



Optimizing resource allocation for geographically-distributed inference by large language models^{☆,☆☆}

Tingyang Sun ^a, Ting He ^{a,*}, Bo Ji ^b, Parimal Parag ^c

^a Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA

^b Department of Computer Science, Virginia Tech, Blacksburg, VA, USA

^c Department of Electrical Communication Engineering, Indian Institute of Science, Bangalore, India

ARTICLE INFO

Keywords:

Large language model
Model parallelism
Block placement
Request routing

ABSTRACT

Large language models (LLMs) have demonstrated extraordinary performance in many artificial intelligence (AI) tasks but are expensive to use, even after training, due to their requirement of high-end GPUs. Recently, a distributed system called PETALS was developed to lower the barrier for deploying LLMs by splitting the model blocks across multiple servers with low-end GPUs distributed over the Internet, which was much faster than swapping the model parameters between the GPU memory and other cheaper but slower local storage media. However, the performance of such a distributed system critically depends on the resource allocation, and how to do so optimally remains unknown. In this work, we present the first systematic study of the resource allocation problem in distributed LLM inference, with focus on two important decisions: block placement and request routing. Our main results include: (i) experimentally validated performance models that can predict the inference performance under given block placement and request routing decisions, (ii) a formulation of the offline optimization of block placement and request routing as a mixed integer linear programming (MILP) problem together with the NP-hardness proof and a polynomial-complexity algorithm with guaranteed performance, and (iii) an adaptation of the offline algorithm for the online setting with the same performance guarantee under bounded load. Through both experiments and experimentally-validated simulations, we have verified that the proposed solution can substantially reduce the inference time compared to the state-of-the-art solution in diverse settings with geographically-distributed servers. As a byproduct, we have also developed a light-weighted CPU-only simulator capable of predicting the performance of distributed LLM inference on GPU servers, which can evaluate large deployments and facilitate future research for researchers with limited GPU access.

1. Introduction

Large language models (LLMs) have yielded exceptional performance in many artificial intelligence (AI) tasks. Modern LLMs pre-trained on large datasets have demonstrated promising utility for diverse applications [1]. However, to achieve state-of-the-art

[☆] This article is part of a Special issue entitled: 'IFIP PERFORMANCE 2025' published in Performance Evaluation.

^{☆☆} This work was partly supported by the National Science Foundation, United States under award CNS-2106294. Parimal was supported by Qualcomm Inc. under Qualcomm University Relations 6G India, Anusandhan National Research Foundation (ANRF) under Grant CRG/2023/008854, and Office of International Relations at IISc under IISc-PSU collaboration research grant.

* Corresponding author.

E-mail address: tinghe@psu.edu (T. He).

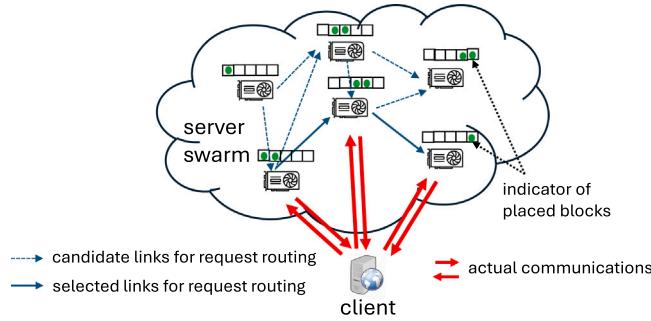


Fig. 1. System architecture for pipeline-parallel LLM inference using client-centric communication.

accuracy, such models often need to contain over 50 billion (50B) parameters, which makes it expensive to work with these models due to the requirement of expensive hardware such as high-end GPUs.

To broaden the accessibility to LLMs, several research groups have open-sourced their pre-trained LLMs [2–5]. However, the sheer size of these models makes it challenging to adopt them, even if no training is needed. Prior studies have shown that the bottleneck of running LLMs is not in the computation speed, as even a consumer-grade GPU like GeForce RTX 3070 has enough processing power to run a complete inference step of a 176B-parameter model within a second [6]. Instead, the bottleneck is in the GPU memory. A 100B-parameter model will require 200 GB of GPU memory to load the model parameters at the standard half precision, and many LLMs are even larger (e.g., GPT-3 has 175B parameters and GPT-4 has over 1.7 trillion parameters). Providing this much GPU memory is financially challenging for many users. Straightforward approaches to address this bottleneck such as compressing the model or offloading the model parameters to cheaper storage media will introduce undesirable side effects. For example, a compressed LLM can still be too large for consumer-grade GPUs and too much compression will lower the inference accuracy [7], while offloading the model parameters to larger but slower storage media like RAM or SSD will be very slow due to the limited bandwidth between such storage media and the GPU [8].

Recently, a model-parallel approach has been proposed to address the above challenge through resource pooling across distributed devices. This approach distributes a model to multiple devices at the granularity of transformer *blocks* (a.k.a. *pipeline parallelism* [9–11]) or neurons (a.k.a. *tensor parallelism* [12–14]) to run large models on devices with small GPUs. In particular, [8] has shown that pipeline parallelism can be used to run LLM inference tasks over *geographically-distributed* servers, each with only a few GB of GPU memory, at a much faster rate than local parameter offloading. This is achieved by letting each server host a subset of consecutive blocks, and each inference *request* (for autoregressive sequence generation) routed through a *chain of servers* that collectively host the entire model, as illustrated in Fig. 1. To reduce the communication cost, caching is used pervasively in such systems. For example, in a state-of-the-art system called PETALS [8], the client of each request uses client-side caches to store the input history for each invoked server, so that once a server fails, the client can use the cached input to bring up a backup server without repeating the processing at other servers; meanwhile, each server uses server-side caches (a.k.a. *attention caches*) to store the past computation results for each ongoing request (i.e., the key-value pairs for the past tokens [15]), so that at each inference step (after the prefill phase), the client only needs to send the partially processed embedding of a single token to the next server and receive the processed embedding back, which only exchanges a few KB of data [8]. The client-side caching implies a hub-spoke communication pattern as shown in Fig. 1, where all the communications during an inference session are anchored at the client, who will forward the partially processed embeddings between consecutive servers in order to update the input cache for each server.

In this work, we consider LLM inference over geographically-distributed servers using *pipeline parallelism* and *client-centric communication* as illustrated in Fig. 1. Although the feasibility of such systems has been validated in [8], there is a lack of fundamental understanding of how to optimally manage their performance. In particular, the current solution in [8] relies on heuristics for resource allocation, e.g., selecting the blocks for each server based on a heuristic “throughput” metric and selecting the server chain for each request based on a graph with heuristic “edge weights” [16]. It remains open how to optimize the performance of such systems, while taking into account the unique characteristics of GPUs and LLM inference tasks. This work aims at filling this gap by rigorously formulating and tackling the *block placement and request routing* problem for a distributed LLM inference system with the architecture in Fig. 1, using PETALS [8] as a concrete example. While our system model and evaluations are based on PETALS, our approach is applicable to any pipeline-parallel LLM inference systems as explained later.

1.1. Related work

Below we will briefly review existing approaches to overcome the GPU memory limitation in running large models and related resource allocation problems.

Model parallelism: Model parallelism overcomes the GPU memory limitation by distributing the model parameters to multiple devices. It is further divided into *pipeline parallelism* that assigns model parameters at the granularity of layers [9–11], and *tensor parallelism* that assigns model parameters at the granularity of neurons [12–14]. Tensor parallelism provides more flexibility in splitting the model at the cost of a higher communication overhead due to the all-to-all communication between devices hosting

adjacent layers, which makes it more suitable for distributed inference within a single multi-GPU server or a cluster of highly connected servers [17]. In contrast, pipeline parallelism only requires communication between pairs of devices at some loss of flexibility, which makes it more suitable for distributed inference across nodes [8]. PETALS [8] uses pipeline parallelism. Other than PETALS, there are a few more systems that support distributed LLM inference across nodes with different system characteristics. For example, vLLM with Ray Serve [18] uses tensor parallelism within each node and pipeline parallelism across nodes by letting servers processing adjacent blocks directly communicate partially processed tokens, Nvidia Dynamo [19] implements several LLM-specific execution capabilities such as disaggregating prefill&decoding phases and offloading attention caches to memory hierarchies, and Amazon EKS [20] simplifies the execution of multi-node inference workloads through integration with AWS services like Amazon EFS and Elastic Fabric Adapter. These systems are mainly designed for datacenter environments with high-bandwidth low-latency connections. In contrast, PETALS [8] is the leading system for LLM inference in geographically-distributed environments. In this work, we focus on geographically-distributed LLM inference and thus build our system model after PETALS, but our solution is generally applicable to any pipeline-parallel LLM inference systems, possibly with minor adaptations (e.g., see Remark after (1)). While the extension of our solution to tensor parallelism is nontrivial, we note that tensor parallelism is generally considered impractical for geographically-distributed settings.

Parameter offloading: Parameter offloading overcomes the GPU memory limitation by swapping model parameters between the GPU memory and a slower but larger storage, typically RAM or SSD [21–23]. When using the model for inference, model parameters are loaded into the GPU just before being used, which in theory allows a server with a small GPU to process any large models that fit into its storage. The drawback of parameter offloading is the overhead in loading and unloading the model parameters, which can take a substantial amount of time (e.g., at least 11 s per token for a 175B-parameter model [8]). For large models (with 50B+ parameters), parameter offloading was shown to be at least an order of magnitude slower than model parallelism [8].

Service function chaining: Under pipeline parallelism, each inference request is served by a chain of servers as illustrated in Fig. 1, which makes the corresponding resource allocation problem (e.g., how to place blocks and how to route requests) analogous to the problem of *service function (SF) placement* and *flow routing* in the context of *service function chaining (SFC)* [24]. SFC provides application-specific services for in-transit traffic through a chain of SFs. SFC placement and routing has been actively studied in recent years, typically with an objective of minimizing operation cost, maximizing flow rate, optimizing Quality of Service (QoS), or optimizing a combination of these metrics [25]. The problems are usually NP-hard and solved by polynomial-time heuristics without guaranteed performance [25,26], except for a few special cases such as [27–29] with approximation guarantee. Despite the conceptual similarity, our problem is fundamentally different from SFC placement and routing. *First of all*, the two domains have very different properties, e.g., different SFs may have heterogeneous resource requirements [26,27,30] and some SFs may change the flow rate [31] or branch an input flow into multiple output flows [30], while each LLM typically has identically structured transformer blocks that require the same amount of resource and input data size. *Moreover*, the bottleneck resource in LLM inference is GPU memory, which differs from the bottleneck resource in SFC (i.e., CPU cycles) in that: (i) the need of GPU memory is not elastic, making congestion minimization approaches like [29] inapplicable, (ii) the GPU memory used to host a block is shared by all the requests processed by the same block, differing from the additive resource consumption for CPU cycles [27,28], and (iii) nontrivial amounts of GPU memory are needed for both hosting blocks and serving inference sessions (due to the need of holding attention caches), causing resource contention between the placement decision and the routing decision that did not exist in service function chaining or other traditional “placement and routing” problems. Additionally, the number of blocks in an LLM can be much larger than the number of SFs in a typical SFC (e.g., BLOOM-176B has 70 blocks and GPT-3 has 96 blocks), making the formulations for SFC that enumerate the processing units [26–29] highly inefficient. Some LLM inference systems also have unique characteristics such as the hub-spoke communication pattern shown in Fig. 1. *There is thus a need of studying resource allocation problems tailored to distributed LLM inference systems.*

1.2. Summary of contributions

We perform a systematic study of the resource allocation problem in distributed LLM inference, with the following contributions:

1. We formulate a joint optimization of the placement of model blocks at servers (called *block placement*) and the selection of server chains for inference requests (called *request routing*) based on performance models experimentally validated on a real distributed LLM inference system called PETALS [8]. Our formulation features a compact representation of the decision variables that allows the problem to be formulated as a mixed integer linear programming (MILP) problem with a polynomial number of variables/constraints.
2. We prove the NP-hardness of the formulated optimization problem, and develop a three-step algorithm (Alg. 1) by decomposing the joint optimization into three subproblems, each optimizing one type of decision variables, which achieves a polynomial complexity and a guaranteed average inference time per token.
3. We then consider the online setting with dynamically arriving requests, for which we show that the offline algorithm can be adapted into a two-time-scale solution that solves the block placement problem via robust optimization and the (online) request routing problem via a variation of shortest-path routing, with link costs designed to approximate the total inference time (including waiting, communication, and computation) incurred at each server. The online algorithm inherits the same performance guarantee as the offline algorithm as long as the number of concurrent requests is within a design limit, which can be tuned to trade off the waiting time and the per-token inference time after waiting.

4. We compare our algorithms against the state-of-the-art algorithms in [8] through both controlled experiments and experimentally-validated simulations in a variety of settings with geographically distributed servers. The results not only show substantial performance improvements by our algorithms (60–80% smaller inference times), but also provide insights on the reasons for such improvements as well as the potential room for further improvements. A byproduct of our evaluation is a CPU-only simulator that is validated to produce reasonable predictions of the actual performance of distributed LLM inference on GPU servers, which can enable the evaluation of large deployments and facilitate future research for researchers with limited GPU access.

Roadmap. Section 2 introduces the problem, Section 3 presents the optimization formulation and the proposed algorithms together with their analysis, Section 4 presents the evaluation results, and Section 5 concludes the paper. The response to previous reviews, proofs, and other supporting materials are provided in Appendix A.

2. Problem formulation

2.1. Target application scenario

We consider geographically-distributed inference by a pre-trained LLM with a decoder-only architecture [32], which is commonly adopted by state-of-the-art LLMs (e.g., GPT and its variants [1,32,33]). The LLM is composed of thin input/output layers to map each input token to an embedding and each output embedding to a probability distribution of the next token, as well as multiple layers of identically structured *transformer blocks* (hereafter referred to as *blocks*), each containing a multi-headed self-attention sub-layer, a fully connected feed-forward sub-layer, and corresponding layer normalizations [32]. The input/output layers only contain a small percentage of the model parameters (e.g., < 3% for BLOOM-176B [8]) and can thus be hosted by the client. The blocks contain most of the model parameters and are thus delegated to the servers [8]. Given a pre-trained LLM with L blocks, we consider a distributed system with client-side and server-side caching like PETALS [8], which uses this LLM to serve autoregressive sequence generation requests (hereafter referred to as *requests*) via a hub-spoke communication pattern as shown in Fig. 1. In this work, we focus on supporting *short-prompt long-response queries*, where users ask concise questions to obtain detailed and comprehensive answers. This is a common type of queries, particularly in educational and technical contexts, and adheres to the best practices in using LLMs [34,35].

2.2. System model

Suppose that the system contains a set of servers V_s and a set of clients V_c , communicating through TCP connections according to the architecture shown in Fig. 1. Each client represents an ingress point for inference requests, which can be a host owned by end users or a proxy server submitting requests on behalf of end users. While it is possible for a server to directly forward its output to the next server to continue processing, PETALS [8] lets the client perform such forwarding as in Fig. 1, in order to maintain client-side caches of the input history for each invoked server to achieve fault tolerance.

Request model: For each inference request, the client initiates an *inference session*. Due to the one-one correspondence between inference requests and inference sessions, hereafter we will use “request” and “session” interchangeably. We will first consider the offline case with a given set of requests to understand the structure of the problem, and then address the online case with sequentially arriving requests. To formulate the offline case, we use \mathcal{R} to denote the total set of inference requests and \mathcal{R}_c to denote the set of requests from client $c \in V_c$. Each request has up to l_{\max}^I input tokens and generates up to l_{\max}^I output tokens,¹ where l_{\max}^I and l_{\max}^I are system parameters with $l_{\max}^I + l_{\max}^I$ upper-bounded by the maximum sequence length supported by the LLM. For short-prompt long-response queries, we have $l_{\max}^I \ll l_{\max}^I$. To formulate the online case, we need to additionally model the state of each server as detailed in Section 3.3.2.

Inference time model: Each server $j \in V_s$ has a *per-block processing time* of up to $\tau_j^I(l_{\max}^I)$ during the prefill phase (i.e., phase for processing the input to generate the first token) and a *per-token-per-block processing time* of τ_j^I during the decoding phase (i.e., phase for autoregressively generating the remaining tokens). Our experiments have validated $\tau_j^I(l_{\max}^I)$ to be independent of the output length l_{\max}^I and τ_j^I to be independent of both the input length l_{\max}^I and the output length l_{\max}^I (see Figs. B.11–B.12). Moreover, the connection between each client c and each server j has a *per-input round trip time (RTT)* of up to $t_{cj}^I(l_{\max}^I)$, representing the communication time (including serialization/deserialization, propagation, queueing, and transmission delays) for transmitting one input sequence from the client to the server and the processed input sequence from the server back to the client. Similarly, the connection also has a *per-token RTT* of t_{cj}^I , representing the communication time for transmitting one token from the client to the server and the processed token from the server back to the client. This means that if a request from client c with input length l_{\max}^I and output length l_{\max}^I is processed by a chain of servers p with each server $j \in p$ processing k_j blocks, it will incur a *total inference time* (including all the client–server communication time and all the server processing time) of

$$\sum_{j \in p} \left(t_{cj}^I(l_{\max}^I) + k_j \tau_j^I(l_{\max}^I) \right) + (l_{\max}^I - 1) \sum_{j \in p} (t_{cj}^I + k_j \tau_j^I), \quad (1)$$

¹ The actual number of generated tokens may be smaller if the end-of-sequence token is detected, but the maximum number of output tokens specified in the request is used in resource allocation.

