Scalable and Fast Inference Serving via Hybrid Communication Scheduling on Heterogeneous Networks

Gonglong Chen[1], Jiamei Lv[2], Kejiang Ye[1*], Tao Gu[3], and Chengzhong Xu[4]
[1]Shenzhen Institutes of Advanced Technology, CAS, Shenzhen, China.
[2]Zhejiang University, Hangzhou, China.
[3]Macquarie University, Sydney, Australia.
[4]University of Macau, Macau SAR, China.
Email: *gl.chen2@siat.ac.cn*, *lvjm@zju.edu.cn*, *kj.ye@siat.ac.cn*, *tao.gu@mq.edu.au*, *czxu@um.edu.mo*

***Abstract*—Advances in large language models (LLMs) have opened up new possibilities across various fields, fueling a new wave of interactive AI applications such as DeepSeek and ChatGPT. Inference serving systems play a crucial role in supporting these applications. Recent research indicates that when cross-server parallelization is enabled in inference serving systems, data synchronization overhead can exceed 65% of the total inference delay, making the reduction of communication overhead essential for speeding up inference. While existing systems accelerate cross-server communications by offloading synchronization operations to programmable switches, they often suffer from limited aggregation throughput under bursty traffic conditions, posing challenges for homogeneous network environments.**

To address these challenges, we propose HeroServe, an innovative inference serving system that leverages heterogeneous networks to accelerate data synchronization in distributed clusters. Our approach enables a fast and scalable inference serving system by employing an offline planner for joint computation allocation and communication scheduling, along with an online scheduler for dynamic traffic management and load balancing. We implement a prototype on a testbed comprising six servers and two programmable switches. Experimental results demonstrate that HeroServe improves scalability by 1.53× while achieving lower latency compared to state-of-the-art solutions.

I. Introduction

According to IDC projections, the global interactive AI market is expected to reach nearly $150 billion by 2027, with a compound annual growth rate (CAGR) of 85.7% [1]. Advances in large language models (LLMs) have opened up new possibilities across various fields, fueling a new wave of interactive AI applications such as DeepSeek [2] and ChatGPT [3]. Inference serving systems are vital for supporting LLM-based interactive AI applications [4], [5]. To retain tens of millions of daily users, these applications require low-latency inference and efficient concurrent request handling [6], [7].

Existing inference service systems improve efficiency through batch scheduling [8], instance deployment optimization [4], [7], and host memory fragmentation reduction [9]. However, these approaches typically focus on scenarios where models are deployed on a single GPU server or in configurations where multiple GPU servers are interconnected via high-bandwidth networks (e.g., InfiniBand [10]). Such networks are assumed to provide stable high-throughput transmission with negligible latency. In contrast, the work in [11] demonstrates that InfiniBand can also experience congestion under bursty or highly competitive traffic conditions, resulting in throughput drops exceeding 60% and doubling latency. Therefore, reducing communication overhead is critical for achieving faster inference speeds.

Recently, a family of In-Network Aggregation (INA) solutions has been proposed to accelerate cross-server communication in distributed systems [12], [13], [14], [15], [16], [17], [18], [19], [20]. These solutions achieve performance gains by offloading key communication behaviors (i.e., collective communication operations [21]) from GPU servers to programmable switches [12], resulting in fewer communication hops and reduced synchronization load. However, existing INA algorithms still suffer from low aggregation throughput in *homogeneous* networks (e.g., Ethernet) [22] when faced with bursty traffic that induces network congestion. For example, [22] shows that aggregation throughput can degrade by nearly 78% under bursty conditions, which severely impacting the scalability of inference serving systems.

To overcome these performance issues, we propose an innovative approach, HeroServe, which leverages *heterogeneous* networks (e.g., combining inter-server links like Ethernet with intra-server links such as NVLink) to accelerate cross-server communications in distributed clusters. Unlike existing approaches that rely solely on *homogeneous* networks, HeroServe exploits the strengths of high-bandwidth NVLink (e.g., 600GBs in A100 [23]) and 100Gbps Ethernet to substantially increase aggregation throughput, alleviate network congestion, and support faster, more scalable inference serving for a larger number of users (as evaluated in Section V).

However, designing an inference serving system over heterogeneous networks introduces significant challenges. **Challenge 1: joint optimization of computation allocation and communication scheduling in heterogeneous environments.** This problem is characterized by dynamic constraints (e.g., changes in available GPU memory during the allocation and release of model instances) and a high-dimensional decision space. For example, the computation resource allocation subproblem alone yields a search space on the order of

*Corresponding author.

$O(N!/(P_{tens}!)^{P_{pipe}})$[1], further complicated by factors such as shortest path selection for communication. To address this challenge, we propose a *scalability-oriented* offline planner that integrates asynchronous processing with heuristic algorithms. The planner formulates a comprehensive joint optimization model for both computation and communication. Heuristic strategies are employed to reduce the overall search space, and the global solution is fine-tuned through techniques such as random perturbations. Furthermore, multiple computations (e.g., constructing an offline matrix of node-to-node shortest paths) are scheduled asynchronously to further reduce problem-solving overhead.

Challenge 2: dynamic traffic management and load balancing. The dynamic, bursty nature of inference traffic leads to fluctuating loads on *heterogeneous* networks. In real time, selecting the best transmission path and communication scheme is difficult because of variable traffic volume and differing link utilization. To tackle this challenge, we propose a *load-aware* online scheduler that continuously monitors current traffic and updates lightweight, distributed tables tracking scheduling policy costs and shared link utilization. By dynamically adjusting the communication strategy and selecting the most favorable transmission routes, the online scheduler effectively balances network resources across both high speed intra-server links and inter-server Ethernet connections, reducing congestion and improving overall inference throughput and latency.

We prototype HeroServe on a testbed consisting of six servers and two programmable switches, and simulate it on large-scale clusters, demonstrating its capability to achieve the scalable and fast inference serving system at the same time (Section V). We evaluate HeroServe with production traces [24], [25]. In particular, we evaluate the end-to-end performance of HeroServe for OPT- 66B and OPT-175B [26] (an open-source LLM similar to the largest GPT-3 model) on NVIDIA GPUs. The experimental results show that HeroServe improves scalability by 1.53×, 1.42× and 1.33×, while maintaining the latency SLAs compared to state-of-the-art solutions DistServe [4], DS-ATP [12] and DS-SwitchML [13].

This paper makes the following contributions:

- We analyze existing inference serving systems for large language models and observe a key limitation that current in-network aggregation algorithms are primarily designed for *homogeneous* networks. This design choice leads to network congestion and lower aggregation throughput under bursty traffic conditions (§II).
- We design HeroServe, an inference serving system that accelerates data synchronization by leveraging *heterogeneous* networks. HeroServe employs an offline planner for joint computation allocation and communication scheduling together with an online scheduler for dynamic traffic management and load balancing (§III).
- We implement a HeroServe prototype on a testbed consisting of six servers and two programmable switches and evaluate it using production traces and large language models on NVIDIA GPUs. The experiments demonstrate that HeroServe significantly improves scalability and reduces latency compared to state-of-the-art solutions. (§IV and §V).

[1] N is the number of GPUs to be partitioned into P_{pipe} pipeline groups of equal size P_{tens} tensor GPUs, i.e., $N = P_{tens} \cdot P_{pipe}$.

II. Background and Motivations

In this section, we provide the background of the LLM inference and discuss the performance issues in existing works to motivate our work.

A. LLM Inference and Applications

LLM Inference. Large language model inference operates through the autoregressive Transformer architecture [27], where input prompts are first tokenized and processed sequentially. During each iteration, the model estimates a probability distribution over the vocabulary, selects the next token via greedy decoding or probabilistic sampling, and appends it to the growing sequence. This cyclic process continues until either an end-of-sequence marker is generated or a predefined maximum sequence length is reached, as documented in foundational studies [28], [29], [30].

LLM Applications. Large language model applications have demonstrated explosive growth, exemplified by ChatGPT's achievement of surpassing 100 million monthly active users within two months of its launch [3]. Enterprise deployments reveal concrete economic impacts, including up to 30% reductions in operational costs, 25%–40% improvements in process efficiency, and annual revenue enhancements reaching tens of millions in specific sectors [31]. Through prompt engineering that reframes diverse tasks (e.g., translation, summarization) as generative problems, LLMs are fundamentally transforming productivity paradigms and cost structures across multiple industries [32].

