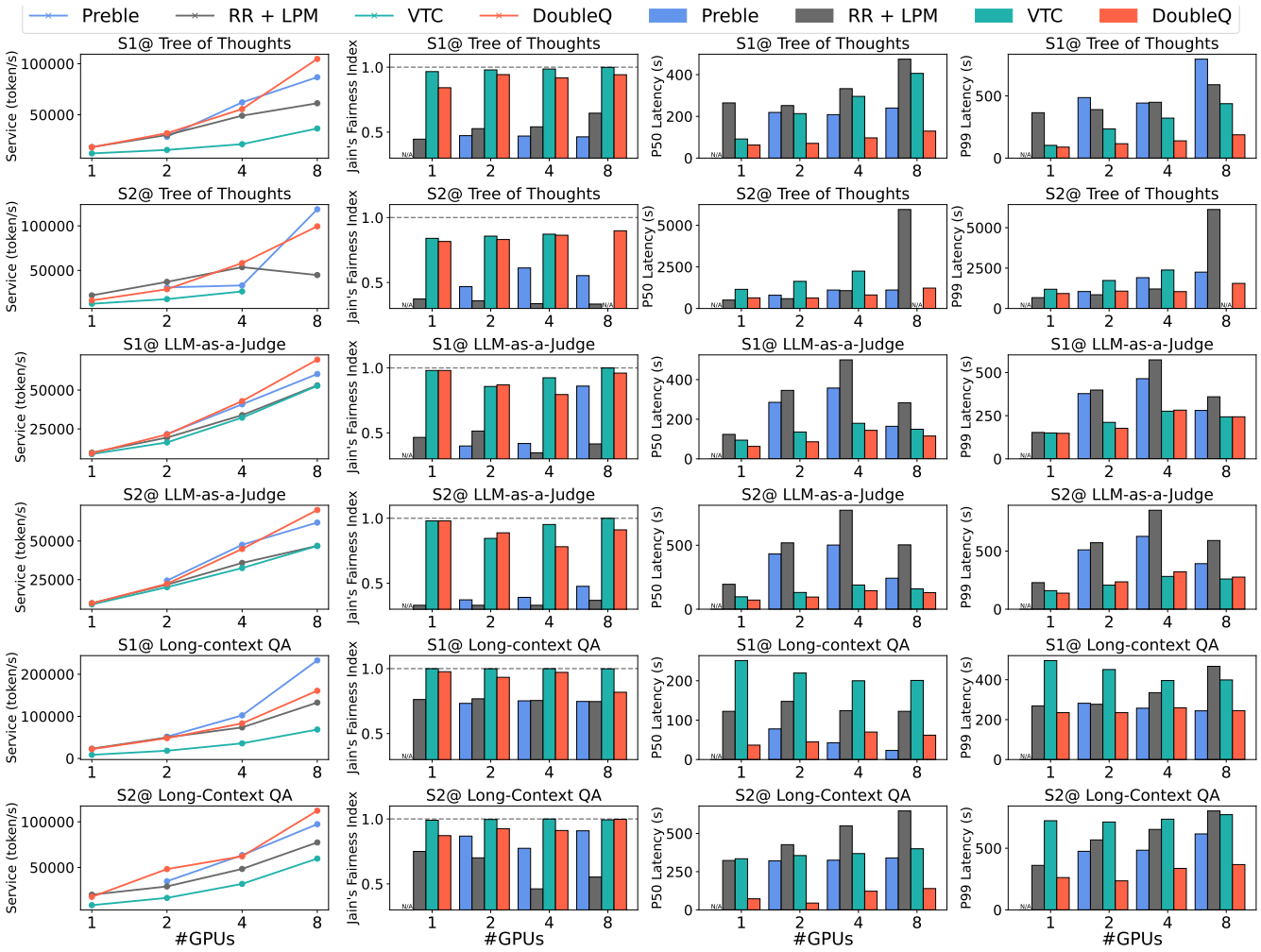


Figure 7: Summary of results across three datasets and two types of misbehaving clients on up to 8 A100 GPUs (8B model). The reported latency represents the average latency for well-behaved clients. The data point for S2@Tree of Thoughts with $D = 8$ is omitted, as it takes too long to complete.

6.2 Results on Synthetic Traces

We present all three metrics across three workloads and two types of misbehaving clients in Fig. 7. Both VTC and DoubleQ provide theoretical fairness guarantees, whereas Preble and RR + LPM do not. The data point for Preble with $D = 1$ is omitted because Preble is designed as a multi-GPU cache-aware prompt dispatch system.

