FlexSP: Accelerating Large Language Model Training via Flexible Sequence Parallelism

Yujie Wang*
Peking University
Beijing, China
alfredwang@pku.edu.cn

Shiju Wang
Beihang University
Beijing, China
21373455@buaa.edu.cn

Shenhan Zhu*
Peking University
Beijing, China
shenhan.zhu@pku.edu.cn

Fangcheng Fu*
Peking University
Beijing, China
ccchengff@pku.edu.cn

Xinyi Liu*
Peking University
Beijing, China
xy.liu@stu.pku.edu.cn

Xuefeng Xiao
ByteDance Inc.
Beijing, China
xiaoxuefeng.ailab@bytedance.com

Huixia Li
ByteDance Inc.
Beijing, China
lihuixia@bytedance.com

Jiashi Li
ByteDance Inc.
Shenzhen, China
lijiashi@bytedance.com

Faming Wu
ByteDance Inc.
Beijing, China
wufaming@bytedance.com

Bin Cui*†
Peking University
Beijing, China
bin.cui@pku.edu.cn

Abstract

Extending the context length (i.e., the maximum supported sequence length) of LLMs is of paramount significance. To facilitate long context training of LLMs, sequence parallelism has emerged as an essential technique, which scatters each input sequence across multiple devices and necessitates communication to process the sequence. In essence, existing sequence parallelism methods assume homogeneous sequence lengths (i.e., all input sequences are equal in length) and therefore leverages a single, static scattering strategy for all input sequences. However, in reality, the sequence lengths in LLM training corpora exhibit substantial variability, often following a long-tail distribution, which leads to workload heterogeneity.

In this paper, we show that employing a single, static strategy results in inefficiency and resource under-utilization, highlighting the need for adaptive approaches to handle the heterogeneous workloads across sequences. To address this, we propose a heterogeneity-adaptive sequence parallelism method. For each training step, our approach captures the variability in sequence lengths and assigns the optimal combination of scattering strategies based on workload characteristics. We model this problem as a linear programming optimization and design an efficient and effective solver to find the optimal solution. Furthermore, we implement our method in a high-performance system that supports adaptive parallelization in distributed LLM training. Experimental results demonstrate that our system outperforms state-of-the-art training frameworks by up to 1.98×. Source code is available at https://github.com/PKU-DAIR/Hetu-Galvatron.

1 Introduction

Large Transformer models [12, 26, 35, 40, 45], represented by Large Language Models (LLMs) [3, 7, 36, 37, 42, 43, 49, 50, 54], have made astonishing achievements in the field of artificial intelligence (AI). Recently, there is an increasing need to extend the context length of LLMs, which represents the maximum supported sequence length of the LLMs [11, 15, 47]. Therefore, long-context LLM training has garnered extensive attention from both the academia and industry [4, 6, 8, 24, 27, 29, 48].

It is well known that training LLMs with longer context lengths demands increasingly more device (e.g., GPU) memory, and a promising paradigm is to parallelize the training inputs over multiple devices. Particularly, sequence parallelism (SP) [6, 19, 22, 24, 27, 29] has emerged as a pivotal technique for long-context LLM training. In a nutshell, SP splits each training input in the sequence dimension and scatters different shards onto multiple devices to amortize the memory consumption. In order to carry out the training, it necessitates communication among the devices to exchange the intermediate results during the forward and backward propagation. By nature, if we wish to support longer training

*School of Computer Science & Key Lab of High Confidence Software Technologies (MOE), Peking University
†Institute of Computational Social Science, Peking University (Qingdao)

inputs, we shall increase the SP degree[1] (usually accompanied by a decrease in the data parallelism degree) to avoid encountering out-of-memory errors, while the communication overhead increases in the meantime, leading to efficiency degradation.

Although SP delivers a method to support long-context LLM training, existing systems assume the training inputs are homogeneous in terms of their lengths and apply the same SP degree to all data parallel model replicas, yet such a fixed-length assumption does not hold in practice. Typically, the training samples of LLMs are usually organized as sequences of tokens, and a training corpus consists of varied-length sequences. In order to address the discrepancy between the fixed-length assumption and the varied-length characteristic, sequence packing is a widely used data preprocessing technique. Specifically, denote c as the maximum number of tokens supported for each model replica. Sequence packing concatenates multiple sequences into a longer one and ensure that the length of the packed sequence does not exceed c. (A sequence will be truncated if it exceeds c by itself.) Meanwhile, the attention masks and position indices are adjusted accordingly to avoid cross-contamination among the sequences. By this means, the training inputs[2] can be (nearly) homogeneous in terms of their lengths, and the model gradients computed over a packed input are identical to that computed over the original, unpacked sequences.

Nevertheless, we find that the aforementioned approach sacrifices the efficiency of short sequence processing to accommodate the memory requirement of long-sequence processing. Fig. 1 showcases an example, where there are 64 devices in total and the context length of the training task is 192K. Using the aforementioned approach, it necessitates an SP degree of 32, resulting in two SP=32 groups. Assume the current training step needs to process five sequences with lengths of 100K, 48K, 48K, 48K, and 48K, respectively. Due to the homogeneous parallelism designs of existing works, no matter how we pack the sequences (Case Homo-1 or Homo-2 in left side of Fig. 1), the short sequences (48K) must be processed with an SP degree of 32, and the All-to-All communication time and computation time of four 48K sequences are 1.2s and 2.8s, respectively. However, if we allow for heterogeneous SP groups (i.e., allow groups to have non-unique SP degrees), we can form one SP=32 group to process the long sequence (100K) and four SP=8 groups to process the short sequences (48K) separately (Case Hetero in right side of Fig. 1). By doing so, the computation time of four 48K sequences remains 2.8s, while the communication time decreases to only 0.2s,and the processing of the short sequences can be accelerated (from 4s to 3s in the example) due to the reduction in communication overhead. As we will detail in §3, common LLM training corpora follow long-tail distributions, i.e., there are very few long sequences while the short ones dominate the corpora. Consequently, by adjusting the SP groups, we can accelerate the training of most sequences, provisioning performance improvement.

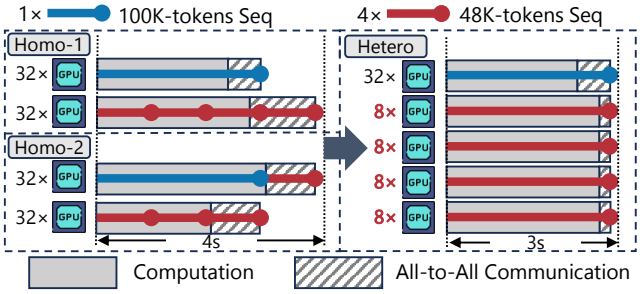


Figure 1. An example of heterogeneity-adaptive SP improving training efficiency for varied-length sequences.

Inspired by this, this work develops FlexSP, a Flexible Sequence Parallelism for the efficient training of LLMs over varied-length sequences. Unlike existing systems that organize training sequences to fit the homogeneous parallelism, FlexSP adapts to the heterogeneous workloads of varied-length sequences by dynamically adjusting the parallelism. Essentially, FlexSP puts forward two key innovations:

- *Heterogeneous Sequence Parallel Groups:* As aforementioned, processing long sequences necessitates a higher SP degree to avoid out-of-memory errors, while processing short sequences has a better efficiency if we use a lower SP degree. Consequently, for each training step, FlexSP adaptively forms multiple heterogenous SP groups so that sequences with diverse lengths can be processed with different groups, making a good tradeoff between memory consumption and training efficiency.
- *Time-Balanced Sequence Assignment:* To minimize the processing time of each sequence individually, a naïve approach is to assign each sequence to the smallest SP group that can handle it. However, due to the long-tail distribution of datasets, short sequences dominate, and such a naïve assignment causes small groups to handle too many workloads, causing a bottleneck. This leads to workload imbalances across the SP groups, where faster groups are forced to wait for slower ones in idle. To deal with this problem, FlexSP meticulously controls which SP group each sequence should be assigned to, striking a good balance across the heterogeneous SP groups.

FlexSP co-optimizes these two factors adaptively according to the sequence length variation of each training step. Specifically, given the sequences with diverse lengths, we formulate a joint optimization problem to maximize training efficiency, which determines how to form the heterogeneous SP groups and how to assign each sequence to the most

[1]For simplicity, we use the abbreviation "SP degree" to denote the parallelism degree of SP. This terminology extends naturally to other parallelism strategies (e.g., data parallelism degree).

[2]To avoid ambiguity, by saying "training inputs", we always mean the packed sequences that are fed into the training process.

suitable group. To solve this problem, we transform it into a Mixed-Integer Linear Programming (MILP) problem by accurately modeling the computation cost, communication cost, and memory consumption of training over varied-length sequences. Subsequently, we decrease the complexity of the MILP problem via a sequence bucketing algorithm based on dynamic programming, making it efficient to solve for practical considerations.

In addition, to resolve the potential issue that there are too many sequences that cannot be processed at once, we manage to chunk the sequences into micro-batches. In particular, we establish a series of propositions based on theoretical analysis and empirical observations. Based on this, a micro-batch chunking algorithm is devised, which aims to minimize the total training time of all micro-batches.

We implement FlexSP on top of PyTorch and conduct experiments with various model sizes, training datasets, and context lengths. Empirical results show that FlexSP outperforms state-of-the-art LLM training systems by up to 1.98×, demonstrating the effectiveness of flexible sequence parallelism in long-context LLM training.

The contribution of this work are summarized as follows: (1) *New System*. We introduce FlexSP, a brand new distributed training LLM system for varied-length corpora. (2) *New Perspective*. To the best of our knowledge, FlexSP is the first to adaptively adjust the parallelism strategies given the diverse lengths, matching the heterogeneous workloads caused by varied-length sequences with heterogeneous parallelism. Existing homogeneous systems overlook the heterogeneous workloads of varied-length contexts, and typically pack sequences to similar lengths, sacrificing the efficiency of short sequences. In contrast, FlexSP is based on first principles, supporting heterogeneous parallelisms that match the diverse workloads, proposing a new perspective of redesigning systems for more efficient training of powerful models (e.g., training over varied-length images/videos). (3) *State-of-the-art Performance*. Extensive experimental results verify that FlexSP consistently achieves the best training efficiency compared to existing works.

