

Hydraulis: Balancing Large Transformer Model Training via Co-designing Parallel Strategies and Data Assignment

Haoyang Li*
lihaoyang@stu.pku.edu.cn
Peking University
China

Fangcheng Fu†
ccchengff@sjtu.edu.cn
Shanghai Jiao Tong
University
China

Sheng Lin*
linsh@stu.pku.edu.cn
Peking University
China

Hao Ge*
gehao@stu.pku.edu.cn
Peking University
China

Xuanyu Wang*
wxyz0001@pku.edu.cn
Peking University
China

Jiawen Niu*
niujiawen705@stu.pku.edu.cn
Peking University
China

Jinbao Xue
jinbaoxue@tencent.com
Tencent
China

Yangyu Tao
brucetao@tencent.com
Tencent
China

Di Wang
diwang@tencent.com
Tencent
China

Jie Jiang
zeus@tencent.com
Tencent
China

Bin Cui*‡
bin.cui@pku.edu.cn
Peking University
China

Abstract

To optimize large Transformer model training, both efficient parallel computing and advanced data management are indispensable. However, current methods often assume a stable and uniform training workload, neglecting data-induced imbalances—arising from both sampling and packing processes—which can impede training performance. Specifically, data sampling imbalance arises from uneven sequence length distribution of the training data, while data packing imbalance stems from the discrepancy between the linear memory complexity and quadratic time complexity of the attention mechanism. To address these imbalance issues, we develop Hydraulis, which jointly optimizes the parallel strategies and data assignment. For one thing, we introduce large model training with dynamic heterogeneous parallel strategies in response to the sequence length variations within and across training iterations. For another, we devise a two-stage data assignment approach, which strikes a good balance in terms of the training workloads both within and across model replicas. Empirical results demonstrate that Hydraulis outperforms existing systems by 1.32-2.66 $\times$. Our source code is available: <https://github.com/PKU-DAIR/Hetu>.

1 Introduction

In recent years, large Transformer models [88] have become pivotal in fields such as natural language processing [10, 22, 78], computer vision [11, 24, 64], video processing [3, 8, 28], speech recognition [5, 23, 33] and intelligent agents [96, 98, 101, 107]. Their success relies on pre-training models with vast parameters on extensive datasets [27, 46].

Within this landscape, two key challenges arise. (a) To achieve better model ability, efforts have been made to scale models to

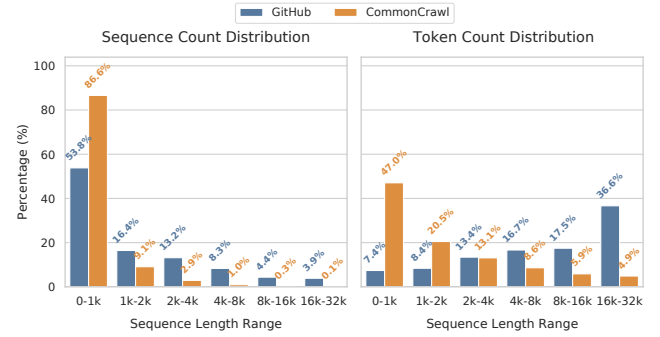


Figure 1: Sequence length distribution of two popular open-sourced datasets. We display the number of sequences and tokens within each length range, with both datasets exhibiting high variance.

larger sizes [46], making large model training substantially time-consuming and memory-intensive. (b) Meanwhile, to move towards Artificial General Intelligence (AGI), input data are often condensed into sequences of tokens to achieve a unified representation [24, 25, 68]. However, this representation naturally exhibits length variability [20]. For instance, Figure 1 presents the sequence length distribution after tokenizing two popular large language models (LLMs) training datasets—CommonCrawl and GitHub—revealing that the sequence lengths are significantly diverse in the real world. Thus, large model training takes as input variable-length data.

To address this, approaches from two perspectives have been developed. (a) From a parallel computing perspective, methods like data parallel [6, 58, 100, 102], tensor parallel [49, 70, 85], pipeline parallel [32, 36, 38, 47, 52, 69, 77], sequence and context parallel [9, 29, 31, 39, 55, 61, 93] have been proposed, and are often combined to train large-scale models. Data parallel operates at the granularity of training pipelines (i.e., data parallel groups containing model replicas that are capable of independent forward and backward propagation) and thus can typically scale by adding more pipelines, while

*School of Computer Science & Key Lab of High Confidence Software Technologies (MOE), Peking University

†School of Artificial Intelligence, Shanghai Jiao Tong University

‡Institute of Computational Social Science, Peking University (Qingdao)

other parallel methods (i.e., tensor, pipeline, sequence and context parallel) are optimized within a single training pipeline to balance pipeline latency and memory usage. (b) From a data management perspective, data packing techniques—combining shorter sequences into longer ones for simultaneous processing—are widely used in the training of both language [7, 50, 51, 74, 90] and vision [16, 21] models to minimize redundant computations.

Despite these advancements, existing training systems typically leverage these two perspectives in a decoupled manner. (a) In parallel computing, most systems overlook the sequence length variation and adopt a static, homogeneous parallel strategy [41, 80, 85, 105], which deploys all training pipelines (a.k.a. model replicas) with the same parallel scheme throughout the training (e.g., all pipelines have identical tensor parallel and pipeline parallel degree).¹ (b) In data management, existing works usually specify a maximum sequence length (a.k.a. context length), pack the original sequences into multiple packed ones that do not exceed the maximum length (termed max-length packing), and distribute the packed sequences evenly across machines [51, 74].

In short, current works enforce a static, homogeneous parallel strategy and a max-length packing on variable-length data. However, our empirical analysis (details shown in §3) reveals that this decoupled design overlooks two critical imbalances.

Firstly, sequences in the real world substantially vary in length by nature [20]. This leads to the *data sampling imbalance*, manifesting as variable sequence lengths within iterations and inconsistent maximum lengths across iterations. When employing a homogeneous parallel strategy, we must ensure the parallel scheme is memory-saving enough (e.g., with a high context parallel or tensor parallel degree) to support the maximum sequence length, which compromises the efficiency of short sequences.

The second is the *data packing imbalance*. Due to the attention mechanism in Transformer, there is a discrepancy between the quadratic time complexity and the linear memory complexity w.r.t. the sequence length [19, 74, 88]. However, existing works adopt max-length packing, achieving memory balance but leaving the training workloads imbalanced within and across pipelines.

Motivated by this, we manage to address these imbalance issues by jointly optimizing the parallel and packing strategies, putting forward co-designs in both the parallel computing and data management perspectives. To this end, we introduce Hydraulis, an innovative system designed for efficient training of large Transformer models. Our major contributions are outlined as follows.

(I) Training with dynamic heterogeneous parallel strategies. Unlike conventional training that relies on a static, homogeneous parallel strategy, we employ dynamic, heterogeneous strategies to address the *data sampling imbalance*. Each heterogeneous strategy can deploy pipelines with varying parallel schemes to handle sequences of varying lengths. Divergent strategies can be used for different iterations, allowing each iteration to employ the most suitable strategy for the ad hoc sequence length distribution.

(II) Disaggregating optimization and propagation with sub-graphs. To support training with dynamic heterogeneous parallel

strategies, we present a novel disaggregation of the optimization phase (i.e., synchronizing gradients and updating the model) and propagation phase (i.e., executing forward and backward propagation in a pipelined manner) in distributed training. This approach maintains consistent model states sharding for optimization, and can transit to arbitrary heterogeneous parallel strategy for propagation. We further introduce subgraphs (i.e., subsets of the complete computation graph) to unify the representation of sophisticated communication needs during propagation (for activations) and optimization-propagation interactions (for parameters and gradients). This allows for flexible combinations of parallel schemes and seamless transitions between parallel strategies.

(III) Two-stage sequence assignment. Given a mini-batch of sequences and an arbitrary strategy, we devise a two-stage approach to eliminate the *data packing imbalance*. In particular, we formulate an optimization problem that aims to determine how packed sequences can be balanced within each pipeline. Based on this, we also strike a good balance across different pipelines by carefully dispatching the original (unpacked) sequences. Besides, this process also estimates latency for each possible strategy, allowing us to evaluate and select the best one for each iteration.

(IV) Data distribution-aware strategy proposal. Guided by the sequence assignment and for practical considerations, we further propose preparing the candidate strategies that are promising in advance, in order to accelerate the strategy enumeration and selection for each iteration. It takes the overall sequence length distribution of the entire dataset into account, and proposes the candidate strategies via a dynamic programming process.

(V) Implementation and Evaluation. We implement Hydraulis on top of Hetu² [56] and conduct extensive experiments to demonstrate that Hydraulis achieves 1.32-2.66× improvement over state-of-the-art training systems, while also offering better scalability.

2 Preliminaries

2.1 Data Sampling and Data Packing

During the training of a Transformer model, each iteration samples a mini-batch of sequences with a pre-defined threshold on the total number of tokens, performs forward and backward propagation on the mini-batch, and updates the model.

Besides, the model is typically trained with a pre-defined context length, representing the maximum number of tokens it can process in a single input [76]. Any sequence exceeding this limit is typically truncated to fit within the context length. But regardless of whether the original sequences are truncated or not, the lengths of the sampled sequences will inevitably exhibit variance.

This variability introduces challenges when constructing batches, as sequences need to be aligned to a common length for efficient batch processing. Traditionally, this issue is addressed through data padding, where shorter sequences are padded with special tokens (e.g., [PAD] tokens) to match the length of the longest one [22, 88]. Although padding ensures uniformity in sequence length and is straightforward to implement, it leads to unnecessary computation on the padded tokens, resulting in wasted computational resources.

As shown in Figure 2, an alternative to padding is packing [50], which seeks to maximize computational efficiency by grouping

¹In this work, we use “parallel scheme” to denote the parallel configuration of a pipeline (each pipeline does not necessarily employ any parallel methods, i.e., it can be deployed on only one GPU). Given the available GPUs, a “parallel strategy” describes the collection of parallel schemes of all pipelines deployed.

²<https://github.com/PKU-DAIR/Hetu>

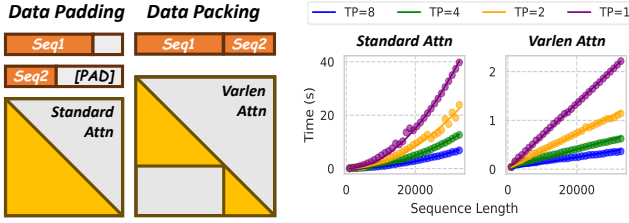


Figure 2: Left: Illustration of padding and packing. Right: Comparison of the attention latency during LLaMA2 7B training on Nvidia A800 GPUs with tensor parallel (TP). Standard attention scales quadratically, while varlen achieves linear scaling relative to the sequence length (1K) before packing.

sequences of varying lengths into a single longer sequence, thereby eliminating the need for padded tokens. To prevent cross-contamination between the packed sequences, positional embeddings and the attention mask must be adjusted to preserve the integrity of the attention mechanism [7, 51, 90]. Fortunately, modern techniques such as FlashAttention v2 have integrated support for block diagonal attention, which natively handles variable-length sequences (i.e., varlen) [18, 19]. Moreover, the varlen interface ensures that the attention complexity scales as $\sum S_i^2$, rather than $(\sum S_i)^2$, where S_i represents the length of the i -th original sequence being packed [74]. This significantly reduces the computational overhead of data packing, making it a more practical and efficient approach.

2.2 Parallel Methods, Schemes, and Strategies

As models grow in both parameter size and computational complexity, distributed training across multiple devices becomes essential. Below we introduce popular parallel methods for large Transformer model training.

Data parallel. In data parallel (DP), the model is replicated across multiple devices, with each device processing a different subset of the input data [6, 58, 100, 102]. After computing the gradients locally, devices must synchronize their model updates, typically through a gradient reduction process, to ensure consistency across devices.

Parallel scheme of one pipeline. In this work, we call each data parallel group one training pipeline, and the parallel scheme describes how a pipeline is deployed. We can broadly classify parallel methods for a pipeline into inter- and intra-op parallel, respectively.

The Inter-op parallel is mainly implemented with pipeline parallel (PP). Particularly, PP divides the model into sequential stages, with each stage assigned to a different device. Data from a mini-batch is further split into micro-batches, which flow through the pipeline in a staged manner. (When training Transformer models, each micro-batch accounts for one packed sequence.) Techniques such as micro-batch scheduling and gradient accumulation are often employed, allowing multiple micro-batches to be processed efficiently [32, 36, 38, 47, 52, 69, 77].

Unlike inter-op parallel, there are several intra-op parallel methods for a pipeline. Tensor parallel (TP) partitions each layer across multiple GPUs and exchanges intermediate results (e.g., via all-gather and reduce-scatter operations) to complete the forward

and backward passes [49, 70, 85]. Additionally, sequence parallel (SP) and context parallel (CP) have been explored to handle extremely long sequences, where a single sequence is partitioned across multiple devices to avoid out-of-memory (OOM) errors, and all-to-all or ring-style peer-to-peer communication would be necessary to perform the attention mechanism over the partitioned sequences [9, 29, 31, 39, 55, 61].

Parallel strategy. Building on the parallel scheme of a single pipeline, the parallel strategy refers to the collection of parallel schemes of all pipelines. However, existing works mainly leverage the same parallel schemes for all pipelines, resulting in a homogeneous parallel strategy. Such a homogeneous design fails to take account of the imbalances caused by data sampling and data packing, which will be discussed in §3.

2.3 Model States Sharding

To reduce memory consumption during training, model states sharding techniques (a.k.a. ZeRO [79] and FSDP [102, 104]) are widely used. These methods extend standard data parallel (DP) by partitioning model states (i.e., parameters, gradients, and optimizer states) across multiple devices, rather than replicating them across all data parallel groups (i.e., pipelines).

The sharding technique is combined with mixed-precision training [67, 73], which uses both 16-bit floating point (FP16 or BF16) and 32-bit floating point (FP32) precisions to balance efficiency and numerical stability. Figure 3a illustrates an example with each pipeline deployed with a tensor parallel degree of 2.

The 32-bit optimizer states and parameters (which consume more memory but are critical for ensuring convergence) are sharded across GPUs, while the 16-bit parameters and gradients (which are more memory-efficient and computationally faster) are maintained during propagation.

As aforementioned, current works mainly adopt a homogeneous parallel strategy, so they usually align the sharding with the parallel strategy, allowing distributed state transformations to be easily achieved via all-gather and reduce-scatter operations.

However, since such transformations can overlap with computation, it is also feasible to adopt more nuanced, non-aligned parameter pulling and gradient pushing using customized communication strategies, which is depicted in Figure 3b and will be elaborated in §5.2.

2.4 Graph-based Representations

The execution of a deep learning model can be represented as a computation graph—a directed acyclic graph (DAG) where nodes correspond to operators (e.g., matrix multiplication, attention) defined by users and edges indicate data dependencies. Modern deep learning frameworks, such as PyTorch [58] and TensorFlow [2], adopt this graph-based representation to encapsulate execution logic. Following this convention, we employ the same representation but extend it with a subgraph abstraction to enable dynamic heterogeneous strategies, which will be presented in §5.3.

