Data-Centric Elastic Pipeline Parallelism for Efficient Long-Context LLM Training

Shiju Wang
Tsinghua University
wangsj25@mails.tsinghua.edu.cn

Yujie Wang
Peking University
alfredwang@pku.edu.cn

Ao Sun
Beijing University of Posts and Telecommunications
maydomine@bupt.edu.cn

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

Zijian Zhu
Tsinghua University
zhuzj23@mails.tsinghua.edu.cn

Bin Cui
Peking University
bin.cui@pku.edu.cn

Xu Han
NLP Group, DCST, IAI, BNRIST
Tsinghua University
han-xu@tsinghua.edu.cn

Kaisheng Ma
Tsinghua University
kaisheng@tsinghua.edu.cn

Abstract

Long context training is crucial for LLM's context extension. Existing schemes, such as sequence parallelism, incur substantial communication overhead. Pipeline parallelism (PP) reduces this cost, but its effectiveness hinges on partitioning granularity. Batch-level PP dividing input samples exhibits high memory consumption in long-context scenario, whereas token-level PP splitting sequences into slices alleviates memory overhead but may incur hardware under-utilization. This trade-off motivates adaptively selecting PP granularity to match resource and workload characteristics. Moreover, sequence length distribution of the real-world dataset exhibits skewness, posing a challenge on PP's workload balance and efficient scheduling. Current static PP scheduling methods overlook the variance of sequence length, leading to suboptimal performance. In this paper, we propose *Elastic Pipeline Parallelism* (EPP) that orchestrates token-level PP and batch-level PP to adapt to resource and workload heterogeneity. We build InfiniPipe, a distributed training system that unleashes the potential of EPP via (1) a resource-aware and workload-balanced sequence processor that splits long sequences and packs short ones; and (2) a co-optimization methodology that jointly optimizes pipeline schedule and gradient checkpointing via a mechanism named *stage-aware chunk-level adaptive checkpointing*. Comprehensive experiments demonstrate that InfiniPipe achieves a 1.69× speedup over state-of-the-art systems.

1 Introduction

In recent years, large language models (LLMs) [3, 4, 8, 12, 29, 41] achieve great success and have profoundly revolutionized many fields. State-of-the-art LLMs progressively support longer contexts, attracting growing attention for efficient long-context training of LLMs.

The massive parameters of LLM necessitate distributed training on a cluster containing multiple nodes, with each equipped with several accelerators (e.g., GPU or NPU). To this end, multiple parallel training strategies such as Data Parallelism (DP) [27, 37], Tensor Parallelism (TP) [21, 34], Sequence Parallelism (SP) [7, 18, 24, 25, 30] and Pipeline Parallelism (PP) [2, 14, 26, 32] have been proposed. Among these strategies, SP is widely adopted for long-context LLM training, which partitions long sequences across devices and introduces communication to perform self-attention operations. However, modern clusters exhibit *bandwidth heterogeneity*: intra-node bandwidth is substantially higher than that of inter-node. As a result, SP is often bottlenecked by inefficient inter-node communication (see Appendix A). In contrast, PP introduces negligible communication overhead compared to SP, utilizing which the costly inter-node communication overhead can be reduced significantly.

However, the effectiveness of PP hinges on its granularity[1]. Batch-level PP, such as DAPPLE [14], packs multiple samples together into a micro-batch but encounters a pronounced memory imbalance across stages when dealing with long sequences, leading to an OOM error as shown in Fig. 1(a). This issue stems from the enlarged micro-batch's granularity, i.e., the number of tokens it contains. Token-level PP (TPP), such as Seq1F1B[38], splits a long sequence into several slices and adopts a fine-grained micro-batch, effectively mitigating this issue. Nevertheless, a hardware under-utilization problem could emerge with an improper chunking strategy due to the lowered computational intensity (Appendix B). Consequently, *it's crucial to determine a proper granularity of PP*, requiring awareness of not only memory constraints but also computational efficiency.

[1]For batch-level PP, we mean the granularity as the number of samples in a micro-batch, i.e., batch size. For token-level PP, it denotes the number of slices to split a sequence into, i.e., $N_{prefill}$ in Fig. 1(a).

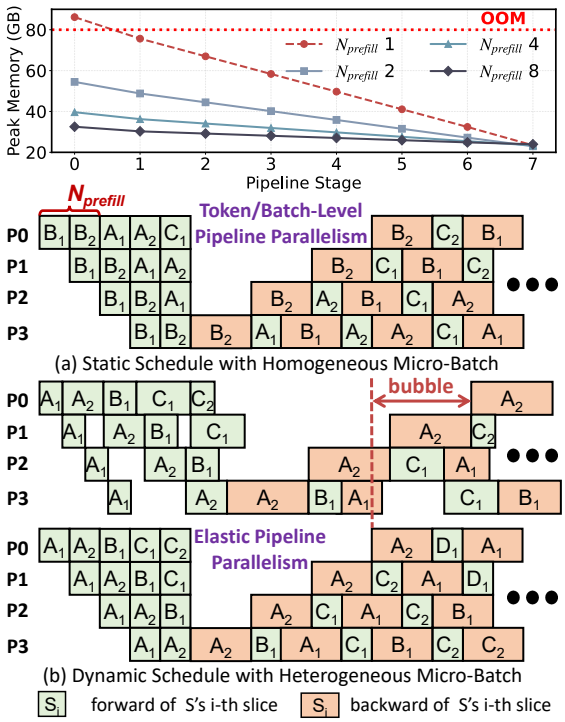

Figure 1. (a) The bottom illustrates DAPPLE ($N_{prefill}$ 1) and Seq1F1B's schedules, where sequences are divided uniformly into $N_{prefill}$ slices, forming homogeneous micro-batches. The upper presents the profiled memory footprint to train GPT-7B on 8 A800 GPUs with a 16K context. Statistics are simulated for DAPPLE due to the OOM error. (b) Heterogeneous micro-batches with B packed from short sequences and the others split from long sequences, requiring a dynamic pipeline schedule. The imbalanced workload introduces pipeline bubbles.

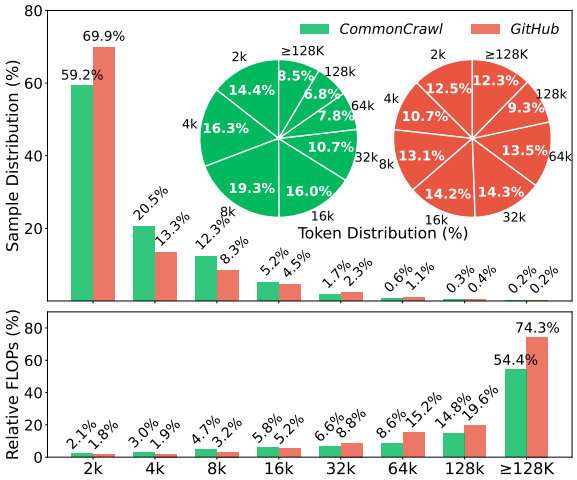

Figure 2. Statistics of sequences grouped by length intervals. The upper subgraph presents the sample and token distribution, while the bottom one denotes the computation FLOPS distribution.

Besides, sequence length distribution of the real-world dataset exhibits skewness, as shown in Fig. 2. Take *GitHub* for example, 91.5% of the sequences have no more than 8K tokens, with only 0.6% of the sequences whose lengths exceed 64K or more. However, these 0.6% sequences contribute to 21.6% of the total tokens and a substantial amount of computation FLOPs. Moreover, recently released LLMs [13, 23, 44] adopt a mixture of long and short sequences for context extension. Llama3 [13] indicates that mixing 0.1% of long-context data with short-context data optimizes performance across both short-context and long-context benchmarks. *The workload heterogeneity reveals an optimization opportunity for workload-aware dynamic pipeline schedule.* Previous works [15, 19, 46] only study dynamic pipeline schedules for batch-level PP, restricting the applicability in long-context training scenarios with limited resources.

In this paper, we propose *Elastic Pipeline Parallelism* (EPP) that demonstrates two key characteristics: *adaptability* and *hybridization*. As for adaptability, it adaptively determines the granularity of PP based on workload and hardware resource (e.g., batch-level PP when resource is sufficient, else token-level PP). For hybridization, it's able to orchestrate batch-level PP and token-level PP, employing a hybrid granularity of PP for varied-length sequences. Specifically, it *splits long sequences* to reduce memory footprint and *packs short ones* to ensure hardware utilization, resulting in *heterogeneous micro-batches* as illustrated in Fig. 5. Moreover, different from current batch-level or token-level PP that assumes *homogeneous micro-batches* and employs a static pipeline schedule, the heterogeneous micro-batches of EPP necessitate *guarantee of workload balance* and a *workload-aware dynamic pipeline schedule*, as illustrated in Fig. 1.

However, fully unleashing the potential of EPP encounters challenges: 1) Precise estimation for memory footprint and time cost is necessary to depict the behavior of EPP. Current cost models [15, 38, 42] do not consider hybrid PP with varied-length input. 2) The workload balance among the *heterogeneous micro-batches* must be ensured and EPP's *adaptability* characteristic requires a *workload-and-resource-aware* methodology to determine the micro-batch's granularity. Previous works [15, 43] consider only workload balance of batch-level PP's *homogeneous micro-batches*. 3) To our best knowledge, we firstly study the dynamic pipeline schedule for hybrid granularity PP, of which the vast optimization space complicates the solving of the optimal pipeline schedule. 4) The existing gradient checkpointing mechanism is not efficient for EPP. Naively disabling or applying full checkpointing degrades computation efficacy. Approaches [5, 17, 20, 39] employing a uniform configuration assume a homogeneous sequence length and ignore workload heterogeneity.