2 Preliminary

2.1 Parallelisms in Distributed Training

As model sizes and data volumes grow rapidly, various distributed parallelism techniques are employed in LLM training to distribute the workload across multiple devices.

2.1.1 Data Parallelism. Data parallelism (DP) [28] splits the data along sample dimension. Each device is responsible for a part of the input samples, and the gradients need to be synchronized across the devices. DP requires each device to maintain a complete copy of the model, which is redundant. To address this, sharded data parallelism (SDP) has been proposed, such as DeepSpeed-ZeRO [38] and PyTorch FSDP [52]. These methods not only split the data but also the model states across devices, allowing each device to store only a fraction of the model states and introducing additional communication to synchronize the model states.

2.1.2 Sequence Parallelism. Sequence parallelism (SP) also splits data, and can be considered a special form of DP. Unlike DP, which splits data across the sample dimension, sequence parallelism splits data across the sequence dimension. SP is designed to mitigate the memory shortage caused by increasingly longer context lengths of LLMs. DeepSpeed-Ulysses [19] proposes Ulysses-style SP, which splits the sequence dimension of linear projection in MLP and attention module, dropout modules, and normalization modules, and employs All-to-All primitives to collect and distribute sequences.

$$\mathbf{Q}_s, \mathbf{K}_s, \mathbf{V}_s = \mathbf{X}_s\mathbf{W}_Q, \mathbf{X}_s\mathbf{W}_K, \mathbf{X}_s\mathbf{W}_V \in \mathbb{R}^{\frac{N}{P}\times d} \tag{1}$$

$$\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h = \text{AlltoAll}(\mathbf{Q}_s, \mathbf{K}_s, \mathbf{V}_s) \in \mathbb{R}^{N\times \frac{d}{P}} \tag{2}$$

$$\mathbf{P}_h = \text{softmax}\left(\frac{\mathbf{Q}_h\mathbf{K}_h^\top}{\sqrt{d}}\right)\mathbf{V}_h \in \mathbb{R}^{N\times \frac{d}{P}} \tag{3}$$

$$\mathbf{O}_s = \text{AlltoAll}(\mathbf{P}_h\mathbf{W}_O) \in \mathbb{R}^{\frac{N}{P}\times d} \tag{4}$$

In the attention module of Ulysses-style SP, each device holds a portion of sequences $\mathbf{X}_s \in \mathbb{R}^{\frac{N}{P}\times d}$ and the complete model parameters $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V, \mathbf{W}_O \in \mathbb{R}^{d\times d}$, where N denotes the total sequence length, P denotes the SP degree, and d denotes the hidden size. After the linear projection of query, key and value (Eq. 1), three rounds of All-to-All communication are employed to collect the complete sequence on each device (Eq. 2). The multi-head attention operation is then calculated on the complete sentence (Eq. 3), and an another round of All-to-All communication is introduced to distribute the sequence across the devices (Eq. 4).

Megatron-LM also proposes Megatron-style SP [22], which splits only the dropout and normalization modules, and requires All-Gather and Reduce-Scatter communication. It can be treated a supplementary scheme proposed to be used in conjunction with Megatron-TP (§2.1.3), aimed at addressing the redundant activation memory usage of Megatron-TP, and they must have the same parallelism degree.

2.1.3 Other parallelisms.

Model Parallelism. Model parallelism distributes the model parameters across the cluster, which can be divided into two categories: tensor parallelism (TP) and pipeline parallelism (PP). TP splits the model vertically. Megatron-TP proposed by Megatron-LM [33] is the most widely used, which splits the tensor multiplication computations in each attention layer and feed-forward layer across multiple devices, incorporating communication to synchronize the computation results. PP [16, 18, 31, 32] partitions the model horizontally, with

GPipe [18], PipeDream-Flush [32] being notable implementations. These approaches divide the model layers into multiple stages placed across different devices, pass intermediate computation results with point-to-point communication, and orchestrate the model execution into a pipeline.

Context Parallelism. Context parallelism (CP) [6, 24, 27, 29] also splits the sequence dimension. Compared to sequence parallelism (SP) which splits dropout and normalization module activations but necessitates complete sentence for attention operation, CP further distributes the attention operation. Specifically, CP distributes sequence dimension of the query, key, and value across multiple devices, and involves additional ring communication to collect key and value for completing attention computations. Such extra communication volume is substantial, thus CP allows the computation to overlap the extra communication overhead by conducting the ring communication and computation of attention operation chunk by chunk.

2.1.4 LLM Training Systems and Hybrid parallelisms.

Modern LLM training systems [33, 39] usually combine multiple dimensions of parallelisms and support hybrid parallelism. For instance, Megatron-LM [33] supports 4D parallelism, including DP, TP (along with Megatron-style SP), PP, CP. DeepSpeed [39] supports SDP (DeepSpeed-ZeRO) and Ulysses-style SP. However, all of these existing parallelism techniques and LLM training systems are designed for training corpora with homogeneous context lengths and employ single and static strategy, ignoring the variability of sequence lengths in real-world datasets.

2.2 Common Techniques in LLMs Training

2.2.1 Gradient Accumulation.

Gradient accumulation is a practical technique to train large batch of data under constrained memory capacity. It simulates the large batch training by splitting data batch into several micro-batches, executing micro-batches sequentially, accumulating gradients and updating model until the last micro-batch finishes.

2.2.2 Sequence Packing.

To train the varied-length sequences together, techniques such as padding or packing are often required. Padding involves extending or truncating all sequences into the same length, which introduces redundant computation and memory overhead due to the excess padding for short sequences. In contrast, sequence packing [23] is a more advanced and commonly used technique that concatenates sequences of different lengths into a single long sequence, where adjustments to the attention masks and position indices are required to prevent cross-contamination. Sequence packing eliminates the redundancy caused by padding the varied-length sequences. [13] proposes *Best-fit Packing*, employing *Best-Fit-Decreasing* (BFD) to pack sequences to best fit device memory. In our work, sequence packing is the default setting.

Table 1. Iteration time (s) of GPT-7B with SP, along with the All-to-All communication ratios of different sequence lengths (seq), batch size (bs) pairs and SP degrees, tested on 64 A100 GPUs, equipped with NVLink for intra-node (SP ≤ 8) and InfiniBand for inter-node (SP > 8) communication.

seq × bs	SP=64	SP=32	SP=16	SP=8	SP=4
4K × 1024	37.2 54.4%	33.6 45.0%	27.9 33.2%	19.2 8.1%	**18.9** **7.3%**
8K × 512	37.6 51.9%	34.9 42.4%	29.1 31.4%	20.9 7.8%	**20.4** **7.1%**
16K × 256	40.6 47.8%	37.6 38.6%	32.6 29.2%	23.4 6.9%	**23.3** **6.2%**
32K × 128	47.5 41.6%	44.0 33.3%	39.4 24.2%	**30.4** **5.7%**	OOM
64K × 64	61.5 34.1%	56.2 25.1%	**51.8** **18.6%**	OOM	OOM
128K × 32	85.0 23.5%	**82.8** **18.5%**	OOM	OOM	OOM
256K × 16	**137.2** **16.4%**	OOM	OOM	OOM	OOM

3 Observation and Opportunities

In this section, we introduce our observations, and illustrate the optimization opportunities.

Observation 1: Long sequences require large parallelism groups, which leads to inefficiency. As mentioned in §1, existing works simply assume a homogeneous sequence length and therefore apply a homogeneous SP degree in each training task. To better illustrate the drawbacks of homogeneous SP design, we conduct a small testbed to assess its efficiency when facing different levels of sequence lengths. Tab. 1 presents the end-to-end execution time along with the proportion of All-to-All communication. Particularly, for each row in Tab. 1, we generate fixed-length sequences until the total token number has reached 4 million, and train these sequences with divergent SP degrees. Firstly, we find that long sequences require larger parallelism groups due to memory constraint. For instance, a 32K sequence can be handled by SP groups with a degree of 8, whereas a 128K sequence requires a degree of at least 32 to fit within device memory. Secondly, larger parallelism groups can result in inefficiencies, primarily because of increased communication overhead. Smaller groups tend to perform better. For example, with 512 sequences of 8K, an SP group with a degree > 8 leads to communication time exceeding 31.4% (9.1s) due to the slow inter-node bandwidth. In contrast, groups with a degree of 8 reduce communication to just 7.8% (1.6s), benefiting from the high-bandwidth intra-node connection. In this case, the computation time remains unchanged (approximately 19s) for variable-degree SP groups, as expected, and the communication is much more efficient on smaller parallelism groups.

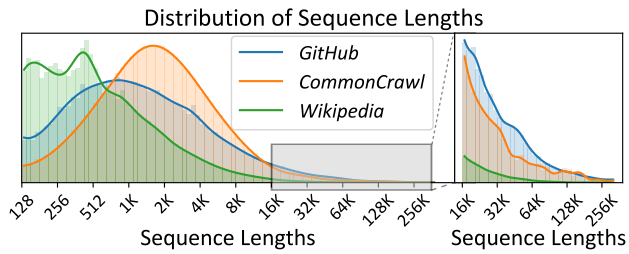


Figure 2. Distribution of sequence lengths across different datasets. The height of each bar represents the percentage of sequences in the corresponding length range. Details of excessively long sequences are expanded into the right panel.

Observation 2: Real-world datasets present skewness in sequence length distribution, exhibiting a long-tail distribution. Fig. 2 illustrates the sequence length distribution for three popular LLM training datasets, which are *GitHub*, *CommonCrawl*, and *Wikipedia*. We observe that all three datasets exhibit a pronounced uni-modal long-tail distribution, with the majority of sequences falling below 8K in length, while only a small fraction of sequences exceed 32K. *GitHub* contains the largest number of excessively long sequences, followed by *CommonCrawl*, with *Wikipedia* having the fewest. Consequently, the sequence lengths in real-world datasets demonstrate a notable skewness, following a long-tailed distribution.