2.5 Limitations of State-of-the-art Systems

State-of-the-art (SOTA) training systems, including Megatron-LM [49, 70, 85] and DeepSpeed [39, 80], have incorporated diverse parallel

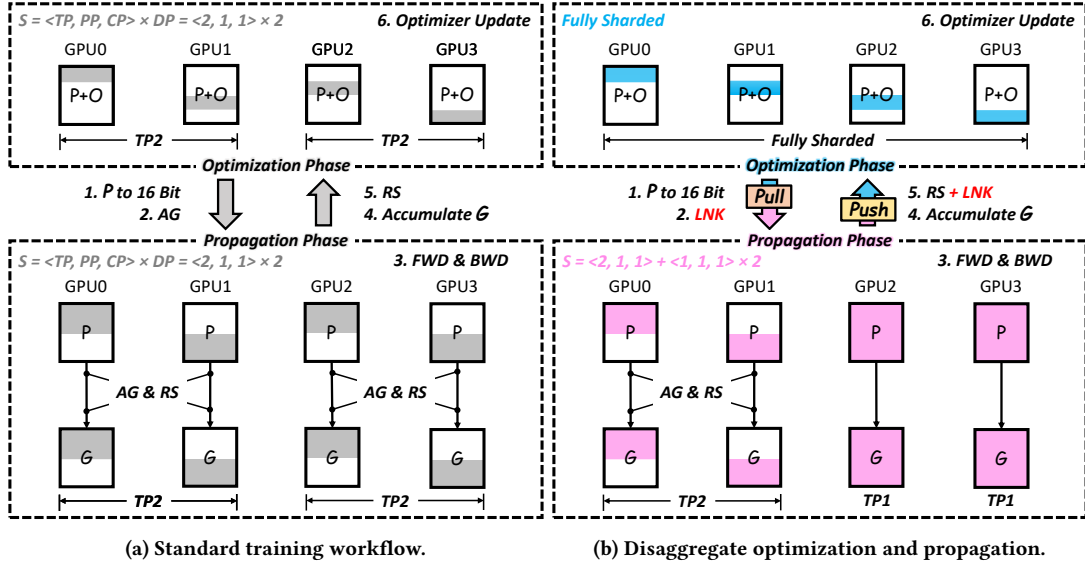


Figure 3: “O, P, G” represents optimizer states, parameters and gradients, respectively. “AG, RS, S, R” represents all-gather, reduce-scatter, send and receive, respectively. “To” represents the data type transfer operator (from 32- to 16-bit floating points), and “LNK” represents the link operator designed by us. (a) The overall workflow of the model states sharding technique combined with mixed-precision training. (b) Our proposed optimization-propagation disaggregation technique, which decouples the distributed strategies for the optimization phase and propagation phase, allowing for arbitrary heterogeneous parallel strategies during propagation.

methods (e.g., Megatron-LM supports DP, TP, PP and CP, while DeepSpeed offers DP and SP) and model states sharding techniques (e.g., Megatron-LM implements ZeRO-1, which shards the optimizer states and the FP32 parameters, while DeepSpeed supports ZeRO 1-3 to allow further sharding FP16/BF16 gradients and parameters). However, they are limited to using one static parallel strategy throughout the training task. A more recent work, HotSpa [30], enables the use of multiple pre-defined parallel strategies and supports switching between them within an iteration. This enables data in different sequence lengths within an iteration to utilize different parallel strategies while accumulating gradients for a global update.

Nonetheless, as will be analyzed in §3, various aspects of data imbalance necessitate support for dynamic heterogeneous strategies. However, all of these systems lack this capability, as they are constrained to homogeneous parallel strategies (i.e., all pipelines use the same parallel scheme) and require model states sharding to align with the parallel strategy. This is due to the following reasons: (a) Existing designs do not support parameter retrieval and gradient aggregation across heterogeneous pipelines, as they rely on homogeneous communication primitives (i.e., all-gather and reduce-scatter) but lack heterogeneous ones. (b) Balancing workload across heterogeneous pipelines is challenging due to the absence of fine-grained data assignment mechanisms and strategy design methods that take data distribution into account.

To address (a), Hydraulix adopts an optimization-propagation disaggregation architecture to facilitate sophisticated parameter retrieval and gradient aggregation, and introduces subgraphs to support arbitrary heterogeneous strategies (§5). To address (b), it proposes a two-stage sequence assignment method for load balancing (§6), and pre-generates distribution-aware heterogeneous parallel strategies before training (§7). Together, these innovations

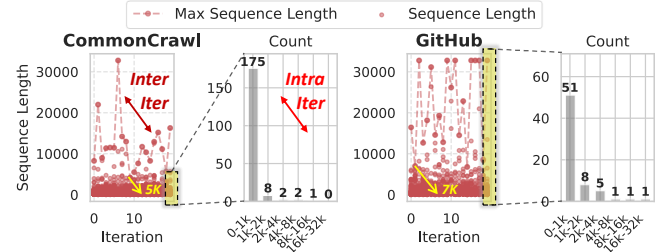


Figure 4: The dot plot on the left shows the fluctuation of maximum sequence length, emphasizing inter-iteration imbalance, while the bar chart on the right presents the distribution of sequence lengths within an iteration (20th iteration), highlighting intra-iteration imbalance.

overcome the limitations of current training systems and improve training efficiency under data imbalance conditions (§8).

3 Imbalances Analysis

In this section, we analyze the imbalance issues encountered during data sampling and packing in training on real-world datasets, and discuss potential optimization opportunities. To ease the analysis, we exemplify the imbalance issues by training the LLaMA 13B model with the CommonCrawl [86] and GitHub [17] datasets over 16 GPUs, with 32K context length and 100K tokens per iteration.

3.1 Data Sampling Imbalance

We first concentrate on the imbalance in terms of data sampling. As shown in Figure 4, we observe two key issues: (a) within iterations, sequence lengths are unevenly distributed, with more short sequences and fewer long ones; (b) across iterations, the sequence

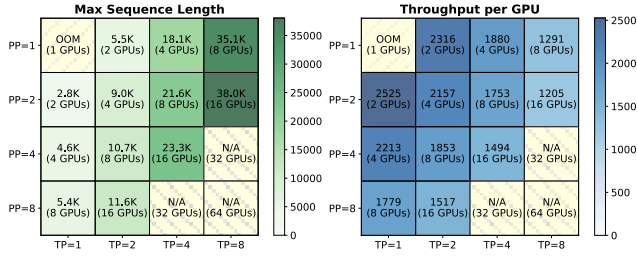


Figure 5: Memory and throughput trade-offs. “OOM” indicates out-of-memory, and “N/A” indicates not available. The experiment is conducted using a 13B LLaMA model with 16 Nvidia A800 GPUs. Throughput is measured in tokens per second and maximum sequence lengths are given by $\text{MaxLen}(\cdot)$ (see §6.1 and Appendix C.1 [1]).

distribution varies significantly, and the maximum sequence length fluctuates considerably, often falling short of the context length. In the following, we analyze the two issues, respectively.

Intra-iteration imbalance. The uneven sequence length distribution within an iteration can be attributed to the inherent property of data. In particular, sequences in the real world generally follow a long-tail distribution, where short sequences dominate and long sequences are rare. Existing studies have also observed such a distribution in many popular training datasets [20]. This imbalance poses both challenges and opportunities when it comes to optimizing overall performance.

The challenges stem from the trade-offs between memory usage and throughput offered by different numbers of GPUs and parallel schemes. As shown in Figure 5, increasing the number of GPUs generally leads to a lower throughput per GPU but allows for a longer maximum sequence length. Meanwhile, under the same number of GPUs, different parallel schemes (e.g., PP, TP and their combinations) also have divergent trade-offs (e.g., replacing PP with TP generally supports longer sequences but at the cost of reduced per-GPU throughput). As aforementioned in §2.2, existing works utilize all GPUs in the cluster uniformly with a homogeneous parallel strategy tailored for the longest context length, compromising the overall training efficiency for memory.

Nevertheless, combined with the length imbalance within a mini-batch, such trade-offs become particularly invaluable: short sequences can be processed using faster but more memory-intensive parallel schemes, while the long sequences can be assigned to slower yet more memory-efficient parallel schemes. Therefore, adopting heterogeneous parallel strategies—where non-unique parallel schemes are applied on different proportions of GPUs based on different sequence lengths—enables better performance (§5).

Inter-iteration imbalance. While intra-iteration imbalance arises from the distribution of sequence lengths within a single mini-batch, another challenge comes from the distribution variations across mini-batches. Particularly, due to the long-tail distribution, rare long sequences may not appear in every iteration (even with a large batch size on a 1024-GPU cluster, the maximum sequence length still varies, as detailed in Appendix A.1 [1]). This leads to fluctuations in data distribution and maximum sequence lengths across iterations. However, current works overlook this issue, using a static parallel strategy for all iterations, which is unsuitable. For

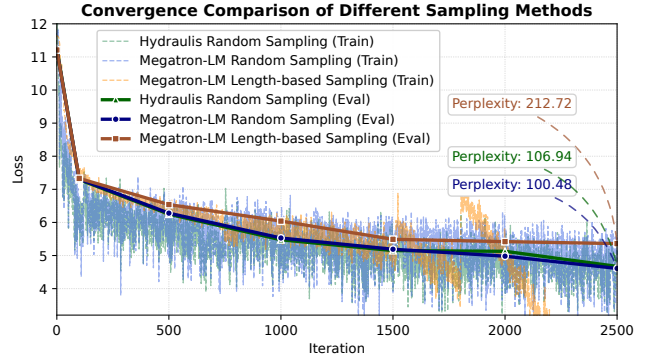


Figure 6: Training and evaluation loss of different sampling methods. We subsample 250M tokens for training and 5M for validation from the GitHub dataset, and train a 13B model on 16 GPUs with 32K context length and 100K tokens per iteration.

example, when an iteration consists mainly of short sequences, parallel schemes designed for long sequences become overkill. To address this, we propose to dynamically adjust the parallel strategy based on the characteristics of each mini-batch to better align the workload, thereby enhancing overall performance across iterations (§5).

Discussions. Readers might suspect that both intra- and inter-iteration sampling imbalance can be mitigated by modifying the sampling order. However, such approaches degrade model convergence because they incur randomness breaches when sampling the mini-batches during training. As shown in Figure 6, we conduct a testbed to evaluate a length-based sampling method using Megatron-LM, which pre-sorts the dataset into three intervals (<4K, 4K–16K, and 16K–32K) and trains on them one by one. (By doing so, the sampling imbalance is mitigated, and the intervals can be trained with different parallel strategies since they differ in the maximum sequence lengths.) The results show that such a method suffers from a worse convergence compared to random sampling, and eventually, we observe a 2× gap in the perplexity metric. In contrast, Hydraulis preserves random sampling and synchronizes all gradients before model update, therefore the perplexity shows no significant deviation. This verifies the necessity of maintaining randomness in large model training.

Additionally, a prior work, HotSPa [30], addresses the data sampling imbalance in a different way—it performs length-based training within each iteration instead of over the entire dataset, in order to preserve the model convergence. By dynamically switching between expert-tuned, homogeneous parallel strategies for different sequence lengths, HotSPa accumulates gradients and performs a single global update per iteration. However, it lacks support for heterogeneous strategies and finer-grained data assignment, and frequent strategy switching introduces additional overhead. We will empirically compare HotSPa and our work in §8.

3.2 Data Packing Imbalance

We then focus on the imbalance issues of data packing. As illustrated in Figure 7, we observe significant distribution differences in the original sequences within packed sequences (i.e., micro-batches)

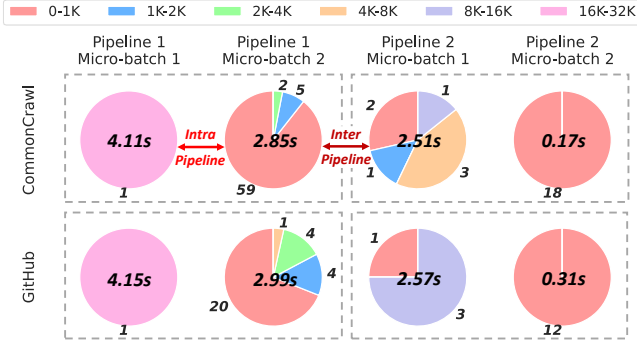


Figure 7: The numbers of original sequences (classified by their lengths) in different micro-batches after packing (each micro-batch accounts for one packed sequence) and their corresponding time cost. The experiment is conducted using a 13B LLaMA model with 16 Nvidia A800 GPUs.

due to varying sequence lengths. These variations occur both within micro-batches of the same pipeline and across different pipelines, resulting in execution time variance. Below, we analyze each aspect’s impact separately.

Intra-pipeline imbalance. Since the memory consumption of each micro-batch is linear to the total length after packing, existing works commonly adopt max-length packing methods, like bin packing [74, 99], to balance memory usage. However, due to the quadratic time complexity of attention [19, 74, 88], the computation latency can hardly be balanced in this way—for example with Figure 2 again, attention latency can be nearly 20× smaller when packing 32 sequences of 1K compared to a single 32K sequence.

In pipeline parallel (PP), this leads to uneven execution times across micro-batches, yet common pipeline scheduling, such as 1F1B [36] or zero-bubble [77], requires uniform execution time of micro-batches to minimize pipeline bubbles. This highlights the importance of taking account of both the linear memory complexity and quadratic time complexity, rather than considering solely the memory as in max-length packing. To address this, we devise a nuanced packing strategy that considers the variation in original sequence lengths, striking a balance in terms of the execution time across different micro-batches within the memory constraint (§6.1).

Inter-pipeline imbalance. Although different pipelines execute concurrently, the training latency is dictated by the slowest pipeline due to the need for gradient synchronization. Simply packing sequences to a fixed maximum length has a similar issue of unbalanced computation load. In some cases, if the sequence number after packing cannot be evenly distributed across pipelines, a more severe imbalance will arise. In contrast to current systems that first pack sequences and then attempt to distribute them evenly, we suggest reversing this process. Specifically, our work first dispatches sequences of varying lengths to different pipelines, and then focuses on packing them within each pipeline. This innovative dispatching paradigm ensures a more balanced load across pipelines, before organizing sequences into micro-batches (§6.2).

4 Overview

We develop Hydraulis to address the training imbalances described above. Figure 8 illustrates the overview.

To tackle the data sampling imbalance, we develop a brand new system that supports training with dynamic heterogeneous parallel strategies (§5). In each heterogeneous parallel strategy, the training pipelines can be associated with non-unique parallel schemes to differentiate between long and short sequences for each iteration, addressing the *intra-iteration imbalance*. And we further support adjusting the strategies dynamically across different iterations to tackle the *inter-iteration imbalance*.

To tackle the data packing imbalance, given a mini-batch of sequences and a strategy, we devise a two-stage assignment (§6). The first stage determines how to dispatch sequences of varying lengths to different pipelines to solve the *inter-pipeline imbalance*. Then, the second stage meticulously packs the sequences within each pipeline to form different micro-batches, addressing the *intra-pipeline imbalance*. This two-stage assignment process also estimates the latency of the given strategy to train the given mini-batch.

The overall routine is as follows. At initialization (step ①), we propose candidate strategies given the training task (§7). For each iteration of the training process, a mini-batch of sequences is drawn, based on which, we enumerate each candidate strategy (step ②), employ the two-stage assignment to obtain its estimated performance (steps ③-④), and select the best one (step ⑤) for subsequent training (step ⑥). The strategy selection (steps ②-④) is done on CPUs, which overlaps with the training (step ⑥) on GPUs.

5 System Designs

In this section, we start by mathematically defining our heterogeneous parallel strategies, then introduce the system techniques proposed to support these strategies and the dynamic transitions between them.

5.1 Dynamic Heterogeneous Strategy Design

Definitions. First, we focus on an independent **parallel scheme**, denoted as P , which describes the parallel configuration of one training pipeline. It can integrate multiple parallel methods. For example, if we employ tensor parallelism (TP) with a degree of 2 to partition individual model layers across devices, further combine it with pipeline parallelism (PP) of degree 4 to distribute these layers into sequential stages, and apply context parallelism (CP) of degree 1 to split input sequences, then P can be represented as $\langle \text{TP}, \text{PP}, \text{CP} \rangle = \langle 2, 4, 1 \rangle$. In the following sections, to simplify representation, we introduce $\text{TP}(P)$, $\text{PP}(P)$ and $\text{CP}(P)$ to represent the TP, PP, and CP degrees for a given P , respectively. Based on this, we define the **parallel scheme space** $\mathbb{P}$, which contains various parallel schemes. For example with TP, PP, and CP again, $\mathbb{P}$ can be defined as

$$\mathbb{P} = \{P | \text{TP}(P) \in \mathbb{D}_T, \text{PP}(P) \in \mathbb{D}_P, \text{CP}(P) \in \mathbb{D}_C, \\ \text{TP}(P) \times \text{PP}(P) \times \text{CP}(P) \leq N\},$$

where $\mathbb{D}_T = \mathbb{D}_C = \{1, 2, \dots, N\}$, $\mathbb{D}_P = \{v \in \mathbb{N}^+ | v \text{ divides } \overline{\text{layers}}\}$, N is the number of GPUs, and $\overline{\text{layers}}$ is the number of layers.

