DSP: DYNAMIC SEQUENCE PARALLELISM FOR MULTI-DIMENSIONAL TRANSFORMERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Scaling multi-dimensional transformers to long sequences is indispensable across various domains. However, the challenges of large memory requirements and slow speeds of such sequences necessitate sequence parallelism. All existing approaches fall under the category of embedded sequence parallelism, which are limited to shard along a single sequence dimension, thereby introducing significant communication overhead. However, the nature of multi-dimensional transformers involves independent calculations across multiple sequence dimensions. To this end, we propose Dynamic Sequence Parallelism (DSP) as a novel abstraction of sequence parallelism. DSP dynamically switches the parallel dimension among all sequences according to the computation stage with efficient resharding strategy. DSP offers significant reductions in communication costs, adaptability across modules, and ease of implementation with minimal constraints. Experimental evaluations demonstrate DSP's superiority over state-of-the-art embedded sequence parallelism methods by remarkable throughput improvements ranging from 32.2% to 10x, with at least 50% communication volume reduction.

1 INTRODUCTION

Efficiently scaling multi-dimensional transformers to accommodate long sequences is necessary across diverse domains, including video generation (Singer et al., 2022; Blattmann et al., 2023; Ma et al., 2024), image generation (Ramesh et al., 2021; Rombach et al., 2022; Liu et al., 2024), protein structure prediction (Jumper et al., 2021), spatial-temporal information processing (Cong et al., 2021), and beyond. The long length of sequences in these applications entails substantial activation memory costs and a notable slowdown in processing speeds, underscoring the need for employing sequence parallelism.

Current methods for sequence parallelism, such as Megatron-SP (Korthikanti et al., 2022), Megatron-LM (Shoeybi et al., 2019), DeepSpeed-Ulysses (Jacobs et al., 2023), and Ring-Attention (Li et al., 2021; Liu et al., 2023a) are all embedded sequence parallelism methods. As shown in Figure 1, these embedded methods shard along a single sequence dimension, which are tailored to the task-specific pattern of modules and introduce intricate communication and complex code modification. However, multi-dimensional neural networks often operate independent computations across multiple sequence dimensions. For instance, for video generation models like OpenSora (Zangwei Zheng, 2024) and Latte (Ma et al., 2024), Spatial-Temporal Attention (Yan et al., 2021) is favored over full Self-Attention (Vaswani et al., 2017), facilitating separate attention computations for temporal and spatial dimensions.

Therefore, there exists an explorable space for a general sequence parallelism abstraction. To adapt to the flexible patterns of multi-dimensional transformers, we introduce Dynamic Sequence Parallelism (DSP) as a novel abstraction of sequence parallelism, characterized by its elegance, high effectiveness, and excellent compatibility with popular transformers. Unlike embedded sequence parallelism, DSP dynamically switches the parallel dimension of sequences during the computation stage with an efficient resharding strategy, a process completely decoupled from the computation module.

DSP offers several advantages over embedded sequence parallelism: 1) *Efficient Communication*: DSP incurs significantly lower communication costs due to its simplified communication patterns and reduced frequency of exchanges. 2) *Adaptability*: DSP seamlessly adapts to most modules without necessitating specific modifications and imposes few limitations on its usage. 3) *Ease of Use*: DSP is

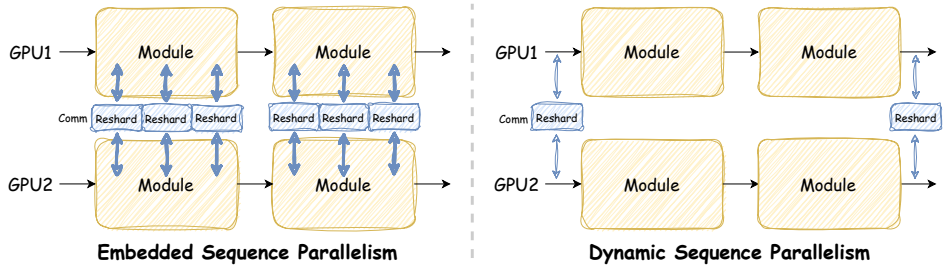

Figure 1: Comparison of Embedded and Dynamic Sequence Parallelism. Reshard means the communication to change sequence parallel layout. The blue arrow represents communication. The number and width of the arrows indicate the volume and frequency of communication, respectively.

remarkably easy to implement, and also provides a simple API shown in Appendix A.4 for users to enable it effortlessly.

Our experiments yield promising results, showcasing DSP's superiority over state-of-the-art embedded sequence parallelism methods. It achieves an end-to-end throughput improvement ranging from 32.2% to 10x and reduces communication volume by at least 75%.

We summarize our contributions as follows:

- We introduce DSP as a novel abstraction of sequence parallelism aimed at effectively scaling multi-dimensional transformers. DSP dynamically switches the parallel dimension of sequences during the computation stage, offering high effectiveness, elegant formalism, and excellent compatibility.
- By significantly reducing communication volume and frequency, DSP improves end-to-end throughput by 32.2% to 10x and reduces communication volume by at least 50% compared to state-of-the-art methods.
- DSP seamlessly integrates with various modifications without requiring specific modifications and imposes few limitations. Its ease of use is highlighted by the minimal code changes needed to incorporate it into existing frameworks with our high-level API.

2 Background and Related Work

Table 1: Meanings of the symbols that are used in this paper.

B	The number of batch sizes	N	The number of GPUs
C	The size of hidden states	n	The n-th GPU
S_i	The i-th sequence dimension	$\mathbf{X}$	The a multi-dimensional sequence tensor
s_i	The status that sequence is sharded from dimension S_i	$\mathbf{X}_{p,n}$	The partition of X assigned to GPU n for sequence parallel
$\hat{s}$	The status that sequence is not sharded		
M	The volume of a sequence tensor		

2.1 Background

Transformer Architecture. Transformer (Vaswani et al., 2017) is a type of neural network architecture that has become highly influential in natural language processing (Devlin et al., 2018; Brown et al., 2020; Reid et al., 2024) and other domains (Dosovitskiy et al., 2020; Jumper et al., 2021; Peebles & Xie, 2023). The Transformer is composed of a stack of layers, each consisting of a multi-head attention (MHA) and a position-wise feed-forward network (FFN). Specifically, the MHA comprises H independently parameterized attention heads, formulated as:

$$\mathrm{MHA}(x) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_H)\mathbf{W}^O, \quad \mathrm{head}_i = \mathrm{Att}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i), \tag{1}$$

Figure 2: Illustration of Multi-Dimensional (2D) Transformer.

$$\mathrm{Att}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}, \quad x_{\mathrm{MHA}} = \mathrm{LayerNorm}(x + \mathrm{MHA}(x)), \tag{2}$$

where $\mathrm{Att}(\cdot)$ denotes the scaled dot-product attention, $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ are query, key, value projections, and LayerNorm is the layer normalization. The output x_{MHA} is fed into the FFN, which consists of two linear transformations with a ReLU activation in between, computed as:

$$\mathrm{FFN}(x) = \max(0, x\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2, \quad x_{\mathrm{out}} = \mathrm{LayerNorm}(x_{\mathrm{MHA}} + \mathrm{FFN}(x_{\mathrm{MHA}})), \tag{3}$$

where $\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2$ are the parameters of the FFN.