B. Inference Serving System

Transformer-based large language model inference involves two core steps: the prefill step and the decoding step [4]. In the prefill phase, the model processes the entire input prompt concurrently to compute intermediate representations and generate the first token. While in the decoding phase, the model sequentially produces subsequent tokens one-by-one [5]. Most existing systems colocate these two phases on the same GPU cluster using continuous batching [8]. However, this colocation introduces prefill-decoding interference because the compute-intensive prefill phase monopolizes GPU compute cycles and memory bandwidth, causing decoding tasks to wait for available resources and experience increased queuing delays. It negatively affect latency metrics, for example, increasing the time-to-first-token (TTFT) and time-per-output-token (TPOT) [4].

To address these issues, recent works [4], [7] have proposed architectures that decouple the prefill and decoding phases by deploying them on separate hardware resources. In these architectures, dedicated prefill instances efficiently process the input prompts, while separate decoding instances handle

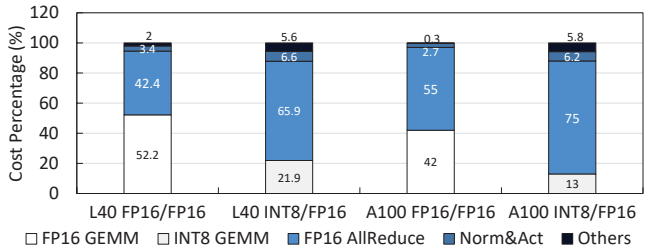
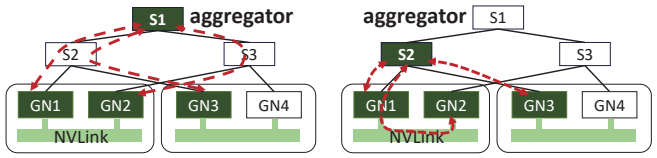


Fig. 1: Prefill cost breakdown of LLaMA-3-70B operations as measured by [33]. Tested on 4×L40/A100 GPUs (TP=4) with a batch size of 8, each with 1024 input and 64 output tokens. NCCL's Ring All-Reduce is applied. The notion of x-ticks (e.g. L40 FP16/FP16) denotes GPU type, model weight precision, and communication precision, respectively



(a) INA utilizes homogeneous networks (Ethernet).

(b) INA utilizes heterogeneous networks (NVLink and Ethernet).

Fig. 2: Comparison of INA over homogeneous and heterogeneous networks. S denotes the switch, GN denotes the GPU and the associated network NICs.

token-by-token generation. This separation allows for independent resource allocation and tailored parallelism strategies for each phase [4], [7].

Performance issues. A critical challenge in existing architectures, such as DistServe [4], is coping with the enormous parameter sizes of current LLMs. To store a large number of model parameters and support a large number of users to access cached data, many studies [21], [34] recommend deploying inference instances across multiple GPU servers. In such scenarios, communication overhead becomes a major contributor to inference latency. As shown in Fig. 1, when enabling parallelism across GPU servers (i.e., synchronization data are transmitted over 100Gbps Ethernet), the communication latency of all-reduce accounts for over 65% of the overall latency on L40 GPU [33], and the latency exceed 75% on A100 due its larger computation FLOPS. However, existing works [4], [7] only try their best to deploy the LLM inference instance on an indivisual GPU server as much as possible, which may sacrify the scalability. Communication issues across GPU servers are not properly resolved, leading to reduced throughput and degraded user experience.

C. In-Network Aggregation

To address the high communication overhead in large-scale distributed systems, researchers and industry practitioners have proposed various in-network aggregation (INA) schemes [12], [13], [14], [15], [16], [17], [18], [19], [20]. INA enables switches to combine packets (e.g., gradients or intermediate activation values) across workers (e.g., GPU), thereby reducing the data volume sent over the network and lowering both latency and bandwidth usage [12], [13].

Performance issues. Existing INA schemes [12], [13] still suffer from high latency because they assume that aggregated data is transmitted over homogeneous networks (e.g., Ethernet). Due to connectivity limitations, packets often traverse detour paths, resulting in extra delays. Fig. 2 illustrates the differences between homogeneous and heterogeneous networks for data aggregation. These GPU cards are coupled with RDMA NICs to enable the GPU Direct feature, which improves transmission efficiency and has been widely adopted in both industry and academia [35], [36], [37], [38]. In our example, each GPU server has two GPUs connected via high-throughput NVLink (e.g., 600GBs), while GPUs across servers use 100Gbps Ethernet links. We consider a scenario that three GPUs deploy the same model instance to perform all-reduce operations using INA. In a homogeneous network (Fig. 2(a)), the aggregation switch is the core switch S1. For 1MB of data, two hops of Ethernet links are required, resulting in an aggregation delay of approximately 160μs. In a heterogeneous network (Fig. 2(b)), GPUs use NVLink to forward data to an access switch S2 before traversing an Ethernet link. This path significantly reduces the delay to about 90μs, nearly 43% lower than the homogeneous solution. Moreover, offloading Ethernet traffic to NVLink further reduces congestion and improves throughput.

III. Design

A. Key Idea and Overview

The goal of HeroServe is to maximize scalability while minimizing the average token generation latency, subject to meeting the Service-Level Agreement (SLA). Scalability is defined as the number of requests served per second, meaning that a higher scalability implies the system can handle more concurrent requests. We target the prefill and decode disaggregated LLM inference architecture, which is widely adopted in many leading LLM inference systems [2], [4], [7], to fully leverage GPU resources and improve inference efficiency.

The key idea and insight. Unlike existing works that either deploy inference instances within GPU servers or reply solely on *homogeneous* networks to accelerate cross-server communications, we take a holistic view by optimizing the inference efficiency across both intra- and inter-GPU deployments in *heterogeneous* networks (e.g., Ethernet and NVLink). We jointly model computation and communication and propose a load-aware online scheduler that dynamically adjusts communication strategies (e.g., INA and ring) and transmission paths. This design effectively exploits *heterogeneous* links to enhance inference scalability and reduce token generation latency.

As shown in Fig. 3, the primary function of the **scalability-oriented offline planner** is to optimize computation allocation and communication strategies. It takes system status (e.g., network topology and etc) and user requirements (e.g., latency SLA) as inputs. Its goal is to maximize scalability while ensuring that latency remains within SLA constraints (see Section III-C). This optimization problem is inherently complex

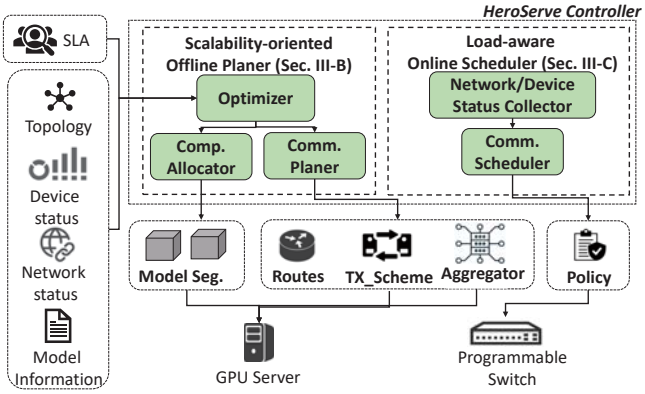


Fig. 3: The overview of HeroServe architecture.

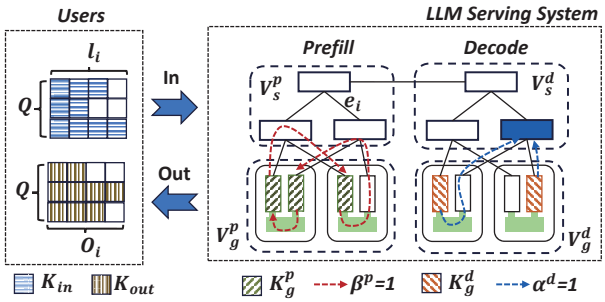


Fig. 4: The model of LLM serving systems.

because it involves dynamic constraints such as changes in available GPU memory as well as high-dimensional decision spaces. For instance, the GPU deployment subproblem alone has a search space on the order of $O(N!/(P_{tens}!)^{P_{pipe}})$, not to mention additional complexities such as shortest path selection. Since solving this problem in constant time is infeasible, we design a heuristic algorithm to address this challenge.