Next, we define our **heterogeneous strategy space** as the linear span of the parallel scheme space: $\mathbb{S} = \text{span}(\mathbb{P})$, meaning that each

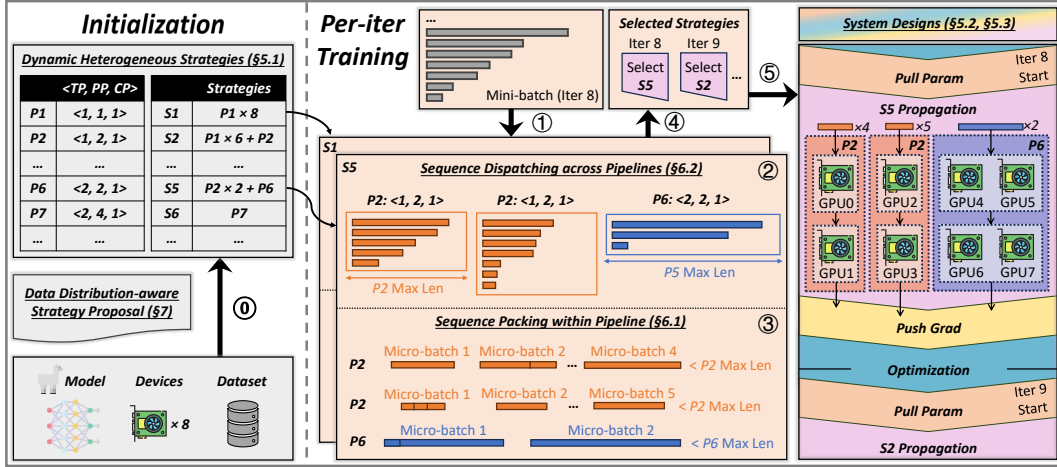


Figure 8: Hydraulics overview. The gray modules represent the initialization part, which is executed only once when training a specific model on fixed devices with a fixed dataset. The remaining modules operate per iteration during training: the beige modules indicate the two-stage sequence assignment, while the rest of the modules illustrate our system that supports dynamic heterogeneous strategies, with different colors representing different training phases. Hydraulics executes step ① at initialization and steps ①–⑤ for each iteration.

heterogeneous strategy can be represented as a linear combination of parallel schemes: $S = \sum_{i=1}^K d_i \times P_i$, which indicates that S consists of $\sum_{i=1}^K d_i$ pipelines of K different parallel schemes, with d_i pipelines deployed with P_i . Figure 8 shows some example parallel schemes (P) and heterogeneous strategies (S) when the number of GPUs (N) is 8.

Strategy execution and transition. During each training iteration, given a mini-batch of sequences, we select the most suitable S to execute.³ The pipelines will be assigned different sequences and execute forward and backward passes concurrently to compute gradients from the corresponding parameters. This process is termed the **propagation phase** (Figure 3a, 3b, step 3). Then, similar to traditional data parallelism (DP), all pipelines synchronize the model gradients and perform updates at the end of each iteration to obtain the new parameters and optimizer states. This process is called the **optimization phase** (Figure 3a, 3b, step 6). Since different iterations inevitably select non-unique strategies, strategy transitions between iterations are necessary (e.g., in Figure 8, iteration 8 uses strategy $S5$, while iteration 9 transitions to strategy $S2$).

Challenges. To support the execution and transition of arbitrary heterogeneous strategies, the conventional approach to handling model states must be revised. As shown in Figure 3a, for current systems that employ a static, homogeneous parallel strategy, it is common practice to shard the model parameters and optimizer states according to the employed homogeneous strategy, and leverage all-gather and reduce-scatter operations to retrieve the parameters and synchronize the gradients, respectively. However, this approach falls short in supporting dynamic heterogeneous strategies, as model partitioning varies significantly across pipelines. For example, in Figure 3b, pipelines with different TP degrees produce parameters and gradients with divergent partitioning granularities

(i.e., tensors are split into non-uniform slices across GPUs). Consequently, it is infeasible to discover a single communication pattern to accomplish the parameter retrieval or gradient synchronization. This limitation drives our development of the LNK operator, which supports parameter retrieval and gradient synchronization in an optimization-propagation disaggregated manner (§5.2), and we further introduce subgraphs to encapsulate diverse communication patterns (§5.3). Furthermore, since gradient synchronization requires all pipelines to participate and is constrained by the slowest one, pipeline heterogeneity also poses significant load-balancing challenges, which we will address in §6.

5.2 Optimization-Propagation Disaggregation

As illustrated in Figure 3b, we separate the optimization phase from the propagation phase, decoupling the GPU’s role as a distributed storage from its computational role during propagation. Specifically, we keep the partitioning of FP32 parameters and optimizer states fixed for the optimization phase⁴, while allowing an arbitrary parallel strategy to be used in the propagation phase. In such a disaggregated design, to ensure correct parameter partitioning during propagation as well as correct gradient partitioning for optimizer update, we establish two types of interaction: (a) *pull*, which retrieves the parameters from the optimization phase to an arbitrary heterogeneous parallel strategy (Figure 3b, steps 1–2); (b) *push*, which synchronizes and scatters gradients back to the optimization phase (Figure 3b, steps 4–5). Compared to the all-gather and reduce-scatter in the conventional design that requires identical, homogeneous parallel strategy in the two phases (Figure 3a, steps 1–2 and 4–5), our *pull* and *push* abstractions enable the use of arbitrary heterogeneous parallel strategies in the propagation phase as well as the seamless transition between different strategies.

To effectively manage *pull* and *push* interactions, we develop a specialized LNK (link) operator to handle complex communication

³The selection in each iteration is based on the two-stage sequence assignment (§6), and we propose the candidate strategies by analyzing the sequence length distribution of the entire dataset (§7).

⁴We use a fully sharded strategy in the optimization phase, yet any other zero-redundancy sharding strategies are also feasible.

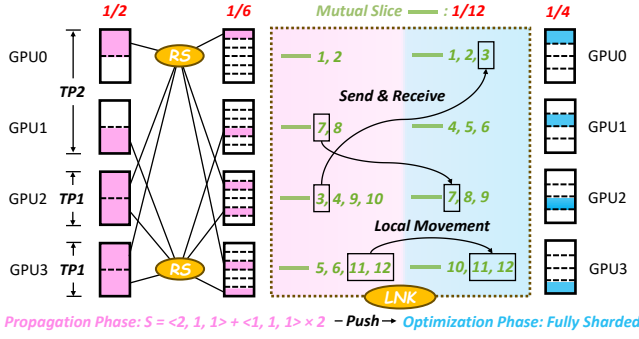


Figure 9: Diagram of RS (reduce-scatter) and LNK (link) operations during *push*. Pink and blue denote slices in the propagation and optimization phases, respectively, while green represents the *mutual slice* (the finest-grained slice shared by both). The partitioning granularity is 4 in the propagation phase and 6 in the optimization phase (after reduce-scatter). Thus, the *mutual slice* has a granularity of 12 (i.e., the least common multiple of 4 and 6).

patterns. As depicted on the right side of Figure 9, this operator utilizes three core primitives: *send*, *receive*, and *local movement*, all operating on a fundamental unit called the *mutual slice*, which represents the finest-grained data partition. These primitives enable dynamic resharding of parameters and gradients between propagation and optimization phases. For instance, in Figure 9, GPU1 generates a gradient slice during propagation, which is further partitioned into two finest-grained *mutual slices* (7 and 8). Since GPU0 requires *mutual slice* 7 during the optimization phase for model updates, it triggers a *send* to transfer *mutual slice* 7 from GPU1 to GPU0. Because optimization states are uniquely partitioned across GPUs (without redundancy), the use of these primitives is deterministic.

Mathematically, assume there are N GPUs, the optimization phase employs a fully sharded strategy, and the propagation phase uses a strategy $S = \sum_{i=1}^K d_i \times P_i$. Let $\overline{TP}_{\max} = \max_{i \in [1, K]} TP(P_i)$. For the *pull* interaction, the *mutual slices* are created by evenly partitioning the global parameter into $\text{LCM}(N, \overline{TP}_{\max})$ segments (where LCM denotes the least common multiple). Each GPU identifies its necessary *mutual slices*. If a slice is local, it uses the *local movement* operation; otherwise, it employs *send* and *receive* to exchange slices. Similarly, for the *push* interaction, as shown on the left side of Figure 9, a reduce-scatter is first performed at the corresponding location to synchronize the gradient. Then the partitioning granularity of the *mutual slice* is $\text{LCM}(N, D \times \overline{TP}_{\max})$, where the additional factor $D = \sum_{i=1}^K d_i \times CP(P_i)$ accounts for gradients reduce-scatter across all model replicas. The LNK can be established still using the *send*, *receive* and *local movement* primitives. Notably, the LNK mechanism operates independently of $PP(P_i)$, as PP doesn't partition individual parameters into slices. From the perspective of heterogeneous strategy execution, the LNK operator resides in different stages across training pipelines. Consequently, communication occurs only after the slowest pipeline reaches the LNK operator (the balancing across pipelines will be addressed in §6).

It is noteworthy that our optimization-propagation disaggregation is fundamentally a generalization of current model states

sharding techniques (i.e., ZeRO [79] and FSDP [104])— when using homogeneous strategies, the LNK operator in the *pull* will be translated into an all-gather operation, and the LNK operator in the *push* will be pruned (leaving a reduce-scatter operation), which are the same as current techniques. For heterogeneous strategies, introducing the LNK operator incurs additional overhead to resolve the discrepancy between strategies. However, the *send* and *receive* communication can be overlapped with computation during propagation, and the *local movement* incurs only minimal latency, so the overhead is small (as evaluated in §8.4).

Last but not least, theoretically speaking, the communication during *pull* and *push* operations is bounded by the model size and does not scale with the number of GPUs. And the worst-case communication volume is exactly the same as that of current model states sharding techniques. Furthermore, computation-communication overlap can be achieved when each micro-batch in the pipeline with P_i contains more than $TP(P_i) \times CP(P_i) \times \frac{F}{B}$ tokens, where F is the GPU's half-precision compute capability (FLOPS), and B is the slowest interconnect bandwidth (bytes/sec). This condition is independent of the model size or the number of GPUs. Due to the space constraint, detailed proof and analysis are provided in Appendix B [1].

5.3 Subgraphs

Although optimization-propagation disaggregation offers the ability to resolve the communication patterns needed for each heterogeneous strategy, it still poses the challenge of how to manage such diverse and complex optimization-propagation interactions, given that there would be numerous candidate heterogeneous strategies. To tackle this challenge, we introduce subgraphs, which are subsets of the complete computation graph. The key observation is that different strategies only have unique patterns in some specific operators, while most user-defined computational operators remain unchanged. Therefore, subgraphs are developed to encapsulate these unique operators from different strategies for efficient representation, management and scheduling.

In particular, we define three subgraph types: *Pull Parameter*, *Push Gradient*, and *Communicate Activation*. Operators for interactions between optimization and propagation phases, like LNK, are placed in the first two, while activation-related communication operators (e.g., all-gather and reduce-scatter for TP, send-receive for PP) are included in the third.

Subgraphs instantiation. As illustrated in Figure 10, users start by defining various computational operators to describe the training model. We then identify all potential points in the computational graph where different heterogeneous strategies might intervene and preserve several empty subgraphs (e.g., between transformer modules for potential pipeline parallel, before and after matrix multiplication for potential tensor parallel). Given a strategy S , the user-defined operators are directly integrated into a new, strategy-specific graph. We then determine how to instantiate all predefined subgraphs with appropriate operators. For example, as depicted at the top of Figure 10, a data transfer operator and a strategy-specific LNK operator are added to instantiate a *Pull Parameter* subgraph. This instantiation process ultimately forms a complete graph for

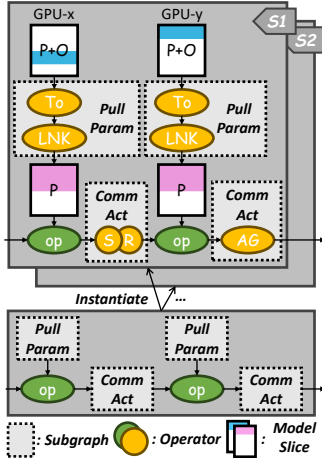


Figure 10: The instantiation of *Pull Parameter* and *Communicate Activation* subgraphs during forward propagation. This involves inserting custom operators (e.g., send-receive, all-gather, LNK) to support arbitrary propagation strategies and optimization-propagation interactions.

a specific S , enabling different GPUs to extract and execute their respective operators.

Subgraphs scheduling. During execution, we optimize overlap by controlling the execution order of subgraphs and computational operators. Each subgraph can be treated as a cohesive unit and integrated into the scheduler as a fused operator. As illustrated in Figure 11, by using a typical `cudaStream` to execute the subgraphs and interleaving them with computational operators, we enable communication within subgraphs to overlap with computation, thereby reducing latency.

6 Two-stage Sequence Assignment

In this section, we present our two-stage sequence assignment approach. Specifically, given a mini-batch of sequences and an arbitrary strategy, we establish an optimization problem to pack sequences precisely, and ensure balanced sequence dispatching based on load estimates of each pipeline, addressing both intra- and inter-pipeline imbalances. In each iteration, we apply the assignment to all candidate strategies, estimate the propagation latency, and select the optimal one.

6.1 Sequence Packing within Pipeline

We first introduce the packing strategy within each pipeline. Assume there are U sequences with lengths $\{l_i\}_{i=1}^U$ assigned to a pipeline configured with parallel scheme P . To pack these sequences into multiple micro-batches, there are two essential terms to determine: the number of micro-batches to craft (denoted as V) and how each sequence should be assigned.

Cost Model. Following prior works on automatic parallel training [57, 60, 103, 105], we develop two profiling-based cost models to estimate memory usage and latency. For a given P , there is a maximum sequence length it can support, denoted as $\text{MaxLen}(P)$ (i.e., sequence exceeding this length will cause OOM on P), which

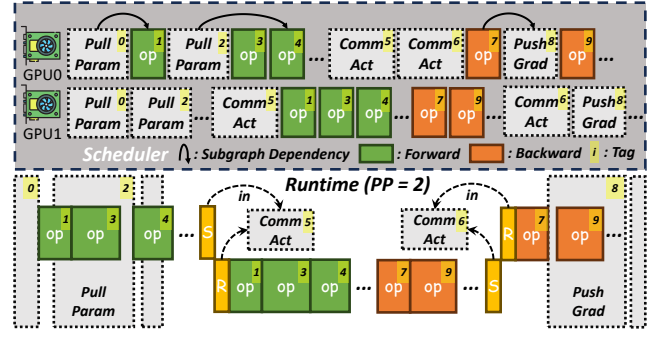


Figure 11: Scheduling subgraphs with pipeline parallel (PP). In the first PP stage, *Pull Parameter* subgraphs are scheduled immediately after the last operators requiring the parameter, and *Push Gradient* subgraphs follow right after the operators that produce or accumulate the gradient. In subsequent PP stages, *Pull Parameter* and *Push Gradient* are grouped at the beginning and end, respectively, to utilize inherent PP bubbles. This scheduling scheme effectively hides communication latency within the *Pull Parameter* and *Push Gradient* subgraphs.

can be determined in advance using our memory model (detailed in Appendix C.1 [1]). Additionally, we introduce a latency model, $T(l, P)$, which estimates the time taken to run a sequence of length l on P for a single forward and backward propagation of a single pipeline stage (detailed in Appendix C.2 [1]). Similar to prior works, re-profiling is required for different model architectures but is performed as a one-shot process per architecture. This overhead is worthwhile given the significant speedup in large-scale training. Moreover, this process is independent of the cluster size, as scaling the cluster only increases the number of pipelines, without affecting the set of valid P or the sequence length l used during training.

Solving for one V . The problem is easy to solve if the number of micro-batches V is given. Specifically, as mentioned in §2.1, by employing the varlen functionality of FlashAttention v2 [18, 19], the latency of a packed sequence can be approximated as the sum of latency for each original sequence. Meanwhile, to address the intra-pipeline imbalance, our goal is to minimize the duration of the longest-running micro-batch. Thus, we can formulate the following problem on top of the minimum sum partition (MSP) problem:

$$\begin{aligned} \arg \min_{o_{i,j}} \max_{1 \leq j \leq V} & \left(\sum_{1 \leq i \leq U} o_{i,j} \times T(l_i, P) \right) \times (PP(P) - 1 + V) \\ \text{s.t.} \quad & \sum_{1 \leq j \leq V} o_{i,j} = 1, o_{i,j} \in \{0, 1\}, \forall i \in [1, U] \\ & \sum_{1 \leq i \leq U} o_{i,j} \times l_i \leq \text{MaxLen}(P), \forall j \in [1, V] \end{aligned} \quad (1)$$

where $o_{i,j} \in \{0, 1\}$ indicates whether the i -th sequence is packed into the j -th micro-batch. Compared to the MSP problem, the problem defined in Eq. (1) additionally ensures that the length after packing must not exceed $\text{MaxLen}(P)$, and the term $PP(P) - 1 + V$ represents the propagation latency of the entire pipeline.