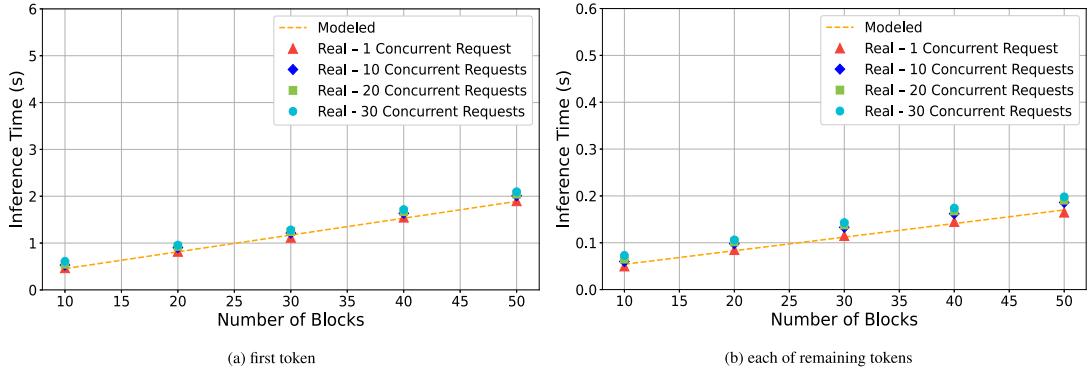


Fig. 2. Inference time vs. #processed blocks on A100 for: (a) first token; (b) each of remaining tokens ($l_{\max}^I = 20$, $l_{\max} = 128$).

where $\sum_{j \in p} (t_{cj}^I(l_{\max}^I) + k_j \tau_j^I(l_{\max}^I))$ is the time to query the servers for the first token, and $\sum_{j \in p} (t_{cj} + k_j \tau_j)$ is the time to query the servers for each of the remaining tokens. In the case of exceptions (e.g., out-of-memory error), additional delays will apply; see Section 3.3.2 for details. In addition, each session incurs some delays at the client due to local computations such as tokenizing the input sequence and extracting the next token from each embedding processed by the servers, but since these delays are not affected by the resource allocation at servers (see Section 2.3), they only contribute a constant shift which is ignored in our model.

Remark: The above model assumes client-centric communication as in Fig. 1. If adjacently traversed servers directly communicate as in [18], the communication times will become $\sum_{(i,j) \in p} t_{ij}^I(l_{\max}^I)$ for the first token and $\sum_{(i,j) \in p} t_{ij}$ for each of the subsequent tokens, where $t_{ij}^I(l_{\max}^I)/t_{ij}$ is the per-input/per-token one-way delay from node i to node j . Our optimization formulation and solution are easily amendable to allow direct server-server communications by simply replacing the client-server RTT t_{cj} with the server-server one-way delay t_{ij} .

Memory consumption model: Each server $j \in V_s$ has a GPU memory of M_j , which denotes the effective memory capacity for storing model parameters and attention caches (see Remark after (2)). The size of each block is s_m (bytes), given by the number of model parameters per block times the number of bytes per parameter. In addition, each server holds an attention cache of the past key-value pairs for each ongoing request routed through the server and each block it processes for that request. For a total sequence length of $l_{\max}^I + l_{\max}$, each attention cache stores two tensors (one for the keys and one for the values), each containing $d_{\text{model}} \cdot (l_{\max}^I + l_{\max})$ parameters, where d_{model} is the embedding dimension of the LLM. Thus, the size of each attention cache is $s_c := 2d_{\text{model}} \cdot (l_{\max}^I + l_{\max}) \cdot \text{dtype_bytes}$ (bytes), where ‘dtype_bytes’ is the number of bytes per parameter in the cache (usually $\text{dtype_bytes} = 2$). This means that a server j storing m_j blocks and processing k_j^r blocks for each request r has a total memory consumption of

$$s_m m_j + s_c \sum_{r \in R} k_j^r. \quad (2)$$

Remark: In addition to storing the model parameters for the hosted blocks and the attention caches, the GPU memory also needs to hold the CUDA context (depending on the CUDA version and the device) as well as a small number of intermediate variables generated during inference (assuming in-place updating), but this space is invariant to the number of placed blocks and the number of hosted inference sessions, and can thus be modeled as a constant overhead in the memory consumption. In practice, there is also some waste of GPU memory due to memory fragmentation. Thus, the memory capacity M_j is the effective memory for resource allocation and should be slightly smaller than the physical memory capacity to avoid the out-of-memory errors.

Experimental validation: We have validated the above models through various experiments based on PETALS [8] and BLOOM-176B [3]. As an example, Fig. 2 compares the per-token inference time incurred at a given server according to our model in (1) with the average time measured from 10 Monte Carlo runs of controlled experiments with various #processed blocks and #concurrent requests, in a basic setting of co-located server and client (where the communication time is just the time for serializing and deserializing tokens), and similar results hold under other settings with network delays. The results validate the accuracy of our proposed model in capturing the linear dependency of the inference time on the number of processed blocks. The results also indicate that the inference time for one request is largely independent of the number of concurrent requests at the same server (as long as they all fit into the GPU memory), thanks to the massive parallel computing capability of GPU. Figs. B.11 and B.12 in Appendix B.1 provide similar comparisons with respect to the input/output length, respectively, again validating the accuracy of our inference time model. Fig. 3a compares the GPU memory consumption predicted by (2) with the actual allocated GPU memory, which validates the accuracy of (2). A closer examination reveals that the first term in (2) (memory used to store blocks) is much larger than the second term (memory used to store attention caches), and only the second term depends on the number of concurrent requests as shown in Fig. 3b. Nevertheless, it is crucial to model the impact of concurrent requests on memory consumption, as the GPU memory is typically the bottleneck resource in LLM inference (see Remark in Section 2.3).

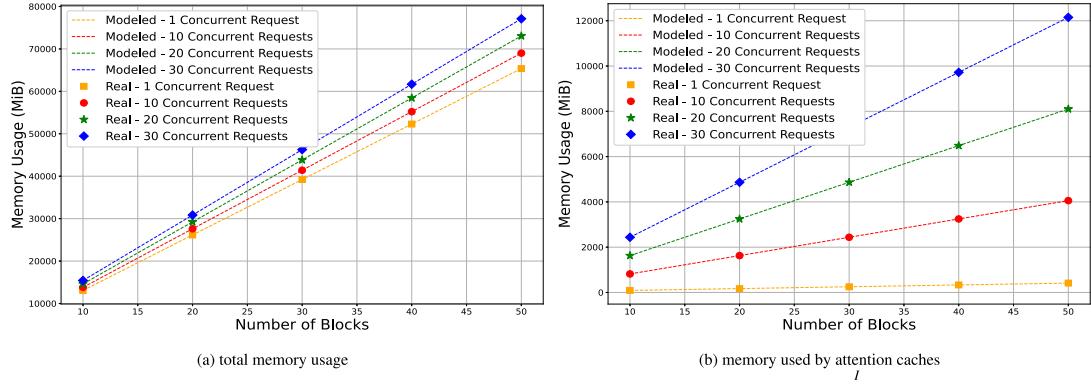


Fig. 3. Memory consumption vs. #blocks on A100 for: (a) total memory usage; (b) attention caches ($l_{\max}^I = 20$, $l_{\max} = 128$) .

Table 1

Main notations (first two rows: free decision variables; rest: input parameters or dependent variables).

Notation	Description
a_j, m_j	Index of first block & #blocks placed on server j
f_{ij}^r, f_p^r	Indicator for request r to be routed on link (i, j) or path p
L	Total #blocks in the LLM under consideration
V_c, V_s	Set of clients/servers
(V, E)	Logical topology for joint block placement and request routing
$(V^c, E_{a,m}^c)$	Logical topology for request routing for client c under block placement (a, m) (see Alg. 1)
$\mathcal{R}, \mathcal{R}_c$	Set of all the requests or requests from client c
l_{\max}^I, l_{\max}	Maximum #tokens in an input/output sequence
s_m, s_c	Size per block and size per attention cache
M_j, \bar{f}_j	Total memory and maximum #parallel sessions at server j
$\tau_j^I(l_{\max}^I), \tau_j$	Per-block processing time at server j during prefill or decoding phase
$t_{c,j}^I(l_{\max}^I), t_{c,j}$	Per-input/per-token RTT between client c and server j
$t_{i,j}^I(l_{\max}^I), t_{i,j}^c$	Per-token inference time of a first/later token at server j for a request from client c routed through (i, j)
\tilde{t}_j, t_{\max}	Amortized inference time and maximum per-token RTT for server j (see (14))
C_b, T_b	Total capacity (in terms of #sessions) and total amortized inference time for block b (see Alg. 1)
$(T_j^I(t), M_j^I(t))_{r=1}^{R_j(t)}$	State of server j at time t (see Section 3.3.2)
$t_{ij}^W(t)$	Waiting time for link (i, j) at time t (see (20))

2.3. Resource allocation problem

We study the joint optimization of how to place the blocks at the servers, referred to as “*block placement*”, and how to select the chain of servers for each request, referred to as “*request routing*”. Our objective is to minimize the average time in serving each request within the resource constraints, with focus on the GPU memory constraint. Table 1 lists the main notations used in our presentation.

Remark 1: The objective of this work is to optimize the inference performance in terms of inference time for a given set of available GPU resources. Other performance measures, e.g., cost of renting GPUs or robustness in the face of unreliable nodes, are left for future work.

Remark 2: In theory, there are other resource constraints besides GPU memory that can limit how the inference requests can be served. For example, each server has a limited processing capacity, and each client–server connection has a limited throughput. In practice, however, GPU memory is usually the bottleneck resource that will be saturated before other resources. For example, for BLOOM-176B [3], a server with an A100 (80 GB) GPU and 100 Mbits/s bandwidth can process over 700 tokens/second when hosting the maximum number of blocks according to [8] (53 blocks) and transfer over 400 tokens/second, which should be enough for over 80 concurrent sessions without causing notable queueing delays, but the available GPU memory only allows 21 concurrent sessions for $l_{\max}^I = 20$ and $l_{\max} = 128$, and even fewer for longer sequences. We will thus focus on the GPU memory constraint.

3. Joint Block Placement and Request Routing (BPRR)

Based on the experimentally validated model (Section 2.2), we will first study BPRR in an offline setting to understand the properties of the problem (Section 3.2), and then use the understanding to develop a solution for the online setting (Section 3.3).

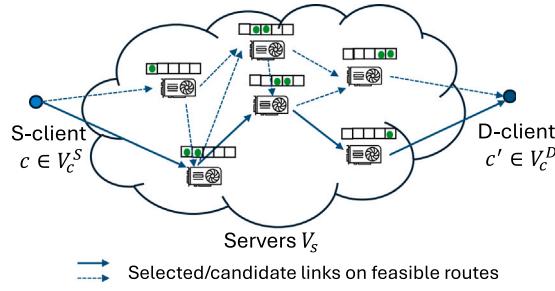


Fig. 4. Logical topology G for request routing and route feasibility condition.

3.1. Preliminaries

Logical topology for routing: To facilitate the optimization formulation, we construct a *logical topology* $G = (V, E)$ as a directed graph illustrated in Fig. 4, where the node set $V := V_c^S \cup V_s \cup V_c^D$ contains the servers V_s , the copies of clients V_c^S viewed as sources of requests (called *S-clients*), and the copies of clients V_c^D viewed as destinations of requests (called *D-clients*). We will use $c \in V_c^S$ and $c' \in V_c^D$ to denote the S-client and the D-client corresponding to a real client $c \in V_c$. Define $t_{cc'} := 0$ and $\tau_{c'} := 0$. The link set E includes the links connecting each S-client with the servers allowed to host the first block, the links between servers allowed to be adjacent on a chain, and the links connecting the servers allowed to host the last block to each D-client. Typically, this leads to full connectivity within V_s and complete bipartite connectivity between V_s and V_c^S/V_c^D . The split of clients into S-clients and D-clients will facilitate the formulation for request routing.

Route feasibility condition: It is easy to see that an optimal block placement should only place *consecutive blocks* at each server, as non-consecutive blocks will increase the number of communications between the client and the server that introduces unnecessary delay. This type of block placement implies that each server can be traversed at most once by each inference session, implying a simple routing path. A *path* p in G is *feasible* for routing the requests from client c if it leads from the corresponding S-client to the corresponding D-client and traverses a chain of servers that collectively host all the blocks of the LLM in order, as illustrated in Fig. 4.

Since each server should host consecutive blocks, we can represent the block placement at each server $j \in V_s$ by the *first block* a_j and the *number of blocks* m_j , i.e., server j stores the blocks in $\{a_j, \dots, a_j + m_j - 1\}$, where² $a_j, m_j \in [L]$ and $a_j + m_j - 1 \leq L$. It is possible for the same block to be placed at multiple servers traversed by an inference session. In this case, we assume that the first server hosting the block will process it according to [36]. That is, if a session traverses server i immediately before server j , the blocks processed at server j will be $\{\max(a_j, a_i + m_i), \dots, a_j + m_j - 1\}$. A chain of servers is feasible for request routing if and only if the block placement satisfies the following condition.

Lemma 3.1. *Assume that each node $j \in V$ stores all the blocks in $\{a_j, \dots, a_j + m_j - 1\}$, where each S-client $c \in V_c^S$ stores a dummy block 0 (i.e., $a_c := 0, m_c := 1$), and each D-client $c' \in V_c^D$ stores another dummy block $L + 1$ (i.e., $a_{c'} := L + 1, m_{c'} := 1$). Then a c -to- c' path p in G is a feasible routing path for client c if and only if*

$$a_j \leq a_i + m_i \leq a_j + m_j - 1, \quad \forall (i, j) \in p. \quad (3)$$

Remark: Intuitively, the condition (3) states that a path is feasible for request routing if and only if after processing all the blocks at the previous servers, the next block can always be found at the next server on the path.

Performance models: Lemma 3.1 allows us to explicitly model how the inference performance depends on block placement and request routing. Specifically, according to Section 2.2, Lemma 3.1 implies that a request from client c routed through a feasible path p traversing link (i, j) will incur a *per-token inference time* of

$$t_{ij}^c := t_{cj} + \tau_j(a_j + m_j - a_i - m_i) \quad (4)$$

at server j after the first token generation (including both computation and communication), where $a_j + m_j - a_i - m_i$ is the number of blocks that are processed for this request at server j . The first token will incur a larger per-token inference time of $t_{ij}^{c,I}(l_{\max}^I) := t_{cj}^I(l_{\max}^I) + \tau_j^I(l_{\max}^I)(a_j + m_j - a_i - m_i)$ at server j .

Meanwhile, by Lemma 3.1, each request routed through link $(i, j) \in E$ will require $s_c(a_j + m_j - a_i - m_i)$ of dedicated memory at server j for the attention caches. This together with the memory for storing m_j blocks leads to a total memory consumption of

$$s_m m_j + s_c \sum_{i:(i,j) \in E} f_{ij}(a_j + m_j - a_i - m_i) \quad (5)$$

at server j , where f_{ij} is the number of requests routed (concurrently) over link (i, j) .

² Throughout this work, we use $[k]$ for a positive integer k to denote the set $\{1, \dots, k\}$.

3.2. Offline setting

We first consider the offline setting with a given set of requests to understand the properties of the problem.

3.2.1. Vanilla formulation

As explained in Section 3.1, the block placement can be concisely encoded by $\mathbf{a} := (a_j)_{j \in V_s}$ and $\mathbf{m} := (m_j)_{j \in V_s}$, which jointly determine which paths are feasible for request routing according to Lemma 3.1. Let $P_c(\mathbf{a}, \mathbf{m})$ denote the set of feasible routing paths for client c under the block placement (\mathbf{a}, \mathbf{m}) . Due to the server-side attention caches, the chain of servers should remain the same for all the tokens of the same sequence. We ensure this by controlling routing at the granularity of requests, represented by $f_p^r \in \{0, 1\}$ that indicates the selection of path $p \in P_c(\mathbf{a}, \mathbf{m})$ for request $r \in \mathcal{R}_c$. Then we formulate the *joint block placement and request routing problem (BP RR)* as

$$\min_{f, a, m} \sum_{c \in V_c} \sum_{r \in \mathcal{R}_c} \sum_{p \in P_c(\mathbf{a}, \mathbf{m})} f_p^r \sum_{(i, j) \in p} t_{ij}^c \quad (6a)$$

$$\text{s.t. } s_m m_j + s_c \sum_{c \in V_c} \sum_{r \in \mathcal{R}_c} \sum_{\substack{p \in P_c(\mathbf{a}, \mathbf{m}): \\ (i, j) \in p, \exists i}} f_p^r (a_j + m_j - a_i - m_i) \leq M_j, \quad \forall j \in V_s, \quad (6b)$$

$$\sum_{p \in P_c(\mathbf{a}, \mathbf{m})} f_p^r = 1, \quad \forall c \in V_c, r \in \mathcal{R}_c, \quad (6c)$$

$$a_j + m_j - 1 \leq L, \quad \forall j \in V_s, \quad (6d)$$

$$f_p^r \in \{0, 1\}, \quad a_j, m_j \in [L], \quad (6e)$$

where the objective (6a) is to minimize the total (and hence the average) per-token inference time over all the requests when ignoring the first token, constraint (6b) ensures that the requests can be served within the GPU memory at each server (where $\sum_{c \in V_c} \sum_{r \in \mathcal{R}_c} \sum_{p \in P_c(\mathbf{a}, \mathbf{m}): j \in p} f_p^r$ is the number of requests routed to server j), constraint (6c) ensures that each request is routed to a feasible path, and constraints (6d)–(6e) ensure the feasibility of the block placement.