Throughput Analysis As previously discussed, ensuring fairness inherently competes with maximizing throughput. However, DoubleQ's effective global cache-aware scheduler and DLPM enable significant performance gains, achieving up to a 2.87× improvement compared to the only other fair algorithm, VTC.

DoubleQ achieves better throughput than RR + LPM, with improvements of up to 2.22×. In the case of S2@Tree of Thoughts, the poor performance of RR + LPM with $D = 8$ compared to $D = 4$ can be attributed to the complex sharing patterns inherent in Tree of Thoughts. Round Robin fails to preserve locality among GPUs, leading to a significant drop in cache hit rate (i.e., from 95% to 50%). This limitation indicates that RR + LPM does not scale effectively when clients submit complex LLM programs.

Compared to Preble, DoubleQ consistently matches or exceeds its performance across all workloads and GPU configurations, demonstrating its ability to sustain high throughput while ensuring fairness. An exception arises for S2@Long-context QA with $D = 8$, where Preble outperforms DoubleQ in throughput. This discrepancy occurs because DoubleQ sacrifices some locality to maintain fairness, resulting in increased prefix recompute overhead. As indicated in Tab. 2, the LooGLE dataset features an exceptionally high prefix length-to-output length ratio. In this case, the cost of recomputing long documents becomes substantial, with the prefill stage

significantly dominating the generation time. Consequently, the Long-context QA workload serves as a worst-case scenario that adversely impacts DoubleQ's throughput. However, when the prefix length-to-output length ratio falls within a reasonable range, the DoubleQ algorithm consistently matches and even slightly surpasses the performance of state-of-the-art non-fair scheduling algorithms. This is achieved through the careful management of load balance and locality trade-offs in the global scheduler, as well as locality and fairness trade-offs in the local scheduler.

Jain's Fairness Index Analysis From the second column in Fig. 7, it is clear that DoubleQ consistently outperforms both Preble and RR + LPM. This is because DoubleQ provides strict fairness guarantees. However, it is slightly less fair than VTC, as DoubleQ relaxes the fairness bounds to improve locality, which leads to higher throughput but slightly worse fairness control. Preble performs slightly better than RR + LPM due to its multi-level priority wait queue, which avoids starvation but cannot provide isolation and strict fairness guarantee. As a result, there remains a notable gap between Preble and DoubleQ.

Well-behaved Clients' Latency Analysis We use the average P50 and P99 latency of well-behaved clients to evaluate the experience of well-behaved clients when a misbehaving client is present. Algorithms focusing on high system efficiency might inadvertently increase latency for well-behaved clients as these schedulers may prioritize the requests from the misbehaving clients to optimize the prefix cache hit rate. Preble and RR + LPM, therefore, can result in up to 7.18× and 9.55× higher latency, respectively, compared to DoubleQ. On average, DoubleQ achieves 2.90× and 4.06× lower latency than Preble and RR + LPM. On the other hand, algorithms that focus solely on fairness will also incur high latency for well-behaved clients due to reduced overall system efficiency. For instance, VTC can lead to latency up to 7.96× higher than DoubleQ, with an average latency increase of 2.98×.

6.3 Results on Real-world Traces

Figure 8 demonstrates the fairness and performance comparison of different schedulers on the real-world multi-turn conversation workload. In this workload, clients 2 and 3 initially send excessive number of requests, and then return to normal midway. A fair scheduler should prevent these two clients from impacting other clients. Due to LPM's prioritization strategy, which favors requests with longer prefix matches, Clients 2 and 3 receive a disproportionately large share of resources. As a result, Clients 0 and 1 suffer from high response times and reduced throughput. In contrast, VTC achieves relatively low response times and maintains high throughput for Clients 0 and 1. However, such strict fair allocation comes at the expense of Clients 2 and 3, who endure substantial response delays, reaching up to 80 seconds.

DLPM achieves a more reasonable distribution of resources, protecting well-behaved clients from the disruptive effects of high request rates by the misbehaving clients. DLPM ensures consistently low response time and high throughput for both well-behaved and previously misbehaving clients. Thus, DLPM not only mitigates the impact of malicious usage patterns but also improves overall system performance and fairness compared to the baseline approaches.