Enumeration of V . To locate the optimal number of micro-batches V^* , we can simply enumerate the value for $V \in [1, U]$, solve Eq. (1), and choose the best one.

Note that Eq. (1) is an Integer Linear Programming (ILP) problem, but its complexity is only related to the number of sequences assigned to P (i.e., U) and the number of micro-batches (i.e., V). These do not increase with cluster size, and the enumeration can be done in parallel, ensuring fast problem-solving even on large-scale clusters (detailed in Appendix F [1]).

6.2 Sequence Dispatching across Pipelines

Next, we consider how the sequences in a mini-batch should be dispatched across the pipelines to address the inter-pipeline imbalance. A naïve approach is to bucket sequences by length. However, given the long-tail distribution of sequence lengths (where long sequences are rare, as shown in Figure 1), pipelines configured for long sequences may receive too few assignments during iterations dominated by short sequences. To improve load balancing, we need a more flexible approach that allows short sequences to be processed by any pipeline. Consider a specific strategy $S = P_1 + \dots + P_D$ satisfying $\text{MaxLen}(P_1) \geq \dots \geq \text{MaxLen}(P_D)$. Assume a mini-batch contains B sequences with lengths $\{l_i\}_{i=1}^B$. We denote J_i as the number of pipelines that the i -th sequence can run on, satisfying $\text{MaxLen}(P_{J_i}) \geq l_i > \text{MaxLen}(P_{J_i+1})$. Thus, l_i can be dispatched to any of $P_1, \dots, P_{J_i}$. Since the pipelines execute concurrently, our goal is to dispatch the B sequences across the D pipelines while minimizing the propagation latency of the longest-running pipeline.

However, the propagation latency of each pipeline cannot be directly computed without solving the packing within it (i.e., §6.1). Fortunately, it is feasible to estimate the lower bound of latency. In particular, let $m_{i,j} \in \{0, 1\}$ indicate whether the i -th sequence is assigned to P_j , then the lower bound of running time of the j -th pipeline can be expressed as (detailed derivation in Appendix E.1 [1]):

$$\sum_{i=1}^B m_{i,j} \times \tau(l_i, P_j) + \tau\left(\max_{1 \leq i \leq B} \{m_{i,j} \times l_i\}, P_j\right) \times (\text{PP}(P_j) - 1) \quad (2)$$

Denote Eq. (2) as $\text{LowerBound}(\{m_{i,j}\}_{i=1}^B, \{l_i\}_{i=1}^B, P_j)$, then the dispatching can be formulated as a (0,1)-ILP problem:

$$\begin{aligned} \arg \min_{m_{i,j}} \quad & \max_{1 \leq j \leq D} \text{LowerBound}(\{m_{i,j}\}_{i=1}^B, \{l_i\}_{i=1}^B, P_j) \\ \text{s.t.} \quad & \sum_{1 \leq j \leq J_i} m_{i,j} = 1, \forall i \in [1, B] \\ & m_{i,j} \in \{0, 1\}, \forall i \in [1, B], \forall j \in [1, D] \end{aligned} \quad (3)$$

The number of decision variables of Eq. (3) is linear w.r.t. number of sequences in a mini-batch (i.e., B) and the number of pipelines in the strategy (i.e., D). In practice, as the cluster scale expands, both will increase proportionally, making the assignment problem-solving cost unacceptable in large-scale scenarios. Fortunately, there are extensive studies about how to approximately solve assignment problems in polynomial time with a guaranteed error bound [13, 26, 84]. Thus, we follow previous works to solve Eq. (3) approximately, which involves running multiple random trials to greedily dispatch sequences and selecting the best trial. Empirical results demonstrate that the approximation errors are consistently lower than 10% compared to directly solving Eq. (3). More details about our approximate solution can be found in Appendix E and Appendix F [1].

7 Data Distribution-aware Strategy Proposal

Ideally, for each iteration, we could explore all possible strategies, estimate their propagation latency via the two-stage sequence assignment, and find the optimal one. However, given the vast size of the strategy space $\mathbb{S}$, this is impractical. Thus, we analyze the dataset in advance and propose a refined subset $\hat{\mathbb{S}} \subset \mathbb{S}$, which only contains the most promising candidates.

Problem definition. Our proposal is based on two considerations regarding intra- and inter-iteration imbalances: First, since intra-iteration imbalance typically reflects the expected distribution of the dataset, we aim to preserve strategies that demonstrate an overall advantage over the entire dataset. Second, due to inter-iteration imbalance, characterized by variations in maximum sequence lengths, we aim to identify non-unique strategies across different maximum lengths. Then, we have the following problem.

PROBLEM 1. Given the number of GPUs N , the dataset $\mathbb{D}$, and an arbitrary sequence length L . With sequence lengths randomly sampled within $[0, L]$ over $\mathbb{D}$, we seek the optimal heterogeneous strategy $S_{L|\mathbb{D}}^*$ from $\mathbb{S}$ that minimizes the expected cost. Then the proposed subset is defined as the union: $\hat{\mathbb{S}} = \bigcup_{L \leq L_{\max}} \{S_{L|\mathbb{D}}^*\}$, where $L_{\max}$ is the maximum sequence length to support (i.e., context length).

We first simplify the problem by forcing each pipeline to process only the sequences within a specific length interval, rather than any arbitrary lengths. This allows the problem to be solved via dynamic programming. Later we lift the restriction by continuous relaxation, enhancing the solution.

Dynamic programming. We define $t[n][l]$ as the minimum achievable propagation latency using up to n GPUs to process all sequences in the dataset with lengths not exceeding l . Our goal is to determine the value $t[N][L]$. The initial states are obvious that $t[n][0] = 0, \forall n > 0$ and $t[0][l] = \infty, \forall l > 0$. Next, we focus on the state transition for dynamic programming. Let $\mathbb{D}_{(l-l', l]}$ denote the sub-dataset containing sequences within the interval $(l-l', l]$, and assume the parallel scheme space comprises K unique schemes $\{P_k\}_{k=1}^K$. Based on our simplification, we only need to consider assigning this sub-dataset to one kind of scheme. Suppose it is assigned to d pipelines of P_k and P_k requires $N(P_k)$ GPUs, then this assignment consumes $d \times N(P_k)$ GPUs in total. Besides, we must ensure $\text{MaxLen}(P_k) \geq l$. Then the propagation latency of this sub-dataset can be approximated as $\frac{1}{d} \sum_{x \in \mathbb{D}_{(l-l', l]}} \tau(x, P_k)$. Meanwhile, the minimum propagation latency for the remaining sequences is $t[n - d \times N(P_k)][l - l']$. Therefore, we have the following state transition expression:

$$\begin{aligned} t[n][l] &= \min \left(t[n-1][l], \min_{(k,d) \in \text{Cond}_{n,l}, l' < l} \{T_{k,d,l'}\} \right), \text{ where} \\ T_{k,d,l'} &= \max \left(t[n - d \times N(P_k)][l - l'], \frac{1}{d} \sum_{x \in \mathbb{D}_{(l-l', l]}} \tau(x, P_k) \right) \\ \text{Cond}_{n,l} &= \{(k, d) \mid \text{MaxLen}(P_k) \geq l, d \times N(P_k) \leq n\} \end{aligned} \quad (4)$$

The use of $\max(\cdot, \cdot)$ in $T_{k,d,l'}$ is because the slowest pipeline determines the overall latency. Based on the dynamic programming, we also obtain the corresponding optimal strategy $S[n][l]$ upon the

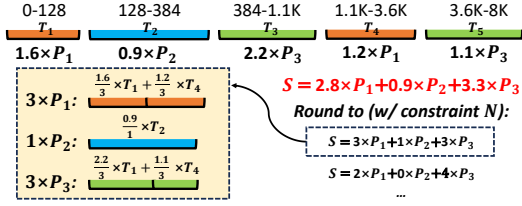


Figure 12: Dynamic programming with non-integer values. Different colors represent various P . The segments indicate sequences within each dataset interval, with their lengths corresponding to latency. Rounding the number of P requires scaling the results.

calculation of $t[n][l]$, and $S[N][L]$ will be viewed as the solution $S_{L|D}^*$ under L in Problem 1.

By leveraging the optimal sub-structure property of dynamic programming, once we determine $S[N][L_{\max}]$, we obtain $S[N][L]$ for all $L \leq L_{\max}$. We can then select all unique $S[N][L]$ from $L = 1$ to $L = L_{\max}$ to form $\hat{S}$.

Continuous relaxation. Based on the dynamic programming, we lift the restriction that each pipeline can only handle a sequence over a single length interval. Specifically, we execute the dynamic programming with non-integer values for n and d . This means sequences in a length interval can use “a portion” of a pipeline, enabling a complete pipeline to support multiple intervals. Consequently, the final solution $S[N][L]$ might involve having non-integer numbers, as exemplified in Figure 12. In such cases, we round and adjust to include all nearby integer solutions in $\hat{S}$, while ensuring that the total number of GPUs does not exceed N .⁵

Undoubtedly, even though the original non-integer solution achieves optimal results, rounding compromises the optimality. Additionally, the data distribution in different iterations does not perfectly align with that of the entire dataset. Due to these limitations, we can only suggest potential strategy candidates, whereas the two-stage sequence assignment is still necessary for selecting optimal strategies.

8 Evaluation

8.1 Experimental Setup

Environment. Our experiments are conducted with 8 nodes, each containing 8 Nvidia A800 GPUs, with a total number of 64 GPUs. GPUs within a node are connected via NVLink with a bandwidth of 400 GB/s, and nodes are interconnected using Infini-Band with a bandwidth of 200 GB/s.

Baselines. We compare Hydraulics with Megatron-LM [49, 70, 85], DeepSpeed [39, 80], and HotSPa [30]. Both Megatron-LM and DeepSpeed employ a single, static homogeneous strategy throughout the training process. HotSPa uses heuristics to configure multiple homogeneous strategies and enables switching between them within every iteration. However, due to the overhead of switching, HotSPa is limited to setting 3-4 strategies. We enumerate all competitors’ parallel strategies and select the optimal one to maximize baseline performance (see Appendix G [1] for details).

⁵The whole algorithm’s worst-case time complexity is $O(KN^2L_{\max}^2)$. In our experiments, we set the step size of n and d to 0.1 and l to 128, with the actual time cost detailed in Appendix F [1].

Models and datasets. Most existing large models are based on the decoder-only Transformer architecture. We select the widely used LLaMA2 series models and conduct experiments with 7B, 13B and 32B model sizes [87], using CommonCrawl [86] and GitHub [17], which are widely used open-source datasets and have served as the major training corpora for many prestigious Transformer models.

Workloads and metrics. By default, we sample 100K tokens per iteration⁶ with a 32K context length, and use the Adam optimizer [48]. We conduct 20 warm-up iterations and then summarize the throughput (tokens per second) or per-iteration latency over 100 iterations.

8.2 End-to-end Experiments

We first evaluate Hydraulics against three baselines across various configurations (baselines’ strategies are detailed in Appendix G [1]). The results in Figure 13 demonstrate that Hydraulics achieves 1.32-2.66 $\times$ improvement over the best baselines.

As DeepSpeed and Megatron-LM are limited to a single fixed strategy, they must adopt less efficient strategies to accommodate the memory consumption of long sequences, degrading the overall efficiency given most sequences are relatively short. Although HotSPa is capable of using multiple homogeneous strategies within an iteration to train sequences of varying lengths, it requires multiple times of model hot switching. This incurs apparent overheads, especially when per-iteration time is short (e.g., in the “GitHub 7B 8GPUs F1” scenario, the average iteration time is merely 6.7s, while nearly 2s is spent on hot switching).⁷ On the contrary, Hydraulics handles variable-length sequences using dynamic heterogeneous strategies, while eliminating the overhead of strategy transition via disaggregating optimization and propagation.

Moreover, none of the baselines consider fine-grained sequence management. This problem is particularly severe for HotSPa— in some cases, there are only 1-2 long sequences per iteration, yet HotSPa is forced to adopt an individual strategy with all GPUs to process such long sequences, leaving a few GPUs with no sequences to process (e.g., in the “CommonCrawl 32B 64GPUs F2” scenario, HotSPa adopts $\langle TP, PP, CP \rangle \times DP = \langle 16, 1, 1 \rangle \times 4$ to process sequences ranging 16K-32K in length, often resulting in 3/4 or 1/2 GPUs being completely unused). Instead, Hydraulics, with the two-stage sequence assignment, meticulously organizes the training workloads of different sequences for better performance, as we will evaluate more in-depth later in §8.4.

8.3 Scalability

We then assess the scalability of different systems by examining three factors: (a) number of GPUs, (b) maximum sequence length, and (c) tokens per iteration. Using the LLaMA 7B model and the CommonCrawl dataset, we start with a default setup of 16 GPUs, 32K maximum sequence length, and 200K tokens per iteration. By systematically altering one factor while maintaining others, we gain insights into each factor’s impact. Results are presented in Figure 14. Additionally, we analyze how (a) data imbalance and (b) problem-solving cost scale with increasing training workloads,

⁶We consider 100K tokens per iteration for our 64-GPU cluster, scaled proportionally from the 16M used in LLaMA with an 8192-GPU cluster [27].

⁷A more detailed analysis of HotSPa’s switching overhead can be found in Appendix H [1].

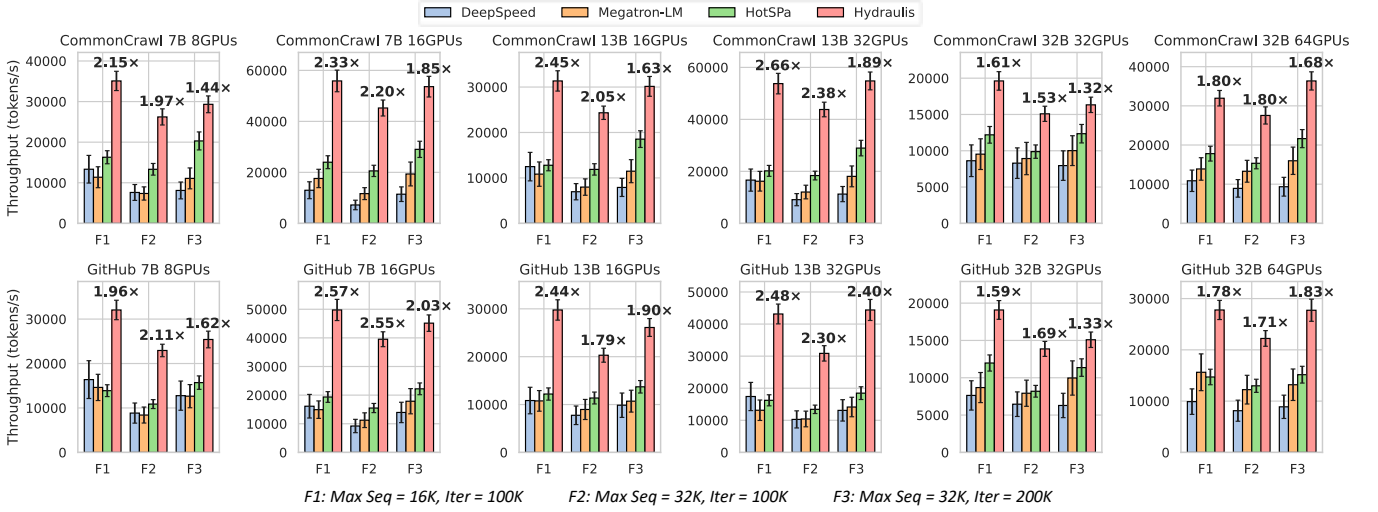


Figure 13: End-to-end evaluation. We test the workloads of three configurations, F1-F3, in each scenario. “Max Seq” represents the number of tokens in the longest sequence, and “Iter” denotes the number of tokens sampled (a.k.a. batch size) in a single iteration. Error bars indicate the standard deviation across iterations.

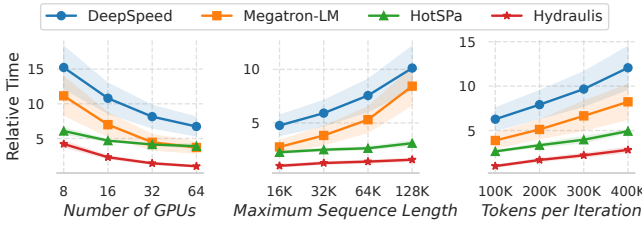


Figure 14: Comparison of scalability across various factors, using the fastest one as the reference for relative changes. Shaded areas indicate the standard deviation across iterations.

particularly when both the cluster size (number of GPUs) and batch size (tokens per iteration) grow proportionally.