Multi-Dimensional Transformer. Multi-dimensional transformers (Ho et al., 2019; Yang et al., 2022) extend the self-attention mechanism of standard transformers to operate over multiple dimensions beyond just one sequence. An example of 2D-Transformer is shown in Figure 2. Let the input multi-dimensional sequence be denoted as $\mathbf{X} \in \mathbb{R}^{[B, S_1, S_2, \ldots, S_K, C]}$, where B is the batch size, $S_1, S_2, \ldots, S_K$ are the sequence lengths along K different sequence dimensions, and C is the hidden size. Multi-dimensional transformer can be formatted as:

$$\mathbf{X}_{\mathrm{reshape}} = \mathrm{Reshape}(\mathbf{X}, [B \times \prod_{j \neq i} S_j, S_i, C]). \tag{4}$$

The transformer block operation is then applied along the i-th sequence dimension of $\mathbf{X}_{\mathrm{reshape}}$.

$$\mathbf{X}_{\mathrm{out}} = \mathrm{transformer_block}(\mathbf{X}_{\mathrm{reshape}}). \tag{5}$$

After applying the transformer block operation along all N dimensions, the final output tensor $\mathbf{X}_{\mathrm{out}}$ has the same shape as the input tensor $\mathbf{X}$. Multi-dimensional Transformer is widely used for applications with multi-dimensional inputs including video data (Xu et al., 2020; He et al., 2021; Geng et al., 2022; Ma et al., 2024), 3D data (Zheng et al., 2021; Chen et al., 2023), protein structure prediction (Jumper et al., 2021; Mirdita et al., 2022), time-series data (Pan et al., 2022; Huang et al., 2022; Deihim et al., 2023) and beyond.

2.2 Related Work

In this section, we discuss the four main parallelism techniques employed in deep learning: data parallelism, tensor parallelism, pipeline parallelism, and sequence parallelism.

Data parallelism (Hillis & Steele Jr, 1986; Li et al., 2020) is one of the most widely adopted parallelism techniques. The input data is partitioned across devices, each processing a subset. Model parameters are replicated, and gradients are summed. ZeRO (Rajbhandari et al., 2019; 2021) optimizes memory by partitioning parameters, states, and gradients across devices, enabling the training of larger models. Tensor parallelism (Shazeer et al., 2018; Shoeybi et al., 2019), or model parallelism, partitions model parameters across devices. Different model parts are assigned to different devices. Pipeline parallelism (Huang et al., 2019; Narayanan et al., 2019; Li & Hoefler, 2021; Liu et al., 2023b) partitions the model into stages executed in parallel across devices. Activations are passed between devices in a pipeline style.

Unlike parameter parallelism discussed earlier, sequence parallelism is a technique specifically designed for distributing long sequences and activation across multiple devices. Here are three main methods of sequence parallelism:

Ring Attention (Li et al., 2021) employs an innovative approach to partitioning the sequence dimension using a ring-style peer-to-peer (P2P) communication pattern to transfer keys and values across GPUs. (Liu et al., 2023a) enhance this method with an online softmax mechanism, allowing for the computation of attention scores without retaining the full sequence length. However, Ring Attention's reliance on P2P communication can be less efficient in high-latency environments. Megatron-SP (Korthikanti et al., 2022) further optimizes activation usage in the attention based on

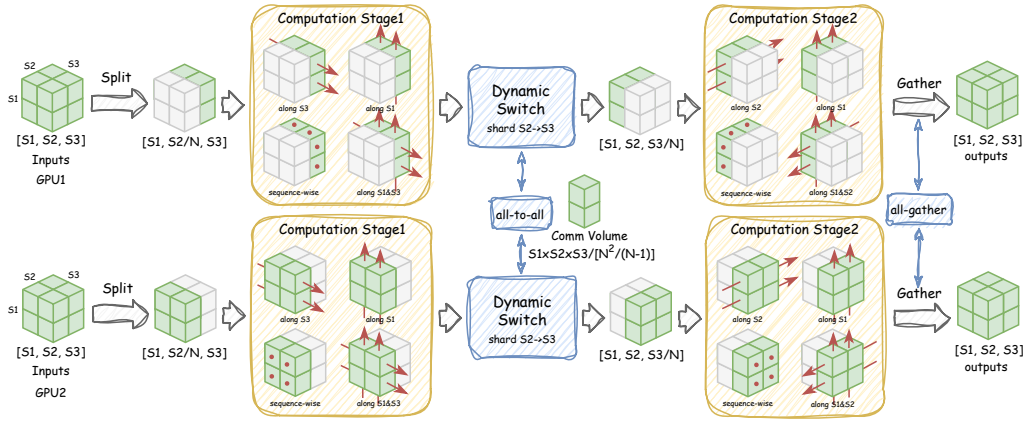

Figure 3: System overview of Dynamic Sequence Parallelism.

tensor parallelism. To transit between tensor parallelism and sequence parallelism in the Transformer block, additional all-gather and reduce-scatter operations are introduced. And it's constrained by the number of attention heads, as self-attention relies on the parallelism of the head dimension of the sequence. DeepSpeed-Ulysses (Jacobs et al., 2023) introduces an innovative approach for training long sequences by utilizing all-to-all collective communication. This method partitions the query, key, and value matrices across attention heads while preserving the original attention computation structure. The process is facilitated by two sets of all-to-all communications that alternate between sequence splitting and attention head splitting. Nevertheless, it is constrained by the number of attention heads as well.