The **load-aware online scheduler** complements the planner by periodically adjusting the transmission policy. It dynamically switches communication schemes (e.g., INA or ring aggregation) and paths to balance traffic and mitigate congestion in heterogeneous networks. The policy takes the amount of synchronization data as input and selects the optimal communication scheme and path based on the estimated bandwidth overhead along each potential route. The decision is guided by a virtual bandwidth utilization ratio maintained for each path in a policy cost table (as detailed in Section III-D). A control center is provided to synchronize the virtual bandwidth utilization ratio across all GPUs.

B. System Model and Basic Notations

Fig. 4 presents the architecture of LLM serving systems. Where the prefill and decode components are deployed separately, as adopted in recent systems [2], [7], [4]. As shown in Fig. 4, each cluster has its own GPU cards, denoted by V_g^p/V_g^d, and access switches, denoted by V_s^p/V_s^d. Note that these GPU cards are coupled with RDMA NICs to enable the GPU Direct feature, which improves transmission efficiency and has been widely adopted in both industry and academia [35], [36], [37], [38]. These clusters communicate through a core switch,

TABLE I: Input parameters of the offline planer.

Input Parameter Name	Symbol
The number of model layers	L
The number of hidden layer	h
The number of heads	A
FFN intermediate size	m
Block size in the attention kernel	b
Model parameter size (B)	R
Input batch size	Q
Input length of each request (B)	$l_0,...,l_{Q-1}$
Input token size in the batch (B)	$K_{\text{in}} = \sum_{i=0}^{Q-1} l_i$
Output length of each request (B)	$O_0,...,O_{Q-1}$
Output tokens size (B)	$K_{\text{out}} = \sum_{i=0}^{Q-1} O_i$
Squared sum of the input lengths	$K_{\text{in2}} = \sum_{i=0}^{Q-1} l_i^2$
The rest of GPU memory capacity (GB)	$M_g = [M_1,\ldots,M_K]$
Network topology	$G = \langle V, E \rangle$
Node set includes switch set $V_s = \{V_s^p, V_s^d\}$ and GPU set $V_g = \{V_g^p, V_g^d\}$	$V = \{V_s, V_g\}$
Edge set	E
Edge bandwidth capacity (bps)	$C = [C(e_1),\ldots,C(e_n)]$
The rest of edge bandwidth (bps)	$B = [B(e_1),\ldots,B(e_n)]$
Request arrival rate (r/s)	λ
In-network aggregation entry size (B)	M_{ina}
SLA latency threshold (s)	$T_{\text{sla}}^{\text{pre}}$, $T_{\text{sla}}^{\text{dec}}$

TABLE II: Output parameters of the offline planer.

Output Parameter Name	Symbol
Parallel parameters include tensor parallel P_{tens}^p, P_{tens}^d, and pipeline parallel P_{pipe}^p, P_{pipe}^d	P_{all}
The set of prefill GPU IDs	K_g^p
The set of decode GPU IDs	K_g^d
The set of shortest connection path between k and a	$P_{(k,a)}$
The set of in-network aggregation switch includes the prefill switch V_{ina}^p, the decode switch V_{ina}^d	$V_{\text{ina}} = \{V_{\text{ina}}^p, V_{\text{ina}}^d\}$
The set of in-network aggregation selector includes the prefill selector α^p, the decode selector α^d	$\alpha = \{\alpha^p, \alpha^d\}$
The set of ring communication selector includes the prefill selector β^p, the decode selector β^d	$\beta = \{\beta^p, \beta^d\}$
The communication related parameters α, β, V_{ina}, $P_{(k,a)}$	CM

enabling independent optimization, maintenance and evolution of prefill and decode. This architectural separation allows system designers to tailor hardware to the specific needs of each cluster: the prefill cluster is compute-bound and benefits from severs with high computational throughput, whereas the decode cluster is memory-bound due to the large KV cache, favoring servers with ample memory capacity [7], [4].

The overall LLM serving system can be modeled as an undirected acyclic graph G as shown in Fig. 4. The graph contains a series of GPU nodes V_g^p/V_g^d and switch nodes V_s^p/V_s^d. The direct links between nodes are represented as edges E. Some edges correspond to Ethernet links, while others to NVLink connections. Each edge has a maximum bandwidth C and an available bandwidth B.

After running the offline planer (as detailed in Section III-C), the deployment locations of GPUs in the prefill and decode clusters, denoted by K_g^p and K_g^d, are obtained. For each parallel GPU group, we compute the current optimal communication scheme (i.e., selecting between INA α and ring communication β) and the corresponding path $P(k,a)$ (as listed in Table II).

At the user side, the size of each request is modeled as l_i. Given a future batch size Q, the total input token count, K_{in}, can be derived (as stated in Table I). Similarly, the total output token count, K_{out}, is estimated. To update these values over time, we utilize state information collected by the online scheduler module and apply a moving average method to dynamically update K_{in} and K_{out}.

C. Scalability-oriented Offline Planer

In this subsection, we provide a detailed description of how to model the aforementioned LLM inference tasks based on the system model introduced earlier. In the following expressions, the absence of superscripts p (prefill) and d (decode) indicates that the parameters have identical derivations for both the prefill and decode clusters.

1) Application Level Metrics: The optimization objective is to maximize the scalability, defined as the number of served requests per second, denoted by H. At the same time, the token generation latency TTFT (Time-To-First-Token) T_{pre} and the token production latency TPOT (Time-Per-Output-Token) T_{dec} must remain below their respective SLA thresholds. Formally, the objective is expressed as:

$$\begin{aligned} \max \quad & H = \frac{1}{T_{req}} \\ \text{subject to} \quad & T_{pre} \leq T_{sla}^{pre} \\ & T_{dec} \leq T_{sla}^{dec} \end{aligned} \tag{1}$$

Here, T_{req} is the latency to serve a single arriving inference request. It can be decomposed as $T_{req} = T_{queue} + T_{serve}$, where T_{queue} represents the queuing delay for arriving requests, and T_{serve} is the inference latency to generate sufficient tokens for each user's request.

We adopt a continuous batching approach. The batch size adapts to the volume of arriving requests. The adaptation scheme follows the existing work [8]. We assume that request arrivals follow a Poisson process. This assumption is justified by the high predictability of the execution times of LLM inference tasks, as detailed in [39], [40], [4]. The queuing delay is estimated using the Pollaczek–Khinchine equation [40]. When there is insufficient memory to serve all requests, an additional queuing delay is incurred. This delay is approximated by: $T_{queue} = \frac{\lambda T_{serve}^2}{2(1-\rho)}$. Where λ is the request arrival rate and $\rho = \lambda T_{serve}$ is the utilization ratio. The inference latency T_{serve} comprises three components: the communication latency T_n, the computation latency T_c, and the KV cache transfer latency T_f from the prefill cluster to the decode cluster. That is,

$$T_{serve} = T_n + T_c + T_f \tag{2}$$

The token generation latency T_{pre} is given by

$$T_{pre} = T_n^{pre} + T_c^{pre} \tag{3}$$

where T_n^{pre} and T_c^{pre} are the communication and computation delays among the parallel GPUs allocated for the prefill cluster, respectively.

The token production latency T_{dec} is defined as the delay between two consecutive output tokens. It includes the communication latency T_n^{dec} and computation latency T_c^{dec} among the parallel GPUs allocated for the decode procedure, along with the KV cache transfer latency T_f. Therefore,

$$T_{dec} = T_n^{dec} + T_c^{dec} + T_f \tag{4}$$

Furthermore, the overall communication latency is defined as $T_n = T_n^{pre} + T_n^{dec}$ and and the overall computation latency as $T_c = T_c^{pre} + T_c^{dec}$.

2) Metrics Modeling: For the **communication latency T_n,** it is determined by the chosen communication optimization strategy. It is modeled as:

$$T_n = T_{pp} + \sum_{s}^{S} T_m(s) \tag{5}$$

where T_{pp} is the synchronization latency among the parallelized pipeline clusters. This latency depends on the number of parallelized pipelines P_{pipe} and is given by $T_{pp} = \sum_{i=1}^{P_{pipe}-1} T_{PP}(i)$. Here, $T_{pp}(i)$ is the synchronization latency between the i-th and $(i+1)$-th pipeline clusters. In particular,

$$T_{pp}(i) = \min \max_{k \in K_g(i+1)} T_{k,a} \tag{6}$$

where $K_g(i+1)$ denotes the set of GPUs in the $(i+1)$-th pipeline group and $T_{k,a}$ represents the shortest latency from GPU k to GPU a. In the prefill cluster, the latency is expressed as $T_{k,a}^p = \sum_{n=1}^{N_h} K_{in} h / B(e_n)$. Where N_h is the number of hops from the GPU k to the GPU a. In the decode cluster, the expression is $T_{k,a}^d = \sum_{n=1}^{N_h} h / B(e_n)$.