Optimization Opportunity: Matching heterogeneous parallelism groups with heterogeneous sequence lengths. Current systems are tailored for homogeneous sequence lengths and employ single, static parallelism strategies throughout the training process. However, dealing with sequences of long-tail distribution in lengths requires large parallelism groups to accommodate the excessively long sequences, which diminishes efficiency for shorter sequences. For instance, when a sequence of 128K exists in the dataset, all sequences of 8K are forced to use SP groups of size at least 32, failing to enjoy the efficient smaller groups. Furthermore, given that short sequences are more common in skewed distributions, this inefficiency is pronounced. Therefore, we propose to adapting appropriate parallelism strategies, crafting heterogeneous parallelism groups to match the heterogeneous workloads caused by varied-length sequences. Specifically, we wish to form small groups for short sequences to improve efficiency, while retaining large groups for long sequences to avoid out-of-memory errors. Additionally, we need to properly control the assignment of sequences to balance the execution time among all parallelism groups. Such flexible, heterogeneity-adaptive strategies will improve communication efficacy and overall system performance.

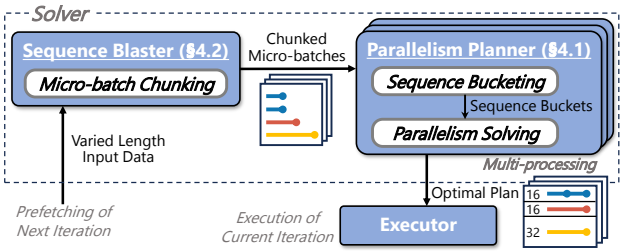


Figure 3. FlexSP system overview.

4 FlexSP

Fig. 3 outlines the system overview of FlexSP, which consists of the solver and the executor. Given a batch of sequences with diverse lengths, the solver deduces the optimal plan of heterogeneous parallelism groups and sequence assignment. In particular, there are two major steps in the solver. Firstly, the sequence blaster chunks the sequences into micro-batches, ensuring that each micro-batch will not be too large to accommodate. Secondly, the parallelism planner is responsible for solving the optimal plan for each micro-batch to minimize its execution time. Following the optimal plan, the executor carries out the training of one iteration.

In this section, we focus on the details of FlexSP's solver. Frequently used notations are listed in Tab. 2.

4.1 Parallelism Planner

We first introduce FlexSP parallelism planner, which deduces the optimal sequence parallelism (SP) strategies and sequence assignment, maximizing training efficiency.

4.1.1 Problem Formulation. We first formulate the optimization problem of FlexSP parallelism planner. Given a data batch containing K sequences $\{\mathcal{S}_k\}$ that vary in lengths, and N devices with device memory budget E, the factors that we need to determine are: (1) the number of SP groups, (2) the parallel degree of each SP group, and (3) which SP group should each sequence be assigned to. Meanwhile, as sequences within different SP groups are processed concurrently, the optimization target is to minimize the maximum execution time of all SP groups.

Since the candidate set of SP degrees is very small[3], and each sequence can only be assigned to one SP group, we transform all decision variables into 0-1 integer variables. In particular, we assume there are P virtual SP groups, where the p^{th} SP group $\mathcal{G}_p$ has an SP degree of d_p. For instance, if there are 2 GPUs, we have three virtual groups with SP degrees of 1, 1, and 2, respectively. Then, we define the group selection vector $\boldsymbol{m} = \langle m_1, m_2, ..., m_P \rangle \in \{0, 1\}^P$, where $m_p = 1$ indicates that $\mathcal{G}_p$ is selected while $m_p = 0$ means the opposite. By doing so, the number of SP groups and the parallel

[3]In common, SP degrees are set as powers of 2 to fit the "binary structure" of chips and networks. Besides, the highest SP degree is restricted by the number of GPUs and the context length.

Table 2. Notations used in this work.

N	The number of available GPUs
P	The number of virtual sequence parallel (SP) groups
$\mathcal{G}_p$	The p^{th} group
d_p	The SP degree of $\mathcal{G}_p$
K	The number of sequences
$\mathcal{S}_k$	The k^{th} sequence
s_k	The sequence length of $\mathcal{S}_k$
Q	The number of buckets after sequence bucketing
$\mathcal{B}_q$	The q^{th} bucket
$\hat{s}_q$	The upper limit of sequence length of $\mathcal{B}_q$
$\hat{b}_q$	The number of sequences in $\mathcal{B}_q$

degree of each SP group can be easily described via $\boldsymbol{m}$. Subsequently, we further define the sequence assignment matrix $\boldsymbol{A} \in \{0,1\}^{K\times P}$, where $A_{k,p} = 1$ represents that $\mathcal{S}_k$ is assigned to $\mathcal{G}_p$. Based on this, we can formulate a joint-optimization problem of the SP group selection and sequence assignment:

$$\underset{\boldsymbol{m}\in\{0,1\}^P;\boldsymbol{A}\in\{0,1\}^{K\times P}}{\arg\min} \quad C \tag{5}$$

$$\text{s.t.} \quad \text{Time}(\{s_k, A_{k,p}\}; d_p) \le C,\ \forall p \in [1,P] \tag{6}$$

$$\text{Memory}(\{s_k, A_{k,p}\}; d_p) \le E, \forall p \in [1,P] \tag{7}$$

$$\textstyle\sum_p d_p \times m_p \le N \tag{8}$$

$$\textstyle\sum_k A_{k,p} \le m_p \times K,\ \forall p \in [1,P] \tag{9}$$

$$\textstyle\sum_p A_{k,p} = 1,\ \forall k \in [1,K] \tag{10}$$

Here, $\text{Time}(\{s_k, A_{k,p}\}; d_p)$ and $\text{Memory}(\{s_k, A_{k,p}\}; d_p)$ denotes the execution time and the memory consumption on each device in SP group $\mathcal{G}_p$, which will be illustrated in §4.1.2 in detail. The optimization target is to minimize the maximum execution time among all SP groups (Cond. (6)). Cond. (7) represents the memory constraint of each device in each SP group. Cond. (8) denotes that the total parallelism degrees of all selected SP groups should not be larger than the cluster device number N. Cond. (9) ensures that no sequences will be assigned a virtual SP group that is not selected. Cond. (10) requires that each sequence must be assigned to one and only one group.

4.1.2 Cost Estimation. To solve the optimization problem (5), it is necessary to estimate $\text{Time}(\{s_k, A_{k,p}\}; d_p)$ and $\text{Memory}(\{s_k, A_{k,p}\}; d_p)$ accurately. Next, we then analyze the memory consumption and execution time of sequence parallelism with input sequences of variant lengths.

Memory consumption has two components: model states and forward activations. Firstly, given a model, in Ulysses-style SP, the memory consumption of model states depends only on the ZeRO-stage applied and the number of available devices N. For instance, when ZeRO-3 is applied, the model states are evenly sharded over N devices, unaffected by SP group selection or sequence assignment. Secondly, for a device SP group $\mathcal{G}_p$ with an SP degree of d_p, the activation memory cost is proportional to the total number of tokens (i.e., the summed lengths of sequences) assigned to $\mathcal{G}_p$, and inversely proportional to the SP degree d_p. This is because sequence parallelism scatters the tokens evenly across the devices within the SP group. Therefore, for each device in SP group $\mathcal{G}_p$, its memory consumption can be estimated as:

$$\text{Memory}(\{s_k, A_{k,p}\}; d_p) = \sum_k \frac{A_{k,p} s_k}{d_p} \text{M}_{token} + \text{M}_{ms}, \tag{11}$$

where M_{ms} denotes the memory consumption of model states memory that is fixed across all devices, M_{token} represents the activation memory cost of each token, and $A_{k,p}$ denotes whether $\mathcal{S}_k$ is assigned to $\mathcal{G}_p$.

Previous works [30, 46, 53] have proposed an effective execution cost model for distributed training of LLMs with fixed-length sequences, commonly utilizing the α-β model [17] $T = \alpha W + \beta$ to estimate the communication and the computation overhead, where W represents the workload (e.g., the computation FLOPs or communication volumes), α reflects the execution rate (e.g., the per-FLOP computation time or the per-byte communication time), and β denotes the fixed overhead (e.g., kernel launch latencies). However, existing works assume homogeneous sequence lengths and fail to accurately estimate costs for varied sequence lengths. Therefore, FlexSP extends the α-β model, making sequence length the independent variable to handle real-world training corpora. Specifically, to adapt to Transformer models, we model the computation cost of the attention mechanism and the other modules separately. The reason is that the computation cost of attention mechanism is positively correlated with the quadratic of sequence length, while the other modules like linear projection have a linear computation cost w.r.t. the sequence length. Besides, sequence parallelism scatters the computation across the devices within an SP group $\mathcal{G}_p$, thus the per-device computation volume is inversely proportional to SP degree d_p. Therefore, by summing the computation cost of all assigned sequences, we estimate the computation overhead as follows:

$$\text{T}_{comp}(\{s_k, A_{k,p}\}; d_p) = \frac{1}{d_p} \sum_k A_{k,p}(\alpha_1 s_k^2 + \alpha_2 s_k) + \beta_1, \tag{12}$$

where $\alpha_1, \alpha_2, \beta_1$ denote the coefficients of the α-β model for computation cost, which are obtained through profiling.

The communication volume of Ulysses-style SP mainly comes from the All-to-All communication, whose volume is proportional to the sequence length s_k and inversely proportional to SP degree d_p [19]. Hence, FlexSP estimate the All-to-All communication cost as follows:

$$\text{T}_{comm}(\{s_k, A_{k,p}\}; d_p) = \frac{1}{d_p v_p} \sum_k A_{k,p} \alpha_3 s_k + \beta_2, \tag{13}$$

where α_3, β_2 are coefficients given by profiling, and v_p represents the interconnect bandwidth of the devices within $\mathcal{G}_p$, which can also be profiled out.