Moreover, these sequence parallelism methods are designed for parallelism within a single sequence dimension. For multi-dimensional transformers, this strategy becomes inefficient due to unnecessary communication. While specialized parallelism for multi-dimensional sequences has been explored in specific domains (Cheng et al., 2024), their applicability remains limited.

3 Dynamic Sequence Parallelism

3.1 Overview

In the realm of multi-dimensional transformers, computation occurs independently for each sequence dimension. To harness this inherent feature, we introduce Dynamic Sequence Parallelism (DSP), an efficient, adaptive and ease-of-use sequence parallelism abstraction for multi-dimensional transformers.

To ensure correct computation logic with sequence parallelism, embedded methods typically require complex and time-consuming communications within computation modules to change the parallel dimension. However, as illustrated in Figure 3, the key feature of DSP is its dynamic switch of parallel dimension at the intersections of computation stages. By resharding only between computation stages dynamically, rather than within them, this approach allows DSP to remain independent of the computation logic within the module. Therefore, DSP eliminates numerous unnecessary communications within modules, and is able to utilize efficient all-to-all operations to switch parallelism dimensions for the intermediate sequence.

For operations involving all sequence dimensions, including the beginning and end of the model, DSP handles them by split and gather operation. Furthermore, we also propose a high-level, user-friendly implementation of DSP compatible with all distributed frameworks based on PyTorch, details in Appendix A.4.

3.2 Problem Definition

In sequence parallelism, the objective is to distribute activation computations across multiple GPUs to reduce the memory overhead caused by long sequences. This approach, however, incurs additional

Table 2: Definition of Dynamic Primitives for DSP. s_i denotes the sequence sharded from dimension i, while $\hat{s}$ indicates the sequence is not sharded. M represents the sequence size, and N signifies the sequence parallel size.

Source Shard	Target Shard	Primitives	Comm Operation	Comm Volume	Freq
s_i	s_i	/	/	/	/
s_i	s_j	Switch	all-to-all	M/N	High
$\hat{s}$	s_i	Split	/	0	Low
s_i	$\hat{s}$	Gather	all-gather	M	Low

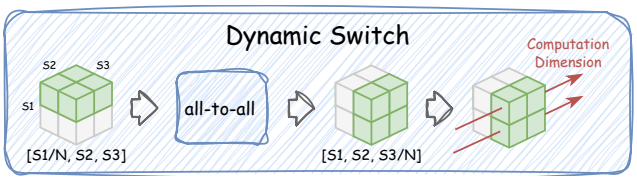

Figure 4: Illustration of dynamic switch.

communication costs between GPUs. Our goal is to optimize this trade-off in the context of multi-dimensional transformers.

Given a multi-dimensional sequence $\mathbf{X} \in \mathbb{R}^{[B,S_1,S_2,\ldots,S_K,C]}$ and a set of N GPUs, where $S_1, \ldots, S_K$ are the sequence along K different sequence dimensions, we aim to partition the computation such that the memory usage per GPU is under capacity while maintaining acceptable communication costs. Let $\mathbf{X}_{p,n}$ denote the partition of $\mathbf{X}$ assigned to GPU n, where p represents the partition strategy. The optimization problem is formulated as:

$$\min_{p} \sum_{n=1}^{N} \mathrm{CommCost}(\mathbf{X}_{p,n}) \;\; s.t. \;\; \mathrm{Memory}(\mathbf{X}_{p,n}) < Capacity. \tag{6}$$

Here, $\mathrm{Memory}(\mathbf{X}p, n)$ denotes the memory usage of partition $\mathbf{X}_{p,n}$ on GPU n, $\mathrm{CommCost}(\mathbf{X}_{p,n})$ represents the communication cost. We aim to achieve a balance that minimizes the overall computational overhead while optimizing GPU resource utilization.

3.3 Dynamic Primitives

In this section, we introduce the key dynamic primitives of DSP, as outlined in Table 2. These three primitives form the cornerstone for implementing DSP across a spectrum of multi-dimensional transformers.

The first condition is that when there is no need to alter sequence parallelism between computation stages, we maintain the shard status of the sequence. This approach significantly reduces unnecessary communication overhead. However, when it becomes necessary to transit parallelism between dimensions, we employ dynamic switch to efficiently transform parallelism. Specifically, as depicted in Figure 4, dynamic switching adjusts the parallelism to a dimension unrelated to the ongoing computation, utilizing highly efficient all-to-all operations.

Assume $\mathbf{X} \in \mathbb{R}^{[B,S_1,\ldots,S_i/N,\ldots,S_j,\ldots,S_K,C]}$ represents the input. The current parallel dimension is i so its sequence length is S_i/N on each device, where N is sequence parallel size. If we want to switch the shard dimension from i to j, the operation can be formulated as follows:

$$\mathbf{Y} = \mathrm{DynamicSwitch}(\mathbf{X}, i, j), \tag{7}$$

where the resulting tensor $\mathbf{Y}$ has the shape $\mathbb{R}^{[B,S_1,\ldots,S_i,\ldots,S_j/N,\ldots,S_K,C]}$. Furthermore, Split and Gather operations facilitate smooth transitions between sharded and non-sharded states. Although these operations may involve increased communication compared to Switch operations, they are

primarily utilized at the onset and conclusion of most networks, and also for some global operations in very rare conditions, rendering their costs negligible.

3.4 Adaptability and Flexibility

Given its decoupling from the computation of modules, DSP exhibits remarkable adaptability, making it compatible with a wide array of transformer variants such as Cross Attention (Hertz et al., 2022; Ma et al., 2024); specialized kernels like FlashAttention (Dao et al., 2022); special attention mechanisms including multi-query attention (Shazeer, 2019) and grouped-query attention (Ainslie et al., 2023); and even beyond like Mamba (Gu & Dao, 2023) and RWKV (Peng et al., 2023). This inherent flexibility enables DSP to seamlessly integrate into diverse transformers without specific modification. Furthermore, while DeepSpeed-Ulysses and Megatron-SP necessitate attention head splitting, DSP's scalability is significantly better because it shards on sequence length, which much larger especially when scaling sequences.

Moreover, DSP's adaptability extends beyond module compatibility to encompass various parallelism methodologies. From conventional data parallelism to more sophisticated approaches like ZeRO and pipeline parallelism, DSP effortlessly integrates with diverse parallel computing paradigms, thereby enhancing scalability and performance across distributed computing environments.