The synchronization communication latency for the s-th step, $T_m(s)$, is computed as:

$$T_m(s) = \alpha(i) \cdot T_{ina}(s) + \beta(i) \cdot T_{ring}(s) \tag{7}$$

where the two binary variables $\alpha(i)$ and $\beta(i)$ select the synchronization method for the i-th GPU group (i.e., $\alpha(i) \in \{0,1\}$, $\beta(i) \in \{0,1\}$, and $\alpha(i) + \beta(i) = 1$). Here, $T_{ina}(s)$ and $T_{ring}(s)$are the latencies for the in-network aggregation (INA) and ring-based approaches, respectively.

The INA latency is decomposed into three phases:

$$T_{ina}(s) = T_{col}(s) + T_{agg}(s) + T_{dis}(s) \tag{8}$$

As introduced in Section II, the in-network aggregation procedure is detailed as three steps: data collection $T_{col}(s)$, parameter aggregation $T_{agg}(s)$, and data distribution $T_{dis}(s)$.

$$T_{col}(s) = \max_{k=1}^{P_{tens}} T_{k,a}^{col}(s), \forall X_{i,j,k} = 1 \tag{9}$$

$$T_{k,a}^{col}(s) = \sum_{e_n \in P_{(k,a)}} \left(\frac{D_{col}^k(s)}{B(e_n)}\right) \tag{10}$$

$T_{k,a}^{col}(s)$ denotes the data collection latency from the k-th GPU to the a-th aggregation switch for the s-th calculation step. $X_{i,j,k}$ specifies the deployment of the j-th tensor in the i-th layer of GPU k. Where $D_{col}^k(s)$ is the number of bits that must

be transferred in the s-th step. For a typical parallel inference, each layer involves two synchronization steps. The data volume can be represented as $D^k_{col} = [D_{col}(a), D_{col}(f)]$ according to [4], [41]. where $D_{col}(a)$ and $D_{col}(f)$ are the communication loads for the attention output and FFN layers, respectively, with $D_{col}(a) = D_{col}(f) = K_{in}h$. Additional parallelizable steps may be incorporated similarly.

We treat $T_{agg}(s)$ as a constant in the programmable switch (approximately $1\,\mu$s as reported in [42], [43]). The distribution latency $T_{dis}(s)$ and the corresponding data volume $D^k_{dis}(s)$ are configured similarly to $T_{col}(s)$ and $D^k_{col}(s)$.

$$T_{ring}(s) = 2(P_{tens} - 1)\frac{D^k_{rg}(s)}{\min_{e_n \in P_{(k,a)}} B(e_n)}, \forall e_n \tag{11}$$

The latency for the ring-based communication scheme is given above. In this expression, $D^k_{rg}(s)$ denotes the expected data volume for the s-th ring all-reduce step. It can be expressed as $D^k_{rg} = [D_{rg}(a) + D_{rg}(f)]$ with $D_{rg}(a) = D_{rg}(f) = K_{in}h/P_{tens}$. The bandwidth utilization constraint for the ring scheme is analogous to that in the previous equation, with $D^k_{col}(s)$ replaced by $D^k_{rg}(s)$.

For the **computation latency** T_c**,** it consists of two components: the prefill T^{pre}_c and the decode T^{dec}_c and can be inferred according to the existing works [4], [44]:

$$T^{pre}_c = \frac{C1}{P^p_{tens}}(4h^2K_{in} + 2hmK_{in}) + \frac{C2}{b \cdot P^p_{tens}}3hK_{in2} + C3 \tag{12}$$

$$T^{dec}_c = \frac{C4}{P^d_{tens}P^d_{pipe}}(4h^2 + 2hm) + \frac{C5}{P^d_{tens}P^d_{pipe}}3hK_{in} + C6 \tag{13}$$

Where $C1, C2, C4$, and $C5$ denote the linear fitting parameters for computational latency. $C3$ is used to quantify other overheads like Python Runtime, system noise and so on [4], while $C6$ denotes the overhead of filling the parallelizable pipeline in the decoding procedure [44]. Similar to the existing works, we use a profiling and interpolation approach to figure out the values of $C1$ to $C6$.

For the **KV cache transferring latency** T_f**,** it can be inferred as follows:

$$T_f = \max_{k \in K_p} T^p_k \tag{14}$$

where T^p_k is the maximum KV cache transmission latency for each prefill GPU $k \in K_p$. During KV cache computation, all prefill GPUs simultaneously transmit their KV caches to the associated decode GPUs (called the prefill/decode GPU pairs that are allocated with the same i-th model layer and j-th tensor segment). Due to different routing paths, each prefill/decode GPU pair incurs a different delay. Thus, the overall latency is determined by the longest delay.

For each prefill GPU k, the transmission latency is modeled as

$$T^p_k = \sum_{z \in K_d} \sum_{(i,j) \in R_{k,z}} T_{i,j}(k,z) \tag{15}$$

where $T_{i,j}(k,z)$ denotes the latency for transferring the KV cache corresponding to the i-th layer and j-th tensor segment from prefill GPU k to decode GPU z. This latency is the sum of the delays across all hops on the path from k to z: $T_{i,j}(k,z) = \sum_{h=1}^{H(k,z)} [D_{i,j}/B(e_h)]$. where $H(k,z)$ is the total number of hops, $D_{i,j}$ is the amount of data to be transferred for the i-th layer and j-th tensor segment. The data volume $D_{i,j}$ is given by $D_{i,j} = 2K_{in}h/A\lceil P_{tens}/A \rceil$.

Algorithm 1: Scalability-oriented Offline Planer.

Input : Parameters shown in Table I.
Output: Parameters shown in Table II.

```
max_H=0;
candi = gen_tp_pp_candi(V_g, R, M_g, R_frac, max_candi);
foreach P_all ∈ candi do
    thread process_prefill_cluster
        m_req = R/(P^p_tens · P^p_pipe · R_frac);
        V_g^p' ← del GPU in V_g^p with memory < m_req;
        if len(V_g^p') < P^p_tens · P^p_pp then
            Continue to the next configuration;
        CM, K_g, T_n^pre=est_network_latency(P_all, V_g', K_in);
        T_c^pre= est_compute_latencyp(P_all, K_in);
    thread process_decode_cluster
        m_req = R/(P^d_tens · P^d_pipe · R_frac);
        V_g^d' ← del GPU in V_g^d with memory < m_req;
        if len(V_g^d') < P^d_tens · P^d_pp then
            Continue to the next configuration;
        CM, K_g, T_n^dec=est_network_latency(P_all, V_g', K_out);
        T_c^dec= est_compute_latencyd(P_all, K_in, K_in2, K_out);
    T_f=est_kvtrans_latency(K_in, A, P_all);
    Update T_pre, T_dec, H based on Eq (1), (4), (5);
    if T_pre ≤ T^pre_sla and T_dec ≤ T^dec_sla and H > max_H then
        Update output parameters;
```

3) Solving the Problem: We observe that after modeling the above LLM inference task, the parameter space in Table II is too large to solve in constant time. To address this issue, we simplify the problem using the following heuristic strategies while minimizing their impact on the solution: 1) For **communication latency**, we first offline compute the pairwise shortest path matrix $P_{(k,a)}$ and minimum latency matrix $D_{(i,j)}$ for all nodes. Next, we group GPUs for tensor parallelism based on interconnection latency using a clustering method. Finally, random perturbations are utilized to further improve the efficiency. 2) For **computational latency**, we compute the minimum required GPUs for P_{all} combinations based on Q and M_g, associated with an empirical upper bound. 3) For **KV cache transfer latency**, we use the offline computed minimum latency matrix $D(i,j)$ and solve for the latency according to Equation 14.

Algorithm 1 presents the procedure of scalability-oriented offline planer. **1) Determine the minimum GPUs.** We calculate the minimum number of GPUs needed for inference based on the model parameter memory R and the reserved memory ratio R_frac at each server: $R/\sum_{k=1}^{K} M_g(k) R\_frac$. Using this minimal count, we generate combinations of P_{pipe} and P_{tens}, returning up to max_candi candidate configurations. (Our experiments show that setting max_candi = twenty usually

Algorithm 2: Estimate network latency.