To tackle these challenges, we introduce and develop InfiniPipe, a novel distributed training system that adopts several key techniques to efficiently implement EPP:

- *An Effective Cost Model (§ 3.1).* We establish an effective cost model that: 1) precisely estimates the computation and communication overhead of the *heterogeneous micro-batch*. 2) provides a stage-aware memory footprint analysis for EPP. 3) considers the impact of gradient checkpointing on overhead estimation.
- *A Workload-Balanced and Resource-Aware Sequence Processor (§ 3.2).* We devise a sequence chunking algorithm that generates workload-balanced heterogeneous micro-batches of proper granularity via a two-phase process: *splitting long sequences* and *packing short ones*. The cost model plays a role in workload overhead and memory footprint estimation, ensuring the algorithm's efficiency and resource awareness, respectively.
- *A Chunk Scheduler that Co-Optimizes Pipeline Schedule with Gradient Checkpointing (§ 3.3).* We develop a co-optimization methodology to tackle the challenges of pipeline schedule and gradient checkpointing, of which the solving space is first confined based on careful discussion of existing approaches' trade-offs and analysis of EPP's observations. Notably, we propose a new checkpointing mechanism called *stage-aware chunk-level adaptive checkpointing* to optimally integrate checkpointing with EPP. Afterward, we solve the optimization problem via dynamic programming and mixed-integer linear programming (MILP), where multiple acceleration techniques are utilized to overlap the solving overhead.

Extensive experiments conducted on various workloads demonstrate that InfiniPipe achieves a speedup of up to 1.69× compared to existing SOTA work. The key contributions of this work can be summarized as follows:

- We identify the limitations of existing long-context training approaches and propose *Elastic Pipeline Parallelism* as a solution.
- We firstly co-optimize the pipeline schedule with checkpointing and propose a new mechanism named *Stage-Aware Chunk-Level Adaptive Checkpointing*.
- We develop InfiniPipe, a brand new distributed LLM training system for varied-length corpora.
- We comprehensively evaluate InfiniPipe to indicate that InfiniPipe has state-of-the-art performance.

2 Preliminaries

2.1 Distributed LLM Training

Distributed training is widely used for accelerating LLM training, and there exist several parallel training strategies.

Data Parallelism. Data Parallelism (DP) assigns training input of sequences to different devices in a DP group, where the model states (parameters, gradients, and optimizer states) are duplicated and gradients need to be reduced to ensure mathematical consistency. To alleviate the overhead of model states, sharded data parallelism (SDP) techniques (e.g., DeepSpeed-ZeRO [36] and PyTorch FSDP [47]) further partition model states across devices. Accordingly, gather and scatter communications are introduced to obtain complete parameters required for LLM's execution and reduce gradients, respectively. The gather and scatter communications in SDP can be overlapped with computation.

Tensor Parallelism. Tensor Parallelism (TP) [34] partitions weight matrices of linear projection along rows or columns in FFN and attention layers. An all-reduce communication is performed to synchronize the results and maintain mathematical consistency. Megatron-SP [21] replaces the all-reduce communication with a pair of reduce-scatter and all-gather communications, reducing activation memory footprint without increasing communication cost.

Sequence Parallelism. Sequence Parallelism (SP) partitions sequences into multiple slices and distributes them among devices in an SP group, introducing communication for self-attention operations. SP is typically categorized into two variants: Ulysses-style SP and Ring-style SP. DeepSpeed-Ulysses [18] proposes Ulysses-Style SP, which first performs three all-to-all communications on query, key, and value to gather the complete sequence and distribute along the head dimension, after which a head parallel attention is performed. An additional all-to-all communication is finally required to redistribute the attention output along the sequence dimension. In contrast, Ring-Style SP [7, 24, 25, 30] exemplified by Context Parallelism (CP) performs self-attention through multiple steps, where the keys and values of different slices are exchanged via p2p communication. A detailed analysis of SP's communication overhead is provided in Appendix A.

Pipeline Parallelism. Pipeline Parallelism (PP) horizontally partitions a model into several parts (stages) that execute sequentially, requiring transmission of activation between two neighboring parts. This transmission introduces negligible communication overhead as it occurs only once. To enhance device occupancy, PP partitions training inputs into micro-batches, which categorizes PP into two variants: 1) *batch-level* PP, like DAPPLE [14] that divides input samples, and 2) *token-level* PP exemplified by Seq1F1B [38] and TeraPipe [28] that further splits a sequence into slices. For the sake of training stability, we focus on synchronous PP with periodic pipeline flushes. Various batch-level pipeline schedules [2, 14, 26, 32, 33] have been proposed. Fig. 1(a) illustrates DAPPLE and Seq1F1B's schedule consisting of three distinct stages: *warmup*, *steady*, and *cooldown*. For the token-level PP's schedule, it's worth noting that *for each slice, the forward pass must be scheduled after its preceding slices, while the backward pass must be scheduled after its subsequent slices.* This is because the query of a token accesses only preceding tokens' keys and values in the forward pass, thus gradients of key and value of a token rely on those

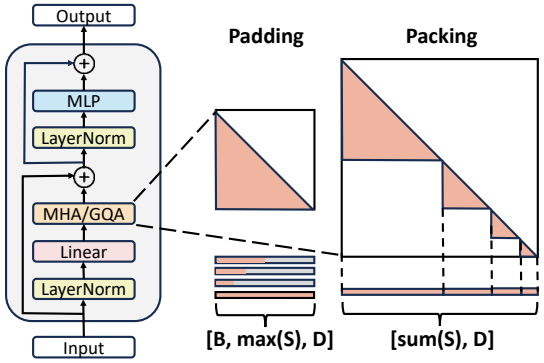

Figure 3. Illustration of sequence packing and padding's difference in attention mask and activation arrangement.

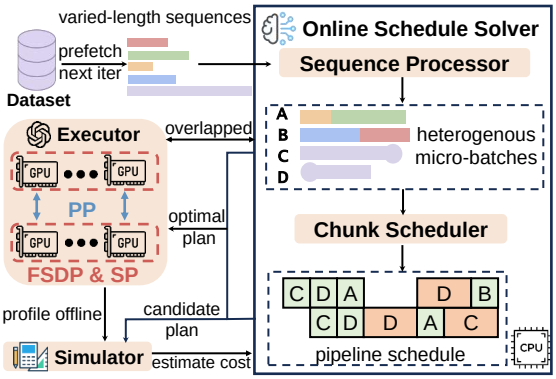

Figure 4. InfiniPipe System Overview

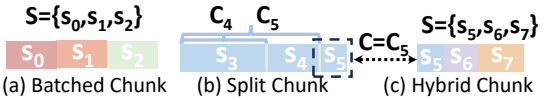

Figure 5. Illustration of *Chunk* (heterogeneous micro-batch in EPP). Slices from the same sequence are colored the same. (b) Two split chunks and a tail slice are generated. (c) The tail slice is packed with short sequences to form a hybrid chunk.

of subsequent tokens in the backward pass. Token-level PP employs a finer-granularity micro-batch of slices, exhibiting a lower memory footprint compared to batch-level PP, as shown in Fig. 1(a). The impact of splitting sequences on computation is discussed in Appendix B.

2.2 Sequence Packing

Techniques such as padding or packing are used to deal with sequences of different lengths. As shown in Fig. 3, padding pads or truncates sequences to the same length, introducing unnecessary computation overhead. In contrast, sequence packing [22], which concatenates multiple input sequences into a single sequence and adjusts attention masks to prevent cross-attention between unrelated tokens from different sequences, emerges as a more modern and efficient alternative.

2.3 Gradient Checkpointing

Gradient checkpointing is a widely adopted technique in LLM training that trades computation for activation memory footprint reduction. Specifically, intermediate activations are freed after the forward pass, but recomputed in the backward pass for gradient computation if checkpointing is applied.

2.4 Gradient Accumulation

To train LLM at a large batch size with limited memory capacity, gradient accumulation is proposed, which updates parameters once using the reduced gradients accumulated from multiple micro-batches and yields the same optimization trajectory.

3 InfiniPipe

Fig. 4 illustrates the architecture of InfiniPipe, which consists of (1) a simulator that provides accurate cost estimation; (2) an online scheduler deduces the optimal EPP schedule plan; and (3) an executor that implements efficient EPP runtime. InfiniPipe operates in two distinct phases: *offline profiling* and *online schedule solving*. In the offline phase, a comprehensive profiling on the executor is performed to refine the simulator's cost model (§ 3.1). In the online phase, the schedule solver fetches a global batch of variable-length sequences, which are first organized by the sequence processor (§ 3.2) into workload-balanced chunks. These chunks are then fed into the chunk scheduler (§3.3), which co-optimizes the EPP pipeline schedule with checkpointing configuration. Finally, the executor carries out the LLM training iteration according to the generated schedule plan. A decoupled architecture is employed, with the executor running on GPUs while the solver operates on CPUs. InfiniPipe integrates 3D parallelism combing Ulysses-Style SP equipped with ZeRO-3 (degree d_s) and PP (degree d_p). Frequently used notations are summarized in Tab. 1.