As shown in Appendix A.4, we demonstrate the API usage of DSP. By calling just four functions without knowing the detailed implementation, DSP can be enabled on PyTorch and is compatible with various distributed frameworks, including FSDP (Zhao et al., 2023), Accelerate (Gugger et al., 2022), DeepSpeed (Rasley et al., 2020), and Megatron-LM (Shoeybi et al., 2019).

4 Theoretical Analysis

We choose 2D-Transformer as described in Equation 5 as our base model, which is widely employed in real-world applications. To be specific, we use the OpenSora (Zangwei Zheng, 2024) variant of 2D-Transformer, an open-source video generation model, where there are two transformer blocks for two sequence dimensions separately. More details can be found in Appendix A.1. We choose DeepSpeed-Ulysses (Jacobs et al., 2023), Megatron-SP (Korthikanti et al., 2022) and RingAttention (Liu et al., 2023a) as baselines, which represent the state-of-the-art sequence parallelism methods employed for processing long sequences with transformers.

4.1 Communication Analysis

The primary advantage of DSP lies in its ability to minimize communication costs and enable scalable communication operations. DSP exploits the inherent characteristics of multi-dimensional transformers to eliminate unnecessary communication, compared with embedded approaches such as Megatron-LM (Shoeybi et al., 2019), Megatron-SP (Korthikanti et al., 2022), RingAttention (Liu et al., 2023a) and DeepSpeed-Ulysses (Jacobs et al., 2023). Consider an activation size of M and a sequence parallel size of N. In a 2D-Transformer, there is one transformer block for each sequence dimension per layer, resulting in two transformer blocks per layer. More details of the implementation of each method are demonstrated in Appendix A.2.

Both Megatron-SP and DeepSpeed-Ulysses require transforming sequence parallelism from a sequence-shard to a head-shard layout. Megatron-SP employs 2 all-gather operations to aggregate the entire sequence and 2 reduce-scatter operations to distribute results in the attention and MLP layers for one transformer block. This results in a total of 8 collective communication operations, leading to a total communication volume of $8M$. Conversely, DeepSpeed-Ulysses incurs 4 communication operations in temporal block for the query, key, value, and output layout transformations. Consequently, the communication volume transmitted per device for an all-to-all communication of size M across N GPUs is $4M/N$. Ring-Attention needs to communicate the entire key and value in the temporal as well, resulting in a total communication volume of $2M$.

In contrast, DSP mitigates communication cost by employing only two all-to-all operations in total two blocks per layer. As shown in Table 4.1, it reduces the communication volume to $2M/N$, significantly outperforming other sequence parallelism techniques. Notably, with merely two all-to-all operations, DSP exhibits efficient scalability even in super-large clusters for training and inference

Table 3: Comparison of DSP with other sequence parallelism methods for 2D-Transformer architectures. M denotes the activation size, and N represents the number of devices. Communication volume refers to the per-layer volume.

Method	Communication Volume	Activation Memory	Parameter Memory	Ease of Use
Ring Attention	$2M$ 👎	👍	👍	👎👎
Megatron-LM	$8M$ 👎👎	👎	👍	👎
Megatron-SP	$8M$ 👎👎	👎	👍	👎
DeepSpeed-Ulysses	$4M/N$ 👍	👍	👍	👍
DSP (ours)	$2M/N$ 👍👍	👍👍	👍	👍👍

on extremely long sequences because the communication volume decreases as the number of nodes increases, rendering DSP an exceptional choice for large-scale distributed training and inference tasks involving extreme long sequences.

4.2 Memory Analysis

Regarding activation memory, since we shard every tensor in the transformer, we are theoretically able to achieve the minimum activation cost, similar to DeepSpeed-Ulysses and Ring Attention. In practice, however, our approach requires less shape transformation and communication overhead, allowing us to further reduce intermediate activation memory compared to other methods. Megatron-SP, on the other hand, needs to hold the entire activation after the all-gather operation, resulting in higher memory requirements for hosting the activation.

As for parameter memory, as discussed in Section 3.4, DSP is compatible with most parameter parallelism strategies. In this analysis, we utilize the ZeRO technique (Rajbhandari et al., 2019) to evenly shard all parameters across different devices. Consequently, our parameter memory footprint can be kept low.

5 Experiments

Experiments are conducted on 128 NVIDIA H100 GPUs, interconnected via NVLink within nodes and InfiniBand across nodes. Our methods and implementations are not dependent on specific hardware architectures and can generalize to other devices, particularly those with less efficient interconnects. We follow the same baseline and settings as discussed in Section 4, utilizing 720M and 3B size Transformer-2D models in our experiments. Despite the existence of various 2D-Transformer variants, their architectures are fundamentally similar. Consequently, we select one base model similar to OpenSora (Zangwei Zheng, 2024), an open-source video generation model, for our study. The code is implemented using PyTorch (Paszke et al., 2019).

In the following evaluations, we focus on addressing the following questions: 1) How is DSP's end-to-end performance compared with other SOTA sequence parallelism? 2) How is DSP's scaling ability when scale to many GPUs? 3) What is DSP's memory consumption like in practice?

5.1 End-to-End Performance

In this section, we compare the end-to-end performance of different sequence parallelism methods on 128 NVIDIA H100 GPUs. We use a combination of sequence parallelism and data parallelism, with the sequence parallelism set to the minimum size for each method. We evaluate across different sequence lengths ranging from 0.5 million to 4 million tokens, which are common usages for video generation. Details can be found in Appendix A.3.2. As shown in Figure 5, DSP is able to outperform DeepSpeed-Ulysses by 32% to 75%, and other methods by up to 10x due to its communication efficiency. As the sequence length becomes longer and the sequence parallel size increases, as DSP's communication volume decreases as the device number increases, our performance's advantage over the baselines becomes even more pronounced. When scaling from 0.5M to 4M tokens, our FLOPS drops by at most 23%, while other methods experience at least a 40% drop.

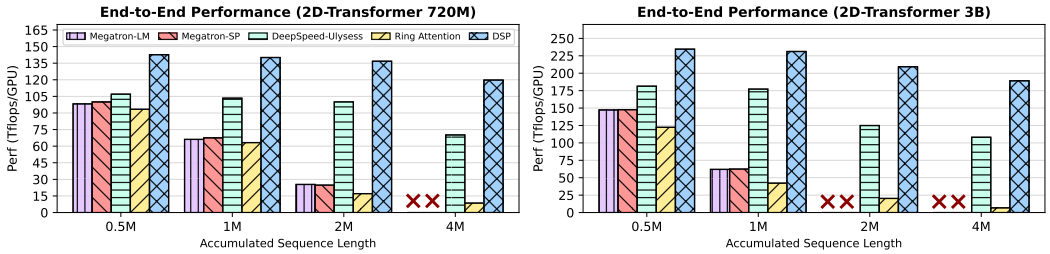

Figure 5: End-to-end performance comparison of different sequence parallel methods combined with data parallel on 128 H100 GPUs. The sequence parallel size is set to minimum for each method.