Input : P_{tens}, P_{pp}, $V_g^{'}$, $K_{in/out}$.
Output: CM, K_g, T_n.

case bandwidth_utilization_is_update
  $D_{(i,j)}$ = gen_latency_matrix($V_g^{'}$, alg=dijkstra);
  $P_{(k,a)}$ = store_shortest_path($V_g^{'}$, alg=dijkstra);
K_g = group_gpu($V_g^{'}$, P_{pp}, $D_{(i,j)}$, alg=k-means-constrained);
Initialize $T_e[i]$ for each group to zero;
foreach *group* $\in K_g$ **do**
  Find V_s with the smallest delay to the group while meeting memory constraints.;
  group.append(V_s); V_{ina}.append(V_s);
foreach *group* $\in K_g$ **do**
  $T_e[group\_id]$ = getlatency(group, $D_{(i,j)}$, $K_{in/out}$);
  Store path for each GPU in group based on $P_{(k,a)}$;
foreach *group* $\in K_g$ **do**
  improvement $\leftarrow$ true;
  while *improvement* **do**
    improvement $\leftarrow$ false;
    g_tmp $\leftarrow$ random.select_group(cluster);
    group', g_tmp' $\leftarrow$ Randomly swap group and g_tmp;
    newLatency = getlatency(group', $D_{(i,j)}$, $K_{in/out}$);
    if $newLatency < T_e[group\_id]$ **then**
      T_e[group_id] = newLatency;
      Update K_g with group' and g_tmp';
      improvement $\leftarrow$ true;
Estimate (P_{pp}-1) inter group latency T_i[];
T_n = sum(T_e) + sum(T_i);
procedure getlatency(group, $D_{(i,j)}$**,** $K_{in/out}$**)**
  T_{ina} = compute_ina_latency(group, $D_{(i,j)}$, $K_{in/out}$);
  T_{ring} = compute_ring_latency(group, $D_{(i,j)}$, $K_{in/out}$);
  if $T_{ina} > T_{ring}$ **then**
    β.append(group_id); return T_{ring};
  else
    α.append(group_id); return T_{ina};

yields near-optimal solutions.) **2) Estimate overheads.** Two threads run simultaneously to compute the computation and communication overheads. Both processes are similar and differ only in the input parameters. We invoke the network overhead estimation function (Algorithm 2) and compute the overheads using Equations 12 and 13. Given the sending and receiving nodes, Dijkstra's algorithm is applied to compute the KV cache transfer latency between the prefill and decode clusters. **3) Select the optimal configuration.** We return the configuration that meets the SLA latency requirements while maximizing throughput. Experimental results indicate that our algorithm typically finds a solution within 10 minutes, a reduction of 28.57% compared to DistServe [4].

Algorithm 2 estimates the network latency as follows: **1): GPU grouping.** We partition all GPUs into P_{pipe} groups, each containing P_{tens} GPUs using a k-means-constrained approach [45]. **2) Communication mode selection.** We compute the communication latency for both the INA and ring schemes using Equation 8 and Equation 11 and choose the mode with the lower latency. **3) Perturbation scheme.** To avoid local optima, GPUs are randomly swapped between groups, and the communication latency is recalculated. If a swap reduces latency, the new assignment is kept. Our experiments show that the algorithm typically converges within five iterations.

D. Load-aware Online Scheduler

To improve inference throughput and minimize token generation latency, it is crucial to evenly distribute request traffic across heterogeneous networks (e.g., Ethernet and NVLink) to maximize bandwidth utilization and mitigate congestion. To achieve this, we propose a load-aware online scheduler that takes the number of tokens (including input and generated tokens) as input and dynamically adjusts the communication scheme (INA and ring) and transmission path to distribute traffic efficiently.

A policy c is defined as a set of routing configurations, e.g., the transmission scheme (INA or ring), the next hop, the transmission path and etc. Given the observed transmission data D, we define the policy cost function $J(c,D)$ as the maximum bandwidth utilization ratio among all transmission links involved with c. For example, a policy using the INA scheme may have GPU1 and GPU2 transmit data to an aggregation switch via two separate paths, with the cost being the higher utilization ratio of these paths. Then the optimization goal can be formulated as follows:

$$c^* = \arg\min_{c \in C} J(c,D) \tag{16}$$

The optimal policy c^* is defined as the one that minimizes the policy cost that is expressed as $J(c,D) = b_c + \delta$. Here, b_c represents the previous cost of the policy c. The term δ, calculated as $D/(T_u b_c)$, represents the estimated additional bandwidth utilization when the transmission task is allocated to policy c, and T_u is the estimation window.

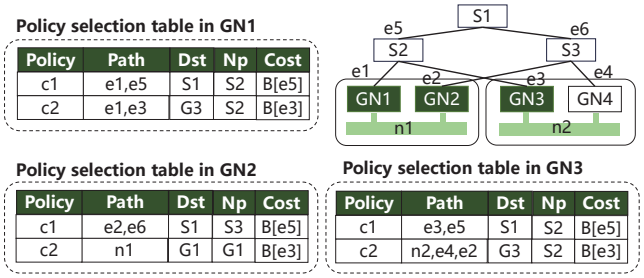
After the optimal policy c^* is selected, the increased bandwidth from data transmission updates all related policy as shown below:

$$b_c^{'} = b_c + \begin{cases} \dfrac{D}{T_u b_c}, & \text{if } c = c^*, \\ \dfrac{D}{T_u b_c} \cdot f_{(c^*,c)}, & \text{if } c \neq c^*. \end{cases} \tag{17}$$

The load penalty function $f_{(c^*,c)}$ quantifies the impact of selecting c^* on unselected policies c. Since selected and unselected policies may share intersecting links, the added load on c^* increases the overhead on its edges, which in turn adds traffic on the shared links of unselected policies. As $f_{(c^*,c)}$ depends on the shared links among multiple paths, it is updated periodically based on the following formula:

$$f_{(c^*,c)} = (1-\gamma) \cdot f_{(c^*,c)} + \gamma \cdot W_{(c^*,c)} \tag{18}$$

Here, $W_{(c^*,c)} = \sum_{e^* \in c^* \cap c} B(e^*) / \sum_{e \in c} B(e)$ represents the sharing ratio between policies c^* and c, considering the network topology and the bandwidth utilization of intersecting links $B(e^*)$, which are monitored by GPUs and programmable switches. The parameter γ is a smoothing factor that controls the update speed of the penalty function.



Policy	Path	Dst	Np	Cost
c1	e1,e5	S1	S2	B[e5]
c2	e1,e3	G3	S2	B[e3]

Policy	Path	Dst	Np	Cost
c1	e2,e6	S1	S3	B[e5]
c2	n1	G1	G1	B[e3]

Policy	Path	Dst	Np	Cost
c1	e3,e5	S1	S2	B[e5]
c2	n2,e4,e2	G3	S2	B[e3]

Fig. 5: The example of policy selection table stored in GPUs. Np denotes the next hop to the destination. c1 denotes the policy of using INA while c2 denotes the policy of using ring.

Figure 5 illustrates a policy cost table stored on GPUs. The table details how GPUs select transmission paths and schemes and how they synchronize these selections. When the NCCL `ncclAllreduce` function is called, each GPU (e.g., $GN1$, $GN2$, and $GN3$) selects the lowest-cost policy from the table. This policy prioritizes its corresponding route and ensures the underlying layer executes the appropriate forwarding entry. In this example, suppose $B[e_5]$ is lower than $B[e_3]$, and policy $c1$ is selected. Next, all GPUs report their selection results to the centralized controller HeroServe. The controller instructs all GPUs (e.g., $GN1$, $GN2$, $GN3$, and $GN4$) to update their policy cost tables synchronously according to Equation 17. These actions are triggered periodically when the `ncclAllreduce` function is executed.

IV. Implementation

We prototype HeroServe with a centralized scheduler and agents on both GPU servers and switches to coordinate model deployment and enforce the online transmission strategy adaptation. The implementation comprises over 5.3K lines of Python code for the controller, over 400 lines of P4 code for the programmable switch data plane, and over 2K lines of C++ code on GPU servers. The GPU-side implementation is built on top of SwiftTransformer [46] which supports high performance model and pipeline parallelism.