Remark: Strictly speaking, to minimize the average time in serving each request, we should minimize

$$\sum_{c \in V_c} \sum_{r \in \mathcal{R}_c} \sum_{p \in P_c(\mathbf{a}, \mathbf{m})} f_p^r \left(\sum_{(i, j) \in p} t_{ij}^{c, I} (l_{\max}^I) + (l_{\max}^I - 1) \sum_{(i, j) \in p} t_{ij}^c \right), \quad (7)$$

which is the total time to complete all the requests. Minimizing (7) is equivalent to minimizing an objective function of the form (6a), except that t_{ij}^c needs to be redefined as follows:

$$\left(\frac{1}{l_{\max}^I} t_{cj}^I (l_{\max}^I) + \frac{l_{\max}^I - 1}{l_{\max}^I} t_{cj} \right) + \left(\frac{1}{l_{\max}^I} \tau_j^I (l_{\max}^I) + \frac{l_{\max}^I - 1}{l_{\max}^I} \tau_j \right) (a_j + m_j - a_i - m_i), \quad (8)$$

which denotes the average per-token inference time incurred by a request from client c at link (i, j) over *all the tokens*. In the case of $l_{\max}^I \ll l_{\max}$, (8) reduces to (4). Even in the general case, all our results remain applicable when redefining t_{ij}^c as in (8). We will thus focus on solving (6). While we have used the maximum input/output length to simplify our formulation, it is easily extensible to the case of heterogeneous input/output lengths as explained in Appendix B.2.

3.2.2. MILP formulation

The vanilla formulation (6) is not computationally tractable because of the implicit and nonlinear dependency of the routing variable f on the block placement variables \mathbf{a} and \mathbf{m} . Moreover, given a block placement (\mathbf{a}, \mathbf{m}) , the number of feasible paths in $P_c(\mathbf{a}, \mathbf{m})$ can be exponential. These limitations motivate us to seek a more tractable formulation. Below we will show that by introducing appropriate auxiliary variables, we can convert the nonlinear exponential-sized optimization (6) into a MILP with polynomial numbers of variables and constraints.

To avoid an exponential number of routing variables, we replace the path-level routing variable f_p^r by a link-level routing variable $f_{ij}^r \in \{0, 1\}$, which indicates if request r is routed over link $(i, j) \in E$ (meaning that the request will be processed by server i and server j consecutively). This simplifies the objective function (6a) into

$$\sum_{c \in V_c} \sum_{r \in \mathcal{R}_c} \sum_{(i, j) \in E} f_{ij}^r (t_{cj} + \tau_j (a_j + m_j - a_i - m_i)). \quad (9)$$

We can guarantee all the requests to be properly routed by imposing a flow conservation constraint

$$\sum_{i \in V} f_{ji}^r = \sum_{i \in V} f_{ij}^r + d_j^c, \quad \forall c \in V_c, r \in \mathcal{R}_c, j \in V, \quad (10)$$

where d_j^c is a constant, defined as 1 if $j = c$ (S-client for client c), -1 if $j = c'$ (D-client for client c), and 0 otherwise. This ensures that each request will be routed from its S-client to its D-client in the logical topology G (Fig. 4).

To model the dependency between request routing and block placement, we leverage Lemma 3.1, which states that it is feasible to route a request from node i to node j if and only if $a_j \leq a_i + m_i \leq a_j + m_j - 1$. Thus, we can ensure route feasibility by requiring

$$a_j f_{ij}^r \leq a_i + m_i, \quad \forall r \in \mathcal{R}, (i, j) \in E, \quad (11)$$

$$(a_i + m_i)f_{ij}^r \leq a_j + m_j - 1, \quad \forall r \in \mathcal{R}, (i, j) \in E, \quad (12)$$

where $\mathcal{R} := \bigcup_{c \in V_c} \mathcal{R}_c$ denotes the total set of requests. This ensures that a request is routed over (i, j) only if the condition in [Lemma 3.1](#) is satisfied.

However, using the above objective and constraints directly will lead to a nonlinear optimization due to the bilinear terms $a_j f_{ij}^r$, $a_i f_{ij}^r$, $m_j f_{ij}^r$, and $m_i f_{ij}^r$ in the objective function (9) and constraints like (11)–(12). Fortunately, because f_{ij}^r is binary, we can convert them into linear forms by introducing auxiliary variables $\alpha_{ij}^r, \beta_{ij}^r, \gamma_{ij}^r, \delta_{ij}^r$ and the corresponding linear constraints (B.4)–(B.7) as detailed in [Appendix B.3](#).

Combining all the above allows us to rewrite the BPRR problem in (6) as follows:

$$\min_{f, a, m} \sum_{c \in V_c} \sum_{r \in \mathcal{R}_c} \sum_{(i, j) \in E} \left(t_{cj} f_{ij}^r + \tau_j (\alpha_{ij}^r + \gamma_{ij}^r - \beta_{ij}^r - \delta_{ij}^r) \right) \quad (13a)$$

$$\text{s.t. } s_m m_j + s_c \sum_{c \in V_c} \sum_{r \in \mathcal{R}_c} \sum_{(i, j) \in E} (\alpha_{ij}^r + \gamma_{ij}^r - \beta_{ij}^r - \delta_{ij}^r) \leq M_j, \quad \forall j \in V_s, \quad (13b)$$

$$\sum_{i \in V} f_{ji}^r = \sum_{i \in V} f_{ij}^r + d_j^c, \quad \forall c \in V_c, r \in \mathcal{R}_c, j \in V, \quad (13c)$$

$$a_j + m_j - 1 \leq L, \quad \forall j \in V_s, \quad (13d)$$

$$\alpha_{ij}^r \leq a_i + m_i, \quad \forall r \in \mathcal{R}, (i, j) \in E, \quad (13e)$$

$$\beta_{ij}^r + \delta_{ij}^r \leq a_j + m_j - 1, \quad \forall r \in \mathcal{R}, (i, j) \in E, \quad (13f)$$

$$\text{(B.4)–(B.7),} \quad (13g)$$

$$f_{ij}^r \in \{0, 1\}, \quad a_j, m_j \in [L], \quad \alpha_{ij}^r, \beta_{ij}^r, \gamma_{ij}^r, \delta_{ij}^r \geq 0, \quad (13h)$$

where (13a) is the linearized expression of the total inference time, (13b) models the GPU memory capacity, (13c) ensures flow conservation, (13d) ensures block placement feasibility, (13e)–(13f) ensure route feasibility, and (13g) defines the auxiliary variables used to linearize the objective function and the constraints. As explained in [Appendix B.3](#), $\alpha_{ij}^r = a_j f_{ij}^r$, $\beta_{ij}^r = a_i f_{ij}^r$, $\gamma_{ij}^r = m_j f_{ij}^r$, and $\delta_{ij}^r = m_i f_{ij}^r$. Plugging these into (13a) and (13b) recovers the original objective (6a) and memory constraint (6b), except that the path-level routing variable f_p^r is replaced by the link-level routing variable f_{ij}^r . The flow conservation constraint (13c) together with the route feasibility constraints (13e)–(13f) and the integer constraint on f_{ij}^r (13h) ensures that each request is routed on a single feasible path as in (6c). The overall optimization is a MILP, where a, m (block placement) and f (request routing) are free variables, and the others are dependent variables.

Remark: The MILP (13) has $O(|\mathcal{R}| \cdot |E|)$ variables, dominated by the routing variable f and the auxiliary variables α, β, γ , and δ . As the number of links $|E|$ is bounded by $O(|V_s|(|V_c| + |V_s|))$, the number of variables is in $O(|\mathcal{R}| \cdot |V_s|(|V_c| + |V_s|))$. Similarly, the number of constraints is also in $O(|\mathcal{R}| \cdot |E|) = O(|\mathcal{R}| \cdot |V_s|(|V_c| + |V_s|))$, dominated by (13e)–(13g). Thus, even if BPRR can be formulated as a MILP, the size of the MILP will grow linearly in the number of requests/clients and quadratically in the number of servers, making it challenging to solve.

Formally, we have shown via a reduction from the optimization version of the *partition problem* [37] that the BPRR problem is hard to solve to optimality as stated below.

Theorem 3.2. *The BPRR problem as formulated in (6) or (13) is NP-hard.*

Remark: In fact, the reduction in the proof of [Theorem 3.2](#) leads to a special case of BPRR with a single client ($|V_c| = 1$). This case represents a common deployment scenario where end users query the system through a proxy (e.g., a Flask web server as in [36]). We have proved that the BPRR problem remains NP-hard even in this special case.

3.2.3. Algorithm design

The NP-hardness of BPRR implies the need of efficient suboptimal algorithms. Although the MILP formulation in (13) allows us to apply existing heuristics for MILP, e.g., LP relaxation plus rounding, such a heuristic still has limited scalability due to the large size of the LP, and more importantly, it does not even guarantee a feasible solution.³ To efficiently solve large instances of the BPRR problem, we propose a three-step algorithm called *Conservative Greedy BPRR* (CG-BPRR) as shown in [Alg. 1](#), by decomposing the BPRR problem into the three subproblems of optimizing m , a , and f sequentially, as explained below.

Step 1: First, we set the number of blocks per server conservatively (line 1) to make sure that each server will have enough remaining GPU memory to hold the attention caches even if all the requests are routed through it. This is because according to the memory consumption model in (5), the memory consumption at server j is upper-bounded by $s_m m_j + s_c |\mathcal{R}| m_j$ (achieved if all the requests are routed to it and all the hosted blocks are processed), and thus storing $\min(\lfloor M_j / (s_m + s_c |\mathcal{R}|) \rfloor, L)$ blocks on server j will guarantee the maximum memory consumption to be feasible.

³ Specifically, the fractional solution $(\tilde{f}, \tilde{a}, \tilde{m})$ to the LP relaxation of (13) may not lead to a feasible integer solution after rounding, as the rounded block placement (a, m) may not place each block on at least one server. This is because the auxiliary variables $\alpha, \beta, \gamma, \delta$ used to linearize the problem can only enforce feasibility when f is binary.

Algorithm 1: Conservative Greedy BPRR (CG-BPRR)

input : set of clients V_c , set of requests $\bigcup_{c \in V_c} \mathcal{R}_c$, #blocks L , size per block s_m , size per cache s_c , set of servers V_s , parameters for each $j \in V_s$ including GPU memory M_j , processing time τ_j , and per-token RTTs $(t_{cj})_{c \in V_c}$.

output: Block placement (a, m) and request routing f

// conservative assignment of #blocks per server:

- 1 $m_j \leftarrow \min(\lfloor M_j / (s_m + s_c |\mathcal{R}|) \rfloor, L)$, $\forall j \in V_s$, where $|\mathcal{R}| = \sum_{c \in V_c} |\mathcal{R}_c|$;
- // greedy block placement:
- 2 $C_b \leftarrow 0$, $T_b \leftarrow \tilde{t}_0 |\mathcal{R}|$, $\forall b \in [L]$;
- 3 **for** each server $j \in V_s$ in increasing order of \tilde{t}_j **do**
- 4 **if** $\exists b \in [L]$ with $C_b < |\mathcal{R}|$ **then**
- 5 $a_j \leftarrow \operatorname{argmax}_{a \in [L-m_j+1]}: C_b < |\mathcal{R}|, \exists b \in \{a, \dots, a+m_j-1\} \sum_{b'=a}^{a+m_j-1} T_{b'}$;
- 6 **else**
- 7 $a_j \leftarrow \operatorname{argmin}_{a \in [L-m_j+1]} (C_a, \dots, C_{a+m_j-1})$;
- 8 $T_b \leftarrow T_b - (\tilde{t}_0 - \tilde{t}_j) \min \left(\max(|\mathcal{R}| - C_b, 0), \bar{f}_j \right)$, $\forall b \in \{a_j, \dots, a_j + m_j - 1\}$;
- 9 $C_b \leftarrow C_b + \bar{f}_j$, $\forall b \in \{a_j, \dots, a_j + m_j - 1\}$;
- // shortest-path request routing:
- 10 **for** each client $c \in V_c$ **do**
- 11 $G_{a,m}^c \leftarrow$ the feasible routing topology for client c under block placement (a, m) , with a node/link set $(V^c, E_{a,m}^c)$ and a cost of t_{ij}^c for each $(i, j) \in E_{a,m}^c$;
- 12 $p_c \leftarrow$ shortest path from the S-client to the D-client in $G_{a,m}^c$;
- 13 $f_{ij}^r \leftarrow \mathbb{1}((i, j) \in p_c)$ ⁴, $\forall r \in \mathcal{R}_c$, $(i, j) \in E$;

Step 2: Next, we greedily place a set of continuous blocks $\{a_j, \dots, a_j + m_j - 1\}$ at each server j in the descending order of “server speeds” so that each placement improves the performance for the worst-performing blocks (lines 2–9). Specifically, we measure the speed of server j by the *amortized inference time* defined as

$$\tilde{t}_j := \frac{1}{m_j} \max_{c \in V_c} t_{cj} + \tau_j m_j = \tau_j + \frac{t_{*j}}{m_j}, \quad (14)$$

where $t_{*j} := \max_{c \in V_c} t_{cj}$ is the *maximum per-token RTT* between any client and server j . Since $\max_{c \in V_c} (t_{cj} + \tau_j m_j)$ is an upper bound on the per-token inference time incurred at server j , \tilde{t}_j denotes the amortized maximum inference time per block at server j , amortized over the blocks hosted by this server. The amortization allows us to compare the speeds of servers with different memory capacities.

Under such amortization, we consider a *relaxed request routing problem*, where each request is routed among servers on a block-by-block basis and incurs a per-block inference time \tilde{t}_j at server j . It is clear that the relaxed request routing must use up the capacity of a faster server (with a smaller amortized inference time) before going to a slower server for each block. Meanwhile, under m_j computed in Step 1, each server j can guarantee to process up to

$$\bar{f}_j := \left\lfloor \frac{M_j - s_m m_j}{s_c m_j} \right\rfloor \geq |\mathcal{R}| \quad (15)$$

requests concurrently without running out of memory,⁵ which defines a *capacity* of the server. It means that under the relaxed request routing, only the fastest server hosting block b will be used to process the block for all the requests.

Thus, to make the best use of fast servers, we place blocks on servers in the increasing order of \tilde{t}_j , where each server j receives the set of continuous blocks that “need service the most”. To measure this, we use a variable C_b to track the *total capacity of servers hosting block b*, and another variable T_b to track the *total amortized inference time all the requests spend on block b* under the optimal relaxed routing. That is, C_b is the sum of \bar{f}_j ’s over all the servers satisfying $a_j \leq b \leq a_j + m_j - 1$ (line 9), and $T_b = \tilde{t}_j |\mathcal{R}|$ if server j is the fastest server hosting block b . To ensure that T_b is well-defined before block b is placed on any server, we introduce a *dummy server 0* (assuming $0 \notin V_s$) with a large capacity $\bar{f}_0 := |\mathcal{R}|$ and a large amortized inference time $\tilde{t}_0 > \tilde{t}_j$ ($\forall j \in V_s$), which initially hosts all the blocks. Introducing this dummy server allows us to meaningfully initialize T_b before any real block placement (line 2), and its large inference time ensures that T_b (as updated in line 8) will be equal to the actual total amortized inference time for block b when $C_b \geq |\mathcal{R}|$. Our idea is to measure the “need of service” by the total amortized inference time, i.e., the next fastest server will receive the set of continuous blocks with the maximum $\sum_{b=a_j}^{a_j+m_j-1} T_b$. One subtle point in applying this idea is that since the relaxed request routing always uses the fastest server for each block, T_b will stop changing after block b is placed, which can cause subsequent servers to be assigned the same blocks over and over again. To avoid such waste of servers, we will only select the blocks according to $\sum_{b=a_j}^{a_j+m_j-1} T_b$ if this set contains at least one unserved block (line 5); otherwise, we will select the set of continuous blocks with the minimum capacities (line 7), where “argmin” selects the first value of a in the lexicographical order of sorted $(C_a, \dots, C_{a+m_j-1})$.

The rationale of Step 2 is that it minimizes the average inference time under the relaxed request routing.