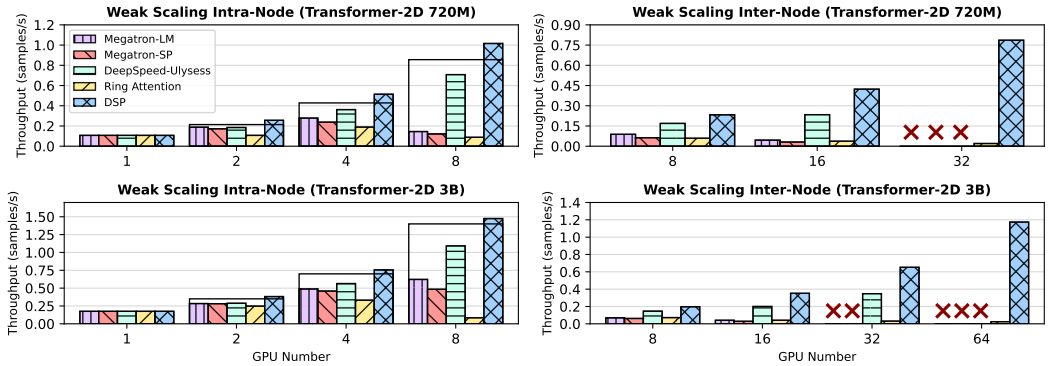

Figure 6: Weak scaling ability evaluation of different methods with sequence parallelism only. "×" denotes out of memory or head. Black boxes represent linear scaling.

5.2 Scaling Ability

This section evaluates the scaling ability of DSP from two perspectives: weak scaling and strong scaling. Weak scaling refers to scenarios where the computational workload per device remains constant while incrementally increasing the number of devices. This setup is analogous to the training stage, where the goal is to scale longer sequences over more GPUs. Strong scaling, on the other hand, is more challenging as it requires keeping the total computational workload constant while incrementally increasing the number of devices. In this case, the computation becomes more sparse on each device. Strong scaling is often employed when the objective is to infer an input sequence rapidly across many GPUs for low-latency applications. The experiments are divided into intra-node and inter-node evaluations due to the different interconnection conditions. Intra-node experiments leverage NVLink interconnect for communication, while inter-node experiments utilize a combination of NVLink and InfiniBand interconnect. More details can be found in Appendix A.3.3.

Weak Scaling. In the weak scaling experiments, to maintain a consistent computational workload for each GPU, the batch size is linearly increased proportional to the number of GPUs, while the sequence length is fixed. As shown in Figures 6, DSP significantly outperforms other methods by more than 80.7%. Moreover, DSP can scale up to 64 GPUs without being limited by the number of attention heads, unlike DeepSpeed-Ulysses and Megatron-SP. Despite scaling to 64 GPUs, DSP maintains an almost linear throughput increase, with only a 15% performance loss from 8 GPUs to 64 GPUs. Additionally, DSP can achieve super-linear scaling for intra-node due to efficient communication.

Strong Scaling. In the strong scaling experiments, both batch size and sequence length are fixed. As shown in Figure 7, DSP can maintain linear scalability when scaling up to 8 GPUs for 720M model and 4 GPUs for 3B model, which covers most practical scenarios. To evaluate the extreme performance capabilities of DSP, we further scale up to 64 GPUs with very little workload per device. Although there is an inevitable performance drop, DSP's throughput remains significantly better than the baselines. As shown in Figure 8, our work can significantly reduce inference latency compared with baselines with the same workload.

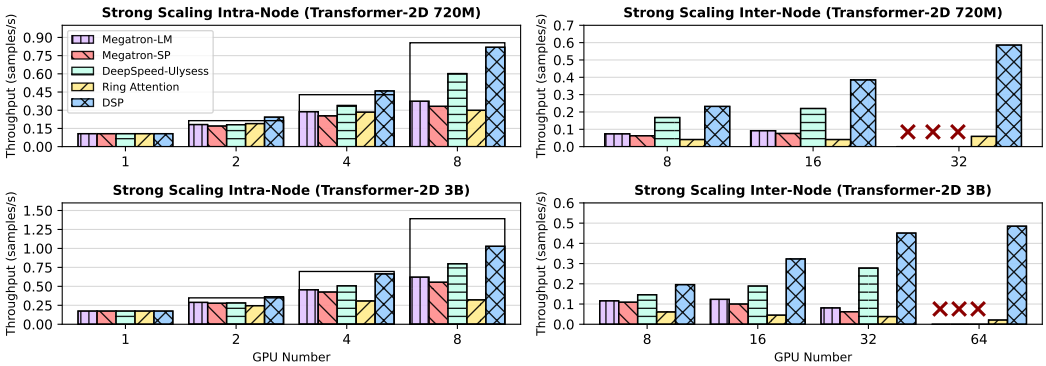

Figure 7: Strong scaling ability evaluation of different methods with sequence parallelism only. "×" denotes out of memory or head. Black boxes represent linear scaling.

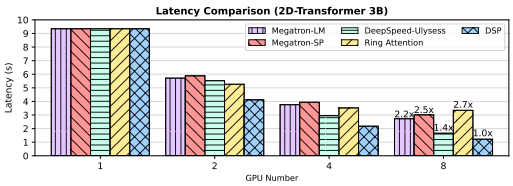

Figure 8: Inference latency comparison of different sequence parallelism methods.

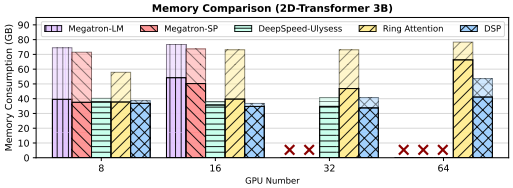

Figure 9: Memory comparison of different sequence parallelism methods.

5.3 Memory Consumption

Figure 9 demonstrates the memory consumption comparison of different baselines in the weak scaling setting. The semi-transparent bar represents the cache memory, while the solid bar represents the allocated memory. The total memory usage is the sum of them. Our approach exhibits the lowest memory usage, scaling efficiently for longer sequences. Furthermore, DSP's memory usage is compact without excessive cache memory bloat, unlike Ring-Attention, Megatron-LM and Megatron-SP.