Agent on Programmable Switches. *1) Data Plane.* In our design, the data plane implements a synchronous in-network aggregation (INA) mechanism. The aggregation memory space is organized as a pool of fixed-size aggregator slots across multiple switch pipelines. *aggregation_table* is an exact–match table with keys based on the port and an aggregator ID (or index) is used to map incoming INA update packets to corresponding aggregator slots. The value field stores a partially aggregated vector (whose elements are represented as fixed–point integers) and a counter indicating the number of contributions received.

2) Control Plane. The central scheduler uniformly allocates and recycles aggregator slots. The switch control plane provides APIs that allow for high–speed updates of the aggregation table entries via vendor–provided low–latency runtime libraries (e.g., using the switch's native runtime API) [43]. It periodically polls hardware counters from the data plane to obtain link utilization metrics. These statistics are then used to update the cost parameters in the online scheduling process.

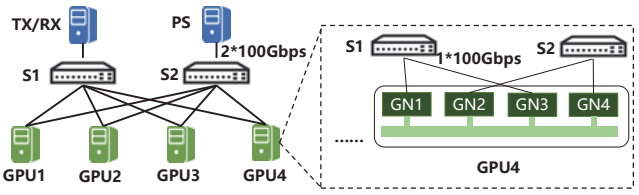


Fig. 6: Testbed. GPU servers and switches are connected with 2tracks.

Agent on GPU Servers. On GPU servers, an agent operates a load-aware online scheduler that dynamically updates route costs using lightweight vectorized operations (e.g., NumPy) based on the current batch size and token generation requirements. It then selects the optimal transmission mode (INA or ring-based) and embeds this decision in the packet headers. Additionally, the agent retrieves the bandwidth utilization ratio of NVLink via the DCGM (Data Center GPU Manager) tool [47].

Central Scheduler. The central scheduler is implemented as a Python application that periodically aggregates static topology data (e.g., GPU server configurations and switch capacities) and dynamic network performance metrics from hardware counters and NIC monitors. After solving the optimization problem, the scheduler disseminates the computed policies (including aggregator assignments, routing cost baselines, and transport mode preferences) via high–speed gRPC to agents on programmable switches and GPU servers, enabling a centralized control–plane update loop that rapidly adapts to load variations and ensures sustained high throughput and low latency across the distributed inference serving system.

V. Evaluation

We evaluate HeroServe through a combination of small-scale testbed experiments and large scale simulations, and comparing it against three of the most relevant existing solutions, i.e., DistServe [4], DistServe-ATP [12], and DistServe-SwitchML (DS-SwitchML) [13]. Among them, DS-ATP and DS-SwitchML represent the integration of ATP asynchronous INA and SwitchML synchronous INA solutions into DistServe, respectively, to improve the efficiency of network transmission. All algorithms are evaluated based on the prefill/decode disaggregated architecture, and all enable continuous batch processing technology to improve inference throughput.

Testbed Deployment. As illustrated in Fig. 6, the testbed comprises six servers and two programmable switches equipped with Tofino 1 ASIC [42]. One server acts as the PS, while another server acts as the transmission/reception server that replays the traffic of ShareGPT [24] and LongBench [25]. The remaining four GPU servers (e.g., two A100 servers and two V100 servers) serve as workers to hosting inference instances. Each GPU server contains four GPU cards with NVLinks enabled. A100 has 40GB memory while V100 has 32GB memory. Each server is equipped with two Mellanox

ConnectX-6 100G dual-port NICs that have four 100Gbps ports in total, which indicates that each GPU card can have different Ethernet links to the switch. The GPU network card of each server is cross-connected with the uplink switch to achieve high-availability deployment (as shown in Fig. 6). Fig. 6 shows the connection scheme of 2tracks, where x in $x$tracks represents the number of access switches.

Simulation Settings. We perform simulations on a physical server equipped with an Intel Core i9-9900K processor and an NVIDIA GeForce RTX 2080 Ti GPU. We utilize a high fielity deep learning simulator APEX [48] to perform the large scale simulation. It supports most common parallelization strategies and provides a enhance memory system modeling to support accurate modeling of in-network collective communication and disaggregated memory systems. In our simulation, we deployed 1200 A100 GPU servers (each equipped with 8 GPUs) to evaluate two network configurations. The first, the 2tracks configuration, utilizes GPU units of 6 servers connected by 2 access switches, amounting to 400 access switches and 27 core switches in total. The second, the 8tracks configuration, employs GPU units of 16 servers connected via 8 access switches, with an overall deployment of 600 access switches and 280 core switches, simulating a scenario with more evenly distributed traffic across a larger number of switches.

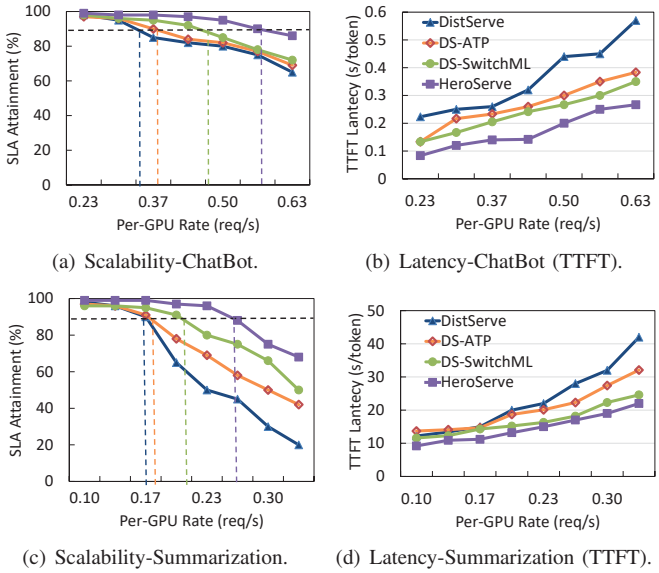
Model and workloads setup. Similar to prior work on LLM serving [4], we choose the OPT [26] model series, which is a representative LLM family widely used in academia and industry. We use FP16 precision in all experiments. We use the ShareGPT dataset [24] for the chatbot application (with the SLA of 2.5s TTFT and 0.15s TPOT) and the LongBench [25] dataset for the summarization application (with the SLA of 15s TTFT and 0.15s TPOT) to test the OPT-66B model on the testbed. For large scale simulation, we test the above chatbot (with the SLA of 4s TTFT and 0.2s TPOT) and summarization (with the SLA of 25s TTFT and 0.2s TPOT) on OPT-175B model under different switch tracks settings (i.e., 2tracks and 8tracks). Since all the datasets do not include timestamps, we generate request arrival times using a Poisson distribution with different request rates.

Results reveal that:

- HeroServe achieves high scalability: it is 1.53×, 1.42× and 1.33× better than DistServe, DS-ATP, and DS-SwitchML respectively.
- HeroServe achieves lowest latency, it significantly reduces per-token latency (TPOT) by about 18.6%-49.2% compared to its counterparts.
- HeroServe can consistently achieve the highest in-network aggregation throughput varying message size from 4MB to 64MB.

A. Testbed Experiments

Scalability. The SLA attainment represents the fraction of requests that meet the latency SLA. In our evaluation, we focus on the maximum per-GPU rate that the system can handle while satisfying the latency requirements for over 90% of requests. As shown in Fig. 7(a) and Fig. 7(c), HeroServe achieves superior scalability than existing solutions in both chatbot and summarization scenarios. Specifically, in the chatbot scenario, HeroServe is 1.53×, 1.42× and 1.33× better than DistServe, DS-ATP, and DS-SwitchML, respectively; while in the summarization scenario, it is 1.68×, 1.58× and 1.35× better than these baselines. This improvement is primarily attributed to its optimal model partitioning and dynamic allocation strategy that leverages both high-bandwidth NVLink and Ethernet for inter-server communication, effectively reducing bottlenecks under high request loads.



(a) Scalability-ChatBot. (b) Latency-ChatBot (TTFT).

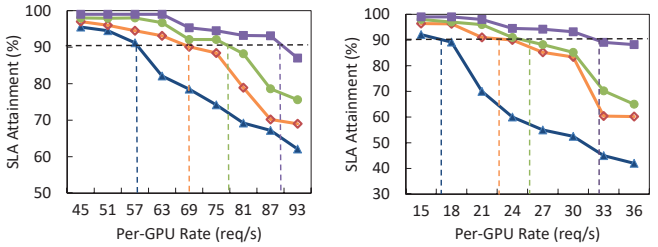
(c) Scalability-Summarization. (d) Latency-Summarization (TTFT).