Table 1. Frequently used notations in this work.

N	The number of cluster's GPUs
e	The element size of data type
D	The hidden dimension of model
L	The number of model's layers
C_k	The context length of the k^{th} chunk
S_k	Lengths of slices in the k^{th} chunk
I_k	Whether the k^{th} chunk is the tail slice of a sequence

3.1 Cost Model

Precise time and memory footprint estimation is crucial to reduce pipeline bubbles and avoid OOM errors. In this section, we present the fundamental cost model employed by InfiniPipe to depict the behavior of EPP.

Previous work FlexSP [42] has proposed an effective cost model for LLMs' distributed training on varied-length corpora. Our cost model extends FlexSP's approach but demonstrates some advantages. Firstly, there are two ways to form an EPP's micro-batch: *batching* and *splitting*. FlexSP's method is only effective for the former, and ours further facilitates the latter and even the combination. Secondly, the impacts of gradient checkpointing on time cost and memory footprint estimation are not considered by FlexSP. Last but not least, FlexSP doesn't consider PP, and we build a stage-aware cost model for EPP. The parameters required by our cost model, including M_{token}, α_1, and β_1 mentioned below, are obtained via offline profiling on the executor and regression fitting.

3.1.1 Definition of *Chunk*.

We firstly introduce *Chunk*, a key concept in InfiniPipe.

As illustrated in Fig. 5, *Chunk* can be categorized into three types: 1) **Batched Chunk**. Short sequences are batched together, resulting in a chunk formulated as S, a set containing multiple slices. 2) **Split Chunk**. A long sequence is split into multiple slices with the last one referred to as *tail slice*. Due to the causal mask of self-attention, a split chunk's computation relies on keys and values of preceding slices. We formulate a split chunk with its length s and context length C. 3) **Hybrid Chunk**. Short sequences can be packed with a *tail slice* s_0 for the sake of workload balance, formulated as a context C for s_0 and a set of slices S (s_0 included). Notably, *packing of two tail slices is avoided* as it forces co-scheduling of two long sequences[2], increasing memory overhead. In summary, all three types of chunks can be expressed using a uniform representation $\{C, S\}$, where C denotes the context length (0 for batched chunk) and S is the set containing the lengths of slices in the chunk.

Estimations on computation and communication overhead as well as memory footprint are then introduced based on the concept of *Chunk*.

3.1.2 Computation and Communication Analysis.

As for computation, we assume a quadratic time complexity with respect to sequence length, consistent with FlexSP. Therefore, the computation time for processing chunk $\{C_k, S_k\}$ during both forward and backward passes is modeled as:

$$\begin{aligned} T_{comp}(C_k, S_k) = \frac{1}{N}(\alpha_1((C_k + s_0)^2 - C_k^2) + \alpha_2 s_0 \\ + \sum_{s\in(S_k-\{s_0\})} (\alpha_1 s^2 + \alpha_2 s)) + \frac{\beta_1}{d_p} \end{aligned} \tag{1}$$

As for communication, V (communication volume), B_{comm} (bandwidth), β_{comm} (latency) and f (frequency) are utilized to model the overhead:

$$T_{comm}(V, f) = (\frac{V}{B_{comm}} + \beta_{comm}) \cdot f, \tag{2}$$

[2]For instance, tail slice A_3 of sequence A is packed with B_2 of sequence B to AB. As a result, A_1, A_2 and B_1 must be scheduled before AB, introducing activation overhead of both A and B.

Specifically, Ulysses-style SP requires four All-to-All communications in each layer, resulting in the following overhead:

$$T_{all2all}(S_k) = (\frac{eD\sum_{s\in S_k} s}{d_s B_{all2all}(d_s)} + \beta_{all2all}(d_s))\frac{4L}{d_p} \tag{3}$$

The total execution time for a chunk comprises both computation and communication components:

$$T_{tot}(C_k, S_k) = T_{comp}(C_k, S_k) + T_{all2all}(S_k), \tag{4}$$

which is applicable in both forward and backward passes.

3.1.3 Stage-Aware Memory Footprint Analysis.

We begin by analyzing the activation memory footprint of a chunk. Consistent with FlexSP, which employs flash-attn [10, 11] and assumes activation overhead proportional to the number of tokens, we further include M_{dkv} representing the overhead of keys and values' gradients in our estimation based on two observations of TPP: 1) *chunks of a sequence execute backward reversely with the last chunk executing backward first.* 2) *gradients for keys and values of all chunks are materialized simultaneously during the backward pass of the last chunk, while freed asynchronously until the completion of its own backward pass.* Consequently, the overall activation memory footprint is modeled as:

$$\begin{aligned} M_{act}(S_k) &= Act(S_k) + M_{dkv}(S_k) \\ &= \frac{M_{token}}{N}\sum_{s\in S_k} s + (1 - I_k)\frac{2eLD}{N}\sum_{s\in S_k} s, \end{aligned} \tag{5}$$

where M_{token} is a model-specific constant representing the activation memory per token.

The total memory footprint for the p^{th} (indexed from 1) pipeline stage comprises the memory allocated for fixed model states $M_{ms}(p)$ and the ever-changing activation memory $M_{act}(p, t)$:

$$M_{tot}(p, t) = M_{ms}(p) + M_{act}(p, t) \tag{6}$$

The peak memory of the 1F1B pipeline occurs during the steady phase, where each pipeline stage p maintains a *constant* number of chunks that have not finished their backward passes, as shown in Fig. 1(a). Let $W_p(t)$, called the *chunks window*, denote the set of these chunks at time t, satisfying:

$$|W_p(t)| = d_p - p + N_{prefill} \tag{7}$$

As activations of all chunks within $W_p(t)$ must be accommodated, the total memory footprint is modeled as:

$$\begin{aligned} M_{tot}(p, t) &= M_{ms}(p) + M_{act}(W_p(t)) \\ &= M_{ms}(p) + \sum_{k\in W_p(t)} M_{act}(S_k) \end{aligned} \tag{8}$$

3.1.4 Combining Gradient Checkpointing Together.

Gradient checkpointing affects estimation for both memory footprint and time cost. Same as common practice in Megatron-LM [34], InfiniPipe applies checkpointing at layer granularity and let l_{ckpt} denote the checkpointed layers.

We first analyze how checkpointing affects memory footprint estimation. There is a key observation when combining

Table 2. Relative standard deviation (RSD) of simulated execution time and chunk's length. "Balance Time" denotes solely balancing time cost while "Balance Time & Length" means our co-optimization approach.

RSD(%)	*Execution Time*	*Chunk Length*
Balance Time	**5.9**	35.2
Balance Time & Length	6.2	**5.5**

TPP with checkpointing: *keys and values of a chunk must be stored regardless of whether checkpointing is applied, as they are accessed by the subsequent chunks, so as their gradients.* However, the other activations of a checkpointed layer can be ignored. The phenomenon drives us to deal with keys and values individually in cost estimation. Specifically, we include not only the input but also keys and values of the checkpointed layers in the checkpointing overhead M_{ckpt}:

$$M_{ckpt}(S_k) = \frac{(3-2I_k)eD \cdot l_{ckpt}}{d_s} \sum_{s \in S_k} s \tag{9}$$

Moreover, checkpointing has no impact on M_{dkv} and a chunk's activation footprint is further reformulated as:

$$M_{act}(S_k) = \frac{L - l_{ckpt} \cdot d_p}{L} Act(S_k) + M_{dkv}(S_k) + M_{ckpt}(S_k), \tag{10}$$

As for time cost estimation, checkpointing only affects the backward pass with a recomputation cost:

$$T_{ckpt}(C_k, S_k) = \frac{l_{ckpt}}{L \cdot d_s} \cdot T_{tot}(C_k, S_k).{}_{fwd} \tag{11}$$

3.2 Sequence Processor

InfiniPipe's sequence processor ingests original varied-length sequences sampled from the dataset and then organizes them into *chunks*. There are two crucial issues to be addressed when designing the processing algorithm: 1) ensuring workload balance across chunks to avoid pipeline bubbles and 2) determining the proper granularity of a chunk that maximizes utilization while adhering to memory limits. In this section, we introduce a resource-aware and workload-balanced chunking algorithm, outlined in Alg. 1.

The algorithm depicts the processor's operation consisting of two steps: splitting long sequences to generate *split chunks* and packing short sequences to form *batched chunks* and *hybrid chunks*. It takes the following inputs: the cost model $\mathcal{M}$; token capacity C, i.e., maximum number of tokens the cluster can accommodate (approximated according to $\mathcal{M}$'s M_{token} and the cluster's memory capacity); sequence lengths S, i,e., the lengths of sequences sampled from dataset; and slice number N, which is automatically tuned and denotes the number of slices to split the longest sequence into.

To begin with, we employ a uniform mesh to shard sequence (except short sequences) into multiple workload-balanced slices and a *tail slice* with less overhead (Line 2). For instance, with a mesh {8K,4K,2K}, sequences with more than 12K tokens will be split into an 8K and 4K slice, followed by a variable-length remainder. The mesh is obtained by splitting the ingested longest sequence into N workload-balanced slices (Line 1), which is based on $\mathcal{M}$'s time cost estimation T_{tot} (Eq. 4) for the backward pass. Time threshold $\mathcal{T}_t$ and token threshold $\mathcal{T}_m$ representing a slice's maximum time cost and token number, respectively, are also derived and will be used in the following packing process.