6 Conclusion and Discussion

In this work, we introduce Dynamic Sequence Parallelism (DSP), a novel sequence parallel abstraction for effectively scaling multi-dimensional transformers to long sequences. Unlike current embedded sequence parallel methods that only shard on single sequence dimension and are tailored to specific patterns, DSP offers a general and elegant solution by dynamically switching the parallel dimension during computation, decoupled from the computation module.

The key advantages of DSP are: 1) substantially reduced communication costs, 2) adaptability across modules without specialized modifications, and 3) remarkable ease of implementation enabled by a simple high-level API. Our experiments demonstrated DSP's superiority, achieving from 32.2% to 10x higher end-to-end throughput and at least 75% lower communication volume compared to state-of-the-art methods. Its elegance and ease of use make it a promising solution for efficient sequence parallelism across a wide range of applications.

Limitations. One limitation of this work is that DSP is specifically designed for multi-dimensional transformers and may not adapt well to single-dimensional ones like language models. Additionally, while there are global operations that involve all sequence dimensions, which are rare in transformer, DSP may not be of optimal efficiency.

Future works. In the future, DSP could expand its scope beyond transformer architectures to architectures including convolution, recurrent, and graph neural networks to utilize its potential across various tasks. Furthermore, automated optimization techniques could enable DSP to dynamically and autonomously determine the most effective switching strategy based on network analysis, thereby optimizing overall system efficiency and efficacy.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebr'on, and Sumit K. Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *ArXiv*, abs/2305.13245, 2023.

A. Blattmann, Tim Dockhorn, Sumith Kulal, Daniel Mendelevitch, Maciej Kilian, and Dominik Lorenz. Stable video diffusion: Scaling latent video diffusion models to large datasets. *ArXiv*, abs/2311.15127, 2023.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

Shengchao Chen, Ting Shu, Huan Zhao, Guo Zhong, and Xunlai Chen. Tempee: Temporal-spatial parallel transformer for radar echo extrapolation beyond auto-regression. *IEEE Transactions on Geoscience and Remote Sensing*, 2023.

Shenggan Cheng, Xuanlei Zhao, Guangyang Lu, Jiarui Fang, Tian Zheng, Ruidong Wu, Xiwen Zhang, Jian Peng, and Yang You. Fastfold: Optimizing alphafold training and inference on gpu clusters. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 417–430, 2024.

Yuren Cong, Wentong Liao, Hanno Ackermann, Michael Ying Yang, and Bodo Rosenhahn. Spatial-temporal transformer for dynamic scene graph generation. *2021 IEEE/CVF International Conference on Computer Vision*, pp. 16352–16362, 2021.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher R'e. Flashattention: Fast and memory-efficient exact attention with io-awareness. *ArXiv*, abs/2205.14135, 2022.

Azad Deihim, Eduardo Alonso, and Dimitra Apostolopoulou. Sttre: A spatio-temporal transformer with relative embeddings for multivariate time series forecasting. *Neural Networks*, 168:549–559, 2023.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Zhicheng Geng, Luming Liang, Tianyu Ding, and Ilya Zharkov. Rstt: Real-time spatial temporal transformer for space-time video super-resolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 17441–17451, 2022.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *ArXiv*, abs/2312.00752, 2023.

Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, Sourab Mangrulkar, Marc Sun, and Benjamin Bossan. Accelerate: Training and inference at scale made simple, efficient and adaptable., 2022.

Lu He, Qianyu Zhou, Xiangtai Li, Li Niu, Guangliang Cheng, Xiao Li, Wenxuan Liu, Yunhai Tong, Lizhuang Ma, and Liqing Zhang. End-to-end video object detection with spatial-temporal transformers. In *Proceedings of the 29th ACM International Conference on Multimedia*, pp. 1507–1516, 2021.

Amir Hertz, Ron Mokady, Jay M. Tenenbaum, Kfir Aberman, Yael Pritch, and Daniel Cohen-Or. Prompt-to-prompt image editing with cross attention control. *ArXiv*, abs/2208.01626, 2022.

W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12): 1170–1183, 1986.

Jonathan Ho, Nal Kalchbrenner, Dirk Weissenborn, and Tim Salimans. Axial attention in multidimensional transformers. *ArXiv*, abs/1912.12180, 2019.

Lei Huang, Feng Mao, Kai Zhang, and Zhiheng Li. Spatial-temporal convolutional transformer network for multivariate time series forecasting. *Sensors*, 22(3):841, 2022.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in Neural Information Processing Systems*, 32, 2019.

Sam Adé Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *ArXiv*, abs/2309.14509, 2023.

John M. Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Zídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andy Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David A. Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. *Nature*, 596:583 – 589, 2021.

Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence C. McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *ArXiv*, abs/2205.05198, 2022.

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

Shenggui Li, Fuzhao Xue, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In *Annual Meeting of the Association for Computational Linguistics*, 2021.

Shigang Li and Torsten Hoefler. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines. *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.

Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023a.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *Advances in Neural Information Processing Systems*, 36, 2024.

Ziming Liu, Shenggan Cheng, Hao Zhou, and Yang You. Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency. *The International Conference for High Performance Computing, Networking, Storage, and Analysis*, pp. 1–13, 2023b.

Xin Ma, Yaohui Wang, Gengyun Jia, Xinyuan Chen, Ziwei Liu, Yuan-Fang Li, Cunjian Chen, and Yu Qiao. Latte: Latent diffusion transformer for video generation. *ArXiv*, abs/2401.03048, 2024.

Milot Mirdita, Konstantin Schütze, Yoshitaka Moriwaki, Lim Heo, Sergey Ovchinnikov, and Martin Steinegger. Colabfold: making protein folding accessible to all. *Nature Methods*, 19(6):679–682, 2022.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15, 2019.

Xiaoxin Pan, Long Wang, Zhongju Wang, and Chao Huang. Short-term wind speed forecasting based on spatial-temporal graph transformer networks. *Energy*, 253:124095, 2022.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *ArXiv*, abs/1912.01703, 2019.

William Peebles and Saining Xie. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 4195–4205, 2023.