As can be seen, FlexSP draws inspiration from the α-β model, using β. to represent the data-independent startup latency, while utilizing the α. to fit the time cost for both communication and computation process according to their respective behaviors. Then, we combine them to estimate the overall execution time of sequence parallelism with varied-length sequences as follows:

$$\text{Time}(\{s_k, A_{k,p}\}; d_p) = \text{T}_{comp} + \text{T}_{comm}. \tag{14}$$

Furthermore, when combining sequence parallelism with ZeRO (especially ZeRO-3), we also estimate the overhead of parameter gathering and gradient synchronization, and also consider the overlapping of computation and communication like previous works [30, 46]. As ZeRO is orthogonal to our work, and its overhead is unrelated with the sequence parallelism nor the sequence lengths, we omit such details in this paper for clarity. Experiments show that the overall cost estimation error is below 6%, detailed in Appendix C.

4.1.3 Problem Solving. According to the problem formulation in §4.1.1 and the overhead estimation in §4.1.2, we can find that all the constraints and the optimization target is linear with respect to the decision variables m_p and $A_{k,p}$. Therefore, the optimization problem (5) turns out a Mixed-Integer Linear Programming (MILP) problem. Although existing advanced MILP solvers like SCIP [5] are capable of solving MILP problems, the number of decision variables in problem (5) is too large and uncontrollable, making it too complex to derive feasible solutions within a reasonable time. To tackle this obstacle, we need to simplify the problem to decrease the number of decision variables.

In particular, since the number of decision variables is proportional to the number of sequences, and sequences with similar lengths should incur similar overhead, we opt to group the sequences into a small number of buckets. In other words, given the sequences with various lengths, we group the sequences with similar lengths in the same bucket and represented by a unified sequence length (typically, the maximum sequence length within the bucket). Although this will introduce certain estimation biases, it can significantly reduce the number of unique sequence lengths and thereby lower the problem complexity. Below, we introduce our sequence bucketing algorithm.

A naïve method for sequence bucketing is to set a fixed length interval for each bucket and use its upper limit to represent the length of sequences within the bucket. For instance, the upper limit of sequence length can be set as multiples of 2K, that is, 0-2K, 2K-4K, 4K-6K, and so on, forming several buckets. However, as discussed in §3, the sequence lengths in real-world datasets exhibit a complex long-tail distribution rather than a uniform distribution. Besides, different datasets exhibit distinct distributions. Consequently, such a naïve bucketing method would inevitably introduce large estimation biases and cannot be generalized.

To reduce the estimation biases caused by bucketing, we adopt an adaptive sequence bucketing mechanism and propose a dynamic programming algorithm to minimize the bucketing deviation. Specifically, given K sequences $\{\mathcal{S}_k\}$, we group them into Q buckets, where the q^{th} bucket $\mathcal{B}_q$ has the upper limit of sequence length $\hat{s}_q$, and containing all sequences satisfying $\hat{s}_{q-1} < s_k \le \hat{s}_q$. The bucketing error can be measured as the total deviation of the sequence length to the upper limit of the bucket it belongs to, and the optimization target of sequence bucketing can be defined as:

$$\arg\min_{\{\hat{s}_q\}} \sum_q \sum_k I[\hat{s}_{q-1} < s_k \le \hat{s}_q](\hat{s}_q - s_k). \tag{15}$$

We solve this bucketing problem via a dynamic programming algorithm. We first sort sequences in ascending order of sequence lengths, i.e., $s_1 \le s_2 \le \ldots \le s_K$, and then define $err[k][q]$ as the minimized error of bucketing the first k sequences into q buckets. Then, starting with $err[0][0] = 0$, we can derive the following state transition formula of dynamic programming:

$$err[k][q] = \min_{j \in [0,k-1]} \{err[j][q-1] + \sum_{i=j+1}^{k} (s_k - s_i)\}. \tag{16}$$

Here, $\sum_{i=j+1}^{k}(s_k - s_i)$ denotes the bucketing error of the q^{th} bucket when selecting $\hat{s}_{q-1} = s_j$ as the upper limit of the $(q-1)^{th}$ bucket $\mathcal{B}_{q-1}$. Through this dynamic programming algorithm, we determine the bucket boundaries that minimizes the bucketing error adaptively to the data, and group the sequences $\{\mathcal{S}_k\}$ into Q buckets $\{\mathcal{B}_q = \{\mathcal{S}_{k_q}\}\}$. In practice, we set bucket number Q as 16 by default.

We now re-formulate the optimization problem based on the bucketed sequences. Given the number of available GPUs N, the device memory capacity E, and K sequences $\{\mathcal{S}_k\}$ as well as Q sequence buckets $\{\mathcal{B}_q = \{\mathcal{S}_{k_q}\}\}$, where bucket $\mathcal{B}_q$ has $\hat{b}_q$ sequences and upper length limit $\hat{s}_q$, we keep the definition of SP groups as problem (5), and re-define the sequence assignment matrix $\hat{\boldsymbol{A}} \in \mathbb{N}_{\ge 0}^{Q \times P}$ such that $\hat{A}_{q,p}$ represents the number of the sequences in the q^{th} bucket $\mathcal{B}_q$ assigned to the p^{th} SP group $\mathcal{G}_p$. Then, we can re-formulate the optimization problem as follows:

$$\arg\min_{\boldsymbol{m} \in \{0,1\}^P; \hat{\boldsymbol{A}} \in \mathbb{N}_{\ge 0}^{Q \times P}} \quad C \tag{17}$$

$$\text{s.t.} \quad \text{Time}(\{\hat{s}_q, \hat{A}_{q,p}\}; d_p) \le C, \ \forall p \in [1, P] \tag{18}$$

$$\text{Memory}(\{\hat{s}_q, A_{q,p}\}; d_p) \le E, \ \forall p \in [1, P] \tag{19}$$

$$\sum\nolimits_p d_p \times m_p \le N \tag{20}$$

$$\sum\nolimits_q \hat{A}_{q,p} \le m_p \times K, \ \forall p \in [1, P] \tag{21}$$

$$\sum\nolimits_p \hat{A}_{q,p} = \hat{b}_q, \ \forall q \in [1, Q] \tag{22}$$

where Cond. (22) ensures that all the sequences in bucket B_q are assigned. It is obvious that the re-formulated optimization problem (17) is also a MILP problem. In practice, FlexSP

utilizes SCIP, an advanced MILP solver library, to solve the problem (17). After obtaining the optimal group selection vector $\boldsymbol{m}^*$ and the optimal sequence assignment matrix $\hat{\boldsymbol{A}}^*$, we can derive the optimal parallelism plan according to $\boldsymbol{m}^*$ and dispatch the training sequences across the SP groups according to $\hat{\boldsymbol{A}}^*$. The solving time of problem (17) is typically within 5-15 seconds, which can be overlapped with the training time of one batch (§5).

4.2 Sequence Blaster

When the input batch contains too many sequences, they cannot be processed together due to the limited memory capacity, and therefore the optimization problem (17) will have no feasible solutions due to memory constraint (19). Gradient accumulation is the common technique for such cases, which splits the global data batch into several micro-batches, executes each micro-batch sequentially and accumulates the model gradients for parameter update. For training systems intended for homogeneous sequence lengths, micro-batch chunking is straightforward — we can simply fix the number of sequences in each micro-batch. However, in our scenario where input sequences are associated with heterogeneous lengths, micro-batch chunking is non-trivial. Therefore, FlexSP designs a sequence blaster to blast the sequences into micro-batches for parallelism planner to determine the optimal sequence parallelism strategies.

Given input data batch $\mathcal{B} = \{\mathcal{S}_k\}$ with K sequences, the sequence blaster blasts the sequences into M disjoint micro-batches $\{\mathcal{M}_i\}$, satisfying $\bigcup_{i=1}^{M} \mathcal{M}_i = \mathcal{B}$. In the following, we summarize several propositions based on theoretical analysis and empirical observations, and introduce our designs of sequence blaster motivated by these propositions.

Takeaway #1: In most cases, having fewer micro-batches is likely to be more efficient.

This takeaway can be deduced from the cost estimation in §4.1.2. Either in computation or communication, there is a fixed overhead term denoted as β that exists for each micro-batch execution, so having more micro-batches introduces more additional overhead. Besides, if we have many micro-batches, which implies that each micro-batch only consists of very few tokens, then the workload distributed to each micro-batch may not be sufficient to fully utilize either the computation capacity or the communication bandwidth. Therefore, a smaller number of micro-batch number M usually gives better efficiency.

However, this does not mean that the smallest M always achieves the best performance. Hence, our sequence blaster first calculates the smallest feasible micro-batch number $M_{min} = \left\lceil \frac{Batch\_Total\_Token}{Cluster\_Token\_Capacity} \right\rceil$. Then, it traverses the micro-batch number in range $[M_{min}, M_{min} + M')$ to find the best one, where M' is the number of trails (5 by default).

Takeaway #2: A smaller variance of sequence lengths within a micro-batch is likely to be more efficient.

This takeaway is based on both the theoretical analysis of execution overhead (§4.1.2) and empirical observations derived by solving the optimization problem (17). Specifically, the memory consumption (Eq. (11)) is linear to the sequence length, while the computation time (Eq. (12)) is quadratic to sequence length. Consequently, as the sequence length s_k increases, the computation overhead increases faster than the memory consumption, leading to imbalance between computation and memory cost. For instance, for two sequences $\mathcal{S}_1$, $\mathcal{S}_2$ with length $s_1 = 4K$, $s_2 = 16K$ within one micro-batch, $\mathcal{S}_1$ is assigned to SP group $\mathcal{G}_1$ with $d_p = 8$, while $\mathcal{S}_2$ is assigned to $\mathcal{G}_2$ with $d_p = 32$. Although the memory consumption of $\mathcal{G}_1, \mathcal{G}_2$ is the same, the computation cost of $\mathcal{G}_2$ is larger than that of $\mathcal{G}_1$ due to the quadratic computation volume of long sequence $\mathcal{S}_2$, which requires $\mathcal{G}_1$ to wait for $\mathcal{G}_2$ to finish and causes computation resource wastage. On the other hand, if we try to align the computation time of sequences with diverse lengths, their memory consumption will be distinct, leading to memory under-utilization. To conclude, larger variance of sequence lengths within a micro-batch leads to resource under-utilization of either computation or memory.