Afterward, a Best-Fit-Decreasing (BFD) algorithm is utilized to pack short sequences with each other or with a tail slice (Lines 3-14). Slightly different from approaches that solely focus on the balance of time cost, our method further takes the balance of chunk length into consideration to avoid memory fragmentation. Specifically, we prioritize packing the longest short sequence *short_seq* with the bucket b with minimum $\frac{b.time}{b.token}$ (Line 9), where the metric indicates the proportion of long sequences in the bucket. Perturbation on $\mathcal{T}_t$ is performed to co-optimize the balance of time cost and length (Line 14). As shown in Tab. 2, our approach achieves a lower variance of chunk length and the same level of balance of time cost.

The sequence processor demonstrates a time complexity of $O(n^2|S|^2)$, introducing negligible overhead.

Algorithm 1: Workload-Balanced Chunking

Input: Cost model $\mathcal{M}$, token capacity C, sequence lengths S, slice number N.
Output: Chunks $\{(C_k, S_k)|k \le n\}$, n is number of chunks

1 $mesh, \mathcal{T}_t, \mathcal{T}_m \leftarrow \mathcal{M}.split(max(S), N)$;
2 $split\_chunks, tail\_slices, short\_seqs \leftarrow split\_seqs(S, mesh)$;
3 $\mathcal{B} \leftarrow initialize\_buckets(tail\_slices)$; `// a sorted list`
4 $\mathcal{M}.descend\_sort\_by\_time(short\_seqs)$;
5 **while** $\neg short\_seqs.is\_empty()$ **do**
6 $short\_seq, flag \leftarrow short\_seqs.pop(0), False$;
7 **if** $\min_{b \in \mathcal{B}}\{b.tokens\} + short\_seq.tokens > \mathcal{T}_m$ **then**
8 $\mathcal{B}.create\_new\_bucket(short\_seq)$;**continue**;
  `// prioritize bucket with lower` $\frac{bucket.time}{bucket.token}$
9 **for** $bucket \in \mathcal{B}$ **do**
10 **if** $bucket.time + short\_seq.time \le \mathcal{T}_t$ & $bucket.tokens + short\_seq.tokens \le \mathcal{T}_m$ **then**
11 $bucket.combine(short\_seq)$;
12 $flag \leftarrow True$;**break**;
13 **if** $\neg flag$ **then**
14 $\mathcal{T}_t \leftarrow \min_{b \in \mathcal{B}}\{b.time\} + short\_seq.time$; **goto** 9;
15 $batched\_chunks, hybrid\_chunks \leftarrow transform(\mathcal{B})$;
16 **return** $split\_chunks \cup batched\_chunks \cup hybrid\_chunks$

3.3 Chunk Scheduler

In this section, we first define our scheduling space through careful trade-off discussions and then present our two-level

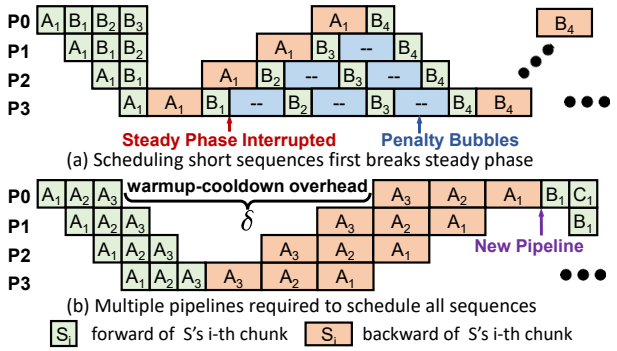

Figure 6. Illustration of pipeline scheduling space. (a) Explanation of sequences' execution order in a 1F1B pipeline. (b) To avoid OOM error, multiple 1F1B pipelines are scheduled, with each introducing an identical warmup-cooldown overhead δ.

approach that jointly optimizes pipeline schedule (via *sequence grouping*, § 3.3.2) and checkpointing configuration (via *stage-aware chunk-level adaptive checkpointing*, § 3.3.3).

3.3.1 Solving Space Definition. It's non-trivial to solve a sophisticated schedule plan due to the vast optimization space: 1) There exist numerous basic scheduling patterns [2, 14, 26, 35] with each exhibiting distinct memory footprint and bubble characteristics. 2) The execution order of varied-length sequences needs to be determined. 3) A checkpointing mechanism tailored for EPP is required to efficiently integrate checkpointing. We begin by introducing some key insights, based on which the solving space is then confined.

Firstly, 1F1B is taken as the basic schedule pattern. GPipe [2] features *forward-then-backward* schedule, which is not memory efficient because it needs to accommodate all micro-batches. Chimera [26] proposes bidirectional PP while it duplicates model states and needs synchronization of gradients. Other 1F1B schedule patterns [28, 32, 35, 38] are also proposed. However, most improvements in these works stem from reducing the bubble of warmup and cooldown phases. As the number of micro-batches increases, these approaches converge to similar performance because the steady phase with negligible bubbles dominates the total execution. To this end, 1F1B schedule [38] is taken as our basic schedule pattern as we focus on training throughput.

Secondly, it's feasible to schedule one or multiple 1F1B pipelines, within which longer sequences are prioritized to execute. To begin with, the longest sequences should be scheduled first within a 1F1B pipeline. As illustrated in Fig. 6(a), sequence A with one chunk is scheduled before sequence B with three chunks. We observe that the steady phase is interrupted due to the unavailability of a backward schedule, introducing severe penalty bubbles. To address this, we prioritize longer sequences in a 1F1B pipeline and enforce this as a fundamental scheduling rule. Moreover, scheduling of multiple 1F1B pipelines is necessary when sequences cannot be scheduled in a single 1F1B pipeline altogether. As shown in Fig. 6(b), sequences B and C cannot be co-scheduled with sequence A due to the limited memory capacity, forcing a new pipeline to be scheduled. Gradient accumulation is enabled among these 1F1B pipelines to ensure consistency. Notably, we do not exploit pipeline bubbles to overlap these 1F1B pipelines, as typically no more than three 1F1B pipelines are scheduled by our solver, offering marginal benefits but complicating the solving process.

Thirdly, employing stage-aware chunk-level adaptive checkpointing. In long-context training, performance degradation may occur without checkpointing because 1) a finer granularity of TPP might be required to alleviate the memory burden, harming hardware utilization, and 2) a number of 1F1B pipelines have to be scheduled for multiple excessively long sequences, leading to prohibitive warmup-cooldown overhead. However, directly applying full checkpointing introduces unnecessary recomputation overhead. Furthermore, EPP's memory footprint disparity and chunk length heterogeneity render suboptimal performance of a uniform and static checkpointing strategy. Specifically, different pipeline stages have varying requirements for checkpointing to reduce memory footprint, and applying checkpointing on longer chunks reduces more activation footprint at the same recomputation cost if chunks are workload-balanced. The trade-offs above inspire us to employ a *stage-aware adaptive checkpointing strategy at chunk granularity.*

In summary, we define our solving space as follows: for a pipeline schedule $\mathcal{P}$, we group sequences and assign sequences of each group to a distinct 1F1B pipeline $P \in \mathcal{P}$; for checkpointing, we apply a customized checkpointing layer $ckpt(p,k)$ for each chunk $\{C_k, S_k\}$ at each pipeline stage p. Let $\mathcal{S}[i]$ represents the set of sequences divided into i chunks, the problem can be formulated as:

$$\begin{aligned} &minimize \sum_{P\in\mathcal{P}} T(P) \\ &\text{s.t.} \bigcup_{P\in\mathcal{P}} \mathcal{S}_P = \mathcal{S}[:], \end{aligned} \tag{12}$$

where $\mathcal{S}_P$ represents the sequences scheduled in P and $T(P)$ denotes the total execution time of P. Thanks to the workload-balanced manner InfiniPipe processes sequences, we ignore the pipeline bubbles in the steady phase and simplify Eq. 12 as a summation of recomputation cost $T_{ckpt}(P)$ and warmup-cooldown overhead δ:

$$minimize \sum_{P\in\mathcal{P}} \delta + T_{ckpt}(P), \tag{13}$$

where δ is a constant approximated as $(d_p - 1) \cdot avg(T_{tot})$. The optimal pipeline schedule $\mathcal{P}_0$ along with checkpointing configuration $ckpt_0(p,k)$, is co-optimized, with detailed methodology provided in § 3.3.2 and § 3.3.3, respectively.