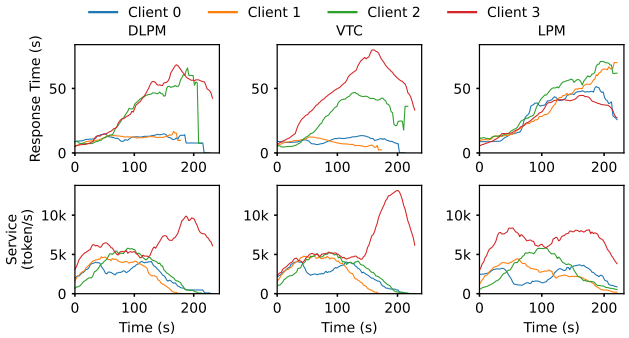


Figure 8: Fairness and performance visualization for the real-world multi-turn conversation workload ($D = 2$). Clients 2 and 3 send requests at a much higher rate than Clients 0 and 1.

7 Ablation Studies

7.1 Visualization of Fairness Properties

We visualize the response time and the services provided by the server to different clients over time in Fig. 9. The experiments use 4 A10G GPUs as the testbed, with all clients sending Tree-of-Thought programs at the same rate and with a consistent branch count of 3. However, client 0 is misbehaving by sending a longer prefix, i.e. 10× longer than well-behaved clients. The maximum value on the x-axis represents the end-to-end completion time of all programs. As observed, DoubleQ achieves the shortest execution time, demonstrating up to 2× speedup compared to VTC and Preble.

From the first row of the figure, we observe that DoubleQ consistently maintains lower response times compared to VTC as it preserves a higher degree of locality, which enhances overall system efficiency. Furthermore, it avoids the excessively high response time caused by schedulers like RR + LPM and Preble, which lack fairness control. These schedulers tend to prioritize serving client 0, resulting in substantial delays for other clients. For instance, as shown in the figure, the service received by clients 1 and 2 between 100s and 700s is almost zero for Preble, causing a queuing latency of up to 600 seconds.

The second and third rows depict the actual service and the service received by each client, respectively. As shown in the second row of Fig. 9, both VTC and DoubleQ achieve an ideal sharing of resources across the 4 GPUs in terms of actual service. In the third row, we can observe that the

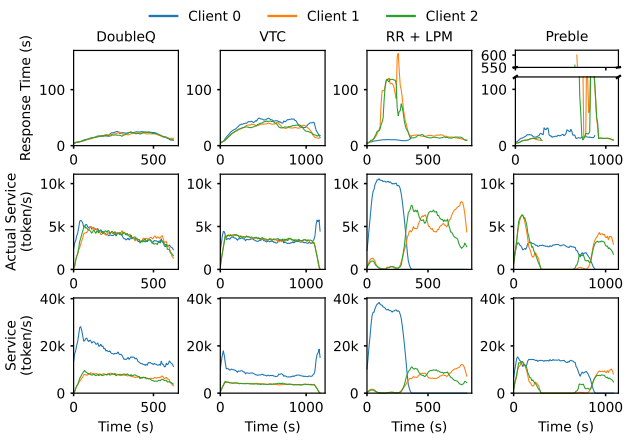


Figure 9: Fairness and performance visualization of different schedulers on Tree-of-Thought workloads with $D = 4$ (3B model + 4 A10G GPUs). The maximum value on the X-axis represents the end-to-end completion time for each scheduler. The actual service is calculated using the cost function defined in §3, which is a weighted sum of the number of extend tokens and the number of output tokens.

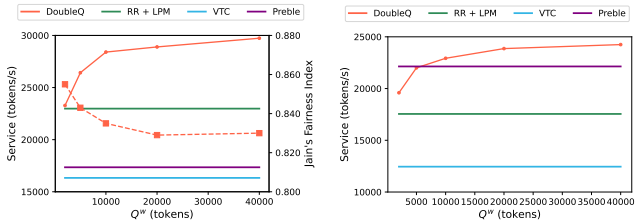
service rate of client 0 is higher than clients 1 and 2 – this is because client 0 has longer prefix sharing and thus lower cost per token. However, due to the relatively low cache hit rate of VTC, it experiences worse end-to-end performance. In contrast, the other two algorithms demonstrate significant unfairness in resource allocation across clients.