Bo Peng, Eric Alcaide, Quentin G. Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, G Kranthikiran, Xuming He, Haowen Hou, Przemyslaw Kazienko, Jan Kocoń, Jiaming Kong, Bartlomiej Koptyra, Hayden Lau, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Xiangru Tang, Bolun Wang, Johan Sokrates Wind, Stansilaw Wozniak, Ruichong Zhang, Zhenyuan Zhang, Qihang Zhao, Peng Zhou, Jian Zhu, and Rui Zhu. Rwkv: Reinventing rnns for the transformer era. In *Conference on Empirical Methods in Natural Language Processing*, 2023.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training a trillion parameter models. *ArXiv*, abs/1910.02054, 2019.

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International Conference on Machine Learning*, pp. 8821–8831, 2021.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020.

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, Ioannis Antonoglou, Rohan Anil, Sebastian Borgeaud, Andrew Dai, Katie Millican, Ethan Dyer, Mia Glaese, Thibault Sottiaux, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, James Molloy, Jilin Chen, Michael Isard, Paul Barham, Tom Hennigan, and et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10684–10695, 2022.

Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in Neural Information Processing Systems*, 31, 2018.

Noam M. Shazeer. Fast transformer decoding: One write-head is all you need. *ArXiv*, abs/1911.02150, 2019.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *ArXiv*, abs/1909.08053, 2019.

Uriel Singer, Adam Polyak, Thomas Hayes, Xiaoyue Yin, Jie An, Songyang Zhang, Qiyuan Hu, Harry Yang, Oron Ashual, Oran Gafni, Devi Parikh, Sonal Gupta, and Yaniv Taigman. Make-a-video: Text-to-video generation without text-video data. *ArXiv*, abs/2209.14792, 2022.

Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

Mingxing Xu, Wenrui Dai, Chunmiao Liu, Xing Gao, Weiyao Lin, Guo-Jun Qi, and Hongkai Xiong. Spatial-temporal transformer networks for traffic flow forecasting. *arXiv preprint arXiv:2001.02908*, 2020.

Bin Yan, Houwen Peng, Jianlong Fu, Dong Wang, and Huchuan Lu. Learning spatio-temporal transformer for visual tracking. *2021 IEEE/CVF International Conference on Computer Vision*, pp. 10428–10437, 2021.

Antoine Yang, Antoine Miech, Josef Sivic, Ivan Laptev, and Cordelia Schmid. Tubedetr: Spatio-temporal video grounding with transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 16442–16453, 2022.

Xiangyu Peng Zangwei Zheng. Open-sora: Democratizing efficient video production for all, April 2024.

Yanli Zhao, Andrew Gu, Rohan Varma, Liangchen Luo, Chien chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16:3848–3860, 2023.

Ce Zheng, Sijie Zhu, Matias Mendieta, Taojiannan Yang, Chen Chen, and Zhengming Ding. 3d human pose estimation with spatial and temporal transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 11656–11665, 2021.

A APPENDIX

A.1 MODEL DETAILS

In the theoretical analyses and evaluation section, we use a Transformer-2D model as our base model, similar to OpenSora (Zangwei Zheng, 2024). However, it is not exactly OpenSora; we have removed its specific cross-attention module to ensure that the performance can be generalized to other models. Therefore, in each layer, there are only two transformer blocks that process two sequence dimensions separately, as shown in Figure 10. Specifically, the two dimensions are temporal t and spatial s for a sequence. Each dimension is processed by a corresponding transformer block, which is a common strategy in many applications.

A.2 PARALLELISM IMPLEMENTATION

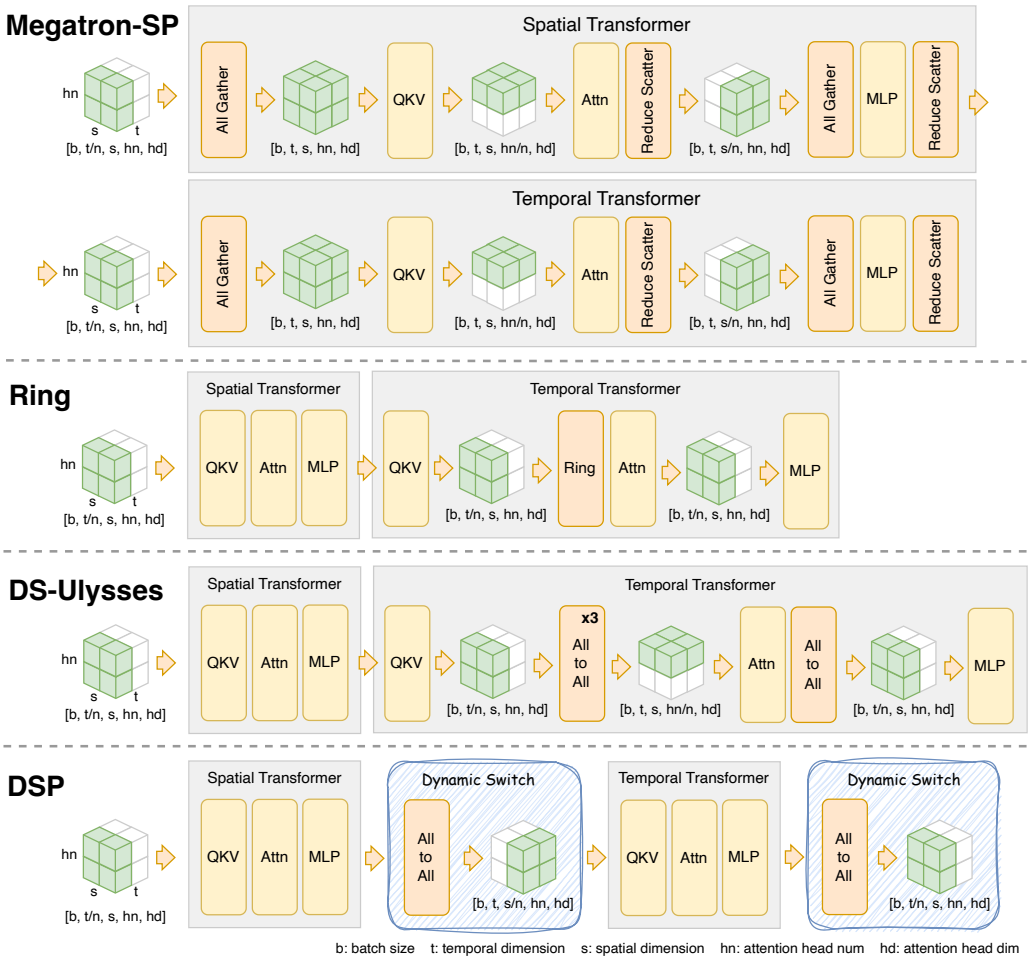

Figure 10: Overview of different sequence parallelism methods for 2D-Transformer.