3.3.2 Sequence Grouping. An interference of pipeline schedule and gradient checkpointing is observed in Fig. 7(a). Indicated by Eqs. 7,8, the pipeline's memory footprint is

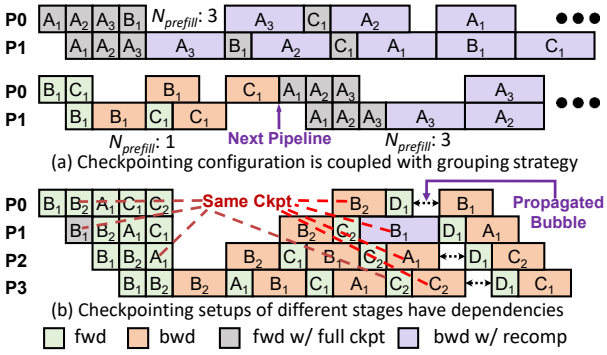

Figure 7. Illustrations of insights about co-optimizing checkpointing with pipeline schedule.

related to $N_{prefill}$, i.e., the number of chunks the longest sequence in a sequence group is split into. Accordingly, when short sequences B and C are grouped with long sequence A, they are forced to apply a tighter checkpointing setup than they are scheduled separately due to the enlarged $N_{prefill}$, introducing more recomputation overhead. Therefore, 1) *sequences of similar lengths should be grouped together* 2) *scheduling more 1F1B pipelines is potential to reduce recomputation cost* $\sum_{P\in\mathcal{P}} T_{ckpt}(P)$ *of Eq. 13 at the cost of severer warmup-cooldown overhead* $\delta \cdot |\mathcal{P}|$, forming the trade-off when optimizing sequence grouping strategy.

We employ a dynamic programming method to resolve the optimal sequence grouping strategy. Let $dp[i]$ represent the minimum cost to schedule sequences with at most i chunks. The state transition equation can be deduced as:

$$dp[i+1] = \min_{0\le k\le i}\{dp[k] + \delta + T_{ckpt}(P)\}, \tag{14}$$

where sequences in $\mathcal{S}[k+1:i+1]$ is scheduled by pipeline P and $T_{ckpt}(P)$ is obtained by applying *stage-aware chunk-level adaptive checkpointing* (§ 3.3.3) on P. By tracking transition states, we derive both the sequence grouping and its corresponding checkpointing configuration.

3.3.3 Stage-Aware Chunk-Level Adaptive Checkpointing. In this section, we elaborate on how we apply optimal checkpointing configuration for a given 1F1B pipeline P assigned n chunks.

To begin with, we analyze the impact of checkpointing on PP and introduce a constraint on $ckpt(p,k)$. As illustrated in Fig. 7(b), full checkpointing is applied to B_1 of the second stage. We observe that checkpointing affects not only the second stage, introducing propagated bubbles of identical size in all the other stages. A key insight is that *applying full checkpointing to the marked chunks* B_2, A_1, *and* C_2 *exploits the propagated bubble and maintains the total execution time unchanged.* Let $ckpt'(p,k)$ represent the number of checkpointed layers for the chunk executing the k^{th} backward pass in the p^{th} pipeline stage (the execution order differs between forward and backward passes). Based on this insight, we yield the following constraint:

$$ckpt'(p,k) = ckpt'(p+i,k+i) = C[k+d_p-p], \tag{15}$$

where C is a set containing $d_p - 1 + n$ independent integer variables. Let $f2b[k]$ map the forward execution order to the backward execution order, we have:

$$ckpt(p,k) = ckpt'(p, f2b[k]) = C[f2b[k]+d_p-p] \tag{16}$$

This formulation reduces the number of optimization variables from $n \cdot d_p$ of $ckpt(p,k)$ to $d_p - 1 + n$ of C, significantly reducing solving overhead.

Afterward, a solution based on MILP is introduced, as outlined in Alg. 2. Fig. 7(b) reveals that recomputation cost $T_{ckpt}(P)$ is related to C with:

$$T_{ckpt}(P) = \hat{F} \cdot \sum_{c\in C} c, \tag{17}$$

where $\hat{F}$ denotes the estimated forward execution time of a model layer. *The optimal checkpointing strategy aims to minimize the recomputation cost* $T_{ckpt}(P)$ *(Eq. 17) with peak memory* $M_{tot}(p,t)$ *(equation 8) not exceeding hardware capacity limit* $\mathcal{G}$, formulated as:

$$\begin{aligned} &\arg\min_{C\in\mathbb{N}^{n+d_p-1}} \hat{F}\cdot\sum_{c\in C} c \\ \text{s.t.}\quad & M_{ms}(p) + \sum_{k\in W_p(t)} M_{act}(S_k) \le \mathcal{G} \quad \forall(p,t), p\le d_p \\ & c \le \frac{L}{d_p}, \forall c\in C \end{aligned} \tag{18}$$

Combining Eqs. 9,10,16, we derive a linearity:

$$M_{act}(S_k) = \mathcal{I}[k] - \mathcal{F}[k]\cdot C_{p,k}, \tag{19}$$

where chunk-specific coefficients $(\mathcal{I}[k], \mathcal{F}[k])$ and $C_{p,k}$ are defined explicitly in Alg. 2. To this end, constraint 18 is further reformulated as a system of linear inequalities in terms of C and the MILP is finally expressed as:

$$\begin{aligned} &\arg\min_{C\in\mathbb{N}^{n+d_p-1}} \sum_{c\in C} c \\ \text{s.t.}\quad & \sum_{k\in W_p(t)} \mathcal{I}[k] - \mathcal{F}[k]\cdot C_{p,k} \le \mathcal{G} - M_{ms} \quad \forall(p,t), p\le d_p \\ & c \le \frac{L}{d_p}, \forall c\in C \end{aligned} \tag{20}$$

After optimizing C and T_{ckpt}, the optimal checkpointing configuration $ckpt_0(p,k)$ can be obtained by Eq. 16.

3.3.4 Overhead Analysis of Chunk Scheduler. The overall algorithm of the chunk scheduler is a dynamic programming process (equation 14) with solving $T_{ckpt}(P)$ as an MILP problem. Therefore, the cost can be approximated as solving $\frac{n(n+1)}{2}$ MILP problems, where n denotes the number of chunks the longest sequences are divided into.

We adopt several optimization techniques in our implementation to accelerate the solving process. Firstly, a multiprocessing method is employed to solve these MILP problems concurrently on multiple cores or CPUs. As the cluster scales, there are also more CPUs available to amortize the solving

Algorithm 2: Stage-Aware Chunk-Level Adaptive Checkpointing Solving Based on MILP

Input: Chunks $\{S_k | k \le n\}$, GPU memory capacity $\mathcal{G}$, chunks windows $W_p(t)$
Output: Checkpointing configuration C and minimum recomputation cost T_{ckpt}

$\mathcal{G}' \leftarrow \mathcal{G} - max(\{M_{ms}(p) | p \le d_p\})$;
for $k \le n$ **do**
  $\mathcal{I}[k] \leftarrow \frac{M_{token}}{N} \sum_{s \in S_k} s + (1 - I_k)\frac{2eLD}{N} \sum_{s \in S_k} s$;
  $w[k] \leftarrow 3 - 2I_k$;
  $\mathcal{F}[k] \leftarrow \left(\frac{M_{token}}{L \cdot d_s} - \frac{eD \cdot w[k]}{d_s}\right) \sum_{s \in S_k} s$;
$C \leftarrow initialize\_integer\_milp\_vars(n + d_p - 1, max = \frac{L}{d_p})$;
for $p \le d_p$ **do**
  for $W_p(t) \in W_p$ **do**
    $C_{p,k} \leftarrow C[d_p - p + f2b[k]]$;
    $cons.add(\sum_{k \in W_p(t)}(\mathcal{I}[k] - \mathcal{F}[k] \cdot C_{p,k}) \le \mathcal{G}')$;
$obj \leftarrow minimize \sum_{c \in C} c$;
$C, T_{ckpt} \leftarrow solve\_milp(cons, obj)$;
return C, T_{ckpt}

overhead. Secondly, we disaggregate the scheduler to run on CPUs and overlap the solving process with the actual training process carried on GPUs by *pre-solving* the schedule plan of the next batch of sequences. In practice, we can finish the solving process within 5 seconds, which is negligible compared to a training iteration and can be fully overlapped.

4 Implementation

We implement InfiniPipe in approximately 5K lines of code using Python, CUDA, and Triton [40]. The SCIP [6] library is leveraged to solve the MILP problems. Built on PyTorch, InfiniPipe integrates the flash-attn [10, 11] library for variable-length sequence packing and adopts NCCL [1] as the communication backend. Additionally, several key points in our implementation are highlighted as follows.

Tailored FSDP for Elastic Pipeline Parallelism. FSDP operates orthogonally to sequence parallelism (SP) and is commonly combined with SP to reduce model state memory overhead. Although PyTorch FSDP [47] serves as the most widely-used implementation, its native version is not compatible with pipeline parallelism with gradient accumulation. InfiniPipe's runtime engine seamlessly integrates PyTorch FSDP with EPP, which features a dynamic pipeline schedule while preserving the ability to overlap ZeRO communications with computation.

Global Buffer for KV Intermediate Activations. TPP requires meticulous management of contexts, where keys, values, and their gradients have unique materialization and eviction patterns, differing from other activations. These patterns introduce additional complexity when combined with activation checkpointing. To prevent memory leaks, we explicitly manage the allocation and deallocation of these intermediate activations by maintaining a global buffer.

5 Experiments

5.1 Experiment Setup

Environments. Our testbed consists of four GPU servers, each equipped with 8 NVIDIA A800-40GB GPUs interconnected via NVLink (400 GB/s bandwidth). Inter-node communication is handled by a 400 Gb/s InfiniBand network. The software stack includes PyTorch 2.4.2, CUDA 11.8, and flash-attn 2.7.4.