⁴ We use $\mathbb{1}(\cdot)$ to denote the indicator function.

⁵ The actual number of requests server j runs concurrently may be larger when not all the m_j blocks are processed at this server for some requests.

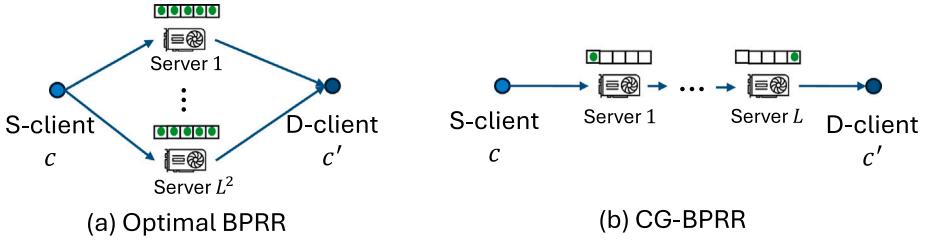


Fig. 5. Example for suboptimality of CG-BPERR (Alg. 1).

Lemma 3.3. *The block placement in lines 2–9 of Alg. 1 minimizes the average per-token inference time over all the requests under the relaxed request routing, assuming that “argmax” in line 5 breaks ties in favor of the smallest index.*

Remark: As shown in the proof of Lemma 3.3, Step 2 of CG-BPERR generates an intuitive block placement that uses the “fastest servers” to cover all the blocks sequentially. That is, if $j \in V_s$ denotes the j th fastest server in terms of $\tilde{\tau}_j$, then $a_k = \sum_{j=1}^{k-1} m_j + 1$ for all $k = 1, \dots, K-1$ and $a_K = L - m_K + 1$, where $K := \min\{k : \sum_{j=1}^k m_j \geq L\}$.

Step 3: Finally, we compute the request routing under the block placement (\mathbf{a}, \mathbf{m}) given by the previous steps, assuming the block placement to be feasible (i.e., every block is hosted by at least one server).

Given a feasible block placement (\mathbf{a}, \mathbf{m}) , the (conditionally) optimal request routing that minimizes the average inference time is given by a subproblem of (13) as follows

$$\min_f \sum_{c \in V_c} \sum_{r \in \mathcal{R}_c} \sum_{(i,j) \in E} t_{ij}^c f_{ij}^r \quad (16a)$$

$$\text{s.t. } s_c \sum_{c \in V_c} \sum_{r \in \mathcal{R}_c} \sum_{i: (i,j) \in E} f_{ij}^r (a_j + m_j - a_i - m_i) \leq M_j - s_m m_j, \quad \forall j \in V_s, \quad (16b)$$

$$(13c), (11)–(12), \quad (16c)$$

$$f_{ij}^r \in \{0, 1\}, \quad \forall r \in \mathcal{R}, (i, j) \in E, \quad (16d)$$

where we have plugged in t_{ij}^c as defined in (4) as a constant. Generally, this is an integer linear programming (ILP) problem with $O(|\mathcal{R}| \cdot |E|)$ variables and $O(|\mathcal{R}| \cdot |E|)$ constraints, which is no easier to solve than (13). However, under the block placement computed by CG-BPERR, we can convert the problem into a simple shortest-path routing problem as stated below.

Lemma 3.4. *Suppose that the block placement computed by lines 1–9 of Alg. 1 is feasible. Then under this block placement, the shortest-path request routing in lines 10–13 is optimal for (16).*

Remark: As explained in Section 3.3, when applying CG-BPERR, the number of requests $|\mathcal{R}|$ is actually a design parameter that can be tuned to ensure the feasibility of the block placement.

3.2.4. Performance analysis

We now analyze the overall performance of CG-BPERR (Alg. 1) in terms of both the complexity and the average inference time.

Complexity: The complexity can be dominated by either greedy block placement (lines 2–9) or shortest-path request routing (lines 10–13). Each **for** loop in lines 4–9 takes time $O(L\bar{m})$ if line 5 is executed (where $\bar{m} := \max_{j \in V_s} m_j$) or $O(L\bar{m} \log \bar{m})$ if line 7 is executed, both in $O(L^2 \log L)$. Each **for** loop in lines 11–13 takes time $O(|\mathcal{R}_c| \cdot |E|)$, dominated by line 13. The overall complexity of Alg. 1 is thus $O(|V_s|L^2 \log L + |\mathcal{R}| \cdot |E|) = O(|V_s| (L^2 \log L + |\mathcal{R}|(|V_c| + |V_s|)))$ (recall that $|E| = O(|V_s|(|V_c| + |V_s|)))$, which is polynomial in the problem size.

Suboptimality: First of all, due to its conservative assignment of blocks, CG-BPERR may not give a feasible solution when a feasible solution exists. For example, we may have a large number of servers ($|V_s| \geq L|\mathcal{R}|$) but also a large number of requests, such that $s_m + s_c |\mathcal{R}| > M_j$ and $s_m + s_c \leq M_j$ ($\forall j \in V_s$). Then CG-BPERR will be unable to place any block, but it is still feasible to satisfy all the requests by placing one block per server and routing each request to a disjoint set of L servers.

Moreover, CG-BPERR can be suboptimal in comparison to the optimal solution even when it is feasible. Consider an example with $|V_c| = 1$ client and $|V_s| = L^2$ servers, where each $j \in V_s$ has a per-token RTT of $t_{cj} \equiv t$ to the client, a per-block processing time of $\tau_j \equiv \tau$, and a GPU memory of $M_j \equiv (L+1)s_m$. Suppose that $s_m = Ls_c$, and the number of requests is $|\mathcal{R}| = Ls_m/s_c = L^2$. Then CG-BPERR will place only one block per server and route each request through a chain of L servers as illustrated in Fig. 5b, incurring an average per-token inference time of $T^g := L(t + \tau)$. However, the optimal solution is to place all the blocks on each server and route only one request to each server as illustrated in Fig. 5a, incurring an average per-token inference time of $T^o := t + \tau L < T^g$.

Performance guarantee: Nevertheless, CG-BPERR provides a guaranteed average inference time as long as its block placement is feasible.

Theorem 3.5. *Without loss of generality, assume that the servers have been sorted into an increasing order of amortized inference times (i.e., $\tilde{t}_1 \leq \dots \leq \tilde{t}_{|V_s|}$). Let $K := \min\{k : \sum_{j=1}^k m_j \geq L\}$ be the number of iterations of lines 3–9 until all the blocks are placed on at least one server. If CG-BP RR gives a feasible solution (i.e., $K < \infty$), then its average per-token inference time T^g is bounded by*

$$T^g \leq \sum_{j=1}^K \tilde{t}_j m_j - \tau_K \left(\sum_{j=1}^K m_j - L \right), \quad (17)$$

where \tilde{t}_j is defined as in (14) and m_j is computed as in line 1 of Alg. 1.

Remark: The upper bound in (17) has an intuitive meaning that it is the per-token inference time under the *worst-case demand*, where all the requests come from a client c that simultaneously achieves $t_{cj} = t_{*j}$ for all $j \in V_s$ (i.e., farthest away from all the servers), and are routed through the chain of servers $1, \dots, K$. This worst-case guarantee will facilitate the adaptation of CG-BP RR for the online setting as explained below.

In addition to the upper bound in (17), we also provide a lower bound on the minimum average per-token inference time in Appendix B.4, which together with the upper bound yields a bounded approximation ratio for CG-BP RR.

3.3. Online setting

In practice, requests usually arrive dynamically, and block placement and request routing are usually performed at different time scales: block placement should be at a *large time scale* to avoid excessive overhead due to frequently loading/unloading blocks, and request routing should be at a *small time scale* to provide timely service to dynamically arriving requests. Nevertheless, the key ideas in CG-BP RR can be applied in the online setting through a two-time-scale solution as follows.

3.3.1. Block placement via robust optimization

Due to its large overhead, one block placement solution should be able to serve a variety of request scenarios with reasonable performance. We address this requirement by treating the block placement problem as a *robust optimization problem*, with the goal of minimizing the *worst-case* average per-token inference time for a target range of scenarios. To this end, we note that the block placement steps in CG-BP RR (i.e., lines 1–9), referred to as *Conservative Greedy Block Placement (CG-BP)*, are already solving a robust optimization. Specifically, given a maximum number of concurrent requests $|\mathcal{R}|$ that the system aims to support, CG-BP places blocks according to the worst case that all the requests are from a client that is farthest away from all the servers (i.e., with a per-token RTT of t_{*j} , $\forall j \in V_s$). Thus, the performance bound in Theorem 3.5 still applies as stated below.

Corollary 3.6. *If CG-BP (i.e., lines 1–9 of Alg. 1) gives a feasible block placement for $|\mathcal{R}|$ requests, then during online inference, the average per-token inference time under the optimal request routing will always be bounded by (17) as long as the number of concurrent requests⁶ is no more than $|\mathcal{R}|$.*

Remark: The parameter $|\mathcal{R}|$ here is a design parameter that denotes the number of requests the system *guarantees* to serve concurrently and can be tuned to ensure the feasibility of CG-BP. Specifically, given the set of servers and their memory capacities, it is easy to see that the block placement given by lines 1–9 of Alg. 1 is feasible if and only if

$$\sum_{j \in V_s} \min(\lfloor M_j / (s_m + s_c |\mathcal{R}|) \rfloor, L) \geq L. \quad (18)$$

To satisfy the condition (18), it suffices for $|\mathcal{R}|$ to satisfy

$$|\mathcal{R}| \leq \left\lfloor \frac{\sum_{j \in V_s} M_j - s_m(L + |V_s|)}{s_c(L + |V_s|)} \right\rfloor, \quad (19)$$

which provides an upper bound on the number of requests that the system can guarantee to serve concurrently under CG-BP. Meanwhile, increasing $|\mathcal{R}|$ has a negative effect of decreasing m_j ($\forall j \in V_s$), which can increase the path length and hence the inference time. Thus, the parameter $|\mathcal{R}|$ effectively controls the “throughput-delay tradeoff” of the system. When the number of concurrent requests exceeds $|\mathcal{R}|$, the new request may have to wait for some of the existing requests to finish (as detailed in Section 3.3.2). Therefore, $|\mathcal{R}|$ could be tuned to optimize the tradeoff between the waiting time and the inference time after waiting. A configuration method that empirically works well is as follows: (i) estimate the mean and the standard deviation (std) of the number of new requests during the processing of a given request, and (ii) set $|\mathcal{R}|$ to the minimum of the mean plus the std and the upper bound in (19). Such estimation can be further adjusted, on a time scale suitable for block placement, to accommodate time-varying demands.

⁶ In the online setting, “concurrent requests” refer to requests that are simultaneously active (i.e., submitted but unfinished) but do not necessarily arrive simultaneously.

3.3.2. Request routing via individually optimal scheduling

Given the block placement by CG-BP, online request routing strives to schedule each incoming request to a feasible path to complete the request as soon as possible. If the number of concurrent requests is within $|\mathcal{R}|$ (the number of concurrent requests CG-BP plans for), then there will be no memory contention, and the new request can be routed optimally as in lines 10–13 of Alg. 1. If the number of concurrent requests exceeds $|\mathcal{R}|$, however, then the new request may have to wait for some existing requests to finish, in which case it is necessary to know the system state in handling the existing requests.

To this end, we track the state of each server as follows. Let $(T_r^j(t), M_r^j(t))_{r=1}^{R_j(t)}$ denote the state of server j at time t , where $R_j(t)$ is the number of existing (and unfinished) requests routed through server j , $T_r^j(t)$ is the remaining time⁷ for a request $r \in \mathcal{R}_j(t)$, and $M_r^j(t)$ is the number of attention caches hosted on server j for request r (i.e., #blocks j processes for r). Without loss of generality, suppose that the existing requests have been sorted into increasing order of $T_r^j(t)$. Then for a new request arriving at time t , the waiting time for link $(i, j) \in E$ to be available for routing the request (i.e., the time until server j has enough cache space for a new request routed through (i, j)) is given by

$$t_{ij}^W(t) := \min \left\{ T_k^j(t) : \left\lfloor \frac{M_j - s_m m_j}{s_c} \right\rfloor - \sum_{r=k+1}^{R_j(t)} M_r^j(t) \geq a_j + m_j - a_i - m_i \right\}, \quad (20)$$

where $T_0^j(t) := 0$. That is, $t + t_{ij}^W(t)$ is the earliest time for server j to have enough memory for the new request if it is routed from server i . The earliest time a path p is available for the request is then given by $t + \max_{(i,j) \in p} t_{ij}^W(t)$.

Given the current system state at time t , we formulate the *individually optimal* (i.e., myopic) scheduling of a new request arriving from client c at time t as

$$\min_{f, t^W} t^W + l_{\max} \sum_{(i,j) \in E_{a,m}^c} t_{ij}^c f_{ij} \quad (21a)$$

$$\text{s.t. } t_{ij}^W(t) f_{ij} \leq t^W, \quad \forall (i, j) \in E_{a,m}^c, \quad (21b)$$

$$\sum_{i \in \mathcal{N}^+(j; a, m)} f_{ji} - \sum_{i \in \mathcal{N}^-(j; a, m)} f_{ij} = d_j^c, \quad \forall j \in V^c, \quad (21c)$$

$$f_{ij} \in \{0, 1\}, \quad \forall (i, j) \in E_{a,m}^c, \quad (21d)$$

where the main variable is $f_{ij} \in \{0, 1\}$ that indicates whether to route the request through link (i, j) . The objective (21a) is to minimize the *total waiting plus inference time* for this request,⁸ (21b) ensures that all the invoked servers have enough memory when the session starts, and (21c) is the flow conservation constraint, with $\mathcal{N}^+(j; a, m)$ and $\mathcal{N}^-(j; a, m)$ denoting the outgoing/incoming neighbors of j in the feasible routing topology $G_{a,m}^c$ for client c under the given block placement (a, m) , and d_j^c denoting the constant defined in (10). While the second term of (21a) has approximated the total inference time by #output tokens multiplied by the time between tokens, it can be refined to model the actual total inference time by redefining t_{ij}^c as in (8).

While (21) is a MILP that is hard to solve directly, we can simplify it by relaxing t_{ij}^W into its upper bound $\sum_{(i,j) \in E_{a,m}^c} t_{ij}^W(t) f_{ij}$, which reduces (21) into a shortest-path routing problem with *waiting-penalized link cost* $t_{ij}^W(t) + l_{\max} t_{ij}^c$ for each $(i, j) \in E_{a,m}^c$. We refer to this solution as *Waiting-penalized Shortest-path Request Routing (WS-RR)*.

Corollary 3.7. Let $p_c(t)$ denote the path selected by WS-RR for client c at time t . Then the cost of this path $\sum_{(i,j) \in p_c(t)} (t_{ij}^W(t) + l_{\max} t_{ij}^c)$ is an upper bound on the completion time of a request r^* arriving from client c at time t . Moreover, if the number of concurrent requests at time t is within $|\mathcal{R}|$ (the number of requests used by CG-BP in computing the block placement), then $p_c(t)$ is optimal under the given block placement.

3.3.3. Summary of online BPRR

Alg. 2 in Appendix B.5 summarizes our proposed two-time-scale solution for the online setting that combines CG-BP and WS-RR. Together, Corollaries 3.6 and 3.7 imply that this combined solution can guarantee a request completion time that is bounded by

$$\begin{cases} l_{\max} \left(\sum_{j=1}^K \tilde{t}_j m_j - \tau_K \left(\sum_{j=1}^K m_j - L \right) \right) & \text{if } \#\text{concurrent requests} \leq |\mathcal{R}|, \\ \sum_{(i,j) \in p_c(t)} (t_{ij}^W(t) + l_{\max} t_{ij}^c) & \text{o.w.,} \end{cases} \quad (22)$$

where the first bound is independent of the system state while the second bound is dependent. We note that while individually optimal scheduling may not be globally optimal, in our solution it is just used to provide a feasible solution in the exceptional case when $\#\text{concurrent requests} > |\mathcal{R}|$ and thus suffices under properly set $|\mathcal{R}|$.

⁷ We can estimate the remaining time for an existing request r using its completion time by (21a) when scheduling r minus the elapsed time.

⁸ In PETALS implementation [16], there is another delay due to retrying inference at each server according to binary exponential backoff with a maximum backoff time (by default 60 s), causing the actual time from the request arrival to the start of inference to be nonlinear and mildly larger than t^W . We have ignored the retry delay to linearize the formulation (21) but considered such delay in our simulation (see Section 4.1).

Table 2

System configuration for experimentation (entries denote RTT and bandwidth).

	Cluster0 (CPU)	Cluster1 (2 A100s)	Cluster2 (7 MIGs)
Cluster0 (CPU)	5 ms, 1 Gbit/s	100 ms, 100 Mbit/s	100 ms, 100 Mbit/s
Cluster1 (2 A100s)	100 ms, 100 Mbit/s	5 ms, 1 Gbit/s	100 ms, 100 Mbit/s
Cluster2 (7 MIGs)	100 ms, 100 Mbit/s	100 ms, 100 Mbit/s	5 ms, 1 Gbit/s

Table 3

Topologies used in simulation.