In Figure 10, we demonstrate the detailed implementation of different sequence parallel methods on 2D-Transformer. The implementation of DeepSpeed-Ulysses (Rasley et al., 2020) is directly adopted from the official OpenSora (Zangwei Zheng, 2024) implementation, while Megatron-SP (Korthikanti et al., 2022) is adopted based on its official implementation. For Ring-Attention (Liu et al., 2023a), we adopt an unofficial implementation for the 2D-Transformer.

Megatron-SP employs four resource-intensive collective communication operations per transformer block. Specifically, it initiates an AllGather operation to aggregate the entire input x, succeeded by

ReduceScatter operations at the output for both attention and MLP modules, culminating in a total communication volume of $8M$.

DeepSpeed-Ulysses adopts the more efficient AlltoAll approach. It leverages all-to-all for query, key, value to transform their shard dimension before attention, and a all-to-all for output after attention. And it only need to communicate in temporal transformer block. Consequently, the communication volume transmitted per device for an AlltoAll communication of size M across N GPUs is $4M/N$.

Ring-Attention is not shown in the figure because it does not require resharding. We implement sequence communication in the temporal transformer as the time axis is split. In the attention module, it needs to pass the key and value to all other devices, resulting in a total communication volume of $2M$.

DSP applies dynamic switching between stages to switch the parallel dimension, which involves two AlltoAll operations, totaling $2M/N$ communication.

A.3 Experiment Settings

A.3.1 Model Size

In the experiments, we use 720M and 3B size for 2D-Transformer. There specific model settings are shown in Table 4.

Table 4: Model settings of 720M and 3B 2D-Transformer.

Model Name	Layers	Hidden States	Attention Heads	Patch Size
720M	28	1152	16	(1, 2, 2)
3B	36	2038	32	(1, 2, 2)

A.3.2 End-to-end Performance

Here is the polished text in a more formal format without using bullet points: In end-to-end performance experiments, 128 GPUs were utilized for all methods. For each method, the minimum sequence parallel size that would not result in out-of-memory errors was employed to reduce communication overhead, with data parallelism employed for the remaining size. ZeRO-2 was used for all methods except Megatron-SP. The specific parallel size is detailed in Table 5.

The accumulated sequence length ranged from 0.5M to 4M, which appears significantly larger than typical text lengths. However, such lengths are common for multi-dimensional tasks. In this case, we followed the workload of video generation. The spatial sequence, representing video resolution, was fixed at 1024x1024. After applying the Variational Autoencoder (VAE) and Patch Embedding, the final length for the spatial sequence was 4096. The temporal sequence, representing video length, scales linearly in the test.

Table 5: End-to-end performance parallel settings. Tuple for methods denotes (sequence_parallel_size, data_parallel_size).

Model Size	Sequence Length	Temporal Sequence	Spatial Sequence	DeepSpeed Ulysses	Megatron SP	Ring Attention	DSP
720M	0.5M	128	4096	(2, 64)	(2,64)	(2, 46)	(2, 64)
	1M	256	4096	(4, 32)	(4,32)	(4, 32)	(4, 32)
	2M	512	4096	(8, 16)	(16,8)	(8, 16)	(8, 16)
	4M	1024	4096	(16, 8)	/	(16, 8)	(16, 8)
3B	0.5M	128	4096	(4, 32)	(4, 32)	(4, 32)	(4, 32)
	1M	256	4096	(8, 16)	(16,8)	(8, 16)	(8, 16)
	2M	512	4096	(16, 8)	/	(16, 8)	(16, 8)
	4M	1024	4096	(32, 4)	/	(32, 4)	(32, 4)

A.3.3 SCALING ABILITY

Table 6: Strong scaling experiment settings.

Model Size	Type	Batch Size	Temporal	Spatial
720M	Intra-Node	1	64	4096
	Inter-Node	1	256	4096
3B	Intra-Node	1	16	4096
	Inter-Node	1	128	4096

Table 7: Weak scaling experiment settings.

Model Size	Type	GPU Number	Batch Size	Temporal	Spatial
720M	Intra-Node	1	1	64	4096
		2	2	64	4096
		4	4	64	4096
		8	8	64	4096
	Inter-Node	8	1	256	4096
		16	2	256	4096
		32	4	256	4096
3B	Intra-Node	1	1	16	4096
		2	2	16	4096
		4	4	16	4096
		8	8	16	4096
	Inter-Node	8	1	128	4096
		16	2	128	4096
		32	4	128	4096
		64	8	128	4096

In weak scaling experiments, as shown in Figure 7 we fix the sequence length and linearly increase the batch size, ensuring that the workload on each device remains constant as the number of devices scales. In strong scaling experiments, as shown in Figure 6, we fix both the sequence length and batch size, keeping the total computation constant. For each experiment, we set the sequence length to the maximum for the least GPU case to fully utilize the computational resources. Specifically, we use the same spatial sequence length and adjust the temporal sequence length to its maximum for each test and sequence parallel size is set to GPU number.

A.4 API

In the API, as illustrated in Figure 11, we provide four functions for users: $dsp_dataloader$, $split$, $gather$, and $dynamic_switch$. The $dynamic_switch$ function will reshard sequences between the computation stages. Users only need to input the source shard dimension and target shard dimension. The $split$ and $gather$ functions assist users in distributing and retrieving the entire sequence when needed. The $dsp_dataloader$ function helps users process their dataloader for sequence parallelism, which requires the same data within a single sequence parallel group.

```
from dsp import dsp_dataloader, split, gather, dynamic_switch

# Preprocess dataloader for sequence parallel
dataloader = dsp_dataloader(dataloader)
...

# Forward pass
x = split(x, tgt_shard=1)
x = f1(x)
x = f2(x)
# Dynamic switch between computation stages
x = dynamic_switch(x, cur_shard=1, tgt_shard=2)
x = f3(x)
# Dynamic switch between computation stages
x = dynamic_switch(x, cur_shard=2, tgt_shard=0)
...
x = gather(x, cur_shard=0)
```

Figure 11: An example to demonstrate DSP's API for Pytorch. Developers can use only four functions to enable DSP. $shard$ refers to the sequence parallel dimension.