A key highlight here is the extremely low throughput observed with Preble. Preble prioritizes dispatching requests to the GPU with the longest prefix-matching length, provided the matching length exceeds a predefined threshold. Between 300 and 600 seconds, client 0's requests are continuously dispatched to a single GPU as the prefix-matching ratio will always exceed the pre-defined threshold. Some requests from clients 1 and 2 get queued at this monopolized GPU, which blocks these clients from generating new requests (i.e., "deeper" thoughts), due to the inherent LLM call dependencies in the Tree-of-thought programs. This results in severe workload imbalance among the GPUs, with the cluster at merely 1/4 of its potential computational capacity.

7.2 Impact of Q^u in DLPM and Q^w in DoubleQ

We now examine the trade-off between locality and fairness using Q^u and Q^w. The impact of Q^u is illustrated in Fig. 1, where increasing Q^u enhances throughput but compromises fairness control. By adjusting the value of Q^u, the server can achieve a tailored trade-off between performance and fairness, defining a new Pareto frontier compared to VTC and LPM.

Fig. 10 illustrates the impact of Q^w on throughput in DoubleQ. To recap, Q^w represents the quantum of service assigned to each worker in DoubleQ, where a larger Q^w typically implies a better locality for requests within a client. As shown in



(a) Throughput of Tree-of-Thoughts with one misbehaving client

(b) Throughput of Tree-of-Thoughts with all well-behaved clients

Figure 10: Impact of Q^w on throughput under different workloads ($D = 4$). The solid line represents throughput, while the dashed line represents Jain's Index. The fairness index in (b) is omitted as it consistently equals 1.

Fig. 10, as Q^w increases, the throughput of DoubleQ also increases, eventually stabilizing and surpassing all other schedulers. The low throughput of Preble, as seen in Fig. 10a, has been explained earlier in §7.1.

Although Q^w is not explicitly included in the fairness bound of DoubleQ as demonstrated in Appendix A.2, it does slightly affect Jain's Fairness Index. Specifically, the index decreases from 0.855 to 0.83 when Q^w increases from 2000 to 40000, due to the more unbalanced dispatching of requests within a client [7].

7.3 Scaling the Number of Clients

We assess DLPM's performance as we increase the number of clients from 5 to 50, using a single A10 GPU as the testbed, while maintaining a constant total request rate. As depicted in Fig. 11, DLPM consistently achieves a service rate comparable to LPM, even as the number of clients increases, whereas VTC consistently underperforms.

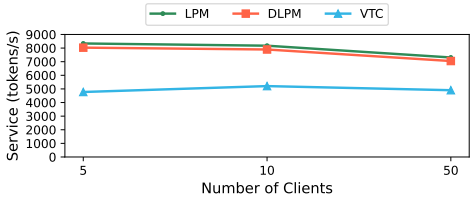


Figure 11: Service rate w.r.t the number of clients on a single A10 GPU (3B model).

Note that as the number of clients rises, the number of distinct prefixes in the same volume of requests increases, which marginally reduces the cache hit rate for both DLPM and LPM, leading to a slight decrease in service rate as the number of clients increases.

[7] When Q^w is set to infinity, the algorithm is reduced to be similar as Preble, which lacks fairness guarantees since the difference in load across workers becomes unbounded, as proven in Theorem A.5

In contrast, VTC's performance is less affected since its cache hit rate is consistently low regardless of the number of clients.

7.4 Mix of Workloads

As a complement to the single-workload scenario discussed earlier, we now explore a more realistic setting where clients handle diverse workloads. As shown in Fig. 12, DLPM consistently achieves better response time and end-to-end execution times compared to the other schedulers. In the LPM scheduler, clients sending Tree-of-Thoughts programs act as misbehaving clients, significantly increasing the response time for other clients. From the second row, we observe that VTC exhibits better fairness control than DLPM, as it provides more evenly distributed actual service across clients. This demonstrates that DLPM sacrifices some degree of fairness to achieve higher throughput.

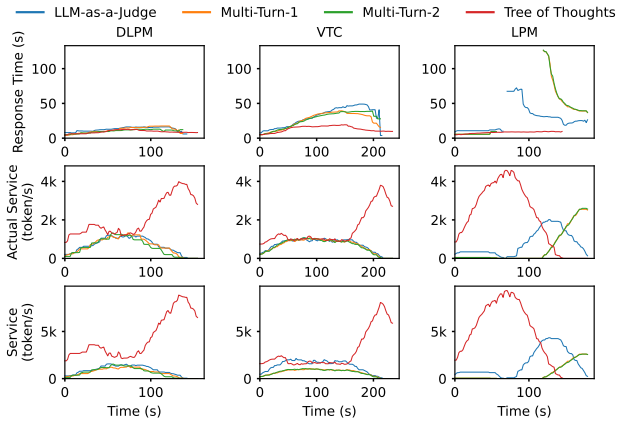


Figure 12: Mix of workloads among four clients: two engage in multi-turn conversations, while the other two send different programs, all within a single-GPU setup (3B model + an A10G GPU).