	AboveNet	BellCanada	GTS-CE
#nodes	23	48	149
#links	62	130	386
Link capacities (Gbps)	1	1	1
Link delays (ms)	[0.100, 13.800]	[0.078, 6.160]	[0.005, 1.081]

4. Performance evaluation

We evaluate the performance of the proposed algorithms against the state of the art through both controlled experiments based on the PETALS system [8] and data-driven simulations based on a simulator we have developed that has been cross-validated with the experiment results.

4.1. Evaluation setup

Evaluation environment: We employ two cross-validated evaluation environments:

(1) *PETALS-based distributed system*: This environment runs real LLM inference workloads on a modified version of PETALS [8] that can take any block placement and request routing decisions as inputs. We deploy two copies of this system: (i) a *smaller deployment* on a server with 3 A100 (80 GB) GPUs, where we leverage the multi-instance GPU technology [38] to partition one of the A100 GPUs into 7 smaller virtual GPUs (referred to as MIGs) that together with the remaining A100s provide 9 servers with one GPU each, and (ii) a *larger deployment* on a server with 8 A100 (80 GB) GPUs, where we partition three of the A100s into 21 MIGs that together with the remaining A100s provide 26 servers with one GPU each. We use the CPUs to emulate the clients. We use the namespace and the traffic control features of Linux [39] to simulate network latency and bandwidth between the nodes as specified in ‘System configuration’ below.

(2) *MATLAB-based simulator*: To overcome the limited scale of the experiments due to limited GPU resources, we develop a customized simulator in MATLAB that implements the block placement and request routing logic of both the original algorithm in PETALS [8] and the proposed algorithm. The simulator has been engineered to replicate the decision of the real system under the same system state, and validated to approximate the overall experiment results over multiple requests (see Section 4.2). We have open-sourced our simulator code⁹ to facilitate future research on distributed LLM inference for researchers with limited GPU access.

System configuration: We employ BLOOM-176B [3] as the LLM, which is one of the largest open-source LLMs, and configure our evaluation environment to mimic a clustered or scattered deployment scenario as follows.

For the *clustered scenario*, we configure the smaller-scale deployment into three clusters with heterogeneous hardware capabilities:

- Cluster0: a cluster containing clients remote to all the servers;
- Cluster1: a cluster containing two high-performance servers represented by the A100 GPUs as well as clients local to these servers;
- Cluster2: a cluster containing seven low-performance servers represented by the MIGs as well as clients local to these servers.

We add the servers to the system in a random order, and follow the prior work [8] in configuring the networking parameters for intra/inter-cluster communications as presented in Table 2.

For the *scattered scenario*, we simulate three topologies of different sizes from the Internet Topology Zoo [40] with link capacities and delays from [41], as presented in Table 3. We compute the RTTs between nodes according to the cumulative delays along the delay-based shortest paths. In each simulated network, we randomly select C nodes as the locations of servers, η fraction of which (randomly selected) are equipped with high-performance servers represented by A100s and the rest are equipped with MIGs. In the experiments, we use AboveNet for the smaller-scale deployment and BellCanada/GTS-CE for the larger-scale deployment, where C and η are set based on #GPUs of each type in each deployment; in the simulations, we vary C and η to evaluate their impacts.

We generate N_R requests according to a Poisson process with rate λ from one of the clusters (in clustered scenario) or a randomly selected node not hosting any server (in scattered scenario), which represents a proxy that queries the system on behalf of end users

⁹ Our simulator code is available at: https://github.com/TingyangSunJeff/LLM_inference_simulator/tree/main.

Table 4

Average per-token inference time (s) under the configuration in [Table 2](#) ($l_{\max}^I = 20$; 100 requests; MATLAB results shown in parentheses).

Client location	Algorithm	0.1 requests/s		0.5 requests/s	
		$l_{\max} = 64$	$l_{\max} = 128$	$l_{\max} = 64$	$l_{\max} = 128$
Cluster0	PETALS	6.23 (5.33)	4.76 (4.74)	6.28 (5.33)	5.14 (4.74)
	Proposed	1.92 (1.59)	1.43 (0.92)	2.00 (1.59)	1.34 (0.92)
Cluster1	PETALS	5.44 (5.17)	4.60 (4.58)	5.56 (5.17)	4.79 (4.58)
	Proposed	1.78 (1.65)	1.04 (0.83)	1.88 (1.65)	1.11 (0.83)
Cluster2	PETALS	5.30 (4.85)	4.85 (4.07)	5.34 (4.85)	5.25 (4.07)
	Proposed	1.79 (1.59)	1.31 (0.92)	1.94 (1.59)	1.37 (0.92)

Table 5

Average per-token inference time (s) under the topologies in [Table 3](#) ($l_{\max}^I = 20$; 100 requests; MATLAB results shown in parentheses).

Topology	Algorithm	0.1 requests/s		0.5 requests/s	
		$l_{\max} = 64$	$l_{\max} = 128$	$l_{\max} = 64$	$l_{\max} = 128$
AboveNet	PETALS	4.98 (4.75)	4.03 (3.88)	5.26 (5.11)	4.58 (4.10)
	Proposed	1.86 (1.63)	1.44 (1.36)	1.97 (1.83)	1.35 (1.05)
BellCanada	PETALS	6.31 (6.03)	3.82 (3.49)	6.74 (6.19)	4.16 (3.41)
	Proposed	1.33 (1.41)	1.26 (0.92)	1.49 (1.41)	1.11 (0.92)
GTS-CE	PETALS	7.05 (6.12)	4.69 (3.47)	6.89 (5.97)	4.89 (3.37)
	Proposed	1.38 (1.41)	0.95 (0.91)	1.35 (1.40)	1.07 (0.91)

(e.g., a Flask web server as in [\[36\]](#)). Unless stated otherwise, we set the design parameter $|\mathcal{R}|$ as discussed after [Corollary 3.6](#). Our experiment results are averaged over 5 Monte Carlo runs and our simulation results are averaged over 20 Monte Carlo runs.

Benchmark: We compare the proposed two-time-scale algorithm (Alg. 2) with the original block placement and request routing algorithm in PETALS [\[8\]](#), where block placement is performed sequentially by letting each newly-added server choose a consecutive range of the most under-served blocks as measured by a heuristic throughput metric, and request routing is performed by letting each client build a graph with heuristic edge weights based on network latency and processing time,¹⁰ and then use a Dijkstra-like algorithm to find the shortest path that traverses the required blocks in order. We set the target number of concurrent requests to the expected number of arrivals during the first inference session plus one standard deviation.

Metrics: Our primary performance metric is the average inference time per token for *all the tokens*, excluding the local processing times at the client (as they are not affected by the resource allocation at servers). In addition, we also measure the average inference times for *the first token* and *each of the remaining tokens*, respectively, for additional insights, as well as the average running time of each algorithm.

4.2. Experiment results and simulator validation

4.2.1. Inference time

Clustered scenario: [Tables 4](#), [C.7](#), and [C.8](#) present the average inference times for all the tokens, the first token in each sequence, and the remaining tokens, respectively, under various client locations (Cluster 0, 1, or 2), BPRR algorithms (PETALS [\[8\]](#) or the proposed), request rates, and sequence lengths. The results show that: (i) the proposed algorithm (Alg. 2) achieves significantly (60–70%+) smaller average inference times compared to the original algorithm in PETALS in all the tested cases, and (ii) our MATLAB simulator produces results that are roughly consistent with the actual experiments. A closer look shows that the performance improvement mainly comes from the time for the first token ([Table C.7](#)), for which our algorithm achieves an order-of-magnitude reduction. Although the improvement in the inference time for the remaining tokens is smaller ([Table C.8](#)), the overall improvement is dominated by the time reduction for the first token. Comparing the results across different settings, we further see that: (i) increasing the output length can reduce the average per-token time due to amortizing the first token’s inference time over more tokens, and (ii) the client location can affect the performance, e.g., a client in Cluster1 has relatively smaller inference times than a client in Cluster0 under the same algorithm due to its proximity to high-performance servers, but the difference is small as the communication time is only a small fraction of the total inference time (e.g., sending one embedding for BLOOM-176B across two clusters only takes about 0.12 s).

Scattered scenario: [Tables 5](#), [C.9](#), and [C.10](#) present the corresponding results under the network topologies in [Table 3](#). The results show qualitatively similar observations as before, e.g., the proposed algorithm significantly reduces the inference times

¹⁰ Unlike $G_{a_m}^c(t)$ in Alg. 2 that uses our validated performance models to compute edge weights, the routing algorithm in PETALS uses heuristic weights, which causes the sum weight along a path to differ from the actual inference time; see [\[16\]](#) for details.

Table 6

Algorithm running time (s) (algorithm running time is independent of client location, request rate, and sequence length).

Scenario	PETALS	Proposed
Clustered	0.0186 ± 0.0013	0.0216 ± 0.0004
AboveNet	0.0190 ± 0.0081	0.0333 ± 0.0128
BellCanada	0.0291 ± 0.0011	0.0287 ± 0.0018
GTS-CE	0.0350 ± 0.0020	0.0320 ± 0.0012

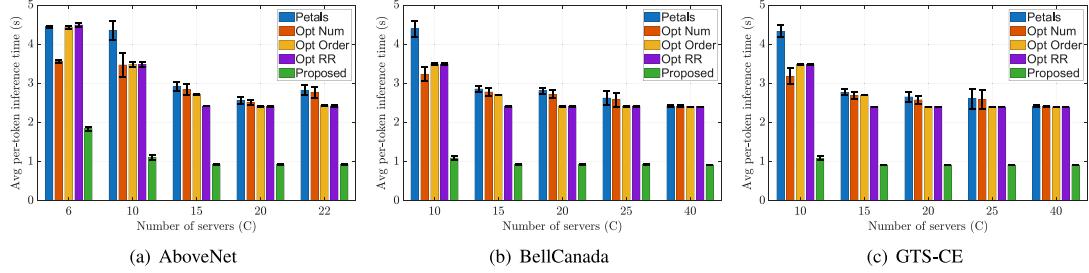


Fig. 6. Inference time per token when varying #servers C ($\eta = 0.2$, $\lambda = 0.5$, $N_R = 100$, $l_{\max}^I = 20$, $l_{\max} = 128$).

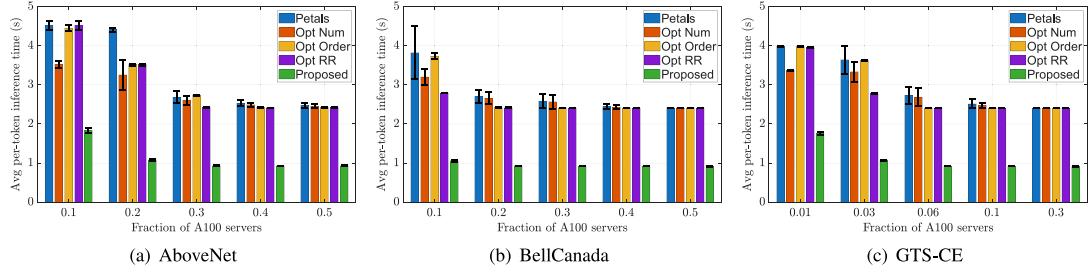


Fig. 7. Inference time per token when varying frac. of high-performance servers η ($C = 0.4 \cdot \#nodes$, $\lambda = 0.5$, $N_R = 100$, $l_{\max}^I = 20$, $l_{\max} = 128$).

in a diverse set of cases with different numbers of servers, different topologies, and different loads (controlled by request rate and sequence length). This validates the generalizability of our observations. We also see bigger improvements for the larger networks (e.g., 60–70% improvement for AboveNet and around 80% improvement for GTS-CE), suggesting promising performance improvement for large deployments.

Remark: We have examined the specific block placement and request routing decisions made by the two algorithms to understand the causes of the observed performance difference. The primary cause turns out to be the difference in how the GPU memory is allocated between model blocks and attention caches: PETALS uses a fixed allocation of attention cache space without considering concurrent sessions [16], which causes it to frequently run out of memory and incur waiting times for incoming requests; in contrast, our solution is designed to handle a certain number of concurrent sessions, which allows it to avoid waiting when properly configured, leading to the significant reduction in the time for the first token. These different memory allocations manifest in the number of blocks placed at each server, where PETALS places 53 blocks on A100 and 4 blocks on MIG, whereas our algorithm (CG-BP) only places 41 blocks on A100 and 3 blocks on MIG. We also note that although our simulator implements the same decision making logic as the real system, it cannot perfectly predict the time for executing the decisions due to complex runtime factors (e.g., memory fragmentation and low-level scheduling) not captured by our system model, which is the cause of the mildly different results. Nevertheless, the simulated performance is overall predictive of the actual performance observed in our experiments.

4.2.2. Algorithm running time

Table 6 shows the average running time of each algorithm evaluated in **Tables 4–5**. To ensure a fair comparison and separate the decision making time from the time of implementing the decisions, we evaluate the running times of both our algorithm and the original algorithm in PETALS [8] based on their MATLAB implementation. Both solutions are fast enough so that the decision making time is negligible compared to the actual inference time.

4.3. Results of experimentally-validated simulations

We now use the validated MATLAB simulator to evaluate a wider range of scenarios based on the topologies in **Table 3**. **Figs. 6–7** show the average inference time per token (over all the tokens) as we vary the available resources in terms of either #servers or the

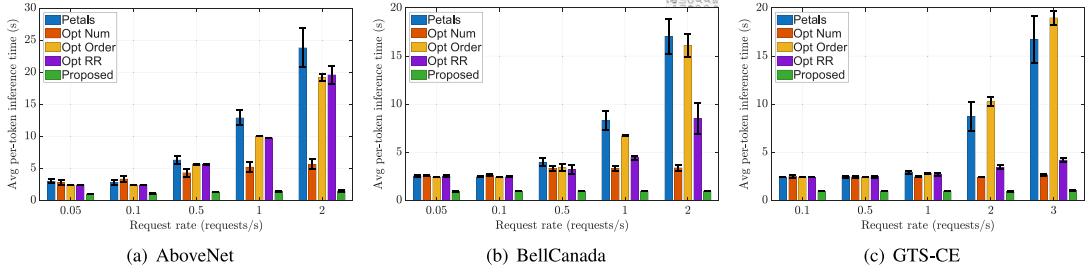


Fig. 8. Inference time per token when varying request rate λ ($C = 0.4 \cdot \# \text{nodes}$, $N_R = 200 \cdot \lambda$, $\eta = 0.2$, $l_{\max}^I = 20$, $l_{\max} = 128$).

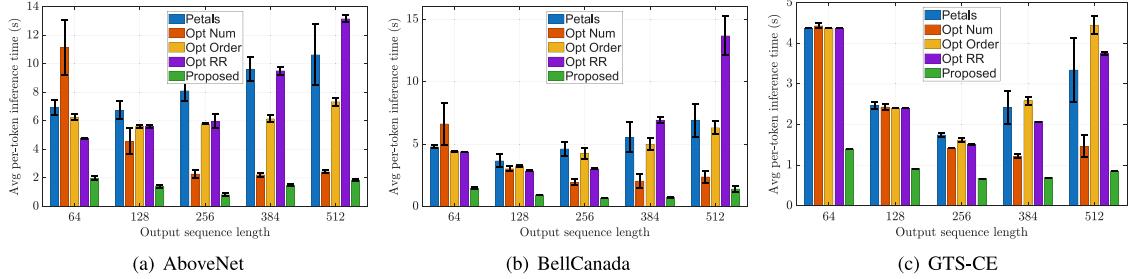


Fig. 9. Inference time per token when varying the sequence length l_{\max} ($C = 0.4 \cdot \# \text{nodes}$, $\eta = 0.2$, $\lambda = 0.5$, $N_R = 100$, $l_{\max}^I = 20$).

fraction of high-performance servers. Figs. 8–9 show the corresponding results as we vary the demands in terms of the request rate or the sequence length. As additional benchmarks, we simulate three variations of PETALS’ algorithm in addition to the original version in [8]: (1) applying PETALS’ block placement algorithm to servers in an optimized order computed by line 3 of our Alg. 2 (‘Optimized Order’), (2) applying PETALS’ block placement algorithm but placing the same number of blocks per server as our algorithm (‘Optimized Number’), and (3) optimizing the request routing according to the MILP (21) under the block placement given by PETALS (‘Optimized RR’). Besides additional benchmarks, these variations also serve the purpose of an ablation study as they each contain one aspect of our proposed solution.

First of all, all the simulations confirm that the proposed algorithm can significantly accelerate the inference in comparison to the original algorithm in PETALS.

Moreover, these results suggest that: (i) optimizing the allocation of GPU memory between model blocks and attention caches as in ‘Optimized Number’ can improve the inference time in most cases, particularly under relatively high demands; (ii) optimizing the order of block placement across servers as in ‘Optimized Order’ may help in some cases, but does not always improve the inference time; (iii) similarly, optimizing request routing as in ‘Optimized RR’ may help in some cases, but can worsen the performance for long sequences (Fig. 9). In contrast, our proposed solution that combines all these ideas is able to improve the performance in all the tested cases. This observation highlights the importance of *systematically formulating and solving the resource allocation problem in distributed LLM inference systems*. Meanwhile, the fact that ‘Optimized Order’ and ‘Optimized RR’ can sometimes underperform the original algorithm in PETALS suggests the suboptimality of the greedy block placement and myopic request scheduling strategy, which leaves potential room for improvement in future work.