Baseline Systems. We compare InfiniPipe against four state-of-the-art distributed training systems: Megatron-LM, DeepSpeed, FlexSP [42], and Seq1F1B [38]. Megatron-LM is the current general-purpose SOTA featuring 4D parallelism comprising TP (equipped with Megatron-style SP), DP (ZeRO-1), CP, and PP. DeepSpeed integrates ZeRO of three stages and Ulysses-style SP. Seq1F1B featuring 1F1B pipeline schedule is included as TPP baseline. The original approach of Seq1F1B divides sequences into a uniform number of chunks, which is not compatible and efficient in varied-length corpora. We implement an enhanced version of Seq1F1B that splits and packs sequences into fixed-sized chunks. FlexSP represents the previous SOTA training system on varied-length corpora.

Workloads. As transformer is the predominant architecture of LLM, we evaluate InfiniPipe to train GPT-series models (7B, 13B, 30B) on two famous real-world datasets: *CommonCrawl* and *GitHub*. The sequence length and token distribution of these two datasets are presented in Fig. 2. Excessively long sequences that exceed the context length are truncated.

Protocols. Baselines except FlexSP rely on sequence padding to handle length variability, as they are originally designed for fixed-length training corpora. For fair comparison, they are developed to support sequence packing. Megatron-LM and DeepSpeed employ the same way to process varied-length sequences into micro-batches as FlexSP. InfiniPipe and Seq1F1B employs Ulysses-style SP intra-node and PP inter-node. For Megatron-LM and DeepSpeed, we manually tune their parallelism strategies according to specific workload requirements. All systems use activation checkpointing configurations optimized for a 96K context length. Evaluation metrics such as iteration time and token throughput are averaged during 20 training iterations. Global batch size refers to the number of sequences in a training iteration.

5.2 End-to-End Performance

We evaluate the performance of InfiniPipe by measuring the average end-to-end time of a training iteration with global batch size fixed to 512, as shown in Fig. 8. Experiments are conducted across various datasets, model sizes, and context lengths. Comprehensive results demonstrate that InfiniPipe

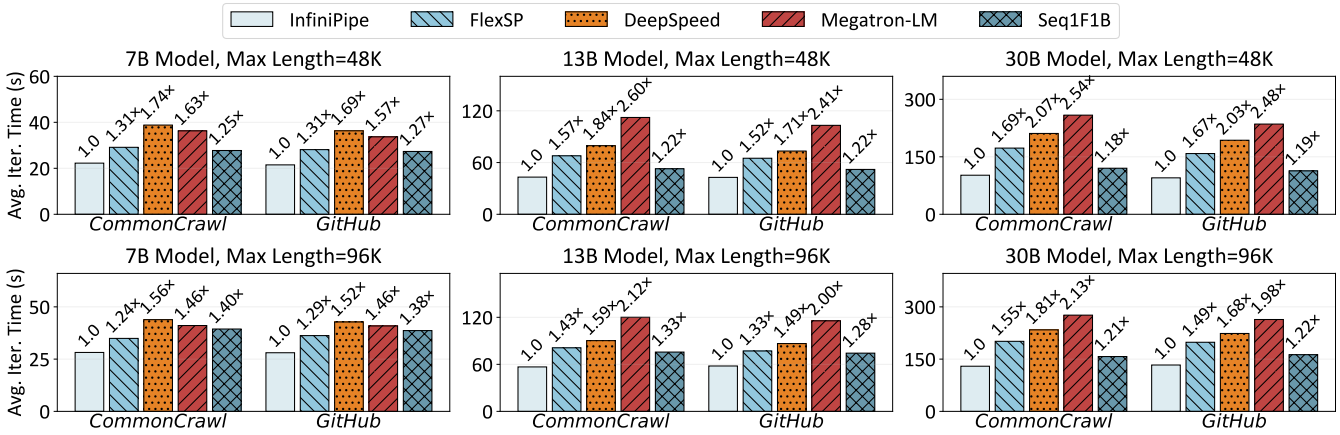

Figure 8. Average end-to-end time of a training iteration under different settings of model sizes, context lengths, and datasets with speedup ratio of InfiniPipe compared to baselines presented. For Megatron-LM, the TP degree is fixed to 8 and the CP degree is set to 2 for the 7B model, while 4 for the others. For DeepSpeed, SP degree is set to 16 for the 7B model and 32 for the others.

consistently outperforms baselines, achieving a maximum speedup of 1.69× compared to FlexSP, 2.07× compared to DeepSpeed, and 2.60× compared to Megatron-LM.

The performance gains of InfiniPipe on baseline systems except Seq1F1B primarily stem from the high communication efficiency of EPP. Specifically, DeepSpeed and FlexSP adopt Ulysses-style SP, where FSDP (ZeRO-3) is required to be applied on the whole cluster to shard the parameters and reduce gradients, resulting in frequent inter-node gather and scatter communications. Moreover, the sequence parallelism pattern of DeepSpeed and Megatron-LM introduces costly inter-node communication overhead and harms training efficiency, which has been discussed in Appendix A. InfiniPipe restricts SP and FSDP communication intra-node by applying PP inter-node, significantly reducing the inter-node communication overhead.

FlexSP leverages heterogeneous sequence parallel groups to reduce the communication overhead of static Ulysses-style SP, where shorter sequences are scheduled with smaller SP groups with efficient intra-node communication. To this end, FlexSP accelerates DeepSpeed and Megatron-LM up to 1.33× and 1.66× respectively. However, this approach introduces workload unbalance across SP groups, and a longer sequence also necessitates being processed by a larger SP group, where the introduced inter-node communication overhead can not be ignored. This drawback is exacerbated when training larger models with limited resources. Correspondingly, the speedup of InfiniPipe compared to FlexSP increases as model size scales, ranging from 1.31× to 1.69× on *CommonCrawl* dataset with context length of 48K.

Seq1F1B suffers from pipeline bubbles resulting from workload unbalance across chunks. As context length scales, the unbalance of workload becomes more pronounced due to the enlarged variance of sequence lengths. As a result, InfiniPipe

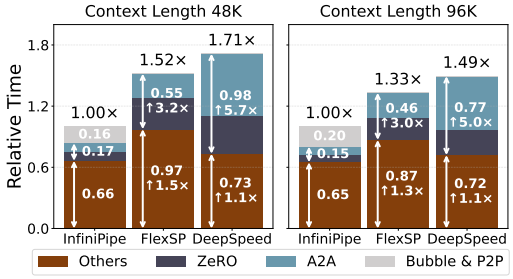

Figure 9. Case Study. End-to-end time breakdown of an iteration to train the 13B model with a fixed batch size of 512. The relative time and corresponding speedup of each component are indicated.

achieves a maximum speedup of 1.27×and 1.40×at a context length of 48K and 96K, respectively. Moreover, Seq1F1B adopts a non-optimal and uniform checkpointing configuration to accommodate the longest sequence, introducing more unnecessary computation when handling relatively small models. The adaptive chunk-level checkpointing pattern of InfiniPipe reduces unnecessary recomputation overhead and further enhances training efficiency.

5.3 Case Study

To better understand InfiniPipe's performance advantages more in depth, we breakdown the end-to-end training time into several components: "ZeRO" (gather and scatter communications of ZeRO-3 that are not overlapped), "A2A" (All-to-All communication in Ulysses-style SP), "Bubble & P2P" (pipeline bubbles in PP) and "Others" (computation, optimizer step and e.t.c). The profiled time cost of each component is shown in Fig. 9.

To begin with, InfiniPipe exhibits a similar performance in computation against DeepSpeed with a 1.11× improvement but outperforms FlexSP from 1.33× to 1.46× which is

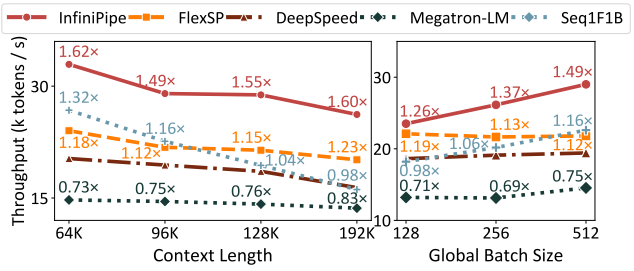

Figure 10. Scalability study. Token throughput measured to train a 13B model under different context lengths and global batch sizes. Indicated improvements are normalized to DeepSpeed.

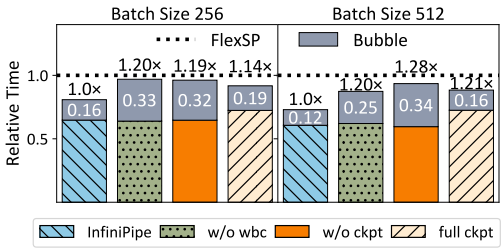

Figure 11. Ablation study. Normalized end-to-end time and bubble overhead to train a 13B model with a 64K context length.

attributed to the unbalanced workload introduced by heterogeneous SP groups in FlexSP. Furthermore, "ZeRO" and "A2A" overhead are the main bottlenecks of baselines. However, these overheads account only for 17% of the total end-to-end training time of InfiniPipe. By employing efficient intra-node communication, InfiniPipe reduces these overheads significantly by up to 3.2× compared to FlexSP and 5.7× compared to DeepSpeed. Last but not least, the bubble ratio is maintained at a relatively low level, less than 20%, thanks to the workload-balanced chunking method and efficient pipeline schedule of InfiniPipe. The advantages above lead to overall speedup of 1.52× and 1.71× compared to FlexSP and DeepSpeed, respectively.