Assessment: number of GPUs. As the number of GPUs increases, all systems experience a sub-linear reduction in latency due to the communication overhead introduced by parallelism. Specifically, for HotSPa, the communication overhead from hot switching does not decrease with the number of GPUs, causing its overall performance to be surpassed by Megatron-LM when using 64 GPUs. In contrast, Hydraulis introduces minimal additional communication overhead (only the LNK operators), which can overlap with computation. Consequently, its latency consistently outperforms other baselines.

Assessment: maximum sequence length. As the maximum sequence length grows, DeepSpeed and Megatron-LM resort to less efficient strategies, resulting in a significant increase in latency. Notably, when Megatron-LM scales from 64K to 128K, it must enable cross-node TP, which greatly increases latency. In contrast, by customizing parallel strategies for sequences of different lengths, HotSPa and Hydraulis are less sensitive to maximum sequence length changes, with Hydraulis demonstrating superior performance over HotSPa.

Assessment: tokens per iteration. As the number of tokens per iteration increases, all systems experience an almost linear increase in latency. However, HotSPa and Hydraulis exhibit a slightly lower growth rate, as they take account of the long-tail distribution of sequence lengths within iterations.

Analysis: data imbalance. When scaled to more GPUs, each iteration typically samples more training data, which may mitigate inter-iteration imbalance because longer sequences become more likely to be selected. To assess this, we conduct a 1024-GPU simulation with 16M tokens sampled per iteration (detailed in Appendix A.1 [1]). The results show that the maximum sequence length fluctuates between 16K-32K, indicating that inter-iteration imbalance still remains. Meanwhile, intra-iteration imbalance persists due to the dataset’s inherent sequence length distribution, while intra- and inter-pipeline imbalances originate from the discrepancy between the attention mechanism’s quadratic time complexity and linear memory complexity relative to sequence length. Since these imbalances are scale-independent, they persist at larger scales.

Analysis: problem-solving cost. We also conduct simulation experiments to evaluate the scalability of problem solving when scaled to 1024 GPUs (detailed in Appendix F [1]), which show that the problem-solving cost is well-bounded. To be specific, the strategy proposal (§7) incurs acceptable overhead (< 2 minutes) as it executes only once prior to training. Meanwhile, both sequence packing (§6.1) and dispatching (§6.2) operate on CPUs and can be fully overlapped with GPU training (i.e., the problem-solving for subsequent iterations occurs concurrently with current iteration training). Furthermore, larger cluster sizes provide additional CPU resources to increase the concurrency of problem solving, allowing us to further amortize the solving time per iteration.

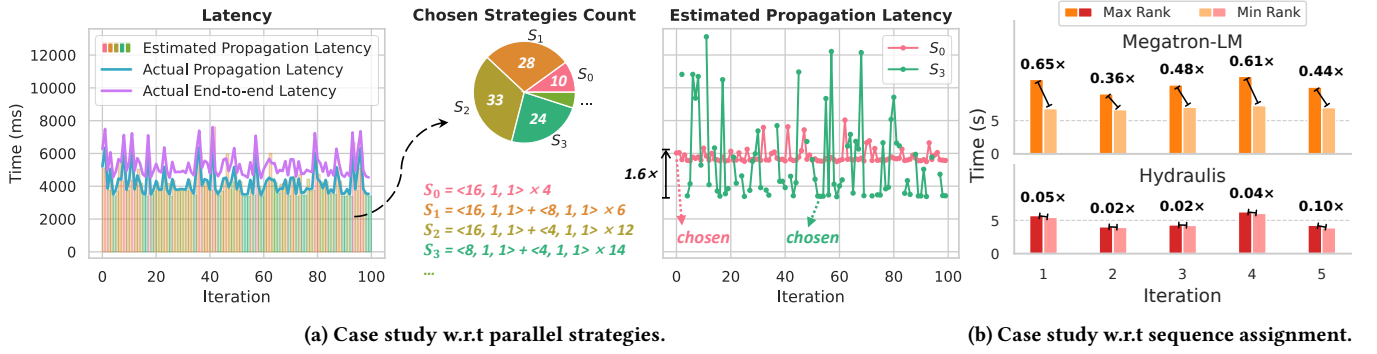


Figure 15: (a) Latency over 100 training iterations, with colors indicating the strategy selected for each iteration. (b) Comparison of load balancing between Hydraulis and Megatron-LM. We present the maximum and minimum propagation latency among all ranks for 5 iterations.

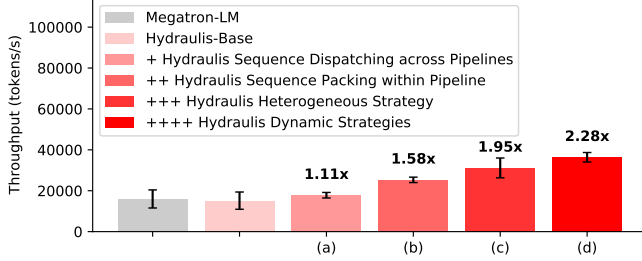


Figure 16: Ablation study. Error bars indicate the standard deviation across iterations. Total throughput changes as different components of Hydraulis are incrementally integrated.

8.4 Performance Interpretation

To further evaluate the impact of the proposed techniques, we conduct experiments by training the 32B model with the Common-Crawl dataset over 64 GPUs.

Case study. Figure 15a shows the optimal strategies selected by Hydraulis, along with the corresponding latencies, and summarizes the frequency of different strategies over 100 iterations. It can be seen that: (a) the actual propagation latency closely matches our estimation in most cases, and (b) variations across iterations lead to different strategy choices.

To understand why using different strategy choices contributes, we compare the estimated latencies of two strategies, S_0 and S_3 , over different iterations. While S_3 achieves $1.6\times$ speedup than S_0 in most cases due to its heterogeneous design, its maximum supportable sequence length is shorter than S_0 , making it inapplicable for certain iterations. Besides, when the sampled batch contains fewer short sequences, it’s hard for S_3 to strike a balance (i.e., many sequences can only be processed by the pipeline with a TP degree of 8 while the other pipelines receive very few sequences), making it less efficient than S_0 . These findings highlight the advantages of heterogeneous strategies, as well as the trade-off between latency and maximum sequence length across different strategies, demonstrating the benefits of supporting dynamic heterogeneous strategies for overall optimal performance.

Additionally, we examine the load balancing. Figure 15b shows that Megatron-LM, with max-length packing and even dispatching of packed sequences, results in a latency difference of $0.36\text{--}0.65\times$

Table 1: Breakdown of latency per iteration for different training strategies. “P” denotes propagation (Figure 3b step 3), “O” refers to optimization (Figure 3b step 6), and “O-P” represents the remaining overhead from the interaction between the two phases (Figure 3b steps 1-2 and 4-5). “E2E” stands for end-to-end training latency.

	Homo	Hetero w/o Overlap	Hetero w/ Overlap
P	6.78s	4.29s (1.58 $\times$)	4.29s (1.58 $\times$)
O	0.02s	0.02s (1.00 $\times$)	0.02s (1.00 $\times$)
O-P	1.11s	1.71s (0.69 $\times$)	1.18s (0.94 $\times$)
E2E	7.91s	6.02s (1.31 $\times$)	5.49s (1.44 $\times$)

between the slowest and fastest ranks. In contrast, Hydraulis reduces this gap to just $0.02\text{--}0.10\times$, verifying the significance of the two-stage sequence assignment.

Ablation study. Figure 16 presents an ablation study by progressively enabling different techniques in Hydraulis to evaluate their contributions: (a) Starting with a static, homogeneous parallel strategy, we dispatch sequences of varying lengths according to Eq. (3) and pack them to the maximum length. (b) Instead of packing sequences to the maximum length, we apply packing based on the results of Eq. (1). (c) We replace the original homogeneous parallel strategy with a heterogeneous strategy that supports the maximum sequence length. (d) Finally, we enable dynamic heterogeneous strategies, utilizing all Hydraulis designs.

The results demonstrate that, on the one hand, when no optimizations are applied (i.e., Hydraulis-Base), our performance is on par with Megatron-LM, confirming the fairness of the comparison. On the other hand, consistent improvements are observed across (a)-(d), highlighting the critical role each system component plays in boosting overall performance.

Latency breakdown. We further investigate the effectiveness of optimization-propagation disaggregation compared to conventional model states sharding, as well as the benefits brought by subgraphs scheduling. As shown in Table 1, we first adopt a fixed homogeneous parallel strategy (with the two-stage sequence assignment enabled), which utilizes the conventional all-gather and reduce-scatter interaction patterns (Figure 3a). Then, we employ dynamic heterogeneous strategies, introducing additional LNK operators (Figure 3b). Besides, we experiment both with and without

subgraphs scheduling to measure the latency improvements from overlapping *pull* and *push* interactions with computation.

It can be observed that the use of dynamic heterogeneous strategies improves the performance of propagation. However, the interaction between optimization and propagation becomes slower due to the additional communication overhead introduced by LNK. Despite this, the overhead remains relatively small and can be efficiently reduced by overlapping with computation, leading to only a minimal increase in latency. Moreover, this additional latency does not scale with the number of tokens per iteration, demonstrating the viability of disaggregating optimization and propagation.

9 Related Work

Optimizations of distributed training. Numerous studies aim to enhance the efficiency of distributed training. In the realm of parallel methods, advancements in pipeline parallel focus on minimizing bubble time through enhanced scheduling [42, 52, 77, 95], while tensor parallel has progressed by overlapping communication with computation [12, 40, 43, 91]. In the context of model states sharding, efforts have been made to balance memory usage and latency through finer-grained sharding [14], efficient communication [65, 89], and multi-level storage solutions [81, 103]. Notably, these approaches are compatible with ours. There are also efforts focusing on automatically searching for the optimal parallel strategies [41, 56, 66, 92, 94, 105]. However, they only consider homogeneous strategies and overlook the data imbalance issues.

Resources disaggregation. The idea of disaggregating storage from compute resources has been well-studied in cloud computing [34, 59, 82] and utilized in many AI applications [4, 45]. Similar concepts have also gained attention in large Transformer model inference, where disaggregating the prefilling and decoding phases allows for better throughput [37, 75, 106]. In our system, we adopt a similar design philosophy by separating the storage-heavy optimization phase from the compute-intensive propagation phase, enabling flexible execution and transitions of parallel strategies.

Intermediate representations of parallel strategies. Graph-based intermediate representations are widely utilized in deep learning systems [2, 35, 44, 53]. By analyzing and optimizing the graphs, models can discover enhanced and specialized parallel strategies. Hydraulics extends this by simultaneously deducing and expressing graphs for different strategies. Using subgraphs, it distinctly highlights the unique communication patterns of each strategy. Besides, any graph-based optimization techniques can be applied to Hydraulics, allowing for further refinement for each strategy.

10 Conclusion and Future Work

In this work, we addressed the often-overlooked issues of data sampling and packing imbalances in existing training systems, and introduced Hydraulics, a novel system that leverages parallel computing and data management co-designs to optimize the training performance of large Transformer models. We proposed training with dynamic heterogeneous parallel strategies to address the data sampling imbalance and a two-stage sequence assignment to tackle the data packing imbalance. Empirical results show that Hydraulics

can achieve up to a 2.66 $\times$ increase in throughput compared to state-of-the-art training systems.

As a possible future direction, we intend to expand our strategy space by integrating more parallel methods (e.g., expert parallel [54, 63]), as our system designs are fully compatible with more forms of parallelism. We also plan to tackle data imbalance challenges in diverse domains (e.g., in the video domain, the significant variation in sequence lengths has already attracted considerable attention [15, 62, 83, 97]). We believe our work can be adapted to a broader range of scenarios and applications.

Acknowledgments

This work is supported by National Natural Science Foundation of China (U23B2048, 62402011), PKU-Tencent joint research Lab, and High-performance Computing Platform of Peking University. Fangcheng Fu and Bin Cui are the corresponding authors.

References

- [1] 2025. Full Version (with Appendix) of Hydraulics. <https://arxiv.org/pdf/2412.07894>.
- [2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283. <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>.
- [3] Anurag Arnab, Mostafa Dehghani, Georg Heigold, et al. 2021. ViViT: A Video Vision Transformer. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. <https://arxiv.org/abs/2103.15691>.
- [4] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiri Šimša, and Chandramohan A. Thekkath. 2023. tf.data service: A Case for Disaggregating ML Input Data Processing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC)*. 358–375. <http://dx.doi.org/10.1145/3620678.3624666>.
- [5] Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. 2020. wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations. In *Advances in Neural Information Processing Systems (NeurIPS)*. <https://arxiv.org/abs/2006.11477>.
- [6] Hao Bai. 2022. Modern Distributed Data-Parallel Large-Scale Pre-training Strategies For NLP models. *arXiv preprint arXiv:2206.06356* (2022). <https://arxiv.org/abs/2206.06356>.
- [7] Yushi Bai, Xin Lv, Jiajie Zhang, Yuze He, Ji Qi, Lei Hou, Jie Tang, Yuxiao Dong, and Juanzi Li. 2024. LongAlign: A Recipe for Long Context Alignment of Large Language Models. *arXiv preprint arXiv:2401.18058* (2024). <https://arxiv.org/abs/2401.18058>.
- [8] Gedas Bertasius, Heng Wang, and Lorenzo Torresani. 2021. Is Space-Time Attention All You Need for Video Understanding?. In *Proceedings of the International Conference on Machine Learning (ICML)*. <https://arxiv.org/abs/2102.05095>.
- [9] William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. 2023. Striped Attention: Faster Ring Attention for Causal Transformers. *arXiv preprint arXiv:2311.09431* (2023). <https://arxiv.org/abs/2311.09431>.
- [10] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS)*. <https://arxiv.org/abs/2005.14165>.
- [11] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, et al. 2020. End-to-End Object Detection with Transformers. In *European Conference on Computer Vision (ECCV)*. <https://arxiv.org/abs/2005.12872>.
- [12] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. 2024. Centauri: Enabling Efficient Scheduling for Communication-Computation Overlap in Large Model Training via Communication Partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 178–191. <https://doi.org/10.1145/3620666.3651379>.
- [13] Cong Chen, Paolo Penna, and Yinfeng Xu. 2020. Online Scheduling of Jobs with Favorite Machines. <http://dx.doi.org/10.1016/j.cor.2019.104868>.
- [14] Qiaoling Chen, Diandian Gu, Guoteng Wang, Xun Chen, YingTong Xiong, Ting Huang, Qinghao Hu, Xin Jin, Yonggang Wen, Tianwei Zhang, and Peng Sun. 2024. InternEvo: Efficient Long-sequence Large Language Model Training via Hybrid Parallelism and Redundant Sharding. <https://arxiv.org/abs/2401.09149>.