Furthermore, comparing across the simulated cases indicates a trend that *the performance improvement achieved by our proposed solution is larger in more resource-constrained scenarios* (e.g., fewer servers/high-performance servers or higher request rate), in which case good resource allocation is more important. We note that in practice, the demands usually grow with the system size. Thus, we further simulate a case of proportionally increasing #servers and request rate in Fig. C.13, which shows a clear trend of widening performance gap between our solution and PETALS, indicating the potential for our solution to achieve even greater performance improvements in larger deployments. We also conduct a sensitivity analysis with respect to the load parameter $|\mathcal{R}|$ as shown in Fig. C.14, which shows that while the configuration of this parameter affects the performance of our solution, its impact is mild and our solution remains superior to the benchmarks over a wide range of actual loads.

Meanwhile, our algorithm has slightly higher running times than the original algorithm in PETALS in some cases (see Figs. C.15–C.20 in Appendix C.2), but the difference is negligible compared to the actual inference time.

5. Conclusion

In this work, we systematically studied the problem of performance optimization for geographically-distributed LLM inference, using PETALS as a concrete example. Using experimentally validated performance models, we formulated the optimal offline block placement and request routing problem as a MILP, proved its NP-hardness, and developed a polynomial-complexity algorithm with

guaranteed performance. We then adapted our algorithm into a two-time-scale solution for the online setting with a guaranteed completion time under bounded load. Our experiments and cross-validated simulations in diverse settings not only confirmed the capability of the proposed algorithm in significantly reducing the inference times, but also provided insights on the key factors driving such improvement. In addition to the developed algorithm, this work also produced a light-weighted CPU-only simulator capable of predicting the performance of distributed LLM inference on profiled GPU servers, which will be open-sourced to facilitate future research on the performance of LLM for researchers with limited GPU access.

CRediT authorship contribution statement

Tingyang Sun: Conceptualization, Data curation, Investigation, Methodology, Software, Writing – review & editing. **Ting He:** Conceptualization, Formal analysis, Methodology, Resources, Supervision, Writing – original draft, Writing – review & editing. **Bo Ji:** Supervision, Writing – review & editing. **Parimal Parag:** Conceptualization, Supervision.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Ting He reports financial support was provided by National Science Foundation. Parimal Parag reports financial support was provided by Qualcomm Inc. Parimal Parag reports financial support was provided by Anusandhan National Research Foundation (ANRF). Parimal Parag reports financial support was provided by Office of International Relations at IISc. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Supporting proofs

Proof of Lemma 3.1. Denote the sequence of servers on p by i_1, \dots, i_n . For the first hop (c, i_1) , $a_{i_1} \leq a_c + m_c = 1 \leq a_{i_1} + m_{i_1} - 1$ implies that block 1 is hosted by server i_1 , i.e., the first $a_{i_1} + m_{i_1} - 1$ blocks can be found in order by traversing p up to i_1 . Similarly, if the first $a_{i_k} + m_{i_k} - 1$ blocks ($k \in [n-1]$) can be found in order up to i_k , then $a_{i_{k+1}} \leq a_{i_k} + m_{i_k} \leq a_{i_{k+1}} + m_{i_{k+1}} - 1$ implies that the first $a_{i_{k+1}} + m_{i_{k+1}} - 1$ blocks can be found in order up to i_{k+1} . Thus, the first $a_{i_n} + m_{i_n} - 1$ blocks can be found in order up to i_n . Moreover, since $L+1 = a_c \leq a_{i_n} + m_{i_n} \leq a_c + m_c - 1 = L+1$, $a_{i_n} + m_{i_n} - 1 = L$. Thus, all the blocks can be found in order along path p , i.e., p is feasible. \square

Proof of Theorem 3.2. We prove this theorem by a reduction from the optimization version of the *partition problem* [37]. Given a set of positive integers W with $\Delta := (\sum_{w \in W} w)/2$, the optimization version of the partition problem aims at finding a partition of W into W_1 and W_2 such that $|\sum_{w \in W_1} w - \sum_{w \in W_2} w|$ is minimized. Without loss of generality, assume all the numbers in W to be even and upper-bounded by¹¹ Δ . We construct an instance of the BPRR problem as follows: set the total number of blocks to $L = 2$; construct a single client $V_c := \{c\}$ with $|\mathcal{R}_c| = \Delta$; construct a server j for each $w_j \in W$, and set s_m , s_c , and M_j such that $s_m + s_c w_j = M_j < 2s_m$; set $t_j := t_{cj} + \tau_j = 1$ for each $w_j \in W$; construct another server 0 (assuming $w_0 \notin W$) with $s_m + s_c \Delta = M_0 < 2s_m$ and $t_0 := t_{c0} + \tau_0 = 2$. This construction is possible as long as $s_m/s_c > \Delta$ (which implies $s_m/s_c > w_j, \forall w_j \in W$). For any partition (W_1, W_2) with $\sum_{w \in W_1} w \geq \sum_{w \in W_2} w$, let $\epsilon := \frac{1}{2}(\sum_{w \in W_1} w - \sum_{w \in W_2} w)$, i.e., $\sum_{w \in W_1} w = \Delta + \epsilon$ and $\sum_{w \in W_2} w = \Delta - \epsilon$. The assumption of W containing only even numbers ensures that Δ and ϵ are both nonnegative integers. By construction, every server can only store one block, server j for each $w_j \in W$ can serve w_j parallel sessions with a per-token inference time of $t_j = 1$, and server 0 can serve Δ parallel sessions with a per-token inference time of $t_0 = 2$. It is easy to see that the optimal objective value of the constructed instance of BPRR conditioned on placing block 1 at the servers in $S_1 := \{j : w_j \in W_1\}$ and block 2 at the servers in $S_2 := \{j : w_j \in W_2\} \cup \{0\}$ is $2\Delta + \epsilon$, achieved by routing the maximum #requests to the faster servers before using server 0. Thus, minimizing the subset sum difference for the partition problem is equivalent to minimizing the objective value of the corresponding BPRR problem. The proof completes by noting that the optimization version of the partition problem is NP-hard [37]. \square

Proof of Lemma 3.3.

As servers are considered in the increasing order of \tilde{t}_j (line 3), the value of T_b for each block b only changes once from $\tilde{t}_0|\mathcal{R}|$ to $\tilde{t}_{j_b}|\mathcal{R}|$, when block b is placed for the first time on a server $j_b \in V_s$. Therefore, with “argmax” breaking ties in favor of smaller index, line 5 will select continuous blocks for each server before all the blocks are placed, i.e., server 1 will host blocks $1, \dots, m_1$, server 2 will host blocks $m_1 + 1, \dots, m_1 + m_2$, etc. This continues until server $K := \min\{k : \sum_{j=1}^k m_j \geq L\}$, for which the “argmax” in line 5 will be achieved at the last possible index $a_K = L - m_K + 1$. The resulting T_b ’s form a monotone increasing piece-wise constant series as shown in Fig. A.10. Under the conservative setting of m_j in line 1 and this block placement, the relaxed request routing will simply route all the $|\mathcal{R}|$ requests through the first K servers, achieving an average per-token inference time of

$$\sum_{j=1}^K m_j \tilde{t}_j - \tilde{t}_K (\sum_{j=1}^K m_j - L), \quad (\text{A.1})$$

¹¹ The first assumption is because we can scale all the numbers by 2 without changing the solution. The second assumption is because if $\exists w^* \in W$ such that $w^* > \Delta$, then the optimal solution is simply $W_1 = \{w^*\}$ and $W_2 = W \setminus \{w^*\}$, and thus it suffices to consider instances where $w \leq \Delta$ for all $w \in W$.

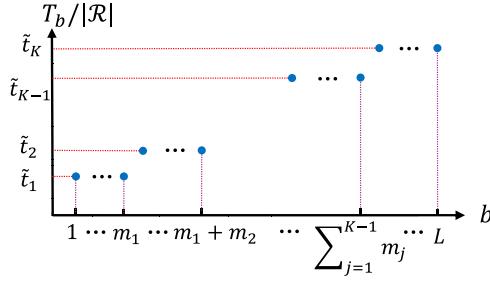


Fig. A.10. Illustration of T_b in Alg. 1 after placing all the blocks.

where the second term is to adjust for the $\sum_{j=1}^K m_j - L$ blocks hosted on both server $K-1$ and server K , as they will only be processed at server $K-1$. Meanwhile, since these are the fastest servers (in terms of \tilde{t}_j) that can host the entire model, the minimum per-token inference time under the relaxed request routing is lower-bounded by (A.1). This completes the proof. \square

Proof of Lemma 3.4. Due to the conservative calculation of the number of blocks per server (line 1), the GPU memory capacity will always be satisfied, i.e., (16b) can be ignored. This decouples (16) into $|V_c|$ independent routing problems for each client.

For each client c , the route feasibility constraints (11)–(12) can be enforced by limiting its request routing to a subgraph $G_{a,m}^c = (V^c, E_{a,m}^c)$ of G , where V^c only contains the servers V_s and the S-client/D-client of c , and $E_{a,m}^c$ only contains feasible routing links, i.e., $(i, j) \in E_{a,m}^c$ if and only if (3) is satisfied. Each link $(i, j) \in E_{a,m}^c$ has a routing cost of t_{ij}^c . It is easy to see that the subproblem of (16) for client c is equivalent to the problem of finding the shortest (i.e., least-cost) path from the S-client to the D-client in $G_{a,m}^c$ (line 12). In absence of the capacity constraint (16b), the optimal paths for all the requests from the same client are the same (line 13). This completes the proof. \square

Proof of Theorem 3.5. Recall that $t_{*j} := \max_{c \in V_c} t_{cj}$. Since by Lemma 3.4 the shortest-path request routing is optimal under the block placement by CG-BPRR, the average per-token inference time T^g achieved by CG-BPRR is upper-bounded by the average per-token inference time under any request routing that is feasible under the block placement given by CG-BPRR. Specifically, consider a solution that routes all the requests through the chain of servers $1, \dots, K$. By the proof of Lemma 3.3, server 1 will host the first m_1 blocks, server 2 will host the next m_2 blocks, etc., and server K will host the last m_K blocks. Thus, servers $1, \dots, K$ collectively host all the blocks. Moreover, due to the conservative computation of m_j in line 1 of Alg. 1, each server has enough capacity to serve all the requests concurrently. Thus, the above routing is feasible. Under this solution, server j ($j = 1, \dots, K-1$) processes m_j blocks, yielding a per-token inference time of

$$t_{cj} + \tau_j m_j \leq t_{*j} + \tau_j m_j = m_j \tilde{t}_j, \quad \forall c \in V_c. \quad (\text{A.2})$$

Meanwhile, server K only processes the last $L - \sum_{j=1}^{K-1} m_j$ blocks, yielding a per-token inference time of

$$t_{cK} + \tau_K (L - \sum_{j=1}^{K-1} m_j) \leq t_{*K} + \tau_K (L - \sum_{j=1}^{K-1} m_j) = t_{*K} \left(\frac{\sum_{j=1}^K m_j - L}{m_K} \right) + (L - \sum_{j=1}^{K-1} m_j) \tilde{t}_K, \quad \forall c \in V_c. \quad (\text{A.3})$$

The average per-token inference time \bar{T}^g under the above routing solution is thus upper-bounded by

$$\begin{aligned} \bar{T}^g &\leq \sum_{j=1}^{K-1} m_j \tilde{t}_j + t_{*K} \left(\frac{\sum_{j=1}^K m_j - L}{m_K} \right) + (L - \sum_{j=1}^{K-1} m_j) \tilde{t}_K \\ &= \sum_{j=1}^K \tilde{t}_j m_j - \left(\sum_{j=1}^K m_j - L \right) \left(\tilde{t}_K - \frac{t_{*K}}{m_K} \right) \\ &= \sum_{j=1}^K \tilde{t}_j m_j - \tau_K \left(\sum_{j=1}^K m_j - L \right), \end{aligned} \quad (\text{A.4})$$

where (A.4) is because $\tilde{t}_K - t_{*K}/m_K = \tau_K$ by the definition in (14). The proof is completed by noting that $T^g \leq \bar{T}^g$. \square

Proof of Corollary 3.6. The result is directly applied by Theorem 3.5. Specifically, if the number of concurrent requests is no more than $|R|$, then the block placement by CG-BP ensures that there is no memory contention between requests, i.e., each request can be routed independently of the others. Since the average per-token inference time under the feasible but possibly suboptimal request routing through servers $1, \dots, K$ is already bounded by (17) as shown in the proof of Theorem 3.5, the average per-token inference time under the optimal request routing must be bounded by (17). \square

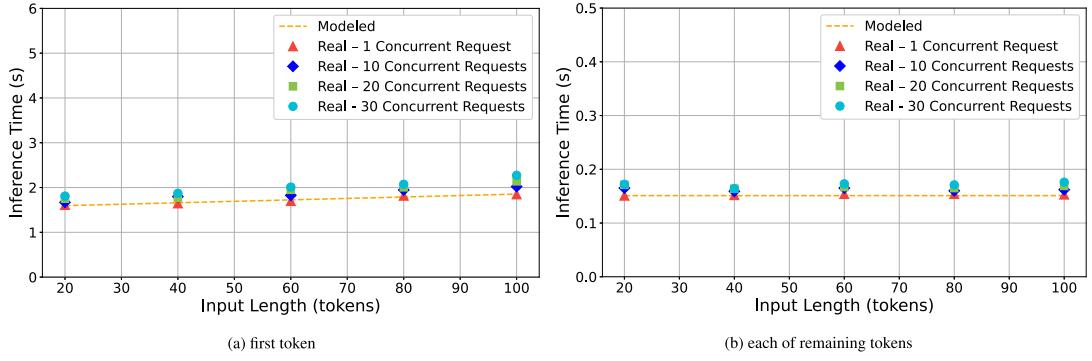


Fig. B.11. Inference time vs. input length on A100: (a) for first token generation (b) for rest of token generation (40 blocks, $l_{\max} = 128$).

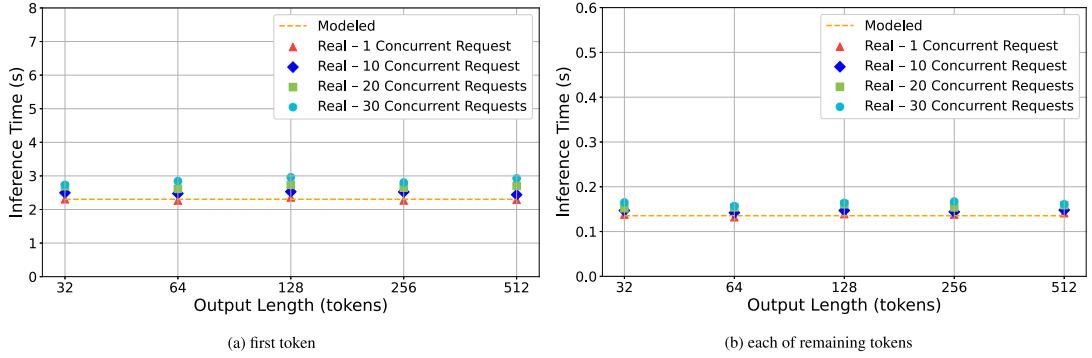


Fig. B.12. Inference time vs. output length on A100: (a) for first token generation (b) for rest of token generation (40 blocks, $l_{\max}^I = 20$).

Proof of Corollary 3.7. The first claim follows from the fact that the completion time of r^* (i.e., the time from its arrival to its completion) if scheduled to path $p_c(t)$ is

$$\max_{(i,j) \in p_c(t)} t_{ij}^A + l_{\max} \sum_{(i,j) \in p_c(t)} t_{ij}^c \leq \sum_{(i,j) \in p_c(t)} (t_{ij}^A(t) + l_{\max} t_{ij}^c). \quad (\text{A.5})$$

The second claim follows from the fact that there is no waiting time if the number of concurrent requests is within $|\mathcal{R}|$, i.e., $t_{ij}^A(t) \equiv 0 \forall (i,j)$. In this case, the bound (A.5) is tight, i.e., the cost of a path equals the completion time on this path. By definition, $p_c(t)$ minimizes the path cost, and thus it achieves the minimum completion time. \square

Appendix B. Other supporting materials

B.1. Additional results on performance model validation

In addition to the number of processed blocks as in Fig. 2, we have also evaluated the inference time incurred at a given server under varying input/output length as shown in Figs. B.11–B.12. These results have validated our inference time model by confirming that the inference time for the first token only depends on the input length l_{\max}^I but not the output length l_{\max} , and the inference time for each of the subsequent tokens does not depend on either the input or the output length. Besides A100, we have also validated our performance models on an MIG, which shows similar results.