5.4 Scalability Study

As shown in Fig. 10, token throughput under different settings of context length and batch size is measured to asses the scalability of InfiniPipe.

Scalability w.r.t. context length. InfiniPipe consistently achieves superior performance against baseline systems when context length is extended from 64K to 192K, achieving a speedup from 1.30×to 1.37×compared to FlexSP and from 1.23×to 1.63×compared to Seq1F1B. As context length scales, the throughput of all systems tends to decrease, attributed to the increased computation overhead per token. Megatron-LM exhibits the least degradation because the quadratic complexity self-attention operator is overlapped in CP's P2P kernel, resulting in similar processing time per token. On the contrary, Seq1F1B appears to be the most sensitive to context length as the increasing variance of sequence lengths results in a more pronounced unbalance of workload across chunks, leading to severe pipeline bubbles. The evaluation is restricted to a maximum context length of 192K due to the limited cluster resource. In fact, InfiniPipe is designed as a scheduling optimization for PP, which is orthogonal to existing optimizations for horizontal parallelisms such as TP, DP, and SP. Therefore, InfiniPipe can be seamlessly integrated with frameworks such as FlexSP and Megatron-LM, enabling further training acceleration and ultra-long context training when deployed at a cluster of larger scale.

Scalability w.r.t. global batch size. As global batch size ranges from 128 to 512, InfiniPipe consistently outperforms baselines and its performance exhibits a growing trend with throughput improved by up to 1.18×. In contrast, the throughput of baseline systems remains almost the same due to a similar computation overhead per token. Benefited from the lowered bubble ratio with more sequences, InfiniPipe delivers a 1.33× and 1.49× throughput improvement compared to FlexSP and DeepSpeed, respectively.

5.5 Ablation Study

To validate the effectiveness of InfiniPipe's key components, i.e., workload-balanced chunking and co-optimization approach of pipeline schedule and checkpointing, we compared InfiniPipe with three ablated versions. Specifically, "w/o wbc" denotes evenly splitting long sequences and packing short sequences into fixed-length chunks, "w/o ckpt" refers to disabling gradient checkpointing, and "full ckpt" represents applying full checkpointing.

As shown in Fig. 11, the variants exhibit distinct computation and pipeline bubble overhead. Despite introducing no recomputation overhead, "w/o ckpt" brings limited computational benefits compared to InfiniPipe due to the degradation of hardware utilization resulting from the finer granularity of a micro-batch. Moreover, the bubble ratios of all methods except "w/o ckpt" decrease as the global batch size scales. This occurs as an increasing number of excessively long sequences forces scheduling of more 1F1B units, introducing severe warmup-cooldown overhead. "w/o wbc" suffers from bubble overhead caused by workload unbalance while "full ckpt" incurs higher computation overhead due to suboptimal checkpointing configuration. Thanks to the co-optimization approach, InfiniPipe consistently outperforms these variants with relatively low bubble ratio and computation overhead.

6 Related Work

Long Context Training. Many sequence parallelism patterns for long context training have been proposed [7, 24, 30],

which can be used to replace Ulysses-style SP and are orthogonal to our method. Other works [16, 42] observe the skewness distribution of sequence length and aim to address workload heterogeneity. These works are orthogonal because optimization for PP and checkpointing are not considered.

Pipeline Parallelism Optimization. Recent works like AdaPipe [39], Mario [31], and SPPO [9] have explored checkpointing and offloading optimizations with PP. However, these works assume homogeneous workloads but we focus on heterogeneous workloads with varied-length input. ChunkFlow [45] introduces hierarchical processing for varied-length sequences, but its methodology resembles the combination of our ablation cases "w/o wbc" and "w/o ckpt", exhibiting suboptimal performance compared to our co-optimization approach. ByteScale [15] and WLB-LLM [43] optimize workload balance of batch-level PP for heterogeneous workload. Their optimizations can be integrated with our methodology for hybrid granularity PP and are orthogonal to our work.

7 Conclusion

In this paper, we propose EPP and build InfiniPipe, a novel LLM distributed training system that efficiently applies pipeline parallelism in long context training of a varied-length corpus. Comprehensive evaluations on various workloads demonstrate that InfiniPipe significantly reduces communication overhead and improves training throughput up to 1.69× compared to existing SOTA systems.

References

[1] Nvidia collective communications library (nccl). https://developer.nvidia.com/nccl, 2021.

[2] Pytorch gpipe. https://pytorch.org/docs/stable/pipeline.html, 2021.

[3] Introducing meta llama 3: The most capable openly available llm to date. https://ai.meta.com/blog/meta-llama-3/, 2024.

[4] Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[5] Beaumont, O., Eyraud-Dubois, L., and Shilova, A. Efficient combination of rematerialization and offloading for training dnns. *Advances in Neural Information Processing Systems 34* (2021), 23844–23857.

[6] Bolusani, S., Besançon, M., Bestuzheva, K., Chmiela, A., Dionísio, J., Donkiewicz, T., van Doornmalen, J., Eifler, L., Ghannam, M., Gleixner, A., Graczyk, C., Halbig, K., Hedtke, I., Hoen, A., Hojny, C., van der Hulst, R., Kamp, D., Koch, T., Kofler, K., Lentz, J., Manns, J., Mexi, G., Mühmer, E., Pfetsch, M. E., Schlösser, F., Serrano, F., Shinano, Y., Turner, M., Vigerske, S., Weninger, D., and Xu, L. The SCIP Optimization Suite 9.0. Technical report, Optimization Online, February 2024.

[7] Brandon, W., Nrusimha, A., Qian, K., Ankner, Z., Jin, T., Song, Z., and Ragan-Kelley, J. Striped attention: Faster ring attention for causal transformers. *CoRR abs/2311.09431* (2023).

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

[9] Chen, Q., Li, S., Gao, W., Sun, P., Wen, Y., and Zhang, T. Sppo: Efficient long-sequence llm training via adaptive sequence pipeline parallel offloading. *arXiv preprint arXiv:2503.10377* (2025).

[10] Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. *CoRR abs/2307.08691* (2023).

[11] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022* (2022), S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds.

[12] DeepSeek-AI, Liu, A., Feng, B., Wang, B., Wang, B., Liu, B., Zhao, C., Dengr, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Xu, H., Yang, H., Zhang, H., Ding, H., Xin, H., Gao, H., Li, H., Qu, H., Cai, J. L., Liang, J., Guo, J., Ni, J., Li, J., Chen, J., Yuan, J., Qiu, J., Song, J., Dong, K., Gao, K., Guan, K., Wang, L., Zhang, L., Xu, L., Xia, L., Zhao, L., Zhang, L., Li, M., Wang, M., Zhang, M., Zhang, M., Tang, M., Li, M., Tian, N., Huang, P., Wang, P., Zhang, P., Zhu, Q., Chen, Q., Du, Q., Chen, R. J., Jin, R. L., Ge, R., Pan, R., Xu, R., Chen, R., Li, S. S., Lu, S., Zhou, S., Chen, S., Wu, S., Ye, S., Ma, S., Wang, S., Zhou, S., Yu, S., Zhou, S., Zheng, S., Wang, T., Pei, T., Yuan, T., Sun, T., Xiao, W. L., Zeng, W., An, W., Liu, W., Liang, W., Gao, W., Zhang, W., Li, X. Q., Jin, X., Wang, X., Bi, X., Liu, X., Wang, X., Shen, X., Chen, X., Chen, X., Nie, X., Sun, X., Wang, X., Liu, X., Xie, X., Yu, X., Song, X., Zhou, X., Yang, X., Lu, X., Su, X., Wu, Y., Li, Y. K., Wei, Y. X., Zhu, Y. X., Xu, Y., Huang, Y., Li, Y., Zhao, Y., Sun, Y., Li, Y., Wang, Y., Zheng, Y., Zhang, Y., Xiong, Y., Zhao, Y., He, Y., Tang, Y., Piao, Y., Dong, Y., Tan, Y., Liu, Y., Wang, Y., Guo, Y., Zhu, Y., Wang, Y., Zou, Y., Zha, Y., Ma, Y., Yan, Y., You, Y., Liu, Y., Ren, Z. Z., Ren, Z., Sha, Z., Fu, Z., Huang, Z., Zhang, Z., Xie, Z., Hao, Z., Shao, Z., Wen, Z., Xu, Z., Zhang, Z., Li, Z., Wang, Z., Gu, Z., Li, Z., and Xie, Z. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024.

[13] Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[14] Fan, S., Rong, Y., Meng, C., et al. DAPPLE: a pipelined data parallel approach for training large models. In *PPoPP* (2021), ACM, pp. 431–445.

[15] Ge, H., Feng, J., Huang, Q., Fu, F., Nie, X., Zuo, L., Lin, H., Cui, B., and Liu, X. Bytescale: Efficient scaling of llm training with a 2048k context length on more than 12,000 gpus. *arXiv preprint arXiv:2502.21231* (2025).

[16] Ge, H., Fu, F., Li, H., Wang, X., Lin, S., Wang, Y., Nie, X., Zhang, H., Miao, X., and Cui, B. Enabling parallelism hot switching for efficient training of large language models. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (2024), pp. 178–194.