8 Related Work

Fairness in ML Workloads ML training workloads have extensively studied the fairness problems in shared clusters [5, 26, 27, 38]. Due to their unique characteristics such as long running time, placement sensitivity, and statistical efficiency (i.e., the amount of progress per unit of data consumed), traditional fair scheduling for big data workloads [15, 16] does not work well. To handle the long-running and placement-sensitive natures of ML training workloads, Themsis [26] proposes new finish-time fairness metrics, and leverages multi-round partial allocation auctions to provide Pareto-efficient and envy-free resource allocations. To consider statistical efficiency for higher cluster-wide resource utilization, Pollux [38] introduces goodput-driven cluster scheduling by jointly optimizing resource allocations and job batch sizes. On the other hand, prior work VTC and our work focus on the LLM inference-time fairness. Compared to VTC, our work co-optimizes both fairness and prefix sharing for higher performance without losing fairness.

Fairness in Other Workloads Fairness is a long-existing topic in networking and operating systems. For example, networking needs to guarantee fairness among different switching ports [42] and during link bandwidth allocation [9, 13, 14, 19, 33]; OS scheduling needs to guarantee fair CPU time share among different processes [20, 46], and fair memory allocations [28]. Fairness is also extensively studied in big data workload scheduling with prominent prior work of Delay Scheduling [53] and Dominant Resource Fairness [11]. Our fair scheduling design is inspired by many prior work such as Deficit Round Robin [42] and Delay Scheduling [53]; but differently, we explicitly optimize for the prefix sharing property in LLM inference workloads while guaranteeing fairness.

Locality in LLM Inference Previous advances in LLM inference focus on batching and memory optimization [21, 52]. SGLang further exploits locality in scheduling to improve LLM inference performance for emerging applications such as multi-turn chatting [57]. It leverages the LPM scheduling with RadixTree to save GPU memory and avoid redundant computations through prefix sharing. Preble [44] further extends LPM into distributed settings to jointly optimize load balancing and prefix caching locality for high throughput. BlendServe [55] co-optimizes GPU resource overlapping and prefix sharing for offline LLM inference, achieving nearly optimal inference throughput. Unlike the above work which only focuses on inference throughput, our work presents a principled way of navigating the trade-off between performance and fairness in multi-client scenarios.

9 Conclusion

This paper introduces the first prefix-aware fair scheduling algorithm for LLM serving, namely, DLPM. We also propose an extension of the algorithm, DoubleQ, to preserve locality with global fairness guarantee in a distributed setup. Our algorithm achieves up to 2.87× higher throughput than state-of-the-art fair scheduling algorithms in LLM like VTC, and 7.18× lower latency for victim clients compared to locality-aware scheduling algorithms like Preble.

References

[1] Linux 2.6.23. Completely fair scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html.

[2] Perplexity AI. Perplexity: Conversational Search Assistant. https://www.perplexity.ai.

[3] Yigal Bejerano, Seung-Jae Han, and Li Li. Fairness and load balancing in wireless lans using association control. In *Proceedings of the 10th annual international conference on Mobile computing and networking*, pages 315–329, 2004.

[4] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

[5] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[6] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[7] Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. *CoRR*, abs/2401.06066, 2024.

[8] DeepSeek. Deepseek-r1-lite-preview release. https://api-docs.deepseek.com/news/news1120, 2024. Accessed: 2024-11-20.

[9] Alan J. Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In Lawrence H. Landweber, editor, *ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, pages 1–12. ACM, 1989.

[10] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[11] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of Networks and Systems Design and Implementation (NSDI)*, 2011.

[12] Github. Github copilot: Your ai pair programmer. https://github.com/features/copilot.

[13] S. Jamaloddin Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proceedings IEEE INFOCOM '94, The Conference on Computer Communications, Thirteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Networking for Global Communications*, pages 636–646. IEEE Computer Society, 1994.

[14] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 157–168. ACM, 1996.

[15] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in Multi-Resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, Savannah, GA, November 2016. USENIX Association.

[16] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 261–276. Association for Computing Machinery, 2009.