B.2. Extension to heterogeneous input/output lengths

In the heterogeneous case where each request $r \in \mathcal{R}$ has an input length l_r^I and an output length l_r , the attention cache size becomes heterogeneous $s_c^r := 2d_{\text{model}} \cdot (l_r^I + l_r) \cdot \text{dtype_bytes}$ (bytes). Our formulation (6) can be easily extended by replacing the objective function (6a) with

$$\sum_{c \in V_c} \sum_{r \in \mathcal{R}_c} \sum_{p \in P_c(a, m)} f_p^r l_r \sum_{(i,j) \in p} t_{ij}^r, \quad (\text{B.1})$$

where t_{ij}^r denotes the per-token inference time incurred by request r at link (i, j) averaged over all the tokens generated for this request, and the constraint (6b) with

$$s_m m_j + \sum_{c \in V_c} \sum_{r \in \mathcal{R}_c} s_c^r \sum_{\substack{p \in P_c(a, m): \\ (i, j) \in p, \exists i}} f_p^r (a_j + m_j - a_i - m_i) \leq M_j, \quad \forall j \in V_s. \quad (\text{B.2})$$

Similar to (8), t_{ij}^r is defined as:

$$\left(\frac{1}{l_r} t_{cj}^I (l_r^I) + \frac{l_r - 1}{l_r} t_{cj} \right) + \left(\frac{1}{l_r} \tau_j^I (l_r^I) + \frac{l_r - 1}{l_r} \tau_j \right) (a_j + m_j - a_i - m_i). \quad (\text{B.3})$$

When $l_r^I \ll l_r$ for all $r \in \mathcal{R}$, (B.3) can be simplified to (4), where c is the client sending request r .

Since this change only affects some constant scaling factors, we can still convert the resulting optimization into a MILP as in Section 3.2.2. With $s_c := \max_r s_c^r$, CG-BPRR can still be used to obtain a feasible solution in the offline setting. We note that since in practice the requests arrive dynamically, the precise input/output length for each request is unknown ahead of time. In this sense, our proposed solution based on (6) allocates resources according to the maximum input and output lengths to ensure feasibility while trying to optimize the worst-case performance, where the maximum input and output lengths are system parameters announced to the clients.

B.3. Linearization of bilinear terms

To linearize the bilinear terms $a_j f_{ij}^r$, $a_i f_{ij}^r$, $m_j f_{ij}^r$, and $m_i f_{ij}^r$, we introduce auxiliary variables $\alpha_{ij}^r, \beta_{ij}^r, \gamma_{ij}^r, \delta_{ij}^r \geq 0$ as well as the following linear constraints. Specifically, using the constraints

$$-(L+1)f_{ij}^r + \alpha_{ij}^r \leq 0, \quad (\text{B.4a})$$

$$-a_j + \alpha_{ij}^r \leq 0, \quad (\text{B.4b})$$

$$a_j + (L+1)f_{ij}^r - \alpha_{ij}^r \leq L+1, \quad (\text{B.4c})$$

we can ensure that $\alpha_{ij}^r = a_j f_{ij}^r$ for both $f_{ij}^r = 0$ and $f_{ij}^r = 1$ (the ' $L+1$ ' is to cover the case of $a_j = L+1$ for $j \in V_c^D$). Similarly, we can ensure that $\beta_{ij}^r = a_i f_{ij}^r$ by the constraints

$$-L f_{ij}^r + \beta_{ij}^r \leq 0, \quad (\text{B.5a})$$

$$-a_i + \beta_{ij}^r \leq 0, \quad (\text{B.5b})$$

$$a_i + L f_{ij}^r - \beta_{ij}^r \leq L, \quad (\text{B.5c})$$

$\gamma_{ij}^r = m_j f_{ij}^r$ by the constraints

$$-L f_{ij}^r + \gamma_{ij}^r \leq 0, \quad (\text{B.6a})$$

$$-m_j + \gamma_{ij}^r \leq 0, \quad (\text{B.6b})$$

$$m_j + L f_{ij}^r - \gamma_{ij}^r \leq L, \quad (\text{B.6c})$$

and $\delta_{ij}^r = m_i f_{ij}^r$ by the constraints

$$-L f_{ij}^r + \delta_{ij}^r \leq 0, \quad (\text{B.7a})$$

$$-m_i + \delta_{ij}^r \leq 0, \quad (\text{B.7b})$$

$$m_i + L f_{ij}^r - \delta_{ij}^r \leq L. \quad (\text{B.7c})$$

B.4. Approximation ratio of CG-BPRR

In addition to the upper bound in Theorem 3.5, we can also lower-bound the inference time as follows. Let $\bar{m}_j := \min(\lfloor M_j / (s_m + s_c) \rfloor, L)$ denote the maximum number of blocks that can be placed at server j (while still able to serve at least one request), and $\tilde{\tau}_{j,c} := \tau_j + t_{cj}/\bar{m}_j$ denote the *minimum amortized inference time* that server j can provide for generating a token for client c . Let $j_k^{(c)}$ denote a server index such that $\tilde{\tau}_{j_1^{(c)},c} \leq \dots \leq \tilde{\tau}_{j_{|V_s|}^{(c)},c}$.

Lemma B.1. *The minimum per-token inference time for client c is lower-bounded by*

$$T_c^o := \sum_{k=1}^{K_c-1} \tilde{\tau}_{j_k^{(c)},c} \bar{m}_{j_k^{(c)}} + \tilde{\tau}_{j_{K_c}^{(c)},c} \left(L - \sum_{k=1}^{K_c-1} \bar{m}_{j_k^{(c)}} \right), \quad (\text{B.8})$$

Algorithm 2: Two-time-scale BPRR for Online Setting

input : set of clients V_c , target #requests $|\mathcal{R}|$, #blocks L , size per block s_m , size per cache s_c , set of servers V_s , parameters for each $j \in V_s$ including GPU memory M_j , processing time τ_j , and per-token RTTs $(t_{cj})_{c \in V_c}$

output: Block placement (\mathbf{a}, \mathbf{m}) and request routing f

// Conservative Greedy Block Placement (CG-BP) :

- 1 $m_j \leftarrow \min(\lfloor M_j/(s_m + s_c|\mathcal{R}|) \rfloor, L), \forall j \in V_s;$
- 2 $C_b \leftarrow 0, T_b \leftarrow \tilde{t}_0|\mathcal{R}|, \forall b \in [L];$
- 3 **for** each server $j \in V_s$ in increasing order of \tilde{t}_j **do**
- 4 **if** $\exists b \in [L]$ with $C_b < |\mathcal{R}|$ **then**
- 5 $a_j \leftarrow \operatorname{argmax}_{a \in [L-m_j+1]: C_b < |\mathcal{R}|, \exists b \in \{a, \dots, a+m_j-1\}} \sum_{b'=a}^{a+m_j-1} T_{b'};$
- 6 **else**
- 7 $a_j \leftarrow \operatorname{argmin}_{a \in [L-m_j+1]} (C_a, \dots, C_{a+m_j-1});$
- 8 $T_b \leftarrow T_b - (\tilde{t}_0 - \tilde{t}_j) \min(\max(|\mathcal{R}| - C_b, 0), \tilde{f}_j), \forall b \in \{a_j, \dots, a_j + m_j - 1\};$
- 9 $C_b \leftarrow C_b + \tilde{f}_j, \forall b \in \{a_j, \dots, a_j + m_j - 1\};$

// Waiting-penalized Shortest-path Request Routing (WS-RR) :

- 10 **for** each new request r arriving from client c at time t **do**
- 11 $G_{\mathbf{a}, \mathbf{m}}^c(t) \leftarrow$ the feasible routing topology for client c under block placement (\mathbf{a}, \mathbf{m}) , with a node/link set $(V^c, E_{\mathbf{a}, \mathbf{m}}^c)$ and a waiting-penalized cost of $t_{ij}^W(t) + l_{\max} t_{ij}^c$ for each $(i, j) \in E_{\mathbf{a}, \mathbf{m}}^c$;
- 12 $p_c(t) \leftarrow$ shortest path from the S-client to the D-client in $G_{\mathbf{a}, \mathbf{m}}^c(t)$;
- 13 $f_{ij}^r \leftarrow \mathbb{1}((i, j) \in p_c(t)), \forall (i, j) \in E;$

where $K_c := \min\{K : \sum_{k=1}^K \bar{m}_{j_k^{(c)}} \geq L\}$. Thus, the minimum average per-token inference time T^o for the requests $\{\mathcal{R}_c\}_{c \in V_c}$ is lower-bounded by $T^o \geq \frac{1}{|\mathcal{R}|} \sum_{c \in V_c} |\mathcal{R}_c| T_c^o$.

Proof of Lemma B.1. It suffices to prove that the per-token inference time for client c is lower-bounded by (B.8). First, we note that the minimum per-token inference time is lower if we relax the request routing to block-by-block routing with a per-block inference time $\tilde{t}_{j,c}$ at server j , because in reality a request cannot be served more than \bar{m}_j blocks at server j and has to incur the entire client-server RTT t_{cj} even if being processed by only a subset of the blocks at server j . Then, under such relaxed request routing, the minimum inference time for a token is achieved by routing to the fastest servers that collectively hold all the blocks, which are servers $j_1^{(c)}, \dots, j_{K_c}^{(c)}$. That is, processing the first $\sum_{k=1}^{K_c-1} \bar{m}_{j_k^{(c)}}$ blocks at servers $j_1^{(c)}, \dots, j_{K_c-1}^{(c)}$, and the remaining $L - \sum_{k=1}^{K_c-1} \bar{m}_{j_k^{(c)}}$ blocks at server $j_{K_c}^{(c)}$. This corresponds to the per-token inference time given in (B.8), which then lower-bounds the actual per-token inference time achievable for client c under any feasible BPRR solution. \square

Combining Theorem 3.5 and Lemma B.1 yields an upper bound on T^g/T^o , which is the approximation ratio for CG-BPRR.

B.5. Two-time-scale algorithm for online BPRR

For completeness, we summarize the proposed BPRR algorithm for the online setting in Alg. 2, which is adapted from the CG-BPRR algorithm for the offline setting (Alg. 1). While Alg. 2 only runs CG-BP once at the beginning, it can be easily extended to adapt the block placement by invoking CG-BP again when the observed number of concurrent requests deviates significantly from the previously targeted value.

Appendix C. Additional evaluation results

C.1. Additional experiment results

Tables C.7–C.8 present the breakdown of the total average inference time in Table 4 into the inference time for the first token and the inference time for each of the remaining tokens. Similarly, Tables C.9–C.10 provide the breakdown of the total average inference time in Table 5.

C.2. Additional simulation results

To evaluate the performance of our solution as the system scales, we proportionally increase the number of servers and the request rate as in Fig. C.13, and the result shows a trend of widening performance gap between our solution and PETALS.

We further evaluate the sensitivity of our solution to the configuration of the load parameter $|\mathcal{R}|$. Under a fixed $|\mathcal{R}|$ computed for a predicted rate λ_{base} , Fig. C.14 shows the performance under different actual rates, which shows that in comparison with setting $|\mathcal{R}|$ according to the actual rates as in Fig. 8, using a fixed $|\mathcal{R}|$ can lead to increased inference time when the actual rate is higher

Table C.7

Average inference time for the first token (s) under the configuration in [Table 2](#) ($I_{\max}^I = 20$; 100 requests; MATLAB results shown in parentheses).

Client location	Algorithm	0.1 requests/s		0.5 requests/s	
		$I_{\max} = 64$	$I_{\max} = 128$	$I_{\max} = 64$	$I_{\max} = 128$
Cluster0	PETALS	275.51 (252.61)	455.61 (427.72)	315.53 (252.61)	508.52 (427.72)
	Proposed	60.43 (73.51)	62.98 (60.91)	66.94 (73.51)	69.62 (60.91)
Cluster1	PETALS	236.34 (252.51)	463.39 (424.94)	298.27 (252.51)	489.85 (424.06)
	Proposed	55.47 (85.50)	59.67 (72.06)	57.37 (85.50)	65.96 (72.06)
Cluster2	PETALS	245.70 (251.95)	470.33 (404.42)	259.69 (251.95)	480.43 (404.42)
	Proposed	50.96 (73.51)	55.72 (60.91)	55.36 (73.51)	59.86 (60.91)

Table C.8

Average inference time for the remaining token (s) under the configuration in [Table 2](#) ($I_{\max}^I = 20$; 100 requests; MATLAB results shown in parentheses).

Client location	Algorithm	0.1 requests/s		0.5 requests/s	
		$I_{\max} = 64$	$I_{\max} = 128$	$I_{\max} = 64$	$I_{\max} = 128$
Cluster0	PETALS	1.96 (1.40)	1.21 (1.41)	1.37 (1.40)	1.18 (1.41)
	Proposed	0.99 (0.45)	0.94 (0.45)	0.96 (0.45)	0.81 (0.45)
Cluster1	PETALS	1.78 (1.25)	0.99 (1.27)	0.91 (1.25)	0.97 (1.27)
	Proposed	0.93 (0.32)	0.58 (0.26)	0.98 (0.32)	0.60 (0.26)
Cluster2	PETALS	1.49 (0.93)	1.18 (0.91)	1.29 (0.93)	1.51 (0.91)
	Proposed	1.01 (0.45)	0.88 (0.45)	1.09 (0.45)	0.91 (0.45)

Table C.9

Average inference time for the first token (s) under the topologies in [Table 3](#) ($I_{\max}^I = 20$; 100 requests; MATLAB results shown in parentheses).

Topology	Algorithm	0.1 requests/s		0.5 requests/s	
		$I_{\max} = 64$	$I_{\max} = 128$	$I_{\max} = 64$	$I_{\max} = 128$
AboveNet	PETALS	270.96 (254.74)	408.59 (316.21)	279.38 (264.81)	486.68 (412.72)
	Proposed	76.48 (73.69)	89.15 (104.19)	81.65 (81.12)	91.41 (75.78)
BellCanada	PETALS	360.94 (353.12)	382.01 (354.06)	378.10 (353.46)	417.17 (353.72)
	Proposed	53.42 (62.75)	59.73 (62.68)	63.78 (62.67)	66.84 (62.70)
GTS-CE	PETALS	393.93 (353.48)	478.71 (354.05)	389.35 (353.46)	505.90 (353.79)
	Proposed	55.30 (62.84)	58.59 (62.66)	58.17 (62.65)	61.22 (62.65)

Table C.10

Average inference time for the remaining tokens (s) under the topologies in [Table 3](#) ($I_{\max}^I = 20$; 100 requests; MATLAB results shown in parentheses).

Topology	Algorithm	0.1 requests/s		0.5 requests/s	
		$I_{\max} = 64$	$I_{\max} = 128$	$I_{\max} = 64$	$I_{\max} = 128$
AboveNet	PETALS	0.77 (0.79)	0.84 (0.92)	0.91 (0.98)	0.78 (0.88)
	Proposed	0.67 (0.52)	0.75 (0.55)	0.70 (0.57)	0.64 (0.46)
BellCanada	PETALS	0.69 (0.53)	0.84 (0.73)	0.80 (0.68)	0.91 (0.66)
	Proposed	0.50 (0.44)	0.79 (0.44)	0.51 (0.44)	0.59 (0.44)
GTS-CE	PETALS	0.92 (0.61)	0.96 (0.71)	0.83 (0.44)	0.95 (0.67)
	Proposed	0.52 (0.43)	0.50 (0.42)	0.49 (0.42)	0.59 (0.43)

than expected, due to not reserving enough memory for attention caches and thus causing some requests to incur extensive waiting. Nevertheless, the increase for our solution ('Proposed') is much smaller than that for the BPRR algorithm of PETALS ('Optimized Number'), demonstrating the robustness of our solution to load prediction.

[Figs. C.15–C.19](#) show the running time of each algorithm in the simulations of [Figs. 6–9](#). The results show that both the proposed algorithm ('Proposed') and the original algorithm in [8] ('Petals') are fast enough so that their execution costs negligible time compared to the actual inference time. The running time for 'Optimized RR' is substantially higher than the others as in this case we directly solve the MILP (21) by the Gurobi optimizer [42]. The same observation holds in [Fig. C.20](#), which corresponds to the case in [Fig. C.13](#) as we proportionally increase both the number of servers and the request rate.

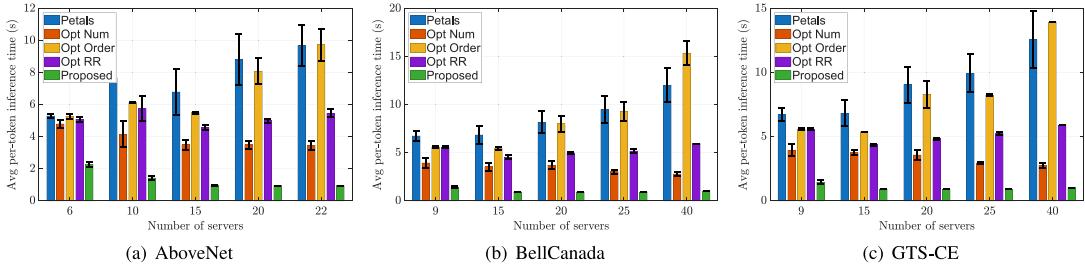


Fig. C.13. Inference time per token when varying the number of servers C and rate $\lambda = (0.1/9) \cdot C$ ($N_R = 100$, $\eta = 0.2$, $l_{\max}^I = 20$, $l_{\max} = 128$).