Fig. 7: Testbed. Scalability and latency over OPT-66B.

Latency. As shown in Fig. 7(b) and Fig. 7(d), HeroServe reduces per-token latency (TPOT) by approximately 18.6%–49.2% compared to its counterparts. This improvement is largely due to its ability to minimize communication delays through integrated, optimized cross-GPU routing over heterogeneous networks, effectively reducing inference synchronization time. In the summarization scenario, although a looser TTFT SLA is allowed due to longer inputs, HeroServe still reduces initial delay (TTFT) by roughly 15.2%–45.2% and the per-token delay (TPOT) by around 11.2%–27.3%, ensuring rapid token generation even when processing lengthy texts.

B. Simulation Experiments

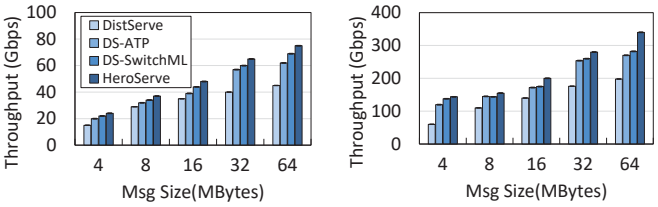
Scalability and Latency. As shown in Fig. 8, HeroServe boosts scalability by approximately 1.09×-1.83× in the 8tracks scenario. In larger model deployments, where more GPUs are used and inter-GPU communication demands, the benefits of HeroServe are even more pronounced than in the OPT-66B scenario. Additionally, HeroServe reduces the per-token delay by roughly 28.4%–42.1%, a crucial improvement given the cumulative delays in large-scale models. In the 2tracks scenario, scalability improvements reach 1.12×-1.94× compared to its counterparts, as the limited Ethernet bandwidth causes DS-ATP and DS-SwitchML—relying solely on Ether-



(a) ChatBot-2tracks. (b) Summary-2tracks.

(c) ChatBot-8tracks. (d) Summary-8tracks.

Fig. 8: Simulation. Scalability with various track settings over OPT-175B.



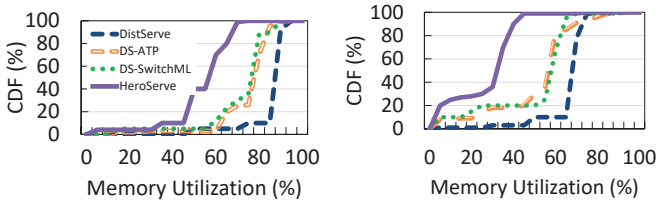
(a) 2 tracks. (b) 8 tracks.

Fig. 9: Simulation. In-network Aggregation.

net for synchronization—to suffer from increased congestion and degraded performance.

In-network Aggregation Throughput. As shown in Fig. 9, HeroServe achieves the highest in-network aggregation throughput compared to existing approaches. Specifically, in the 2tracks scenario, HeroServe improves throughput by 71.7%, 26%, and 20.1% over DistServe, DS-ATP, and DS-SwitchML, respectively. This performance is mainly due to its heterogeneous communication scheduling strategy, which leverages additional network bandwidth to transfer data more efficiently.

Memory Efficiency of Storing KV Cache. As depicted in Fig. 10, HeroServe consistently maintains the lowest memory utilization in both 2tracks and 8tracks scenarios. Its high transmission efficiency results in more frequent KV cache refreshes, reducing memory usage. Additionally, an online algorithm adaptively adjusts the communication mode and path based on the request load, evenly distributing traffic across the heterogeneous network, reducing congestion, and expediting data forwarding. This approach keeps the number of concurrently processed user requests in memory at a lower level.



(a) 2 tracks. (b) 8 tracks.

Fig. 10: Simulation. Memory efficiency, summarization, OPT-175B, 0.07 req/s.

VI. Related Work

Inference Serving Systems. There has been plenty of work on inference serving recently [2], [5], [7], [4], [49], [50], [51], [8], [6] Among them, DistServe [4] and SplitWise [7] improve the performance of large language models (LLMs) serving by disaggregating the prefill and decoding computation, eliminating prefill-decoding interferences. AlpaServe [49] focuses on employing model parallelism to statistically multiplex the GPU execution thus improving the resource utilization. However, existing systems lack optimization for tensor-parallel communication, leading to significant delays. Deploying LLMs on extensive GPU clusters exacerbates latency and throughput issues. In contrast, HeroServe leverages heterogeneous communication links and unified computation-communication modeling to markedly enhance distributed deployment performance, reduce token generation delays. FastServe [6] uses preemptive scheduling to minimize latency with a skip-join Multi-Level Feedback Queue scheduler. FlashCommunication [33] proposes a low-bit compression technique designed to alleviate the tensor parallelism communication bottleneck during inference. Orca [8] proposes a iteration-level scheduling that schedules execution at the granularity of iteration (instead of request) where the scheduler invokes the execution engine to run only a single iteration of the model on the batch. They are all orthogonal to my work.

In-network Aggregations. Recent works on In-network Aggregations (INA) can be classified into three categories. First, switch offloading techniques (e.g., ATP[12], SwitchML[13], PA-ATP[14], ZEBRA[52], DSA[53], NetReduce [20], ASK [18]) accelerate training by offloading gradient aggregation to programmable switch dataplanes, incorporating approaches such as progress-aware transmission and priority-based preemption. Second, network-aware scheduling and resource management approaches (e.g., NetPack [16], INAlloc [19]) optimize job placement by leveraging dynamic estimation and precise valuation of compute, bandwidth, and switch memory resources. Third, INA routing and system frameworks (e.g., In-Go [17], AggTree [54], FreeINA [55], NetRPC [56], ClickINC [15]) streamline in-network communication through refined routing algorithms, batch size adjustments, and unified programming abstractions for INC-enabled applications. However, all these designs target Ethernet-based INA transmissions without considering the challenges of heterogeneous networks (i.e., the NVLink and Ethernet), leaving room for HeroServe to significantly improve the LLM inference scalability and

reducing the token generation latency.

VII. Conclusion and Future Works

This paper introduces HeroServe, an inference serving system that accelerates data synchronization by leveraging *heterogeneous* networks. HeroServe employs a scalability-oriented offline planner that formulates a comprehensive optimization model to jointly consider computation allocation and communication scheduling. To deal with the dynamic inference traffic, we propose a load-aware online scheduler that continuously monitors current traffic and updates lightweight. We prototype HeroServe on a testbed consisting of six servers and two programmable switches. Experimental results show that HeroServe improves scalability by 1.53× while maintaining the latency SLA compared to the state-of-the-art solution.

In the future, there are several avenues to explore. First, for scenarios without NVLink, we will investigate how to leverage high-performance PCIe bandwidth for intra-server communication while avoiding performance degradation due to cross-NUMA effects. Additionally, we plan to design a mechanism that enables rapid scaling in and out to achieve finer-grained scheduling of computational resources.

VIII. Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 92267105), Guangdong Basic and Applied Basic Research Foundation (No. 2023B1515130002), Guangdong Special Support Plan (No. 2021TQ06X990), Key Research and Development and Technology Transfer Program of Inner Mongolia Autonomous Region (2025YFHH0110), Shenzhen Basic Research Program (No. JCYJ20220818101610023, KJZD20230923113800001).

References

[1] I. D. Corporation, "Worldwide ai and generative ai spending guide," 2023. [Online]. Available: https://www.idc.com/getfile.dyn?containerId=IDC_P33198&attachmentId=47522841

[2] DeepSeek-AI, D. Guo, and etc., "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[3] T. Guardian, "Chatgpt reaches 100 million users two months after launch," 2023. [Online]. Available: https://www.theguardian.com/technology/2023/feb/02/chatgpt-100-million-users-open-ai-fastest-growing-app

[4] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "Distserve: disaggregating prefill and decoding for goodput-optimized large language model serving," in *Proceedings of USENIX OSDI*, 2024.

[5] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve," in *Proceedings of USENIX OSDI*, 2024.

[6] B. Wu, Y. Zhong, Z. Zhang, S. Liu, F. Liu, Y. Sun, G. Huang, X. Liu, and X. Jin, "Fast distributed inference serving for large language models," *arXiv preprint arXiv:2305.05920*, 2024.

[7] P. Patel, E. Choukse, C. Zhang, A. Shah, I. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," in *Proceedings of ACM/IEEE ISCA*, 2024.