[17] Herrmann, J., Beaumont, O., Eyraud-Dubois, L., Hermann, J., Joly, A., and Shilova, A. Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory. *arXiv preprint arXiv:1911.13214* (2019).

[18] Jacobs, S. A., Tanaka, M., Zhang, C., Zhang, M., Song, S. L., Rajbhandari, S., and He, Y. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *CoRR abs/2309.14509* (2023).

[19] Jiang, C., Jia, Z., Zheng, S., Wang, Y., and Wu, C. Dynapipe: Optimizing multi-task training through dynamic pipelines. In *Proceedings of the Nineteenth European Conference on Computer Systems* (2024), pp. 542–559.

[20] Korthikanti, V., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models, 2022. *URL https://arxiv.org/abs/2205.05198*.

[21] Korthikanti, V., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models. *CoRR abs/2205.05198* (2022).

[22] Krell, M. M., Kosec, M., Perez, S. P., and Fitzgibbon, A. Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance. *arXiv preprint arXiv:2107.02027* (2021).

[23] Li, A., Gong, B., Yang, B., Shan, B., Liu, C., Zhu, C., Zhang, C., Guo, C., Chen, D., Li, D., et al. Minimax-01: Scaling foundation models with lightning attention. *arXiv preprint arXiv:2501.08313* (2025).

[24] Li, D., Shao, R., Xie, A., Xing, E. P., Gonzalez, J. E., Stoica, I., Ma, X., and Zhang, H. Lightseq: Sequence level parallelism for distributed training of long context transformers. *CoRR abs/2310.03294* (2023).

[25] Li, D., Shao, R., Xie, A., Xing, E. P., Ma, X., Stoica, I., Gonzalez, J. E., and Zhang, H. Distflashattn: Distributed memory-efficient attention for long-context llms training. In *First Conference on Language Modeling* (2024).

[26] Li, S., and Hoefler, T. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2021), pp. 1–14.

[27] Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., and Chintala, S. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow. 13*, 12 (2020), 3005–3018.

[28] Li, Z., Zhuang, S., Guo, S., Zhuo, D., Zhang, H., Song, D., and Stoica, I. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning* (2021), PMLR, pp. 6543–6552.

[29] Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[30] Liu, H., Zaharia, M., and Abbeel, P. Ring attention with blockwise transformers for near-infinite context. *CoRR abs/2310.01889* (2023).

[31] Liu, W., Li, M., Tan, G., and Jia, W. Mario: Near zero-cost activation checkpointing in pipeline parallelism. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (2025), pp. 197–211.

[32] Liu, Z., Cheng, S., Zhou, H., and You, Y. Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2023), pp. 1–13.

[33] Narayanan, D., Phanishayee, A., Shi, K., Chen, X., and Zaharia, M. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning* (2021), PMLR, pp. 7937–7947.

[34] Narayanan, D., Shoeybi, M., Casper, J., et al. Efficient large-scale language model training on GPU clusters using megatron-lm. In *SC* (2021), ACM, pp. 58:1–58:15.

[35] Qi, P., Wan, X., Huang, G., and Lin, M. Zero bubble (almost) pipeline parallelism. In *The Twelfth International Conference on Learning Representations* (2024).

[36] Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: memory optimizations toward training trillion parameter models. In *SC* (2020), IEEE/ACM.

[37] Sergeev, A., and Balso, M. D. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR abs/1802.05799* (2018).

[38] Sun, A., Zhao, W., Han, X., Yang, C., Zhang, X., Liu, Z., Shi, C., and Sun, M. Seq1f1b: Efficient sequence-level pipeline parallelism for large language model training. *arXiv preprint arXiv:2406.03488* (2024).

[39] Sun, Z., Cao, H., Wang, Y., Feng, G., Chen, S., Wang, H., and Chen, W. Adapipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (2024), pp. 86–100.

[40] Tillet, P., Kung, H.-T., and Cox, D. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (2019), pp. 10–19.

[41] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Canton-Ferrer, C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models. *CoRR abs/2307.09288* (2023).

[42] Wang, Y., Wang, S., Zhu, S., Fu, F., Liu, X., Xiao, X., Li, H., Li, J., Wu, F., and Cui, B. 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, Volume 2* (2025), pp. 421–436.

[43] Wang, Z., Cai, A., Xie, X., Pan, Z., Guan, Y., Chu, W., Wang, J., Li, S., Huang, J., Cai, C., et al. Wlb-llm: Workload-balanced 4d parallelism for large language model training. *arXiv preprint arXiv:2503.17924* (2025).

[44] Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).

[45] Yuan, X., Xu, H., Shen, W., Wang, A., Qiu, X., Zhang, J., Liu, Y., Yu, B., Lin, J., Li, M., et al. Efficient long context fine-tuning with chunk flow. *arXiv preprint arXiv:2503.02356* (2025).

[46] Zhao, H., Tian, Q., Li, H., and Chen, Z. {FlexPipe}: Maximizing training efficiency for transformer-based models with {Variable-Length} inputs. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)* (2025), pp. 143–159.

[47] Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., and Li, S. Pytorch FSDP: experiences on scaling fully sharded data parallel. *Proc. VLDB Endow. 16*, 12 (2023), 3848–3860.

Table 3. End-to-end time and the proportion of all-to-all communication to train GPT-13B employing Ulysses-style SP under different parallel degrees and context lengths (batch size fixed to 1).

SP Degree	32K	48K	72K	108K	162K
8	1.128 7.4%	2.101 5.9%	4.202 4.5%	8.793 3.2%	18.922 2.2%
16	1.188 47.5%	1.936 43.7%	3.378 37.6%	6.262 30.4%	12.207 23.4%
32	0.854 51.5%	1.307 50.5%	2.147 46.1%	3.766 39.4%	7.005 31.8%

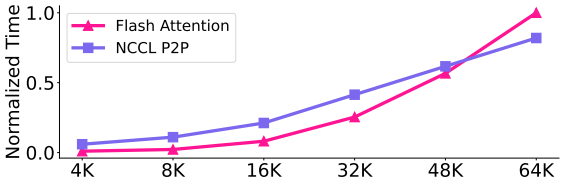

Figure 12. Profiled time of flash-attn and NCCL's p2p kernel under varying sequence lengths. The hidden dimension is set to 5120, and p2p communication is performed inter-node.

A Communication Overhead Analysis of Sequence Parallelism

Ulysses-style SP exhibits a lower communication complexity $O(\frac{L \cdot S \cdot D}{d_p})$ compared to $O(L \cdot S \cdot D)$ of TP, where L denotes the model's layers, S represents the sequence length, D refers to the model's hidden dimension and d_p reflects the parallel degree. However, the communication benefit is limited when applied on multiple machines. Considering the topology of modern clusters, the inter-node communication complexity of Ulysses-style SP is still $O(L \cdot S \cdot D)$, introducing severe communication overhead due to the low inter-node communication bandwidth. An experiment is conducted to evaluate the all-to-all communication overhead. As shown in Tab. 3, the communication overhead becomes the bottleneck of Ulysses-Style SP, accounting for a large portion of end-to-end time when scaling to multiple machines ($d_p > 8$). The performance degradation intensifies when deployed on larger-scale clusters with shorter context lengths.

As for Ring-style SP, it relies on overlapping p2p communication with the attention operation to achieve high performance. However, the parallel pattern faces limitations for a heterogeneous workload. Exemplify by CP of Megatron-LM, the self-attention is carried out by overlapped flash-attn [10, 11] and NCCL [1] p2p kernels. The overlapping of flash-attn and NCCL's p2p is only achievable for sequences longer than 64K, as demonstrated in Fig. 12. To this end, CP introduces high communication overhead as short sequences dominate a real-world dataset.

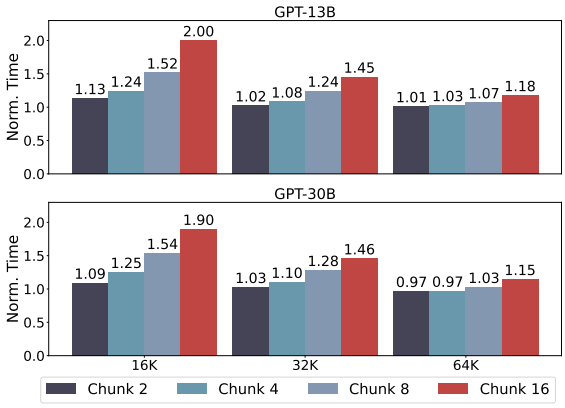

Figure 13. Profiled time to carry out forward and backward passes for a sequence using different chunking strategies. "Chunk" refers to the number of slices to split the sequence into. The time is normalized relative to training without chunking.

B Impact of Sequence Chunking on Computation Efficiency

Although sequence chunking of token-level PP effectively addresses batch-level PP's unbalanced memory footprint problem, there exists a side-effect on computation efficiency. Specifically, the granularity of a micro-batch, i.e., the number of tokens it contains, is closely related to the compute intensity of GPU operators such as GEMM in FFN layers. Sequence chunking leads to a finer granularity of a micro-batch and may harm hardware utilization. Experiments are conducted for 13B and 30B models under various context lengths and chunking strategies, where the slowdown ratio compared to training without chunking is used to quantify this impact. As shown in Fig. 13, a performance degradation is observed with shorter sequence exhibiting more sensitive to the chunking strategy.