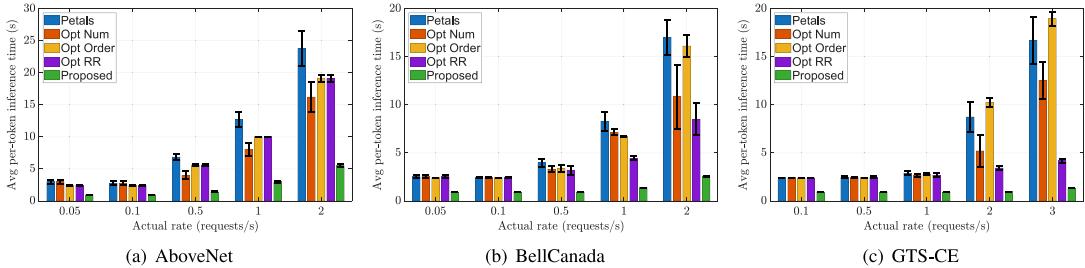


Fig. C.14. Inference time per token when setting $|\mathcal{R}|$ according to λ_{base} and varying actual request rate λ ($C = 0.4 \cdot \# \text{nodes}$, $\lambda_{\text{base}} = 0.5$, $N_R = 200 \cdot \lambda$, $\eta = 0.2$, $l_{\max}^I = 20$, $l_{\max} = 128$).

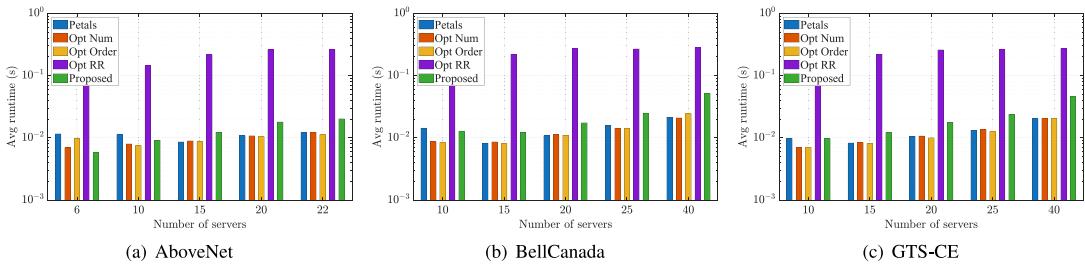


Fig. C.15. Algorithm running time when varying $\# \text{servers } C$ ($\eta = 0.2$, $\lambda = 0.5$, $N_R = 100$, $l_{\max}^I = 20$, $l_{\max} = 128$).

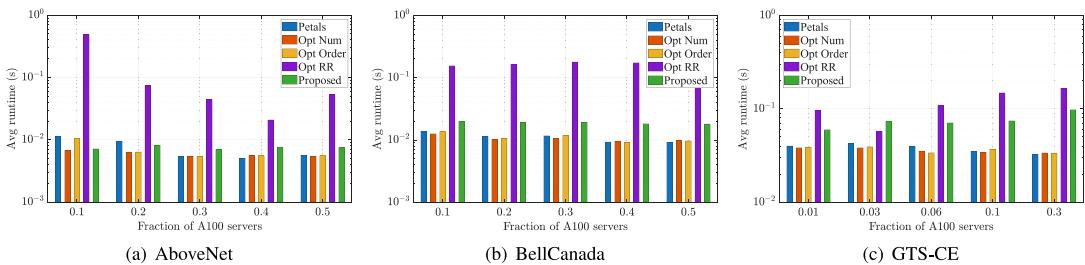


Fig. C.16. Algorithm running time when varying frac. of high-performance servers η ($C = 0.4 \cdot \# \text{nodes}$, $\lambda = 0.5$, $N_R = 100$, $l_{\max}^I = 20$, $l_{\max} = 128$).

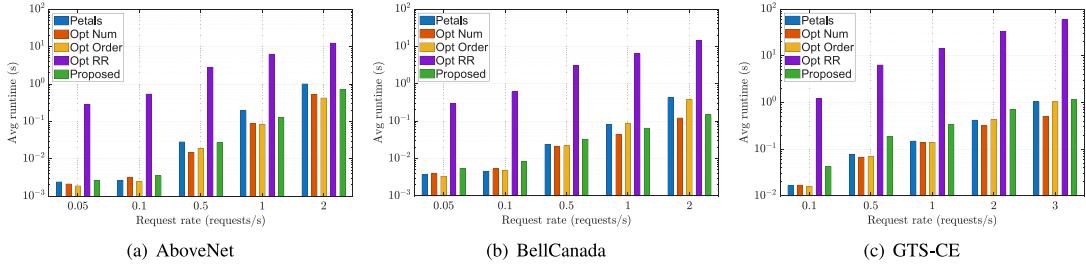


Fig. C.17. Algorithm running time when varying request rate λ with $N_R = 200 \cdot \lambda$ ($C = 0.4 \cdot \# \text{nodes}$, $\eta = 0.2$, $l_{\max}^I = 20$, $l_{\max} = 128$).

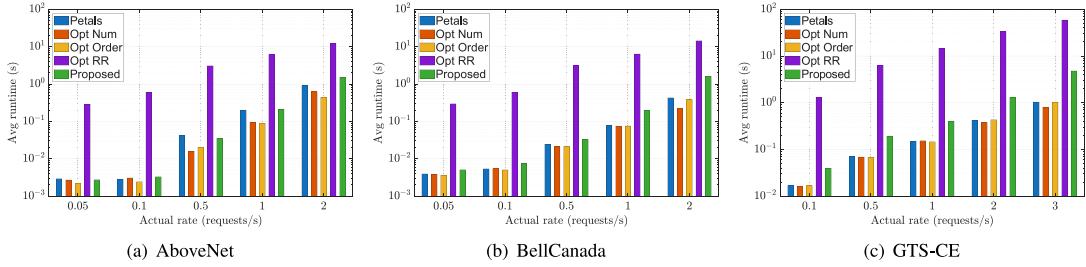


Fig. C.18. Algorithm running time when varying actual rate λ ($C = 0.4 \cdot \# \text{nodes}$, $\lambda_{\text{base}} = 0.5$, $N_R = 200 \cdot \lambda$, $\eta = 0.2$, $l_{\max}^I = 20$, $l_{\max} = 128$).

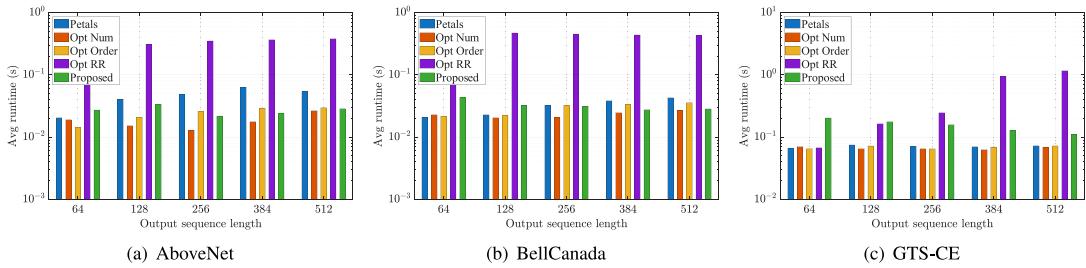


Fig. C.19. Algorithm running time when varying the sequence length l_{\max} ($C = 0.4 \cdot \# \text{nodes}$, $\eta = 0.2$, $\lambda = 0.5$, $N_R = 100$, $l_{\max}^I = 20$).

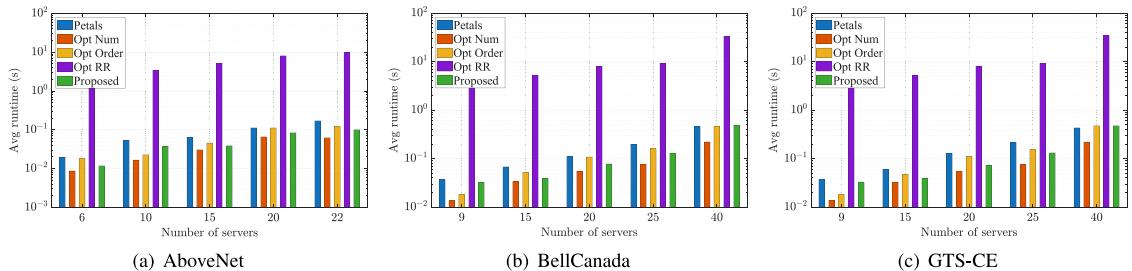


Fig. C.20. Algorithm running time when varying the number of servers C and rate $\lambda = (0.1/9) \cdot C$ ($N_R = 100$, $\eta = 0.2$, $l_{\max}^I = 20$, $l_{\max} = 128$).

References

- [1] Tom B. Brown, Benjamin Mann, Nick Ryder, et al., Language models are few-shot learners, in: International Conference on Neural Information Processing Systems, 2020, p. 25, Article 159.
- [2] Susan Zhang, Stephen Roller, Naman Goyal, et al., OPT: Open pre-trained transformer language models, 2022, arXiv preprint [arXiv:2205.01068](https://arxiv.org/abs/2205.01068), (2022).
- [3] Teven Le Scao, Angela Fan, Christopher Akiki, et al., BLOOM: a 176B-parameter open-access multilingual language model, 2023, [arxiv:2211.05100](https://arxiv.org/abs/2211.05100) [cs.CL], <https://arxiv.org/abs/2211.05100>.
- [4] Hugo Touvron, Thibaut Lavril, Gautier Izacard, et al., LLaMA: Open and efficient foundation language models, 2023, [arxiv:2302.13971](https://arxiv.org/abs/2302.13971) [cs.CL], <https://arxiv.org/abs/2302.13971>.
- [5] Hugo Touvron, Louis Martin, Kevin Stone, et al., Llama 2: Open foundation and fine-tuned chat models, 2023, [arxiv:2307.09288](https://arxiv.org/abs/2307.09288) [cs.CL], <https://arxiv.org/abs/2307.09288>.

- [6] NVIDIA, Ampere GA102 GPU architecture, 2020.
- [7] Feiyang Chen, Ziqian Luo, Lisang Zhou, Xuetong Pan, Ying Jiang, Comprehensive survey of model compression and speed up for vision transformers, 2024, arXiv preprint [arXiv:2404.10407](https://arxiv.org/abs/2404.10407), (2024).
- [8] Alexander Borzunov, Max Ryabinin, Artem Chumachenko, et al., Distributed inference and fine-tuning of large language models over the internet, *Adv. Neural Inf. Process. Syst.* 36 (2023) 12312–12331.
- [9] Yanping Huang, Youlong Cheng, Ankur Bapna, et al., GPipe: Efficient training of giant neural networks using pipeline parallelism, *Adv. Neural Inf. Process. Syst.* 32 (2019) https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf.
- [10] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, et al., PipeDream: generalized pipeline parallelism for DNN training, in: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.
- [11] Bowen Yang, Jian Zhang, Jonathan Li, et al., PipeMare: Asynchronous pipeline parallel DNN training, in: *Proceedings of Machine Learning and Systems*, 2021, pp. 269–296.
- [12] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, ImageNet classification with deep convolutional neural networks, *Commun. ACM* 60 (6) (2017) 84–90.
- [13] Tal Ben-Nun, Torsten Hoeft, Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, *ACM Comput. Surv.* 52 (4) (2019) 43, (Aug 2019), Article 65.
- [14] Zhenheng Tang, Shaoxuai Shi, Xiaowen Chu, Wei Wang, Bo Li, Communication-efficient distributed deep learning: A comprehensive survey, 2020, [CoRR abs/2003.06307](https://arxiv.org/abs/2003.06307), (2020).
- [15] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al., Attention is all you need, in: *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 6000–6010.
- [16] BigScience Workshop, Petals, 2025, Source code repository. <https://github.com/bigscience-workshop/petals>.
- [17] vLLM Project, vLLM: Distributed inference and serving, 2025, <https://docs.vllm.ai/en/stable/index.html>. (Accessed 01 September 2025).
- [18] A. Sprenger, Ray vLLM inference, 2025, https://github.com/asprenger/ray_vllm_inference. (Accessed 01 September 2025).
- [19] NVIDIA, A datacenter scale distributed inference serving framework, 2025, <https://github.com/ai-dynamo/dynamo>. (Accessed 01 September 2025).
- [20] Amazon Web Services, Amazon elastic kubernetes service, 2025, <https://aws.amazon.com/eks/>. (Accessed 01 September 2025).
- [21] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, Sujeeth Bharadwaj, Training large neural networks with constant memory using a new execution algorithm, 2020, *CoRR abs/2002.05645*, (2020).
- [22] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, et al., ZeRO-offload: Democratizing Billion-Scale model training, in: *2021 USENIX Annual Technical Conference*, USENIX ATC 21, 2021, pp. 551–564.
- [23] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, Yuxiong He, ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning, in: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, p. 14, Article 59.
- [24] Ahmed M. Medhat, Tarik Taleb, Asma Elmangoush, et al., Service function chaining in next generation networks: State of the art and research challenges, *IEEE Commun. Mag.* 55 (2) (2017) 216–223.
- [25] Weihan Chen, Xia Yin, Zhiliang Wang, Xingang Shi, Jianguan Yao, Placement and routing optimization problem for service function chain: State of art and future opportunities, in: Xingming Sun, Jinwei Wang, Elisa Bertino (Eds.), *Artificial Intelligence and Security*, Springer Singapore, Singapore, 2020, pp. 176–188.
- [26] Insun Jang, Dongeun Suh, Sangheon Pack, György Dán, Joint optimization of service function placement and flow distribution for service function chaining, *IEEE J. Sel. Areas Commun.* 35 (11) (2017) 2532–2541.
- [27] Qixia Zhang, Yikai Xiao, Fangming Liu, John C.S. Lui, Jian Guo, Tao Wang, Joint optimization of chain placement and request scheduling for network function virtualization, in: *IEEE 37th International Conference on Distributed Computing Systems*, ICDCS, 2017, pp. 731–741.
- [28] Linqi Guo, John Pang, Anwar Walid, Joint placement and routing of network function chains in data centers, in: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 612–620.
- [29] Xiaojun Shang, Zhenhua Liu, Yuanyuan Yang, Network congestion-aware online service function chain placement and load balancing, in: *Proceedings of the 48th International Conference on Parallel Processing*, 2019, p. 10, Article 46.
- [30] Maryam Jalalitabar, Yang Wang, Xiaojun Cao, Branching-aware service function placement and routing in network function virtualization, in: *IEEE Conference on Network Function Virtualization and Software Defined Networks*, NFDV-SDN, 2019, pp. 1–6.
- [31] Wenrui Ma, Oscar Sandoval, Jonathan Beltran, Deng Pan, Niki Pissinou, Traffic aware placement of interdependent NFV middleboxes, in: *IEEE Conference on Computer Communications*, INFOCOM, 2017, pp. 1–9.
- [32] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, Improving language understanding by generative pre-training, 2018, <https://api.semanticscholar.org/CorpusID:49313245>.
- [33] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, Language models are unsupervised multitask learners, 2019, <https://api.semanticscholar.org/CorpusID:16002533>.
- [34] Mediatech Group, The impact of prompt length on LLM performance: A data-driven study, 2025, <https://mediatech.group/prompt-engineering/the-impact-of-prompt-length-on-llm-performance-a-data-driven-study/>. (Accessed 01 September 2025).
- [35] Reddit /r/PromptEngineering, Shorter prompts lead to 40% better code generation by LLMs, 2025, https://www.reddit.com/r/PromptEngineering/comments/1d06wx4/shorter_prompts_lead_to_40_better_code_generation/. (Accessed 01 September 2025).
- [36] Alexander Borzunov, Dmitry Baranov, Tim Dettmers, et al., Petals: Collaborative inference and fine-tuning of large models, 2023, [arxiv:2209.01188](https://arxiv.org/abs/2209.01188).
- [37] Brian Hayes, Computing science: The easiest hard problem, *Am. Sci.* 90 (2) (2002) 113–117.
- [38] NVIDIA Corporation, NVIDIA multi-instance GPU user guide, 2025, <https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html>.
- [39] Daniel Schubert, Benedikt Jaeger, Max Helm, Network emulation using linux network namespaces, *Network* 57 (2019).
- [40] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, Matthew Roughan, The internet topology zoo, *IEEE J. Sel. Areas Commun.* 29 (9) (2011) 1765–1775.
- [41] Steven Gay, Pierre Schaus, Stefano Vissicchio, Repetita: Repeatable experiments for performance evaluation of traffic-engineering algorithms, 2017, arXiv preprint [arXiv:1710.08665](https://arxiv.org/abs/1710.08665), (2017).
- [42] Gurobi Optimization, LLC, Gurobi optimization, 2025, <https://www.gurobi.com/>. (Accessed 01 September 2025).