Motivated by this, FlexSP sequence blaster first sorts the input sequences according to their lengths, and ensures that sequences with smaller variance of lengths are blasted into one micro-batch.

Takeaway #3: The total token number of each micro-batch should be made as evenly distributed as possible.

This takeaway focuses on striking a balance in memory consumption across micro-batches, which is proportional to the total token number. It is designed to prevent potential out-of-memory (OOM) situations when splitting micro-batches and also to avoid under-utilization of device memory. This guidance also contributes to takeaway #1, as imbalanced token blasting leads to more micro-batches. Therefore, we design a memory-balanced micro-batch chunking algorithm based on dynamic programming, detailed in Appendix A.

4.3 Overall Workflow of FlexSP Solver

We now introduce the overall workflow of FlexSP solver, as illustrated in Alg. 1. Given the data batch $\mathcal{B}$, we first calculate the minimum feasible micro-batch number M_{min} in Line 2 based on cluster memory capacity, as discussed in §4.2, and then traverse micro-batch number M starting from M_{min} in Line 3. For each traversed M, the sequence blaster (§4.2) is invoked to blast the sequences into M micro-batches $\{\mathcal{M}_i\}$, $i \in [1, M]$ (Line 5). Subsequently, for each micro-batch $\mathcal{M}_i$, we first group sequences into Q buckets (Line 7) and then utilize the parallelism planner (Line 8) to optimize the sequence parallelism strategies for the current micro-batch data, which solves the MILP problem as discussed in §4.1. Line 9 gathers the optimal time and strategy of each micro-batch to form the results for the whole data batch $\mathcal{B}$, and Line 11 finds the

Algorithm 1: FlexSP Solver Workflow

Input: Data Batch $\mathcal{B} = \{\mathcal{S}_k\}$ with K sequences, # Buckets Q, # Devices N, Device Memory Capacity E, # Trails M'
Output: Minimized time T^*, Parallelism Plan $\mathcal{P}^*$

1 $T^*, \mathcal{P}^* \leftarrow \infty, None$;
2 $M_{min} \leftarrow$ get_min_microbatch_num($\mathcal{B}, N, E$);
3 **for** M *in* $M_{min}, M_{min}+1, \ldots, M_{min}+M'-1$ *in parallel* **do**
4 $\quad T_{\mathcal{B}}, \mathcal{P}_{\mathcal{B}} \leftarrow 0, []$;
5 $\quad \{\mathcal{M}_i\} \leftarrow$ Sequence_Blaster($\mathcal{B}, M$);
6 $\quad$ **for** $\mathcal{M}$ *in* $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_M$ *in parallel* **do**
7 $\quad\quad \{B_q\} \leftarrow$ Sequence_Bucketing($\mathcal{M}, Q$);
8 $\quad\quad T_{\mathcal{M}}, \mathcal{P}_{\mathcal{M}} \leftarrow$ Parallelism_Planner($\{B_q\}, N, E$);
9 $\quad\quad T_{\mathcal{B}} \leftarrow T_{\mathcal{B}} + T_{\mathcal{M}}$; $\mathcal{P}_{\mathcal{B}}.extend(\mathcal{P}_{\mathcal{M}})$;
10 $\quad$ **if** $T_{\mathcal{B}} < T^*$ **then**
11 $\quad\quad T^*, \mathcal{P}^* \leftarrow T_{\mathcal{B}}, \mathcal{P}_{\mathcal{B}}$;
12 **return** $T^*, \mathcal{P}^*$;

best parallelism plan $\mathcal{P}^*$ with minimum execution time T^* across various tries of micro-batch number.

To improve the efficiency of the solver, FlexSP employs a two-level multi-process solving technique to parallelize the solving process. Specifically, FlexSP explores various micro-batch numbers in parallel (Line 3) with multiple processes, and optimizes the parallelism strategies of each micro-batch in parallel (Line 6) as well. Through this technique, the solving overhead of the FlexSP solver is close to the overhead of solving one round of the MILP problem in §4.1, typically within 5-15 seconds, and is independent of the number of sequences nor the number of micro-batches. Therefore, the solving efficiency of FlexSP solver is guaranteed.

5 Implementation

We build the proposed method as an efficient LLM training system, FlexSP. We implement the FlexSP solver with Python and C++, leveraging the SCIP [5] library for solving the MILP problem. We then develop the FlexSP runtime engine on top of PyTorch for executing the training process based on the strategies optimized by the solver. In our implementation, we use NCCL [1] as the communication backend. As for the attention kernel, we utilize the state-of-the-art flash-attn [9, 10] library's interface for varied-length sequence packing to perform attention computation. As for parallelisms, we implement the Ulysses-style SP similar to DeepSpeed-Ulysses [19] and implement ZeRO with PyTorch FSDP [52]. We now introduce several key points in our implementation of FlexSP for efficient training.

Hot Switching and Group Management. FlexSP implements sequence parallelism in a hot switching manner to deal with the varied parallelism strategies for the distinct input data. Given each micro-batch as well as the corresponding parallelism strategies, FlexSP generates the SP communication groups dynamically and scatters the data into the corresponding group. In order to avoid redundant creation and storage overhead of communication groups, FlexSP maintains a NCCL group pool to manage the complicated SP groups. Specifically, FlexSP generates communication groups on the fly, and new groups are created only when necessary, while existing ones are reused to optimize resource usage. Therefore, in FlexSP, dynamically adjusting the SP groups does not incur any overhead if the groups are cached. The number of communication groups needed for each GPU is up to $\log N$, where N is the number of GPUs.[4] In our evaluation, creating $\log 64 = 6$ communication groups takes under 10 seconds, negligible compared to the overall training time.

Disaggregating Solving and Training. For each training data batch, there are two phases, i.e., the solver deduces the optimal plan (i.e., a combination of parallelism strategies and sequence assignment) and the executor carries out the training. Since the problem solving is on CPUs while the training is on GPUs, we disaggregate the two phases to facilitate overlapping. In particular, on each GPU node (machine), we establish a service of the solver, which takes as input the lengths of one data batch, and runs Alg. 1 to deduce the optimal plan. Subsequently, we manage a distributed storage to gather the plans, and the executor sequentially reads one plan per iteration to train. By doing so, FlexSP solves the problem for multiple data batches concurrently, and the problem solving is also overlapped with the training process.

6 Experiments

6.1 Experimental Setups

Baseline Systems. We compare our system with the state-of-the-art (SOTA) distributed LLM training systems, i.e., Megatron-LM [33] and DeepSpeed [39]. Megatron-LM supports 4D-parallelism, including TP (with Megatron-style SP), PP, DP (ZeRO-1), and CP. DeepSpeed supports DeepSpeed-ZeRO and Ulysses-style SP. Both these systems are tailored for homogeneous sequence lengths and only support training LLMs with a single, static parallelism strategy. Our system, namely FlexSP, is adaptive to the heterogeneous workloads of varied-length sequences, and is capable to generate the optimal sequence parallelism strategies adaptively.

For further evaluation of our adaptive feature, we also introduce a variant of FlexSP as a baseline, FlexSP-BatchAda. Unlike DeepSpeed which employs one static strategy along the whole training process, FlexSP-BatchAda adaptively applies the most efficient homogeneous SP strategy for each data batch, e.g., two SP=32 groups for the first batch and

[4] Because the sizes of SP groups are always powers of 2, we let each GPU to always pair with its neighbors. For example, with $N = 4$ GPUs, there are at most 3 communication groups, i.e., [0,1], [2,3], and [0,1,2,3], with each GPU associated with 2 (= $\log N$) groups.

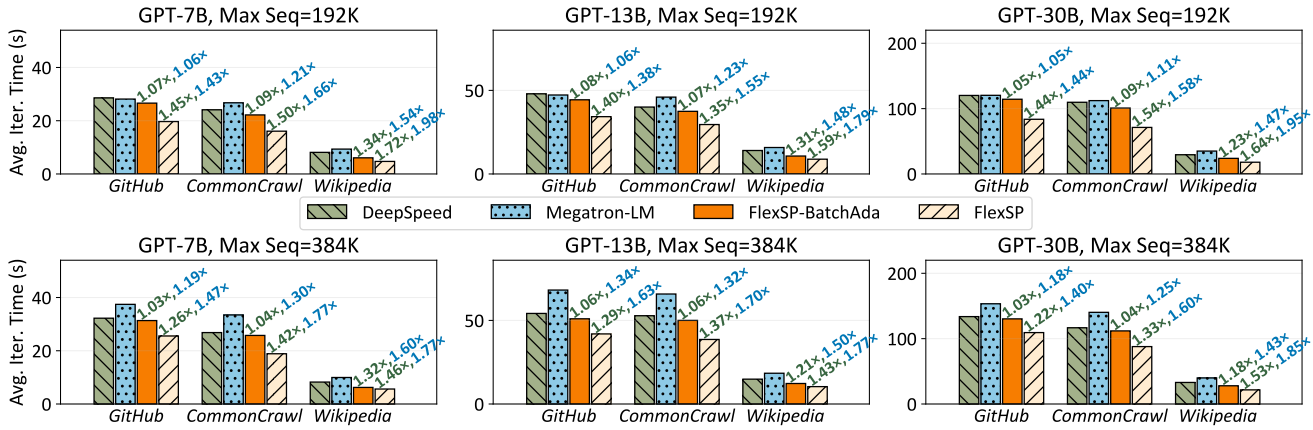


Figure 4. End-to-end evaluation (in seconds per iteration) for specific model sizes and maximum context lengths (Max Seq) across three datasets, shown in each sub-figure. Speedup ratios compared to DeepSpeed (green, left) and Megatron-LM (blue, right) are indicated.

eight SP=8 groups for the second batch. Compared to FlexSP-BatchAda, FlexSP not only allows adaptive strategies across data batches, but also supports heterogeneous SP strategies within each data batch, e.g., mixing and executing one SP=32 group and four SP=8 groups concurrently.

Hardware Environments. We conduct all the experiments on a GPU cluster with 8 nodes, with each node consisting of 8 NVIDIA A100 40GB GPUs equipped with NVLink. All nodes are interconnected by 400Gbps InfiniBand network.