[17] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 21:1, 1984.

[18] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[19] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. In *International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 37–48. ACM, 2004.

[20] The kernel development community. CFS Scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html.

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

[22] Tian Lan, David Kao, Mung Chiang, and Ashutosh Sabharwal. An axiomatic theory of fairness in network resource allocation. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, 2010.

[23] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. Loogle: Can long-context language models understand long contexts? *arXiv preprint arXiv:2311.04939*, 2023.

[24] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[25] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

[26] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient gpu cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[27] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.

[28] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 208–222, 2006.

[29] Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Prompting LLMs for efficient parallel generation. In *The Twelfth International Conference on Learning Representations*, 2024.

[30] OpenAI. Gpt-4 technical report, 2023.

[31] OpenAI. Rate limit. https://platform.openai.com/docs/guides/rate-limits?context=tier-free, 2023.

[32] OpenAI. Learning to reason with llms. https://openai.com/index/learning-to-reason-with-llms/, 2024. Accessed: 2024-11-20.

[33] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[34] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *In the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*, UIST '23, New York, NY, USA, 2023. Association for Computing Machinery.

[35] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

[36] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

[37] Pranav Putta, Edmund Mills, Naman Garg, Sumeet Motwani, Chelsea Finn, Divyansh Garg, and Rafael Rafailov. Agent q: Advanced reasoning and learning for autonomous ai agents. *arXiv preprint arXiv:2408.07199*, 2024.

[38] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

[39] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

[40] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023.

[41] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in Serving Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 965–988, 2024.

[42] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Trans. Netw.*, 4(3):375–385, 1996.

[43] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

[44] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient Distributed Prompt Scheduling for LLM Serving. 2024.

[45] Vikranth Srivatsa, Dongming Li, Yiying Zhang, and Reyna Abhyankar. Can Scheduling Overhead Dominate LLM Inference Performance? A Study of CPU Scheduling Overhead on Two Popular LLM Inference Systems. https://mlsys.wuklab.io/posts/scheduling_overhead/.

[46] Ion Stoica and Hussein Abdel-Wahab. Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation. *Old Dominion Univ., Norfolk, VA, Tech. Rep. TR-95-22*, 1995.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[48] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv: Arxiv-2305.16291*, 2023.

[49] Wikipedia. Max-min fairness. https://en.wikipedia.org/wiki/Max-min_fairness.

[50] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

[51] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[52] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 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, 2022.

[53] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: a Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278, 2010.

[54] Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. Rest-mcts*: Llm self-training via process reward guided tree search. *arXiv preprint arXiv:2406.03816*, 2024.

[55] Yilong Zhao, Shuo Yang, Kan Zhu, Lianmin Zheng, Baris Kasikci, Yang Zhou, Jiarong Xing, and Ion Stoica. BlendServe: Optimizing Offline Inference for Auto-regressive Large Models with Resource-aware Batching. *arXiv preprint arXiv:2411.16102*, 2024.

[56] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

[57] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, 2023.

A Appendix

A.1 Proof for Local DLPM

Theorem A.1 (Service Bound). *Consider any execution of the DLPM scheme in which client i is backlogged. After any K_i rounds (where q_i is replenished K_i times) from t_1 to t_2, the difference between $K_i \cdot Q^u$ (i.e., the service that client i should have sent) and $W_i(t_1,t_2)$ (i.e., the service that client i actually received) is bounded by* $\max(Q^u,U)$*, where* $U = w_e \cdot L_{input} + w_q \cdot M$.

Proof. Let $q_i(t)$ denote the deficit counter value of client i at time t. Since the deficit counter will only be refilled when $q_i \leq 0$ (line 7) by Q^u, we have

$$q_i(t) \leq Q^u \tag{1}$$

Now we prove through induction:

$$q_i(t) > -U \tag{2}$$

- At the beginning, all $q_i(0) = 0$. Equation (2) holds.
- We then prove if at time t, Equation (2) holds, then for $t' > t$, Equation (2) also holds.
- At line 7, $q_i(t') = q_i + Q^u > q_i > -U$. Equation (2) holds.
- Since line 24 will be reached only when $q_i > 0$, $q_i(t') = q_i - w_e \cdot extend\_length(r) > -w_e \cdot L_{input}$. Equation (2) holds.
- At line 27, since $q_i(t') = q_i - w_q \cdot |\{r|client(r) = i, r \in B\}|$ will be repeated for n steps until some requests are finished. Therefore, we have $q_i(t') \geq q_i - n \cdot w_q \cdot |\{r|client(r) = i, r \in B\}|$. Since the number of decoded tokens cannot exceed the server's maximum token capacity M, $n \cdot |\{r|client(r) = i, r \in B\}| \leq M$. We then have $q_i(t') = q_i - w_q \cdot M > -U$. Equation (2) holds.