- [15] Shimin Chen, Xiaohan Lan, Yitian Yuan, Zequn Jie, and Lin Ma. 2024. TimeMarker: A Versatile Video-LLM for Long and Short Video Understanding with Superior Temporal Localization Ability. *arXiv preprint arXiv:2411.18211* (2024). <https://arxiv.org/abs/2411.18211>.
- [16] Rohan Choudhury, Guanglei Zhu, Sihan Liu, Koichiro Niinuma, Kris M. Kitani, and László Jeni. 2024. Don't Look Twice: Faster Video Transformers with Run-Length Tokenization. In *Advances in Neural Information Processing Systems (NeurIPS)*. Spotlight, <https://arxiv.org/abs/2411.05222>.
- [17] CodeParrot. [n.d.]. GitHub Dataset. <https://huggingface.co/datasets/codeparrot/github-code>.
- [18] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. *arXiv preprint arXiv:2307.08691* (2023). <https://arxiv.org/abs/2307.08691>.
- [19] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*. <https://arxiv.org/abs/2205.14135>.
- [20] Harm de Vries. 2023. In the long (context) run. <https://www.harmdevries.com/post/context-length/>.
- [21] Mostafa Dehghani, Basil Mustafa, Josip Djolonga, Jonathan Heek, Matthias Minderer, Mathilde Caron, Andreas Steiner, Joan Puigcerver, Robert Geirhos, Ibrahim Alabdulmohsin, Avital Oliver, Piotr Padlewski, Alexey Gritsenko, Mario Lučić, and Neil Houlsby. 2023. Patch n' Pack: NaViT, a Vision Transformer for any Aspect Ratio and Resolution. *arXiv preprint arXiv:2307.06304* (2023). <https://arxiv.org/abs/2307.06304>.
- [22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. <https://arxiv.org/abs/1810.04805>.
- [23] Li Dong, Shuang Xu, and Bo Xu. 2018. Speech-Transformer: A No-Recurrence Sequence-to-Sequence Model for Speech Recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. <https://arxiv.org/abs/1804.07133>.
- [24] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, et al. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations (ICLR)*. <https://arxiv.org/abs/2010.11929>.
- [25] Fei Du, Xin-Jian Ma, Jing-Ru Yang, Yi Liu, Chao-Ran Luo, Xue-Bin Wang, Hai-Ou Jiang, and Xiang Jing. 2024. A Survey of LLM Datasets: From Autoregressive Model to AI Chatbot. *Journal of Computer Science and Technology* 39, 3 (2024), 542–566. doi:10.1007/s11390-024-3767-3
- [26] Ran Duan and Seth Pettie. 2014. Linear-time approximation for maximum weight matching. *Journal of the ACM (JACM)* 61, 1 (2014), 1–23.
- [27] Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024). <https://arxiv.org/abs/2407.21783>.
- [28] Haoqi Fan, Bo Xiong, Karttikeya Mangalam, et al. 2021. Multiscale Vision Transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. <https://arxiv.org/abs/2104.11227>.
- [29] Jiarui Fang and Shangchun Zhao. 2024. USP: A Unified Sequence Parallelism Approach for Long Context Generative AI. *arXiv preprint arXiv:2405.07719* (2024). <https://arxiv.org/abs/2405.07719>.
- [30] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2024. Enabling Parallelism Hot Switching for Efficient Training of Large Language Models. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*. <https://dl.acm.org/doi/pdf/10.1145/3694715.3695969>.
- [31] Diandian Gu, Peng Sun, Qinghao Hu, Ting Huang, Xun Chen, Yingdong Xiong, Guoteng Wang, Qiaoling Chen, Shangchun Zhao, Jiarui Fang, Yonggang Wen, Tianwei Zhang, Xin Jin, and Xuanzhe Liu. 2024. LoongTrain: Efficient Training of Long-Sequence LLMs with Head-Context Parallelism. *arXiv preprint arXiv:2406.18485* (2024). <https://arxiv.org/abs/2406.18485>.
- [32] Lei Guan, Dong-Sheng Li, Ji-Ye Liang, Wen-Jian Wang, Ke-Shi Ge, and Xi-Cheng Lu. 2024. Advances of Pipeline Model Parallelism for Deep Learning Training: An Overview. *Journal of Computer Science and Technology* 39, 3 (2024), 567–584. doi:10.1007/s11390-024-3872-3
- [33] Anmol Gulati, James Qin, Chung-Cheng Chiu, et al. 2020. Conformer: Convolution-augmented Transformer for Speech Recognition. In *Proceedings of Interspeech*. <https://arxiv.org/abs/2005.08100>.
- [34] Zhiyuan Guo, Zijian He, and Yiyang Zhang. 2023. Mira: A Program-Behavior-Guided Far Memory System. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. 692–708. <https://doi.org/10.1145/3600006.3613157>.
- [35] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 2023. Graphene: An IR for Optimized Tensor Computations on GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS)*. 302–313. <https://doi.org/10.1145/3582016.3582018>.
- [36] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *arXiv preprint arXiv:1806.03377* (2018). <https://arxiv.org/abs/1806.03377>.
- [37] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. <https://arxiv.org/abs/2401.11181>.
- [38] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Zhifeng Chen, Yanping Hu, Maxim Krikun, Quoc V. Le, and Yonghui Chen. 2019. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems (NeurIPS)*. 103–112. <https://proceedings.neurips.cc/paper/2019/file/093f65e08a295f8076b1c5722a46aa2-Paper.pdf>.
- [39] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuai-wen Leon Song, Samyam Rajbhandari, and Yuxiong He. 2023. DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models. *arXiv preprint arXiv:2309.14509* (2023). <https://arxiv.org/abs/2309.14509>.
- [40] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodahi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Oli Saarikivi. 2022. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 402–416. <https://doi.org/10.1145/3503222.3507778>.
- [41] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the 2nd Conference on Machine Learning and Systems (MLSys)*. 1–13. https://proceedings.mlsys.org/paper_files/paper/2019/file/b422680f3db0986dd7f8f126baaf0fa-Paper.pdf.
- [42] Chenyu Jiang, Zhen Jia, Shuai Zheng, Yida Wang, and Chuan Wu. 2024. DynaPipe: Optimizing Multi-task Training through Dynamic Pipelines. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*. 542–559. <https://doi.org/10.1145/3627703.3629585>.
- [43] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 745–760. <https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng>.
- [44] Tian Jin, Gheorghe-Teodor Bercea, Tung D. Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, and Alexandre E. Eichenberger. 2020. Compiling ONNX Neural Network Models Using MLIR. <https://arxiv.org/abs/2008.08272>.
- [45] Xin Jin, Zhihao Bai, Zhen Zhang, Yibo Zhu, Yinmin Zhong, and Xuanzhe Liu. 2024. DistMind: Efficient Resource Disaggregation for Deep Learning Workloads. *IEEE/ACM Trans. Netw.* 32, 3 (Jan. 2024), 2422–2437. <https://doi.org/10.1109/TNET.2024.3355010>.
- [46] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020). <https://arxiv.org/abs/2001.08361>.
- [47] Sangwoo Kim, Soomin Kim, Daewoon Kim, Junghyun Kim, Woonhyuk Kim, Seung-won Choi, Jongsoo Park, Sungjoo Yoon, and Minjoon Kim. 2020. Torchpipe: On-the-fly pipeline parallelism for training giant models. In *Proceedings of Machine Learning and Systems (MLSys)*. 551–563. <https://proceedings.mlsys.org/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf>.
- [48] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations (ICLR)*. <https://arxiv.org/abs/1412.6980>.
- [49] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing Activation Recomputation in Large Transformer Models. *arXiv preprint arXiv:2205.05198* (2022). <https://arxiv.org/abs/2205.05198>.
- [50] Mario Michael Krell, Matej Kosec, Sergio P. Perez, and Andrew Fitzgibbon. 2022. Efficient Sequence Packing without Cross-contamination: Accelerating Large Language Models without Impacting Performance. *arXiv preprint arXiv:2107.02027* (2022). <https://arxiv.org/abs/2107.02027>.
- [51] Achintya Kundu, Rhui Dih Lee, Laura Wynter, Raghu Kiran Ganti, and Mayank Mishra. 2024. Enhancing Training Efficiency Using Packing with Flash Attention. *arXiv preprint arXiv:2407.09105* (2024). <https://arxiv.org/abs/2407.09105>.
- [52] Joel Lamy-Poirier. 2023. Breadth-First Pipeline Parallelism. In *Proceedings of Machine Learning and Systems, D. Song, M. Carbin, and T. Chen (Eds.), Vol. 5*. Curran, 48–67. https://proceedings.mlsys.org/paper_files/paper/2023/file/24e845415c1486dd582a9d639237f9-Paper-mlsys2023.pdf.

- [53] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308
- [54] Dmitry Lepikhin, Hyukjoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. arXiv:2006.16668 [cs.CL]. <https://arxiv.org/abs/2006.16668>
- [55] Dacheng Li, Rulin Shao, Anze Xie, Eric P. Xing, Xuezhe Ma, Ion Stoica, Joseph E. Gonzalez, and Hao Zhang. 2024. DISTFLASHATTN: Distributed Memory-efficient Attention for Long-context LLMs Training. *arXiv preprint arXiv:2310.03294* (2024). <https://arxiv.org/abs/2310.03294>
- [56] Haoyang Li, Fangcheng Fu, Hao Ge, Sheng Lin, Xuanyu Wang, Jiawen Niu, Xupeng Miao, and Bin Cui. 2025. Hetu v2: A General and Scalable Deep Learning System with Hierarchical and Heterogeneous Single Program Multiple Data Annotations. arXiv:2504.20490 [cs.DC]. <https://arxiv.org/abs/2504.20490>
- [57] Haoyang Li, Fangcheng Fu, Hao Ge, Sheng Lin, Xuanyu Wang, Jiawen Niu, Yujie Wang, Hailin Zhang, Xiaonan Nie, and Bin Cui. 2025. Malleus: Straggler-Resilient Hybrid Parallel Training of Large-scale Models via Malleable Data and Model Parallelization. *Proc. ACM Manag. Data* (2025). doi:10.1145/3725322
- [58] Shuang Li, Zeyi Yao, Haibin Zheng, Dehao Zhang, Xing Song, Yuxiong Yang, and Cho-Jui Hsieh. 2020. Pytorch distributed: Experiences on accelerating data parallel training. In *Proceedings of the VLDB Endowment*. 2236–2248. <https://arxiv.org/abs/2006.15704>
- [59] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*. 1–12. doi:10.1109/HPCA.2012.6168955
- [60] Sheng Lin, Fangcheng Fu, Haoyang Li, Hao Ge, Xuanyu Wang, Jiawen Niu, Yaofeng Tu, and Bin Cui. 2025. LobRA: Multi-Tenant Fine-Tuning over Heterogeneous Data. *Proceedings of the VLDB Endowment* 18, 8 (April 2025), 2616–2625. doi:10.14778/3742728.3742752
- [61] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context. *arXiv preprint arXiv:2310.01889* (2023). <https://arxiv.org/abs/2310.01889>
- [62] Ruyang Liu, Haoran Tang, Haibo Liu, Yixiao Ge, Ying Shan, Chen Li, and Jiankun Yang. 2024. PPLLaVA: Varied Video Sequence Understanding With Prompt Guidance. *arXiv preprint arXiv:2411.02327* (2024). <https://arxiv.org/abs/2411.02327>
- [63] Xinyi Liu, Yujie Wang, Fangcheng Fu, Xupeng Miao, Shenhan Zhu, Xiaonan Nie, and Bin Cui. 2025. NetMoE: Accelerating MoE Training through Dynamic Sample Placement. In *The Thirteenth International Conference on Learning Representations (ICLR 2025)*. <https://openreview.net/forum?id=1qP3lsatCR>
- [64] Ze Liu, Yutong Lin, Yue Cao, et al. 2021. Swin Transformer: Hierarchical Vision Transformer using Shifted Windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. <https://arxiv.org/abs/2103.14030>
- [65] Avinash Maurya, Jie Ye, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. Deep Optimizer Stars: Towards Scalable Training of Transformer Models using Interleaved Offloading. In *Proceedings of the 25th International Middleware Conference (MIDDLEWARE)*. 404–416. <http://dx.doi.org/10.1145/3652892.3700781>
- [66] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2022. Galvatron: Efficient Transformer Training over Multiple GPUs Using Automatic Parallelism. *Proceedings of the VLDB Endowment* 16, 3 (November 2022), 470–479. doi:10.14778/3570690.3570697 <http://dx.doi.org/10.14778/3570690.3570697>
- [67] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. *arXiv preprint arXiv:1710.03740* (2018). <https://arxiv.org/abs/1710.03740>
- [68] Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Cernocky. 2012. Subword language modeling with neural networks. (2012). <https://www.fit.vut.cz/person/imikolov/public/rnnlm/char.pdf>
- [69] Deepak Narayanan, Keshav Santhanam, Aaron Harlap, et al. 2021. Memory-Efficient Pipeline-Parallel DNN Training. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*. <https://arxiv.org/abs/2006.09503>
- [70] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Mostofa Patwary, Rajiv Puri, and Bryan Catanzaro. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. <https://dl.acm.org/doi/10.1145/3458817.3476209>
- [71] NVIDIA. 2023. Deep Learning Performance. <https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html>
- [72] NVIDIA. 2023. GPU Matrix Multiplication. <https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html>
- [73] NVIDIA. 2023. Mixed Precision Training. <https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html>
- [74] NVIDIA. 2023. Sequence Packing Optimization in NeMo. https://docs.nvidia.com/nemo-framework/user-guide/latest/nemotoolkit/features/optimizations/sequence_packing.html
- [75] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative LLM inference using phase splitting. <https://arxiv.org/abs/2311.18677>
- [76] Saurav Pawar, S. M. Towhidul Islam Tonmoy, S. M. Mehedi Zaman, Vinija Jain, Aman Chadha, and Amitava Das. 2024. The What, Why, and How of Context Length Extension Techniques in Large Language Models – A Detailed Survey. <https://arxiv.org/abs/2401.07872>
- [77] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. 2023. Zero Bubble Pipeline Parallelism. *arXiv preprint arXiv:2401.10241* (2023). <https://arxiv.org/abs/2401.10241>
- [78] Colin Raffel, Noam Shazeer, Adam Roberts, et al. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. In *Journal of Machine Learning Research (JMLR)*. <https://arxiv.org/abs/1910.10683>
- [79] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Zhang, and Yuxiong He. 2020. ZeRO: Memory optimization towards training a trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [80] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD 2020)*. ACM, 3505–3506. <https://doi.org/10.1145/3394486.3406703>
- [81] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. <https://arxiv.org/abs/2101.06840>
- [82] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiyang Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 69–87. <https://www.usenix.org/conference/osdi18/presentation/shan>
- [83] Yuzhang Shang, Bingxin Xu, Weitai Kang, Mu Cai, Yuheng Li, Zehao Wen, Zhen Dong, Kurt Keutzer, Yong Jae Lee, and Yan Yan. 2024. Interpolating Video-LLMs: Toward Longer-sequence LMMs in a Training-free Manner. *arXiv preprint arXiv:2409.12963* (2024). <https://arxiv.org/abs/2409.12963>
- [84] David B. Shmoys and Éva Tardos. 1993. An Approximation Algorithm for the Generalized Assignment Problem. *Math. Program.* 62, 1–3 (Feb. 1993), 461–474. <https://doi.org/10.5555/3113606.3113856>
- [85] Mohammad Shoeybi, Mostofa Patwary, Rajiv Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019). <https://arxiv.org/abs/1909.08053>
- [86] TIUAE. [n. d.]. CommonCrawl Dataset. <https://huggingface.co/datasets/tiuuae/falcon-refinedweb>
- [87] Hugo Touvron, Louis Martin, Alex Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Denis Bashlykov, Siddharth Batra, Akhilesh Bhargava, Shruti Bhosale, et al. 2023. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023). <https://arxiv.org/abs/2302.13971>
- [88] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*. 5998–6008. <https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf>
- [89] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Connor Holmes, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. 2023. ZeRO++: Extremely Efficient Collective Communication for Giant Model Training. <https://arxiv.org/abs/2306.10209>
- [90] Shuhe Wang, Guoyin Wang, Yizhong Wang, Jiwei Li, Eduard Hovy, and Chen Guo. 2024. Packing Analysis: Packing Is More Appropriate for Large Models or Datasets in Supervised Fine-tuning. *arXiv preprint arXiv:2410.08081* (2024). <https://arxiv.org/abs/2410.08081>
- [91] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivas Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. 2023. Overlap Communication with Dependent Computation via Decomposition in Large Deep Learning Models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023)*. ACM, 93–106. doi:10.1145/3567955.3567959
- [92] Yujie Wang, Youhe Jiang, Xupeng Miao, Fangcheng Fu, Shenhan Zhu, Xiaonan Nie, Yaofeng Tu, and Bin Cui. 2024. Improving Automatic Parallel Training via Balanced Memory Workload Optimization. *IEEE Transactions on Knowledge and Data Engineering* 36, 8 (August 2024), 3906–3920. doi:10.1109/tkde.2024.3370614 <http://dx.doi.org/10.1109/tkde.2024.3370614>