Experimental Workloads. We conduct experiments on GPT-series LLMs of three different sizes, GPT-7B, GPT-13B, GPT-30B. Refer to Appendix B.1 for more details. We choose three different datasets, including *GitHub*, *CommonCrawl*, and *Wikipedia*. Fig. 2 displays the distribution of sequence lengths of these datasets. We also evaluate each system on these datasets under different maximum context length limits, i.e., 384K and 192K. The sequences exceed the maximum context length limit will be eliminated during training.

Protocols. We apply sequence packing for all systems. Specifically, for baseline systems Megatron-LM, DeepSpeed and FlexSP-BatchAda, we use the *Best-fit Packing* [13] as introduced in §2. For FlexSP, the solver will automatically determine the sequence packing. As for the parallelism strategy, we manually tune the most efficient strategy for baseline systems under different workloads, including parallelism degrees of DP, TP, PP, CP, SP. We also apply activation checkpointing strategies for each system to accommodate model training with a context length of 384K. We fix the global batch size of each training step as 512 for all workloads, and record the average iteration time over 40 iterations after 10-iteration's warm-up. Refer to Appendix B.2 for details.

6.2 End-to-End Performance

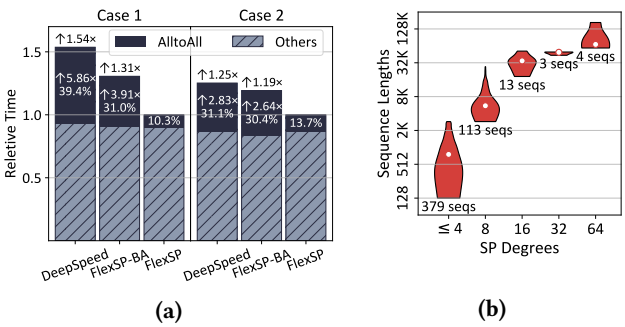
We compare the end-to-end performance of each system in Fig. 4, which shows the average iteration time of each system across different workloads. The results demonstrate that across all the model sizes, datasets, and context lengths, FlexSP consistently outperforms all baseline systems, achieving a maximum speedup of 1.72× compared to DeepSpeed and 1.98× compared to Megatron-LM.

We first analyze the performance gain of FlexSP compared to SOTA systems. The advantages of FlexSP primarily arise from the communication gains achieved by its flexible sequence parallelism strategy. As mentioned in §3, the parallelism group needs to be large enough to shard excessively long sequences to fit the model into device memory. For instance, under 384K maximum context length, DeepSpeed requires SP=64 while Megatron requires TP=16, CP=4 or TP=8, CP=8. Such large parallelism groups must communicate with slow inter-node network bandwidth, thus leading to inefficient communication. SOTA systems maintain a homogeneous and static parallelism strategy along the training process, forcing all sequences in the dataset to utilize the large groups with slow inter-node bandwidth, which is inefficient for shorter sequences. On the contrary, FlexSP allows shorter sequences to enjoy the higher communication efficiency within smaller parallelism groups, while maintaining larger groups for long sequences to satisfy the memory constraint. For instance, FlexSP may assign a sequence with 100K into a group with SP=32 to avoid OOM errors, while scattering sequences with 16K into a group with SP=8 to enjoy the fast intra-node connection. Such flexible strategy effectively reduces the communication overhead and contributes to the system efficiency of FlexSP.

Furthermore, the strength of FlexSP is correlated with the long-tail distribution of sequence lengths — a more pronounced long-tail leads to greater communication benefits,

Table 3. Details of heterogeneous SP groups employed in each micro-batch of each case. Each $d \times m$ indicates we form m SP=d groups, and each $\langle \cdots \rangle \times x$ indicates the set of heterogeneous SP groups is employed for x micro-batches ($\times 1$ is omitted).

	Case 1	Case 2
DeepSpeed	$\langle 64 \rangle \times 5$	$\langle 64 \rangle \times 7$
FlexSP-BatchAda	$\langle 16 \times 4 \rangle \times 5$	$\langle 32 \times 2 \rangle \times 7$
FlexSP	$\langle 32, 16, 8 \times 2 \rangle$ $\langle 8 \times 8 \rangle \times 2$ $\langle 8 \times 7, 4 \times 2 \rangle$ $\langle 1 \times 64 \rangle$	$\langle 64 \rangle$ $\langle 32, 16 \times 2 \rangle$ $\langle 16 \times 3, 8 \times 2 \rangle$ $\langle 8 \times 8 \rangle \times 2$ $\langle 1 \times 64 \rangle$



(a) **(b)**

Figure 5. (5a) Breakdown of end-to-end time (All-to-All+Others) in case study (BA is short for BatchAda). (5b) Distribution of sequence lengths assigned to different SP degrees in Case 2, visualized as a violin plot. The white circle indicates the median.

resulting in a more significant speedup. As shown in Fig. 2, the *Wikipedia* dataset has the greatest skewness in three datasets. Over 96% of the sequences in *Wikipedia* are below 8K, considerably greater than those in *GitHub* and *CommonCrawl*, and the proportion of sequences exceeding 32K is much smaller than those in the other datasets. Compared to SOTA systems, such great skewness benefits FlexSP to achieve speedup of up to 1.98× on *Wikipedia*, while the speedups on *CommonCrawl* and *GitHub* are slightly lower, up to 1.77× and 1.63×, respectively.

Then, we analyze the performance of FlexSP-BatchAda, which employs homogeneous strategy within each data batch but allows adaptive strategies across data batches. It also gains benefits from communication and achieves speedup ratio up to 1.34× and 1.60× compared to DeepSpeed and Megatron-LM, respectively. However, due to its homogeneous strategy within each batch, its performance gain on *GitHub* and *CommonCrawl* is relatively low, as these datasets possess long sequences in many data batches, forcing these batches to use large and inefficient parallelism groups. In comparison, FlexSP allows heterogeneous and adaptive strategies at a nuanced granularity, both among and within data batches, further increasing the potential for reducing communication overhead and achieving acceleration up to 1.42× compared to FlexSP-BatchAda.

6.3 Case Study

To analyze the performance gains of FlexSP more clearly, we conduct an in-depth case study on two iterations of GPT-7B on *CommonCrawl* with a maximum context length of 384K.

We present the parallelism strategies, i.e., details of the employed SP groups, in Tab 3. We also break down the end-to-end time and highlight the portion of All-to-All communication in Fig 5a. It can be seen that the major difference lies in the All-to-All communication overhead, which is the source of FlexSP's performance gain. For DeepSpeed, its All-to-All communication accounts for up to 40% of the total runtime, which is due to the large SP group (SP=64) and the limited inter-node bandwidth across 8 nodes. FlexSP-BatchAda adapts the SP strategies for batches (four SP=16 groups for Case 1 and two SP=32 groups for Case 2), and reduces communication cost compared to DeepSpeed, especially in Case 1 where SP=16 cuts down the communication to 31%. FlexSP further optimizes communication through adaptive strategies at a finer granularity, leveraging smaller SP groups (e.g., SP=1, 4, 8) to process shorter sequences, which significantly reduces communication over low-bandwidth inter-node connections, cuts down the All-to-All time to around 10%, and achieves a reduction of up to 5.86× in All-to-All time and a 1.54× speedup in overall end-to-end time.

To further explore FlexSP's flexible strategy, we present the distribution of sequence lengths assigned to different SP degrees in Case 2, as shown in Fig. 5b. In FlexSP, sequences of diverse lengths are assigned to appropriate SP degree groups, with shorter sequences showing a clear preference to lower SP degrees so that the All-to-All communication cost can be minimized. Meanwhile, due to the long-tail property of the datasets, (relatively) short sequences may be routed to SP groups with (relatively) higher parallel degree, striking a good balance across all SP groups. This highlights the effectiveness of FlexSP solver in optimizing the flexible parallelism strategies for sequences with varied lengths.

6.4 Scalability Study

To evaluate the scalability of each system, we conduct experiments on *CommonCrawl*, varying both cluster size and maximum context length. The results, measured as token throughput per GPU, are presented in Fig. 6.

Scalability w.r.t. # GPUs. We begin by evaluating performance across GPU clusters with 16, 32, and 64 GPUs, with a maximum context length of 128K. The results indicate that FlexSP consistently outperforms other systems, achieving a maximum speedup of 1.48× compared to DeepSpeed.

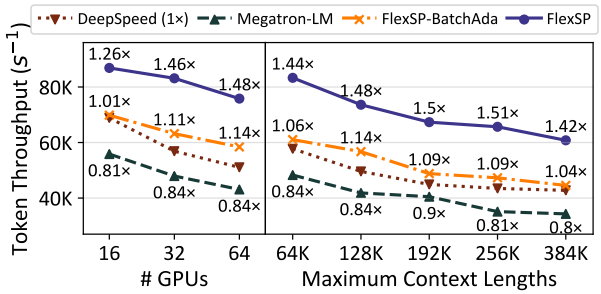


Figure 6. Scalability study measured as token throughput per GPU. The speedup rates are measured w.r.t. DeepSpeed.

Table 4. Token estimation bias of bucketing methods.

Token Error	*Github*	*CommonCrawl*	*Wikipedia*
DP Bucketing	**0.7%**	**0.5%**	**2.3%**
Naïve Bucketing	13.4%	8.8%	22.1%

Furthermore, we find that as the cluster size increases, the reduced inter-node bandwidth brings negative impact on training throughput. For instance, when scaling from 16 to 32 GPUs, DeepSpeed and Megatron-LM only achieve sublinear speedup of 1.65× and 1.71×, respectively. This is because the bandwidths of 32 and 64 GPUs on our cluster are lower than that of 16 GPUs, which leads to poor scalability of SOTA systems. However, FlexSP is much more robust to such bandwidth decrease, achieving 1.91× when scaling from 16 to 32 GPUs and 1.82× from 32 to 64 GPUs. FlexSP's sound scalability attributes to the adaptive strategies and the utilization of the high bandwidth of intra-node NVLink.