Therefore, we have $W_i(t_1,t_2) = K_i \cdot Q^u - q_i(t_2)$. Combining Equation (2) and Equation (1), we have:

$$|W_i(t_1,t_2) - K_i \cdot Q^u| = |q_i(t_2)| \leq \max(Q^u,U) \tag{3}$$

□

Theorem A.2 (Latency Bound). *Let $A(r)$ and $D(r)$ denote the arrival time and dispatch time of a request r. Assume there are in total n clients, $\forall t_1,t_2$, if at t_1, a client f is not backlogged and has no requests in the running batch, then the next request r_f with $t_1 < A(r_f) < t_2$ will have its response time bounded: $D(r_f) - A(r_f) \leq 2 \cdot (n-1) \cdot \frac{Q^u+U}{a}$, where a is the lower bound of the server capacity.*

Proof.
- Since there is no running batch of f in the system, r_f will be selected for the next request for f.
- Earlier, we have shown that the service bound for backlogged clients compared to either backlogged or non-backlogged clients is $2 \cdot (Q^u + U)$.
- From t_1 to $D(r_f)$, $W_f(t_1, D(r_f))$ will be within $2 \cdot (Q^u + U)$ of service received by other clients.
- Since at Line 12, q_f is set to 0 when f rejoins, the maximum number of tokens served before f is served again is: $2 \cdot (n-1) \cdot (Q^u + U)$, where $n-1$ is the $n-1$ other clients.
- Given that a is the lower bound of the server capacity, the dispatch time for f is therefore bounded: $D(r_f) - A(r_f) \leq 2 \cdot (n-1) \cdot \frac{Q^u+U}{a}$.

□

A.2 Proof for DoubleQ Scheduling

Theorem A.3 (Service Bound). *Consider any execution of the DoubleQ Scheduling scheme in which client i is backlogged. The difference between $\sum_{w \in W} k_{i,w} \cdot Q^u$ (i.e., the service that client i should have sent) and W_i (i.e., the service that client i actually received) is bounded by* $\max(Q^u,U) \times |W|$*, where $U = w_e \cdot L_{input} + w_q \cdot M$. Let $k_{i,w}$ is the number of times client i has been replenished at worker w.*

Proof. Let $k_{i,w}$ be the number of times the client i has replenished quantum locally at worker w. We want to show for a client i:

$$\sum_{w \in W} k_{i,w} \cdot Q^u - \sum_{w \in W} (k_{i,w} \cdot Q^u - q^u_{i,w}(t)) \leq \max(Q^u,U) \times |W|$$

Let $q^u_{i,w}(t)$ denote the deficit counter value for worker w of client i at time t. Since the deficit counter will only be refilled when $q^u_{i,w}(t) \leq 0$ (line 7) by Q^u, we have

$$q^u_{i,w}(t) \leq Q^u \tag{4}$$

Now we prove through induction:

$$q^u_{i,w}(t) > -U \tag{5}$$

- At the beginning, all $q^u_{i,w}(t) = 0$. Equation (5) holds.
- We then prove if at time t, Equation (5) holds, then for $t' > t$, Equation (5) also holds.
- At line 7, $q^u_{i,w}(t') = q^u_{i,w}(t) + Q^u > q^u_{i,w}(t) > -U$. Equation (5) holds.
- Since line 24 will be reached only when $q^u_{i,w}(t) > 0$, $q^u_{i,w}(t') = q^u_{i,w}(t) - w_e \cdot L_{input} > -w_e \cdot L_{input}$. Equation (5) holds.
- At line 23, since $q^u_{i,w}(t') = q^u_{i,w}(t) - w_q \cdot |\{r|client(r) = i, r \in B\}|$ will be repeated for n steps until some requests are finished. Therefore, we have $q^u_{i,w}(t') = q^u_{i,w}(t) - n \cdot w_q \cdot |\{r|client(r) = i, r \in B\}|$. Since the number of decoded tokens cannot exceed the server's maximum token capacity M, $n \cdot |\{r|client(r) = i, r \in B\}| \leq M$. We then have $q^u_{i,w} = q^u_{i,w}(t) - w_q \cdot M > -w_e \cdot L_{input} - w_q \cdot M$. Equation (5) holds.