[8] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for Transformer-Based generative models," in *Proceedings of USENIX OSDI*, 2022.

[9] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of ACM SOSP*, 2023.

[10] NVIDIA, "Nvidia connectx infiniband adapters," 2025. [Online]. Available: https://www.nvidia.com/en-us/networking/infiniband-adapters/

[11] Y. Zhang, Y. Liu, Q. Meng, and F. Ren, "Congestion detection in lossless networks," in *Proceedings of ACM SIGCOMM*, 2021.

[12] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift, "ATP: In-network aggregation for multi-tenant learning," in *Proceedings of USENIX NSDI*, 2021.

[13] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling distributed machine learning with In-Network aggregation," in *Proceedings of USENIX NSDI*, 2021.

[14] Z. Li, J. Huang, T. Zhang, S. Zhou, Q. Wang, Y. Li, J. Liu, W. Jiang, and J. Wang, "Pa-atp: Progress-aware transmission protocol for in-network aggregation," in *Proceedings of IEEE ICNP*, 2023.

[15] W. Xu, Z. Zhang, Y. Feng, H. Song, Z. Chen, W. Wu, G. Liu, Y. Zhang, S. Liu, Z. Tian, and B. Liu, "Clickinc: In-network computing as a service in heterogeneous programmable data-center networks," in *Proceedings of the ACM SIGCOMM*, 2023.

[16] B. Zhao, W. Xu, S. Liu, Y. Tian, Q. Wang, and W. Wu, "Training job placement in clusters with statistical in-network aggregation," in *Proceedings of ACM ASPLOS*, 2024.

[17] J. Bao, G. Zhao, H. Xu, H. Wang, and P. Yang, "Ingo: In-network aggregation routing with batch size adjustment for distributed training," in *Proceedings of IEEE/ACM IWQoS*, 2024.

[18] Y. He, W. Wu, Y. Le, M. Liu, and C. Lao, "A generic service to provide in-network aggregation for key-value streams," in *Proceedings of ACM ASPLOS*, 2023.

[19] B. Zhao, C. Liu, J. Dong, Z. Cao, W. Nie, and W. Wu, "Enabling switch memory management for distributed training with in-network aggregation," in *Proceedings of IEEE INFOCOM*, 2023.

[20] S. Liu, Q. Wang, J. Zhang, W. Wu, Q. Lin, Y. Liu, M. Xu, M. Canini, R. C. C. Cheung, and J. He, "In-network aggregation with transport transparency for distributed training," in *Proceedings of ACM ASPLOS*, 2023.

[21] R. Y. Aminabadi, S. Rajbhandari, M. Zhang, A. A. Awan, C. Li, D. Li, E. Zheng, J. Rasley, S. Smith, O. Ruwase, and Y. He, "Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale," *arXiv preprint arXiv:2207.00032*, 2022.

[22] N. Gebara, M. Ghobadi, and P. Costa, "In-network aggregation for shared machine learning clusters," in *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[23] "Nvidia a100," 2025. [Online]. Available: https://www.nvidia.com/en-us/data-center/a100/

[24] "Sharegpt teams," 2023. [Online]. Available: https://sharegpt.com/

[25] Y. Bai, X. Lv, and etc., "Longbench: A bilingual, multitask benchmark for long context understanding," *arXiv preprint arXiv:2308.14508*, 2024.

[26] S. Zhang, S. Roller, and etc., "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of NIPS*, 2017.

[28] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Proceedings of NIPS*, 2020.

[29] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *Proceedings of NIPS*, 2022.

[30] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.0836*, 2020.

[31] G. Xexéo, F. Braida, M. Parreiras, and P. Xavier, "The economic implications of large language model selection on earnings and return on investment: A decision theoretic model," *arXiv preprint arXiv:2405.17637*, 2024.

[32] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, Jan. 2023.

[33] Q. Li, B. Zhang, L. Ye, Y. Zhang, W. Wu, Y. Sun, L. Ma, and Y. Xie, "Flash communication: Reducing tensor parallelization bottleneck for fast large language model inference," *arXiv preprint arXiv:2412.04964*, 2024.
[34] Z. Yao, Y. Xu, H. Xu, Y. Liao, and Z. Xie, "Efficient deployment of large language models on resource-constrained devices," *arXiv preprint arXiv:2501.02438*, 2025.
[35] S. Chen, R. Jiang, D. Yu, J. Xu, M. Chao, F. Meng, C. Jiang, W. Xu, and H. Liu, "Kvdirect: Distributed disaggregated llm inference," *arXiv preprint arXiv:2501.14743*, 2024.
[36] A. Gangidi, R. Miao, S. Zheng, S. J. Bondu, G. Goes, H. Morsy, R. Puri, M. Riftadi, A. J. Shetty, J. Yang, S. Zhang, M. J. Fernandez, S. Gandham, and H. Zeng, "Rdma over ethernet for distributed training at meta scale," in *Proceedings of ACM SIGCOMM*, 2024.
[37] R. Qin, Z. Li, W. He, J. Cui, F. Ren, M. Zhang, Y. Wu, W. Zheng, and X. Xu, "Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot," in *Proceedings of USENIX FAST*, 2025.
[38] C. Zhao, S. Zhou, L. Zhang, C. Deng, Z. Xu, Y. Liu, K. Yu, J. Li, and L. Zhao, "Deepep: an efficient expert-parallel communication library," https://github.com/deepseek-ai/DeepEP, 2025.
[39] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like clockwork: Performance predictability from the bottom up," in *Proceedings of USENIX OSDI*, 2020.
[40] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, and I. Stoica, "AlpaServe: Statistical multiplexing with model parallelism for deep learning serving," in *Proceedings of USENIX OSDI*, 2023.
[41] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.
[42] Intel, "Intel intelligent fabric processors," Online: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors.html, 2024.
[43] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng, T. Feng, F. Ning, K. Chen, and C. Guo, "Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing," in *Proc. of USENIX NSDI*, 2022.
[44] S. Zhao, F. Li, X. Chen, X. Guan, J. Jiang, D. Huang, Y. Qing, S. Wang, P. Wang, G. Zhang, C. Li, P. Luo, and H. Cui, "vpipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel dnn training," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 489–506, 2022.
[45] H. Liu, J. Chen, J. Dy, and Y. Fu, "Transforming complex problems into k-means solutions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 7, pp. 9149–9168, 2023.
[46] L. Shengyu and Z. Yinmin, "Swifttransformer: High performance transformer implementation in c++," 2025. [Online]. Available: https://github.com/LLMServe/SwiftTransformer
[47] "Nvidia dcgm," 2025. [Online]. Available: https://developer.nvidia.com/dcgm
[48] Y.-C. Lin, W. Kwon, R. Pineda, and F. N. Paravecino, "Apex: An extensible and dynamism-aware simulator for automated parallel execution in llm serving," 2024. [Online]. Available: https://arxiv.org/abs/2411.17651
[49] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, and I. Stoica, "AlpaServe: Statistical multiplexing with model parallelism for deep learning serving," in *Proceedings of USENIX OSDI*, 2023.
[50] D. Xu, H. Zhang, L. Yang, R. Liu, G. Huang, M. Xu, and X. Liu, "Fast on-device llm inference with npus," in *Proceedings of ACM ASPLOS*, 2025.
[51] N. Corporation, "Fastertransformer," 2019.
[52] H. Pan, P. Cui, Z. Li, R. Jia, P. Zhang, L. Zhang, Y. Yang, J. Wu, M. Lauren, and G. Xie, "Zebra: Accelerating distributed sparse deep training with in-network gradient aggregation for hot parameters," in *Proceedings of IEEE ICNP*, 2024.
[53] H. Wang, Y. Qin, C. Lao, Y. Le, W. Wu, and K. Chen, "Preemptive switch memory usage to accelerate training jobs with shared in-network aggregation," in *Proceedings of IEEE ICNP*, 2023.
[54] J. Nie and W. Wu, "Aggtree: A routing tree with in-network aggregation for distributed training," in *Proceedings of IEEE IPCCC*, 2023.
[55] Y. Li, W. Li, Y. Yao, Y. Du, and K. Li, "Host-driven in-network aggregation on rdma," in *Proceedings of IEEE INFOCOM*, 2024.
[56] B. Zhao, W. Wu, and W. Xu, "NetRPC: Enabling In-Network computation in remote procedure calls," in *Proceedings of USENIX NSDI*, 2023.