Scalability w.r.t. maximum context length. We extend the evaluation on 64 GPUs with maximum context lengths ranging from 64K to 384K. The token throughput of all systems tends to decrease due to the increased computational FLOPs associated with longer sequences. FlexSP consistently maintains its optimal performance under different context length limit, achieving a speedup ratio between 1.42× and 1.51×. Furthermore, we find the speedup ratio for 64K and 384K is slightly lower than that for 256K, which is reasonable. For a shorter context length limit, such as 64K, the long-tail property of the dataset is weakened, resulting in fewer opportunities for adaptive optimization. On the other hand, for a longer context length, like 384K, the computation overhead of extremely long sequences consumes a significant amount of time, which also reduces the speedup.

6.5 Ablation Study

To evaluate the efficacy of key components within the FlexSP solver, i.e., the dynamic programming (DP) sequence bucketing in parallelism planner (§4.1), and the sequence sorting

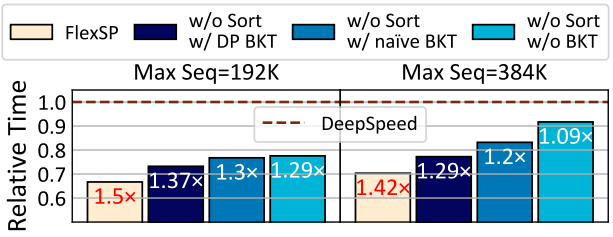


Figure 7. Ablation studies. FlexSP adopts sequence sorting (Sort) in sequence blaster and DP bucketing (BKT) algorithm.

mechanism in sequence blaster (§4.2), we compare the performance of complete version of FlexSP against various ablated versions on *CommonCrawl*, as shown in Fig. 7. Sequence sorting in sequence blaster helps reduce sequence length variance within each micro-batch. Disabling this mechanism negatively impacts overall performance. Additionally, replacing the DP sequence bucketing with a naïve even-sized bucketing introduces more biases into the bucketing estimation, leading to worse performance. Finally, removing the bucketing mechanism entirely increases the complexity of the MILP problem, causing the solver to fail in producing a satisfactory solution within limited time.

We also evaluate the token estimation bias of the naïve bucketing and our dynamic-programming-based (DP) optimal sequence bucketing in parallelism planner (§4.1). We show the maximum token error ratio, i.e., error token number divided by total token number, on different datasets in Tab. 4. We find that our optimal bucketing algorithm effectively reduces the estimation error to lower than 2.3%, while naïve bucketing introduces error up to 22%.

6.6 Complexity and Scalability of FlexSP Solver

We demonstrate that the MILP problem solving in FlexSP solver scales well and can be fully overlapped by the training. Due to the sequence bucketing, the MILP solving time (Eq. (17)) is mainly affected by the number of GPUs (N) and the number of buckets. Since the number of buckets is small, we focus on the scalability against N. When there are more GPUs, the problem solving for each iteration takes longer, while the training time maintains at a similar level (as it is common to scale the batch size proportionally to N). However, recall that FlexSP disaggregates the problem solving on CPUs and the training on GPUs (§5). In modern clusters, using more GPUs also indicates more CPUs are available to solve the MILP for different batches concurrently, so the solving time can be amortized and we can still overlap the problem solving well. To elaborate, we conduct a simulation experiment in Fig. 8 to assess the time cost with GPU counts varying from 64 to 1024. We show the estimated training time, solving time and amortized solving time below (solving time divided by the number of GPU nodes, i.e., $N/8$) per iteration in seconds. We find that the estimated training

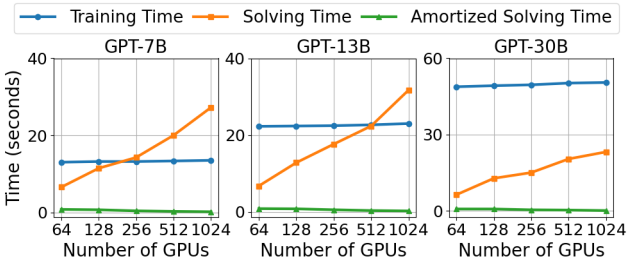


Figure 8. Estimated per-iteration training time, solving time and amortized solving time (solving time divided by the number of GPU nodes, i.e., $N/8$) of FlexSP Solver.

time does remain similar. Although the per-iteration solving time may exceed training time when $N \geq 256$, the amortized solving time is much lower, and is fully overlappable.

More Experimental Results. Due to the space limitation, we put more results and analysis in the Appendix, including more details of our experimental setups in Appendix B, estimation accuracy of cost models in Appendix C, performance analysis of SOTA systems in Appendix D, discussion about integrating context parallelism in Appendix E. Please refer to our Appendix for more details.

7 Related Work

Parallel Training Optimization. During hybrid parallel training, the optimization of parallel strategies is crucial. Many advanced parallelism optimization techniques [21, 30, 44, 46, 51, 53] are developed to automate the tuning of parallel strategies. However, these works are designed mainly for homogeneous training corpora, while FlexSP focuses on flexible strategies for data with heterogeneous lengths.

Long Context Training. Efforts to optimize long context training have led to various elaborate parallel strategies, such as ring attention for LLMs [6, 24, 27, 29], though they often suffer from communication overhead and inefficiencies with severe communication cost. These methods are orthogonal, and can be integrated into FlexSP seamlessly, detailed in Appendix E.Other works aim to support long context training by extending attention mechanism [4, 8] and optimizing position embedding [14, 55], which are also orthogonal.

Heterogeneous Cluster Training. Training efficiency on heterogeneous GPU clusters is the main topic of these works [20, 25, 34, 41], focusing on the heterogeneity of hardware. In contrast, FlexSP emphasizes flexible parallelism to address the workload heterogeneity of varied-length data.

8 Conclusion

In this paper, we propose an efficient system to accelerate LLM training via flexible and adaptive sequence parallelism. Specifically, FlexSP addresses the workload heterogeneity and optimizes the parallelism strategies for varied-length training corpora in real-world datasets. Experiments demonstrate that FlexSP outperforms SOTA systems by up to 1.98×.

Acknowledgments

This work is supported by National Science and Technology Major Project (2022ZD0116315), National Natural Science Foundation of China (U23B2048, 62402011), Beijing Municipal Science and Technology Project (Z231100010323002), China National Postdoctoral Program for Innovative Talents (BX20230012), China Postdoctoral Science Foundation (2024M750103), Beijing Natural Science Foundation (4244080), research grant No. IPT-2024JK29, ByteDance-PKU joint program, and High-performance Computing Platform of Peking University. Fangcheng Fu and Bin Cui are the corresponding authors.

References

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

[2] 2025. FlexSP Appendix. https://github.com/AFDWang/ASPLOS25-FlexSP-Supplemental-Material/blob/main/Appendix.pdf.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[4] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *CoRR* abs/2004.05150 (2020). arXiv:2004.05150 https://arxiv.org/abs/2004.05150

[5] Suresh Bolusani, Mathieu Besançon, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Christoph Graczyk, Katrin Halbig, Ivo Hedtke, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Julian Manns, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Dieter Weninger, and Lixing Xu. 2024. *The SCIP Optimization Suite 9.0.* Technical Report. Optimization Online. https://optimization-online.org/2024/02/the-scip-optimization-suite-9-0/

[6] 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. *CoRR* abs/2311.09431 (2023). https://doi.org/10.48550/ARXIV.2311.09431 arXiv:2311.09431

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.

[8] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive Language Models beyond a Fixed-Length Context. In *ACL*. 2978–2988.

[9] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. *CoRR* abs/2307.08691 (2023). https://doi.org/10.48550/ARXIV.2307.08691 arXiv:2307.08691

[10] 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 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/67d57c32e20fd0a7a302cb81d36e40d5-Abstract-Conference.html

[11] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, et al. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. arXiv:2405.04434 [cs.CL] https://arxiv.org/abs/2405.04434

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. 4171–4186.

[13] Hantian Ding, Zijian Wang, Giovanni Paolini, Varun Kumar, Anoop Deoras, Dan Roth, and Stefano Soatto. 2024. Fewer Truncations Improve Language Modeling. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. https://openreview.net/forum?id=kRxCDDFNpp

[14] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. 2024. LongRoPE: Extending LLM Context Window Beyond 2 Million Tokens. arXiv:2402.13753 [cs.CL] https://arxiv.org/abs/2402.13753

[15] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[16] Lei Guan, Dong-Sheng Li, Jiye Liang, Wen-Jian Wang, Ke-shi Ge, and Xicheng Lu. 2024. Advances of Pipeline Model Parallelism for Deep Learning Training: An Overview. *J. Comput. Sci. Technol.* 39, 3 (2024), 567–584. https://doi.org/10.1007/S11390-024-3872-3

[17] Roger W. Hockney. 1994. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Comput.* 20, 3 (1994), 389–398. https://doi.org/10.1016/S0167-8191(06)80021-9

[18] Yanping Huang, Youlong Cheng, Ankur Bapna, et al. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*.

[19] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. 2023. DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models. *CoRR* abs/2309.14509 (2023). https://doi.org/10.48550/ARXIV.2309.14509 arXiv:2309.14509

[20] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, et al. 2022. Whale: Efficient giant model training over heterogeneous {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 673–688.

[21] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *MLSys*.

[22] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing Activation Recomputation in Large Transformer Models. *CoRR* abs/2205.05198 (2022). https://doi.org/10.48550/ARXIV.2205.05198 arXiv:2205.05198

[23] Mario Michael Krell, Matej Kosec, Sergio P Perez, and Andrew Fitzgibbon. 2021. Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance. *arXiv preprint arXiv:2107.02027* (2021).

[24] 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. In *First Conference on Language Modeling*.

[25] Dacheng Li, Hongyi Wang, Eric P. Xing, and Hao Zhang. 2022. AMP: Automatically Finding Model Parallel Strategies with Heterogeneity 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*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/2b4bfa1cebe78d125fefd7ea6ffcfc6d-Abstract-Conference.html

[26] Longhai Li, Lei Duan, Junchen Wang, Chengxin He, Zihao Chen, Guicai Xie, Song Deng, and Zhaohang Luo. 2023. Memory-Enhanced Transformer for Representation Learning on Temporal Heterogeneous Graphs. *Data Sci. Eng.* 8, 2 (2023), 98–111.