- [93] Yujie Wang, Shiju Wang, Shenhan Zhu, Fangcheng Fu, Xinyi Liu, Xuefeng Xiao, Huixia Li, Jia Shi Li, Faming Wu, and Bin Cui. 2025. FlexSP: Accelerating Large Language Model Training via Flexible Sequence Parallelism. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2025)*. ACM, 421–436. doi:10.1145/3676641.3715998
- [94] Yujie Wang, Shenhan Zhu, Fangcheng Fu, Xupeng Miao, Jie Zhang, Juan Zhu, Fan Hong, Yong Li, and Bin Cui. 2025. Spindle: Efficient Distributed Training of Multi-Task Large Models via Wavefront Scheduling. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2025)*. ACM, 1139–1155. doi:10.1145/3676641.3715992
- [95] Houming Wu, Ling Chen, and Wenjie Yu. 2024. BitPipe: Bidirectional Interleaved Pipeline Parallelism for Accelerating Large Models Training. <https://arxiv.org/abs/2410.19367>.
- [96] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, Qi Zhang, and Tao Gui. 2025. The rise and potential of large language model based agents: a survey. *Science China Information Sciences* 68, 2 (17 Jan 2025), 121101. doi:10.1007/s11432-024-4222-0
- [97] Yifei Xia, Suhan Ling, Fangcheng Fu, Yujie Wang, Huixia Li, Xuefeng Xiao, and Bin Cui. 2025. Training-free and Adaptive Sparse Attention for Efficient Long Video Generation. arXiv:2502.21079 [cs.CV] <https://arxiv.org/abs/2502.21079>
- [98] Weikai Xu, Chengrui Huang, Shen Gao, and Shuo Shang. 2025. LLM-Based Agents for Tool Learning: A Survey. *Data Science and Engineering* (26 Jun 2025). doi:10.1007/s41019-025-00296-9
- [99] Andrew Chi-Chih Yao. 1980. New Algorithms for Bin Packing. In *Journal of the ACM*, Vol. 27. Association for Computing Machinery, New York, NY, USA, 207–227. doi:10.1145/322186.322187 <https://dl.acm.org/doi/pdf/10.1145/322186.322187>.
- [100] Yang You, Zhao Zhang, Cho-Jui Hsieh, et al. 2020. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. In *8th International Conference on Learning Representations (ICLR 2020)*. <https://openreview.net/forum?id=Syx4wnEtvH>.
- [101] Huangzhao Zhang, Kechi Zhang, Zhuo Li, Jia Li, Yongmin Li, Yunfei Zhao, Yuqi Zhu, Fang Liu, Ge Li, and Zhi Jin. 2024. Deep learning for code generation: a survey. *Science China Information Sciences* 67, 9 (20 Aug 2024), 191101. doi:10.1007/s11432-023-3956-3
- [102] Ruizi Zhang, Tianyu Liu, Will Feng, Andrew Gu, Sanket Purandare, Wanchao Liang, and Francisco Massa. 2024. SimpleFSDP: Simpler Fully Sharded Data Parallel with torch.compile. *arXiv preprint arXiv:2411.00284* (2024). <https://arxiv.org/abs/2411.00284>.
- [103] Pinxue Zhao, Hailin Zhang, Fangcheng Fu, Xiaonan Nie, Qibin Liu, Fang Yang, Yuanbo Peng, Dian Jiao, Shuaipeng Li, Jinbao Xue, Yangyu Tao, and Bin Cui. 2025. MEMO: Fine-grained Tensor Management For Ultra-long Context LLM Training. *Proceedings of the ACM on Management of Data* 3, 1 (Feb. 2025), 1–28. doi:10.1145/3709703
- [104] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Mylène Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (2023), 3848–3860. doi:10.14778/3611540.3611569
- [105] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 559–578. <https://www.usenix.org/conference/osdi22/presentation/zheng>.
- [106] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 193–210. <https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin>.
- [107] Xuanhe Zhou, Zhaoyan Sun, and Guoliang Li. 2024. DB-GPT: Large Language Model Meets Database. *Data Science and Engineering* 9, 1 (01 Mar 2024), 102–111. doi:10.1007/s41019-023-00235-6

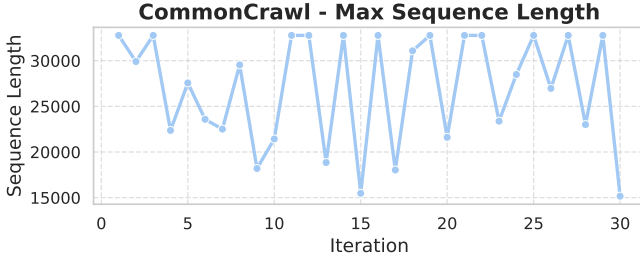


Figure 17: Curve of maximum sequence length across iterations. We sample 1.6M tokens from the CommonCrawl dataset per iteration and track the longest sequence.

A Data Imbalance

In this section, we will provide more evidence to support claims about data imbalance in §3. Firstly, we show that increasing the batch size does not really eliminate the variation in maximum sequence length across iterations. Subsequently, we conduct experiments to examine whether the quadratic time complexity of the attention mechanism impacts end-to-end performance.

A.1 Larger Batch Size

In our experiment, we use 64 GPUs, processing 100K tokens per iteration. Intuitively speaking, as we scale up to 1024 GPUs, the number of tokens per iteration should increase to 1.6M. To examine whether a larger batch size would resolve the inter-iteration imbalance, we further conduct an experiment to sample 30 consecutive iterations on the CommonCrawl dataset, with 1.6M tokens per iteration. Throughout this process, we observe that the maximum sequence length still fluctuates, and falls between 15K and 32K. While this fluctuation is less pronounced compared to the scenario of using 100K tokens per iteration, it still indicates an apparent inter-iteration imbalance.

A.2 End-to-end Influence of Attention

To better demonstrate the impact of the quadratic time complexity of the attention mechanism on end-to-end training, we train the 7B model. Sequences are packed in mini-batches using the first-fit-decreasing bin packing algorithm [74], with a maximum length of 32K. Each mini-batch contains 100K tokens, and we manually ensure that they will form 4 packed sequences (if not, we just skip that iteration for this demonstration experiment). Figure 18 shows the results. Despite carefully controlling tokens and the number of packed sequences, training time still fluctuates by about 20% in both settings. This variation primarily stems from differences in the computational cost of the attention mechanism, while other computational operations and communication overhead remain relatively stable.

B Communication in O-P Disaggregation

B.1 Communication Bound

In this section, we provide formal proof of the communication bound in O-P disaggregation.

We consider N GPUs and a model parameter of size W , and analyze the communication volume when each GPU participates as either a receiver or a sender during the *pull* operation, denoted as V_{recv} and V_{send} , respectively.

For the receiver side analysis, consider a GPU belonging to a pipeline that adopts a parallel scheme $P = \langle TP, PP, CP \rangle$ during the propagation phase. The GPU must receive the complete parameter partition corresponding to its pipeline stage. Specifically:

- If the parameter resides in the GPU’s current pipeline stage, it needs to receive a parameter partition of size $\frac{W}{TP}$.
- If the parameter does not belong to the current stage, the receiving volume becomes 0 as no pull operation is required.

Therefore, we have:

$$V_{\text{recv}} = \frac{W}{TP} \quad \text{or} \quad V_{\text{recv}} = 0$$

For the sender side analysis, our fully-sharded partitioning strategy during the optimization phase ensures each GPU stores exactly one parameter slice of size $\frac{W}{N}$. When M GPUs (where $M \leq N$) request this particular parameter slice during propagation, the sending GPU’s communication volume is given by:

$$V_{\text{send}} = \frac{W}{N} \times M$$

Based on this, we can directly derive the worst-case communication scenario, which occurs when the parallel strategy takes the form $S = \langle 1, 1, C \rangle \times \frac{N}{C}$, where C is any positive integer divisor of N . This configuration represents the case where:

- The system contains no tensor or pipeline parallelism ($TP = 1$, $PP = 1$).
- Every GPU requires all parameter slices ($M = N$).

Consequently, each GPU’s communication volume reaches its theoretical maximum:

$$V_{\text{recv}} = \frac{W}{1} = W \quad \text{and} \quad V_{\text{send}} = \frac{W}{N} \times N = W$$

This worst-case scenario exactly corresponds to training with a homogeneous fully-sharded ZeRO-1 strategy, where the communication cost per GPU is bounded by W and becomes independent of the total GPU count N .

An analogous analysis applies to the *push* operation.

B.2 Computation-Communication Overlap

In this section, we analyze the condition for overlapping communication with computation based on the derived communication bound.

For a matrix multiplication operation under the pipeline with parallel scheme $\langle TP, PP, CP \rangle$, consider input X with shape $[t/CP, h]$ and weight matrix W with shape $[h, h'/TP]$, where t denotes the number of tokens, h and h' represent hidden state dimensions. During the computation of this matrix multiplication, we aim to *pull* the parameter for the next matrix multiplication. As established in Appendix B.1, the maximum amount of data communicated would be $h \times h'$ (i.e., the whole parameter size). Assuming half-precision (e.g., BF16) is used, the total communication volume is:

$$\text{Comm}_{\text{bytes}} = 2hh'$$

And the computational workload (FLOPs) for the matrix multiply (counting multiply-add as 2 operations) is:

$$\text{FLOPs} = 2 \cdot \underbrace{\left(\frac{t}{CP}\right)}_{\text{rows}} \cdot \underbrace{h}_{\text{mid-dim}} \cdot \underbrace{\left(\frac{h'}{TP}\right)}_{\text{cols}} = \frac{2thh'}{CP \cdot TP}$$

Let F denote the GPU's half-precision compute capability (FLOPs) and B the interconnect bandwidth (bytes/sec). The computation time T_{comp} and communication time T_{comm} are:

$$T_{\text{comp}} = \frac{\text{FLOPs}}{F} = \frac{2thh'}{CP \cdot TP \cdot F}$$

$$T_{\text{comm}} = \frac{\text{Comm}_{\text{bytes}}}{B} = \frac{2hh'}{B}$$

To overlap communication with computation, we require:

$$T_{\text{comp}} \geq T_{\text{comm}}$$

Substituting the expressions and simplifying:

$$\frac{2thh'}{CP \cdot TP \cdot F} \geq \frac{2hh'}{B}$$

Canceling common terms ($2hh' \neq 0$):

$$\frac{t}{CP \cdot TP \cdot F} \geq \frac{1}{B}$$

Solving for t :

$$t \geq TP \cdot CP \cdot \frac{F}{B}$$

This indicates that overlapping requires the token count t to exceed a threshold determined solely by the parallel scheme (TP , CP) and hardware ratio (F/B), independent of parameter dimensions h , h' .

As a concrete example, consider an A800 cluster running the 32B model with a pipeline configured with $\langle 16, 1, 1 \rangle$, corresponding to Figure 15a. The hardware specifications are (assuming the worst-case scenario where all communication goes through InfiniBand):

$$F = 19.5 \times 10^{12} \text{ FLOPs}$$

$$B = 200 \times 10^9 \text{ bytes/sec (worst-case)}$$

The token threshold is then:

$$t_{\text{threshold}} = TP \cdot CP \cdot \frac{F}{B} = 16 \times 1 \times \frac{19.5 \times 10^{12}}{200 \times 10^9} = 1560$$

This means that, theoretically, sequence lengths exceeding 1.5K after packing are sufficient to enable overlap (note that both GPU compute power and communication bandwidth may vary in real-world settings, so adjustments will be made based on empirical measurements). In practice, this condition is fully achievable in our setup.

C Cost Model

In this section, we elaborate on the memory model $\text{MaxLen}(P)$, which determines the maximum sequence length, and the latency model $T(l, P)$, which estimates the propagation time for a specific sequence length l , given a parallel scheme P .

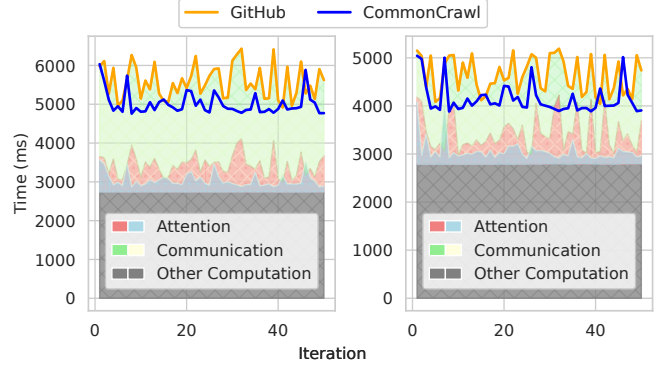


Figure 18: Latency breakdown and fluctuation when training LLaMA 7B on 8 Nvidia A800 GPUs. The left uses pipeline parallel (PP) without data parallel (DP), while the right uses data parallel (DP) without pipeline parallel (PP).

C.1 Memory Model

Consider a model with hidden size H , number of layers $\overline{\text{layers}}$, and vocab size V , and there are N available GPUs. During the propagation phase, we utilize the parallel scheme $\langle TP, PP, CP \rangle$, while the optimization phase employs a fully sharded strategy. For a sequence length l (i.e., micro-batch shape $[1, l]$), memory usage is calculated as follows. (a) For activations, the memory cost per micro-batch is $\frac{\overline{\text{layers}}}{PP} \times \frac{l \times H}{TP \times CP} \times A$, where A is a constant related to the model architecture, determined through profiling. The cumulative memory peak of all micro-batches (accumulated PP times in the first pipeline) is $\overline{\text{layers}} \times \frac{l \times H}{TP \times CP} \times A$. (b) For model states, given that some are fully sharded while others are related to $\langle TP, PP, CP \rangle$, the total memory usage of all layers is $\frac{\overline{\text{layers}} \times H^2}{N} \times B \times \alpha + \frac{\overline{\text{layers}}}{PP} \times \frac{H^2}{TP} \times B \times (1 - \alpha)$. Here, B is a constant related to the model structure, obtainable via profiling, and α is associated with the distribution and precision of model states. For instance, if the optimization phase uses 32-bit optimizer states and parameters, while the propagation phase uses 16-bit parameters and gradients, then $\alpha = \frac{32 \times 3}{32 \times 3 + 16 \times 2} = \frac{3}{4}$. Similarly, for the embedding table, the memory cost is $\frac{H \times V}{N} \times 16 \times \alpha + \frac{H \times V}{TP} \times 16 \times (1 - \alpha)$. The total memory peak is the sum of these costs, forming a linear function of l . Then, the x-coordinate of this function at the y-coordinate corresponding to GPU memory capacity minus a safety margin provides the maximum sequence length $\text{MaxLen}(P)$ supported by $P = \langle TP, PP, CP \rangle$. It's important to note that we can apply a similar modeling approach when incorporating other forms of parallel methods, such as context parallel and sequence parallel.

C.2 Latency Model

The latency is directly modeled as $T(l, P_k) = a_k \times l^2 + b_k \times l + c_k$, where a_k, b_k, c_k are coefficients related to P_k . The quadratic term reflects the quadratic scaling of attention computation with sequence length, while the linear term accounts for other computations and activation communications. The constant term represents additional overheads such as kernel launch delays. By profiling for P_k and adjusting l , we fit the curve and determine these coefficients.

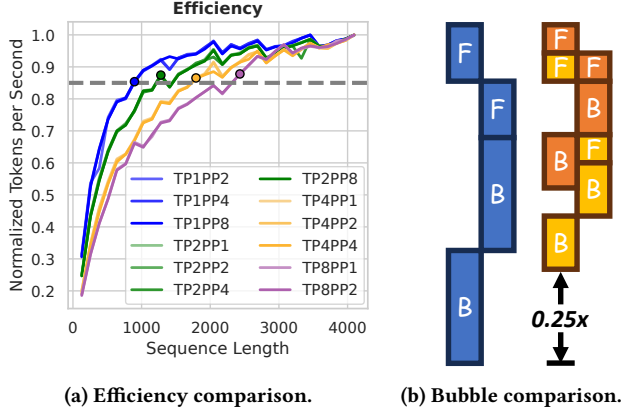


Figure 19: (a) LLaMA 13B training efficiency across different parallel schemes, measured in tokens per second. (b) Latency changes pre- and post-sequence packing for two equal-length sequences when $PP = 2$, “F” and “B” indicate forward and backward propagation, respectively.

D Enumeration Pruning

In this section, we will further discuss two trade-offs during packing as a supplement. Based on this, we can perform pruning optimization to avoid some unnecessary enumeration of the number of micro-batches.

- (1) **Less packing, smaller micro-batch size, lower GPU utilization:** As shown in Figure 19a, we assess the GPU’s efficiency in processing sequences of specific lengths by dividing the propagation latency by the number of tokens in the sequence. Our observations indicate that efficiency approaches optimal only when the sequence length exceeds a certain threshold, which is related to intra-op parallel (e.g., TP) but independent of inter-op parallel (e.g., PP). This is due to GEMMs (General Matrix Multiplications) [72]: intra-op parallel reduces the computational load of single matrix multiplication, while inter-op parallel only reduces the number of matrix multiplications. When sequences are short or the intra-op parallel degree is large, the scale of a single matrix multiplication becomes small, leading to under-utilization of the GPU’s SMs and significant kernel launch overhead [71]. Therefore, insufficient packing on some parallel schemes may result in many under-utilized micro-batches, reducing the performance.
- (2) **More packing, smaller number of micro-batches, higher pipeline bubble rate:** As shown in Figure 19b, packing inevitably reduces the number of micro-batches, increasing the bubble rate. In the example shown, there is about a 0.25 $\times$ performance slowdown. Therefore, packing sequences to the memory limit or the mini-batch maximum is not necessarily optimal.