We have $W_i(t_1,t_2) = \sum_{w \in W} (k_{i,w} \cdot Q^u - q^u_{i,w}(t_2))$. Combining Equation (5) and Equation (4), we have:

$$|\sum_{w \in W} k_{i,w} \cdot Q^u - W_i(t_1,t_2)| \leq \max(Q^u,U) \times |W| \tag{6}$$

□

Theorem 5.1 (Service bound between backlogged clients). *At any time interval $[t_1,t_2)$, $\max_i W_i(t_1,t_2) - \min_i W_i(t_1,t_2) \leq 2\cdot|W|\cdot(U+Q^u)$. The difference between the maximum service among all backlogged clients and the minimum service among all backlogged clients is bounded by $2\cdot|W|\cdot(U+Q^u)$, where $|W|$ is the number of workers.*

Proof. • From Theorem 4.1, the service bound for each worker is: $2\cdot(U+Q^u)$.
- Since if a client is backlogged, it will have a request and hence be backlogged in all workers. This is because from Line 3, requests will be distributed to all workers and credit for each worker is exhausted, before replenishing the credits for all workers.
- Therefore, the service bound for DoubleQ is $2\cdot|W|(U+Q^u)$.

□

Theorem 5.2 (Service bound between backlogged and non-backlogged clients). *Consider any execution of the DoubleQ scheme. Client f that is continuously backlogged during time interval $[t_1,t_2)$ should not receive less service than another client, g, that is not continuously backlogged during the same time interval, where $W_g(t_1,t_2) - W_f(t_1,t_2) \leq 2\cdot(U+Q^u)\cdot|W|$.*

Proof. • f is continuously backlogged in all workers.
- g is not continuously backlogged in at least one worker.
- From Lemma 4.2, the service bound is $|W|\cdot(2U+2Q^u)$ between backlogged and either backlogged or non-backlogged clients.

□

Theorem A.4 (Latency Bound). *Let $A(r)$ and $D(r)$ denote the arrival time and dispatch time of a request r. Assume there are in total n clients, $\forall t_1,t_2$, if at t_1, a client f is not backlogged and has no requests in the running batch, then the next request r_f with $t_1 < A(r_f) < t_2$ will have its response time bounded: $D(r_f) - A(r_f) \leq (n-1)|W|\cdot\frac{2U+2Q^u}{a}$, where a is the lower bound of the server capacity.*

Proof. • Since there is no running batch of f in the system, r_f will be selected for the next request for f.
- Earlier, we have shown that the bound between a backlogged client and a non-backlogged client in DoubleQ to be $\max_i W_i - \min_i W_i \leq (2U+2Q^u)|W|$.
- Therefore the maximum number of tokens served before f is served again is: $(n-1)\cdot(2U+2Q^u)|W|$, where $n-1$ is the $n-1$ other client.
- Given that a is the lower bound of the server capacity, the dispatch time for f is therefore bounded: $D(r_f) - A(r_f) \leq (n-1)|W|\cdot\frac{2U+2Q^u}{a}$.

□

Theorem A.5 (Infinite Q_w is not fair). *Consider any execution of the DoubleQ Scheduling scheme in which client Q_w is infinite. Such scheduling scheme is not fair.*

Proof. • The requests will not be perfectly load balanced to all workers.
- Proof by counterexample: client f sends requests with large prefix matching. Requests from that client will be sent to the same worker hosting the prefix.
- Another client g sends requests with zero prefix matching, the requests will be load-balanced to all workers because of Line 8.
- Client g will be able to get unboundedly more service compared to f as it is replenished more, due to being scheduled to more workers, despite both being backlogged.

□

A.3 Per-Client Round-Robin Can Achieve Fairness.

Theorem A.6 (Service between backlogged or non-backlogged clients is unbounded). *In any interval $[t_1,t_2)$, The difference between the maximum service among all backlogged clients and the minimum service among all backlogged or non-backlogged clients is bounded by a constant independent of the time interval $t_2 - t_1$.*

Proof. • The client requests will be load-balanced to all workers.
- Therefore, when a client is backlogged, it is backlogged on all workers.
- We can apply the bound derived in DoubleQ, multiplied by the number of workers $|W|$, similar to §A.2.

□