[27] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. 2023. Sequence Parallelism: Long Sequence Training from System Perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 2391–2404. https://doi.org/10.18653/V1/2023.ACL-LONG.134

[28] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (2020), 3005–3018. https://doi.org/10.14778/3415478.3415530

[29] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context. *CoRR* abs/2310.01889 (2023). https://doi.org/10.48550/ARXIV.2310.01889 arXiv:2310.01889

[30] 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. *Proc. VLDB Endow.* 16, 3 (2022), 470–479. https://doi.org/10.14778/3570690.3570697

[31] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *SOSP*. 1–15.

[32] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*. PMLR, 7937–7947.

[33] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, et al. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *SC*. ACM, 58:1–58:15.

[34] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young-ri Choi. 2020. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 307–321. https://www.usenix.org/conference/atc20/presentation/park

[35] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[36] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[37] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *JMLR* (2020).

[38] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *SC*. IEEE/ACM.

[39] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *SIGKDD*. 3505–3506.

[40] Yunfan Shao, Zhichao Geng, Yitao Liu, Junqi Dai, Hang Yan, Fei Yang, Li Zhe, Hujun Bao, and Xipeng Qiu. 2024. CPT: a pre-trained unbalanced transformer for both Chinese language understanding and generation. *Sci. China Inf. Sci.* 67, 5 (2024). https://doi.org/10.1007/S11432-021-3536-5

[41] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2020. AccPar: Tensor Partitioning for Heterogeneous Deep Learning Accelerators. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 342–355. https://doi.org/10.1109/HPCA47549.2020.00036

[42] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023). https://doi.org/10.48550/ARXIV.2302.13971 arXiv:2302.13971

[43] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). https://doi.org/10.48550/ARXIV.2307.09288 arXiv:2307.09288

[44] Colin Unger, Zhihao Jia, Wei Wu, et al. 2022. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *OSDI*. 267–284.

[45] 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 *NeurIPS*. 5998–6008.

[46] 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* (2024).

[47] Aiyuan Yang, Bin Xiao, Bingning Wang, Borong Zhang, Ce Bian, Chao Yin, Chenxu Lv, Da Pan, Dian Wang, Dong Yan, Fan Yang, Fei Deng, Feng Wang, Feng Liu, Guangwei Ai, Guosheng Dong, Haizhou Zhao, Hang Xu, Haoze Sun, Hongda Zhang, Hui Liu, Jiaming Ji, Jian Xie, JunTao Dai, Kun Fang, Lei Su, Liang Song, Lifeng Liu, Liyun Ru, Luyao Ma, Mang Wang, Mickel Liu, MingAn Lin, Nuolan Nie, Peidong Guo, Ruiyang Sun, Tao Zhang, Tianpeng Li, Tianyu Li, Wei Cheng, Weipeng Chen, Xiangrong Zeng, Xiaochuan Wang, Xiaoxi Chen, Xin Men, Xin Yu, Xuehai Pan, Yanjun Shen, Yiding Wang, Yiyu Li, Youxin Jiang, Yuchen Gao, Yupeng Zhang, Zenan Zhou, and Zhiying Wu. 2023. Baichuan 2: Open Large-scale Language Models. arXiv:2309.10305 [cs.CL] https://arxiv.org/abs/2309.10305

[48] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big Bird: Transformers for Longer Sequences. In *Annual Conference on Neural Information Processing Systems 2020 (NeurIPS 2020)*.

[49] Huangzhao Zhang, Kechi Zhang, Zhuo Li, Jia Li, Jia Li, Yongmin Li, Yunfei Zhao, Yuqi Zhu, Fang Liu, Ge Li, et al. 2024. Deep learning for code generation: a survey. *Sci. China Inf. Sci.* 67, 9 (2024), 191101. https://doi.org/10.1007/s11432-023-3956-3

[50] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. *CoRR* abs/2205.01068 (2022). https://doi.org/10.48550/ARXIV.2205.01068 arXiv:2205.01068

[51] Zhen-Xing Zhang, Yuan-Bo Wen, Han-Qi Lv, Chang Liu, Rui Zhang, Xia-Qing Li, Chao Wang, Zi-Dong Du, Qi Guo, Ling Li, Xue-Hai Zhou, and Yun-Ji Chen. 2024. AI computing systems for LLMs training: a review. *J. Comput. Sci. Technol.* (2024). https://doi.org/10.1007/s11390-024-4178-1

[52] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle 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. https://doi.org/10.14778/3611540.3611569

[53] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K.

Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

[54] Xuanhe Zhou, Zhaoyan Sun, and Guoliang Li. 2024. DB-GPT: Large Language Model Meets Database. *Data Sci. Eng.* 9, 1 (2024), 102–111. https://doi.org/10.1007/S41019-023-00235-6

[55] Dawei Zhu, Liang Wang, Nan Yang, Yifan Song, Wenhao Wu, Furu Wei, and Sujian Li. 2024. LongEmbed: Extending Embedding Models for Long Context Retrieval. arXiv:2404.12096 [cs.CL] https://arxiv.org/abs/2404.12096

A Momery-balanced Micro-batch Chunking in Sequence Blaster

We illustrate the memory-balanced micro-batch chunking algorithm in Sequence Blaster based on dynamic programming. Specifically, given a batch of sequences $\mathcal{B} = \{\mathcal{S}_k\}$ that has already been sorted according to takeaway #2, we split them into consecutive M micro-batches, and micro-batch $\mathcal{M}_i$ contains all sequences k satisfying $j_{i-1} \le k < j_i$, where j_i is the ending indices for $\mathcal{M}_i$ ($j_0 = 0$). To balance the token amount of each micro-batch, we aim to minimize the maximum total token number of each micro-batch as follows:

$$\arg\min_{\{j_i\}} \max_{i\in[1,M]} \{ \sum_{k\in[j_{i-1},j_i)} s_k \}. \tag{23}$$

Again, we solve the problem via dynamic programming. Denote $DP[k][i]$ as the optimal value when blasting the first k sequences into i micro-batches. Starting with $DP[0][0] = 0$, we can solve the problem via the following state transition formula:

$$DP[k][i] = \min_{j\in[i-1,k-1]} \{\max\{DP[j][i-1], \sum_{l\in[j+1,k]} s_l\}\}, \tag{24}$$

where $\sum_{l\in[j+1,k]} s_l$ represents the total token number of the i^{th} micro-batch when splitting micro-batch at the $\mathcal{S}_j$. $DP[K][M]$ denotes the optimal solution, and the optimized values of j_i splits the global batch data into M micro-batches with balanced memory consumption.

B Details of Experimental Setups

B.1 Model Configuration

Tab. 5 presents the specific configurations of the GPT-7B, 13B and 30B models used in our experiments. The parameter number of each model in Tab. 5 is under maximum context length of 384K, where the positional embedding contains 1-2 billion parameters.

Table 5. Model configuration (384K max context length).

Model	# Layers	# Param	Hidden Dim
GPT-7B	32	7.85B	4096
GPT-13B	40	14.03B	5120
GPT-30B	60	32.72B	6656

B.2 Protocols

For fair comparison, we manually tune the most efficient parallelism strategies for all baseline systems under different workloads. For DeepSpeed, the optimal strategy is usually among SP=64 or SP=32 with ZeRO-3, and for Megatron-LM, the optimal strategy is usually among TP=8, CP=8, or TP=16, CP=4, or TP=8, CP=4, DP=2 with ZeRO-1.

We also apply activation checkpointing strategies for each system to make sure all systems can fit the models without out-of-memory issues. For GPT-7B, activation checkpointing is unnecessary to support a 384K context length on 64 GPUs. For GPT-13B, we only checkpoint MLP layers, while for GPT-30B, almost all layers are checkpointed to support 384K context length.

C Estimation Accuracy of Cost Models

We evaluate the accuracy of the cost estimator (§4.1.2) utilized in FlexSP across diverse configurations (as shown in Tab. 1), including sequence parallelism degree, batch size, and sequence length. Fig. 9 compares the deviation of the estimated cost and the empirical execution time. As can be seen, our overhead estimator adeptly approximates the execution overhead, with discrepancies consistently remaining below 5%. The accurate estimations rendered by the estimator facilitates performance of our system.

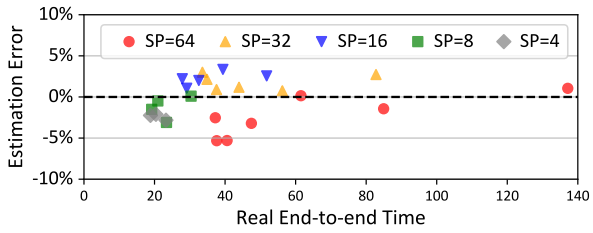


Figure 9. Estimation accuracy.

D Performance Analysis of SOTA Systems

Here we analyze the performance of SOTA systems, i.e., DeepSpeed and Megatron-LM. As shown in Fig. 4, in most cases, DeepSpeed has similar or better performance than Megatron-LM. This is attributed to the different mechanism of CP in Megatron-LM and SP in DeepSpeed, as well as the skewness of datasets. CP usually has much more communication volumn than the All-to-All in SP. Although CP leverages the overlap between attention computation and KV transmission to hide the communication overhead, in scenarios with limited inter-node bandwidth and a majority of short sequences in datasets, the attention computation often fails to hide the communication. Therefore, Megatron-LM's performance is usually constrained by its higher communication volume compared to DeepSpeed and FlexSP.

E Integrating Context Parallelism

Context parallelism [6, 24, 27, 29] is also an important technique for long context training, as illustrated in §2.1.3. As a different paradigm to scatter sequences across devices, CP is orthogonal to our work, and can be integrated into FlexSP. Specifically, similar to FlexSP utilizing ZeRO and flexible SP, we can employ TP, ZeRO and CP, fix the parallelism degree of TP, and employ the flexible sequence parallelism strategy of FlexSP to achieve flexible CP, which adaptively adjusts the CP group size according to the varied-length sequences. We'll integrate CP into our system as the future work.