Based on these two observations, we propose that the number of micro-batches should be neither too large nor too small. Otherwise, it could lead to low GPU utilization or a high bubble rate, respectively. Therefore, we first profile each parallel scheme and introduce the value $\text{UtilLen}(P)$, representing the fully utilized sequence length of P at which its efficiency curve reaches a certain threshold (0.85 in our experiments). This provides an upper bound

Algorithm 1: Greedy-based approximate solver for sequence dispatching across pipelines.

Input: Sequence lengths $\{l_1, \dots, l_B\}$. Maximum strategy indices $\{J_1, \dots, J_B\}$. Parallel schemes of the pipelines $\{P_1, \dots, P_D\}$. Number of trials T ;

Output: Optimal cost O_{best} . Dispatch matrix M_{best} ;

```

1 Initialize  $O_{\text{best}} \leftarrow \infty, M_{\text{best}} \leftarrow \emptyset$ ;
2 for  $\text{trial} = 1$  to  $T$  do in parallel
3   Randomize permutation  $\pi$  to shuffle sequences;
4   Initialize  $C_j, E_j, m_{i,j} \leftarrow 0, \forall i \in [1, B], j \in [1, D]$ ;
5   for  $k = 1$  to  $B$  do
6      $i \leftarrow \pi_k, O_{\min} \leftarrow \infty, j^* \leftarrow -1$ ;
7     for  $j = 1$  to  $J_i$  do
8        $l_{\max} \leftarrow \max(l_j, \max_i \{m_{i,j} \times l_i\})$ ;
9        $C'_j \leftarrow C_j + \tau(l_i, P_j)$ ;
10       $E'_j \leftarrow \tau(l_{\max}, P_j) \times (\text{PP}(P_j) - 1)$ ;
11       $O_{\max} \leftarrow \max(C'_j + E'_j, C_k + E_k \text{ for } k \neq j)$ ;
12      if  $O_{\max} < O_{\min}$  then
13         $O_{\min} \leftarrow O_{\max}, j^* \leftarrow j$ ;
14       $m_{i,j^*} \leftarrow 1, C_{j^*} \leftarrow C'_{j^*}, E_{j^*} \leftarrow E'_{j^*}$ ;
15    $O_{\text{trial}} \leftarrow \max_j \{C_j + E_j\}$ ;
16   if  $O_{\text{trial}} < O_{\text{best}}$  then
17      $O_{\text{best}} \leftarrow O_{\text{trial}}, M_{\text{best}} \leftarrow \{m_{i,j}\}_{B \times D}$ ;
18 return  $O_{\text{best}}, M_{\text{best}}$ ;
```

for the number of micro-batches. Additionally, the maximum sequence length supported by the pipeline configured with P (i.e., $\text{MaxLen}(P)$) offers a good lower bound. Denote the sequences dispatched to the pipeline with parallel scheme P as $l_1, \dots, l_U$ and V as the number of micro-batches, then the enumeration of V can be pruned from the original $V \in [1, U]$ to:

$$V \in \left[\max \left(\frac{\sum_{i=1}^U l_i}{\text{MaxLen}(P)}, 1 \right), \min \left(\frac{\sum_{i=1}^U l_i}{\text{UtilLen}(P)}, U \right) \right]$$

E Details of Sequence Dispatching across Pipelines

In this section, we begin by deriving Eq. (2). We then describe the approximation algorithm for solving the sequence dispatching problem across pipelines (i.e., Eq. (3)) and compare it to the precise solution.

E.1 Derivation of the Lower Bound

Consider Eq. (1) for the d -th pipeline. Let the optimal solution be $o_{i,j}^*$, then the optimal objective can be split into two terms:

$$\begin{aligned}
A &= \max_{1 \leq j \leq V} \left(\sum_{1 \leq i \leq U} o_{i,j}^* \times \tau(l_i, P_d) \right) \times (\text{PP}(P_d) - 1) \\
B &= \max_{1 \leq j \leq V} \left(\sum_{1 \leq i \leq U} o_{i,j}^* \times \tau(l_i, P_d) \right) \times V
\end{aligned}$$

For A, consider the longest dispatched sequence length, denoted as $l_{i\star} = \max_{1 \leq i \leq B} \{m_{i,d} \times l_i\}$. Since $\sum_{1 \leq j \leq V} o_{i\star,j}^* = 1$ and $o_{\star,j}^* \in \{0, 1\}$, there must exist a $j_\star$ such that $o_{i\star,j_\star}^* = 1$. Thus, we have:

$$\begin{aligned} A &\geq \sum_{1 \leq i \leq U} o_{i,j_\star}^* \times \tau(l_i, P_d) \times (\text{PP}(P_d) - 1) \\ &\geq o_{i\star,j_\star}^* \times \tau(l_{i\star}, P_d) \times (\text{PP}(P_d) - 1) \\ &\geq \tau\left(\max_{1 \leq i \leq B} \{m_{i,d} \times l_i\}, P_d\right) \times (\text{PP}(P_d) - 1) \end{aligned}$$

For B, it is evident that:

$$\begin{aligned} \sum_{1 \leq i \leq U} o_{i,j}^* \times \tau(l_i, P_d) &\leq \max_{1 \leq j \leq V} \left(\sum_{1 \leq i \leq U} o_{i,j}^* \times \tau(l_i, P_d) \right), \forall j \in [1, V] \\ \Rightarrow B &= \max_{1 \leq j \leq V} \left(\sum_{1 \leq i \leq U} o_{i,j}^* \times \tau(l_i, P_d) \right) \times V \\ &\geq \sum_{1 \leq j \leq V} \left(\sum_{1 \leq i \leq U} o_{i,j}^* \times \tau(l_i, P_d) \right) \\ &= \sum_{1 \leq i \leq B} m_{i,d} \times \tau(l_i, P_d) \end{aligned}$$

By summing these two terms, the dispatching lower bound for the d -th pipeline is obtained:

$$\sum_{i=1}^B m_{i,d} \times \tau(l_i, P_d) + \tau\left(\max_{1 \leq i \leq B} \{m_{i,d} \times l_i\}, P_d\right) \times (\text{PP}(P_d) - 1)$$

E.2 Approximate Solver

Our developed approximation algorithm is presented in Alg. 1. We start by randomly shuffling the sequences and then dispatch them one by one (lines 1-6). For each dispatch, we aim to prevent the current slowest pipeline P_j from worsening, by considering the accumulated sequence cost C_j and extra cost E_j (lines 7-14). Specifically, for each potential pipeline j of the current sequence i (line 7), we simulate dispatching the sequence to observe the cost change. We first compute the maximum sequence length based on the current sequence and those already dispatched to the pipeline (line 8). Then, we calculate the new accumulated cost C'_j by adding the latency of the current sequence i on pipeline j (line 9). After determining the new extra cost E'_j , which is associated with the maximum sequence length and the pipeline parallel (PP) degree (line 10), we evaluate the maximum total cost $O_{\max}$ across all pipelines (line 11). Afterward, we select the pipeline j^* that minimizes $O_{\max}$ and update the dispatch matrix and costs accordingly (lines 12-14). This process (i.e., a trial) is repeated T times to select the best result (lines 15-17).⁸ Since multiple trials can be parallelized using multiple processes, the actual execution speed is relatively fast (details in Appendix F), and the gap compared to the optimal solution of Eq. (3) is relatively small (details in Appendix E.3).

E.3 Performance Comparison

We analyze the gap between our proposed dispatching algorithm, Alg. 1 (referred to as *greedy dispatching*), and the optimal solution to the original optimization problem, Eq. (3) (referred to as *ideal dispatching*). As illustrated in Figure 20, we compare the ratios of estimated propagation latency (i.e., the optimal objective given by

⁸ T is set to 100 in all our experiments.

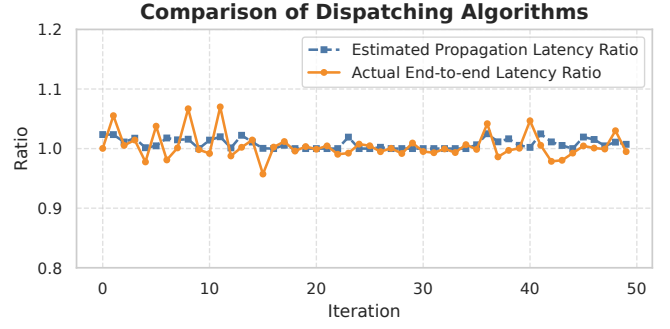


Figure 20: Comparison of estimated propagation latency and actual end-to-end latency between *greedy dispatching* and *ideal dispatching*. The experiments use a 32B model on the CommonCrawl dataset, with 64 GPUs. Each iteration handles 200,000 tokens with a maximum sequence length of 32K.

the algorithm) and actual end-to-end latency by dividing the *greedy dispatching* results by the *ideal dispatching* results. The estimations closely align, and the actual end-to-end latency ratios fall between 0.9 and 1.1. This indicates that there is no significant performance difference between the two approaches. Considering the subsequent time analysis in Appendix F, we ultimately opt for *greedy dispatching* due to its faster solution speed and superior scalability.

F Time Cost and Scalability

In this section, we discuss the computation times of all mathematical models and their scalability. The experiments are conducted on the CommonCrawl dataset using a 32B model configuration with 64 GPUs, a maximum sequence length of 32K, and 200K tokens per iteration.

Table 2 presents the solving time for various components of our mathematical model. To enhance computational efficiency, we employ multi-processing for all components except the *strategy proposal*. The *strategy proposal* process relies on a dynamic programming algorithm, which is inherently sequential and, therefore, not suitable for parallelization. For the parallelized components, we utilize 10 CPU processes, concurrently solving the corresponding problem for **10 iterations**, and record the resulting latency. To further study the scalability of our planning algorithms, apart from running real end-to-end training on the existing testbed, we also simulate a scaled-up scenario with 1024 GPUs, where the number of tokens per iteration is proportionally increased to 3.2 million.

For the *strategy proposal*, its computational complexity scales linearly with the number of GPUs in the cluster. Consequently, the time cost increases significantly from 64 GPUs to 1024 GPUs. However, since this component only executes once, its time cost—remaining under 2 minutes—is acceptable.

For the *ideal dispatching* (i.e., Eq. (3)), while the time cost is acceptable at 64 GPUs, scaling to 1024 GPUs results in superlinear growth. This is due to the complexity being proportional to both the number of pipelines and the number of sequences per iteration—both of which grow dramatically at larger scales. In contrast, *greedy*

Table 2: Solving time for each subproblem. The time cost for *ideal dispatching*, *greedy dispatching*, and *packing* is listed for 10 iterations combined. Hydraulis ultimately chooses the *greedy dispatching* scheme over the *ideal dispatching* scheme due to its faster speed.

	Strategy Proposal	Ideal Dispatching	Greedy Dispatching	Packing
64 GPUs & 200K tokens per iteration	2.03s	40.21s	5.46s	10.15s
1024 GPUs & 3.2M tokens per iteration	107.21s	>1000s	38.56s	13.46s

Table 3: Optimal training configurations. DP refers to the data parallel degree, TP to the tensor parallel degree, PP to the pipeline parallel degree, CP to the context parallel degree, SP to the Ulysses sequence parallel degree, and AC indicates the use of activation checkpointing. For HotSPa, there are 2-3 distinct strategies.

LLaMA 7B	8 GPUs		16 GPUs	
	F1	F2 & F3	F1	F2 & F3
DeepSpeed	DP2SP4	SP8	DP4SP4	DP2SP8
Megatron-LM	DP2TP2PP2	TP4PP2	DP4TP2PP2	DP2TP4PP2
HotSPa	DP2TP2PP2, DP8	TP4PP2, DP4TP2PP2, DP2TP4PP2	DP4TP2PP2, DP16	DP2TP4PP2, DP4TP2PP2, DP16
LLaMA 13B	16 GPUs		32 GPUs	
	F1	F2 & F3	F1	F2 & F3
DeepSpeed	DP4SP4	DP2SP8	DP8SP4	DP4SP8
Megatron-LM	DP4TP4	DP2TP8	DP8TP4	DP4TP8
HotSPa	DP4TP4, DP8TP2	DP2TP8, DP4TP4, DP8TP2	DP8TP4, DP16TP2	DP4TP8, DP8TP4, DP16TP2
LLaMA 32B	32 GPUs		64 GPUs	
	F1	F2 & F3	F1	F2 & F3
DeepSpeed	DP8SP4 + AC	DP4SP8 + AC	DP16SP4 + AC	DP8SP8 + AC
Megatron-LM	DP2TP8PP2	DP2TP8CP2	DP4TP8PP2	DP4TP8CP2
HotSPa	DP2TP8PP2, DP4TP4PP2	DP2TP16, DP2TP8PP2, DP4TP4PP2	DP4TP8PP2, DP8TP4PP2	DP4TP16, DP4TP8PP2, DP8TP4PP2

Table 4: HotSPa’s switching overhead and iteration time comparison.

Configuration	Tokens per Iteration	HotSPa Switching Time	HotSPa Iteration Time (incl. Switching)	Hydraulis Iteration Time
(a) 7B, 32 GPUs	100	1.54s	5.86s	2.21s
(b) 32B, 32 GPUs	100	2.17s	12.21s	7.21s
(c) 32B, 64 GPUs	200	2.21s	13.19s	7.22s

dispatching (i.e., Alg. 1) offers a significantly faster alternative, effectively controlling the growth in time cost. This makes it a more practical choice for large-scale deployments.

For the *packing* (i.e., Eq. (1)) part, we measure the time required to solve all the ILPs. Each ILP addresses the allocation of sequences within a single pipeline, and its scale remains independent of both the number of GPUs and the total number of sequences per iteration. Moreover, since these ILPs are completely independent of each other, they can be solved in parallel, resulting in a relatively stable solving time across different scales.

Overall, considering the training latency of Hydraulis in our experiments, the time cost for a single iteration is approximately 5.5 seconds, leading to a total of around 55 seconds for 10 iterations. Regardless of whether the cluster consists of 64 GPUs or 1024 GPUs, the computation time of our algorithms can be effectively overlapped with GPU execution. In the implementation of Hydraulis, we employ a dedicated process for training and adopt a producer-consumer model to coordinate planning and training. Specifically, the planning processes, acting as producers, solve the sequence dispatching and packing problems, generating execution

plans. Meanwhile, the training process, acting as the consumer, retrieves and executes the plans for the current iteration. This design ensures that GPUs remain fully saturated with training workloads throughout execution, thereby achieving efficient utilization of computational resources.

G Optimal Training Configurations

Table 3 illustrates the manually tuned configurations that achieve the best training performance under different scenarios in §8.2. In terms of HotSPa:

- For scenario F1, with a maximum sequence length of 16K, two strategies are employed: (a) The first strategy handles sequences of 4K to 16K. (b) The second strategy handles sequences less than 4K.
- For scenarios F2 and F3, with a maximum sequence length of 32K, three strategies are employed: (a) The first strategy handles sequences of 16K to 32K. (b) The second strategy handles sequences of 4K to 16K. (c) The third strategy handles sequences less than 4K.

H HotSPa’s Switching Overhead

In this section, we analyze the switching overhead of HotSPa under three configurations, as detailed in Table 4. From configuration (a) to (b), we scale the model size, and from (b) to (c), we scale the total training setup, proportionally to LLaMA’s configuration (16M tokens per iteration for 8192 GPUs).

First, when the training workload is relatively small, HotSPa’s switching overhead becomes significant. For example, in configuration (a), 1.54s out of the total 5.86s per iteration (26%) is spent on strategy switching, negatively impacting end-to-end performance.

Second, when scaling from configuration (b) to (c), doubling the data parallel (DP) degrees causes HotSPa’s switching to occur independently within each 32-GPU group. This keeps the switching overhead relatively stable. However, since the tokens per iteration also scale, the overhead remains consistently around 17% of the total iteration time. (Note that our experimental environment is equipped with 400 GB/s NVLink and 200 GB/s IB, which are sufficiently fast.) In contrast, Hydraulis overlaps strategy switching with computation via O-P disaggregation (as evaluated in §8, Table 1), making it a more scalable solution.

Finally, even if the switching overhead were excluded, HotSPa would still lag behind Hydraulis. This is due to the advantages of Hydraulis in terms of strategy optimization and sequence assignment: For one thing, HotSPa only supports homogeneous strategies at any given time, whereas Hydraulis enables dynamic heterogeneous strategies. For another, Hydraulis incorporates fine-grained sequence dispatching and packing, which HotSPa lacks.